Carmen Cristina Miranda de Lima

Um sistema de "mirroring" FTP e HTTP que optimiza recursos usando uma estratégia de avaliação retardada

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
1999

Carmen Cristina Miranda de Lima

Um sistema de "mirroring" FTP e HTTP que optimiza recursos usando uma estratégia de avaliação retardada

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Mestre em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
1999

Para os meus Pais e para o meu Marido.

Agradecimentos

Quero agradecer toda a orientação científica e pedagógica que me foi dada pelo Dr. Rogério Reis e pelo Professor Doutor Luís Damas, que me permitiu realizar este trabalho. Agradeço especialmente ao Dr. Rogério Reis todas as sugestões e comentários que fez à escrita desta dissertação.

Gostaria de agradecer igualmente ao Professor Doutor Fernando Silva por me ter revelado os segredos do Latex.

Obrigado a todos os meus colegas que me incentivaram e me prestaram o seu apoio em diversas alturas.

Desejo agradecer ao Centro de Informática da Universidade do Porto todas as condições e meios que me disponibilizou e que permitiram o desenvolvimento deste trabalho.

Gostaria de apresentar um agradecimento especial aos meus pais, por todo o apoio e carinho que sempre me dedicaram e por todos os esforços que fizeram para me proporcionar uma boa educação.

Por fim, quero agradecer carinhosamente ao meu marido Manuel pelo seu constante encorajamento, mesmo nas alturas em que eu desanimava e me tornava impossível de aturar.

Resumo

Um servidor ftp contém normalmente, para além dos ficheiros mantidos localmente, cópias de ficheiros distribuídos por outros servidores, evitando assim duplicação do gasto de largura de banda, resultante das diversas transferências dos mesmos ficheiros, efectuadas por cada utilizador que os pretenda obter. É necessário garantir que estas cópias estão actualizadas, recorrendo por exemplo a software de replicação (*mirroring*), que periodicamente verifica a existência de novos ficheiros ou de ficheiros alterados no servidor, procedendo nesse caso à sua cópia. A periodicidade com que são executadas essas cópias é determinada pelo administrador do sistema e definida por exemplo através do *cron*, não sendo assim imediata a sua actualização.

Este tipo de processo obriga à transferência de grande quantidade de informação, que nem sempre corresponde às necessidades dos utilizadores. Os objectivos deste tipo de servidor ficam assim gorados, uma vez que os utilizadores continuam a recorrer aos repositórios originais, ocasionando aumento desnecessário de tráfego e duplicação de informação.

Uma solução mais eficiente, é baseada num modelo de avaliação retardada (*lazy evaluation*), isto é, num modelo em que a transferência de ficheiros é feita na altura da sua solicitação se ainda não residir no servidor, de forma transparente para o utilizador. Este tem acesso a todos os ficheiros, sem se aperceber que alguns não existem no servidor, e são transferidos de forma automática e imediata quando solicitados.

Este tipo de servidor permite a parametrização da rede nacional ou da rede local, de forma a favorecer as ligações a servidores centrais, penalizando as ligações a outros servidores ftp, desencorajando a sua utilização.

Este processo pode ser estendido ao protocolo http, permitindo deste modo manter localmente cópias de páginas existentes noutros servidores.

Abstract

Ftp is one of the most used Internet services. Some of the ftp servers and the files they store are very popular which leads to both overloading of the servers and of the lines used to access them. In order to minimize this problem many local administrators maintain local *mirror* sites of the more common ftp sites.

Mirror sites are ftp servers that maintain an exact copy of the files from one or more ftp servers. This help reduce both long distance traffic and download time for local users. There are however some inconvenients with this approach. The copies are made on a periodical basis, and so they are not always in tune with the originals. There are however some problems with this strategy which may occur when the mirrored site changes frequently and when only some subtrees of the original site are accessed locally. By copying the entire sites frequently one risks generating the unnecessary long distance traffic which one was trying to avoid.

One more efficient solution is based on a lazy evaluation model. On that model the transfer of files is made on demand, if they don't already exist on the local server. Users can access all files without knowing that some of them do not exist on the server, but are automatically and concurrently transferred when they are requested.

This kind of server allows the national network parameterization, promoting connections to central servers and discouraging connections to other ftp servers.

This process can also be extended to the http protocol. We can keep in the local server, a copy of the pages maintained by other http servers.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

A Internet, que ainda há poucos anos tinha apenas algumas centenas de utilizadores normalmente ligados a instituições universitárias ou de investigação, vê o seu número de utilizadores aumentar diariamente. Deste modo, a rede deixou de "pertencer" a um número restrito de peritos em informática, abrangendo actualmente um vasto número de pessoas que a utilizam no trabalho ou para lazer, o que obrigou ao desenvolvimento de ferramentas que facilitem a sua utilização, tornando-a deste modo mais fácil de consultar.

Os utilizadores da Internet procuram normalmente uma só coisa: informação. A maior parte das instituições estatais e comerciais disponibilizam informação de uma forma atraente, permitindo uma consulta rápida e simples. Um dos aspectos que tem registado um maior desenvolvimento é o das compras através da rede. É possível deste modo efectuar transações sem ter de sair de casa, como por exemplo, comprar discos, livros ou até marcar férias.

Grande parte destas instituições colocam nos seus "sites" programas que podem ser transferidos para o computador pessoal de qualquer utilizador. O tipo de informação distribuído desta forma é muito vasto e compreende nomeadamente software, sistemas operativos, imagens ou jogos. Os sistemas operativos "freeware", como o Debian, que estão instalados em muitos servidores, são distribuídos desta forma. Qualquer pessoa que tenha acesso à Internet pode obte-los através da execução de programas clientes `ftp` ou através de *browsers* como o Netscape, que usam o protocolo HTTP. Quem é que nunca teve acesso deste modo a jogos de computador ?

Esta informação pública é mantida nos servidores "originais" e copiada para vários servidores espalhados por todo o mundo, permitindo assim um mais rápido acesso e evitando um excesso de pedidos às máquinas originais. É necessário garantir a coerência da informação assim disponibilizada, isto é, a informação contida nestes

servidores tem que ser igual à dos servidores originais. Com esta finalidade são executados programas que copiam a estrutura de directórios do servidor original e verificam periodicamente a existência de novos ficheiros, procedendo neste caso à sua transferência.
Estes programas obrigam à transferência de grandes quantidades de informação que podem nunca ser utilizadas. Os ficheiros copiados nem sempre correspondem aos pedidos dos utilizadores, que continuam deste modo a preferir transferi-los a partir dos repositórios originais, gorando deste modo um dos objectivos deste tipo de servidores.

Para evitar situações como esta, propomos um sistema que permita transferir para os servidores locais, apenas os ficheiros solicitados e não toda a estrutura de directórios, como acontece nos programas existentes, conforme já foi referido.
É assim mantida localmente informação sobre a estrutura dos directórios remotos, sendo os ficheiros transferidos na altura da sua solicitação, se ainda não existir uma cópia na máquina local. Os utilizadores deste sistema têm acesso a toda a informação, de forma transparente, como se ela existisse realmente no servidor.
Para garantir a coerência da informação local é necessário definir políticas de validade dos dados locais, quer estes digam respeito a directórios ou a ficheiros.

Procurar informação na Internet nem sempre é uma tarefa simples pois por vezes não existem dados suficientes acerca do local onde a podemos obter, apesar de existirem vários sistemas de procura de informação.
A rede de cálculo científico nacional – `RCCN` – contem vários "mirrors" oficiais. Muitos destes servidores mantêm a mesma informação. Como não existe uma lista que refira os servidores existentes e descreva os *mirrors* que cada um está a manter, torna-se difícil aos utilizadores saberem qual o servidor "mais próximo" onde podem obter os dados que pretendem.

Com o sistema implementado é possível construir um servidor central que permite organizar a informação contida nos vários servidores da RCCN. Deste modo garantimos que não há duplicação de dados e facilitamos a procura de informação.

1.2 Objectivos

Os programas de *mirroring* transferem para os servidores locais vastas quantidades de informação, tendo estes que possuir discos suficientemente grandes para alojar os ficheiros assim obtidos. Muitos destes ficheiros nunca são utilizados.
Os administradores do sistema consultam regularmente os *logs* do programa de *mirroring* e do próprio `ftp`, tentando deste modo obter a lista dos ficheiros mais solicitados. Com base nestes dados, podem reconfigurar o programa de *mirroring* para que não seja efectuada a cópia de ficheiros que nunca são acedidos, tentando deste modo diminuir o espaço de disco utilizado. Este processo obriga a uma intervenção constante do administrador, muitas vezes infrutífera, pois este não consegue ter uma noção exacta das necessidades dos seus utilizadores.

O sistema implementado – `rfs` (*reflected filesystem*) – permite manter servidores ftp que possuem cópias da informação contida noutros servidores, sem ter de duplicar todos os ficheiros no disco local. Deste modo evita-se o gasto desnecessário de espaço de disco, pois só são guardados os ficheiros que são solicitados por algum utilizador. Com este tipo de serviço diminui-se a largura de banda utilizada, pois não se transferem de uma só vez grandes blocos de informação e apenas se copiam os ficheiros necessários.

Este tipo de sistema é facilmente administrado. É necessário definir apenas a configuração inicial do programa, onde se indicam os servidores onde vamos obter os ficheiros que estamos a manter localmente e o directório local onde estes residem. Periodicamente podem ser executados programas que fornecem estatísticas sobre os pedidos solicitados, de forma a termos uma noção da utilização do servidor. Podem igualmente ser executados programas que removem da "cache" ficheiros que não são utilizados há muito tempo. Note-se que o programa tem um processo automático de proceder a esta "limpeza", sempre que não haja suficiente espaço livre de disco.

A implementação do programa é feita a nível da camada de ficheiros, utilizando o sistema de ficheiros `userfs`, que permite que um processo de um utilizador sem privilégios especiais seja visto como um sistema de ficheiros Linux.
Deste modo não é necessário alterar o código do programa servidor de ftp – `ftpd`. O programa funcionará com qualquer versão do *ftpd* e responderá a pedidos HTTP.
Foi usada para a sua implementação a linguagem de programação C, o que permitiu a utilização de bibliotecas que acompanham o *userfs* e definem a comunicação com o kernel.

Este sistema permite manter igualmente servidores de `http` que possuem cópias de páginas existentes noutros servidores. É assim possível manter numa só máquina *mirrors* de `ftp` e `http`, usando o mesmo processo.

1.3 Estrutura da Tese

Este texto está estruturado da seguinte forma:

O **Capítulo 2** apresenta a noção de sistema de ficheiros e o seu funcionamento, referindo o conceito de sistema virtual de ficheiros, superbloco, *inode* e ficheiro. São descritas as operações que permitem manipular cada uma das estruturas referidas. Considerou-se fundamental a apresentação destes conceitos pois o programa é implementado a nível da camada de ficheiros.

No **Capítulo 3** é apresentado o sistema de ficheiros `userfs` e as alterações efectuadas ao código para que este funcionasse em versões do kernel 2.2.

O **Capítulo 4** introduz o trabalho desenvolvido nesta dissertação. É referida a estrutura da "cache", o ficheiro de configuração e a a forma como os dados são representados internamente. São igualmente expostas as políticas adoptadas para gerir o espaço de disco e para validar a informação guardada na “cache”.

No **Capítulo 5** são abordados pormenores de implementação. É indicada a forma como se processa a comunicação com o kernel e com os servidores de ftp e são descritas as operações implementadas.

No **Capítulo 6** são indicados os programas desenvolvidos que permitem administrar o sistema implementado.

No **Capítulo 7** apresenta-se um estudo da performance do programa.

No **Capítulo 8** são apresentadas conclusões do trabalho efectuado e sugeridos trabalhos futuros.

Capítulo 2

Sistema de ficheiros

Um sistema de ficheiros no Unix, é uma estrutura hierárquica de ficheiros que residem normalmente num disco, podendo no entanto ser implementada noutro dispositivo, nomeadamente numa disquete, fita magnética ou memória RAM. Nalguns casos, o sistema de ficheiros pode ser implementado como um recurso remoto, acessível através da rede, como acontece no `nfs` (*network file system*). Os ficheiros que formam o sistema são de vários tipos, nomeadamente, ficheiros regulares, directórios, *devices*, *FIFOs*, *links* e *sockets*, permitindo assim que todos os recursos do sistema sejam acedidos através de ficheiros.

Um dos aspectos essenciais do Unix é a forma como usa ficheiros para representar todos os recursos. Uma impressora, um modem, ou qualquer *device* são vistos como ficheiros. Deste modo podemos executar comandos que manipulam ficheiros, para aceder a todos os recursos do sistema.
Por exemplo, o comando

% cat teste.c > /dev/ttyp7

envia para a janela associada ao *device* ttyp7 o conteúdo do ficheiro *teste.c*.

O sistema de ficheiros [TW97], além de guardar os dados contidos nos ficheiros que o constituem, mantém também a estrutura do próprio sistema. Cada ficheiro é representado por uma estrutura `inode`, contendo toda a informação acerca do ficheiro, excepto o seu nome. A informação referida inclui o tipo do ficheiro, as permissões de acesso, a identificação do dono, o seu tamanho, a data de acesso, e apontadores para os blocos de dados. O nome do ficheiro é guardado no directório. A estrutura do directório é formada pelos nomes dos ficheiros e pelos números dos respectivos *inodes*. São os directórios que permitem a estruturação hierárquica do sistema de ficheiros.

O sistema Linux permite a coexistência de vários tipos de sistemas de ficheiros, como por exemplo, *ext, ext2, minix, umsdos, msdos, vfat, proc*. Toda a informação necessária à gestão do sistema de ficheiros é guardada no `superbloco`.

Em Unix, os sistemas de ficheiros não são acedidos através de identificadores dos dispositivos que os contêm (como por exemplo, o seu nome), mas são combinados numa única estrutura hierárquica em árvore, que representa o sistema de ficheiros como uma entidade única. Cada sistema de ficheiros novo, é acrescentado a essa árvore à medida que é "montado". Esta "montagem" pode ser feita em qualquer directório, que passa a designar-se "ponto de montagem".

O vasto número de sistemas de ficheiros suportado pelo Linux, é sem dúvida uma das principais razões da sua crescente utilização. Isto só é possível graças a uma interface ao kernel unificada, o sistema virtual de ficheiros (`VFS`) [BBD+98]. Este fornece as chamadas ao sistema que permitem gerir ficheiros, mantém as estruturas internas e envia os pedidos aos sistemas de ficheiros apropriados.
O VFS pode ser visto como uma interface que liga o kernel do sistema operativo aos vários sistemas de ficheiros, conforme está ilustrado na figura 2.1.

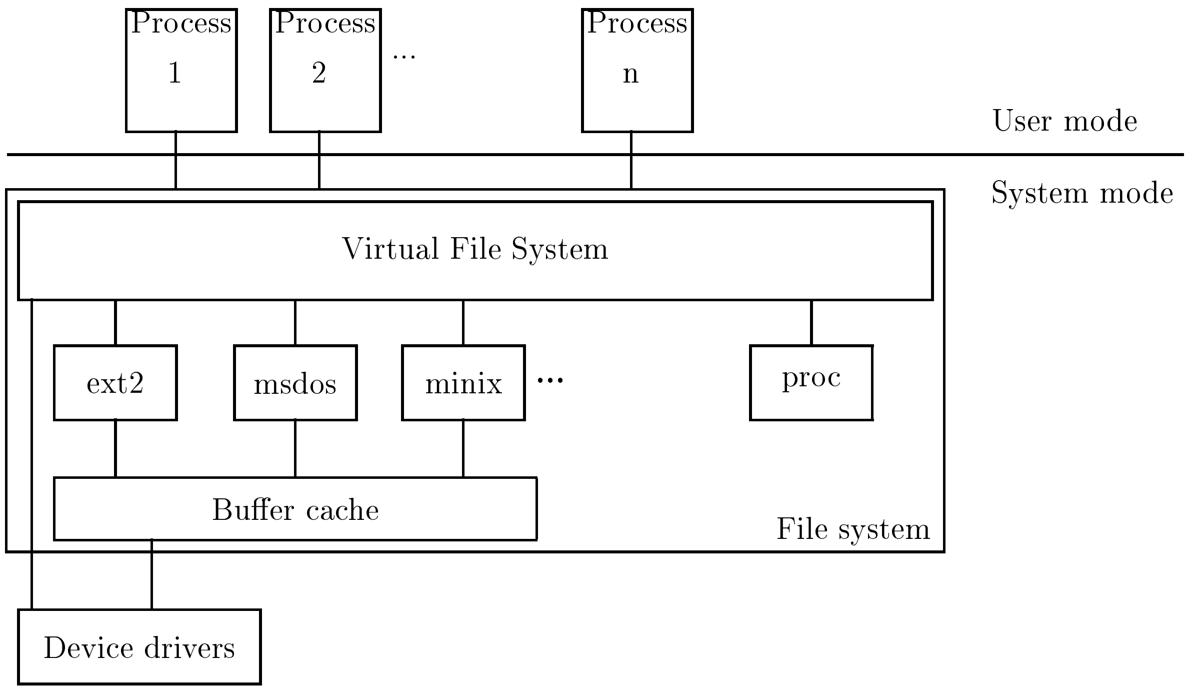

Figura 2.1: Camadas de um sistema de ficheiros

2.1 VFS

A representação dos dados num disco duro ou numa disquete, varia de caso para caso, dependendo do tipo de sistema de ficheiros existente. No entanto, existe uma camada inferior comum – o sistema virtual de ficheiros – que permite que os utilizadores vejam os dados sempre do mesmo modo e usem os mesmos comandos independentemente do

tipo de sistema de ficheiros em que residem.
Por exemplo, ao executar o comando

% cat filename

o VFS redirecciona o pedido de leitura do ficheiro para a função adequada, implementada pelo sistema de ficheiros onde reside o ficheiro *filename*.

