Controlador de Dispositius en Xarxa

Treball Fi de Carrera

César Hernández Bañó
Consultor: Agustin Fernández Jimenez

Enginyeria Tècnica en Informàtica de Sistemes

UOC, Gener 2002

Index

Objectius

L'objectiu d'aquest projecte de fi de carrera és facilitar l'ús de dispositius presents en una xarxa d'ordinadors.

Aquest projecte[1] crea una capa de software que oculta el nivell de xarxa per a un ordinador, podent usar qualsevol dispositiu en xarxa com si es trobés físicament connectat a l'ordinador en qüestió.

Això ens permetrà, per exemple, reproduir un arxiu de música en una targeta de so instal·lada a un altre ordinador, mentre que el programa reproductor s'executa en el nostre; també, poder veure des del nostre monitor el que conté la pantalla d'altre ordinador, etc. O sigui, un dispositiu es podrà usar des de qualsevol ordinador present en una xarxa.

El fet de crear una capa de software facilita l'ús de qualsevol dispositiu; hem de tenir en compte que hi ha diferents implementacions per l'ús de dispositius en una xarxa: per usar impressores, LPD, per usar sistemes de fitxers, NFS... El meu projecte fa que sigui innecessari usar aquestes implementacions, ja que, qualsevol dispositiu que voldrem usar (present en una xarxa) serà accessible a qualsevol ordinador tal com si aquest estigués present al mateix.

El projecte s'ha creat per executar-se en sistemes operatius Linux, per les versions 2.4.x del nucli, en processadors Intel o compatibles. Per poder usar-lo no fa falta tenir una xarxa local, ni tan sols connexió per mòdem, ja que es pot executar en un sol ordinador. S'utilitza la llibreria de connexió TCP/IP de Linux (*socket*), amb lo qual, l'ús de dispositius tant pot ser a través d'una xarxa local, mitjançant Internet, etc.

El projecte consta de 3 parts:

1. El mòdul del sistema operatiu. Aquest és una peça de software que s'executa com un component del sistema operatiu (*kernel*). S'ha hagut de revisar amb molta cura ja que qualsevol errada pot bloquejar tot el sistema.

2. El client. Aquest programari s'encarrega de fer la connexió entre el sistema operatiu i el servidor

3. El servidor; s'encarrega d'atendre les peticions del client i executar-les als dispositius.

1 A partir d'ara, l'anomenarem CDX (Controlador de Dispositius en Xarxa)

1. Descripció dels dispositius en Linux

A continuació, es farà una descripció del funcionament dels dispositius en Linux, com es defineixen, quin tractament els hi dóna el sistema operatiu, etc.

S'entén per dispositiu qualsevol objecte de maquinari o programari, gestionat pel sistema operatiu, i sobre el qual podem fer operacions de lectura/escriptura per tal d'obtenir, transferir o emmagatzemar informació. Podem trobar exemples de dispositius tals com un teclat, un ratolí, una disquetera (de maquinari) o també un comptador d'instruccions de la CPU o un terminal virtual (de programari).

En Linux, tots els dispositius són tractats com a fitxers especials. Aquests són especials en el sentit de que no es tracta de fitxers normals, sinó que són identificats com a dispositius. Normalment, totes les entrades de dispositius estan localitzades al directori */dev* de Linux.

1.1 Atributs dels dispositius

Tots els dispositius tenen una sèrie d'atributs en Linux:

· Tipus: Es diferencien entre dispositius de bloc i dispositius de caràcter. Un dispositiu de bloc és un tipus de dispositiu en el qual totes les operacions de transferència es fan en un bloc de dades, i mitjançant un buffer; típicament, són dispositius d'emmagatzemament de dades, com un disc dur. Els dispositius de caràcter són tots els altres, o sigui, s'accedeixen byte a byte i sense buffer.

· Número major: Aquest número ens indica de quin dispositiu es tracta, si és una targeta de so, un mòdem, un ratolí... Mitjançant el número major (per abreujar, *major*) i el tipus, el sistema operatiu sap quin és el controlador que gestiona el dispositiu. Aquest número té un rang, que és determinat per un byte (de 0 a 255).

· Número menor: Aquest número (en anglès, *minor)* indica al sistema operatiu quin dispositiu estem usant, quan hi ha 2 o més del mateix tipus; per exemple: si tenim dues targetes de so, les dues tindran el mateix numero *major*, però diferent numero *minor.* També s'emmagatzema mitjançant un byte.

· Nom: Per suposat, i com qualsevol altre fitxer, ha de tenir un nom, que sigui identificatiu del dispositiu que gestiona. Normalment s'utilitza una abreviatura del nom real del dispositiu seguit del número *minor.* Hem de tenir en compte,

però, que aquest nom només és significatiu per l'usuari, no pel sistema operatiu. El S.O. utilitza un altre mecanisme per identificar-lo, el qual veurem seguidament.

1.2 Tipus de Controladors

Tots els dispositius en Linux són gestionats per un programari, anomenat controlador (*device* en anglès). Es fa la distinció entre dos tipus de controladors:

1. Controlador resident dins el mateix nucli del sistema operatiu. Aquest està integrat dins el *kernel*, i està sempre present a la memòria.

2. Controlador no resident al sistema (mòdul). Aquest controlador es presenta com un fitxer executable no integrat al *kernel*, que es carrega per tal de gestionar un dispositiu. Tot i que és un fitxer executable, s'executa a nivell de sistema operatiu, i no com a nivell d'usuari[1]. El CDX és un controlador d'aquest tipus.

Ambdós tipus de controladors són equivalents; això vol dir que, en principi, es pot crear un controlador d'un o altre tipus. No obstant, el segon tipus presenta una avantatge, i és el fet que, per tal de crear-lo, no hem de modificar cap peça de codi del nucli del sistema operatiu. Es pot escriure com un programa normal i, amb unes opcions adequades de compilació (que veurem més endavant), convertir-lo en un mòdul.

Aquests mòduls es poden carregar a la memòria del sistema operatiu de dues maneres:

1. Manualment: Mitjançant una ordre del sistema operatiu, carreguem el mòdul indicat a la memòria.

2. Automàticament: Fent us d'un fitxer de configuració del sistema operatiu i del carregador de mòduls, el sistema operatiu sap quin mòdul ha de carregar per a un dispositiu específic.[2]

En el moment que volem utilitzar un dispositiu, primer hem d'obrir-lo. Per tal de fer això, hem d'executar l'ordre del sistema operatiu *open* indicant-li el nom i la ruta del mateix. És aquí on el sistema operatiu detecta que volem obrir un dispositiu, i ha d'identificar-lo. Per tal d'això, es serveix del tipus (bloc o caràcter) i del número *major*. Una vegada ha llegit això, primer busca dins el *kernel* per veure

1 Recordem que als sistemes operatius hi ha, com a mínim, dos nivells d'execució de processos: el de sistema operatiu (o privilegiat) que pot realitzar qualsevol operació de codi màquina, i el de usuari (o restringit) que no pot realitzar instruccions reservades

2 Veure el manual de Linux (*man*) de "modules.conf"

si té un controlador resident que gestioni el dispositiu. Si no el troba, llavors mira si hi ha algun mòdul carregat a la memòria que el gestioni. I si tampoc el troba, llavors intenta carregar automàticament del sistema d'arxius a la memòria el mòdul adequat. Si s'ha trobat algun controlador que el gestioni, ja el podem utilitzar.

1.3 Operacions sobre els dispositius

Tots els dispositius tenen una sèrie d'operacions comunes que es poden realitzar sobre ells: obrir dispositiu, tancar, llegir, escriure, posicionar punter... S'ha de distingir, però, entre les ordres que executa un programa (que crida a una llibreria de funcions de sistema operatiu) i les ordres que rep directament el controlador (ordres del sistema operatiu); p.ex: a nivell de llibreria, per obrir fitxers, tenim les ordres *open i fopen*[1], mentre que el controlador només té la funció *open* (i amb diferents paràmetres i valor de retorn que la de llibreria)[2]

Aquí descriurem les ordres a nivell de controlador, i per al tipus de dispositius de caràcter, que és el que utilitza el CDX. Abans, però, fem una aclaració dels tipus de dades utilitzats:

· *loff_t, long long*: Representen enters de 64 bits.

· *int, long, long int, ssize_t, size_t*: Representen enters de 32 bits.

· *file*: Aquesta és una estructura que representa un descriptor de fitxer; és l'encarregada de tenir la informació d'un fitxer que s'ha obert (i per tant també un dispositiu), tal com: mode d'obertura, posició del punter de lectura/escriptura, etc[3].

· *inod*e: Aquesta és una estructura que representa el fitxer que s'ha obert, contenint la informació que l'identifica: data de modificació, propietari, etc[4]. Hem de distingir entre aquesta estructura i l'anterior, *file ;* mentre aquesta última fa referència a l'apertura del fitxer, *inode* fa referència al fitxer mateix, a la informació emmagatzemada al sistema d'arxius.

Veiem a continuació les ordres dels dispositius de caràcter, en el format de declaració de funcions en el llenguatge C:

loff_t (*llseek) (struct file *, loff_t, int);

Aquesta funció serveix per posicionar el punter de lectura/escriptura sobre

1 Consultar el manual de les ordres *open* i *fopen*
2 Consultar l'estructura *file_operations* a */usr/include/linux/fs.h*
3 Veure estructura *file* a "3.7 El descriptor de fitxer"
4 Veure estructura *inode* a */usr/include/linux/fs.h*

el dispositiu. El segon paràmetre és el desplaçament a aplicar, i el tercer indica des d'on s'aplica el desplaçament (0=Desplaçament absolut, 1=Desplaçament relatiu a la posició inicial, 2=Desplaçament relatiu al final del fitxer)

ssize_t (*read) (struct file *, char *, size_t, loff_t *);
Aquesta ordre serveix per llegir dades d'un dispositiu. El segon paràmetre indica la direcció on estan situades les dades; el tercer, el tamany de les dades. I el quart, el desplaçament sobre el descriptor de fitxer.

ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
Anàlogament a la funció anterior, aquesta serveix per escriure dades a un dispositiu.

unsigned int (*poll) (struct file *, struct poll_table_struct *);
Aquesta funció ens diu si un dispositiu pot ser llegit o escriure dins d'ell.

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
Aquesta funció serveix per cridar ordres específiques d'un dispositiu, tals com escollir la freqüència de mostreig en una targeta de so, canviar els bits/s en un port sèrie, etc. El tercer paràmetre indica la ordre a realitzar, i el quart està relacionat amb les dades mateixes (que veurem més endavant).

int (*mmap) (struct file *, struct vm_area_struct *);
Aquesta funció serveix per tal de que el dispositiu pugui usar una zona de memòria del procés que la invoca.

int (*open) (struct inode *, struct file *);
Aquesta funció s'encarrega d'obrir el dispositiu.

int (*release) (struct inode *, struct file *);
Aquesta funció es crida quan s'allibera el descriptor de fitxer (file) que utilitza el dispositiu. Intuïtivament, podríem dir que es crida al tancar el dispositiu, però pot ser que el descriptor de fitxer estigui sent usat per un altre procés[1].

int (*fsync) (struct file *, struct dentry *, int datasync);
Aquesta funció s'utilitza per enviar al dispositiu dades que encara romanen a la memòria i per tant, que no han estat traspassades encara.

int (*fasync) (int, struct file *, int);
Aquesta funció s'utilitza en les notificacions asíncrones de dades.

1 Veure capítols “Descripció del CDX”, “Tècniques usades en el CDX”

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
Aquestes dues funcions s'encarreguen de fer múltiples lectures/escriptures a diferents zones de memòria.

Totes aquestes ordres les rep el controlador; per tal de cridar-les, des d'un procés, s'utilitzen funcions anàlogues presents a la llibreria del S.O., tal i com s'ha comentat abans.

Normalment, quan hi ha un error, retornen un valor negatiu amb el codi d'error, i si no n'hi ha, retornen 0 o un valor positiu (depenent de la ordre).

Cada dispositiu de caràcter ha de tenir, com a mínim, les operacions *open* i *release*. Les altres ordres poden ser implementades o no.

