IME-USP

Instituto de Matemática e Estatística da Universidade de São Paulo

Proposta de Projeto – Sistemas de Arquivos no Linux

MAC5759 - Sistemas de Objetos Distribuídos

São Paulo, 11 de junho de 2002

Márcio Rodrigo de Freitas Carneiro
Livio Baldini Soares
Roberto Pires de Carvalho

1 Introdução

Esse documento visa esboçar o projeto final de MAC 5759 – Sistemas de Objetos Distribuídos. O projeto será um **sistema de arquivos distribuídos** para o sistema operacional Linux, cujos servidores serão implementados em CORBA.

Na segunda seção será delineado a arquitetura básica que se pretende utilizar para a implementação, na terceira seção busca-se uma justificativa para a arquitetura adotada juntamente com as dificuldades técnicas e conceituais encontradas, de forma a melhor esclarecer os detalhes da arquitetura. Na quarta seção é feito uma breve introdução ao *Virtual File System* (VFS) do Linux[1]. Finalmente, na quinta seção, descreve-se os objetos utilizados no sistema, juntamente com uma breve descrição de suas interfaces (i.e. seus métodos).

2 Arquitetura utilizada

A arquitetura básica será dividida em duas partes, para facilitar a especificação: a parte **servidora** e a parte **cliente**.

2.1 Arquitetura do Servidor

O servidor do sistema de arquivos distribuídos a ser implementado se baseia essencialmente em CORBA. É esperado que os clientes realizem toda a comunicação com o servidor via chamadas remotas do CORBA. Além disso iremos estruturar a *hierarquia* do sistema de arquivos (i.e. diretórios e nomes) via um servidor de nomes CORBA.

A princípio, apenas quatro classes de objetos serão necessárias para representar as entidades correspondentes aos objetos do VFS. Entretanto, precisamos apresentar o VFS para depois especificar os objetos e interfaces necessárias. Isso é feito posteriormente, nas seções 5 e 6. Apesar disso achamos que a *arquitetura* geral do sistema pode ser descrita aqui sem esses detalhes, que serão apresentados depois.

O uso de um servidor de nomes CORBA proporciona uma grande flexibilidade, já que podemos designar arquivos e diretórios diferentes a servidores de arquivos distintos, com **transparência** ao cliente, o que é incomum nos sistemas de arquivos distribuídos comumente utilizados no Linux, como o NFS, em que é necessário especificar o servidor e diretório externo a ser importado. Além de não haver transparência, esse tipo de montagem é **estática** e tem que ser manualmente reconfigurada quando o servidor ou diretório mudam.

O servidor de arquivos responsável por certos diretórios ou arquivos será executado como um processo na mesma máquina em que estão esses arquivos e diretórios. Esses arquivos *locais* poderão estar formatados em qualquer tipo de sistema de arquivos já que o processo servidor de arquivos fará chamadas ao sistema simples como `open`, `close`, `read`, `write`, `flush`, etc. Um esboço do diagrama da arquitetura do servidor de arquivos pode ser visto na figura 1.

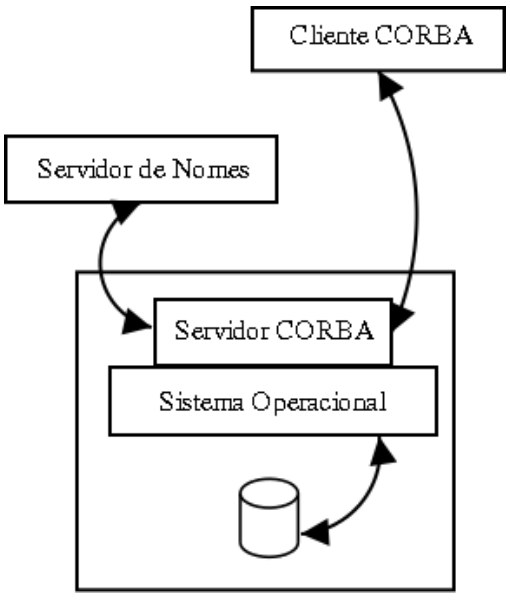

Figura 1: Diagrama da arquitetura do servidor

[1] A palavra **Linux** é muitas vezes usada nesse texto com o sentido de "núcleo do Linux", embora isso não seja explicíto, o leitor deve fazer a distinção quando cabível.

Numa perspectiva puramente técnica, o servidor de arquivos nada mais é do que um envólucro (*wrapper*) para as chamadas ao sistema local que manipulam os arquivos e diretórios acessíveis localmente[2].