As rotinas de iniciação de cada sistema de ficheiros, registam o respectivo sistema no sistema virtual de ficheiros, executando a função `register_filesystem` (implementada no ficheiro fs/super.c do kernel), que preenche a estrutura `file_system_type` (definida no ficheiro linux/fs.h) constituída pela seguinte informação:

- Rotina de leitura do superbloco.
- Nome do tipo de sistema de ficheiros.
- Indicação da necessidade de um *device*.
- Apontador para a estrutura seguinte.

Por exemplo, a execução de

```
struct file_system_type minix = {
       minix_read_super,
       ``minix'',
       1,
       NULL };

register_filesystem(&minix);
```

regista o sistema de ficheiros denominado "minix", cuja rotina de leitura do superbloco é *minix_read_super*. É necessário um *device* a partir do qual o sistema é montado.

Quando um sistema de ficheiros é montado, o VFS tem de ler o seu superbloco, de forma a obter apontadores para rotinas que efectuam determinadas acções. Para isso executa a respectiva rotina de leitura do superbloco. O VFS mantém uma lista dos sistemas de ficheiros montados e dos superblocos respectivos.

2.1.1 Montar um sistema de ficheiros

O comando `mount` permite montar um sistema de ficheiros, sendo necessário referir o seu nome, o *device* físico onde reside e o directório onde vai ser montado. Cabe ao VFS averiguar se o sistema de ficheiros indicado é suportado pelo kernel e se o directório onde vai ser montado existe. Assim, este verifica se na estrutura de dados `file_system_type` existe alguma referência a esse sistema de ficheiros, obtendo deste modo o apontador para a sua rotina de leitura do superbloco. Seguidamente, o VFS procura o *inode* do directório indicado, verifica se se trata efectivamente de um directório e se não existe nenhum sistema de ficheiros aí montado. Note-se que um directório não pode ser usado simultaneamente como "ponto de montagem" de vários sistemas de ficheiros.

Cada sistema de ficheiros montado é representado pela estrutura `super_block` (definida em linux/fs.h), que contém informação sobre todo o sistema a que concerne, nomeadamente

device Identificador do *device* que contém este sistema de ficheiros.

blocksize Tamanho dos blocos, em bytes.

lock Indica se o superbloco está a ser utilizado exclusivamente por um processo.

wait Apontador para a fila de espera dos processos que pretendem utilizar o superbloco, que está a ser acedido exclusivamente por outro processo.

dirty Indica que o superbloco foi alterado.

type Apontador para a respectiva estrutura de dados `file_system_type`.

op Apontador para a estrutura que refere as rotinas que operam no superbloco.

magic Identificador único do sistema de ficheiros.

time Data da última alteração.

covered Apontador para o *inode* onde o sistema de ficheiros está montado.

mounted Apontador para o *inode* da raiz do sistema de ficheiros.

spec Informação necessária ao sistema de ficheiros, que depende do seu tipo.

Para preencher esta estrutura de dados é executada a rotina de leitura do superbloco do respectivo sistema de ficheiros. Esta rotina, obtém os dados necessários, a mais baixo nível. Se não for possível efectuar essa leitura ou se o *device* não for do tipo de sistema pretendido, o comando `mount` falha.

2.1.2 Operações do superbloco

A estrutura de dados superbloco contém, conforme já foi referido, um apontador para as operações que permitem aceder ao sistema de ficheiros. Estas operações são usadas pelo VFS para ler e escrever *inodes* e o superbloco, e para obter informação acerca do sistema de ficheiros. Convertem deste modo, a informação contida no *device* físico, em estruturas mantidas em memória que são independentes do tipo de sistema de ficheiro, ficando assim ocultada a sua verdadeira representação física.

O sistema de ficheiros pode implementar as seguintes operações:

read_inode(inode) Esta função é responsável pelo preenchimento da estrutura *inode* correspondente ao *inode* indicado como parâmetro.

notify_change(inode,attr) Notifica o sistema de ficheiros da ocorrência de alterações ao *inode* indicado, devido a chamadas ao sistema. A maior parte dos sistemas de ficheiros não implementam esta função. O parâmetro *attr* refere a seguinte estrutura.

```
struct iattr {
 unsigned int ia_valid;   /* flags para os componentes
                             alterados */
 umode_t      ia_mode;    /* novas permissões de acesso */
 uid_t        ia_uid;     /* novo dono */
 gid_t        ia_gid;     /* novo grupo */
 off_t        ia_size;    /* novo tamanho */
 time_t       ia_atime;   /* data do último acesso */
 time_t       ia_mtime;   /* data da última alteração */
 time_t       ia_ctime;   /* data de criação */
};
```

write_inode(inode) Escreve a estrutura do *inode* indicado.

put_inode(inode) Esta função é chamada por `iput()` se o *inode* já não for necessário. A sua principal tarefa é a remoção física do ficheiro correspondente.

put_super(sb) O VFS chama esta função quando pretende desmontar o sistema de ficheiros, libertando assim o superbloco.

write_super(sb) Esta função é utilizada para guardar informação sobre o superbloco. Não garante necessariamente a consistência do sistema de ficheiros, uma vez que isso depende do facto de este permitir ou não a existência de uma *flag* que indique inconsistência. Esta função garante a sincronização com o *device* físico e é ignorada em sistemas de ficheiros que permitem apenas leitura.

statfs(sb,statfsbuff) As chamas ao sistema `statfs` e `fstatfs` despoletam a execução desta função, que apenas preenche a estrutura `statfs` (definida em asm/statfs.h).

```
struct statfs {
 long    f_type;     /* tipo do sistema de ficheiros */
 long    f_bsize;    /* tamanho dos blocos  */
 long    f_blocks;   /* número total de blocos */
 long    f_bfree;    /* número de blocos livres */
 long    f_bavail;   /* número de blocos livre disponíveis
                        para utilizadores vulgares */
 long    f_files;    /* número total de inodes */
 long    f_ffree;    /* número de inodes livres */
 fsid_t  f_fsid;     /* identificador do sistema de ficheiros */
 long    f_namelen;  /* tamanho máximo do nome de um ficheiro */
 long    f_spare[6]; /* não utilizado */
};
```

remount_fs(sb,flags,options) Altera o estado do sistema de ficheiros. A tabela 2.1 refere os possíveis valores do parâmetro *flags*. Esta função introduz os novos valores no superbloco e restaura a consistência do sistema de ficheiros.

Se alguma das operações não for implementada pelo sistema de ficheiros, o seu valor é NULL.

Macro	Valor	Descrição
MS_RDONLY	1	Sistema de ficheiros é só de leitura
MS_NOSUID	2	Ignorar os bits S
MS_NODEV	4	Não permite acesso a ficheiros relativos a devices
MS_NOEXEC	8	Não permite a execução de programas
MS_SYNCHRONOUS	16	Escrita para o disco imediata
MS_REMOUNT	32	Altera as flags

Tabela 2.1: *Flags* de `mount` referidas no superbloco

2.1.3 Inode

Quando um sistema de ficheiros é montado, é criado o respectivo superbloco, que refere no campo *mounted* a estrutura do *inode* da raiz do sistema. Esta estrutura contém a seguinte informação:

device Identificador do *device* que contém o ficheiro.

ino Número do *inode*.

mode Tipo de ficheiro e permissões de acesso.

nlink Número de *hard links*.

uid Número que identifica o dono.

gid Número que identifica o grupo.

size Tamanho do ficheiro.

atime Data do último acesso.

mtime Data da última alteração.

ctime Data da criação.

blksize Tamanho dos blocos.

blocks Número de blocos.

sem Semáforo para controlo de acesso.

wait Fila de espera.

count Número de componentes do sistema que estão a usar o ficheiro.

wcount Número de acessos para escrita.

dirt Indica que ocorreram alterações no *inode*.

sb Refere o superbloco do sistema de ficheiros.

op Apontador para a estrutura que refere as rotinas que operam no *inode*.

A informação contida nesta estrutura é constituída com base nos dados fornecidos pelo sistema de ficheiros subjacente, sendo indicados pela rotina que implementa a operação `read_inode`.
Os *inodes* são representados na memória de duas formas:

- Por listas circulares duplamente ligadas, que começam em *first_inode*. A procura de um *inode* nesta lista não é muito eficiente, pois a lista inclui *inodes* livres, isto é, *inodes* que não estão a ser utilizados.
- Por tabelas *hash*.

A função `iget(sb,nr)` (declarada em linux/fs.h) fornece o *inode* referente ao superbloco sb e ao *inode* cujo número é nr. Se este não fizer parte da tabela *hash*, é seleccionado um *inode* livre, através da execução da função `get_empty_inode` (definida em fs/inode.c). A sua estrutura é preenchida e acrescentada à tabela *hash*. Um *inode* obtido através de `iget()` deve ser libertado usando a função `iput()` (implementada em fs/inode.c), que decrementa o contador *count*.

2.1.4 Operações de inodes

A estrutura de dados *inode* possui um campo que refere as operações que o sistema de ficheiros pode implementar para a manipulação de *inodes*, estando estas indicadas na estrutura `inode_operations` (definida em linux/fs.h). Esta estrutura inclui as seguintes operações:

create(dir,name,len,mode,res_inode) Esta função é chamada pela função do VFS `open_namei()` (implementada em fs/namei.c) que inicialmente obtém um *inode* livre. `Create()` coloca no directório correspondente ao *inode dir*, o nome do ficheiro *name* cujo tamanho é indicado em *len*. Se o sistema de ficheiros não implementar a função `create()`, o VFS retorna o erro EACCESS (definido em linux/errno.h).

lookup(dir,name,len,res_inode) São fornecidos à função o nome de um ficheiro (*name*) e o seu tamanho (*len*). A função `lookup()` verifica se existe no directório *dir* o ficheiro indicado. Neste caso, coloca o seu *inode* em *res_inode*.

link(oldinode,dir,name,len) Esta função define um *link*. Um *link* é uma entrada num directório, que atribui um nome diferente a um ficheiro já existente no mesmo sistema de ficheiros. Deste modo podemos atribuir vários nomes a um mesmo ficheiro. Todos estes nomes referem o mesmo *inode*. Quando é criado um *link*, o campo *nlink* da correspondente estrutura *inode* é incrementado uma unidade. Um *inode* só deixa de existir quando forem removidos todos os seus *links*, isto é, quando o campo *nlink* contiver o valor 0.

A função `link` atribui o nome *name* ao ficheiro *oldinode*. Se a função não for implementada, a função do VFS que a chamou retorna o erro EPERM (definido em linux/errno.h).

unlink(dir,name,len) Remove o ficheiro *name* do directório indicado por *dir*. Se a função não for implementada, a função do VFS que a chamou retorna o erro EPERM (definido em linux/errno.h).

symlink(dir,name,len,symname) A função define um *link* simbólico – *name* – no directório *dir*. Um *link* simbólico é implementado como um ficheiro cujo conteúdo é o nome de outro ficheiro, que pode residir noutro sistema de ficheiros. Quando o sistema encontra um *link* simbólico, le o seu conteúdo para obter o nome do ficheiro que está a referir. Se o ficheiro referido for substituído por outro com o mesmo nome, o *link* passa a apontar para o novo ficheiro, pois a referência é feita através do nome do ficheiro.

O parâmetro *len* indica o tamanho do nome do ficheiro referido em *name*. O *link* aponta para o ficheiro *symname*. Se a função não for implementada, a função do VFS que a chamou retorna o erro EPERM.

mkdir(dir,name,len,mode) Define um subdirectório de *dir*, com o nome *name* e com permissões de acesso *mode*.

rmdir(dir,name,len) Remove do directório *dir* o subdirectório *name*. A função deve verificar se o directório a remover está vazio, se está a ser usado por algum processo e se tem permissões para efectuar a remoção.

mknod(dir,name,len,mode,rdev) Esta função define um novo *inode* com o modo *mode*, que terá o nome *name* no directório *dir*. Se o *inode* referir um ficheiro relativo a *devices*, o parâmetro *rdev* indica o número do *device*.

rename(odir,oname,olen,ndir,nname,nlen) Esta função altera o nome de um ficheiro. O nome *oname* é removido do directório *odir* e é acrescentado ao directório *ndir* o nome *nname*.

readlink(inode,buf,size) Esta função copia para *buf* o nome do ficheiro apontado pelo *link* simbólico *inode*. Se a função não for implementada pelo sistema de ficheiros, a chamada ao sistema retorna o erro EINVAL (definido no ficheiro linux/errno.h).

follow_link(dir,inode,flag,mode,r_inode) Retorna em *r_inode* o *inode* do ficheiro para onde o *link* simbólico aponta.

bmap(inode,block) O subsistema de gestão da memória [GC94] é uma das partes mais importantes do sistema operativo Linux. Para solucionar a constante falta de memória foram desenvolvidas estratégias, sendo a memória virtual a mais bem sucedida. Permite que o sistema aparente ter mais memória do que realmente tem, partilhando-a pelos vários processos em execução e permite igualmente a execução de programas cujo tamanho excede o da memória física.

Cada processo tem o seu próprio espaço de endereçamento virtual. Quando um programa é executado, a sua imagem tem de ser colocada no espaço de endereçamento virtual do processo – o ficheiro é mapeado na memória. A imagem é colocada na memória física, à medida que as partes do programa são referidas pela aplicação em execução. Deste modo evita-se que todo o programa seja colocado na memória física.

Esta função permite o mapeamento de ficheiros na memória. O argumento *block* indica o número de blocos lógicos de dados no ficheiro. Este número deve ser convertido por `bmap()` no número de blocos lógicos do media.

truncate(inode) Esta função tem a finalidade de encurtar um ficheiro ou alterar o seu tamanho, ficando este com o valor indicado no campo *size* do respectivo *inode*. Se a função não for implementada não é gerada nenhuma mensagem de erro.

permission(inode,flag) Verifica se o ficheiro correspondente ao *inode* tem as permissões de acesso referidas em *flag*. Estas permissões definem quem pode ler, escrever ou executar o ficheiro. Se a função não for implementada, a chamada ao sistema do VFS que a referiu, verifica as permissões usuais do Unix, não sendo assim habitualmente necessária a sua implementação.

smap(inode,sector) Esta função permite fundamentalmente a criação de ficheiros de *swap* em sistemas de ficheiros UMSDOS.

2.1.5 A estrutura do ficheiro

Em sistemas multi tarefa como o Linux, surgem problemas quando vários processos pretendem aceder simultaneamente a um ficheiro, para leitura ou escrita. Para evitar problemas de sincronização e permitir a partilha do acesso a ficheiros pelos diversos processos, foi introduzida uma nova estrutura `file` com a seguinte informação:

mode Permissões de acesso ao ficheiro.

pos Posição corrente dentro do ficheiro.

flags *Flags* de abertura do ficheiro.

count Número de acessos.

owner Número que identifica o dono do ficheiro.

inode Número do *inode* correspondente.

op Operações para manipulação do ficheiro.

2.1.6 Operações sobre ficheiros

A estrutura `file_operations` (definida em linux/fs.h) contém as funções que permitem abrir, fechar, ler e escrever ficheiros.

lseek(inode,file,offset,origin) Esta função permite indicar a posição a partir da qual o ficheiro vai ser acedido para leitura ou escrita.

read(inode,file,buf,count) Copia do ficheiro *file* para *buf*, o número de bytes indicado em *count*. Se a função não for implementada é retornado o erro EINVAL (definido em linux/errno.h).

write(inode,file,buf,count) Copia de *buf* para o ficheiro *file* o número de bytes indicado em *count*.

readdir(inode,file,dirent,count) Retorna a entrada no directório *dirent* situada na posição *count*. Se a função não for implementada o VFS retorna ENOTDIR (definido em linux/errno.h).

select(inode,file,type,wait) Verifica se existem dados a ler ou escrever para o ficheiro. Só é útil se o ficheiro for um *socket* ou um *driver*.

ioctl(inode,file,cmd,arg) Define parâmetros específicos do *device*.

map(inode,file,vm_area) Mapea parte de um ficheiro para o espaço de endereçamento dos utilizadores, do processo actual, isto é, coloca-o no espaço de endereçamento virtual do processo. A estrutura *vm_area* descreve as características dessa área de memória; os componentes *vm_start* e *vm_end* indicam os endereços inicial e final da área de memória.

open(inode,file) Abre um ficheiro.

release(inode,file) A função é chamada quando a estrutura do ficheiro é libertada.

fsync(inode,file) Garante que todos os *buffers* do ficheiro foram actualizados e que os dados que contêm foram escritos no *device*.

fasync(inode,file,on) É executada a pedido da chamada ao sistema `fcntl`.

check_media_change(dev) Esta função só é relevante para *devices* formados por blocos que suportam a troca do media.

revalidate(dev) É chamada após a troca de um media, de forma a restaurar a consistência do *device*.

