4
Resultados Experimentais

Diversos testes foram realizados para avaliar o comportamento do sistema. Os testes têm o objetivo de comparar a taxa de transferência obtida pelos métodos de cópia baseados em FTP, NIO e CORBA, bem como avaliar o comportamento do sistema na presença de vários clientes concorrentes, considerando as operações de transferência de arquivos e as operações de navegação e criação de arquivos remotos. Além disso, verificamos a escalabilidade do sistema e realizamos comparações de desempenho entre os canais de acesso remoto CORBA e o NFS.

O ambiente de teste consiste em um conjunto de máquinas situadas no laboratório Tecgraf (PUC-Rio), conectadas por uma rede local de alta velocidade e, para os testes de longa distância, a Rede GIGA[1] foi utilizada entre máquinas situadas na PUC-Rio e USP. As tabelas 4.1 e 4.2 apresentam a configuração das máquinas utilizadas na PUC-Rio e na USP, respectivamente. Em redes de alta velocidade, o disco rígido utilizado possui uma grande influência nas taxas de transferências obtidas. As características dos discos utilizados nas máquinas da PUC-Rio, segundo o fabricante, podem ser vistas na tabela 4.3.

Visando avaliar a capacidade das redes envoldidas nos experimentos, a velocidade de comunicação entre as máquinas foi medida com o uso da ferramenta Iperf[2]. A tabela 4.4 apresenta as taxas de transferência obtidas nos experimentos. A coluna "leitura do disco" indica se os dados usados durante os testes foram lidos do disco e tiveram influência na taxa de transferência obtida. Segundo alguns testes realizados conjuntamente com o suporte da Rede GIGA, observamos que o tamanho do *buffer* TCP possui uma grande influência nas taxas de transferência obtidas em redes de longa distância. Durante os experimentos apresentados na tabela 4.4 o tamanho do *buffer*

[1] http://www.giga.org.br/
[2] http://dast.nlanr.net/Projects/Iperf/

Processador	1 x Intel Pentium 4 1.80GHz
Memória RAM	512 MB
Sistema Operacional	Linux version 2.4.22
Disco Rígido	IDE UltraATA/100 (ST340016A)

Tabela 4.1: Características do Servidor (PUC-Rio)

Processador	4 x Intel Xeon 3.00GHz
Memória RAM	2 GB
Sistema Operacional	Linux version 2.6.10
Disco Rígido	SCSI

Tabela 4.2: Características do Servidor (USP)

Interface	Ultra ATA
Taxa de Transferência Sustentável (MB/s)	De 24 a 31
Cache de Disco (MB)	2
Velocidade de Rotação (RPM)	7200

Tabela 4.3: Características do Disco

Origem	Destino	Leitura do Disco	Taxa (Mbps)
n1 (PUC)	n9 (PUC)	SIM	277
n1 (PUC)	n9 (PUC)	NAO	631
csgrid (PUC)	giga (USP)	SIM	301
csgrid (PUC)	giga (USP)	NAO	623

Tabela 4.4: Velocidades Obtidas nos Testes da Rede

estava definido como 1,34MB. Esse foi o valor sugerido como configuração padrão, pelo suporte, após a realização dos experimentos sobre a rede.

Os primeiros experimentos realizados mostraram uma taxa de transferência efetiva dos arquivos em torno de 30% da capacidade máxima obtida da rede. Esses experimentos foram realizados através da transferência de arquivos utilizando métodos de cópia baseados no protocolo FTP, no uso de NIO, e pela utilização de comandos de cópia no linux, como o `scp`. Como obtivemos um desempenho muito abaixo da capacidade da rede, utilizamos a ferramenta *Bonnie*[3] para realizar alguns experimentos visando verificar o desempenho dos discos locais. A taxa de escrita no disco, medida através da ferramenta, foi de apenas 28,11MB/s, o que corresponde a 35% da capacidade da rede local. Pelo uso do Bonnie, do Iperf e dos experimentos iniciais, percebemos que o sistema estava limitado efetivamente pelo desempenho dos discos e não pela rede em

[3]http://www.textuality.com/bonnie/

si.