Apesar de ser um modelo simplista para o servidor, acredita-se que será possível agregar algumas outras funcionalidades, como autenticação do cliente, criptografia de dados, compressão de dados, entre outros. Nenhum desses serviços existem atualmente para o notório sistema de arquivos distribuídos, o **NFS**. Ainda não está estruturada a forma como será disponibilizado esses serviços, mas um dos interesses desse projeto é permitir ao desenvolvedor inserir novas funcionalidades ao sistema de arquivos distribuídos de forma fácil, mesmo que não seja uma funcionalidade projetada de antemão. Por isso parece que a opção mais adequada será o uso de interceptadores CORBA, porém isso ainda está em estudo.

2.2 Arquitetura do Cliente

O cliente do sistema de arquivos distribuídos terá uma arquitetura mais complicada do que a do servidor. Nossa intenção nesse projeto é deixar a comunicação com o servidor CORBA transparente às aplicações do sistema. Para atingir essa transparência, com um desempenho aceitável, a maneira mais adequada é inserir um novo sistema de arquivos dentro do **sistema operacional** utilizado. Dessa forma as aplicações poderão continuar usando as chamadas ao sistema sem ter "ciência" de que estão acessando arquivos remotos.

A fim de atingir essa transparência para as aplicações, o cliente do sistema de arquivos distribuídos será dividido em duas partes: um **módulo** para o núcleo do Linux, e um cliente CORBA.

O módulo do nosso sistema cliente tem como função disponibilizar um sistema de arquivos para ser montado localmente. Além de registrar o sistema de arquivos (utilizando a chamada `register_filesystem` do VFS), esse módulo deve disponibilizar ao núcleo todas as operações que um sistema de arquivos precisa implementar (via a estrutura `super_block` do VFS). Na realidade, a priori, os membros do `super_block` deveriam ser ponteiros para funções que acessam o servidor CORBA remoto. Essa solução não foi adotada e explicaremos o porquê na próxima seção.

A solução que achamos mais apropriada é fazer com que o módulo do núcleo converse com um cliente CORBA local, via um **dispositivo virtual**. As funções do `super_block` na verdade serão então envólucros para escritas ao **dispositivo virtual** que serão lidas pelo cliente. Assim o cliente CORBA local é um processo em espaço de usuário, externo ao núcleo, que tem a função de se comunicar com o servidor de arquivos CORBA remoto, e servidor de nomes CORBA. Após o recebimento das mensagens do servidor de arquivos, o cliente deve repassar tudo ao módulo do núcleo, que por sua vez disponibiliza à aplicação que estaria requisitando as informações/alterações sobre os arquivos ou diretórios em questão.

A figura 2 graficamente demonstra a estrutura básica que será implementada no cliente do sistema.

3 Dificuldades no Cliente

Ao projetar o cliente do sistema encontrou-se uma dificuldade, cuja solução resultou na arquitetura adotada. O principal problema encontrado foi descobrir como realizar a comunicação entre o módulo (ou núcleo em si) e o servidor CORBA externo. Duas opções foram cogitadas para solucionar o problema:

1. Inclusão de um ORB dentro do núcleo do Linux. Essa solução, de fato, resolveria o nosso problema já que o módulo teria um *stub* e ORB próprio e a comunicação entre o cliente e servidor CORBA seria bastante eficiente.

[2]Nada impede, entretanto, que o sistema de arquivos sendo acessado pelo servidor de arquivos não seja local de fato. Esse fato será transparente ao processo, bastando que o núcleo disponibilize os arquivos desejados.

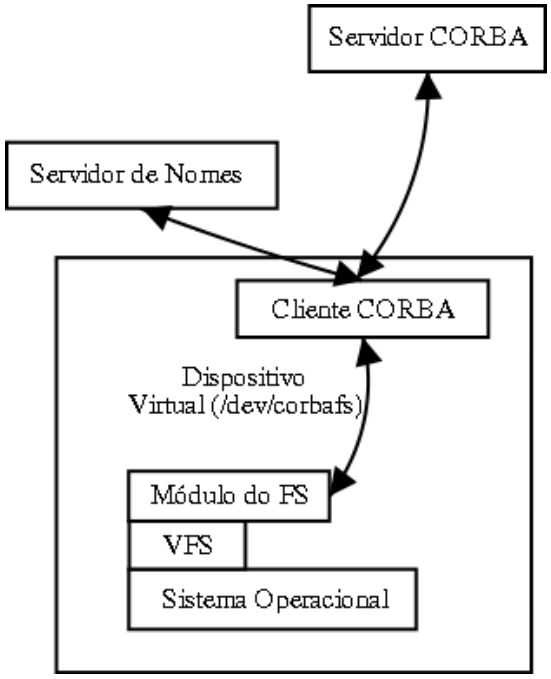

Figura 2: Diagrama da arquitetura do cliente

2. ORB cliente como processo "externo", em espaço de usuário. Essa solução é um pouco mais complexa, pois exige mais uma camada separando o módulo/núcleo e o servidor de arquivos. Além do mais, pode acarretar em alguma degradação de desempenho.