2.1.7 "Cache" de directórios

O VFS mantém uma "cache" com entradas correspondentes a directórios, cuja finalidade é aumentar a velocidade no acesso aos directórios mais comuns [Rus]. Deste modo, quando é solicitada a listagem de um directório é procurada a sua referência na "cache". Só são guardados aí os directórios com nomes mais pequenos (até 15 caracteres) pois são os utilizados com mais frequência.

A "cache" de directórios é uma tabela *hash* onde cada entrada aponta para uma lista de directórios com o mesmo valor *hash*. Este valor é calculado com base no número do *device*, no número do *inode* do directório e do seu nome. A "cache" é estruturada em 2 níveis que operam com base no algoritmo LRU (Least Recently Used). Quando um directório é colocado pela primeira vez na "cache" é introduzido no nível 1, o que leva à remoção da entrada mais antiga. Quando o directório for acedido novamente, a sua entrada é colocada no nível 2, o que pode levar à remoção da entrada mais antiga. As entradas que se encontram neste nível são as que são mais frequentemente referidas.

As funções que acedem à "cache" são as seguintes:

dcache_add Coloca uma entrada na tabela.

dcache_lookup Interroga a tabela.

2.1.8 O sistema de ficheiro *Proc*

O sistema de ficheiros *Proc* demonstra o poder do sistema virtual de ficheiros do Linux. Fornece informação sobre o estado corrente do kernel e dos processos em execução. A cada processo em execução é atribuído um directório /proc/pid, onde pid é o identificador do processo.

O conteúdo dos ficheiros é gerado à medida que é solicitado. Quando o VFS interroga o sistema de ficheiros *Proc*, solicitando acesso a ficheiros, este cria os ficheiros com base na informação do kernel.

2.1.9 Conclusão

Com o tipo de interface exposto, qualquer utilizador pode criar um novo tipo de sistema de ficheiros. Basta para isso que defina as estruturas superbloco, *inode* e ficheiro e implemente as operações necessárias à sua manipulação, fornecendo deste modo todos os dados que o VFS necessita.

Esta facilidade de criação de novos tipos de sistemas de ficheiros demonstra bem a flexibilidade do sistema operativo Linux.

2.2 Módulos

O kernel do Linux é um programa único onde todos os seus componentes têm acesso a todas as suas estruturas de dados e rotinas. Todos os *devices* e sistemas de ficheiros usados são permanentemente incorporados no kernel. Quando a sua configuração é alterada, o kernel tem de ser recompilado. Os *drivers* e sistemas de ficheiros incluídos ocupam permanentemente espaço de memória, mesmo que sejam raramente usados.
Alternativamente podemos partir o kernel em várias unidades que comunicam entre si. Assim, podemos mais facilmente acrescentar novos componentes ao kernel, sem ter de configurar e construir um novo kernel, diminuindo assim o seu tamanho. O Linux permite acrescentar e remover componentes do sistema operativo (módulos), de forma dinâmica, à medida que estes forem necessários. Os módulos são conjuntos de código que podem ser dinamicamente ligados ao kernel em qualquer altura, após a iniciação do sistema.

Os módulos podem ser inseridos ou removidos no kernel recorrendo aos comandos `insmod` e `rmmod`. O próprio kernel pode solicitar a inserção ou remoção de módulos à medida que são usados, através do *daemon* `kerneld`. Um módulo faz parte do kernel do mesmo modo que o código normal do kernel. Tem os mesmos "direitos" que o resto do código que constitui o kernel. Quando se tenta remover um módulo, o kernel verifica se este não está a ser usado e neste caso notifica o módulo, de forma a este libertar recursos do sistema que tenha usado.

2.3 Kernel 2.2

O kernel 2.2 [Goob] apresenta algumas alterações a nível da implementação do sistema de ficheiros.
Foi acrescentada uma nova "cache" de directórios – dentry – que optimiza as operações de procura de directórios, diminuindo 4 vezes o tempo de procura [Gooa]. A estrutura `dentry` contém a seguinte informação:

d_count Número de acessos.

d_flags

d_inode *Inode*.

d_parent Directório pai.

d_mounts Informação sobre a raiz dos sistemas de ficheiros montados.

d_covers Informação sobre os pontos onde os sistemas de ficheiros estão montados.

d_hash Tabela *hash* de procura.

d_lru Lista LRU.

d_child Filhos do directório pai.

d_subdirs Filhos.

d_alias Lista de *alias*.

d_name Nome da entrada.

d_time Usado pela operação *d_revalidate*.

d_op Operações que manipulam a estrutura.

d_sb Superbloco. Raiz da árvore de directórios.

d_reftime Data da última utilização.

d_fsdata Dados específicos do sistema de ficheiros.

As operações que permitem manipular a estrutura *dentry* e que podem ser implementadas pelo sistema de ficheiros, são as seguintes:

d_revalidate(dentry) É chamada quando o VFS necessita de revalidar a *dentry*.

d_hash(dentry,name) Acrescenta a entrada *name* à tabela *dentry*.

d_compare(dentry,name1,name2) Compara as duas estruturas cujo directório pai é *dentry*.

d_delete(dentry) É chamada quando a última referência a *dentry* é removida, o que significa que ninguém a está a usar.

d_release(dentry) Liberta a entrada.

d_iput(dentry,inode) Obtém o *inode* da *dentry*.

As declarações de funções foram alteradas, tendo sido substituído o parâmetro *inode* por *dentry*. A ordem por que são apresentadas, também foi alterada em alguns casos. É necessário ter em conta esta nova ordem, ao declarar as operações nas respectivas estruturas.

2.3.1 Superbloco

A estrutura `file_system_type` é agora constituída por:

- Nome do tipo de sistema de ficheiros.
- *Flags*. Podem ter os seguintes valores:
  - FS_REQUIRES_DEV
  - FS_NO_DCACHE Apenas faz *dcache* do que for necessário.
  - FS_NO_PRELIM Evita que as estruturas *dentry* sejam carregadas previamente.
  - FS_IBASKET O sistema de ficheiros chama `free_ibasket()`.
- Rotina de leitura do superbloco.
- Apontador para a estrutura seguinte.

Assim, para montar o sistema de ficheiros "minix", é necessário declarar

```
struct file_system_type minix = {
      "minix",
      FS_REQUIRES_DEV,
      minix_read_super,
      NULL
};
```

Foram acrescentadas as seguintes operações que manipulam o superbloco:

delete_inode(inode) Remove o *inode* referido.

clear_inode(inode) Limpa o *inode*. Indica que este deixou de ser utilizado.

umount_begin(sb)

A operação `notify_change` passa a ter como primeiro argumento um apontador para *dentry* em vez de um apontador para *inode*, como acontecia nas versões anteriores do kernel.

2.3.2 Inode

A estrutura `inode` não sofreu muitas alterações. O campo *dirt*, que indicava a ocorrência de alterações no *inode*, deixou de existir. Foi acrescentado um apontador para a estrutura *dentry*.

Foram acrescentadas as seguintes operações à estrutura `inode_operation`:

readpage(file,page) Lê uma página de memória a partir do *device* físico.

writepage(file,page) Escreve para o *device* físico a página da memória.

updatepage(file,page,offset,count,sync)

revalidate(dentry) Revalida a entrada.

Cada *inode* pode estar indicado em duas listas. Uma delas é a tabela *hash* de *inodes*, usada para procuras. A outra é uma lista ligada onde é representado o estado do *inode*, que tem um dos seguintes valores.

in_use *Inode* é válido. O campo *i_nlink* é maior que zero.

dirty O *inode* é válido, mas a sua informação tem de ser actualizada.

unused O *inode* pode ser reutilizado.

Todas as outras operações, com excepção de `bmap, truncate, permission` e `smap`, sofreram alterações nos seus argumentos. Em vez do parâmetro

> struct inode *inode

passa a ser referido o parâmetro

> struct dentry *dentry

2.3.3 Ficheiros

A estrutura `file` passa a conter informação acerca do número que identifica o dono do ficheiro e o grupo, um campo de erro e um apontador para a respectiva estrutura *dentry*, o que provoca a eliminação do campo *inode*.

Esta versão do kernel introduz duas novas chamadas ao sistema, `pread` e `pwrite`, que permitem que um processo leia ou escreva num ficheiro, a partir de uma certa posição. É idêntico à execução de `lseek` seguida de `read` ou `write` conforme se pretenda ler ou escrever. A diferença reside no facto de permitir acessos simultâneos ao ficheiro. Para permitir a utilização destas novas chamadas ao sistema, é acrescentado um novo parâmetro às operações `read` e `write`, que indica a posição dentro do ficheiro.

As chamadas ao sistema `select` e `poll` permitem que um processo aguarde pela ocorrência de eventos I/O, com origem em vários descritores de ficheiros. No kernel 2.0 era referida na estrutura `file_operations` a operação `select`. No kernel 2.2, esta é substituída pela operação `poll`, que proporciona uma maior flexibilidade.

Foi excluído o parâmetro *struct inode *inode* da maior parte das outras operações, com excepção de `ioctl`, `open`, `release`, `chek_media_change` e `revalidate`.

O VFS implementa a chamada ao sistema `open()` e utiliza o nome do ficheiro indicado como argumento, para efectuar uma procura na "cache" de directórios *dentry*. Este é um método rápido de transformar o nome de um ficheiro na *dentry* correspondente. Para executar este método o VFS chama a operação `lookup` que é implementada pelo sistema de ficheiros onde o ficheiro reside, indicando o *inode* do directório pai. Depois de obter a apropriada entrada na estrutura *dentry*, o VFS pode efectuar a abertura do ficheiro.

Capítulo 3

Userfs

O `userfs` [Fit] é um mecanismo através do qual, um processo em execução com as permissões do utilizador que o lançou, pode ser visto como um sistema de ficheiros. A sua função é o redireccionamento dos pedidos do VFS para o processo do utilizador, vulgarmente denominado cliente, que efectivamente implementa o sistema de ficheiros. O `userfs` é um módulo do kernel, podendo a sua inserção ser efectuada em qualquer altura após o arranque do sistema, recorrendo para isso aos comandos que formam o pacote de software `modutils`.

Neste capítulo é explicado o funcionamento do *userfs* e são referidos os requisitos que os programas clientes devem satisfazer.

3.1 Iniciação do módulo

A versão actual do `userfs` – userfs-0.9.5 – só suporta versões do kernel 2.0. Para inserir o módulo é necessário possuir os comandos que formam o *modutils*. Este é constituído por comandos que permitem inserir e remover módulos e listar os módulos que estão carregados.

Assim, após a compilação do *userfs*, o módulo pode ser inserido no kernel através da execução do comando

% insmod userfs.o

3.2 Programas clientes

Para iniciar a execução de um programa cliente, que, como foi referido, é o programa que efectivamente implementa o sistema de ficheiros, é necessário montar o sistema de

ficheiros num ponto da árvore de directórios do sistema.
É assim executado o comando,

% muserfs client mpoint args

onde *client* refere o programa cliente, cujos argumentos são indicados em *args*. O sistema de ficheiros é montado no directório *mpoint*.

`Muserfs` é um comando inserido no userfs-0.9.5, que efectua os seguintes procedimentos:

- Verifica se o programa cliente existe.
- Verifica se *mpoint* é um directório e se o utilizador que lançou o programa `muserfs` é o seu dono.
- Lança outro processo que executa o programa cliente com os argumentos *args*. Define os canais de comunicação com o cliente, para escrita e leitura.
- Verifica se o cliente implementa a operação *mount*.
- Monta o sistema de ficheiros no directório indicado, usando a chamada ao sistema `mount`. Acrescenta ao ficheiro /etc/mtab uma nova linha, indicando o directório onde o sistema de ficheiros *userfs* está montado.

O programa `arcfs` é o exemplo de um cliente, que permite montar um ficheiro comprimido com *tar*, como um sistema de ficheiros. Assim, depois de iniciar o programa

% muserfs arcfs /tmp userfs.tgz

podemos listar o conteúdo do directório /tmp e ter acesso a todos os ficheiros que formam o userfs.tgz, como se eles aí residissem.

```
% ls /tmp
Announce   Makefile   clients/   genser/   lib/
Copying    Readme     contrib/   kernel/
```

O sistema desenvolvido neste trabalho – `rfs` – é outro exemplo de um programa cliente.

3.3 Funcionamento

O sistema de ficheiros *userfs* regista-se no kernel através da execução de:

```
  init_module(void) {

   struct file_system_type ufst = {
     userfs_read_super,
     ``userfs'',
      0,
      NULL
   };

   register_filesystem(&ufst);

 }
```

O VFS executa a rotina de leitura do superbloco `userfs_read_super` que efectua os seguintes passos:

- Verifica se os descritores de ficheiros que permitem comunicar com o kernel, são válidos. O kernel envia pedidos ao programa através do descritor *fromkern* e o cliente envia as respostas através do descritor *tokern*.
- Define as operações que manipulam o superbloco.
- Envia ao programa cliente um pedido `up_mount` de forma a obter o *handle* da raiz do sistema de ficheiros.
- Define as operações que manipulam *inodes* e ficheiros e verifica quais dessas operações são implementadas pelo cliente.
- Envia ao cliente um pedido `up_iread` de forma a obter os dados sobre o *inode* da raiz do sistema de ficheiros.

3.3.1 Comunicação entre o kernel e o cliente

Quando o VFS envia ao *userfs* pedidos de execução de operações, este redirecciona-os para o programa cliente.
Por exemplo, se um utilizador pretender ler o ficheiro /tmp/Readme referido na secção anterior, o VFS redirecciona o pedido para o sistema de ficheiros que contém o ficheiro indicado. Para isso, verifica qual a função do sistema de ficheiros *userfs* que implementa

a operação *read* e executa-a. O *userfs* por sua vez, envia o pedido de *read* ao programa cliente, que efectivamente executa a leitura.

A comunicação entre o kernel e o processo cliente é feita através de dois descritores de ficheiros ligados ao processo. Normalmente são utilizadas *pipes*, mas também podem ser usados *sockets*, *devices* ou ficheiros. Esta comunicação é feita através do envio de mensagens, formadas por duas partes. A primeira é o cabeçalho, que é constituído por:

- Número de sequência - identifica a mensagem.
- Tipo de operação - indica qual a operação que se pretende executar.
- Tipo de mensagem - a mensagem pode referir um pedido, uma averiguação ou uma resposta. Pedidos e averiguações são sempre enviados pelo kernel para o processo cliente. Este apenas envia mensagens de resposta.
- Tamanho dos dados que se seguem.
- Versão do protocolo.

O cabeçalho das mensagens de resposta enviadas pelo cliente ao kernel, contém um campo extra – errno – que indica o número do erro ocorrido, ou zero, caso não tenha ocorrido nenhum erro. Os cabeçalhos das respostas referem sempre o mesmo número de sequência da correspondente pergunta ou averiguação.
A segunda parte da mensagem é formada pelos dados resultantes da execução do pedido.

3.3.2 Ocorrência de vários pedidos

O VFS pode executar várias chamadas ao sistema, que necessitam de dados fornecidos pelo sistema de ficheiros *userfs*, originando assim pedidos ao programa cliente. Pode ser enviado um pedido antes do pedido anterior ter obtido uma resposta, havendo assim normalmente vários processos à espera de resposta. Neste caso, apenas um dos processos fica em espera activa, enquanto os outros "adormecem".

É mantida uma lista dos processos que estão adormecidos. Os dados relativos a cada um destes processos são guardados na estrutura `pwlist` ilustrada na figura 3.1.

O processo activo aguarda que seja enviada uma resposta pelo cliente. Quando isso acontece, verifica se a resposta é para si, isto é, verifica se o número da resposta corresponde ao número sequencial do pedido que enviou. Neste caso, descodifica os dados da resposta e acorda um dos processos que está na fila de espera. Caso contrário, acorda o processo que enviou ao cliente a pergunta com aquele número sequencial e continua à espera de mais respostas enviadas pelo cliente.

```
struct pwlist
{
 struct wait_queue *bed;   /* onde está a dormir  */
 int     why;              /* motivo porque acordou */
 int     seq;              /* número do pedido */
 upp_repl repl;            /* cabeçalho da resposta */
 char    *more;            /* dados extra */
 int     pid;              /* número do processo que o acordou */

 struct pwlist *next;      /* processo seguinte na lista de espera */
 struct pwlist *prev;      /* processo anterior na lista de espera */
};
```

Figura 3.1: Estrutura `pwlist`

Quando um processo acorda, verifica o que motivou esse acontecimento e age do seguinte modo, de acordo com o motivo:

- Se tiver sido acordado pelo processo activo por este ter terminado a sua execução, torna-se ele próprio o novo processo activo, ficando a aguardar por uma resposta do cliente.
- Se tiver sido acordado porque o programa cliente enviou a resposta ao seu pedido, descodifica os dados recebidos.
- Se tiver ocorrido uma falha na comunicação com o cliente, retorna um erro.