Por outro lado, durante os experimentos iniciais, os métodos baseados em NIO e FTP foram praticamente equivalentes. Isso indica que o uso da linguagem Java como implementação para o serviço não necessariamente caracteriza um problema de desempenho para as operações de transferência de arquivos, dada a eficiência do mecanismo utilizando NIO. A documentação do NIO aponta para um desempenho elevado das operações devido à utilização de chamadas de funções de transferência ao nível do sistema operacional.

Pelo uso do comando `strace`, no linux, constatamos que durante as operações de cópia, a máquina virtual Java realiza uma chamada para a função `sendfile`, provida diretamente pelo sistema operacional. A assinatura dessa função especifica dois descritores de arquivos e o número total de dados que devem ser transferidos do descritor de origem para o descritor de destino. A documentação da função `sendfile` diz que: "*Because this copying is done within the kernel, sendfile() does not need to spend time transferring data to and from user space*". Ou seja, como a cópia é realizada internamente ao núcleo do sistema operacional, a função `sendfile` não gasta tempo transferindo os dados para a memória da aplicação. Isso confirma que o uso de NIO, sempre que possível, fornece um desempenho no mínimo equivalente a aplicações nativas e faz com que chamadas externas à máquina virtual sejam utilizadas na transferência.

A função de cópia de arquivos é uma função crítica no sistema. O desempenho dessa função precisa ser o mais alto possível e eventualmente todo o sistema poderia ter sido implementado em uma linguagem de mais alto desempenho. É importante salientar que Java, mesmo com um desempenho geral inferior a C, oferece uma ponte para chamadas a nível do sistema operacional, e é capaz de obter um desempenho elevado nas operações de cópia via NIO.

Após os experimentos iniciais, outros métodos de transferência foram adicionados e a avaliação das condições da carga na máquina de destino passaram a ser registradas durante os experimentos. Como veremos nas próximas seções, os diversos métodos de transferência visam avaliar a influência de diversos parâmetros nas operações de cópia de arquivos, testando os fatores que podem influenciar na taxa de transferência, como, por exemplo, o tamanho do *buffer* TCP em redes de longa distância.

4.1
Testes de Escalabilidade e Desempenho

A topologia das máquinas durante os testes de escalabilidade e desempenho consistia em um único servidor de destino e vários clientes concorrentes. Dessa forma, tínhamos o objetivo de simular uma sobrecarga no servidor e avaliar o seu comportamento na presença de vários clientes simultâneos. A figura 4.1 apresenta um diagrama das máquinas e mostra os fluxos de dados e controle entre elas. O mestre é responsável por comandar a cópia simultânea de diversos arquivos. Os nós n1 a n6 servem como origem de dados, e o nó n9 é responsável por receber os arquivos que estão sendo enviados. Tipicamente, utilizamos seis máquinas para o envio de dados e temos um número variável de *threads* de cópia, fazendo com que o número de arquivos recebidos simultaneamente por n9 possa chegar a várias dezenas.

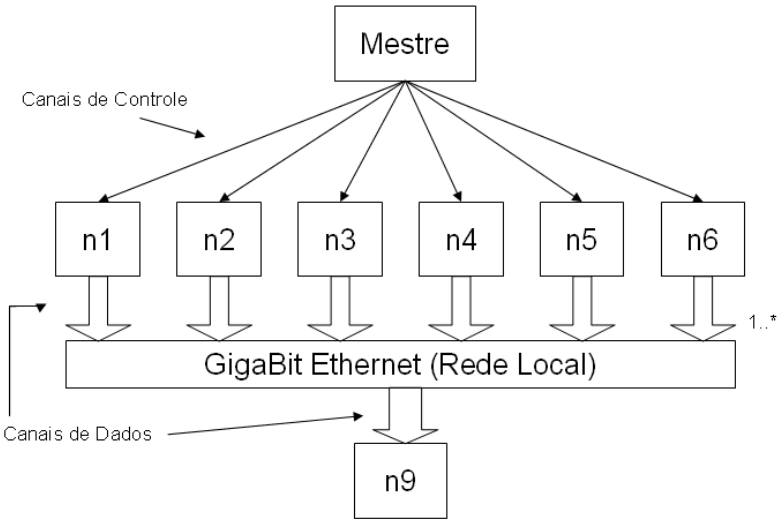

Figura 4.1: Topologia (rede local)

