3
GridFS

O desenvolvimento do GridFS foi motivado pelas necessidades de compartilhamento de arquivos no CSBase, uma infra-estrutura para *clusters* e grades computacionais desenvolvida pelo nosso grupo de pesquisa. De acordo com as caracterísitcas apresentadas no Capítulo 1 e com as considerações apresentadas no Capítulo 2, nenhum dos sistemas existentes possuía todas as características desejadas. Dessa forma, optamos por definir e implementar um componente que agregasse todas as características e integrá-lo ao CSBase. Originalmente, o CSBase apresentava uma grande dependência em relação a existência de um mecanismo de compartilhamento de arquivos externo, nos moldes do NFS, que limitava a aplicabilidade do sistema em ambientes de grade. Dessa forma, desenvolvemos o GridFS como um mecanismo de compartilhamento de arquivos sobreposto aos sistemas de arquivos nativos dos nós da grade, viabilizando o compartilhamento de dados entre eles. A especificação e implementação do sistema foi norteada por requisitos de interoperabilidade, desempenho, escalabilidade, simplicidade e pouca dependência em relação a sistemas externos.

Apesar de ter o CSBase como usuário direto, o sistema foi concebido para atender qualquer aplicação que necessite do acesso e armazenamento de arquivos em uma rede de computadores. Mecanismos de transferência de dados com desempenho equivalente ao FTP estão presentes de forma a viabilizar o uso do sistema em aplicações que demandem a manipulação de grandes volumes de dados. O GridFS define uma interface CORBA para o acesso aos diversos sistemas de arquivos locais. Devido à disponibilidade de mapeamentos da interface CORBA para diversas linguagens, nosso servidor de arquivos pode interagir com diversos clientes, escritos em qualquer linguagem que suporte esse padrão.

Nas próximas seções, apresentamos as principais características do sistema; discutimos alguns aspectos de implementação; comentamos sobre o processo de desenvolvimento utilizado; descrevemos algumas formas alternativas

para acesso aos dados, incluindo o suporte a aplicações legadas; e apresentamos algumas limitações do sistema.

3.1 Características

Baseado nos sistemas de arquivos distribuídos e na observação do comportamento de sistemas de arquivos como o NTFS (Windows) e ext3 (Linux), definimos as características que o GridFS deveria possuir. A linguagem Java, por ter estabelecido uma API de manipulação de arquivos portável, também serviu como base para a especificação da interface exportada pelo GridFS. Os mecanismos de cópias de dados foram inspirados na arquitetura cliente-servidor, usada ao transferir arquivos via FTP. Sempre que possível, tentamos espelhar em nossa API a mesma semântica encontrada nas operações sobre os sistemas de arquivos locais. Ou seja, um usuário habituado a desenvolver sistemas usando a API de acesso a arquivos oferecida por linguagens como C ou Java, encontrará no GridFS uma API semelhante, mas que opera sobre arquivos remotos.

3.1.1 Federação de Servidores

Uma federação de servidores de arquivos pode ser criada através da definição de mapeamentos de um diretório para um outro servidor. Ou seja, é possível fazer com que um diretório em uma determinada árvore de arquivos referencie um outro servidor de arquivos executando em uma plataforma diferente da plataforma original, permitindo a integração de diversos sistemas em um único espaço de nomes virtual. Esse conceito é semelhante ao conceito de *mount points* do UNIX e essa funcionalidade visa aumentar a escalabilidade do sistema, bem como permitir a integração de diversos sistemas de arquivos locais em uma mesma árvore de diretórios distribuída.

3.1.2
Interface Orientada a Objetos

Visando oferecer uma interface de programação mais amigável para os desenvolvedores de aplicações, uma abordagem orientada a objetos foi adotada para definir a API do GridFS. Essa interface permite uma clara representação dos arquivos remotos, dos canais de leitura e escrita, e do próprio servidor de arquivos. Uma referência para um arquivo remoto permite, por exemplo, que canais de leitura e escrita sejam obtidos, ou que novos arquivos sejam criados em um diretório. Os canais de acesso, por sua vez, definem métodos como *read*, *write*, *seek* e *skip*, para que dados possam ser lidos ou escritos nos arquivos. Uma referência para o servidor de arquivos viabiliza que informações gerais sejam obtidas, como o espaço livre no sistema de arquivos remoto.

Vale salientar que no caso de arquivos remotos, o usuário não precisa trabalhar com caminhos absolutos para realizar a chamada dos métodos. Considerando a existência dos *mount points*, o uso de caminhos relativos evita que para cada chamada atuando sobre um caminho absoluto, o sistema precise avaliar os redirecionamentos e localizar o objeto efetivamente apontado pelo caminho. Ou seja, de posse de uma referência para um diretório, ao se obter um filho específico, o objeto retornado aponta diretamente para o servidor que mantém o arquivo.

O sistema é capaz de fornecer o acesso aos dados de duas formas: 1) fornecendo uma abstração de canais de leitura e escrita sobre os arquivos remotos, que disponibiliza o conteúdo de um arquivo através de chamadas remotas; e 2) permitindo que os arquivos sejam copiados entre os servidores, fazendo com que uma aplicação regular possa acessar o arquivo localmente, independente do uso de chamadas remotas. No segundo caso, um sistema de mais alto nível pode estar realizando a orquestração das atividades, como, por exemplo, preparando o ambiente de execução em uma máquina e realizando a cópia dos arquivos necessários para que uma determinada aplicação possa ser executada com sucesso.

3.1.3
Escalabilidade

A alocação de recursos no servidor deve ser realizada com cautela. O número de arquivos servidos por um determinado processo pode ultrapassar

dezenas de milhares de elementos e, considerando uma abordagem orientada a objetos, a criação de um objeto representando cada arquivo pode inviabilizar a disponibilização de um sistema de arquivos dessa magnitude. Diversos clientes acessam os arquivos e o momento onde um usuário remoto deixa de acessar o arquivo não pode ser facilmente determinado devido a falhas do cliente e desconexões repentinas.

Para a navegação sobre a árvore de arquivos distribuída, a alocação dos recursos no servidor deve ser realizada de uma forma independente do número de arquivos e clientes. Já para as operações de leitura e escrita de dados, visando uma maior eficiência dessas operações, alguns recursos são alocados no servidor em função dos canais de leitura e escrita abertos. Um descritor de arquivo associado a cada canal aberto é mantido no servidor com o objetivo de permitir um acesso otimizado. Ou seja, o servidor não possui estado para as operações de navegação na árvore, mas armazena estado em relação aos arquivos abertos. Um mecanismo de *lease* foi utilizado com o objetivo de evitar alocações indefinidas dos descritores de arquivos, como no caso de perda de conexão por parte dos clientes. Após um determinado período de tempo, um canal não utilizado é automaticamente finalizado, liberando os recursos no servidor.

3.1.4 Desempenho

Para que o sistema seja efetivamente utilizado como mecanismo de transferência de arquivos, ele deve possuir um desempenho no mínimo equivalente às demais técnicas existentes para essa tarefa. Caso o desempenho obtido pelo sistema seja inferior, existe uma tendência à sua substituição por soluções com um maior desempenho. Eventualmente, chamadas a código nativo podem ser necessárias para permitir que o desempenho máximo seja obtido. Ao considerar aplicações desenvolvidas em Java, o uso de *Java Native Interface* permite que códigos compilados especificamente para uma determinada plataforma possam ser utilizados em regiões críticas da aplicação, onde o aumento de desempenho justifica o uso de um código não portável. No caso da transferência de arquivos, a linguagem Java disponibiliza um mecanismo portável e otimizado para cópia de dados entre arquivos locais, ou entre um arquivo e a rede. O uso desse mecanismo faz com que o desempenho de transferência de dados fornecido pelo sistema operacional seja obtido em Java, sem comprometer a portabilidade do

sistema. Dado que as otimizações são feitas no nível da máquina virtual, onde as chamadas de código nativo para a transferência são efetivamente realizadas, não precisamos realizar chamadas a bibliotecas externas diretamente e ainda assim mantemos um bom desempenho.