2. Descripció del CDX

En aquest capítol es farà una descripció del funcionament del CDX: quines parts el composen, com s'estableix la comunicació entre elles, etc. Avancem però que el CDX només implementa els dispositius de caràcter, no de bloc, donat que, sinó, el treball seria molt més extens.

2.1 Funcionament general

El CDX actua com un controlador de qualsevol dispositiu de caràcter que estigui situat a un ordinador connectat en xarxa. O sigui, quan vulguem usar un dispositiu de caràcter d'un altre ordinador, haurem d'usar el dispositiu que crea el CDX, i aquest s'encarregarà d'enviar la ordre per la xarxa cap al dispositiu de l'ordinador de destí corresponent.

Per tal de diferenciar quin dispositiu volem usar, el CDX utilitza el número minor, i, amb l'ajuda del fitxer de configuració, farà la traducció del minor cap a un ordinador remot i una ruta cap a un dispositiu dins el sistema d'arxius de l'ordinador. Així, totes les entrades de dispositius remots tindran un número major comú (actualment, el 254), i el minor servirà per distingir quin dispositiu i ordinador usem.

A fi de facilitar la configuració del CDX, s'estableix que tots els dispositius remots estaran al directori */dev/cdx*. A partir d'aquest directori, trobarem tants directoris com ordinadors remots vulguem usar. Llavors, anomenarem les entrades de dispositiu amb el mateix nom que el dispositiu en l'ordinador en xarxa, però tenint en compte posar el mateix número major per a tots, i cada un amb minor diferent. En el capítol següent es parlarà més àmpliament de com es fa la traducció, i com usar el fitxer de configuració. Avancem només, però, un exemple:

> Suposem que tenim 3 ordinadors connectats a través d'una LAN, anomenats cada un: Zeus, Apolo i Neptú. Des de l'ordinador Zeus volem fer us de la targeta de so instal·lada a Apolo, i del mòdem present en Neptú. A Zeus haurem d'instal.lar el mòdul CDX i el client, i als altres dos, el servidor. Sabem que la targeta de so s'anomena "/dev/dsp" i el mòdem s'anomena "dev/ppp". Llavors, tindrem dues entrades de dispositiu a Zeus, "dev/cdx/Apolo/dsp" i "/dev/cdx/Neptu/ppp", amb minor 1 i 2 corresponentment (Fig. 1). Llavors, si, des de Zeus volem usar la targeta de so instal·lada a Apolo, farem referència a "/dev/cdx/Apolo/dsp", i és aquí on entrarà en

funcionament el CDX, que, mitjançant la configuració, sap que el valor minor 1 fa referència a l'ordinador Apolo, al dispositiu "/dev/dsp"

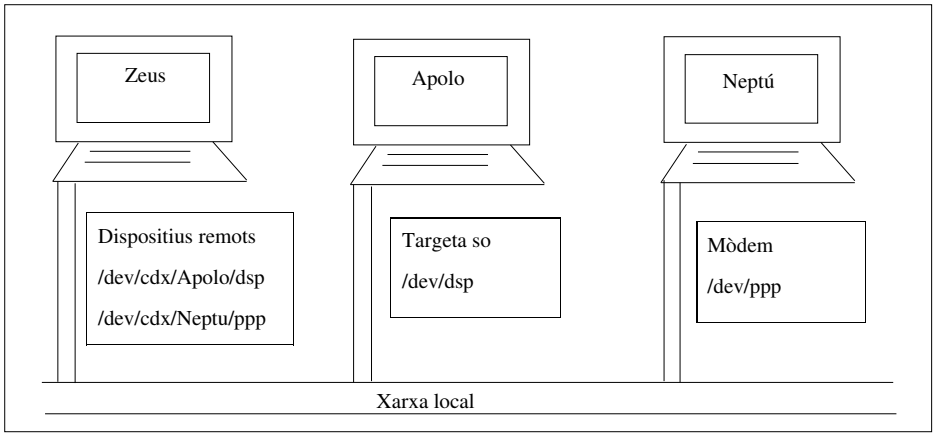

Fig. 1. Exemple de dispositius remots

2.2 Parts del CDX

El CDX es composa de tres parts: el mòdul del kernel, el client i el servidor. El mòdul és l'encarregat de rebre peticions d'ordres procedents de processos cap a dispositius remots, les traspassa al client, i el client les envia per xarxa cap el servidor. Quan l'ordre s'executa a l'ordinador servidor, s'envia el resultat del servidor cap al client, i aquest el traspassa cap el mòdul, el qual, finalment, l'envia cap el procés.

Així tindríem, en l'ordinador client, el mòdul CDX i el programa client. I en l'ordinador (o ordinadors) que vulguem usar algun dispositiu, el servidor(Fig. 2). Es va pensar en una implementació en només dues parts, el servidor i el mòdul (el qual fa de pont entre el procés que crida la ordre i el servidor, mitjançant la connexió per xarxa), però va descartar-se degut a dues raons:

1. La complexitat de programació: es més difícil programar parts del sistema operatiu que programes d'usuari

2. La dificultat de depuració: en efecte, si la part del sistema operatiu no funciona bé, podem bloquejar tot l'ordinador.

Així, va escollir-se la divisió en 3 parts, revisant al màxim la part del mòdul, per evitar qualsevol error.

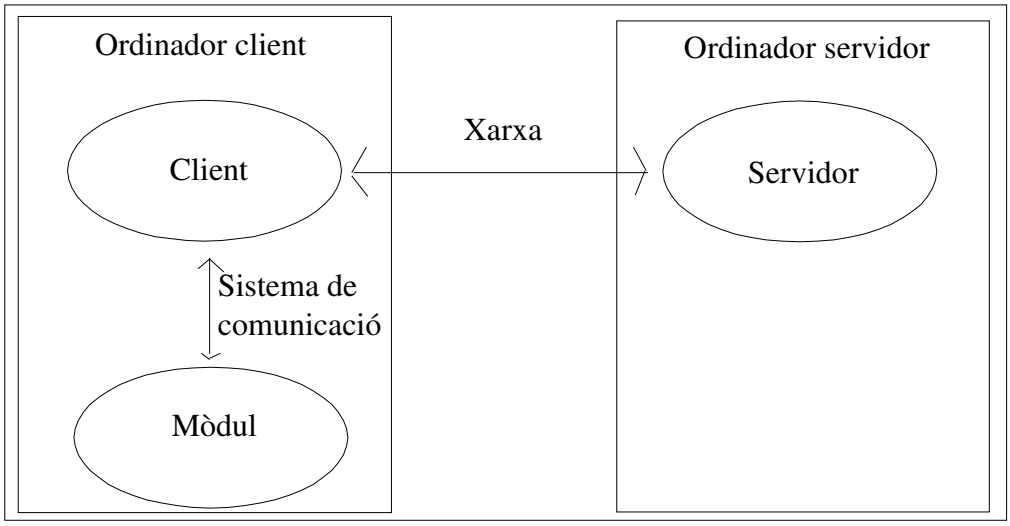

Fig. 2. Esquema de comunicació del CDX

2.3 Interacció entre el mòdul i el programa client

Com ja he comentat, en l'ordinador client s'executarà el mòdul i el programa client (*cdx_client*). El mòdul l'hem de carregar al sistema operatiu, amb l'ordre "*insmod cdx.o"*; llavors, hem d'executar el client. Llavors, s'ha d'establir una comunicació entre el client i el mòdul. Per tal d'això, s'utilitzarà una entrada especial de dispositiu, amb el mateix major que utilitza el CDX, però amb un minor especial, que no representa cap dispositiu remot, sinó un dispositiu especial de comunicació directa amb el mòdul. La comunicació s'estableix mitjançant ordres ioctl (del client cap el mòdul) i senyals (del mòdul al client).

Així, en el moment d'executar el client, s'obre el dispositiu especial de control (anomenat "*/dev/cdx/cdx")*; per tal d'evitar usos accidentals o malintencionats al mòdul, just després d'obrir-lo, s'ha d'enviar una cadena de text identificativa, que s'estableix com a: "Protocol comunicacio amb modul CDX V1.0" . També ens serveix per controlar diverses versions del protocol. Aquest fet l'he anomenat "registrar el client"

Aquesta cadena s'envia amb una ordre especial *ioctl* cap el mòdul. Si el mòdul l'accepta, tindrem el client registrat, i l'ordre retorna un valor d'O.K. Això també fa que el mòdul emmagatzemi l'identificador del procés client (*PID*[1]) per a usos

1 Veure "3.6 Estructura d'un procés"

futurs.

Hem de fer un aclariment: en principi, el mòdul refusarà acceptar cap registre de client si la cadena no és igual a la que disposa. Si en un futur es canvia el format del protocol, llavors la cadena s'hauria de canviar. És aquí on el mòdul podria decidir acceptar comunicacions de diferents versions del protocol, comportant-se adequadament.

Recordem ara el concepte de l'estructura *file*, que representa un descriptor de fitxer. Aquesta sempre s'envia en qualsevol ordre de dispositiu, i és diferent per a cada vegada que s'obre un dispositiu. Llavors, cada ordre de dispositiu que es rebi, segons el valor de *file*, anirà destinada a un dispositiu remot o un altre.

2.3.1 Atenció de les ordres d'obertura

Introduïm breument el concepte de *procés fill*: és aquell procés resultat d'executar una ordre del S.O. (en el cas de Linux, fork()), que és una còpia exacte del procés originari (procés pare). Hereta tota l'àrea de dades i codi, i també tots els descriptors de fitxers oberts.

Amb això, podem veure que, un procés que obri un dispositiu, i després creï un procés fill, estarà compartint les mateixes estructures *file* amb el fill per aquest dispositiu que ja estava obert abans.

Amb el concepte de procés fill que acabem de definir, podem explicar com s'atenen les ordres als diferents dispositius: en el moment que el mòdul rep una ordre d'obertura, avisa al client [1](si és que està registrat) dient-li que s'ha invocat una ordre d'obertura de dispositiu. Llavors, el client llegeix del mòdul el número minor del dispositiu a obrir, i el valor de l'adreça del descriptor de fitxer (*file*). Immediatament es crea un procés fill per tal que atengui les peticions cap el descriptor de fitxer associat. El procés pare registra el fill cap el mòdul, dient-li el codi identificatiu del procés fill (PID) i el valor del file. És llavors quan el fill ha d'enviar l'ordre d'obertura cap el servidor.

Aquest procés fill (*client fill*) serà avisat directament pel mòdul cada vegada que es rebi una ordre de dispositiu que vagi associada al *file* que gestiona.

Així podem veure que hi haurà dos tipus de comunicació entre el mòdul i el client:

1 S'avisa mitjançant senyals (SIGNALS) de Linux. Veure següent capítol per a més informació

1. Quan hi hagi una ordre d'obertura, s'avisarà el client pare, per tal que creï un procés fill

2. En la resta d'ordres, s'avisarà el client fill corresponent al *file* associat.

2.3.2 Atenció de les ordres als dispositius

Per tal de saber quins fills atenen els descriptors de fitxers, el mòdul guarda una llista amb tots els *file* oberts i l'identificador de procés (PID) del client fill que el gestiona. Llavors, quan es rep una ordre sobre un dispositiu (sempre que no sigui d'obertura) s'avisa mitjançant una senyal al procés fill que atén el *file* corresponent.

El fill llavors ha de llegir el tipus d'ordre i els seus paràmetres; per tal d'això, crida al mòdul mitjançant una ordre *ioctl*.

El mòdul ha de saber quina ordre i paràmetres ha de passar al procés fill; és aquí on entra en joc la "llista de peticions": aquesta és un vector d'estructura de peticions pendents ser executades, que emmagatzema, per cada petició demanada, el tipus d'ordre, els seus paràmetres, el file associat, el PID del procés que la crida...

Llavors, quan el mòdul rep una ordre de lectura de petició per part d'un procés fill, aquest busca en la llista de peticions la primera que es correspon al file que gestiona el procés fill. Al donar les dades al procés, es canvia l'estat de la petició per tal d'avisar que ha estat recollida (però no executada encara).