Um registro detalhado sobre o estado da máquina n9 foi realizado pelo uso do pacote *sysstat*[4] para coleta de informações sobre o sistema operacional Linux. Diversas informações sobre o uso de CPU, operações de leitura e escrita no disco, uso de memória e carga média da máquina foram registradas durante as diversas execuções dos experimentos, que consistiam na cópia de vários gigabytes de dados entre os clientes e o servidor. O uso de grandes volumes de dados foi propositadamente utilizado para evitar que mecanismos de *caching* de arquivos no sistema operacional tivessem uma grande influência

[4]http://perso.wanadoo.fr/sebastien.godard/

na velocidade de acesso aos dados. Dado que o GridFS tipicamente deve transferir arquivos grandes, a utilização de arquivos pequenos poderia trazer resultados não realísticos em função da disponibilidade dos dados na *cache* do sistema operacional, o que não seria garantidamente verdade durante o uso do GridFS em ambientes de produção.

Nas seguintes subseções, descrevemos cada um dos mecanismos utilizados durante a cópia dos arquivos e, em seguida, apresentamos alguns gráficos comparando os diversos métodos. Para cada método, definimos um nome que o identifica, e eventualmente alguns parâmetros utilizados para alterar as suas propriedades.

4.1.1 FTP

O método `PureFTPd` utiliza um servidor FTP[5] que permite o recebimento dos arquivos no servidor de destino. Cada uma das máquinas clientes utiliza um cliente FTP e faz uso de um *script* com comandos FTP para transferir o arquivo da máquina do servidor de origem, para a máquina do servidor de destino. Esse método foi integrado ao GridFS e permite a transferência de qualquer arquivo gerenciado pelo GridFS para uma outra área gerenciada pelo sistema, desde que os clientes e servidores FTP estejam devidamente instalados e configurados.

4.1.2 NIO

A nova biblioteca de IO (NIO) de Java [27], como visto anteriormente, permite transferências de dados com um desempenho elevado, provenientes de chamadas realizadas diretamente ao sistema operacional. A biblioteca define duas formas básicas de comunicação: o NIO bloqueante e o NIO não bloqueante. Eles diferem basicamente pela forma como as operações de IO são realizadas. A seguir, apresentamos os diversos métodos baseados em NIO, e entramos em mais detalhes sobre as propriedades bloqueantes e não bloqueantes utilizadas em cada método. Um descritor de arquivos é aberto pelo uso de alguma das classes básicas de IO de Java, como a `FileOutputStream`,

[5] http://pureftpd.sourceforge.net/

`FileInputStream` ou `RandomAccessFile` e utilizado como fonte ou destino dos dados.

Bloqueante

O NIO bloqueante é utilizado quando se deseja fazer com que as operações de entrada e saída bloqueiem o fluxo de execução do programa, ou seja, a cada operação de leitura ou escrita, a linha de execução (*thread*) do programa é interrompida até que alguma informação seja efetivamente manipulada. Esse método permite a implementação de protocolos de comunicação de uma forma simplificada, dado que tanto o fluxo quanto a ordem de chegada e envio de informações é altamente previsível.

A classe `FileOutputStream` permite a criação de um objeto para a escrita de dados no sistema de arquivos local, através de um fluxo seqüencial para o arquivo. Tipicamente, o servidor principal possui um *loop* que aceita conexões; para cada conexão estabelecida, uma *thread* é criada e realiza o protocolo pré-determinado para permitir a recepção do arquivo. Após a leitura do cabeçalho inicial, o restante da informação disponível no soquete, o arquivo propriamente dito, é transferido para o objeto `FileOutputStream` pela utilização de um canal de alto desempenho especificado pelo NIO. Ao término da transferência, a *thread* é finalizada. Vale salientar que, para realizar o recebimento de vários arquivos simultaneamente, o servidor precisa disparar várias linhas de execução, uma para cada cópia.