3.1.5
Dados Históricos e Metadados

O sistema mantém dados históricos sobre as transferências realizadas entre os servidores e é capaz de fornecer a taxa média com que os dados trafegam entre dois deles. Essa é uma informação valiosa para que um agendador de processos possa avaliar se o custo de transferência de dados justifica a utilização de uma determinada máquina na grade computacional. Além disso, o sistema possui suporte a metadados, fazendo com que tuplas (campo, valor) possam ser usadas para armazenar qualquer tipo de informação sobre o arquivo, tais como descrições textuais sobre o conteúdo do arquivo ou *previews* de arquivos multimídia. Cada arquivo mantem o seu próprio conjunto de tuplas em um arquivo auxiliar e chamadas remotas são realizadas para alterar ou recuperar o valor das tuplas. Ao definir um suporte a metadados, o GridFS facilita o uso de um mecanismo externo de indexação e busca sobre a federação de servidores. Alguns testes preliminares foram realizados usando o Lucene [23], um sistema de busca baseado em texto, para indexar e recuperar referências remotas com base nos metadados dos arquivos.

3.1.6
Persistência das Referências para Objetos Remotos

As referências aos arquivos remotos são definidas em função do servidor e arquivo que elas representam. O processo servidor pode ser reiniciado sem a invalidação das referências fornecidas aos clientes, dado que o novo processo assume exatamente as mesmas características do processo anterior. Dessa forma, os clientes que possuem referências aos arquivos não precisam atualizá-las, pois as mesmas voltam a ser válidas no momento da reinicialização. Contudo, devido a natureza otimizada dos canais de acesso aos dados, eles são encerrados e precisam ser reabertos para que o acesso seja retomado.

3.1.7
Interoperabilidade

Ao considerar um sistema que atenderá diversas aplicações e cenários de uso, ele deve ter a característica de dar suporte a clientes desenvolvidos em diferentes linguagens e de poder ser executado sobre diferentes plataformas e sistemas operacionais. Ou seja, o sistema deve ser interoperável tanto do ponto de vista dos clientes, como dos ambientes onde ele pode ser executado. A implementação de clientes em linguagens como C/C++, Java, entre outras, e a compatibilidade do sistema com o Windows, Linux, AIX, Solaris, e outros sistemas operacionais deve ser possível e, para isso, o sistema deve se basear em padrões amplamente estabelecidos.

3.2
Aspectos de Implementação

De acordo com as características citadas anteriormente, optamos pelo middleware CORBA para definição das interfaces dos objetos remotos e pela linguagem Java para implementação do serviço. Pelo uso dessas tecnologias, podemos atender as características propostas e desenvolver um sistema portável e interoperável. Diversos aspectos foram considerados para garantir que o sistema seria capaz de atender a um grande número de usuários e que seu desempenho de cópia de arquivos fosse equivalente ao encontrado pelas técnicas de transferência atuais, como o FTP.

O *middleware* CORBA possui uma linguagem de definicão de interfaces, *Interface Definition Language* - IDL, que permite que as funcionalidades de um sistema sejam definidas de uma forma interoperável entre várias linguagens e ambientes. Uma interface CORBA (IDL) foi definida para cada um dos tipos de objetos remotos.

Na listagem 3.1, podemos ver a definição dos `typedef`s, que caracterizam alguns vetores utilizados nas declarações das funções. `Path` representa um caminho como uma sequência de `string`s. Todos os elementos da sequência, exceto o último, representam um diretório do sistema de arquivos local, ou um *mount point* definido para um outro servidor de arquivos. O último elemento da sequência pode representar um arquivo, um diretório, ou um *mount point*. O `FileSequence` representa um vetor de `RemoteFile`, ou seja, um vetor de

arquivos remotos.

```
1  typedef sequence<string> Path;
2  typedef sequence<RemoteFile> FileSequence;
```

Listagem 3.1: Definição dos typedefs

A interface `RemoteFile` representa um arquivo remoto. Um arquivo, como na maioria dos sistemas de arquivos tradicionais, pode representar tanto um diretório quanto um arquivo regular. A listagem 3.2 apresenta algumas funções disponibilizadas nessa interface e que estão relacionadas com a criação e navegação na árvore de arquivos distribuída. Apesar de todas as funções possuírem o seu conjunto de exceções definidos na IDL, optamos por suprimí-los nessa listagem inicial para aumentar a legibilidade do código. A função `createDirectory` cria um diretório com o caminho especificado. Esse diretório será criado internamente ao diretório representado pelo objeto sobre o qual o método foi invocado. Caso o caminho possua mais de um elemento, todos os elementos intermediários serão criados. A função `createFile` tem um comportamento análogo, mas cria um arquivo vazio como elemento final. A função `isDirectory` indica se a referência atual aponta efetivamente para um diretório. A função `addMountPoint` cria um *mount point* para um outro arquivo, potencialmente em um sistema de arquivos disponibilizado por outro GridFS. Caso o caminho possua mais de um elemento, semelhantemente a criação de diretório e arquivos, todos os elementos necessários são criados e o último elemento representa o *mount point* propriamente dito. A definição do *mount point* atua sobre um arquivo remoto e recebe uma referência para o alvo, ou seja, um outro arquivo remoto. Assim, um cliente em uma terceira máquina é capaz de interligar dois servidores do GridFS. A idéia dos *mount points* é permitir que uma federação do serviço seja criada, onde existe apenas uma árvore lógica para os arquivos armazenados na grade, mas diversos servidores são responsáveis pelos arquivos distribuídos. A função `removeMountPoint` remove um *mount point* sem modificar o arquivo remoto para o qual o ele aponta. A função `remove` remove um arquivo ou diretório; caso o arquivo esteja em uso (leitura ou escrita), ou caso o diretório não esteja vazio, uma exceção será lançada.

```
1  interface RemoteFile {
2    RemoteFile createDirectory(in Path name)
3    RemoteFile createFile(in Path name)
4    boolean isDirectory()
5    boolean addMountPoint(in Path name, in RemoteFile target)
6    RemoteFile removeMountPoint(in Path name)
7    boolean remove()
```

```
8   string getName()
9   Path getFullName()
10  boolean copyTo(in RemoteFile destination, in string method)
11  boolean moveTo(in RemoteFile destination, in string method)
12  boolean truncate(in unsigned long long size)
13  long long lastModified()
14  OctetSequence hash()
15  RemoteFile getChild(in Path name)
16  FileSequence getChildren()
17  RemoteFile getParent()
18  unsigned long long size()
19  FileServer getFileServer()
20  [...]
21 };
```

Listagem 3.2: Definição de RemoteFile

A função `getName` retorna o nome do arquivo, onde apenas o último elemento do caminho do arquivo é retornado. A função `getFullName` retorna a sequência de `string`s que deve ser percorrida desde a raiz do servidor onde o arquivo é hospedado até que o mesmo seja atingido. A função `copyTo` copia o conteúdo de um arquivo sobre o arquivo de destino. O destino também deve ser um arquivo e será sobrescrito com o conteúdo do arquivo sobre o qual a função de cópia foi chamada. Como veremos posteriormente, um mecanismo extensível permite a definição de diferentes métodos de cópia que podem, por exemplo, usar os canais de acesso remoto, definir protocolos para a transferência dos arquivos via rede ou mesmo utilizar protocolos como o FTP. A função `moveTo`, caso esteja movendo um arquivo dentro de um mesmo GridFS, utiliza funções do sistema de arquivo local para renomear o arquivo, o que caracteriza que o arquivo foi movido. Caso exista mais de um GridFS envolvido, a operação `moveTo` é tratada como uma operação de cópia seguida por uma remoção. A função `truncate` atribui um valor de tamanho arbitrário ao arquivo. A função `hash` calcula o *hash* MD-5 do arquivo, que pode ser usado para verificar a integridade de um determinado arquivo em relação ao *hash* do arquivo original.