En el moment que el client l'ha recollit, passa l'ordre al servidor, i quan li retorna les dades de resultat, envia el retorn al mòdul. En aquest cas, el mòdul buscarà la primera ordre pendent de ser retornada, i corresponent al file que gestiona el fill. Llavors, al trobar-la, es retornarà el resultat al procés.

Hem d'aclarir dues coses, que es dedueixen directament de la descripció dels processos fills:

1. Pot haver, pel mateix file, més d'una petició pendent de ser recollida. Podríem pensar que això no pot ser possible, donat que un file és diferent per a cada dispositiu obert, encara que s'obri el mateix exactament. Però hem de considerar l'herència de processos, i el fet que un procés fill heretarà tots els descriptors de fitxers (file) oberts del pare; amb això, podríem tenir que pare i fill executessin ordres al mateix dispositiu, i consegüentment, usant el mateix file.

2. Mai hi haurà, pel mateix file, més d'una petició pendent de ser retornada. Això és degut a que el client fill només llegeix i executa una petició cada vegada.

2.3.3 Posant processos en espera

Al parlar de quan s'executa una ordre cap el mòdul, no hem dit què passa amb el procés que la fa fins que es rep el resultat. El que succeeix és que posem el procés en un estat d'espera (*sleeping* en anglès, "dormint"). No podríem fer que, quan es demanés una ordre, i després d'afegir-la a la llista de peticions i avisar el client fill retornéssim de la funció; això només ha de passar quan tinguem el resultat de la ordre.

Així, quan el fill ha executat la ordre i rebut el resultat, avisa el mòdul. Aquest busca la primera petició pendent de retorn, i, segons recordem, aquesta també emmagatzema el PID del procés que la va fer. Amb això, podem despertar (*wakeup)* el procés mitjançant el PID i retornar-li el resultat.

2.3.4 Tancament de dispositius

Quan el dispositiu ha de ser tancat, s'envia l'ordre del procés cap el dispositiu, que és recollida pel mòdul CDX. Aquest, en lloc d'enviar una senyal genèrica de petició al client fill, envia un altre tipus de senyal (en aquest cas, SIGINT). Això es fa així donat que, aquesta senyal, és la mateixa que s'envia al executar una cancel·lació de procés (CTRL+C); així, quan es cancel·la el client, rebrà l'ordre de tancament de dispositiu. Llavors, el client fill la rep, envia la senyal de tancament de dispositiu al servidor, i quan ha finalitzat, avisa al mòdul amb el resultat, amb lo qual, el procés fill finalitza.

Hem de dir també, que quan volem finalitzar tot el programa client, al prémer CTRL+C, s'envien senyals de finalització al pare i als fills. Llavors el pare espera fins que acabin tots els processos fills abans de finalitzar ell mateix.

2.4 Interacció entre el client i el servidor

Fins ara només hem parlat de què passa entre el mòdul i el client. Ara veurem com es comuniquen client i servidor.

Considerem, com a mínim, un servidor executant-se (ja sigui en un ordinador remot o en el mateix en el que s'executa el client). Quan es rep una ordre d'obertura sobre un dispositiu, és recollida pel mòdul i traspassada cap el client. Llavors, és el client qui ha de passar-la al servidor. Aquesta petició implica obrir un dispositiu en

l'ordinador remot, i sempre serà el mateix client fill qui enviarà les peticions per aquest descriptor de fitxer associat. Així, de la mateixa manera que hem fet amb el client, el servidor crearà tants processos fills com dispositius es vagin obrint.

Recordem que client i servidor es comuniquen mitjançant una xarxa TCP, fent us de la llibreria *socket* de Linux. Així, quan es vol establir una comunicació, hem de fer una ordre de connexió TCP, al port 33333, que és el que escolta el servidor, amb la qual, el servidor crearà un procés fill per atendre les peticions amb aquest client.

Totes les ordres enviades entre client i servidor utilitzen un format de dades anomenat: "Trama CDX", el qual va encapsulat dins una trama TCP. En el següent capítol es parlarà mes àmpliament d'aquest format.

2.4.1 Identificació del client cap el servidor

Al igual que succeeix amb la connexió entre el client i el mòdul, aquest primer ha d'enviar una cadena identificativa al servidor, per evitar-nos connexions accidentals per part d'un altre programa i també pel control de versions en el protocol de trames CDX. La cadena a enviar és: "Protocol comunicacio de xarxa CDX V1.0".

Llavors, quan el servidor rep la cadena identificativa, si és la mateixa que usa el servidor, retornarà un valor de O.K. Sinó, retornarà error. Al igual que el registre del client amb el mòdul, el servidor pot decidir acceptar connexions de clients amb protocol de connexió diferent.

2.4.2 Atenció de les ordres als dispositius

Com hem dit abans, cada servidor fill atendrà un descriptor de fitxer diferent. Cada un tindrà establerta una connexió TCP amb el client corresponent. Llavors, quan es genera una ordre de dispositiu, el client fill l'envia cap el servidor associat en forma de trama CDX. Llavors, aquest executa la ordre en el dispositiu.

Quan s'ha executat la ordre, s'envia una trama CDX del servidor fill cap el client associat amb el resultat de la mateixa.

2.4.3 Posant el client en espera

Anàlogament amb el que passava amb el procés que executa la ordre, el client que envia la trama s'ha de posar en un estat d'espera, fins que rep el resultat.

Afortunadament, i de manera lògica, això es fa automàticament, ja que el client, després d'enviar la trama, executa una ordre *read* de connexió, amb lo qual, es posarà automàticament en espera fins que hi hagi alguna dada disponible per llegir (o sigui, quan es rebi el retorn per part del servidor).

2.5 Tractament dels errors

Hem de descriure quins errors es poden generar en tot el procés d'execució d'una ordre cap a un dispositiu, i des de quina part de tot el CDX poden provenir: generat pel mòdul, pel client o pel servidor. També parlarem del codi d'error que rep el procés que executa la ordre, p.ex: quan el client intenta connectar-se amb el servidor i aquest no respon, no es pot retornar el missatge de "connexió refusada" (és el que es genera quan no es pot establir una connexió TCP), ja que el procés que executa la ordre no sap, en principi, que estem establint una connexió per xarxa. En aquest cas, es retornarà un missatge dient al procés que "el dispositiu no existeix".

A part d'això, també es té el missatge d'error "natural" generat per l'execució de l'ordre sobre el dispositiu en el servidor; en aquest cas, el codi d'error es passa directament al procés sense cap traducció. Aquest codi no és tractat ni pel mòdul, ni pel client ni pel servidor; és el procés que crida l'ordre qui ha d'actuar adequadament.

2.5.1 Errors generats pel mòdul

El mòdul CDX pot generar molts missatges d'error, degut a vàries causes: esgotament de la memòria, client no autoritzat, etc. Tots els missatges d'error generats s'escriuen al registre d'events del sistema operatiu, normalment es tracta del fitxer "/var/log/messages"ÿ. Aquí es mostrarà la causa real de l'error, a diferència del codi que rep el procés, que sol ser "Error d'entrada/sortida" o "Dispositiu inexistent", segons el cas.

2.5.2 Errors generats pel client

El client pot generar diversos errors, deguts a dues causes:

1. Errors generats en la comunicació amb el servidor: si el servidor no respon, si ha tancat la connexió inesperadament... En aquests casos, s'enviarà un error de "Entrada/Sortida" o de "Dispositiu Inexistent".

2. Errors generats dins el mateix client: degut a esgotament de la memòria o de comunicació amb el mòdul: Normalment, es retornarà el missatge de “Error de entrada/sortida”

El missatge amb l'error real s'escriu a la pantalla, a la sortida estàndard de dades (*stdout)*.

2.5.3 Errors generats pel servidor

Aquí també tenim que es poden generar diversos errors segons la causa, també semblants: memòria, comunicació, etc. Aquí, però, se li fa un tractament diferent; quan s'envia la trama CDX de retorn al client, se li retornen dos tipus d'error: un és el codi que rebrà directament el procés, sense tractament. L'altre és un codi, en principi, acordat entre el client i servidor, amb la causa real de l'error, que ha de tractar el client adequadament. Cal dir que, de moment, el codi d'error destinat al client no s'utilitza, sinó que es deixa per una futura implementació.

Al igual que el client fa, s'envia a la pantalla el missatge d'error amb la causa real.

2.6 Ordres de dispositius suportades pel CDX

Les ordres de dispositius es poden classificar en tres tipus: lectura de dades (es retorna del servidor al client un bloc de dades), escriptura de dades (s'envia del client al servidor un bloc de dades), i mixtes (s'envia i es retorna un bloc de dades). A part d'això, totes les ordres tenen uns paràmetres que s'han d'enviar del client al servidor, i un valor de retorn del servidor al client. Les ordres suportades pel CDX i la seva classificació és:

1. Ordres de lectura: read, ioctl

2. Ordres d'escriptura: open, write, lseek, close,ioctl

3. Ordres mixtes: ioctl

Aquestes ordres són les més usades, i suficients per a que puguin funcionar tots els dispositius de caràcter; la resta, són poc usades.

Degut a l'estructura usada en les trames CDX, abans de rebre una trama sencera,

es pot saber la seva longitud, assignant suficient memòria per recollir les dades. Això és perquè el primer camp de la trama indica la longitud del bloc que ve a continuació. Es detallarà més àmpliament les trames CDX al següent capítol.

Comentem també, que aquestes ordres seran enviades tant entre mòdul i client, i entre client i servidor, els quals tenen dos mètodes de comunicació diferents: ordres CDX i trames CDX, però els dos són equivalent.

Seguidament, veurem per separat els tres tipus d'ordres.

2.6.1 Lectura de dades

Aquestes ordres implica enviar del client al servidor només els paràmetres de la lectura. El servidor s'encarrega d'assignar memòria suficient per guardar el bloc de dades a llegir. En el moment que s'han llegit, són enviades cap el client, juntament amb el valor de retorn, i és el client qui s'encarrega d'enviar-les al mòdul (realment és el mòdul qui les "agafa" del client, però això es veurà en el següent capítol).

2.6.2 Escriptura de dades

Aquestes impliquen enviar unes dades a escriure cap el servidor, el qual ha de disposar de memòria suficient per recollir-les. Llavors, el servidor executa l'ordre amb les dades rebudes, i el resultat obtingut és enviat cap el client, el qual l'envia al mòdul. Destaquem, però, que potser la única ordre "real" d'escriptura de dades és *write*, les altres només escriuen paràmetres al dispositiu, no un bloc de dades.

Comentem aquí a part l'ordre lseek: aquesta, quan es cridada des de la llibreria del sistema utilitza 32 bits pels valors de desplaçament:

```
off_t lseek(int fildes, off_t offset, int whence);
```

En canvi, quan es crida la funció del kernel, s'utilitza 64 bits:

```
loff_t (*llseek) (struct file *, loff_t, int);
```

Això és així per tal de que el nucli del sistema estigui preparat per a una implementació en 64 bits. Llavors, el mòdul CDX utilitza 64 bits en aquesta ordre; en el servidor, s'utilitza l'ordre:

```
off64_t lseek64(int fildes, off64_t offset, int whence);
```

I està present a la llibreria del sistema[1].

2.6.3 Ordres mixtes

Aquestes ordres (dins les quals, només hi troben la ordre ioctl), impliquen enviar un bloc de dades i rebre'n un altre. Veiem també, que aquesta ordre la trobem dins les tres classificacions, degut a que la seva funció pot ser qualsevol que entengui directament un dispositiu: escollir freqüència de mostreig, formatejar un disc, llegir una seqüència de vídeo... El problema sorgeix amb la dificultat de saber de quin tipus (lectura/escriptura/mixta) d'ordre tracta en cada moment la ioctl, el tamany del buffer emprat o la quantitat de paràmetres. És per això que ho expliquem més àmpliament a continuació.

2.6.4 Ordre ioctl

La ordre ioctl és definida a la llibreria del S.O. com el següent:

int ioctl(int d, int peticion, ...)