3.3.3 *Handles*

O elemento base de um sistema de ficheiros é o *inode*, que é univocamente numerado dentro de cada sistema de ficheiros. Como cada sistema montado tem a sua própria numeração, um *inode* fica identificado pelo seu número e pelo identificador do sistema de ficheiros onde reside. O kernel deve ser capaz de identificar de forma inequívoca um *inode*. O número de um *inode* – *handle* – é gerado pelo sistema de ficheiros e é usado pelo kernel para referir um determinado ficheiro. Os *handles* devem ser consistentes. Se um processo depois de abrir um ficheiro, o fechar e o reabrir mais tarde, espera que este tenha o mesmo número de *inode* uma vez que se trata do mesmo ficheiro. Se o kernel ler o conteúdo de um directório recorrendo à operação `up_readdir`, espera receber como resposta o *handle* e o nome de cada um dos ficheiros que o constituem. Se o *handle* for igual para todos, o kernel conclui que todos os nomes do directório referem o mesmo ficheiro.
É o programa cliente que deve indicar o *handle* de cada ficheiro e garantir a sua consistência.

3.3.4 *Mount*

O pedido mais importante é o de *mount*. O programa cliente tem que implementar obrigatoriamente este pedido, indicando como resposta o *handle* da raiz do sistema de ficheiros. O kernel envia seguidamente várias mensagens de averiguação, para verificar quais as operações implementadas pelo cliente. Este deve responder com mensagens de resposta normais, indicando no campo de erro o valor zero, no caso de implementar a operação indicada, ou ENOSYS (definido em linux/errno.h) caso contrário.

3.3.5 Leitura de inodes

O pedido mais usual é o de leitura de *inodes*. O processo cliente deve preencher uma estrutura similar à estrutura *inode*, do kernel. O campo *nlink* não deve ter o valor 0, pois indica o número de nomes que o *inode* tem, isto é, o número de entradas num directório que referem esse *inode*.

Quando o kernel pretende obter um *inode* envia o pedido `up_iread`. Quando não existir nada no kernel que use o *inode*, é enviado o pedido `up_iput` que pode ser precedido de `up_iwrite` se tiverem ocorrido alterações ao *inode* durante a sua utilização. O cliente pode não implementar estas operações, se tal não for necessário.

3.4 Alterações efectuadas ao userfs

Conforme já foi referido, a versão original do *userfs* só suporta versões do kernel 2.0. Para funcionar com o kernel 2.2.6 foi necessário efectuar alterações ao código do programa. Nesta secção serão referidas as alterações efectuadas em cada um dos ficheiros que constituem o módulo.

Todas as referências à função `memcpy_to_fs` foram alteradas, uma vez que esta função, que permite copiar qualquer número de bytes para o segmento de memória do utilizador, deixou de existir no kernel 2.2.6. Foi substituída por `copy_to_user`, que tem a mesma finalidade.

3.4.1 Modulo.c

Devido à alteração da estrutura `file_system_type`, foi necessário modificar a informação sobre *userfs* que é enviada ao kernel durante o registo do sistema de ficheiros.

```
struct file_system_type ufst = {
     ‘‘userfs’’,
      0,
```

```
        userfs_read_super,
        NULL
};
```

3.4.2 Super.c

Neste ficheiro são implementadas as funções que permitem manipular o superbloco. A forma como são declaradas na estrutura `userfs_super_operations` foi alterada devido à nova representação de `super_operation`.

A operação `notify_change` tem como primeiro argumento um apontador para *dentry* em vez de um apontador para *inode*.
Assim a função

> userfs_notify_change(struct inode *inode, struct iattr *iattr)

foi substituída por

> userfs_notify_change(struct dentry *dentry, struct iattr *iattr)

e foi acrescentada à função, a linha

> struct inode *inode = dentry->d_inode;

de forma a obter o *inode* necessário ao resto do código.

3.4.3 Inode.c

São declaradas em `userfs_inode_operations`, as operações implementadas que permitem manipular *inodes*. Como a estrutura `inode_operations` foi alterada relativamente à versão 2.0 do kernel, foi necessário modificar a ordem por que são declaradas as funções e acrescentar linhas com o valor NULL na posição correspondente às novas operações introduzidas nesta estrutura, pois o programa não as implementa.

A maior parte das funções aqui representadas apresentam novos argumentos. São indicadas seguidamente as operações que sofreram alterações:

- **userfs_lookup**

  A função

  > userfs_lookup(struct inode *dir, char *name, int len,
  > struct inode **result)

foi substituída por

userfs_lookup(struct inode *dir, struct dentry *dentry)

O nome do ficheiro e o seu tamanho são obtidos a partir da estrutura *dentry*.

Esta função retorna o *inode* do ficheiro indicado. O seu valor era indicado na estrutura *result*, mas uma vez que esta estrutura deixou de ser um argumento, foi necessário alterar o modo como é retornado. Foi assim acrescentada a chamada à função

d_add(dentry,inode)

(definida em linux/dcache.h), que acrescenta a entrada à tabela *hash* e define os valores de *d_inode*.

- **userfs_create**

  A função

  userfs_create(struct inode *dir, char *name, int len,
  int mode, struct inode **result)

  foi substituída por

  userfs_create(struct inode *dir, struct dentry *dentry, int mode)

  O nome do ficheiro e o seu tamanho são obtidos a partir da estrutura *dentry*. Esta função cria o ficheiro indicado e obtém um *inode* que o identifique. Esse *inode* era colocado em *result*. Uma vez que esta variável já não faz parte da lista de argumentos, foi necessário alterar a forma como o seu valor é passado ao kernel. É assim utilizada a função do VFS

  d_instantiate(dentry,inode);

  (definida em linux/dcache.h), que preenche a informação sobre o *inode* contida na estrutura *dentry*.

- **userfs_mknod, userfs_mkdir, userfs_symlink, userfs_unlink, userfs_rename**

  Todas estas funções sofreram a mesma alteração. O nome do ficheiro e o tamanho desse nome deixaram de ser indicados como argumentos e passou a referir-se a estrutura *dentry*. A partir desta estrutura podemos obter o nome e o tamanho do ficheiro, sendo assim acrescentado ao código de cada uma destas funções:

  char *name = dentry->d_name.name;
  int len = dentry->d_name.len;

  À função `userfs_unlink` foi acrescentada a linha d_delete(dentry)

(definida em linux/dcache.h), para que seja apagada a estrututa *dentry* correspondente ao ficheiro removido.

- **userfs_link**

  A declaração da função

  userfs_link(struct inode *oldino, struct inode *dir, char *name, int len)

  foi substituída por

  userfs_link(struct dentry *old_dentry, struct inode *dir,
  struct dentry *new_dentry)

  A partir da estrutura *old_dentry* obtemos o *inode oldino*. Os valores de *name* e *len* são obtidos a partir de *new_dentry*.

- **userfs_readpage**

  O primeiro argumento da função deixa de ser um apontador para um *inode*, sendo substituído por um apontador para a estrutura *file*. Com base nesta estrutura obtemos o *inode*.

  struct inode *inode = file->f_dentry->d_inode;

3.4.4 File.c

As funções implementadas neste ficheiro são declaradas em `userfs_file_operations`. Foi necessário alterar esta estrutura, devido às modificações verificadas na estrutura `file_operations`.
Foram assim efectuadas alterações ao código das seguintes funções:

- **userfs_read, userfs_write**

  Destas duas funções é retirado o primeiro argumento que refere um apontador para a estrutura *inode* e acrescentado o argumento *loff_t *ppos*.

- **userfs_readdir, userfs_mmap**

  Os argumentos destas funções deixam de incluir um apontador para o *inode*. O seu valor pode ser obtido através da informação contida na correspondente estrutura *file*.

  struct inode *inode = file->f_dentry->d_inode;

Capítulo 4

Rfs

O `Rfs` é, como o nome indica, um sistema de ficheiros que disponibiliza estruturas de directórios localizadas noutras máquinas. Estas estruturas são duplicadas localmente, sendo efectuada a cópia de ficheiros e a criação de subdirectórios na altura da sua solicitação, se ainda não existirem na máquina local. As cópias locais são mantidas num directório "cache" assim como os ficheiros de configuração do programa.

A finalidade deste sistema é a criação de servidores ftp centralizadores que contenham informação existente noutros servidores sem a transferir toda de uma só vez.
O sistema é normalmente montado no directório público de ftp. Quando os utilizadores acedem a este ponto do sistema de ficheiros, estão na realidade a visualizar a informação que o *rfs* lhes fornece e não o conteúdo real do directório. Para efectuar tal "simulação" o sistema recorre à informação contida na "cache".

A implementação do *rfs* é feita ao nível da camada de ficheiros tendo por base o sistema de ficheiros `userfs`. O sistema de ficheiros é montado, recorrendo ao utilitário `muserfs`. São indicados como parâmetros o directório "cache" e o ponto de montagem.

% muserfs rfs /usr1/cache ~ftp/pub/Linux

Neste capítulo são explicados os conceitos básicos do sistema implementado, a estrutura da "cache" e a representação interna dos dados. São referidas políticas de gestão do espaço de disco e de validação dos dados contidos na "cache".

4.1 Ficheiro de configuração

A iniciação do `rfs` baseia-se no ficheiro de texto `.config` que contém a configuração necessária ao funcionamento do sistema. Aí se encontra a indicação dos directórios básicos que estamos a manter, e do local onde os respectivos directórios originais estão disponíveis. Se este ficheiro não existir ou não possuir uma sintaxe correcta, o funcionamento do sistema é inviabilizado.

O ficheiro `.config` encontra-se no directório “cache”. As linhas que começam por # são comentários e são ignoradas, assim como as linhas em branco. As declarações são formadas por uma palavra chave e pelo valor correspondente.

As palavras chave válidas são as seguintes:

Partition y|n Indica se o directório “cache” se encontra ou não, numa partição do disco que lhe é exclusivamente destinada.

Max_space *Mb* espaço de disco (em Mb) disponível. Só é necessária a sua indicação quando não existe uma partição para uso exclusivo do `rfs`.

Lo_space *Kb* Número de *Kb* que é obrigatório manter livres. Quando este valor é atingido é forçada a remoção de ficheiros.

Hi_space *Kb* Espaço livre em disco (em *Kb*) que é necessário obter, quando o processo de remoção de ficheiros é activado. São removidos ficheiros até ser atingido este valor.

Cache_entry entry Indicação dos directórios que estamos a manter.
entry corresponde a

dir site:/path update expire

dir Directório da “cache”.

site Máquina remota onde está disponível o directório original.

path Directório original na máquina remota.

update Durante quanto tempo (dias, horas ou minutos) um ficheiro ou directório da “cache” é considerado actualizado.

expire Ao fim de quanto tempo (dias, horas ou minutos) um ficheiro ou directório é considerado obsoleto.

O motivo da existência dos dois valores *update* e *expire* é exposto na secção 4.6.

Na figura 4.1 temos o exemplo de um ficheiro de configuração. A “cache” não se encontra numa partição de disco exclusivamente por si ocupada. São-lhe reservados 1000Mb de disco e devem existir sempre 1000Kb de espaço livres. Quando este valor for atingido os ficheiros são removidos até se obterem 2000Kb livres.
A “cache” possui apenas um directório – debian – onde é mantida uma cópia da estrutura de ficheiros /pub/Linux/debian que se encontra no computador ftp.up.pt.
A informação contida neste directório é válida por 1 dia, findo o qual é necessário proceder à sua actualização. Se esta não for feita no prazo de 2 dias o directório é considerado inválido sendo os ficheiros que nele se encontram, removidos.

Exemplo

```
partition n
max_space 1000
lo_space 1000
hi_space 2000
cache_entry   debian   ftp.up.pt/pub/Linux/debian    1d 2d
```

Figura 4.1: Ficheiro de configuração .config

4.2 Estrutura da "cache"

O directório "cache" é indicado no lançamento do programa sendo o seu nome passado como parâmetro deste. Aí vão residir as cópias dos ficheiros originais obtidas a partir das máquinas remotas, sendo a transferência efectuada sempre que é pedido por um utilizador um ficheiro que ainda não existe na "cache". São igualmente mantidos neste directório os ficheiros de configuração essenciais ao funcionamento do programa.

Inicialmente deve ser constituída apenas pelo ficheiro `.config` referido na secção anterior. Com base na informação nele contida são criados os directórios indicados pela instrução *cache_entry*, correspondendo cada um a um *mirror* que estamos a manter. Em cada um destes directórios vai ser gerada uma estrutura de ficheiros idêntica à da máquina remota. Os subdirectórios serão criados à medida que for obtida informação acerca da sua existência e conteúdo. Os ficheiros são colocados nos respectivos directórios preservando o seu nome original.

Quando um utilizador solicita a listagem do conteúdo de um directório que ainda não é conhecido pelo `rfs`, é estabelecida uma ligação à máquina remota de forma a obter a informação necessária à satisfação do pedido. Esta é guardada localmente num ficheiro `.list` debaixo do directório em questão. São seguidamente criados todos os subdirectórios.
O ficheiro `.list` possui uma linha por entrada, com a seguinte estrutura:

Mod size date name link

Mod Número representando o tipo de ficheiro (d, -, l) e as suas permissões.

Size Tamanho do ficheiro em bytes.

Date Data da última alteração.

Name Nome do ficheiro.

Link Se for um *link*, refere o ficheiro para onde está a apontar.

Inicialmente pensou-se em não criar os subdirectórios e representar toda a estrutura de ficheiros num único ficheiro `.list`, o que obrigaria a consultas e alterações constantes deste. Se vários utilizadores estivessem a aceder ao programa simultaneamente o processo tornar-se-ia moroso, pois sendo o acesso para escrita exclusivo só um poderia estar a alterar este ficheiro. Este tipo de abordagem teria a vantagem de economizar *inodes* visto que eliminava a criação dos subdirectórios. Tendo em conta que actualmente o número de *inodes* não é um recurso escasso, preferiu-se dar mais atenção ao desempenho do sistema.

Na figura 4.2 podemos ver um exemplo da estrutura da "cache" e a forma como a informação nela contida é vista pelo utilizador.

```
Uma cache com a estrutura

 cache |- .config
       |- debian
           |- .list
           |- dists
           |- doc
           |- indices
           |- tools
       |- gnu

será vista pelo utilizador, como

  % dir debian

   -rw-r--r--   1 ftp   ftp        6259 Aug 17  1998 README
   drwxr-xr-x   6 ftp   ftp        1024 Mar 31 15:28 dists/
   drwxr-xr-x   4 ftp   ftp        1024 Mar 31 15:42 doc/
   drwxr-xr-x   2 ftp   ftp        1024 Mar 31 15:18 indices/
   drwxr-xr-x   2 ftp   ftp        1024 Mar 31 15:42 tools/
```

Figura 4.2: Directório "cache"

4.2.1 Logs

Os *logs* do programa são mantidos no subdirectório da "cache" `log`, não sendo visíveis pelos utilizadores.

error contém todas as mensagens de erro com a indicação da data e hora a que ocorreram.

expire contém os nomes dos ficheiros removidos e as correspondentes datas de criação e remoção. Com estes dados podemos saber quanto tempo cada ficheiro permaneceu na "cache".

4.3 Representação interna dos dados

O utilizador ao aceder ao directório onde está montado o sistema visualiza uma estrutura de ficheiros que efectivamente não existe no disco, mas é simulada pelo `rfs`. Para permitir esta funcionalidade o `rfs` tem que possuir em memória toda a estrutura "visível", isto é, necessita de ter sempre presente o nome dos ficheiros e directórios que a formam, quer estes existam ou não na "cache" e toda a informação que lhes é inerente, como por exemplo o seu tamanho e data de criação.

Essa informação é mantida numa lista `Entry` cuja estrutura está indicada na figura 4.3.

Na figura 4.4 está representada a estrutura `Ino` que possui a seguinte informação:

- Identificação do dono do ficheiro.
- Identificação do grupo a que pertence.
- Permissões do ficheiro e seu tipo.
- Data da criação.
- Data da última modificação.
- Data da última alteração de estado.
- Tamanho do ficheiro em bytes.

Quando o programa inicia preenche as estruturas referidas com a informação contida na "cache".