Um servidor GridFS deve ser instalado e executado para cada recurso (sistema de arquivos) para o qual se deseja fornecer o acesso remoto. Para cada servidor GridFS, um objeto `FileServer` é disponibilizado e compartilhado por todos os `RemoteFile`s. A listagem 3.3 apresenta algumas funções disponibilizadas pelo `FileServer`. A função `getName` recupera o nome simbólico do servidor de arquivos. A função `getRoot` retorna a referência para o arquivo que corresponde ao diretório raiz da árvore exportada. A função `getFreeSpace` recupera o espaço disponível no sistema de arquivos local do servidor. A função

`getAverageTransferRateToHost` indica qual a taxa média de transferência obtida nas transferências para o servidor indicado como parâmetro. Por fim, a função `getCopyServerAddress` permite que o endereço do servidor específico para um determinado método seja recuperado, fazendo com que o cliente possa estabelecer uma conexão direta e transferir o arquivo para o servidor de acordo com o protocolo apropriado.

```
1 interface FileServer {
2   string getName()
3   RemoteFile getRoot()
4   long long getFreeSpace()
5   double getAverageTransferRateToHost(in string fileServerName)
6   string getCopyServerAddress(in string method)
7 };
```

Listagem 3.3: Definição de FileServer

Tipicamente, uma referência para um objeto `FileServer` é obtida usando um endereço `corbaloc` no seguinte formato:

```
corbaloc::1.2@<host>:<port>/standardImplName/FileServerPOA/FileServer
```

Dado esse endereço, podemos obter facilmente uma referência para um `FileServer` executando em um determinado servidor e porta. Apenas os campos `<host>` e `<port>` precisam ser alterados para refletir a localização do `FileServer` desejado.

A escrita e leitura de dados dos arquivos pode ser feita pelo uso dos canais de acesso. A listagem 3.4 apresenta uma continuação da interface `RemoteFile`, mostrando as funções com o objetivo de fornecer os canais de acesso para um arquivo remoto. A definição das interfaces dos canais está apresentada na listagem 3.5.

```
1 interface RemoteFile {
2   [...]
3   ReadChannel getReadChannel()
4   WriteChannel getWriteChannel()
5   RandomAccessChannel getRandomAccessChannel()
6 };
```

Listagem 3.4: Definição de RemoteFile

```
1   interface Channel {
2     boolean isOpen()
3     RemoteFile getFile()
```

```
4     void close()
5   };
6
7   interface WriteChannel : Channel {
8     long write(in unsigned long size, in OctetSequence buffer)
9   };
10
11  interface ReadChannel : Channel {
12    unsigned long long skip(in unsigned long long num)
13    unsigned long read(in unsigned long size,
14                       inout OctetSequence buffer)
15  };
16
17  interface RandomAccessChannel : ReadChannel, WriteChannel {
18    void seek(in unsigned long long num)
19  };
```

Listagem 3.5: Definição dos Canais

Em relação às características de metadados, o GridFS define os elementos CORBA conforme a listagem 3.6. Os metadados estão relacionados apenas aos arquivos e não é possível associar metadados aos diretórios. Cada tupla (campo,valor) associa uma chave a uma sequência de bytes que armazenam a forma persistente do objeto associado.

A função `setMetadata` modifica os metadados associados a um determinado arquivo, potencialmente removendo todas as tuplas caso uma sequência de tamanho zero seja passada como parâmetro. Ou seja, independente dos metadados existentes antes da chamada para essa função, o novo conjunto de metadados será definido somente pela sequência de metadados passada como parâmetro. Por outro lado, a função `updateMetadata` realiza uma operação de atualização, eventualmente aumentando a sequência de metadados armazenados pelo GridFS, para o arquivo. Caso uma tupla com uma determinada chave já tenha sido armazenada previamente, o valor associado será atualizado para satisfazer o valor presente na tupla que foi passada como parâmetro. Caso nenhuma tupla tenha sido previamente armazenada com a chave, uma nova tupla será armazenada, garantindo o mapeamento existente na sequência de metadados passada como parâmetro. A função `getAllMetadata` recupera toda a sequência de metadados associada a um arquivo remoto e a função `getMetadata` recupera apenas a sequência de metadados associados aos campos informados como parâmetro, e é especialmente útil quando desejamos obter um conjunto limitado de metadados, evitando um tráfego desnecessário na rede.

```
1  struct Metadata {
2    string field;
3    OctetSequence value;
4  };
5
6  typedef sequence<string> FieldSequence;
7
8  typedef sequence<Metadata> MetadataSequence;
9
10 interface RemoteFile {
11   [...]
12   void setMetadata(in MetadataSequence metadata)
13   void updateMetadata(in MetadataSequence metadata)
14   MetadataSequence getAllMetadata()
15   MetadataSequence getMetadata(in FieldSequence fields)
16 };
```

Listagem 3.6: Definição dos Metadados

Diversas exceções também foram definidas para representar situações onde uma solicitação do usuário não pôde ser atendida. A listagem 3.7 apresenta as exceções definidas no sistema. Elas podem ser agrupadas em três tipos principais: 1) as exceções nas linhas 1-7 indicam que uma solicitação não pôde ser realizada, devido a uma inconsistência na solicitação do cliente em relação ao sistema de arquivo local. Por exemplo, a tentativa de criação de um elemento em um diretório que já possui um elemento com o mesmo nome irá lançar uma exceção `FileAlreadyExistsException`. Caso o cliente informe um caminho inválido como parâmetro para um método, uma exceção `InvalidPathException` será lançada. Um caminho é inválido quando ele possui caracteres não aceitos pelo sistema de arquivos local, ou quando ele tenta referenciar uma área externa a raiz exportada pelo serviço (através de operadores como ".."). A `FileInUseException` indica que um arquivo não pode ser removido pois existem canais de acesso abertos para o mesmo; 2) as exceções nas linhas 8-9 indicam situações de erro onde o sistema não foi capaz de atender a solicitação do usuário devido a invalidação da referência previamente obtida pelo cliente. Essa situação ocorre caso um arquivo tenha sido removido do sistema de arquivos local, ou caso uma tentativa de operação sobre um canal de acesso fechado seja realizada; e 3) a `ServerException` indica que o sistema encontrou uma situação inesperada e não pode atender a solicitação. Essa exceção normalmente é causada por problemas de configuração do serviço. Sempre que possível, mensagens elucidativas estão presentes para permitir a depuração do servidor ou o tratamento adequado das exceções.

```
1   exception FileAlreadyExistsException { Path name; };
2   exception FileNotFoundException { Path name; };
3   exception NotFileException { Path name; };
4   exception NotDirectoryException { Path name; };
5   exception NotEmptyException { Path name; };
6   exception InvalidPathException { Path name; };
7   exception FileInUseException { Path name; };
8   exception ClosedChannelException { Channel ch; };
9   exception InvalidStateException { string message; Path name; };
10  exception ServerException { string message; };
```

Listagem 3.7: Definição das Exceções

A seguir, apresentamos a função `createFile` em detalhes, servindo como um exemplo da metodologia utilizada para implementar as funcionalidades do sistema. Lembrando que um `FileServer` pode ser obtido a partir do seu endereço `corbaloc`, e considerando que o cliente já possui uma referência para um `RemoteFile`, que pode ter sido obtida a partir da obtenção da raiz exportada pelo `FileServer`, a função `createFile` pode ser chamada com o objetivo de criar um novo arquivo no sistema de arquivos remoto.