El valor de retorn sempre serà un enter. Dins els paràmetres, trobem el primer, que és el valor del descriptor de fitxer usat per la llibreria, i el segon, que és un número de comanda escollit arbitràriament que se li ha d'enviar al dispositiu; seguint amb l'exemple de la targeta de so, un número de comanda s'usaria per canviar la freqüència de mostreig, l'altre per commutar entre estèreo/mono, etc. Fem constar que aquest número pot tenir el mateix valor per ordres sobre diferents dispositius[2].

A partir del segon paràmetre, pot haver-hi quants vulguem, 0 o més, tot i que la majoria d'ordres són de 0 o 1 paràmetre addicional.

En canvi, la definició de l'ordre ioctl que rep el controlador de dispositiu és la següent:

int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);

Els dos primers paràmetres són l'inode (identificació del fitxer) i el file (descriptor de fitxer). El tercer paràmetre és el número de la comanda (la “peticio” en la primera definició). I el quart paràmetre tant pot ser el valor del primer

1 Veure fitxer /usr/include/unistd.h
2 Veure el manual de “ioctl_list”

paràmetre de la ordre ioctl (en crides amb només un paràmetre addicional), com un punter a una adreça de memòria (on estan situats tots els paràmetres addicionals o on s'han de guardar les dades llegides), o pot no ser cap valor significatiu (en crides amb 0 paràmetres addicionals).

En principi, d'això podem deduir tres problemes:

1. Dificultat per determinar si l'ordre és d'escriptura, lectura o mixta

2. Dificultat per determinar si l'ordre té algun paràmetre addicional

3. Dificultat per determinar el tamany del bloc de dades usat

Per tal de solucionar-lo, inicialment, podríem establir tota una taula amb les comandes possibles, i els dispositius associats i que ens digués aquesta informació: tamany, paràmetres i tipus. Això, però, implicaria tenir una quantitat molt gran de dades, que hauria de modificar-se sempre que alguna definició d'ioctl variés. També, estaríem en contra dels objectius d'aquest projecte, un dels quals és el d'establiment d'un controlador universal per a qualsevol tipus de dispositiu, no fer un tractament especial per cada un d'ells.

Sembla ser que, els programadors del sistema Linux, també es van plantejar una qüestió semblant. Diem això perquè, a partir de les versions del kernel 2.2, es recomana no establir el número de comanda del dispositiu arbitràriament, sinó amb un format estandarditzat, codificant el tipus de comanda i el tamany del bloc de dades dins els 32 bits que ocupa el valor de la comanda:

31.....30	16.....29	15.....8	7.....0
2 bits	14 bits	8 bits	8 bits
Direcció	Tamany dades	Tipus comanda	Numero de la comanda

La direcció ens indica si l'operació és de lectura, escriptura, mixta, o no implica dades. El tamany de les dades indica els bytes que ocupa la estructura a escriure/llegir. El tipus ha de ser un número diferent per a cada dispositiu, escollit arbitràriament. I el número de la comanda és l'ordre a realitzar dins el dispositiu.

Llavors, amb aquesta informació continguda dins el paràmetre de comanda, el CDX es comporta adequadament, podent distingir el tipus de comanda i el tamany

de les dades tractades[1].

Desafortunadament, només una petita part de tots els controladors segueixen aquest estàndard.

2.7 Diagrama d'interaccions i estats del CDX

En aquest diagrama (Fig. 3) es vol fer una descripció gràfica de la comunicació entre les diferents parts del CDX i dels estats que travessen cada una, :

· Un procés pot estar dormint o esperant resposta del mòdul.
· El mòdul pot estar esperant resposta per aquest procés en particular, o inactiu
· El client fill pot estar adormint o esperant resposta del servidor
· El servidor fill pot estar adormint o executant la ordre

1 Veure capítol següent per a més informació

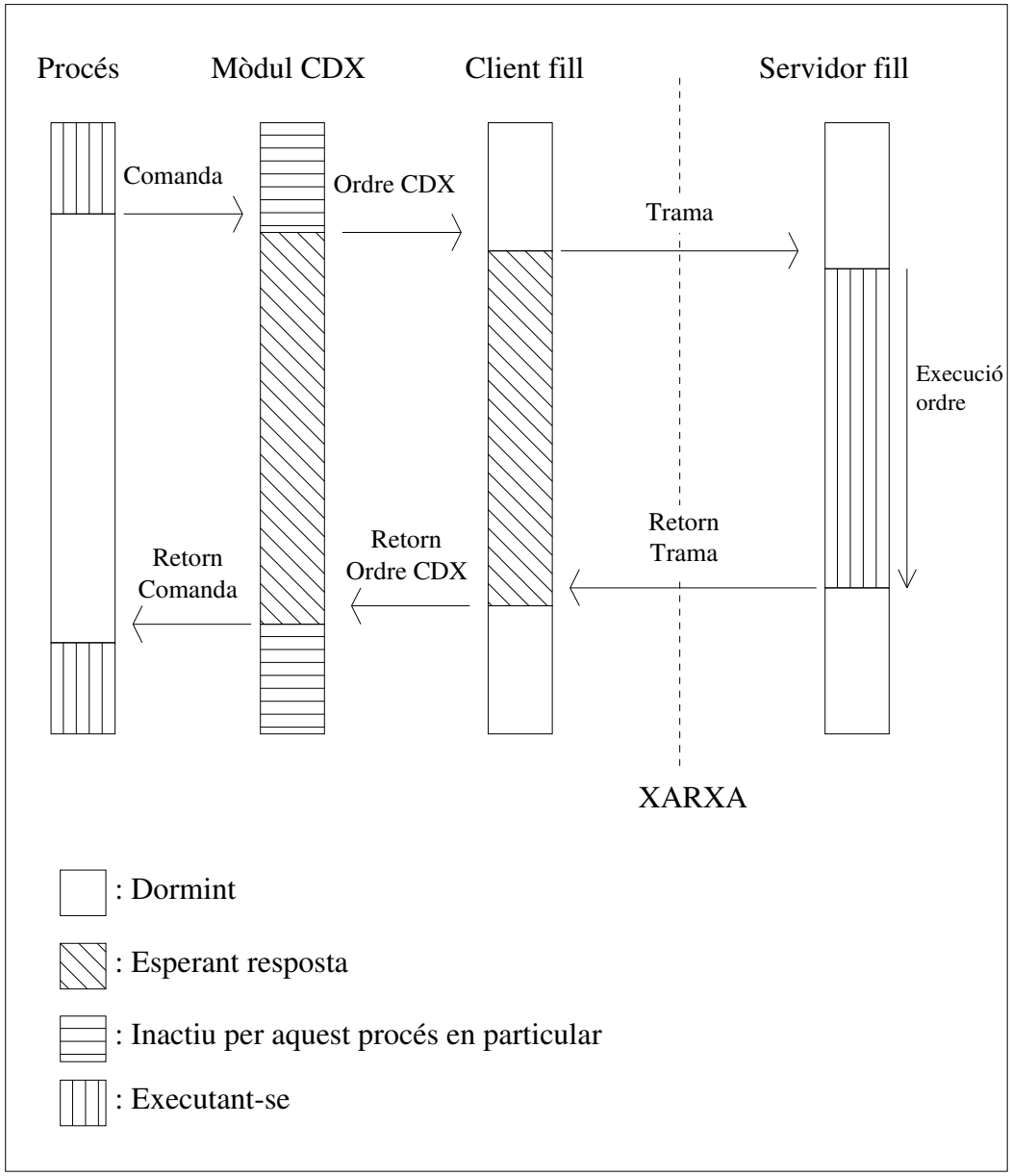

Fig. 3. Diagrama d'interaccions i estats del CDX

3. Tècniques usades en el CDX

En aquest capítol farem una explicació àmplia de totes les tècniques usades en aquest projecte, es dirà el "com" s'ha fet, mentre que en el capítol anterior es deia el "què" s'ha fet.

Cal destacar que totes les estructures de dades, funcions, i lògica de programació s'ha tingut que crear totalment de zero, excepte, com és clar, de les trucades a les llibreries del sistema o del nucli.

3.1 Fitxers de codi font utilitzats

Aquí descriurem els fitxers de codi utilitzats pel CDX, i les funcions que realitza cadascun:

· cdx.c : En aquest fitxer es troba tot el codi font del mòdul CDX. Al compilar-lo, genera el mòdul cdx, cdx.o. Al principi d'aquest fitxer trobem totes les funcions auxiliars que es fan servir dins el mòdul; després, trobem les ordres de dispositiu. Dins d'aquesta segona part, una ordre molt important és la del control de les ordres ioctl, que té dues funcions:

1. Atendre les ordres ioctl provinents d'un procés.

2. Respondre les ordres ioctl enviades des del client, que són la base de la comunicació entre aquest i el mòdul.

· cdx.h : En aquest fitxer es troben vàries definicions usades en la comunicació entre mòdul i client, i en el mòdul mateix. També trobem uns macros usats per a la conversió de dades en punters, útils al tractar amb trames CDX i ordres CDX, per exemple:
#define VALOR_long_long(p) (*((long long *)(p)))
Aquesta macro fa que podem tractar el contingut d'un punter com un tipus de dades long long.

· cdx_util.c : Aquí trobem vàries funcions usades tant pel client com pel servidor, tals com assignació de memòria, connexió per xarxa, i lectura de fitxer de configuració.

· cdx_util.h : En aquest fitxer es troben les declaracions de les funcions presents

a cdx_util, i algunes definicions de valors.

· cdx_client.c : Aquí es troba el codi corresponent al client pare.

· cdx_client_fill.c : Aquí trobem el codi corresponent al fill del client.
Aquests dos fitxers, juntament amb cdx_util.c, s'enllacen per tal de formar un sol fitxer executable, cdx_client.

· cdx_servidor.c : Aquí es troba el codi corresponent al servidor pare.

· cdx_servidor_fill.c : Aquí trobem el codi corresponent al fill del servidor.
Aquests dos fitxers, juntament amb cdx_util.c, formaran un sol fitxer executable, cdx_servidor.

Els fitxers de codi font usats per fer proves (provaX.c) es detallaran en el capítol de proves, donat que no es necessiten al CDX pel seu funcionament.

3.2 Fitxer de configuració del CDX

Al igual que en la majoria d'aplicacions de Linux/UNIX, la configuració es troba emmagatzemada en fitxers de text, la qual podem canviar amb qualsevol editor. També, com a Linux, una línia que comenci amb el caràcter (#) significa un comentari, amb el qual, el programa ignora aquesta línia. Normalment, s'emmagatzema, per cada línia, una comanda de configuració, separant els paràmetres per espais (o codi de tabulació).

Aquest fitxer de configuració l'he anomenat “cdx_importa”, ja que importa dispositius d'ordinadors remots. Idealment, i seguint la filosofia de Linux, hauria d'estar emmagatzemat al directori /etc, però el CDX el localitza al directori actual, per facilitar la configuració.

En aquest fitxer, s'emmagatzema la correspondència entre el número minor del dispositiu CDX i l'ordinador i dispositiu remot. El primer camp és el número minor, el segon el nom de l'ordinador remot (ja sigui una adreça IP, un nom registrat dins el DNS, o un nom d'ordinador local), i el tercer camp és el nom del dispositiu dins l'ordinador remot.