- Cria a estrutura inicial com a informação do directório onde é montado o sistema. Toda a estrutura vai ser representada em subdirectórios deste.
- Lê o ficheiro `.config` e cria uma entrada para cada directório nele referido.
- Lê o ficheiro `.list` de cada directório, caso exista. Com base na informação nele contida cria entradas para os ficheiros e directórios nele referidos.

```
 struct Entry {
  Entry *next;
  Entry *prev;
  char *name;       /* nome do ficheiro   */
  char *source;     /* nome do ficheiro original */
  char *link;       /* se for um link, refere o nome do ficheiro */
  char change_d[6]; /* quanto tempo o ficheiro é válido */
  char exp_d[6];    /* ao fim de quanto tempo expira */
  short valid;
  Entry *pdir;      /* directório pai */
  Ino *ino;         /* estrutura com a informação do inode */
  union {
   struct {
    int fd;         /* descritor do ficheiro, quando este está aberto
                        para leitura */
    int status;
   }file;           /* só para ficheiros */
   struct {
    Entry *list;    /* aponta para lista dos ficheiros filhos */
    short init;     /* indica se possui ou não informação acerca do
                       seu conteúdo */
   }subdir;         /* só para directórios */
  } data;
};
```

Figura 4.3: Estrutura Entry

- Percorre recursivamente todos os directórios existentes na "cache" repetindo para cada um o procedimento anterior.

Sempre que é transferido um ficheiro ou obtida a listagem do conteúdo de um directório, são criadas as entradas respectivas na estrutura. Todas as alterações ao estado de um ficheiro repercutem-se imediatamente na estrutura que o representa. Quando deixa de haver referência a um ficheiro a sua informação é retirada da lista. No caso de se tratar de um directório são igualmente retirados todos os ficheiros lá residentes.

Nem toda a informação necessária ao funcionamento do programa pode ser obtida com base no conteúdo da "cache" e dos ficheiros de configuração. Há dados que convém manter num registo permanente de forma a estarem disponíveis em futuras

```
struct inode {
     uid_t uid;
     uid_t gid;
     umode_t mode;
     time_t atime;
     time_t mtime;
     time_t ctime;
     off_t size;
};
```

Figura 4.4: Estrutura Inode

reiniciações do programa, pois não poderiam ser obtidos de outra forma. É assim mantido no directório "cache" um ficheiro `.info` com a seguinte informação relativa a todos os ficheiros existentes na "cache":

fname Nome do ficheiro.

creation Data em que foi criado. Uma vez que a data do ficheiro é alterada, ficando com a do ficheiro original, é necessário saber quando foi criado na "cache" para se poder verificar a validade do mesmo.

last Data em que se verificou pela última vez a validade do ficheiro, isto é, em que se concluiu que era igual ao original.

actime Data em que o ficheiro foi acedido pela última vez. Esta informação poderia ser obtida a partir dos dados que o sistema de ficheiros mantém, mas deste modo facilitamos a rapidez de execução de programas de gestão da "cache".

md5 Sempre que a máquina remota fornece o *md5 checksum* (explicado na subsecção 4.4.2.1) este é aqui guardado. É necessário para averiguar se o ficheiro da "cache" é ou não igual ao original.

n_cache Número de pedidos de leitura do ficheiro que se encontrava no directório "cache".

n_ftp Número de pedidos de leitura do ficheiro que originaram uma ligação com a máquina remota, por este não se encontrar na "cache" ou a cópia existente não ser válida.

Uma vez que o ficheiro é consultado e alterado frequentemente optou-se por um formato *dbm* em vez de texto, pois torna os acessos muito mais rápidos.

4.4 Validade da informação da "cache"

A informação contida na "cache" deve retratar de forma precisa o que se passa com a respectiva estrutura de ficheiros original, tendo que ser visíveis localmente as alterações efectuadas na árvore original. Assim sendo, a "cache" não pode ser considerada válida indefinidamente, sendo necessário proceder a actualizações periódicas.
Não existe forma automática do programa ser notificado sempre que ocorram alterações nos dados originais. É assim forçoso adoptar uma política de validade, que permita manter a coerência da informação local. Considerar toda a informação desactualizada sempre que se processa um pedido não é a melhor abordagem, pois provocaria uma ligação sistemática à máquina remota, gorando um dos objectivos do programa. É assim necessário especificar a periodicidade com que a avaliação é feita.
A solução adoptada foi o estabelecimento da validade para os dados locais, imposta pelos parâmetros *update* e *expire* indicados no ficheiro de configuração `.config`.

São referidos seguidamente os procedimentos a efectuar para validar a informação contida em directórios e ficheiros.

4.4.1 Listagem do conteúdo de directórios

Quando é solicitada a listagem do conteúdo de um directório cuja informação tenha sido previamente obtida, o programa tem que verificar a sua validade. Para isso obtém a data da última alteração efectuada ao ficheiro `.list` contido no correspondente directório da "cache", adiciona-lhe o valor change_d (correspondente ao valor referido pelo parâmetro *update* no ficheiro `.config`) e verifica se ultrapassa a data actual. Em tal caso, conclui-se que a informação pode não estar actualizada. É então efectuada uma ligação à máquina remota de forma a obter os dados necessários à actualização da informação local. É construído um novo ficheiro `.list` e procedem-se às seguintes verificações:

- Se existirem novos directórios são criados localmente e introduzidas na estrutura `Entry` (lista mantida em memória com toda a informação da estrutura de ficheiros da "cache") as entradas correspondentes.

- Se algum directório deixou de existir é removida da estrutura `Entry` a entrada correspondente e as entradas de todos os ficheiros e subdirectórios que o compunham, caso existam. A árvore de directórios subjacente é movida para outra com o mesmo nome acrescido de .RMD, sendo posteriormente removida. A remoção é processada quando termina a avaliação do directório sendo lançado um processo para a efectuar.

- Se existirem novos ficheiros é inserida a entrada correspondente na estrutura `Entry`.

- Se algum ficheiro deixou de existir é removida a entrada da estrutura `Entry` correspondente. Caso exista uma cópia na "cache", esta é removida e acrescentada uma notificação do ocorrido ao ficheiro de *log* `expire`.

- Se ocorrer alguma alteração nos dados relativos a um ficheiro, nomeadamente a sua data, tamanho ou *checksum*, processa-se à actualização da entrada correspondente e à remoção da cópia mantida na "cache", caso exista.

Caso não seja possível obter a informação necessária à actualização, devido à impossibilidade temporária de estabelecer ligação com a máquina remota, verifica-se se a existência do ficheiro `.list` ultrapassa o valor referido em expire_d (corresponde ao valor referido pelo parâmetro *expire* no ficheiro `.config`). Se não ultrapassar considera-se a informação contida na "cache", válida. Caso contrário, a informação sobre o directório considera-se expirada. O conteúdo do directório é removido da "cache" e as entradas na estrutura `Entry` dos ficheiros e directórios filhos são removidas. O utilizador recebe como resposta ao seu pedido uma mensagem de erro.

4.4.2 Ficheiros

Quando um utilizador pede para ler um ficheiro que já se encontra na "cache" é necessário verificar se esta cópia é válida, isto é, se é igual ao ficheiro original. Para isso é obtida a data de criação da cópia local, que se encontra guardada no ficheiro *dbm* `.info`, adiciona-se-lhe cache_d e verifica-se se o valor resultante excede a data actual. Se exceder, é efectuada uma ligação à máquina remota para se obterem os dados sobre o ficheiro em questão.
O tipo de avaliação que é feita para verificar a validade da cópia local depende do tipo de serviço fornecido pela máquina remota. São considerados os casos em que é fornecido o *checksum* e os casos em que o servidor de ftp não implementa esta possibilidade.

Sempre que não seja possível estabelecer a ligação à máquina remota consideramos o ficheiro local actualizado se a sua permanência na "cache" não ultrapassar o valor indicado em expire_d. Se este valor tiver sido atingido, o ficheiro local é removido e o pedido do utilizador inviabilizado, recebendo este como resposta uma mensagem de erro.

4.4.2.1 Md5

Uma "*one-way hash function*" [Sch96] é uma função que opera sobre mensagens M de tamanho arbitrário e produz um valor *hash* h de tamanho fixo, tal que

$$h = H(M)$$

Existem várias funções que pegam em entradas de tamanho arbitrário e geram saídas de tamanho fixo, mas as *one-way hash function* têm características adicionais que as tornam *one-way*.

- Dado M é fácil calcular h.
- Dado h é difícil calcular M tal que H(M)=h.
- Dado M é difícil encontrar outra mensagem M' tal que H(M)=H(M').

O objectivo destas funções é fornecer uma "impressão digital" de M que seja única.

MD5 é uma *one-way hash function* que produz um valor de 128-bits. Se for aplicada a um ficheiro gera um valor *checksum* que é único. É difícil encontrar outro ficheiro cuja aplicação de MD5 dê o mesmo valor. Deste modo podemos de forma segura garantir que dois ficheiros são iguais se tiverem o mesmo *checksum MD5*.
Tendo em conta as características mencionadas considerou-se o *checksum Md5* a forma mais segura de verificar se um ficheiro foi alterado.

A última versão do ftpd (*daemon ftp*) permite obter o *checksum* de um ficheiro recorrendo a dois métodos: CRC e MD5. É enviado ao servidor um pedido de definição do método a utilizar

SITE CHECKMETHOD md5

e pedido o *checksum* do ficheiro utilizando esse método de cálculo.

SITE CHECKSUM filename

O *checksum* do ficheiro local é obtido a partir do correspondente registo incluído no ficheiro *dbm* `.info`. Se o seu valor for diferente do enviado pelo servidor ftp , conclui-se que ocorreram alterações no ficheiro, estando deste modo a cópia local desactualizada. É assim necessário proceder a uma retransmissão do ficheiro.

4.4.2.2 Data e tamanho do ficheiro

Quando não for possível obter o *checksum Md5* do ficheiro, a validação é baseada na sua data e tamanho. Se algum destes factores tiver sido alterado, concluímos que a cópia local não é igual ao ficheiro original. O ficheiro local é removido e procede-se à sua retransmissão.

4.5 Erros

No decorrer da execução do programa podem surgir várias situações que originem erros, podendo estes interromper a sua normal execução ou impedir a satisfação de pedidos.

Analisemos situações de erro que inviabilizem ou deturpem o funcionamento do `rfs`.

Ficheiro .config não existe. O programa pára pois fica sem referência dos directórios que está a manter e da configuração inicial básica.

Erros no ficheiro .config. Se existirem erros sintácticos no ficheiro ou faltar referir algum parâmetro o programa pára.

Erros no ficheiro .info. Se este ficheiro for danificado torna-se impossível obter dados que permitam avaliar correctamente a validade dos ficheiros contidos na "cache". Se por exemplo, não for possível obter a data da última actualização de um ficheiro, não podemos ajuizar se este expirou. Neste caso consideramos que atingiu o valor change_d e é necessário verificar se algo mudou no ficheiro original.

Erros na criação de directórios ou ficheiros. O pedido do utilizador não é satisfeito se não for possível criar a cópia local ou o ficheiro `.list`.

Durante o estabelecimento da ligação com a máquina remota podem ocorrer os seguintes erros:

Nome inválido. O nome da máquina está mal escrito ou o `dns` não está operacional.

Erro na ligação. Não existe conectividade ou a máquina remota não tem o serviço ftp activado.

Utilizador inválido. A máquina remota não permite a entrada por ftp do utilizador, por exemplo anónimo, pois atingiu o número máximo destes utilizadores.

Erro na ligação de dados. O ftp usa 2 canais de comunicação. Um de controlo por onde são transmitidos os comandos e os erros e outro de dados. Quando é estabelecida uma ligação ftp está apenas presente o canal de controlo. Para transmitir um ficheiro ou a listagem do conteúdo de um directório é necessário estabelecer uma nova ligação noutro *port*, através da qual são enviados os dados.

Torna-se em qualquer dos casos impossível satisfazer o pedido, recebendo o utilizador a indicação da ocorrência de um erro. Recebe a mensagem *"Resource temporarily unavailable"* correspondente ao erro EAGAIN (definido no ficheiro linux/errno.h).

Podem também ocorrer erros durante a ligação.

Erros de leitura Não consegue obter uma resposta da máquina remota.

Erros de escrita Não consegue enviar os pedidos à máquina remota.

Resposta errada Não recebe a resposta adequada ao pedido efectuado.

Estes erros ocorrem devido à interrupção da ligação ao servidor ftp. O utilizador recebe como resposta a mensagem *"Interrupted function call"* correspondente ao erro EINTR (definido no ficheiro linux/errno.h).

Todos os erros mencionados originam uma mensagem informativa que é acrescentada ao ficheiro de *log* `error`, com excepção dos que impedem o arranque do programa, sendo neste caso a mensagem enviada para o `stderr`.

4.6 Gestão do espaço de disco

O espaço de disco ocupado pela "cache" deve ser verificado regularmente de forma a evitar que o programa pare por falta de espaço de disco e que sejam danificados ficheiros.

Consideramos duas situações distintas:

- A "cache" encontra-se num sistema de ficheiros utilizado exclusivamente por si.
- O sistema de ficheiros onde se situa a "cache" é partilhado com outros ficheiros do sistema.

A forma como é gerido o espaço de disco tem abordagens diferentes consoante o caso em que se insere.
No primeiro caso o cálculo do espaço livre recorre à chamada do sistema `statfs` que indica o número de blocos livres existentes no sistema de ficheiros.
No segundo caso tem que ser indicado (no ficheiro de configuração) o espaço total que a "cache" pode ocupar, uma vez que partilha o disco com outros ficheiros. Cabe ao administrador do sistema garantir que esse espaço efectivamente existe evitando que o crescimento da "cache" ponha em risco o funcionamento do sistema e que esta tenha realmente disponível o espaço que lhe foi atribuído. O programa tem que ter sempre presente o espaço de disco que a "cache" está a ocupar. Este valor, ao contrário do que sucedia no primeiro caso, tem que ser calculado pelo próprio `rfs`. Quando o programa arranca é somado o tamanho de todos os ficheiros que constituem a "cache"; este valor é actualizado sempre que se verificarem alterações na estrutura da "cache". Foi considerada a hipótese de manter este valor guardado num ficheiro, evitando assim o seu cálculo sempre que o programa é lançado. Se ocorrer uma falha no sistema ou uma paragem súbita do programa este ficheiro pode ser corrompido não apresentando assim o valor correcto. Uma vez que esta abordagem não garante fiabilidade preferimos a

solução adoptada, pois os custos do tempo de execução do cálculo do espaço ocupado não são consideráveis.

Em qualquer dos casos tem que ser mantido algum espaço livre para os ficheiros de *log*. No ficheiro de configuração `.config` é referido o espaço de disco disponível no caso da "cache" possuir um sistema de ficheiros só para si e duas margens *lo_space* e *hi_space*.
Sempre que o espaço livre é inferior a *lo_space* executa-se um processo de "limpeza da cache" até recuperar o espaço *hi_space*. Esta análise é feita antes da criação de um ficheiro de forma a garantir que existe espaço suficiente para o guardar.

A verificação feita está retratada na figura 4.5.

```
/* verifica se existe espaço para um ficheiro de tamanho size */
checksize(int size)
{

  struct statfs buff;
  int space;
  int len;

  if(Partition) {
    statfs(cachedir,&buff);
    space = buff.f_bavail;   /* blocos livres */
   } else /* se não possuir uma partição só para si */

    /* max_space é o espaço máximo que pode ocupar, em Mb
        total_space é o número de bytes ocupado */

    space = (max_space * 1000) - (total_space / 1024);

  len = space - size/1024;

  if ( len < lo_space) { /* nao pode transferir sem remover */
     return 1;
  } else if (len < hi_space){/* pode transferir e tem que remover */
    return 2;
  }
  return 0;
}
```

Figura 4.5: Rotina `checksize`

Conforme já foi referido, sempre que não haja espaço suficiente é necessário remover ficheiros para libertar espaço. São removidos aqueles que não foram acedidos há mais tempo e no caso de existirem vários com datas de acesso iguais, é removido o que tiver menor tamanho, minimizando assim os custos de uma futura retransmissão.
A rotina de limpeza constroi uma lista dos ficheiros que formam a “cache”, ordenada por ordem crescente da data de acesso, com base na informação contida no ficheiro `.info`. Se este ficheiro tiver sido danificado, a informação necessária é obtida percorrendo recursivamente o directório “cache” de forma a obter a data da última alteração de cada ficheiro.
Os ficheiros que não são acedidos há mais tempo vão sendo removidos até se recuperar o espaço livre indicado por *hi_space*.

Capítulo 5

Implementação

Neste capítulo são referidos pormenores da implementação do `rfs`. É explicada a forma como se processa a comunicação com o kernel e expostas as operações implementadas. É também indicado o modo como são estabelecidas as ligações a servidores ftp, para obtenção dos dados necessários à manutenção da informação local.

5.1 Comunicação com o kernel

A comunicação entre o kernel e o `rfs` é feita através de *pipes*, por onde são transmitidos ao programa pedidos e são enviadas as respectivas respostas.
Os pedidos do sistema de ficheiros virtual são enviados ao programa `rfs` sob a forma de mensagens constituídas por um cabeçalho e por dados opcionais. O programa recorre à função `decode_up_preamble` incluída na biblioteca userfs_types.h para recuperar os dados contidos no cabeçalho.
Assim, após a execução de

decode_up_preamble(up_preamble *sp, unsigned char *buf)

a estrutura sp é preenchida com a seguinte informação:

version Versão do protocolo.

seq Número de identificação do pedido. A resposta deve referir este valor para que se saiba a que pedido corresponde.

op Tipo de operação.

isreq Tipo de mensagem. Pode ser um pedido – UP_REQ – ou uma interrogação – UP_ENQ.

size Indica o tamanho dos dados que se seguem.