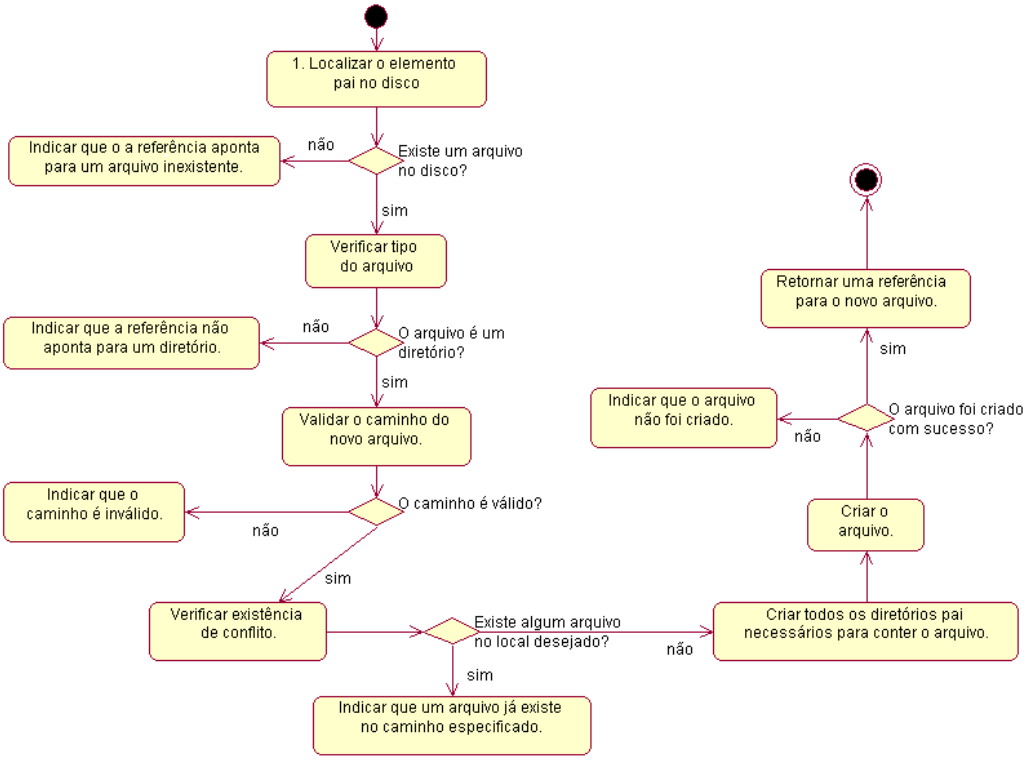

Figura 3.1: Diagrama de Atividades (createFile)

O diagrama de atividades para essa função é apresentado na figura 3.1. Ele apresenta todos os passos necessários para a criação de um arquivo no

servidor. Diversas verificações precisam ser realizadas, para que a operação seja realizada com sucesso ou para que uma exceção específica seja lançada. As demais funções do sistema foram implementadas seguindo o mesmo nível de datalhamento exposto nesse diagrama. A listagem 3.8 apresenta o exemplo completo da assinatura da função `createFile`, levando em consideração as exceções lançadas de acordo com o diagrama de atividades.

```
1 interface RemoteFile {
2 [...]
3   RemoteFile createFile(in Path name)
4       raises(FileAlreadyExistsException, NotDirectoryException,
5              InvalidPathException, InvalidStateException,
6              ServerException);
7 [...]
8 };
```

Listagem 3.8: Exceções da função createFile

A interface IDL CORBA completa do sistema pode ser vista no Apêncice A.

3.2.1 Escalabilidade em CORBA

Um desafio inicial do projeto era o de permitir a disponibilização de um sistema de arquivos local de uma grande magnitude, composto por centenas de milhares de arquivos. Em um cenário de objetos distribuídos, diversos clientes podem recuperar referências para os objetos nos servidores. Ao recuperar uma referência, o cliente é capaz de realizar chamadas de métodos remotos como se eles fossem locais. O middleware é responsável por encaminhar as chamadas e o servidor atende a solicitação, de acordo com o objeto referenciado.

O ciclo de vida do cliente independe do servidor. Durante a execução, clientes podem perder a conexão abruptamente, ou os processos podem ser reiniciados devido às falhas. Dessa forma, ao criar objetos locais e fornecer a referência remota aos clientes, o servidor pode perder o controle sobre quais objetos ainda são necessários, dado que os clientes que possuem a referência podem ser desativados sem aviso prévio.

Como discutido anteriormente, a alocação de recursos no servidor não pode depender do número de arquivos locais, ou do número de clientes navegando na árvore do sistema. Uma possível solução consiste em disponibilizar

um único objeto no servidor, representando todos os arquivos, e fazer com que a cada chamada o caminho absoluto do arquivo seja informado. Essa abordagem, apesar de eliminar a necessidade de vários objetos, dificulta o tratamento otimizado dos *mount points*. Caso o caminho completo do arquivo seja informado a cada chamada de método, o servidor responsável por tratar a raiz do espaço de nomes sempre será contactado e todos os redirecionamentos existentes no caminho precisarão ser resolvidos para que o arquivo seja efetivamente acessado. Uma forma escalável, que independe da quantidade de arquivos existentes e ainda assim dá suporte à utilização eficiente dos *mount points* é possível pela utilização de políticas definidas em CORBA [24].

Usualmente, para cada objeto no servidor, o ORB cria um identificador e o anexa à referência que será enviada ao cliente. Assim, para cada identificador, existe um mapeamento para um objeto local ao servidor. Essa abordagem é usada pelo POA (*Portable Object Adapter*) conhecido por *RootPOA*. Porém, alguns outros mecanismos definidos em CORBA podem ser utilizados, entre eles temos o *ServantLocator* e o *DefaultServant*.

Existem diversas configurações para o POA. De acordo com o padrão de acesso aos objetos remotos e da necessidade em se manter um estado para cada objeto, uma determinada configuração pode ser mais indicada. É importante salientar que não existe uma configuração ideal para todos os casos, e que as diversas políticas do POA influenciam em aspectos como escalabilidade, simplicidade de programação, gerenciamento de estado dos objetos, entre outros. Durante este trabalho, avaliamos o comportamento de três configurações do POA: o RootPOA, o ServantLocator e o DefaultServant.

O *servant* é o objeto que implementa a interface remota exportada pelo objeto. Para cada chamada, um *servant* é localizado e a chamada do método adequado é realizada. No caso do *RootPOA*, para cada arquivo cuja referência foi entregue a um cliente, é necessário a criação de um novo *servant* possuindo a informação sobre o arquivo representado pelo objeto e uma entrada mapeando a identificação do objeto no *servant* correspondente é mantida pelo POA. Considerando o número de arquivos que pretendemos atender, essa abordagem claramente apresenta problemas de escalabilidade.