Un exemple de fitxer de configuració seria el següent:

0 localhost /dev/dsp
1 localhost /dev/lp0
2 localhost /dev/fb0

3 localhost /dev/psaux
4 localhost /dev/radio

10 Apolo /dev/dsp
11 Apolo /dev/lp
12 Apolo /dev/fb0

20 Neptu /dev/dsp
21 Neptu /dev/lp0
22 Neptu /dev/fb0
23 Neptu /dev/radio

En aquest exemple, usem dispositius tant de l'ordinador local (localhost) com de dos ordinadors remots, Apolo i Neptu. Llavors, dins l'ordinador local s'hauria de tenir una estructura adequada, dins el directori /dev/cdx, dels dispositius usats (Fig. 1)

Nom dispositiu	Major	Minor
|-- localhost		
| |-- dsp	254	0
| |-- lp0	254	1
| |-- fb0	254	2
| |-- psaux	254	3
| |-- radio	254	4
|		
|-- Apolo		
| |-- dsp	254	10
| |-- lp	254	11
| |-- fb0	254	12
|		
|-- Neptu		
|-- dsp	254	20
|-- lp0	254	21
|-- fb0	254	22
|-- radio	254	23

Fig. 1. Jerarquia de dispositius dins l'ordinador local

El fitxer “cdx_importa” es llegeix al principi d'executar el client, utilitzant la

funció present a cdx_util.c:

int cdx_llegeix_n_camps (FILE *f, int *linia_nova, int n, ...);

El primer paràmetre és el descriptor de fitxer, el segon és un indicador que internament utilitza la mateixa funció i que hem d'inicialitzar a 1, el tercer paràmetre indica el número de camps a llegir en el fitxer de configuració, i a partir d'aquí, hem de passar paràmetres (de tipus char *) on s'emmagatzemarà cada camp llegit, com a cadena de caràcters. Aquesta funció s'ha de cridar contínuament, una vegada per cada línia, fins que trobem el final de fitxer o un codi d'error.

Aquesta funció, quan és cridada des de *client.c*, s'emmagatzema cada camp dins un vector d'una estructura especial:

```
//Estructura de la taula de dispositius importats
struct t_taula_importacio {
  unsigned char minor;
  char host[255];
  char device[255];
};

//Vector de dispositius importats
struct t_taula_importacio taula_importacio[CDX_MAX_DISPOSITIUS];
```

La part de codi que crida la funció i emmagatzema cada camp adequadament és la següent:

```
//Obrir fitxer configuracio
if ((f_configuracio=fopen(CDX_IMPORTA,"r"))==NULL) {
  printf ("cdx_client: Fitxer %s no trobat!\n",CDX_IMPORTA);
  exit(1);
}

//Llegir Dades
dispositius=0;
do {
  linia_nova=1;
  i=cdx_llegeix_n_camps(f_configuracio,&linia_nova,3,s0,s1,s2);
  if (i<=0) {
    if (i==CDX_NO_MES_DADES)
      printf ("cdx_client: No hi ha més dades. OK\n");
    if (i==CDX_FALTEN_PARAMETRES) {
      printf ("cdx_client: Falten camps a la linia\n");
```

```
        exit(i);
      }
      if (i==CDX_SOBREN_PARAMETRES) {
        printf ("cdx_client: Sobren parametres\n");
        exit(i);
      }
    }
    else {
      taula_importacio[dispositius].minor=atoi(s0);
      strcpy(taula_importacio[dispositius].host,s1);
      strcpy(taula_importacio[dispositius].device,s2);
      printf  ("cdx_client: Minor: %d Host:%s  Device:%s\n",
           taula_importacio[dispositius].minor,
           taula_importacio[dispositius].host,taula_importacio[dispositius].device);
      dispositius++;
     }

    } while (!feof(f_configuracio));
    fclose(f_configuracio);
```

3.3 Utilització dels mòduls

Com ja hem comentat en el capítol anterior, la inserció de mòduls es pot fer de manera automàtica o manual.

Pel mètode manual, es buscarà el mòdul indicat (mitjançant la ordre *insmod*) dins el directori de mòduls (normalment, a partir de /lib/modules). Si aquí no es troba, es mira en el directori actual[1].

Pel mètode automàtic, hem de tenir adequadament configurat un fitxer, anomenat /etc/modules.conf, en el qual podem indicar quin mòdul gestiona cada dispositiu, mitjançant el número major. Per exemple:

```
alias char-major-81 bttv
```

Això indica que, quan vulguem fer us d'un dispositiu amb major igual a 81, hem de carregar el mòdul bttv (corresponent a una targeta sintonitzadora de T.V.)

Ara bé, quan deixem d'usar el dispositiu el mòdul encara restarà a la memòria; si

1 Realment, el procés és més complicat, fent ús d'una llista de dependència de mòduls i també d'àlies, però no és objecte d'aquest TFC explicar-ho. Per a més informació, consultar el manual de *depmod, modprobe, insmod*

volem treure'l, hem de cridar el programa del S.O. *rmmod*. Abans de finalitzar el mòdul, es crida a la funció *cleanup_module,* on es poden fer tasques d'alliberament de memòria, etc (en efecte, al mòdul CDX, s'allibera tota la memòria usada en la llista de peticions i buffers d'intercanvi).

Aquesta ordre de treure el mòdul de la memòria només pot ser executada si no s'està usant el dispositiu per cap procés; això és gestionat mitjançant un comptador dins el mòdul (*usage count)* que s'incrementa cada vegada que s'obre el dispositiu, i es decrementa cada vegada que s'allibera el file[1].

3.4 Missatges de depuració

En totes les parts del CDX (mòdul, client i servidor) s'ha disposat d'un mètode per facilitar la depuració i el control d'errades en la depuració. Es realitza mitjançant l'escriptura de missatges amb les operacions que es realitzen. Aquests missatges, des del mòdul, es registren al fitxer d'events del sistema operatiu, mentre que des del client o el servidor van a parar a la pantalla (veiem que s'escriuen igual que si fossin missatges d'error).

Ara bé, quan no s'ha de realitzar tasques de depuració, pot resultar molt incòmode (i fins i tot lent) tenir tants missatges impresos. Per tal d'això, tots els missatges generats porten una condició del pre-compilador, depenent de si està definida la variable DEBUG. Si no està definida, el missatge no es genera.

Hem de dir també que el missatge està encapçalat pel nom de la part del CDX que el genera, ja sigui: “CDX” (el mòdul), ”cdx_client” (el client pare), ”cdx_client_filll”, ”cdx_servidor”, ”cdx_servidor_fill”, ”cdx_util” (provinent d'alguna funció dins de cdx_util.c)

Llavors, quan no volem veure els missatges, només hem d'esborrar (o marcar-la com a comentari) la definició:

```
#define DEBUG
```

que es troba present cap el principi de cada fitxer font.

Veiem un exemple al mòdul, present a la funció *device_read*:

1 Veure el manual de *lsmod* per tal de veure els mòduls que hi ha a la memòria i el *usage count de cadascun.*

```
.......
#ifdef DEBUG
  printk ("CDX: Peticio read minor: %d, file: %p, pid:"
            "%d\n",minor,file,current->pid);
  printk ("CDX: longitut:%d\n",longitut_fragment);
#endif
.......
```

S'utilitza *printk*, que envia el missatge al registre d'events del S.O.

I un exemple de missatge en cdx_client_fill, dins la funció llegir_dispositiu:

```
.........
#ifdef DEBUG
  printf ("cdx_client_fill: Ordre read, longitut: %u\n\n",
          longitut);
#endif
........
```

Veiem en canvi, que aquí s'utilitza printf.

3.5 Senyals a Linux

Les senyals al sistema operatiu Linux (com a tots els UNIX), formen part dels sistemes de comunicació de processos (IPC en terminologia Linux, *InterProcess Comunication)*. Aquestes es generen cridant a ordres del S.O., indicant el tipus de senyal a enviar i el procés en qüestió que l'ha de rebre.

Un procés pot preparar-se a rebre diferents tipus de senyals, amb lo qual, si rep una per la qual estigui preparat, es cridarà una funció (que ell hagi indicat), amb lo qual, sabrà que ha rebut aquesta senyal.

Veiem un exemple de registre de senyal per part d'un procés:

```
  if (signal(SIGUSR1,tractament_senyal)==SIG_ERR) {
    printf ("Error al registrar senyal SIGUSR1\n");
    return  -1;
  }
```

Es crida a la funció signal, amb el tipus de senyal, i la funció que la tracta, en aquest cas, *tractament_senyal*, definida així:

```
void tractament_senyals (int s) {
.................
  printf (“rebuda senyal SIGUSR1\n”);
.............
}
```

La variable d'entrada *s* ens indica el tipus de senyal rebut, útil per a crear una sola funció que gestioni varis tipus de senyals.

Quan un procés vol enviar a un altre una senyal ha de cridar la funció:

```
int kill(pid_t pid, int sig);
```

Per la majoria de tipus de senyals, si un procés rep una senyal per la qual no ha definit una funció que la gestioni, es provocarà la terminació del procés.

Hem de considerar dos punts importants amb les senyals:

1. Al executar un programa, si nosaltres premem les tecles CTRL+C, enviarem al programa en execució la senyal SIGINT

2. Hi ha una sèrie de senyals que no poden ser interceptades per un procés, que són: SIGKILL, SIGSTOP. La primera, provocarà la terminació immediata d'un procés. La segona, provoca la seva aturada.

Una cosa més a considerar, que afecta al funcionament del client i el servidor. Quan un procés fill acaba la seva execució, aquest espera que el pare reculli el seu valor de terminació. Mentre això no passa, el procés fill resta en un estat anomenat *zombie*. Per tal d'assabentar-se el pare que el fill ha terminat, aquest primer rebrà una senyal de tipus SIGCHLD (terminació de procés fill), el qual ha de tractar adequadament. Aquesta és una senyal que no provoca la terminació del procés si no es gestiona.

3.6 Estructura d'un procés

El sistema operatiu, per a cada procés que s'executa al sistema, manté una estructura associada al mateix (comunment, el PCB, *Process Control Block*), anomenada *task_struct*[1]. Aquesta estructura emmagatzema dades com: la prioritat

1 Veure el fitxer /usr/include/linux/sched.h

del procés, el propietari, el temps que porta executant-se, el PID (codi identificatiu del procés), la senyal rebuda (si n'hi ha)... Però un camp molt important és el que indica si el procés està en execució o no, el camp *state*. Aquest pot valdre: TASK_RUNNING (indica que el procés està en execució), TASK_INTERRUPTIBLE (procés adormit, però que pot despertar-se mitjançant una senyal), TASK_UNINTERRUPTIBLE (procés adormit, i no pot despertar-se mitjançant una senyal).

Modificant adequadament aquest camp podem dormir o despertar un procés; així veiem la funció *dormir_proces*, present al mòdul CDX*:*

```
//Dormir proces en curs
void dormir_proces(void)
{
#ifdef DEBUG
  printk("CDX: proces %d dormint\n",current->pid);
#endif
  current->state=TASK_INTERRUPTIBLE;
  schedule();
}
```

current fa referència al PCB del procés en execució. La funció del kernel *schedule*() s'utilitza per cridar al planificador de processos i que esculli el següent procés a executar.

Per tal de despertar un procés, hem de fer:

```
wake_up_process(peticio->proces);
schedule();
```

On *peticio* apunta a la petició que ha fet el procés en qüestió.

3.7 El descriptor de fitxer (file)

Com hem vingut comentant, totes les ordres que rep el mòdul porten associat el descriptor de fitxer. El S.O. s'encarrega de crear un de nou per cada fitxer (i per tant, dispositiu) que s'obre. La seva estructura és la següent (definida dins de /usr/include/linux/fs.h):

```
struct file {
  struct list_head          f_list;
  struct dentry             *f_dentry;
  struct vfsmount           *f_vfsmnt;
  struct file_operations    *f_op;
```

```
  atomic_t              f_count;
  unsigned int          f_flags;
  mode_t                f_mode;
  loff_t                f_pos;
  unsigned long         f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct fown_struct    f_owner;
  unsigned int          f_uid, f_gid;
  int                   f_error;

  unsigned long         f_version;

  /* needed for tty driver, and maybe others */
  void                  *private_data;
};
```

Veiem alguns camps que es fan servir dins el CDX:

· El camp *flags* indica de com es comporta el fitxer a l'obrir-lo: si s'ha de crear un de nou, si s'ha d'afegir dades, si s'ha d'obrir normalment...

· El camp *mode* indica els permisos a emprar quan s'obre un fitxer nou: si pot ser llegit, escrit, etc.

Aquests dos camps són enviats també des del mòdul cap el servidor, per tal d'obrir el dispositiu adequadament.