Realizamos duas implementações de métodos de cópia baseados no NIO bloqueante. Elas objetivam avaliar as características de algumas variantes na forma de implementação do protocolo de transferência, levando em consideração o tamanho do *buffer* TCP utilizado nas operações de transferência, bem como o número de fluxos paralelos durante a cópia de um arquivo.

– Variação do *buffer* TCP

  Em redes de longa distância, a latência de comunicação deve ser considerada no cálculo do tamanho do pacote a ser enviado pela rede, de forma a considerar o aspecto de desempenho e o uso de memória. O método `BLOCKING_NIO-<BUFFER_SIZE>` visa avaliar o impacto do tamanho do *buffer* TCP no desempenho de comunicação na Rede GIGA. Três valores para `<BUFFER_SIZE>` foram definidos gerando os seguintes métodos: `BLOCKING_NIO-SMALL_BUFFER`,

BLOCKING_NIO-REGULAR_BUFFER e BLOCKING_NIO-LARGE_BUFFER, os quais possuem os valores de 4, 128 e 2048 KBytes, respectivamente.

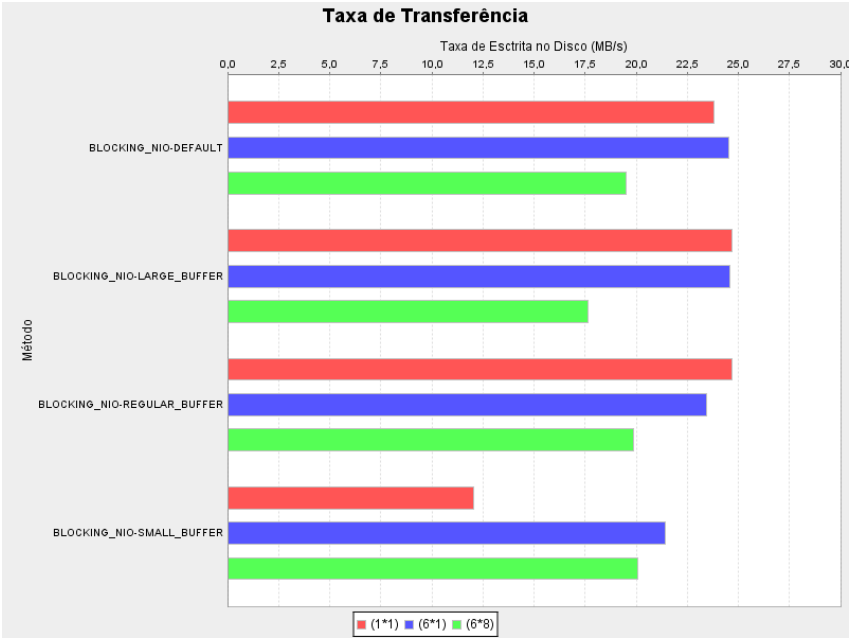

Figura 4.2: Variação do Tamanho do Buffer TCP (rede local)

A figura 4.2 apresenta a taxa de transferência obtida para esses métodos a partir de um experimento executado na rede local. A legenda indica o número de clientes concorrentes que acessavam o servidor em um determinado momento. A tupla tem o formato <c>x<t>, onde <c> indica o número de máquinas que estavam atuando como clientes e <t> corresponde ao número de *threads* disparadas por cada cliente. Logo, a legenda "(6x8)", por exemplo, indica a presença de 48 transferências concorrentes, oriundas dos seis nós. O valor do *buffer* para BLOCKING_NIO-DEFAULT é dependente da configuração da máquina e, durante nossos experimentos, estava atribuído a 700000 bytes. Pela figura, podemos observar que um *buffer* pequeno, mesmo em uma rede de baixa latência, degradou o desempenho da transferência "(1x1)" de uma forma significativa. Para o caso de múltiplas transferências simultâneas, a variação do tamanho do *buffer* não teve uma grande influência em função de termos o limite do disco como fator limitante.