O *ServantLocator* permite que o *servant* seja localizado no momento da requisição e que um mecanismo de localização específico seja definido de acordo com a aplicação. A partir da implementação dos métodos de `pre_invoke` e `pos_invoke`, os procedimentos para recuperação e descarte do *servant* podem ser realizados. Ao utilizar o *ServantLocator*, a identificação do objeto é definida

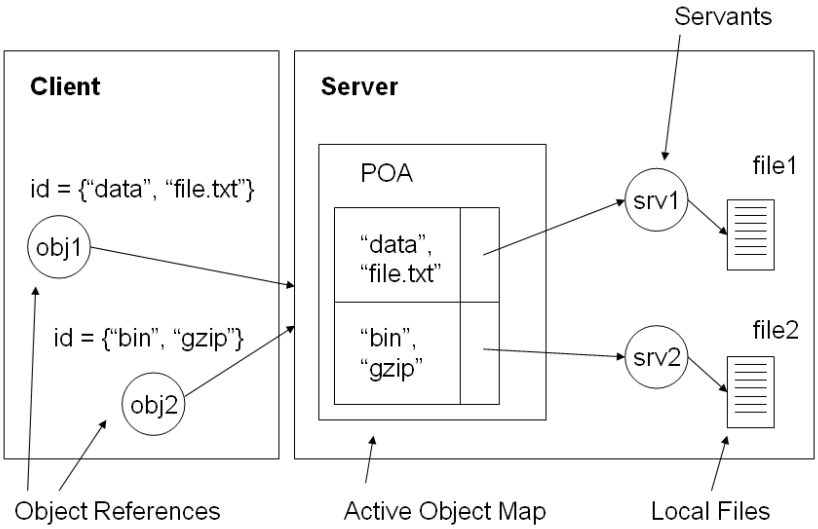

Figura 3.2: Abordagem do *RootPOA*

pelo programador e pode possuir informações relevantes sobre a criação, localização ou parametrização do *servant* que deverá atender a solicitação. No caso de servidor de arquivos, podemos usar o caminho do arquivo no sistema de arquivos local como identificação, eliminando assim a necessidade da manutenção de um mapeamento entre identificadores e arquivos. Sempre que uma chamada for feita ao sistema, podemos criar um novo *servant* com base no caminho contido no identificador.

Já o *DefaultServant* se caracteriza pela existência de um único *servant*. Dessa forma, um único objeto previamente instanciado é responsável por atender todas as solicitações dos clientes remotos. Nesse mecanismo, o programador do servidor também deve informar a identificação do objeto. Assim, no momento da requisição, a informação contida na referência do cliente é utilizada para parametrizar o *servant* e localizar o arquivo em disco, atendendo a solicitação do cliente. Dessa forma, evitamos os pontos negativos do *RootPOA*, com a necessidade da instanciação e mapeamento de vários *servants*, bem como o esforço adicional do *ServantLocator* em buscar ou instanciar um *servant* apropriado para cada solicitação. Ou seja, utilizando o *DefaultServant*, o servidor é capaz de fornecer um número arbitrário de referências sem a necessidade de alocação de nenhum recurso adicional por arquivo ou cliente. As figuras 3.2 e 3.3 apresentam as abordagens discutidas em relação ao *RootPOA* e *DefaultServant*, respectivamente.

Diversos experimentos foram realizados para validar o funcionamento de cada mecanismo. Durante os testes, pudemos observar um grande consumo de memória utilizado pelo RootPOA, e um consumo de memória equivalente en-

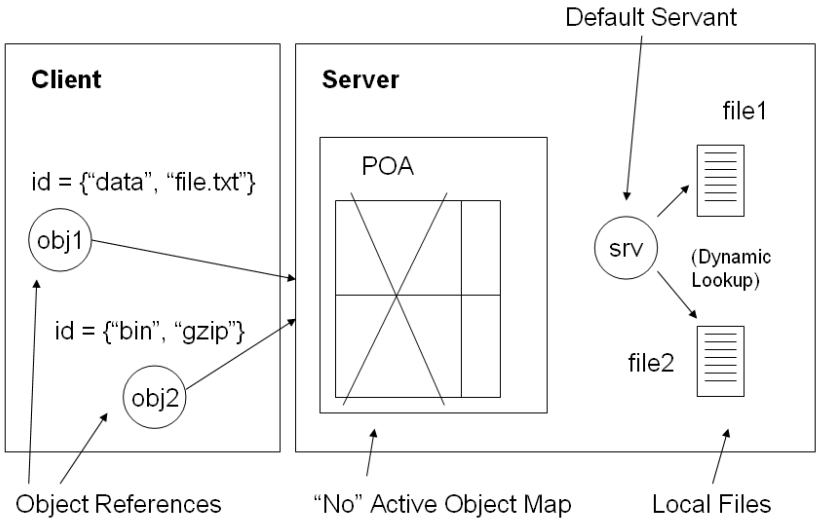

Figura 3.3: Abordagem do *DefaultServant*

tre as políticas *ServantLocator* e *DefaultServant*. Na figura 3.4, apresentamos o consumo de memória na criação de arquivos remotos, onde, para cada arquivo criado, uma referência é retornada para o cliente. Três curvas são mostradas, relacionando o consumo de memória ao número de arquivos criados no servidor. As curvas do *RootPOA* e *DefaultServant* foram obtidas com a utilização de um protótipo implementado com o objetivo de tomar algumas decisões para a implementação final. Como comparação, apresentamos a curva *DefaultServant (atual)* que representa o consumo de memória na implementação final do sistema em função do número de arquivos criados. A curva do *ServantLocator*, obtida pela execução do protótipo, foi omitida por apresentar um comportamento semelhante a curva *DefaultServant*.

Na nossa implementação inicial, o *ServantLocator* buscava o *servant* relacionado a referência remota em uma cache, caso o *servant* não fosse localizado, um novo *servant* era criado e adicionado a cache. O *DefaultServant* possuia um único *servant* que era parametrizado de acordo com a referência remota. A cache do *ServantLocator* possuia um tamanho pequeno e por isso as curvas *ServantLocator* e o *DefaultServant* eram semelhantes. O maior consumo de memória entre o protótipo e a implementação final se devia a existência de um mapeamento entre identificadores e caminhos no disco. Esse mapeamento era necessário para permitir que o sistema rastreasse as operações de *moveTo()*, fazendo com que qualquer cliente continuasse tendo uma referência válida caso um arquivo fosse movido. Durante os testes no protótipo, concluímos que o consumo de memória e a limitação da escalabilidade em relação a esse mapeamento não justificavam o seu uso, e optamos por manter a semântica de

que um arquivo movido é equivalente a um arquivo que foi apagado e recriado em outro local. Ou seja, no momento que um arquivo é movido, as referências para o antigo arquivo deixam de ser válidas.