O valor sp->size é fundamental para averiguar a existência de mais informação enviada pelo kernel. Se for diferente de zero significa que existem dados adicionais com o tamanho referido, que complementam o pedido, efectuando-se de seguida a sua leitura. É necessário verificar igualmente se o tipo de operação indicado em sp->op é implementado pelo programa, sendo retornado o valor ENOSYS (definido em linux/errno.h) caso não o seja. A descodificação dos dados depende do seu tipo, reflectindo este a operação solicitada. Para executar a descodificação são usadas funções decode_upp_xxx_s, onde xxx indica o tipo de operação.

Na figura 5.1 está referida a rotina de leitura dos pedidos do kernel.

A resposta é formada por uma mensagem de tipo "UP_RESP" constituído pelo cabeçalho e pelos dados resultantes da satisfação do pedido. O número – seq – que a identifica, tem de ser igual ao do pedido correspondente. O cabeçalho, similar ao enviado pelo kernel com o pedido, contém um campo extra – errno. Se ocorrer algum erro durante a execução do pedido é indicado o seu valor neste campo. Caso contrário, o campo *errno* contém o valor zero.
O cabeçalho é codificado através da chamada à função `encode_upp_repl` e o resto dos dados são codificados recorrendo a funções do género `encode_upp_xxx_r`, onde xxx refere o tipo de operação em questão.

A figura 5.2 refere a rotina de envio de respostas ao kernel.

5.2 Interrogações

Após o início do programa `rfs`, o kernel envia várias mensagens do tipo "Enquiry" para averiguar quais as operações suportadas pelo programa. Este deve responder com mensagens de resposta normais, contendo o campo de erro o valor 0 se o tipo de operação for suportada ou ENOSYS (definido em linux/errno.h) caso não o seja. Não é necessário executar nenhuma operação adicional, nem enviar dados na resposta.

As operações suportadas pelo programa são as seguintes:

- mount - pedido do *inode* da raiz do sistema de ficheiros.
- readdir - leitura do conteúdo de um directório.
- iread - obtenção dos dados sobre o *inode*.
- open - abertura de um ficheiro.
- close - fecho de um ficheiro.
- read - leitura de um ficheiro.

```
int
readpkt(int fd, up_preamble *pre, unsigned char *buf)
{
  int presz = sizeof_up_preamble(pre);
  int rd;
  char pbuf[presz];

  memset(pbuf,0,sizeof(pbuf));

  rd = myread(fd,pbuf,presz);

  decode_up_preamble(pre,pbuf);

  if (pre->isreq != UP_ENQ && pre->isreq != UP_REQ) {
    fprintf (stderr, "pre->isreq = %d\n", pre->isreq);
    return -1;
  }

  if (pre->version != UP_VERSION)
  {
   fprintf (stderr, "Version mismatch: us=%d them=%d\n",
           UP_VERSION, pre->version);
   return -1;
   }

  if (pre->size != 0)
  {
     bzero(buf,strlen(buf));
     if (myread (fd, buf, pre->size) != pre->size)
     {
        perror ("readpkt, body read");
        return -1;
    }
  }

 return 1;
}
```

Figura 5.1: Rotina de leitura dos pedidos do kernel

```
static int
writepkt (int fd, upp_repl *repl, unsigned char *buf)
{
  int replsz = sizeof_upp_repl (repl);
  unsigned char pbuf[replsz];
  unsigned int wr;
  unsigned char *p;

  assert (repl->version == UP_VERSION);
  assert (repl->isreq == UP_REPL);

  p = encode_upp_repl (repl, pbuf); /* codifica o cabeçalho */

  assert (replsz == p - pbuf);

  wr = mywrite (fd, pbuf, replsz);  /* envia o cabeçalho */
  if (wr != replsz)
  {
    perror ("writepkt, header write");
    return -1;
  }


  if (repl->size > 0 && repl->err_no == 0)
  { /* envia o corpo da mensagem */
    if (mywrite(fd, buf, repl->size) != repl->size)
    {
      perror ("writepkt, body write");
      return -1;
    }


 return 1;
}
```

Figura 5.2: Rotina de escrita das respostas a enviar ao kernel

- lookup - verificação da existência de um ficheiro.
- readlink - obtenção do nome para onde o *link* aponta.

5.3 Operações implementadas

Quando um utilizador tenta aceder ao directório onde é montado o sistema de ficheiros *userfs*, o kernel envia vários pedidos ao processo `rfs` de forma a obter a informação necessária à elaboração da resposta. Os pedidos efectuados dependem do tipo de pergunta solicitada pelo utilizador.

Consideremos o sistema de ficheiros montado no directório ~ftp/pub/Linux e a "cache" situada em /usr1/cache.

- Se o utilizador executar o comando

  % ls -l ~ftp/pub/Linux

  o kernel envia os seguintes pedidos ao `rfs`

  1. up_open /usr1/cache
  2. up_readdir /usr1/cache 0
  3. up_readdir /usr1/cache 1
  4. up_readdir /usr1/cache n , onde n é um inteiro entre 2 e o número de ficheiros existentes no directório+2.
  5. up_lookup
  6. up_iread
  7. os pedidos 4, 5 e 6 são repetidos para todos os ficheiros que constituem o directório.
  8. up_close /usr1/cache

- Se o utilizador solicitar acesso a um ficheiro, executando o comando

  % cat ~ftp/pub/Linux/debian/README

  o kernel envia os seguintes pedidos

  1. up_lookup /usr1/cache debian
  2. up_iread /usr1/cache/debian
  3. up_lookup /usr1/cache/debian README
  4. up_iread /usr1/cache/debian/README
  5. up_open /usr1/cache/debian/README

6. up_read /usr1/cache/debian/README
7. up_close /usr1/cache/debian/README

O programa `rfs` ao receber um pedido relativo à operação de tipo up_xxx, executa a função do_xxx para obter a resposta.
Os detalhes de implementação das funções do_xxx são expostos nas subsecções seguintes.

5.3.1 do_mount

O primeiro pedido enviado pelo kernel assim que o programa é iniciado, é `up_mount`. Este pedido, que tem de ser obrigatoriamente implementado pelo programa `rfs`, retorna um apontador para o *inode* da raiz do sistema de ficheiros.

O kernel espera receber como resposta, o valor codificado da estrutura `upp_mount_r`, que é constituída unicamente por um *handle*. É enviado como *handle* um apontador para a estrutura `Entry` onde se encontram guardados os dados do directório "cache".

A figura 5.3 ilustra a função `do_mount` implementada.

```
do_mount (unsigned char *buf, upp_repl *repl)
{
   upp_mount_r snd;

   snd.root.handle = (Ulong) root;
   repl->size = encode_upp_mount_r(&snd, buf) - buf;

   return 0;
}
```

Figura 5.3: Função `do_mount`

5.3.2 do_iread

Para obter toda a informação relativa a um *inode*, o kernel envia um pedido `up_iread` e um apontador para o ficheiro em questão.
Inicialmente o programa descodifica os dados enviados,

decode_upp_iread_s(upp_iread_s rcv, char *buf) obtendo deste modo um *handle* para o ficheiro cujo *inode* se pretende. Note-se que todos os *handles* são apontadores para a estrutura `Entry`, referindo a entrada sobre a qual se deseja informação.

ent = (Entry *) rcv.handle.handle;

Como resposta é enviada a estrutura `upp_iread_r` constituída por

- mode - tipo de ficheiro e suas permissões.
- nlink - número de *links*.
- uid - número de identificação do dono do ficheiro.
- gid - número de identificação do grupo a que pertence.
- size - tamanho do ficheiro.
- atime - data do último acesso ao ficheiro.
- mtime - data da última modificação.
- ctime - data da última alteração de estado.
- rdev - tipo de *device*.
- blcksize - tamanho dos blocos.
- blocks - número de blocos ocupados.

Os campos desta estrutura são preenchidos com a informação contida na estrutura `Entry` correspondente ao ficheiro em questão. Seguidamente os dados da resposta são codificados recorrendo à função

encode_upp_iread_r(upp_iread_r snd, char *buf)

5.3.3 do_lookup

O pedido `up_lookup` é enviado pelo kernel a fim de verificar a existência de um ficheiro dentro de um directório.
O nome do ficheiro e o *handle* do directório pai são obtidos após a descodificação dos dados enviados, através da função

decode_upp_lookup_s(upp_lookup_s rcv, char *buf)

O *handle* enviado é um apontador para a estrutura `Entry` do directório pai. Se não existir na "cache" informação sobre o conteúdo do directório, é efectuada uma ligação à máquina remota a fim de a obter.

Para averiguar a existência do ficheiro indicado no pedido, é percorrida a lista dos "ficheiros filhos". Se não for encontrado nenhum ficheiro com esse nome é retornado o valor ENOENT (definido em linux/errno.h). Caso contrário e se o nome referir um ficheiro existente na "cache", procede-se à verificação da sua validade, sendo estabelecida uma ligação à máquina remota para garantir a actualização da cópia local, se tal for necessário.
Na primeira versão do programa, esta verificação era efectuada na função `do_open`, que é referida na subsecção seguinte. Se um utilizador executasse o comando

% ls -l filename

e se a informação mantida na "cache" sobre o ficheiro *filename* estivesse desactualizada, como por exemplo o seu tamanho, as alterações não surgiam na resposta. O kernel obtém os dados sobre o *inode*, como resposta ao pedido `up_iread`. Como a verificação da validade dos dados era feita posteriormente, durante a execução de `do_open` o kernel não era informado das alterações. Efectuando a validação durante `do_lookup` estas serão repercutidas no kernel pois o pedido `do_iread` é posterior a este.

Como resposta ao pedido, é enviado o *handle* do ficheiro codificado, após execução da função

encode_upp_lookup_r(upp_lookup_r snd, char *buf)

5.3.4 do_open

Sempre que um utilizador solicita a execução da chamada ao sistema `open`, o kernel envia um pedido do tipo `up_open`. Juntamente é referido o *handle* do ficheiro e as credenciais do processo que originou o pedido, incluindo o número que identifica o utilizador e os grupos a que este pertence.

Após a descodificação dos dados resultante da execução da função

decode_upp_open_s (upp_open_s rcv, char *buf)

é obtida a estrutura `Entry` correspondente ao *handle* indicado e são feitas as seguintes verificações:

- Se for um directório retorna 0.
- Se for um ficheiro, tenta abrir o correspondente ficheiro da "cache".

- Se existir uma cópia do ficheiro na "cache", abre-a e coloca na respectiva estrutura `Entry`, o descritor do ficheiro obtido. Uma vez que o programa mantém um registo do número de acessos a um ficheiro, é necessário actualizar a correspondente entrada do ficheiro *dbm*.`info`, incrementando o valor do campo *n_cache*.
- Se não existir uma cópia do ficheiro na "cache", procede do seguinte modo:
  - Verifica se existe espaço em disco suficiente para alojar uma cópia do ficheiro.
  - Lança um processo para transferir o ficheiro, efectuando uma ligação ftp à máquina remota.
  - Espera que se inicie a transmissão do ficheiro.
  - Abre o ficheiro local, colocando na correspondente estrutura `Entry` o descritor do ficheiro obtido.

Note-se que não é necessário proceder à verificação da validade do ficheiro, pois esta foi efectuada previamente, durante a execução da rotina `do_lookup`. Um pedido `up_open` é sempre precedido por um pedido `up_lookup`, de forma a obter o *handle* do ficheiro.

5.3.5 do_readdir

Para obter a listagem do conteúdo de um directório, o kernel envia vários pedidos `up_readdir`, indicando o *handle* do directório que pretende ler e um *offset*. O *offset* refere a posição do ficheiro dentro do directório, sobre a qual se pretende questionar o sistema de ficheiros.

A fim de descodificar os dados enviados é executada a função

decode_upp_readdir_s(upp_readdir_s rcv, char *buf)

A resposta inclui, o nome e o *handle* do ficheiro situado na posição solicitada e um *offset* que é utilizado pelo kernel para calcular o novo *offset* a enviar no pedido `up_readdir` seguinte. Como as entradas no directório não correspondem a ficheiros existentes fisicamente no disco, como acontece noutros sistemas de ficheiros, considera-se o *offset* de uma entrada do directório como tendo o valor 1. Deste modo o seu valor é igual ao do *offset* do pedido anterior acrescido de uma unidade.
Quando for atingido o final do directório, é retornado um *offset* 0.

Quando o *offset* indicado no pedido for zero, é verificada a consistência da informação mantida na "cache" sobre o directório em questão. Se tiver sido atingida a sua data de validade, é efectuada uma ligação ftp à máquina remota, de forma a obter o conteúdo do directório original e de proceder à actualização da informação local. É retornada a entrada ".".

Se o *offset* do pedido for 1 é retornado o nome ".." e o *handle* correspondente à estrutura `Entry` do directório pai.

Para outros valores do *offset* é percorrida a lista dos "ficheiros filhos" até encontrar a entrada correspondente ao *offset* indicado.

```
while(ent && rcv.off--)
     ent=ent->next;
```

Os dados da resposta são codificados usando a função

encode_upp_readdir_r(upp_readdir_r snd, char *buf)

5.3.6 do_read

O pedido `up_read` é enviado pelo kernel quando se pretende ler um ficheiro. São indicados como argumentos, a posição no ficheiro a partir da qual se pretende efectuar a leitura – `off` – e o número de bytes a ler.

Após a descodificação dos dados, através da execução da rotina

decode_upp_read_s(upp_read_s rcv, char *buf)

é obtido o apontador para a estrutura `Entry` correspondente e o descritor do ficheiro previamente aberto durante a execução do pedido `up_open`. Para efectuar a leitura do número de bytes indicado, é necessário situar-se no ficheiro na posição referida em *off*.

```
ret = lseek(fd, rcv.off, SEEK_SET);
r = read(fd,buf,rcv.size);
```

O ficheiro que se está a ler pode estar a ser copiado a partir da máquina remota nesse momento. Se os dados pretendidos ainda não existiram no disco tem que aguardar pela sua transferência.

Como resposta ao pedido, são enviados os bytes lidos e o seu tamanho. Se o final do ficheiro tiver sido atingido, é enviado o valor 0.

```
upp_read_r snd;

snd.data.nelem = r;
snd.data.elems = buf;

encode_upp_read_r(&snd, mbuf);
```

5.3.7 do_readlink

Para obter o nome do ficheiro para onde está a apontar o *link*, o kernel envia o pedido `up_readlink`.
Para obter o *handle* enviado juntamente com o pedido, é necessário descodificar os dados, recorrendo à função

decode_upp_readlink_s(upp_readlink_s rcv, char *buf)

É usado seguidamente o apontador para a estrutura `Entry` assim obtido e lido o campo *link* que contém o nome do ficheiro para onde está a apontar.

```
upp_readlink_r snd;
snd.name.nelem= strlen(ent->link);
snd.name.elems = ent->link;

encode_upp_readlink_r(&snd,buf);
```

5.3.8 do_close

Quando a leitura de um ficheiro termina, é efectuado o pedido `up_close` para o fechar. Se o apontador indicado no pedido referir um ficheiro, é fechado o descritor de ficheiros que foi previamente aberto durante a execução de `do_open` e retirada da correspondente estrutura `Entry` a sua referência.

```
close(ent->data.file.fd);
ent->data.file.fd=-1;
```

Não é enviado nenhum dado com resposta.

5.4 Comunicação com os servidores ftp

Sempre que é necessário transferir um ficheiro para o disco local ou obter informação acerca do conteúdo de um directório remoto, são estabelecidas ligações aos servidores ftp que possuem o repositório de software em que estamos interessados.
Para implementar as rotinas do programa que permitem efectuar estas ligações foi usada a descrição do protocolo ftp em rfc 959 [PR], com o intuito de obter a descrição das mensagens recebidas e enviadas pelo servidor, da ordem por que devem ser executadas e dos erros que podem ocorrer.

A comunicação entre o servidor ftp e o cliente é um diálogo constante. Assim, o utilizador envia comandos ftp e o servidor responde de forma a manter o cliente informado acerca do estado de execução dos pedidos. As respostas são constituídas por números de 3 dígitos e por texto. A informação contida nestes números é suficiente para identificar a resposta, não sendo necessário examinar o texto que a constitui.
Cada um dos dígitos da resposta tem um significado especial. O primeiro indica se o pedido pode ser ou não atendido ou se a informação indicada é insuficiente. Se o cliente quiser saber qual o tipo de erro ocorrido, examina o segundo dígito. O terceiro contém informação mais precisa sobre os erros.

O primeiro dígito pode ter 5 valores possíveis:

1yz A acção pedida está a ser iniciada. Indica que o pedido foi aceite.