3.8 Assignació de memòria dins el mòdul

Dins el mòdul necessitem assignar memòria per diverses causes: per mantenir la llista de peticions, per disposar de buffers d'intercanvi de memòria entre el procés i el client... Llavors, dins del sistema operatiu disposem de les següents funcions (definides dins "/usr/include/linux/malloc.h"):

```
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);
```

La funció d'assignar memòria, kmalloc, demana 2 paràmetres: el tamany a assignar, i un valor estàtic que indica la prioritat i la manera com ha de reaccionar el procés si es necessita memòria virtual (de disc), i per tant ha de posar-se en estat d'espera (dormint):

GFP_KERNEL: Assignació normal de memòria. El procés pot posar-se en estat d'espera

GFP_BUFFER: Semblant al primer, diferint en el mode en que es demana memòria virtual

GFP_ATOMIC: El procés mai es posarà en estat d'espera; utilitzat per controladors d'interrupcions

GFP_USER: Semblant a com es demana memòria des d'un procés d'usuari

GFP_HIGHUSER,__GFP_HIGHMEM: Igual a l'anterior, però es demana memòria de la part alta (high mem). Aquest tipus de memòria només és present en alguns tipus d'arquitectures (Intel).

__GFP_DMA: Es demana memòria utilitzable en transferències de dades mitjançant DMA.

En el mòdul CDX es demana memòria del primer tipus (GFP_KERNEL). Hem de tenir en compte que mai es poden demanar més de 128kb cada vegada.

Tenint això, ara veiem per a quins casos necessitem assignar memòria:

· Llista de peticions: Segons recordem, el mòdul guarda una llista de peticions pendents de ser executades. S'ha definit que es poden mantenir com a màxim 512 peticions. Una petició té la següent estructura:

```
struct e_peticio {
  size_t              length; //longitut de la peticio
  unsigned char       ordre; //OPEN, READ, WRITE...

  struct file         *file;
  unsigned char       buffer_transmissio[20];

  struct task_struct *proces; //procés que havia fet la petició
  unsigned char       estat; //estat de la peticio (lliure=nul.la o pendent)

  //Longitut de les dades a transmetre en la trama
  //(excepte les dades en les ordres write)
  unsigned char       longitut_buffer;

  int                 cmd; //comanda cridada en una ordre ioctl pendent de retorn
  long long           valor_retorn; //valor de retorn pel proces que ha fet la peticio

  struct e_peticio    *seguent; //seguent peticio a la llista. NULL si no hi ha
                                //seguent

};
```

A l'hora de reservar espai per la llista podríem fer-ho de dues maneres:

1. Demanar memòria per cada petició nova que s'afegeix. Això podria resultar poc eficient, ja que faria molt lent l'execució del mòdul.

2. Demanar memòria inicialment per totes les 512 peticions. Això, que podria resultar eficient, presenta un inconvenient, i és el fet de que podríem sobrepassar el límit dels 128Kb en assignació. Tot i que tal i com està implementat el mòdul no sobrepassaríem aquest límit, hem de pensar en futures millores, i per tant, en l'ampliació de l'estructura de petició.

Llavors, s'ha optat per una implementació diferent d'aquestes dues: quan es necessita assignar memòria per una petició, es demana un número fixe de peticions (50 en aquest cas), i es va lliurant memòria dins d'aquest bloc. Quan ja hem demanat 50 en aquest bloc, es torna a demanar un de nou. Amb això, ens assegurem que mai assignem un bloc de memòria major que el límit (actualment, cada bloc de memòria ocupa 2.8 Kb). Els blocs de memòria que es van assignant, es guarden dins un vector d'estructura e_bloc_memoria:

```
struct e_cdx_bloc_memoria {
  unsigned char *direccio;          //adreça del bloc assignat
  int peticions_assignades;         //número de peticions assignades a aquest bloc
};
```

L'assignació de blocs de memòria es realitza amb la funció:

int assignar_bloc_memoria (void)

No es pot alliberar un bloc de memòria, si no que s'han d'alliberar tots de cop (quan es treu el mòdul de la memòria)

L'assignació de peticions noves es realitza amb la funció:
char *assignar_peticio (void)

Aquesta s'encarrega de veure si encara no hi ha 50 peticions dins un bloc de memòria, i retornant l'adreça de memòria corresponent. Si s'ha arribat a 50, es crida a *assignar_bloc_memoria* per tal de demanar-ne un de nou.

Quan volem alliberar una petició, es marca el camp *estat* dins la petició per indicar que està lliure. Així, sempre que es vol demanar una petició, es cridarà a la funció:
struct e_peticio *afegir_peticio (void)

La qual s'encarrega de llegir una per una, llegint el camp *estat*. Si no es troba cap petició lliure, es crida a *assignar_peticio*

· Buffers temporals de dades entre el procés i el client: Aquesta és l'altra circumstància en la qual s'ha de demanar memòria. Recordem que per traspassar les dades entre client i procés (ja sigui per fer lectures, o com a resultat d'escriptures) hem de tenir un buffer temporal, en el qual es copiïn les dades d'una part i es traspassin a l'altra. En aquest cas, s'ha optat per una assignació directa de memòria, fent us de *kmalloc*, però restringint el tamany del bloc de dades (actualment, és de 16Kb). Per tal d'això, es manté una llista amb tots els clients fills i el buffer de dades usat :

```
struct e_fills_client {
  pid_t pid;                  //pid del proces fill
  struct file *file;          //file que gestiona el fill
  unsigned char *buffer_intercanvi;       //buffer que emmagatzema les dades
                                          //de read, write, ioctl
};
```

Aquest buffer d'intercanvi s'assigna en el moment de registrar un nou client fill.

Ara podríem preguntar-nos: que passa si l'operació implica més de 16Kb? No passaria res, donat que el mòdul s'encarrega de dividir cada operació de lectura o escriptura en vàries que utilitzin com a màxim 16Kb de memòria.

3.9 Assignació de memòria dins el client/servidor

Dins el client i el servidor també es necessita assignar memòria, per tal de traspassar blocs de dades en operacions de lectura o escriptura, i per llegir trames CDX. En aquest darrer cas, la funció de lectura de trames (*llegir_trama)* s'encarrega d'assignar un bloc suficientment gran per llegir la trama.

En el cas del servidor, que hagi de llegir un bloc de dades d'un dispositiu, s'utilitza una funció emprada dins *llegir_trama*, que és *assigna_memoria,* la qual mira si el bloc que es disposava abans era suficientment gran per ser ocupat pel nou bloc. Si no ho és, se'n demana un de nou, mitjançant la funció *malloc* de la llibreria del S.O. Això es fa així per tal de no estar assignant i alliberant contínuament blocs de memòria, ja que es faria molt lent.

3.10 Comunicació mitjançant ordres CDX

Com ja hem comentat al capítol anterior, la comunicació entre el client i el mòdul s'estableix mitjançant senyals i ordres ioctl. Les senyals són enviades des del mòdul al client per avisar de que hi ha pendent una ordre de ser executada. Les ordres ioctl són enviades des del client cap el mòdul, per recollir paràmetres de les ordres a efectuar (ioctl de lectura) o per retornar el resultat de les operacions (ioctl d'escriptura).

Aquests paràmetres s'envien amb una estructura determinada, que són les anomenades *ordres CDX.*

Cada vegada que s'ha d'efectuar una ordre, des del mòdul es guarda en un buffer de dades contingut dins l'estructura de peticions, anomenat buffer_transmissio[1], els paràmetres de la ordre a efectuar (ordre CDX). Llavors, s'envia la senyal al client de que hi ha una petició pendent, i quan el client la rep, llavors crida al mòdul una ordre *ioctl* adequada, que li retornarà les dades de la ordre CDX.

Totes les ordres CDX que pot rebre un client fill són sol·licitades mitjançant la senyal *CDX_SENYAL_PETICIO_ALTRES*, i porten, en el seu primer byte, el número d'ordre a efectuar (excepte l'ordre *close*, que és sol·licitada mitjançant una senyal pròpia, *CDX_SENYAL_PETICIO_CLOSE,* i que no té paràmetres)

La única senyal que pot rebre el client pare per part del mòdul és la d'obertura de dispositiu (*CDX_SENYAL_PETICIO_OPEN*), i per tant, no té número d'ordre donat que és única

Hem de fer una consideració especial amb les ordres d'escriptura de dades, tals com *write* i *ioctl* (d'escriptura). Aquestes ordres, el client primer recull els paràmetres, tals com longitud a escriure. Llavors, el client, ha d'assignar memòria suficient per poder recollir aquestes dades del mòdul. Quan la té assignada, ha de recollir les dades mitjançant una altre ordre ioctl, i el mòdul les hi ha de traspassar.

Aquí trobem un problema greu: el fet de que aquestes dades provenen del procés que ha executat la ordre, i el mòdul no les pot transferir directament del procés al client, donat que estan en un espai d'adreces diferents. Per poder solucionar aquest fet, el mòdul, abans d'avisar el client, ha copiat les dades de l'escriptura en el seu buffer temporal del client (buffer_intercanvi dins de e_fills_client). Recordem que aquest buffer té un tamany màxim, i si l'operació implica més dades de les permeses, el mòdul s'encarregarà de dividir-la en vàries.

1 Veure “3.8 Assignació de memòria dins del mòdul”

Llavors, quan vol recollir les dades, les té disponibles en el mòdul, en el seu espai de dades, i s'han d'enviar al client. Hem de fer constar aquí que, totes les operacions que implica còpia de dades entre el mòdul i el client, i entre el mòdul i un procés, relacionen dos espais d'adreces de memòria: el de procés i el de sistema. Per tal de usar l'espai de dades de procés dins el mòdul no es pot fer directament, fent us de punters, sinó que s'han d'usar funcions especials de transferència de dades, presents al sistema operatiu. Aquestes funcions són: *copy_to_user i copy_from_user,* implementades des del CDX com:

```
//Pasar dades de l'espai de memoria del sistema a l'usuari
void escriure_a_usuari
(char *punter,unsigned long arg,int longitut)
{
  copy_to_user((void *)arg,(void *)punter,longitut);
}

//Pasar dades de l'espai de memoria de l'usuari al sistema
void llegir_de_usuari
(char *punter,unsigned long arg,int longitut)
{
  copy_from_user((void *)punter,(void *)arg,longitut);
}
```

Així, quan el mòdul vol traspassar les dades al client, ha de fer us de *escriure_a_usuari*.

Quan el client ja té el resultat de la operació, ha de cridar el mòdul, mitjançant una ordre ioctl amb l'estructura de la ordre CDX adequada. Llavors, el mòdul recull el resultat (i si és una ordre de lectura, les dades llegides) fent us de la funció *llegir_de_usuari*. En aquest darrer cas, que tractem una ordre de lectura, el mòdul ha de fer l'operació inversa a quan es té una escriptura: en lloc de copiar al buffer temporal les dades que li envia el procés i passar-les al client, ha de copiar les dades del client a un buffer temporal.

Quan ja ha efectuat la lectura del resultat (incloent els paràmetres i les dades de lectures si n'hi ha) avisa el procés adormit que esperava aquesta ordre, el qual, quan es desperti, tindrà les dades en el buffer temporal, i podrà copiar-les en l'espai d'adreces adequat (el del procés que havia efectuat l'ordre).

Veiem a continuació un exemple d'ordre CDX, corresponent a l'assignació d'un client fill per part del pare:

Comanda ioctl: CDX_IOCTL_ASSIGNAR_FILL

Paràmetres:

32 bits: Valor del *file* que gestiona el fill

32 bits: PID del client fill

3.11 Comunicació mitjançant trames CDX

Les trames CDX són el mètode de comunicació entre el client i el servidor. Quan el client vol avisar el servidor per tal de que executi una ordre, li envia una trama amb els paràmetres de la ordre. El servidor, per retornar el resultat, li envia també una trama CDX.

El format d'una trama CDX és el següent:

Posició	*Longitud*	*Descripció*
0	32 bits	Longitud del que ve a continuació
4	8 bits	Tipus de trama, indicant la ordre a realitzar o el resultat d'una d'elles
5	N	Dades característiques de cada trama

Aquest és el format genèric, tant si les ordres s'envien del client al servidor com del servidor cap el client.