Apesar da opção (1) parecer ser mais simples e mais eficiente infelizmente a solução adotada foi a da opção (2). As razões para não adotar a solução (1) são basicamente três. Primeiramente haveria a necessidade de integração de um ORB no núcleo do Linux. Procurou-se um ORB estável e pequeno para o núcleo do Linux e o único encontrado foi kORBit[3] que aparentemente não está estável e cujo desenvolvimento está atualmente parado. Além disso o módulo a ser desenvolvido não funcionaria em qualquer núcleo do Linux, apenas naqueles com o *patch* incluindo esse ORB. Em segundo lugar os ORBs vistos consomem **muita** memória, o que é extremamente indesejável para inserção dentro do núcleo de um sistema operacional. Por fim o tempo hábil e energia necessária para viabilizar a integração de algum ORB existente (ou criação de novo) no núcleo do Linux, seria demais para esse projeto.

Com isso, foi decidido utilizar a alternativa (2). Essa alternativa também trouxe outro problema que é o de se fazer um *upcall*, isto é, o núcleo fazer uma chamada a uma função de um processo em espaço de usuário. O *upcall* é uma operação atualmente não permitida pelos núcleos padrões do Linux. Existe um *patch* que fornece essa funcionalidade, desenvolvida por Alessandro Rubini[4], que pode ser vista em: `http://www.linux-mag.com/2000-11/-gear_01.html`. Entretanto essa nova funcionalidade exigiria, novamente, muitas mudanças ao núcleo padrão do Linux, e tomaria muito tempo e energia.

Uma maneira que se pensou para "simular" os *upcalls* seria o uso de chamadas **RPC**s locais para o processo cliente. Para esse projeto, quase decidiu-se implementar uma solução usando chamadas **RPC**s. Era de se esperar que essas chamadas não ocasionam muito custo adicional, já que todas as chamadas deveriam ser chamadas locais (na mesma máquina), entretanto não se tem como comprovar essas suposições até experimentar-se uma implementação. Outro problema é que a complexidade do código deve aumentar, pois seria necessário criar mais uma interface e utilizar mais outro protocolo de comunicação.

Outra solução para a falta de *upcalls* no núcleo do Linux, é a solução adotada pelo projeto Coda FS[5], sistema de arquivos distribuídos desenvolvido pelo universidade CMU. No Coda FS o processo cliente rodando comunica-se com o módulo do núcleo através de uma entrada no `/dev/`. O processo cliente é um gerenciador de *cache* (Venus) que se comunica com o servidor (Vice) utilizando **RPC**. O gerenciador escreve o arquivo em um sistema de arquivos local e notifica o módulo no-

[3] `http://korbit.sourceforge.net/`

[4] Um dos autores do Linux Device Drivers, disponível em `http://www.oreilly.com/catalog/linuxdrive2/`

[5] http://www.coda.cs.cmu.edu/

vamente via o dispositivo virtual. Assim o núcleo pode manipular o arquivo com operações em um sistema de arquivos local (como o *ext2*).

Apesar da arquitetura do Coda FS ser extremamente avançada, permitindo tolerância a falhas e realização de operações desconectadas, ela não será usada. A desvantagem nesse caso é a dificuldade de implementação de um gerenciador de *cache* utilizando o disco como meio de troca de informações com o núcleo. Outro grande empecilho é a dificuldade em manter os arquivos consistentes no servidor e entre os cliente; essa operação torna-se mais complicada com o uso de *caches* locais.

Por fim, ao pesquisar o Parallel Virtual File System[6] (PVFS), cuja arquitetura foi fortemente baseada na do Coda FS, achou-se que essa seria a arquitetura ideal de comunicação entre o módulo do núcleo e o cliente local. A comunicação usada no PVFS entre o módulo e o cliente local (no caso do PVFS é um *daemon* ao invés de um gerenciador de *cache* como é o do Coda FS) é bem parecida com a do Coda FS, só que não é feito o uso de algum sistema de arquivos local.

Basicamente, os *upcalls* são implementados utilizando o dispositivo virtual `/dev/pvfsd`. O *daemon*, chamado `pvfsd`, lê as requisições desse dispositivo, e quando elas são satisfeitas, escreve uma resposta no mesmo dispositivo. O módulo do kernel então, utiliza funções do núcleo como `copy_from_user` e `copy_to_user` para transferir os dados dos arquivos sendo manipulados entre o núcleo e o cliente local.

4 Virtual File System Layer

O VFS[7] é uma camada abstrata existente em diversos núcleos Unix que pretende fornecer uma interface bem definida para implementação de sistemas de arquivos. A interface do VFS é estruturada em cima de tipos de objetos genéricos, e uma série de métodos que são chamados a partir desses objetos.