2yz A acção pedida foi completada com sucesso. Pode ser enviado outro pedido.

3yz O comando foi aceite, mas a acção pedida está suspensa à espera de mais informação, que deve ser enviada pelo cliente.

4yz O comando não foi aceite e a acção pedida não foi efectuada. A condição que originou o erro é temporária, podendo o pedido ser solicitado de novo.

5yz O comando não foi aceite e a acção pedida não foi efectuada. O cliente não deve repetir o pedido.

O segundo dígito tem o seguinte significado, dependendo do seu valor:

x0z A resposta indica a existência de erros sintácticos, ou de comandos não implementados.

x1z Os dados constituem respostas a pedidos de informação, como por exemplo a pedidos de ajuda “help”.

x2z Estas respostas referem-se às ligações de dados ou de controlo.

x3z Os dados constituem respostas a pedidos de *login*.

x4z Não especificado.

x5z Estas respostas estão relacionadas com o sistema de ficheiros. Podem indicar falta de espaço livre em disco ou a inexistência do ficheiro solicitado.

O terceiro dígito, fornece informação mais detalhada acerca do que é referido pelo segundo dígito.

O texto associado com cada resposta não é obrigatório, mas sim recomendado, podendo o seu conteúdo ser alterado de acordo com o comando com que está associado. Quando o texto ocupa várias linhas, o cliente deve ser informado dessa situação, de forma a saber quando a resposta acaba. A primeira linha terá, neste caso, um formato especial indicando que mais linhas se seguem. Começará pelo código da resposta, seguido imediatamente por "-". A última linha refere apenas o mesmo código.

```
Exemplo:

    123-primeira linha
    segunda linha
    123
```

5.4.1 Estabelecimento da ligação ao servidor ftp

O servidor ftp aceita pedidos ftp no *port* 21. É necessário estabelecer um canal de comunicação entre as máquinas local e remota, através do qual serão transmitidos os comandos de controlo e os erros ocorridos.
São assim efectuados os seguintes passos:

- É criado um `socket` [Ste90] de tipo Sock_Stream que usa a família de protocolos Af_Inet (Arpa Internet Protocols). Este tipo de *socket* permite ligações nos dois sentidos, sequenciais e fiáveis, que garantem a não existência de duplicação ou perda de dados. Se a transmissão de dados não for possível num período de tempo razoável, a ligação é quebrada e o utilizador é notificado da ocorrência do erro ETIMEDOUT (definido em linux/errno.h).
- A transmissão de dados só é possível se o *socket* estiver no estado "conectado". Deste modo, é estabelecida a ligação ao servidor no *port* 21.

Após o estabelecimento da ligação, o servidor envia uma "mensagem de boas vindas", com o código 220. O cliente tem de esperar por esta mensagem, não podendo enviar nenhum pedido antes da sua recepção. Se o servidor não poder aceitar pedidos, responde com o código 120.

Se não for possível estabelecer a ligação ao servidor ftp, o pedido do utilizador pode ficar inviabilizado. Neste caso, ele recebe como resposta, a mensagem de erro "Resource temporarily unavailable".

5.4.2 Login

Um servidor ftp permite acesso ao seu sistema de ficheiros mediante a identificação do utilizador, sendo necessário indicar o seu nome e senha de acesso. A maior parte destes servidores permitem acesso anónimo, não sendo assim necessário possuir uma conta na máquina. Neste caso deve ser enviado como senha de acesso, o endereço electrónico do utilizador. O acesso fica restrito ao directório público de ftp.

É assim enviado o pedido

USER ftp

Se o pedido for aceite o servidor envia uma das seguintes respostas:

- 230 Guest login ok. Access restrictions apply.
- 331 Guest login ok, send your complete e-mail address as password.

Se a resposta referir o código 230 não é necessário enviar a senha de acesso. Caso contrário é feito o pedido

PASS rfs@domain

onde "domain" refere o domínio da máquina local.

Se a acção for bem sucedida o servidor envia a resposta

230 Guest login ok. Access restrictions apply.

5.4.3 Alteração do directório corrente

Quando é estabelecida uma ligação ftp como utilizador anónimo, tem-se acesso ao directório público ftp do servidor, sendo este o directório corrente. É possível entrar nos seus subdirectórios alterando o directório corrente.
É assim enviado ao servidor o pedido

CWD dir

que pretende alterar o directório corrente para o indicado em "dir".

As respostas possíveis a este pedido são:

- 250 CWD command successful.
- 500 CWD: command not understood.
- 501 Sintaxe error.
- 502 CWD: command not implemented.

- 421 Service not available.
- 530 Please login with USER and PASS.
- 550 No such file or directory.

Se a mensagem recebida não referir o código 250 e se não for possível satisfazer o pedido do utilizador, este recebe uma mensagem de erro.

5.4.4 Canal de transferência de dados

Uma ligação ftp necessita de 2 canais de comunicação: um para transferência de comandos de controlo e de erros e outro para envio de dados. Ao estabelecer uma ligação ao *port* 21 do servidor ftp, fica apenas definido o canal de controlo. Assim sendo, é preciso criar posteriormente o canal de dados.

Existem duas formas de estabelecer uma ligação para transferência de dados:

- O cliente envia ao servidor a indicação do *port* a ser usado na conexão de dados. O servidor tem que estabelecer uma ligação ao cliente no *port* indicado.
- O servidor envia ao cliente a indicação do *port* a que o cliente deve conectar-se. O servidor aguarda que o cliente estabeleça a ligação, em vez de ser o próprio servidor a faze-lo, como sucedia no caso anterior.

Foi adoptado o segundo método, deixando-se ao critério do servidor a escolha de um *port*.

Com essa finalidade é enviado o pedido

PASV

O servidor pode enviar como resposta, uma das seguintes mensagens:

- 227 Entering passive mode
- 500 PASV: command not understood.
- 501 Sintaxe error.
- 502 PASV: command not implemented.
- 421 Service not available.
- 530 Please login with USER and PASS.

Se o pedido for aceite, a resposta refere o endereço do computador e o *port*, a que o cliente se deve conectar. Estes dados são o resultado da concatenação do endereço internet de 32-bits da máquina e do número de 16-bits do *port*, partidos em campos de 8-bits, sendo o valor de cada campo transmitido como um número decimal.

É assim criado um *socket* de tipo Sock_Stream e criada uma ligação à máquina e ao *port* enviados pelo servidor.

Se não for possível estabelecer a ligação de dados, a ligação ao servidor ftp é terminada. Se por este motivo não for possível efectuar o pedido do utilizador, este recebe uma mensagem de erro.

5.4.5 Listagem de directórios ou ficheiros

Para obter a listagem do conteúdo de um directório ou informação acerca de um ficheiro, é enviado ao servidor ftp o pedido

LIST name

Se "name" indicar o nome de um directório, o servidor deve enviar a lista dos ficheiros que o constituem. Se indicar um ficheiro, o servidor deve enviar a informação corrente sobre esse ficheiro. Em qualquer dos caso, os dados são enviados através do canal de comunicação de dados.

O servidor envia através do canal de controlo a resposta ao pedido efectuado. A mensagem enviada é uma das seguintes:

- 125 Data connection already open; transfer starting.
- 150 Opening ASCII mode data connection for /bin/ls.
- 450 File unavailable.
- 501 Sintaxe error.
- 502 LIST: command not implemented.
- 421 Service not available.
- 530 Please login with USER and PASS.

A mensagem recebida pelo programa `rfs` deve conter os códigos 125 ou 150. Caso contrário, não será possível obter os dados pretendidos. Os dados enviados pelo servidor são lidos a partir do *socket* de dados já criado.
Se "name" referir um directório, é criado o ficheiro `.list` correspondente, onde são

colocados os dados enviados pelo servidor.
Se "name" referir um ficheiro, os dados enviados pelo servidor incluem o modo, tamanho e data do ficheiro. Estes dados são usados para verificar se a cópia local está actualizada e para actualizar a informação local sobre o ficheiro.

No fim da transmissão de dados, o servidor envia pelo canal de controlo uma das seguintes mensagens:

- 250 Request file action completed.
- 226 Transfer complete. Closing data connection.
- 425 Can't open data connection.
- 426 Transfer aborted.
- 451 Request action aborted; local error in processing.

Se a resposta não incluir os códigos 250 ou 226 significa que ocorreram erros na transmissão dos dados, ficando o pedido efectuado, sem resposta.

5.4.6 Transferência de ficheiros

São descritos seguidamente os pedidos a enviar ao servidor, para efectuar a transferência de ficheiros e obter os dados que o programa deve guardar.

5.4.6.1 Data do ficheiro

A fim de obter a data em que foi efectuada a última alteração a um ficheiro, é enviado o pedido

MDTM filename

Se o servidor efectuar a acção solicitada, envia como resposta a mensagem

213 date

onde "date" indica a data pretendida no formato yyyymmddhhmmss (ano, mês, dia, hora, minutos, segundos) .

5.4.6.2 Formato dos dados

Os dados podem ser enviados pelo servidor, em formato ascii ou binário. Como não se sabe à priori qual o tipo do ficheiro, optou-se por transferir sempre os dados dos ficheiros em formato binário.

É assim enviado o pedido

TYPE I

O servidor envia como resposta uma das seguintes mensagens:

- 200 Type set to I.
- TYPE: command not understood.
- 501 Sintaxe error.
- 502 TYPE : command not implemented.
- 421 Service not available.
- 530 Please login with USER and PASS.

5.4.6.3 Transferência do ficheiro

Para obter uma cópia do ficheiro remoto, é enviado ao servidor o pedido

RETR filename

Se o pedido for aceite é enviada uma das mensagem

- 125 Data connection already open; transfer starting.
- 150 Opening BINARY mode data connection for filename

É iniciada a leitura do canal de dados, através do qual é transmitido o conteúdo do ficheiro indicado no pedido.
Quando a transmissão terminar, o servidor envia pelo canal de controlo uma das mensagens seguintes, no caso do pedido ser bem sucedido:

- 250 Request file action completed.
- 226 Transfer complete. Closing data connection.

Caso contrário, envia

- 425 Can't open data connection.
- 426 Transfer aborted.
- 451 Request action aborted; local error in processing.

A cópia do ficheiro transferido é guardada no directório "cache", sendo a sua data de criação alterada de forma a preservar a data do ficheiro original. É utilizada para esse fim, a data resultante do pedido *MDTM*.
É guardada no ficheiro *dbm* `.info` informação relativa ao ficheiro, como por exemplo a data da sua transferência.

5.4.6.4 Checksum

O programa utiliza o *cheksum md5* do ficheiro, para verificar se a cópia local está actualizada, conforme já foi referido no capítulo anterior.

É necessário enviar ao servidor ftp o pedido

SITE CHECKMETHOD MD5

indicando o método que se pretende usar no cálculo do *checksum*.

Se o servidor não implementar o comando, envia a mensagem de erro

500 SITE CHECKMETHOD: command not understood.

Neste caso, não será possível utilizar este mecanismo de averiguação da validade do ficheiro, quando tal for necessário.
No caso do servidor ter enviado como resposta a mensagem

200 OK

é pedido o *checksum* do ficheiro através do envio do pedido

SITE CHECKSUM filename

a que o servidor deve responder com a mensagem

200 checksum

O valor assim obtido, é guardado no ficheiro `.info` na linha correspondente ao ficheiro.

Capítulo 6

Administração

Neste capítulo são apresentados programas que permitem gerir o espaço em disco ocupado pela "cache" e que fornecem meios para analisar a performance do `rfs` e o número e frequência dos acessos a ficheiros aí mantidos.

6.1 Programas de gestão

A informação contida na "cache" deve ser revista periodicamente, de forma a proceder à remoção de ficheiros que não foram pedidos, evitando assim o gasto desnecessário de espaço. Este processo é efectuado automaticamente pelo `rfs`, quando o espaço livre de disco for inferior ao valor indicado no ficheiro de configuração `.config`, conforme foi referido no capítulo 4.
No entanto, sempre que se pretender, pode-se forçar a "limpeza da cache", diminuindo assim a possibilidade de despoletamento do processo de limpeza durante a execução do `rfs`, o que poderia impossibilitar a transferência de ficheiros.

Foi criado com esse fim o utilitário `clean`, que aceita um único parâmetro indicando o número de dias que um ficheiro pode permanecer na "cache" sem ser utilizado. Todos os ficheiro cuja data do último acesso exceda o valor referido, serão removidos.
O programa recorre ao ficheiro *dbm*.`info` para obter a data de acesso de cada ficheiro, constituindo assim uma lista (*actdata*) com a seguinte informação:

fname Nome do ficheiro.

actime Data em que o ficheiro foi pedido pela última vez.

A figura 6.1 refere a rotina que constitui a lista referida.

A lista é posteriormente ordenada, por ordem crescente da data de acesso. A função

```
 create_list() {

  int len =0;
  actdata **list;

  if ((db = dbm_open(IFILE,O_RDONLY,0600))==NULL) {
    fprintf(stderr,"Can't open dbm file: %s\n",strerror(errno));
    _exit(0);
  }

  for (key = dbm_firstkey(db); key.dptr != NULL;
                   key = dbm_nextkey(db)) {
    db_data = dbm_fetch(db,key);
    bcopy(db_data.dptr,&new,db_data.dsize);

    if(new.last) {
      list[len] = (actdata *)malloc(sizeof(actdata));
      strcpy(list[len]->fname,new.fname);
      list[len]->actime = new.actime;
      len++;
    }
  }
  dbm_close(db);
}
```

Figura 6.1: Rotina

qsort((char *)list, len, sizeof(actdata *), cmp);

ordena a lista *list* constituída por *len* elementos, utilizando para esse fim a função `cmp`, que compara duas datas e retorna os valores -1, 0 ou 1, conforma a primeira data seja menor, igual ou maior do que a segunda.

A lista ordenada assim obtida, é percorrida, efectuando-se os seguintes cálculos para cada ficheiro aí referido:

- Subtrai-se à data actual o valor da data em que o ficheiro foi acedido pela última vez. Note-se que as datas indicam o número de segundos desde 1970.
- Converte-se em dias o valor resultante, dividindo-o por 60*60*24.

- Se o valor obtido for superior ao número indicado como parâmetro do programa, o ficheiro é removido da "cache" e a informação contida no ficheiro `.info` é actualizada de forma a repercutir esse facto.

Como a lista está ordenada por ordem crescente, o processo pára quando for encontrado um ficheiro que não necessite de ser removido.

Se o ficheiro `.info` tiver sido danificado, a lista é constituída percorrendo recursivamente o directório "cache" e obtendo a data em que cada ficheiro foi acedido pela última vez. Para isso recorre-se à chamada ao sistema `stat` que fornece os dados pretendidos.

É mantido no directório "log" o ficheiro `clean` onde é guardada informação acerca dos ficheiros removidos.
É indicado:

- O nome do ficheiro.
- A data em que foi colocado na "cache".
- A data em que o ficheiro foi removido.
- A data em que foi pedido pela última vez.

Estes dados são utilizados por programas de estatística, como veremos seguidamente.

6.2 Programas de estatística

A avaliação das vantagens deste tipo de sistema é um factor importante e indispensável. Só assim se poderá verificar a utilidade do trabalho efectuado e os benefícios que traz a todos os utilizadores e administradores de servidores ftp.
Com esta finalidade, foi criado o utilitário `report` que fornece os seguintes dados ocorridos num certo intervalo de tempo.

- Número total de acessos.
- Número de pedidos de ficheiros já existentes na "cache".
- Número de pedidos de ficheiros que não existiam na "cache", tendo sido copiados a partir da origem.
- Número de pedidos que não foram satisfeitos, devido a falhas de ligação aos servidores a partir dos quais se efectuam as cópias.
- Ficheiros mais solicitados.

- Durante quanto tempo as cópias locais se mantiveram actualizadas.
- Ficheiro removidos por não serem acedidos há muito tempo.

O programa guarda no ficheiro `.access` a data em que foi executado pela última vez, podendo assim referir no relatório produzido o intervalo de tempo a que os dados se reportam.
Com base nos dados contidos no ficheiro *dbm* `.info`, é constituída uma lista (*statistic*) com a seguinte informação:

fname Nome do ficheiro.

cache Número de pedidos de leitura do ficheiro que se encontrava no directório "cache".

ftp Número de pedidos de leitura do ficheiro que originaram uma ligação à máquina remota, por este não existir na "cache" ou a cópia local não estar actualizada.

Para que estes dados não sejam reutilizados em futuras execuções do programa, são colocados a zero os correspondentes campos *n_cache* e *n_ftp* do ficheiro `.info`. São removidas as linha que referem ficheiros que já não existem na "cache". Uma vez que estes dados são meramente informativos, não pondo em causa o funcionamento do programa, pensou-se ser este o procedimento correcto. Note-se que esta informação fica guardada em relatórios.