Para o caso de uma rede de longa distância, como no caso da Rede GIGA entre a PUC-Rio e a USP, a latência de comunicação passa a possuir valores significativos, e o tamanho do *buffer* utilizado tem uma grande influência sobre o desempenho obtido na transferência.

A figura 4.3 apresenta a topologia dos testes na Rede GIGA. Podemos observar que a conectividade entre as máquinas da PUC-Rio e USP

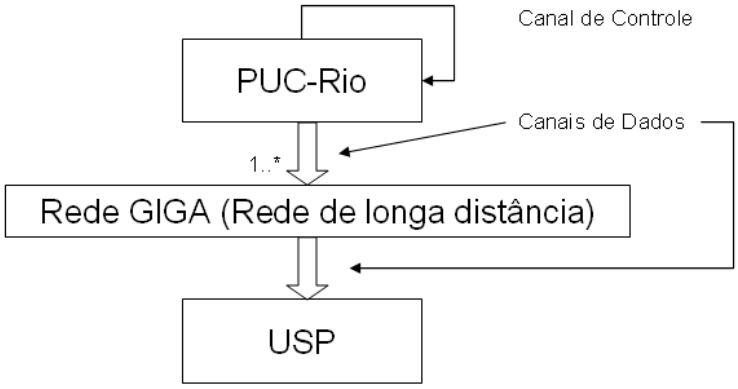

Figura 4.3: Topologia (Rede GIGA)

estava limitada a apenas dois servidores. A figura 4.4 apresenta o desempenho obtido sobre a Rede GIGA. A ferramenta `sar` não estava disponível nesse ambiente. Dessa forma, optamos por obter as medidas de desempenho utilizando os dados obtidos no nível da aplicação. Como podemos observar, um *buffer* pequeno, sobre uma rede de maior latência, degrada o desempenho de forma significativa.

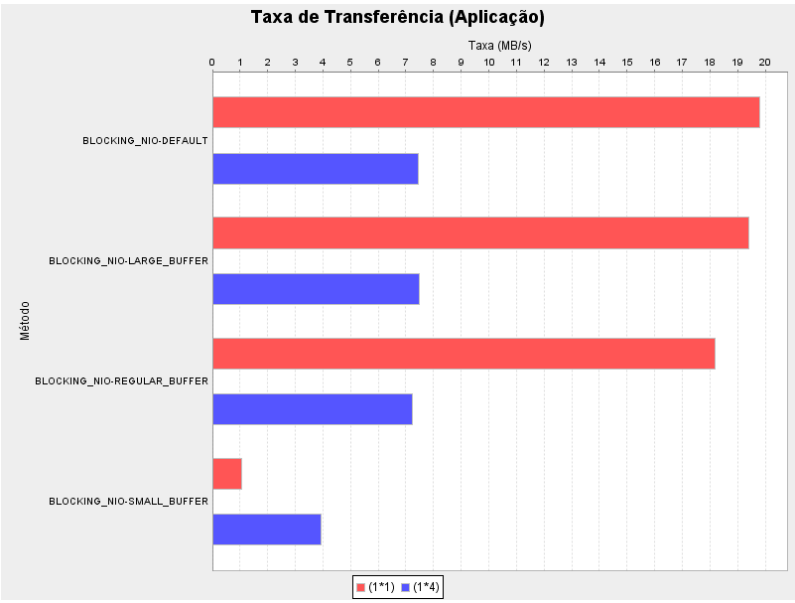

Figura 4.4: Variação do Tamanho do Buffer TCP (Rede GIGA)