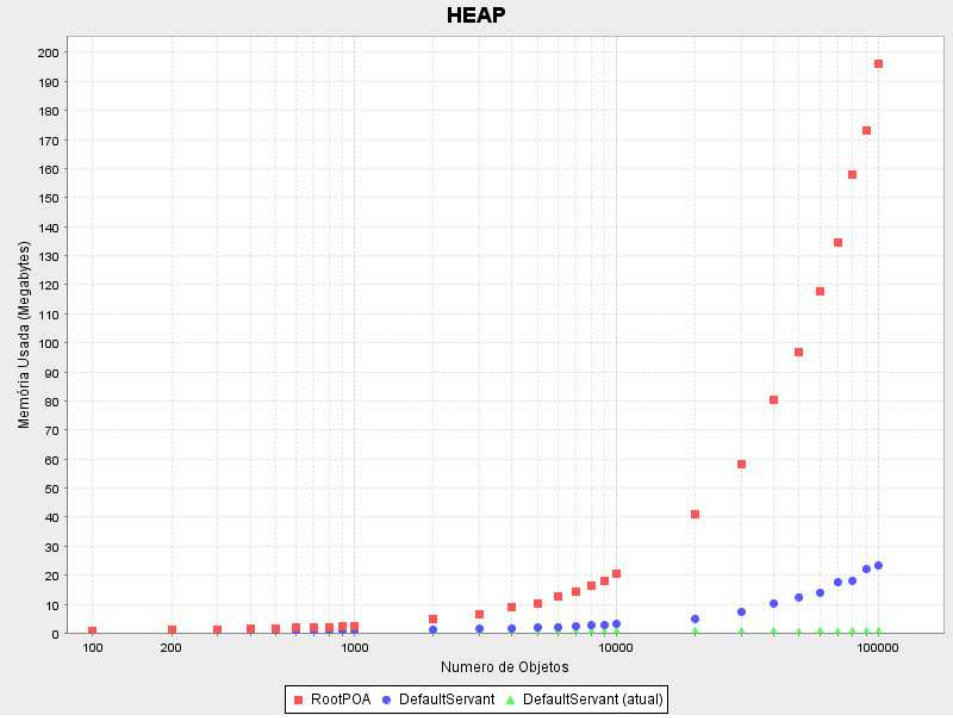

Figura 3.4: Consumo de Memória da Máquina Virtual

Pelas observações realizadas nos experimentos sobre o protótipo, decidimos usar a política *DefaultServant* para servir todos os arquivos na versão final. Ao usar essa política, minimizamos os problemas de escalabilidade ao tratar centenas de milhares de arquivos nos sistemas de arquivos locais, e eliminamos o problema de coleta de lixo distribuída decorrente do fornecimento de referências aos clientes. Ao mesmo tempo, mantemos uma implementação simples e clara.

Da forma como implementamos o serviço, é possível a utilizacão de caminhos relativos para a obtenção de filhos de qualquer diretório da árvore distribuída. Uma referência para um `RemoteFile` é obtida sem nenhum custo de memória para o servidor e possui toda a identificação necessária para localizar o arquivo. Como discutido anteriormente, uma outra solução escalável poderia consistir na fusão das funcionalidades de `FileServer` e `RemoteFile`, fazendo com que um caminho absoluto fosse necessário sempre que se desejasse chamar as funcões sobre algum arquivo. Essa solução apresenta algumas desvantagens: a cada chamada, todo o caminho absoluto precisaria ser tratado e todos os *mount points* no caminho precisariam ser resolvidos; a API do sistema se tornaria menos amigável, por remover a interface `RemoteFile` e forçar o uso

de caminhos absolutos sempre. Ou seja, com o uso do *DefaultServant*, somos capazes de manter um servidor sem estado, no que se refere aos `RemoteFile`s, e de disponibilizar um número arbitrário de arquivos usando um único objeto no servidor.

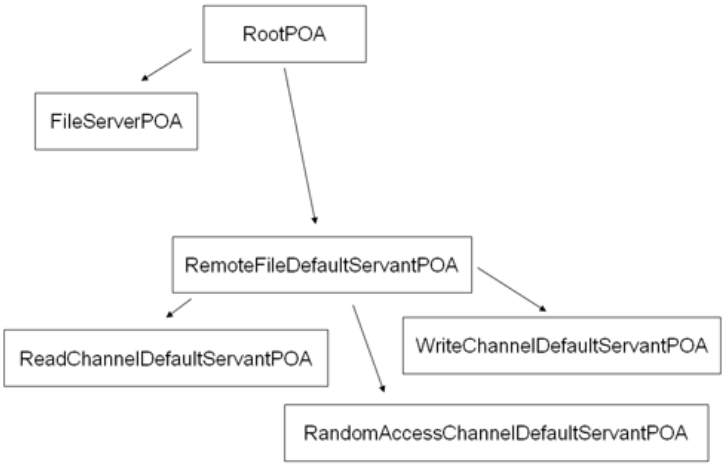

Figura 3.5: Hierarquia de POAs

Como diversas interfaces são exportadas pelo servidor de arquivos, tais como `FileServer`, `RemoteFile`, `ReadChannel`, `WriteChannel` e `RandomAccessChannel`, optamos por utilizar um POA para cada uma delas. Essa abordagem simplifica a implementação do servidor, pois, dado que a política *DefaultServant* possui um único *servant*— responsável pelas solicitações, fazemos com que cada interface possua um POA e um *servant* específico para tratá-las. A figura 3.5 mostra a hierarquia de POAs utilizados no sistema, ela serve apenas como uma organização lógica dos POAs, dado que as políticas não são diretamente herdadas entre eles. Uma referência para o RootPOA pode ser obtida através de uma requisição ao ORB, e em seguida é possível recuperar seus filhos usando seu nome como identificador.

As políticas utilizadas para o POA responsável pelo `FileServer` são mostradas na tabela 3.1. Para que o `FileServer` tivesse um endereço `corbaloc` fixo e definido pelo programador do servidor de arquivos, o tipo de política `IdAssignmentPolicy` foi estabelecido como `USER_ID`. Assim como o tipo de política `LifespanPolicy` foi determinado como sendo `PERSISTENT`, para representar um objeto cujo ciclo de vida independe do processo servidor que o mantém. Dessa forma, uma referência para o objeto `FileServer` pode ser recuperada pelo cliente através de um endereço fixo e permanece válida mesmo que o processo servidor seja reiniciado.

Já para o `RemoteFile`, o tipos de política `RequestProcessingPolicy`

Tipo da Política	Valor Padrão	Valor Usado
LifespanPolicy	`TRANSIENT`	`PERSISTENT`
IdAssignmentPolicy	`SYSTEM_ID`	`USER_ID`

Tabela 3.1: Políticas do POA usadas para o FileServer

Tipo da Política	Valor Padrão	Valor Usado
LifespanPolicy	`TRANSIENT`	`PERSISTENT`
IdAssignmentPolicy	`SYSTEM_ID`	`USER_ID`
RequestProcessingPolicy	`USE_ACTIVE_OBJECT_MAP_ONLY`	`USE_DEFAULT_SERVANT`
ServantRetentionPolicy	`RETAIN`	`NON_RETAIN`

Tabela 3.2: Políticas do POA usadas para o RemoteFile

Tipo da Política	Valor Padrão	Valor Usado
IdAssignmentPolicy	`SYSTEM_ID`	`USER_ID`
RequestProcessingPolicy	`USE_ACTIVE_OBJECT_MAP_ONLY`	`USE_DEFAULT_SERVANT`
ServantRetentionPolicy	`RETAIN`	`NON_RETAIN`

Tabela 3.3: Políticas do POA usadas para os Canais

foi definido como `USE_DEFAULT_SERVANT` para fazer com que um único *servant* fosse utilizado para responder as solicitações referentes aos arquivos remotos. O tipo de política `ServantRetentionPolicy` foi definido como `NON_RETAIN` para fazer com que o POA não mantenha nenhum mapeamento de *object ids* para *servants*, fazendo com que todas as solicitações sejam repassadas ao *servant* padrão. O uso de `RETAIN` para esse tipo de política faria com que o POA sempre buscasse um *servant* associado ao *object id* antes de repassar a solicitação ao *servant* padrão. A tabela 3.2.1 apresenta as políticas utilizadas pelo `RemoteFile`.