Donat que en la posició 0 sempre trobem la longitud de la restant, per tal de llegir i enviar una trama s'utilitzen les funcions següents, definides a cdx_util.c:

```
int llegir_trama(int sock,char **buffer,int *longitut_buffer)
{
  int longitut;

  longitut=llegir_longitut(sock);
  if ( (assigna_memoria(buffer,longitut_buffer,longitut)) <0) return -1;

  if ((llegir_socket(sock,*buffer,longitut)) <0) return -1;

  return longitut;
}
```

La funció *llegir_longitut* s'encarrega de llegir els primers 4 bytes de la trama CDX. La funció *assigna_memoria* s'encarrega d'ampliar (si fa falta) el tamany de la memòria que ja es tenia assignada, per poder albergar tota la trama CDX. Això la funció *llegir_trama* ho sap amb el tercer paràmetre que se li passa, que indica el

tamany del buffer anterior. Si el tamany no ha variat (tenim suficient memòria per llegir la trama) *longitut_buffer* i *buffer* no canvien. Si el tamany ha variat (s'ha hagut de demanar més memòria) *longitut_buffer* i *buffer* canvien, indicant el nou tamany del buffer i la seva adreça, respectivament.

La funció d'escriure una trama CDX és:

```
int enviar_trama(int sock,char *buffer,int longitut)
{

  int n;

  n=longitut;

  if ( (write(sock,&n,4)) <0 ) return -1;
  if ( (write(sock,buffer,longitut)) <0 ) return -1;

  return 0;
}
```

És responsabilitat del procés que la crida de tenir, en el primer byte de l'adreça buffer, el número de la trama CDX.

Hem de fer constar que totes les codificacions de números en la trama CDX es fa seguint el format Little Endian, de Intel, o sigui, el byte menys significatiu primer. Això s'hauria de tenir en compte en una possible implementació del CDX per a plataformes Motorola, en les quals la codificació de números és Big Endian.

Veiem ara un exemple d'una trama CDX, corresponent al retorn de les dades llegides en una ordre READ:

Trama CDX: RETORNA_READ
Paràmetres:
- 8 bits: Ordre CDX (RETORNA_READ)
- 32 bits: Codi de retorn pel procés que l'ha executat
- 32 bits: Codi d'error pel client
- N Bytes: Dades llegides

En la trama CDX es troba implícita els 4 primers bytes que indiquen la longitud de la mateixa.

3.12 Tractament de la ordre ioctl

Aquí dedicarem tot un apartat al processament de la ordre ioctl, no l'enviada des del client cap el mòdul, i que forma part de la comunicació amb ordres CDX, sinó de les que genera un procés cap un dispositiu CDX.

Com ja vam veure abans, la dificultat aquí sorgia en esbrinar el número de paràmetres que usava, el tamany de les dades implicades i la direcció de la transferència (si és de lectura/escriptura). Vam veure que tot això es codificava dins el número de la ordre ioctl (en poques operacions, desafortunadament). Llavors, el S.O. disposa d'una sèrie de macros per extreure'n les parts[1]:

· _IOC_DIR(nr): Ens retorna la direcció de l'operació. Si és de lectura, retornarà _IOC_READ, si és d'escriptura, _IOC_WRITE, si és de lectura i escriptura, _IOC_READ | _IOC_WRITE, i si no implica cap transferència, _IOC_NONE.

· _IOC_TYPE(nr): Ens retorna el tipus de comanda, el número identificatiu d'aquest dispositiu.

· _IOC_NR(nr): Ens retorna el número de la comanda a efectuar.

· _IOC_SIZE(nr): Ens retorna el tamany de les dades a transferir.

Llavors, el tractament que se li dóna a aquesta ordre, tant en les trames CDX com en les ordres CDX és el següent:

· Llegim la direcció de la transferència, fent us de _IOC_DIR.

· Si ens diu que és una operació de lectura o escriptura, o mixta, es tracta adequadament, fent transferències de dades de tamany _IOC_SIZE.

. Si ens diu que l'operació no té transferència de dades, s'assumeix que la ordre ioctl de la llibreria s'ha cridat amb un paràmetre, passant el valor d'aquest paràmetre i indicant que el tamany és 32 bits (un enter).

Això dóna bons resultats en les comandes ioctl que fan servir la codificació "estandarditzada", però no funciona bé en les que no la segueixen.

1 Veure fitxer /usr/include/asm/ioctl.h

4. Proves realitzades

Les proves efectuades al CDX han consistit tant en l'execució d'ordres del sistema operatiu com en els fitxers de proves. També, abans d'aconseguir una plena funcionalitat, s'han efectuat proves a totes les funcions presents tant al mòdul, al client o al servidor.

En els exemples aquí mencionats, se suposa que tenim el fitxer de configuració amb el contingut adequat, i les entrades de dispositiu corresponent, per tal de fer ús dels dispositius CDX que es mencionen.

Veiem primer l'execució d'ordres al S.O:

Una prova molt senzilla consisteix a enviar un text a la impressora, per tal d'això executem la ordre:

echo Això es una prova amb el CDX > /dev/cdx/localhost/lp0

Veiem que immediatament surt imprés el text per la impressora.

Una altra prova consisteix a llegir un bloc de dades de la targeta de so; per tal de fer això, seleccionem la font de lectura de dades de la targeta de so (fent us d'un programa mesclador, com aumix) i establim el micròfon. Llavors, fem una ordre:

cat /dev/cdx/localhost/dsp

I veurem que, mentre parlem, els caràcters que surten per pantalla van variant.

D'aquest tipus d'ordres es poden provar amb la majoria de dispositius, enviant o rebent dades entre dispositius.

Un aspecte molt interessant és que la majoria de programes que utilitzen dispositius, es poden configurar per canviar la ruta al dispositiu emprat. Per exemple, a un programa reproductor de fitxers de música, li podem dir, amb un paràmetre, el dispositiu a utilitzar. Llavors, li especifiquem el nostre dispositiu CDX que controla la targeta de so, i ja la podrem fer servir, reproduint la musica en la targeta de so d'un altre ordinador.

Per exemple, pel reproductor de fitxers .MID, timidity:

timidity -B 200 fitxer.mid -o /dev/cdx/localhost/dsp -s 8000 -Od8M -E WPV -f

I pel reproductor de fitxers .MP3:

mpg123 -s -m --8bit -r 8000 fitxer.mp3>/dev/cdx/localhost/dsp

La resta de paràmetres s'utilitzen per disminuir l'ample de banda.

4.1 Fitxers de prova

Veiem ara els fitxers de prova executats:

prova1.c:

Aquesta és una prova molt senzilla, usant la targeta de so. La prova primer obre el dispositiu, llegeix un bloc de dades, executa una ordre IOCTL per tal de variar la freqüència, i després tanca el dispositiu.

prova2.c:

Aquest fitxer s'encarrega de llegir dades de la targeta de so (usant com a font de dades un CD d'àudio) i les envia novament a la targeta de so. Utilitza ordres ioctl per canviar la freqüència i els bits de mostreig, i els canals (estèreo). També usa un buffer de lectura d'un segon.

prova3.c:

Aquesta prova s'encarrega d'enviar contínuament dades d'un dispositiu a un altre, fent us d'un buffer. Tant els dispositius com el tamany del buffer l'hem d'indicar nosaltres, per exemple, executant:

./prova3 /dev/cdx/Apolo/dsp /dev/dsp 32768

prova4.c:

Aquest programa s'encarrega de fer una còpia d'una meitat de la pantalla en l'altra, fent us de la funció de posicionament (llseek). Cal tenir configurat el framebuffer de Linux per tal d'executar-la.

prova5.c

Aquesta prova, semblant a l'anterior, s'encarrega d'intercanviar aleatòriament i de manera contínua, línies de la pantalla, fent us també de la funció llseek.

prova6.c

Aquesta prova s'encarrega d'assolir el màxim de peticions simultànies, provocant un error. Per tal de fer això, obre un dispositiu, i va creant processos fills que llegeixen del mateix. Arriba a crear fins al màxim de peticions +3, per veure el missatge d'error. En aquest cas, veurem el missatge d'error quan s'assigni la petició número 513.

Hem d'executar, quan retorni l'error, l'ordre : “killall prova6”, per tal d'eliminar tots els processos fills que es queden executant.

Podrem veure el missatge d'error generat en el mòdul examinant el registre d'events, que serà una cosa semblant a:

Jan 27 05:57:25 Cesar kernel: CDX: S'ha assignat el maxim de peticions

4.2 Connexió del CDX per Internet

Donat que el CDX fa us de la llibreria sockets de Linux, es pot comunicar també mitjançant Internet. Donat que la majoria de nosaltres ens connectem mitjançant mòdem, no tindrem adreça fixa d'Internet, amb lo qual, cada vegada que ens connectem, la IP variarà.

Per tal de conèixer-la, podem fer us de la ordre *route* del sistema operatiu. Llavors, tenint la IP assignada a aquest ordinador, modifiquem el fitxer de configuració CDX, i en el nom del dispositiu remot li donem l'adreça IP de l'ordinador remot (aquell en que s'executarà el servidor).

Tenint tot això configurat, s'han pogut provar tots els fitxers de proves, encara que, lògicament, la velocitat d'execució ha estat molt menor, degut a l'amplada de banda del mòdem (màxim 56Kbps).

4.3 Audioconferència mitjançant el CDX

En aquest apartat, es detallarà com és de fàcil poder realitzar una conferència de veu a través d'una xarxa, fent us exclusivament del CDX. Suposem que tenim dos ordinadors connectats en xarxa, preferentment una LAN, anomenats Apolo i Neptu. Els dos disposen de targeta de so, amb altaveus i micròfon. En cadascun, hem d'executar el CDX (mòdul, client i servidor), tenint dins de la configuració el dispositiu de la targeta de so (/dev/dsp), o sigui, desde Apolo:

1 Neptu /dev/dsp

I desde Neptu:
1 Apolo /dev/dsp

Llavors, per establir la conferència, hem d'enviar a cada un dels dos ordinadors, el so llegit des del micròfon de l'altre.

Primer, establim la font de gravació com el micròfon, fent us d'un programa mesclador, tal com *aumix*, o *xmixer*.

Llavors, executem, des de Apolo:

cat /dev/cdx/Neptu/dsp > /dev/dsp

Amb lo qual, escoltarem el so enviat des de Neptu.

I des de Neptu executem la ordre complementària:
cat /dev/cdx/Apolo/dsp > /dev/dsp

Amb lo qual, haurem establert una audioconferència sense usar cap altre programa més que el CDX (recordem que la ordre cat només llegeix un fitxer/dispositiu, i el símbol > ens redirigeix la sortida cap a un altre fitxer)[1].

Hem de fer constar aquí que també es pot realitzar entre ordinadors connectats a Internet amb mòdem, però degut a la baixa velocitat de connexió (56 Kbps) el so es va retardant, i cada vegada arriba més desfasat en el temps.

Hem de fer un aclariment, i es que aquí hem fet la audioconferència mitjançant, des de cada ordinador, la lectura del micròfon remot i l'enviament a la targeta de so local. Igualment, es podria fer llegint el so del micròfon local i enviant-lo als altaveus remots, mitjançant:

1 Fem notar aquí que, quan s'obre el dispositiu de so, s'estableix, per lectura i gravació, una freqüència de 8000Hz, en mono, i 8 bits

```
cat /dev/dsp > /dev/cdx/Neptu/dsp
i
cat /dev/dsp > /dev/cdx/Apolo/dsp
```

5. Conclusions

En aquest projecte s'ha intentat facilitar l'ús de dispositius presents en una xarxa. Per tal d'això, s'ha hagut de tractar temes molt variats: connexió en xarxa, control de processos del sistema, us de dispositius, assignació i espais de memòria, gestió d'errors i depuració, mètodes de sincronització i comunicació de programes...

Els objectius marcats per aquest TFC s'han aconseguit plenament, aconseguint tot un programari funcional per a l'ús de dispositius en xarxa.