O teste de paralelismo envolvia a realização de quatro cópias simultâneas entre o servidor da PUC e da USP. Pelo que observamos no teste de desempenho, esse procedimento degradou a velocidade de transferência por fazer com que quatro arquivos fossem lidos do disco em paralelo. Visando obter uma confirmação sobre o desempenho do disco, executamos um *script* com os comandos na listagem 4.1 e 4.2 para verificar o

comportamento do acesso aos dados em situações onde quatro arquivos são lidos concorrentemente, e na situação onde um único arquivo é lido por vez. De acordo com o tempo de execução das operações, temos uma taxa de leitura compartilhada de 7,46 MB/s (para quatro arquivos em paralelo), e uma taxa de 28,44 MB/s quando temos a leitura de apenas um arquivo, de forma sequencial. Ou seja, a leitura compartilhada dos quatro arquivos realmente degradou a velocidade de leitura, provavelmente por aumentar a necessidade do uso de *seeks* no disco, e por evitar a utilização otimizada do *buffer* do disco.

```
1  [csgrid:~] date ;
2  Sun Mar  5 18:42:20 BRT 2006
3  [csgrid:~] cat 1GB_1 > /dev/null & ;
4  [1] 21875
5  [csgrid:~] cat 1GB_2 > /dev/null & ;
6  [2] 21876
7  [csgrid:~] cat 1GB_3 > /dev/null & ;
8  [3] 21877
9  [csgrid:~] sleep 1; #retardo para o último cat.
10 [csgrid:~] cat 1GB_4 > /dev/null ;
11 [3]  + Done                       cat 1GB_3 > /dev/null
12 [2]  + Done                       cat 1GB_2 > /dev/null
13 [1]  + Done                       cat 1GB_1 > /dev/null
14 [csgrid:~] date ;
15 Sun Mar  5 18:51:29 BRT 2006
16 [csgrid:~] ps ;
17 PID TTY          TIME CMD
18 21119 pts/1    00:00:00 csh
19 21906 pts/1    00:00:00 ps
20 [csgrid:~] date ;
21 Sun Mar  5 18:51:29 BRT 2006
```

Listagem 4.1: Desempenho na Leitura (Paralelo)

```
1 [csgrid:~] date ;
2 Sun Mar  5 18:57:01 BRT 2006
3 [csgrid:~] cat 1GB_1 > /dev/null ;
4 [csgrid:~] cat 1GB_2 > /dev/null ;
5 [csgrid:~] cat 1GB_3 > /dev/null ;
6 [csgrid:~] date ;
7 Sun Mar  5 18:58:49 BRT 2006
```

Listagem 4.2: Desempenho na Leitura (Sequencial)

Por fim, devido à baixa taxa de transferência para um *buffer* muito pequeno, observamos que a utilização de cópias paralelas aumenta a

taxa de transferência agregada em situações onde o limite do disco não é atingido.

Como a configuração da rede havia sido realizada visando maximizar o desempenho das operações da transferência de dados, o valor padrão para o tamanho do *buffer*, 700000 bytes, trouxe bons resultados e foi utilizado durante os demais experimentos.

– Múltiplos Fluxos

O método `STREAM_BLOCKING_NIO-<STREAMS_NUMBER>` foi criado para que diversos canais NIO fossem usados paralelamente para a transferência de um mesmo arquivo. Esse método visa avaliar o impacto da existência de vários canais paralelos no desempenho geral da cópia. Para cada um dos fluxos, a informação adicional sobre o *offset* do fluxo no arquivo de destino é passada no cabeçalho e é utilizada para que a escrita dos diversos fluxos ocorra na região adequada do arquivo. Podemos observar um maior custo computacional dessa abordagem, dado que para cada fluxo existe a criação de uma nova *thread*. Além disso, a classe Java `RandomAccessFile` (que permite o acesso randômico a um arquivo) é utilizada para que as diversas partes do arquivo possam ser escritas concorrentemente e pode, potencialmente, realizar um maior consumo de recursos da máquina, devido a sua maior complexidade em relação a classe `FileOutputStream`. O parâmetro `<STREAMS_NUMBER>` indica o número de fluxos que foi usado na transmissão dos arquivos, e, consequentemente, o número de *threads* utilizado na operação de cópia.