Dessa maneira é mais fácil implementar um sistema de arquivos para esses Unices, bastando fornecer uma implementação para os métodos dos objetos do VFS para a criação de um sistema de arquivos novo. Os objetos básicos conhecidos pelo VFS são: arquivos, sistemas de arquivos, *inodes*, e nomes de *inodes*. Cada um desses objetos possuem um conjunto de métodos especificados nas estruturas `super_operations` (para o sistema de arquivos), `file_operations` (para arquivos), `inode_operations` para *inodes*, e `dentry_operations` para nomes dos arquivos.

A seguir são apresentados os métodos necessários para implementar um sistema de arquivos usando o VFS do Linux. Note que nem todos esses métodos precisam ser implementados necessariamente, o Linux fornece alguns métodos genéricos para uso geral. As estruturas foram retirados da versão 2.4.19-pre2 do núcleo do Linux, mas não devem apresentar muitas diferenças entre versões. Os argumentos dos métodos foram removidos para deixá-los mais simples e claros. Esses métodos são extremamente importantes para podermos definir objetos e interfaces CORBA consistentes com as necessidades do VFS.

```
struct super_operations {
  void (*read_inode) ();
  void (*dirty_inode) ();
  void (*write_inode) ();
  void (*put_inode) ();
  void (*delete_inode) ();
  void (*put_super) ();
  void (*write_super) ();
  void (*write_super_lockfs) ();
  void (*unlockfs) ();
  int (*statfs) ();
  int (*remount_fs) ();
  void (*clear_inode) ();
  void (*umount_begin) ();
};

struct file_operations {
  struct module *owner;
  loff_t (*llseek) ();
  ssize_t (*read) ();
  ssize_t (*write) ();
  int (*readdir) ();
```

[6] http://parlweb.parl.clemson.edu/pvfs/

[7] Muitas vezes essa camada do VFS é chamada de *Vnode*

```
  unsigned int (*poll) ();
  int (*ioctl) ();
  int (*mmap) ();
  int (*open) ();
  int (*flush) ();
  int (*release) ();
  int (*fsync) ();
  int (*fasync) ();
  int (*lock) ();
  ssize_t (*readv) ();
  ssize_t (*writev) ();
  ssize_t (*sendpage) ();
};

struct inode_operations {
  int (*create) ();
  struct dentry * (*lookup) ();
  int (*link) ();
  int (*unlink) ();
  int (*symlink) ();
  int (*mkdir) ();
  int (*rmdir) ();
  int (*mknod) ();
  int (*rename) ();
  int (*readlink) ();
  int (*follow_link) ();
  void (*truncate) ();
  int (*permission) ();
  int (*revalidate) ();
  int (*setattr) ();
  int (*getattr) ();
};

struct dentry_operations {
  int (*d_revalidate) ();
  int (*d_hash) ();
  int (*d_compare) ();
  int (*d_delete) ();
  void (*d_release) ();
  void (*d_iput) ();
};
```

5 Objetos do sistema

Com o estudo preliminar do VFS, fica intuitivo definir os objetos e interfaces que devem ser projetados nesse sistema. Projetou-se quatro objetos que correspondem, justamente, às quatro entidades do VFS: sistema de arquivo, arquivo, *inode* e *dentry*.

As interfaces desses objetos devem corresponder às operações especificadas para cada uma dessas entidades do VFS, descritas na seção quatro. Dessa forma deve existir um mapeamento um a um entre as operações do VFS, implementadas no módulo do núcleo, e os métodos dos objetos no servidor de arquivos. As interfaces não serão explicitadas aqui, já que deverão ser análogas às operações listadas anteriormente.

A interação entre o cliente e esses objetos no servidor se dará basicamente via o servidor de nomes. Cada servidor que deseja exportar um diretório tem que registrar o seu objeto correspondente ao *super_block* no servidor de nomes. Assim, os clientes CORBA pegarão uma IOR no servidor de nomes para o objeto *super_block* de um servidor de arquivos. Com esse objeto, o cliente poderá pegar os outros objetos, como um objeto *inode*, ou *dentry*.

Referências

[BC01] Daniel P. Bovet and Marco Cesati. ***Understanding the Linux Kernel***. O'Reilly, 1st edition, 2001.

[Bra98] Peter J. Braam. **The Coda Distributed File System**. *http://www.coda.cs.cmu.edu/ljpaper/-lj.html*, 1998.

[Lab00] Parallel Architecture Research Lab. **Parallel Virtual File System Description**. *http://parlweb.parl.clemson.edu/-pvfs/desc.html*, 2000.

[RC01] Alessandro Rubini and Jonathan Corbet. ***Linux Device Driver***. O'Reilly, 2nd edition, 2001. http://www.xml.com/ldd/chapter/book/.