Ara bé, té una sèrie de limitacions que podrien implementar-se per tal de fer el programari més complet:

· El tipus de dispositius tractats només són els de caràcter; s'hauria d'estendre també als de bloc, i podríem, per exemple, fer us d'una gravadora de CD-ROM present en un ordinador remot.

· El tractament de les ordres ioctl: Per poder integrar completament tots els dispositius, caldria tenir tot una taula sencera, amb totes les comandes ioctl i els dispositius del sistema contenint el tipus de paràmetres que usen. Això, però, és un treball llarg i s'hauria de mantenir contínuament.

· Introduir un sistema d'autorització de dispositius per part del servidor, amb els dispositius que exporta, que, segons l'ordinador del que vinguin les peticions i el dispositiu a tractar, autoritzar-les o no. En part, en el fitxer “cdx.h” es té la definició del fitxer de configuració a tractar (CDX_EXPORTA)

· Implementar les ordres de dispositius no presents al CDX: poll, mmap, etc.

Veiem també que el rendiment del CDX, quan s'executen moltes ordres seguides de petit tamany, disminueix enormement. Això ve donat pel fet de que si executem moltes ordres seguides, amb trames CDX de petit tamany, estem usant molta més amplada de banda de xarxa que el que ocupen les trames (recordem que hem d'enviar les capçaleres TCP, IP, i la del nivell de xarxa). Aquest fet es pot veure executant programes reproductors de música, donat que envien contínuament ordres ioctl de petit tamany. En aquest cas, la solució passa per baixar la freqüència i els bits de mostreig.

Bibliografia

Linux Device Drivers, 2nd Edition
© Juny 2001 Alessandro Rubini & Jonathan Corbet

Xarxes de Computadors II
Setembre 1999
© Universitat Oberta de Catalunya

Linux Kernel Hackers' Guide
© 1996,1997 Michael K. Johnson, johnsonm@redhat.com
http://www.redhat.com:8080/HyperNews/get/khg.html

Linux Kernel Module Programming Guide
© 1999 Ori Pomerantz
http://www.linuxdoc.org

The Linux Programmer's Guide
© 1995 Sven Goldt , Sven van der Meer, Scott Burkett, Matt Welsh
http://www.linuxdoc.org

The Linux Kernel
© 1996-1999 David A Rusling
http://www.linuxdoc.org

Linux Kernel release 2.4.9
© 2001 Linus Torvalds

Annex A. Ordres CDX

En aquest annex es descriuen totes les ordres CDX, indicant el seu format. Sempre són ordres ioctl cridades des del client (tant el pare com el fill) cap el mòdul. El nom de la ordre és el número de la comanda ioctl a enviar, definits a "cdx.h"

En tots els casos, s'especifica si els paràmetres són d'entrada (els dóna el client cap el mòdul) o de sortida (els retorna el mòdul cap el client). Normalment, totes les ordres començades per "CDX_IOCTL_LLEGIR_" tenen paràmetres de sortida, i les que comencen per "CDX_IOCTL_RETORNA_" tenen paràmetres d'entrada. El valor de retorn que genera la funció ioctl, serà negatiu si hi ha hagut error, i 0 o positiu si no hi ha error.

Només les 3 primeres ordres CDX (registrar client, llegir open i assignar fill) són efectuades pel client pare. Totes les altres les crida el client fill.

Les ordres write i ioctl, que impliquen llegir dades del mòdul, primer faran una crida a la ioctl CDX_IOCTL_LLEGIR_ALTRES, i després a una altra específica de la seva funció.

CDX_IOCTL_REGISTRAR_CLIENT	
Registra el client pare cap el mòdul.	
Paràmetre (entrada)	Descripció
1	N bytes: Cadena de registre (definida dins cd.h->CDX_IDENTIFICACIO), acabada amb un byte amb valor 0

CDX_IOCTL_LLEGIR_OPEN	
Dóna al client pare els paràmetres corresponents a la primera petició d'obertura pendent.	
Paràmetre (sortida)	Descripció
1	32 bits: *file*
2	8 bits: *minor*
3	32 bits: *flags*
4	32 bits: *mode*

CDX_IOCTL_ASSIGNAR_FILL	
Registra un client fill cap el mòdul.	
Paràmetre (entrada)	Descripció
1	32 bits: *file* que gestiona el client fill
2	32 bits: PID del client fill

CDX_IOCTL_RETORNA_OPEN	
Retorna el valor de l'operació d'obertura de dispositiu.	
Paràmetre (entrada)	Descripció
1	32 bits: Valor de retorn pel procés que ha efectuat la ordre

CDX_IOCTL_RETORNA_CLOSE	
Retorna el valor de l'operació de tancament de dispositiu.	
Paràmetre (entrada)	Descripció
1	32 bits: Valor de retorn pel procés

CDX_IOCTL_LLEGIR_ALTRES	
Dona al client els paràmetres de l'operació a efectuar, ja sigui: read, write, ioctl, llseek.	
Paràmetre (sortida)	Descripció
1	8 bits: Operació a efectuar: CDX_ORDRE_READ / CDX_ORDRE_WRITE / CDX_ORDRE_IOCTL / CDX_ORDRE_LLSEEK
2	Paràmetres variables, segons la ordre a efectuar: CDX_ORDRE_READ-> 32 bits: Longitud a llegir CDX_ORDRE_WRITE-> 32 bits: Longitud a escriure CDX_ORDRE_IOCTL-> 32 bits: Comanda ioctl CDX_ORDRE_LLSEEK-> 64 bits: *Offset*
3	Paràmetre usat només en la ordre llseek-> 32 bits: *whence*

CDX_IOCTL_RETORNA_READ	
Retorna les dades resultants d'una operació de lectura.	
Paràmetre (entrada)	Descripció
1	32 bits: Valor de retorn pel procés
2	32 bits: Adreça on estan les dades

CDX_IOCTL_LLEGIR_WRITE	
Dóna al client fill les dades a escriure al dispositiu. Prèviament s'ha efectuat l'ordre CDX_IOCTL_LLEGIR_ALTRES per rebre la longitud del bloc a escriure.	
Paràmetre (entrada)	Descripció
1	N bytes: Dades a escriure

CDX_IOCTL_RETORNA_WRITE	
Retorna el valor resultant d'una ordre d'escriptura.	
Paràmetre (entrada)	Descripció
1	32 bits: Valor de retorn pel procés

CDX_IOCTL_LLEGIR_IOCTL	
Dóna al client fill les dades a transferir en una ordre ioctl d'escriptura. Només s'ha de cridar quan la direcció sigui d'escriptura o mixta (i la longitud>0) o si no hi ha direcció de transferència (en aquest cas es rebran 4 bytes del paràmetre). Prèviament s'ha efectuat l'ordre CDX_IOCTL_LLEGIR_ALTRES per rebre la longitud del bloc a transferir.	
Paràmetre (entrada)	Descripció
1	N bytes: Dades a transferir

CDX_IOCTL_RETORNA_IOCTL	
Retorna les dades resultants d'una operació ioctl.	
Paràmetre (entrada)	Descripció
1	32 bits: Valor de retorn pel procés
2	32 bits: Adreça on estan les dades (si l'operació no implica lectura de dades, aquest camp es pot deixar amb valor desconegut)

CDX_IOCTL_RETORNA_LLSEEK	
Retorna el valor resultant d'una ordre de modificació de punter de lectura/escriptura.	
Paràmetre (entrada)	Descripció
1	64 bits: Valor de retorn pel procés

Annex B. Trames CDX

En aquest annex es descriuen totes les trames CDX disponibles, indicant el seu format. En les trames CDX, es troba implícit la longitud en els primers 4 bytes.

El primer byte (després dels 4 bytes de la longitud) ha de contenir el número corresponent al tipus de trama CDX (la ordre). Aquests valors estan definits dins de “cdx_xarxa.h”. A continuació d'aquests 5 bytes, venen els paràmetres específics de cada ordre.

Les ordres que tenen nom començat amb “RETORNA_” són les enviades des del servidor cap el client, com a resposta d'una ordre enviada en sentit contrari. En aquests tipus d'ordre, si es retorna un codi d'error, es fa sempre amb valor negatiu. Un valor 0 o positiu significa que no hi ha hagut error.

CDX_TRAMA_IDENTIFICACIO	
Trama enviada des del client al servidor per registrar-se.	
Paràmetre	Descripció
1	N bytes: Cadena de registre (definida dins cdx_xarxa.h->CDX_IDENTIFICACIO_XARXA), acabada amb un byte amb valor 0

CDX_TRAMA_RETORNA_IDENTIFICACIO	
Trama retornada pel servidor i diu si el client s'ha registrat correctament. Si hi ha error, tant el servidor fill com el client fill s'han de tancar.	
Paràmetre	Descripció
1	8 bits: Codi d'error. Si és 0, client fill no registrat. Si és diferent de 0, client fill registrat.

CDX_TRAMA_OPEN	
Ordre enviada per obrir un dispositiu a l'ordinador servidor	
Paràmetre	Descripció
1	N bytes: Ruta i nom del dispositiu a obrir, acabat amb un byte amb valor 0
2	32 bits: flags d'obertura de dispositiu
3	32 bits: mode d'obertura de dispositiu

CDX_TRAMA_RETORNA_OPEN	
Trama enviada des del servidor amb el retorn de l'ordre d'obertura.	
Paràmetre	Descripció
1	32 bits: Codi d'error pel procés
2	32 bits: Codi d'error pel client

CDX_TRAMA_CLOSE
Ordre enviada per tancar el dispositiu. No té paràmetres.

CDX_TRAMA_RETORNA_CLOSE	
Trama amb el retorn de l'ordre de tancament.	
Paràmetre	Descripció
1	32 bits: Codi d'error pel procés
2	32 bits: Codi d'error pel client

CDX_TRAMA_READ	
Ordre enviada per llegir dades des del dispositiu	
Paràmetre	Descripció
1	32 bits: Longitud a llegir

CDX_TRAMA_RETORNA_READ	
Trama enviada des del servidor amb el retorn i les dades llegides de l'ordre de lectura.	
Paràmetre	Descripció
1	32 bits: Codi d'error pel procés (si és >=0, indica bytes llegits)
2	32 bits: Codi d'error pel client
3	N bytes: Dades llegides (si hi ha un error del procés, aquest camp no hi és)

CDX_TRAMA_WRITE	
Ordre enviada per escriure dades al dispositiu	
Paràmetre	Descripció
1	32 bits: Longitud a escriure
2	N bytes: Dades a escriure

CDX_TRAMA_RETORNA_WRITE	
Trama enviada des del servidor amb el retorn de l'ordre d'escriptura.	
Paràmetre	Descripció
1	32 bits: Codi d'error pel procés (si és >=0, indica bytes escrits)
2	32 bits: Codi d'error pel client

CDX_TRAMA_IOCTL	
Ordre enviada amb els paràmetres i les dades corresponents a una ordre ioctl	
Paràmetre	Descripció
1	32 bits: Número de la comanda ioctl
2	N bytes: Dades a escriure. En una ordre d'escriptura, el tamany vindrà codificat dins el número de la comanda. Si no ho és, contindrà els 4 bytes corresponents al paràmetre passat a la ordre ioctl.

CDX_TRAMA_RETORNA_IOCTL	
Retorna al client les dades llegides (si n'hi ha) i el valor de retorn de la ordre ioctl.	
Paràmetre	Descripció
1	32 bits: Codi d'error pel procés
2	32 bits: Codi d'error pel client
3	N bytes: Dades llegides. Aquest camp només és present si l'ordre és de lectura o mixta.

CDX_TRAMA_LLSEEK	
Ordre enviada per posicionar el punter de lectura/escriptura	
Paràmetre	Descripció
1	64 bits: Desplaçament
2	32 bits: Paràmetre *whence* de llseek

CDX_TRAMA_RETORNA_LLSEEK	
Trama enviada des del servidor amb el retorn de l'ordre de posicionament.	
Paràmetre	Descripció
1	64 bits: Codi d'error pel procés
2	32 bits: Codi d'error pel client