É obtido seguidamente o total dos pedidos efectuados.

```
access_count(statistic *ac)
{

 int tc=0;
 int tf=0;
 statistic *list;

 for (list=ac;list!=NULL;list=list->next) {
   tc = list->cache +tc;
   tf=list->ftp +tf;
 }


 fprintf(stderr,"Total de pedidos:   %d\n", tc+tf);
 fprintf(stderr,"Total de acessos a cache: %d\n",tc);
 fprintf(stderr,"Total de ligações ftp : %d\n",tf);
}
```

A partir da lista *statistic* são calculados os 10 ficheiros mais solicitados.

A figura 6.2 mostra um exemplo de um relatório assim gerado.

```
Report from May  4 11:50:05 1999 until  May 10 12:00:01 1999


Total de pedidos:         150
Total de acessos a cache: 90
Total de ligações ftp :   60



********************* TOP 10 **********************

/usr1/cache/debian/README                       9
/usr1/cache/debian/README.mirrors.html          4
/usr1/cache/debian/doc/social-contract.txt      3
/usr1/cache/debian/doc/00-INDEX                 2
/usr1/cache/debian/doc/bug-log-access.txt       2
/usr1/cache/debian/tools/README                 2
/usr1/cache/debian/doc/FAQ/debian-faq.ps.gz     2
/usr1/cache/debian/doc/FAQ/debian-faq-9.html    2
/usr1/cache/debian/doc/mailing-lists.txt        2
/usr1/cache/debian/README.mirrors.txt           2
```

Figura 6.2: Relatório de acessos

O relatório deve mencionar igualmente os erros ocorridos durante a execução do programa, o que permite avaliar a sua eficiência.

Baseando-se no ficheiro de “log” `error`, é indicado o número de erros ocorridos, referindo os erros verificados durante o estabelecimento de ligações a servidores ftp, durante a transferência de dados e os que se deveram a respostas inadequadas de servidores ftp.

No ficheiro de “log” `expire` estão indicadas as remoções de ficheiros que ocorreram devido a falta de espaço na “cache”, ou por estarem desactualizados. Em cada um dos casos é indicado o nome do ficheiro e o tempo que permaneceu na “cache”. Se a sua remoção se deveu a falta de espaço, é igualmente indicada a data em que foi acedido pela última vez.

O programa `clean`, conforma foi referido, reporta as remoções de ficheiros efectuadas, indicando no ficheiro de "log" `clean` a data em que ocorreram e, para cada ficheiro removido, a sua data de criação e data em que foi acedido pela última vez. Estes dados são igualmente reportados no relatório apresentado.

Capítulo 7

Análise de performance

O programa desenvolvido foi colocado em teste durante 2 meses, no servidor ftp da Universidade do Porto – ftp.up.pt. A "cache" encontra-se num sistema de ficheiros ocupado exclusivamente por si, conforme está indicado no ficheiro de configuração ilustrado na figura 7.1.

Exemplo

```
partition y
max_space 9600
lo_space 1000
hi_space 2000
cache_entry  debian-non-us  non-us.debian.org/debian-non-US  1d   5d
```

Figura 7.1: Ficheiro de configuração .config

Pretendeu manter-se localmente um "mirror" do directório debian-non-Us localizado na máquina non-us.debian.org.
Todos os utilizadores que acederam ao subdirectório Linux/debian-non-us, do directório público de ftp da máquina ftp.up.pt, tiveram acesso a informação facultada pelo `rfs`, de forma transparente.

O software mantido neste servidor é utilizado pelo Centro de Informática da Universidade do Porto na instalação de novas máquinas e na sua actualização. Existem igualmente outros servidores que fazem "mirroring" deste. Na análise dos resultados devemos ter em consideração estes factos, pois provocam vários acessos a toda a estrutura de directórios.

São apresentados na tabela 7.1 os acessos verificados durante 20 dias, para leitura de directórios e ficheiros. Em cada um dos casos é referido o número total de pedidos efectuados, o número de pedidos que provocaram uma ligação à máquina remota, os pedidos que solicitaram informação existente na "cache" e os erros ocorridos.

Tipo	Pedidos	Ligações ftp	Acessos cache	Erros
Leitura de directórios	3493	960	2524	9
Leitura de ficheiros	1324	395	912	17

Tabela 7.1: Acessos

O directório debian-non-us é formado por 47 subdirectórios. Cada uma da máquinas que fazem diariamente *mirroring* deste directório, solicita a leitura de cada um dos subdirectórios, desencadeando deste modo 48 pedidos de leitura dos seus conteúdos. A informação da "cache" é válida por um dia, conforme estipulado no ficheiro de configuração do `rfs`. Assim, por cada directório é estabelecida uma ligação à máquina remota de forma a garantir a consistência dos dados locais. São deste modo efectuadas 48 ligações diárias.

Durante o intervalo de tempo a que os dados se reportam, verificou-se a existência de 105 ficheiros desactualizados. O número máximo de ficheiros residentes na "cache" durante este período foi de 290. Na figura 7.2 estão indicados os 10 ficheiros mais solicitados.

```
*******************                             *******************
  Nome do Ficheiro                               Número de Pedidos
*******************                             *******************

 ls-lR.patch.gz                                          32
 ls-lR.gz                                                32
 ls-lR                                                   32
 indices-non-US/md5sums                                  30
 indices-non-US/dsync.list                               30
 indices-non-US/md5sums.gz                               30
 dists/potato/non-US/main/binary-i386/Packages.gz        27
 project/trace/nonus.debian.org                          27
 indices-non-US/Packages-all-nomsdosnames.gz             27
 project/trace/pandora.debian.org                        27
```

Figura 7.2: Os 10 ficheiros mais solicitados

Todos os erros ocorridos deveram-se a falhas de conectividade com a máquina remota.

Comparativamente, um *mirror* mantido por outro programa, por exemplo o `mirrordir`, que transfere todos os ficheiros, obriga a um gasto de espaço de disco de 77Mb. O espaço máximo mantido pelo `rfs` durante a fase de testes foi de 40Mb, donde podemos concluir que existem ficheiros que são utilizados raramente.

Programas como o `mirrordir` actualizam a informação local, comparando o conteúdo dos directório locais e remotos de forma a verificar a existência de novos ficheiros e ficheiros alterados. Nesta caso, procedem à sua transferência imediata. O tempo de execução destes programas depende do número de transferências que tenham que efectuar, isto é, do número de alterações verificadas na estrutura dos directórios.
O `rfs` compara igualmente o conteúdo dos directórios locais e remotos, mas no caso de existirem ficheiros alterados só os transfere posteriormente quando forem solicitados por algum utilizador. O tempo de execução da actualização dos directórios é basicamente o mesmo, independentemente da existência ou não de alterações às suas estruturas. Pode ser apenas agravado pela remoção de ficheiros que foram alterados.

O primeiro utilizador que solicite um ficheiro que ainda não se encontra na "cache", esperará mais tempo pela resposta, uma vez que este tem que ser transferido da máquina remota. Este tempo é aproximadamente o mesmo que esperaria se efectuasse o pedido directamente à máquina remota. A resposta a pedidos subsequentes é muito mais rápida, pois já existe uma cópia local.

Não nos pareceu significativa a análise dos tempos de resposta, pois estes dependem de factores externos, como o "estado da ligação" e do tempo que podem demorar as transferências dos ficheiros. Estes factores variam consoante o tráfego e a carga da máquina remota.

Capítulo 8

Conclusão

O objectivo inicial deste trabalho compreendia o desenvolvimento de um sistema de ficheiros por avaliação retardada.
Os objectivos que nos propusemos atingir foram os seguintes:

- Manter *mirrors* sem duplicar localmente toda a informação existente nos servidores remotos, diminuindo assim o espaço de disco ocupado.
- Manter localmente uma "cache" com todos os dados acerca da estrutura dos directórios remotos.
- Transferir os ficheiros apenas na altura da sua solicitação, se nesse momento ainda não existir uma cópia local.
- Diminuir a largura de banda gasta na transferência de ficheiros.
- Definir políticas de validade da informação local, de forma a mante-la consistente.
- Permitir que este sistema possa ser usado em servidores ftp e http.

Para uma eficiente concretização do trabalho efectuado foi necessário um estudo profundo do funcionamento do kernel, dando especial relevo a tudo o que se refere a sistemas de ficheiros e à forma como são implementados. Como o programa desenvolvido assenta no sistema de ficheiros `userfs`, foi dedicada especial atenção à sua funcionalidade e implementação.
Foi igualmente estudado o protocolo ftp, uma vez que foi a solução adoptada para transferir ficheiros para a máquina local e obter todos os dados acerca dos seus atributos e dos conteúdos dos directórios remotos.

Dos resultados obtidos parece-nos legítimo concluir que a estratégia adoptada foi bem sucedida.

8.1 Trabalho futuro

A comunicação entre as máquinas local e remota foi estabelecida com base no protocolo ftp. Poder-se-ia utilizar igualmente o protocolo `HTTP` (*Hypertext Transfer Protocol*) [Fie], permitindo deste modo acesso a servidores http, além do acesso a servidores ftp já implementado.

Seria necessário indicar no ficheiro de configuração `.config` qual o método a utilizar, recorrendo para isso à nomenclatura usual de um URL,

> método://máquina:port/directório

Assim, se se pretendesse comunicar com um servidor ftp, incluía-se no ficheiro `.config`, por exemplo, a linha

> cache_entry debian ftp://ftp.up.pt/debian 1d 3d

Se o protocolo pretendido fosse o http, a linha teria então a seguinte forma:

> cache_entry debian http://ftp.up.pt/debian 1d 3d

Para estabelecer a ligação ao servidor http é necessário criar um *socket* e conectar-se ao servidor no *port* indicado na configuração. Se não tiver sido indicado o *port* é utilizado o *port* usual, 80.

8.1.1 Protocolo HTTP

Após o estabelecimento de uma ligação ao servidor http (normalmente no *port* 80), o cliente pode enviar pedidos, com a seguinte estrutura:

> método url versão

Método Pode referir um dos seguintes valores:

- OPTIONS - Pedido de informação acerca das opções de comunicação disponíveis.
- GET - Pedido de obtenção dos dados indicados no URL.
- HEAD - Pede que seja enviado na resposta apenas o cabeçalho.
- POST - Pede ao servidor que aceite o que é referido em URL.

- PUT - Pede que o servidor guarde em URL o ficheiro enviado.
- DELETE - Pedido de remoção dos dados indicados em URL.

URL Identifica o recurso pretendido.

Versão Versão do protocolo HTTP.

A resposta do servidor é formada pelo cabeçalho e pelo corpo da resposta, separados por uma linha em branco. O cabeçalho começa por uma linha de estado constituída pela versão do protocolo e por um código numérico de estado, formado por 3 dígitos. O primeiro dígito define o tipo de resposta. Pode ter 5 valores possíveis:

1yz Informativo. Indica que o pedido foi recebido.

2yz Bem sucedido. O pedido foi recebido com sucesso e aceite.

3yz Redireccionado. São necessários mais procedimentos para satisfazer o pedido.

4yz Erro do cliente. O pedido tem erros sintácticos.

5yz Erro do servidor. O servidor não conseguiu satisfazer um pedido válido.

O cabeçalho permite enviar informação que não pode ser incluída na linha de estado, nomeadamente:

- Data.
- Tipo da informação contida no corpo da resposta, isto é, se se refere a texto ou a um ficheiro binário.
- Tamanho em bytes do corpo da resposta.

O corpo da resposta contém os dados solicitados ou uma mensagem de erro em formato html.

8.1.1.1 Listagem de directórios

Para obter a listagem do conteúdo de um directório é enviado um pedido GET, indicando o URL pretendido. O servidor envia no corpo da resposta, uma lista com o nome de cada ficheiro que constitui o directório indicado, a data em que foi alterado pela última vez e o seu tamanho. A resposta é enviada em formato html.

A resposta ao pedido

GET http://ftp.up.pt/Linux/debian-non-us HTTP/1.1

é ilustrada na figura 8.1.

O programa `rfs` deve analisar o código de estado da resposta e no caso de este referir um pedido bem sucedido, deve ignorar todas as linhas que se seguem até receber a linha <HR>. Seguidamente deve analisar os dados recebidos, extraindo a informação necessária à constituição do ficheiro `.list`, onde é indicado o conteúdo de cada directório.

8.1.1.2 Transferência de ficheiros

Para obter uma cópia de um ficheiro é enviado o URL pretendido, num pedido GET. O servidor envia no corpo da resposta o conteúdo do ficheiro.
No caso do pedido ser bem sucedido o `rfs` deve ignorar todas as linhas até receber uma linha em branco. As linhas seguintes têm o conteúdo do ficheiro.

A resposta ao pedido

GET http://ftp.up.pt/Linux/debian/README HTTP/1.1

é ilustrada na figura 8.2.

```
HTTP/1.1 200 OK
Date: Tue, 08 Jun 1999 10:01:11 GMT
Server: Apache/1.3.3 (Unix) Debian/GNU
Connection: close
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
 <HEAD>
  <TITLE>Index of /Linux/debian-non-us</TITLE>
 </HEAD>
 <BODY>
<H1>Index of /Linux/debian-non-us</H1>
<PRE><IMG SRC="/icons/blank.gif" ALT="  "> <A HREF="?N=D">Name</A>             <A HREF="?M=A">Last modified</A> <A HREF="?S=A">Size</A>
<A HREF="?D=A">Description</A>
<HR>
<IMG SRC="/icons/back.gif" ALT="[DIR]"> <A HREF="/Linux/">Parent
    Directory</A>        07-Jun-1999  09:34     -
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="dists/">dists/
    </A>                 07-Jun-1999 06:31      -
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="indices-non-US/">
    indices-non-US/</A>  07-Jun-1999  06:31     -
<IMG SRC="/icons/unknown.gif" ALT="[   ]"> <A HREF="ls-lR">ls-lR</A>
                         06-Jun-1999  18:52     57k
<IMG SRC="/icons/compressed.gif" ALT="[CMP]"> <A HREF="ls-lR.gz">
   ls-lR.gz</A>          06-Jun-1999  18:52     6k
<IMG SRC="/icons/compressed.gif" ALT="[CMP]"> <A HREF="ls-lR.patch.gz">
   ls-lR.patch.gz</A>    06-Jun-1999  18:52     1k
<IMG SRC="/icons/folder.gif" ALT="[DIR]"> <A HREF="project/">project/
   </A>                  07-Jun-1999 06:31      -
</PRE></BODY></HTML>
```

Figura 8.1: Resposta a um pedido de leitura de um directório

```
HTTP/1.1 200 OK
Date: Tue, 08 Jun 1999 10:07:19 GMT
Server: Apache/1.3.3 (Unix) Debian/GNU
Last-Modified: Wed, 12 May 1999 04:46:00 GMT
ETag: "1803-327-37390788"
Accept-Ranges: bytes
Content-Length: 807
Connection: close
Content-Type: text/plain
X-Pad: avoid browser bug

See http://www.debian.org/ for information about Debian GNU/Linux.
Two Debian releases are available on the main site:

Debian 2.1, or slink.  Access this release through dists/stable.
Debian 2.1 was released Monday, March 8, 1999.  Release notes,
including installation and upgrade information, are in
dists/stable/main/Release-Notes

Unstable, or potato.  Access this release through dists/unstable.
The current development snapshot is named potato.  Active development
is continuing here.

Older releases of Debian are at http://archive.debian.org/debian-archive

--- Other directories:
doc             Debian documentation.
indices         Various indices of the site.
project         Experimental packages and other miscellaneous files.
tools           Tools for creating boot disks and booting Linux.
```

Figura 8.2: Resposta a um pedido de leitura de um ficheiro

Referências

[BBD+98] M. Beck, H. Bohne, M. Dziadzka, R. Kunitz, U. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 2st edition, 1998.

[Fie] R. Fielding. *Hypertext Transfer Protocol – HTTP/1.1*. http://sunsite.doc.ic.ac.uk/packages/rfc/rfc2068.txt.

[Fit] Jeremy Fitzhardinge. *Userfs - Filesystem Implemented as User Processes*. ftp://rufus.w3.org/pub/veillard/userfs-0.9.5.tar.gz.

[GC94] Berny Goodheart and James Cox. *The Magic Garden Explained*. Prentice Hall, 1st edition, 1994.

[Gooa] Richard Gooch. *Kernel API changes from 2.0 to 2.2*. http://www.atnf.csiro.au/~rgooch/linux/docs/porting-to-2.2.html.

[Goob] Richard Gooch. *Overview of the Virtual File System*. http://www.atnf.csiro.au/~rgooch/linux/docs/vfs.txt.

[PR] J. Postel and J. Reynolds. *File Transfer Protocol*. http://sunsite.doc.ic.ac.uk/packages/rfc/rfc959.txt.

[Rus] David Rusling. *The Linux Kernel*. http://sunsite.unc.edu/linux/LDP/tlk/tlk.html.

[Sch96] Bruce Schneider. *Applied Cryprography*. John Wiley & Sons, 2st edition, 1996.

[Ste90] W. Richard Stevens. *Unix Network Programming*. Prentice Hall, 1990.

[TW97] Andrew Tanenbaum and Albert Woodhull. *Operating Systems - Design and Implementation*. Prentice Hall, 2st edition, 1997.