Por fim, a tabela 3.3 apresenta as políticas utilizadas nos POAs relacionados aos canais de acesso remoto. Ele difere do POA do `RemoteFile` apenas pelo fato de que os canais não são persistentes. No caso dos canais, como temos um estado associado a cada objeto, poderíamos ter usado o *RootPOA* para gerenciar os *servants*. Para manter a homogeneidade de POAs referentes aos objetos que tratam os arquivos remotos, e para definir o mecanismo de *leasing* sobre objetos independentes de CORBA, optamos por utilizar o *DefaultServant* também nos canais de acesso. O identificador do canal é dado por um valor numérico gerado pelo sistema, e o estado é mantido em uma tabela *hash* que mapeia o identificador em um descritor de arquivos.

Pelo uso das políticas apresentadas, garantimos a persistência dos objetos

que representam os arquivos remotos e o servidor de arquivos e simplificamos o o ciclo de vida dos objetos.

3.2.2 Desempenho no Acesso aos Dados

A leitura e escrita de dados remotos acontecem por meio de canais. Os canais definem as operações de *read* e *write*, que modificam o conteúdo do arquivo. Além das operações de escrita e leitura, existem funções que permitem a movimentação do ponteiro no caso de canais de leitura (*skip*), e a atribuição de um valor arbitrário no caso de um canal de acesso randômico (*seek*).

Diferentemente dos arquivos remotos, e dada a assinatura das funções de leitura de dados sem o *offset* onde a próxima operação deve ocorrer, um identificador estático para o canal não possuiria todas as informações necessárias para permitir o acesso aos dados. Considerando que optamos por disponibilizar as funções sem a necessidade de se especificar o *offset*, o ponteiro do arquivo faz parte do estado do canal e deve ser mantido no servidor para cada canal aberto. Para cada tipo de canal, foi definido um POA com a política *DefaultServant* e um identificador numérico é criado para cada canal aberto. A cada chamada, o *servant* utiliza o identificador para recuperar o descritor de arquivos associado ao canal e realizar a operação sobre o arquivo correspondente.

Visando evitar o consumo indefinido de memória, um mecanismo de *leases* foi implementado de forma que um descritor de arquivo sem utilização por um determinado período de tempo seja automaticamente removido, liberando os recursos no servidor. Dessa forma, evitamos os problemas ocasionados por clientes que encerram sua conexão sem fechar o canal.

Durante a fase de prototipação, foram usados dois mecanismos para a implementação dos canais de leitura e escrita. O primeiro utilizava chamadas de métodos remotos CORBA para o envio e recepção de dados pela rede. O segundo definia a criação de soquetes TCP e um protocolo específico para a manipulação dos dados. Apesar do melhor desempenho da implementação usando soquetes, especialmente no caso da escrita e leitura de pequenos volumes de dados, optamos pela utilização apenas do mecanismo CORBA na versão final. A implementação via soquetes aumentou a complexidade do código fonte e acreditamos que a criação de um código completo para garantir

o uso desse mecanismo em um ambiente real, iria gerar um grande esforço e que, por fim, resultaria em um desempenho equiparável ao mecanismo CORBA pela complexidade das validações que seriam necessárias.

Em contra partida, um mecanismo alternativo para transferência de grandes volumes de dados foi definido. Esse mecanismo é capaz de transferir todo o conteúdo de um um arquivo diretamente entre dois servidores, sem que o cliente tenha que recuperar os canais de leitura e escrita e realizar a cópia ativamente. Com base na implementação da interface `DataTransferMethod` (Java), diversos métodos de cópia podem ser criados e utilizados nas operações de cópia, tornando esse mecanismo altamente extensível e permitindo a experimentação de diversos protocolos de comunicação. Os métodos atualmente implementados serão descritos no capítulo 4 e fazem uso dos canais CORBA, da nova biblioteca de IO de Java (NIO), e do protocolo FTP.

Para cada servidor de arquivos existente, podemos ter vários servidores de cópia, um para cada método e, no momento em que um arquivo deve ser enviado, uma conexão dedicada para algum dos servidores de cópia é estabelecida. No caso dos métodos baseados em NIO, após a realização do protocolo inicial, a transferência do arquivo é realizada através de uma chamada a biblioteca NIO, que normalmente transfere todo o conteúdo do arquivo através de chamadas ao sistema operacional, sem que a máquina virtual precise tratar os dados no nível da aplicação.

Com essas formas de comunicação, acreditamos que o sistema possui um espectro de funções de transferência capaz de suprir as necessidades dos diversos ambientes onde ele será empregado. Desde leitura e escrita de blocos em arquivos remotos, através dos canais CORBA, até a cópia de grandes volumes com um alto desempenho, através de NIO.

Realizamos testes com um cliente baseado na API de soquetes para verificar a compatibilidade do mecanismo NIO com o uso de soquetes regulares. Pudemos constatar que o uso de NIO não acarreta no tráfego de nenhum dado adicional de controle, ou seja, que é possível utilizar um servidor NIO e um cliente não implementado em Java e que trata o soquete de uma forma convencional. Visando validar o desempenho da transferência via NIO e CORBA, um estudo comparativo entre os diversos métodos foi realizado. O FTP foi usado como referência por ser tradicionalmente usado nesse cenário. Diversos testes de desempenho, considerando o FTP, CORBA e NIO, serão apresentados no capítulo 4.

3.2.3
Mount Points

Para permitir a integração de diversos servidores de arquivos distribuídos, o conceito de *mount point* foi utilizado e um diretório localizado em um servidor pode referenciar um arquivo remoto qualquer. O arquivo referenciado pode ser tanto um diretório quanto um arquivo de dados, dado que a mesma interface representa essas duas entidades. Essa funcionalidade permite a criação de uma federação do serviço, englobando diversos sistemas de arquivos locais.

Um aumento na escalabilidade do sistema é possível definindo árvores de diretórios que representam as várias máquinas existentes na grade. Considerando um bom particionamento dos dados, cada servidor será responsável por atender uma parcela dos usuários, aumentando a escala do sistema ao adicionar novos servidores e permitindo uma melhor distribuição dos dados existentes.

Para cada *mount*, ou redirecionamento, um arquivo local com o nome do *mount point* será criado contendo a localização do alvo. Esse arquivo reserva a localização no disco e evita que um redirecionamento seja sobrescrito por um arquivo criado externamente. Ao navegar em uma árvore de diretórios local, podemos encontrar o arquivo que caracteriza o *mount*. O redirecionamento deve ser seguido, permitindo que a navegação distribuída ocorra com sucesso. É importante notar que uma chamada ao servidor pode desencadear outras chamadas aos servidores responsáveis pelos redirecionamentos. Vale salientar que uma vez seguido o redirecionamento, a referência retornada aponta diretamente para o servidor referenciado e, com isso, as operações subseqüentes sobre o arquivo remoto são atendidas diretamente pelo GridFS apropriado. Dessa forma, após a recuperação de um arquivo referenciado por um *mount point*, a chamada a função `getParent`, por exemplo, irá retornar o pai do arquivo corrente, e não o diretório onde o *mount* está definido.

3.2.4
Estimativa de Taxas de Transferência

Considerando o escalonamento de processos em um ambiente de computação em grade, a informação sobre o tempo necessário para realizar a transferência dos dados, entre a origem e o candidato a processar o dado, pode ser um critério importante. Um processador mais lento pode ser utilizado, caso o custo associado à transferência do dado para um servidor com maior

capacidade de processamento seja demasiadamente elevado. Considerando a existência de réplicas da informação em diversos pontos de armazenamento, a existência de alguma estimativa do tempo de transferência dos dados também pode ser utilizada na escolha da fonte a ser usada.

Segundo [25], a modelagem analítica de sistemas, apesar de amplamente utilizada para sistemas compostos por CPU, discos e aspectos de rede a nível de roteador, não se aplica diretamente ao tratar sobre a transferência de arquivos em um ambiente demasiadamente complexo. Isso se deve especialmente pela natureza imprevisível do comportamento do sistema, e pela falta de um conjunto completo dos dados a serem modelados. Em contrapartida, esse mesmo artigo propõe uma técnica que utiliza dados históricos na tentativa de prever o comportamento futuro, considerando todo o esforço da transferência de dados fim a fim, sem tratar os componentes individualmente.

No GridFS, cada `FileServer` mantém um histórico com o somatório do número de bytes transferidos e o tempo total gasto nas operações de cópias de arquivos para outros servidores. Uma média aritmética com o total de *bytes* transferidos em relação ao tempo total gasto é calculada independentemente para cada `FileServer`. Essa média está disponível através de uma chamada de função para cada `FileServer`, em relação aos demais `FileServer`s com os quais já houve uma comunicação no passado. Ou seja, com base nas informações do histórico, os valores das médias são gerados e podem ajudar no escalonamento dos processos na grade.

3.3 Processo de Desenvolvimento

Visando validar a corretude das operações implementadas pelo sistema, um conjunto de testes automatizados foi definido e pode ser executado para constatar a conformidade do sistema em relação à sua especificação. Além disso, no processo de manutenção e na criação de novas funcionalidades, os testes podem ser re-executados de forma a garantir a harmonia entre o código pré-existente e a nova funcionalidade implementada.

Como um mecanismo complementar aos testes, e como forma de validá-los, a cobertura do código fonte exercitado durante os testes pode ser avaliada e o número de linhas efetivamente executadas pode ser medido. Assim, podemos analisar quais trechos do código não estão sendo executados e avaliar a

necessidade da criação de novos testes para contemplar uma maior abrangência do código fonte.

Os testes foram desenvolvidos com base na especificação do serviço e cada teste exercita as várias possibilidades de parametrização e contexto para cada chamada remota. Foram implementados dezenas de testes, e cada teste se subdivide em sub-testes para verificar as diferentes possibilidades na parametrização e no estado do sistema de arquivos local. Por exemplo, o teste de criação de arquivos exercita todo o diagrama de atividades mostrado na figura 3.1.

O Ant [26] foi extensivamente utilizado como ferramenta para compilação. No GridFS, ele visa facilitar a compilação em diversos ambientes e plataformas e ajudar na execução do servidor e dos testes automatizados. O alvo padrão do arquivo de configuração realiza a compilação da IDL CORBA, compila o código fonte automaticamente gerado, bem como os demais códigos fonte do sistema, e por fim, gera os arquivos jars de distribuição. Os demais alvos estão definidos no arquivo de configuração e são úteis para executar o servidor de arquivos e os testes automatizados.

3.4 Outros Mecanismos de Acesso aos Dados

Além dos canais de acesso remoto via CORBA e dos mecanismos de transferência em CORBA e NIO, dois outros sistemas foram integrados ao GridFS para permitir o suporte a aplicações legadas e prover um ponto de acesso ao sistema que independa de CORBA ou Java. Esses mecanismos não necessitam de nenhum componente especial nos clientes, nem mesmo da presença de um cliente GridFS ou da JVM.

3.4.1 Servidor FTP

Um servidor FTP[1] foi adaptado para dar suporte ao sistema. O usuário pode executar um cliente FTP para contactar o servidor e usar todos os dados armazenados na federação de servidores GridFS. Essa abordagem permite o

[1] http://incubator.apache.org/ftpserver/

acesso simplificado do usuário ao GridFS, sob o custo de ter o servidor FTP como mediador da comunicação. As requisições FTP são traduzidas para o GridFS usando o cliente GridFS internamente ao servidor FTP.

3.4.2 FUSE

O FUSE[2] é um módulo do linux que permite a implementação de um sistema de arquivos pelo usuário, de forma que o mesmo possa ser acessado através de um *mount* no sistema de arquivos local. Para permitir que aplicações legadas acessem os dados armazenados no GridFS, implementamos um sistema de arquivos com o FUSE. O sistema interage com um servidor GridFS permitindo que os dados remotos sejam apresentados em um caminho regular no disco e que aplicações legadas acessem os arquivos. Dessa forma, aplicações e comandos como *vi*, *cp*, *mkdir*, etc., podem ser executadas usando caminhos regulares, refletindo em ações nos servidores do GridFS.

A implementação atual da integração do FUSE no GridFS está disponível como prova de conceito e faz uso do FUSE-J[3], um *binding* Java para o FUSE. Devido ao elevado número de chamadas JNI, entre a implementação do FUSE e do cliente GridFS implementado em Java, acreditamos que o desempenho dessa abordagem possa ficar relativamente baixo. A implementação de um sistema de arquivos com o FUSE, que possua um cliente CORBA implementado em C++ é possível e pode trazer melhoras significativas no desempenho.

3.5 Limitações do Sistema

Esta seção tem por objetivo apresentar algumas limitações do sistema em relação às funcionalidades que poderiam ser esperadas por parte dos usuários, e que não foram atacadas neste trabalho.

[2] http://fuse.sourceforge.net/
[3] http://www.select-tech.si/fuse/

3.5.1
Operações de Remoção

Uma referência para um arquivo pode se tornar inválida caso o arquivo seja removido do sistema de arquivos local ou por um outro cliente que recuperou a referência para o mesmo arquivo. Nesse caso, qualquer operação sobre o arquivo lança uma exceção indicando que o arquivo foi removido. Se o arquivo for removido e recriado, o cliente que possui uma referência para o arquivo antigo passa a referenciar automaticamente o novo arquivo. A referência ao arquivo aponta simplesmente para um caminho no disco do servidor. Essa semântica pode não ser a desejada pelo usuário.

As operações de remoção de um elemento remoto estão definidas apenas para as folhas do sistema de arquivos local, ou seja, arquivos e diretórios vazios. As operações de remoção recursiva de diretórios resultariam em operações demasiadamente complexas, potencialmente envolvendo vários servidores e o tratamento de erros na remoção de arquivos poderia ser realizado de várias maneiras distintas. Dessa forma, optamos por deixar a implementação da semântica de remoção recursiva para o usuário. Acreditamos que a imposição de uma semântica única pelo sistema não atenderia uniformemente a todos os usuários e iria gerar ambigüidades e dúvidas por parte da interpretação da interface remota.

3.5.2
Acesso Concorrente

Diversos canais de leitura e escrita podem ser obtidos para os arquivos de dados. Semelhantemente às classes de IO de Java, nenhum mecanismo de gerenciamento do acesso concorrente aos dados foi definido. Vários escritores podem estar modificando um mesmo arquivo concorrentemente e o resultado final no arquivo depende da ordem como os escritores realizaram as operações de escrita.

Um mecanismo de bloqueio, onde um escritor necessita de acesso exlusivo ao arquivo, enquanto vários leitores podem acessar o arquivo simultaneamente (dado que nenhum escritor possui o bloqueio para o arquivo) poderia ter sido implementado. Porém, de acordo com a complexidade envolvida em tais mecanismos, optamos por manter uma semântica equivalente aos sistemas de arquivos tradicionais, onde o acesso para leitura e escrita aos arquivos é

irrestrito e pode acontecer simultaneamente por vários processos.

3.5.3 Caching

A cópia de arquivos entre servidores poderia ignorar comandos redundantes, como tentar copiar um arquivo que já foi previamente copiado e que não sofreu alteração. Porém, esse controle iria gerar um esforço relativamente alto para manter a consistência dos dados e ainda não é claro se o benefício que a implementação traria a justifica. Logo, o controle sobre as operações fica sob a responsabilidade do cliente do sistema.