A figura 4.5 apresenta a variação de desempenho obtida ao se utilizar diferentes números de fluxos paralelos na rede local do *cluster* do Tecgraf. Não constatamos nenhuma diferença significativa entre as três configurações e acreditamos que esse mecanismo deva apresentar um comportamento mais interessante em redes de alta latência e banda estreita, por permitir uma maior paralelização dos dados na rede. Porém, esse experimento não foi contemplado neste trabalho.

Não Bloqueante

O NIO não bloqueante permite que os objetos registrem o interesse pelos eventos de comunicação, que indicam, por exemplo, que algum cliente deseja abrir um soquete com o servidor ou que algum dos canais possui dados a serem lidos. Dessa forma, uma única linha de execução é mantida no servidor,

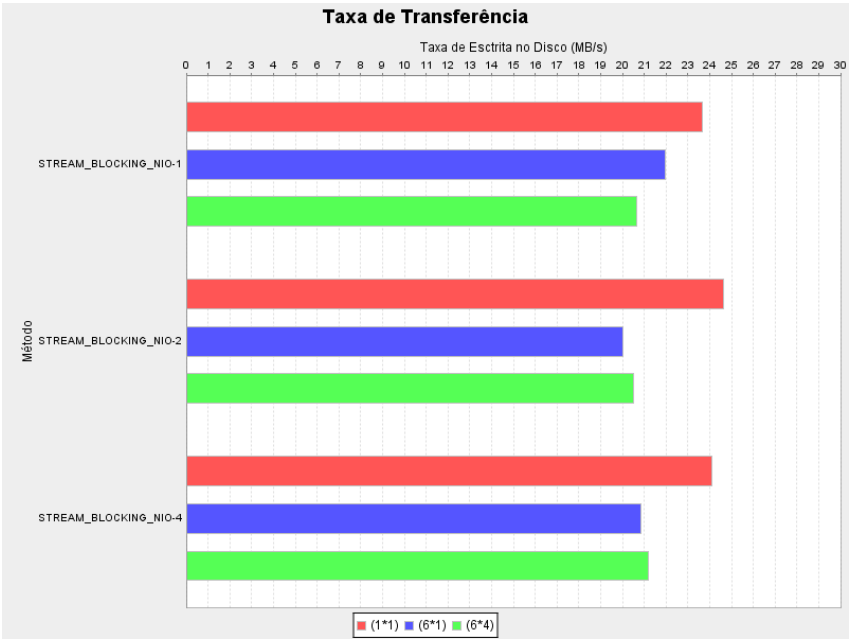

Figura 4.5: Variação do Número de Fluxos Paralelos

e sempre que algum evento acontece, uma funcão de tratamento pode ser chamada para manipular o evento específico. Devido à natureza multiplexada em que os eventos ocorrem, a implementação de um servidor correto pode requerer um tratamento um pouco mais complexo, se comparado com o NIO bloqueante. O método `NON_BLOCKING_NIO` foi implementado visando evitar a criação de uma *thread* para cada transferência. Pelo fato de que apenas uma linha de execução é mantida no servidor, independentemente do número de transferências concorrentes, acreditamos que a carga da máquina permaneça em níveis mais baixos e que um eventual ganho de desempenho seja obtido em comparação aos métodos baseados em NIO bloqueante. Como esse método não possui nenhuma parametrização específica, os resultados serão apresentados apenas na seção 4.1.4, em comparação aos demais métodos existentes.

4.1.3 CORBA

O método `CORBA-<BLOCK_SIZE>` foi criado para permitir a avaliação do desempenho dos canais de acesso remoto aos arquivos. Uma solicitação é feita a um servidor de origem, que abre o arquivo localmente e cria um canal de acesso remoto para escrever o conteúdo do arquivo no servidor de destino. Esse método é parametrizado pelo `<BLOCK_SIZE>`, que indica o número de bytes que devem ser enviados a cada chamada remota. O tamanho do bloco influencia

no número de chamadas remotas que são necessárias para enviar o arquivo. Logo, o uso de blocos maiores requer um menor número de chamadas remotas, dado um tamanho fixo de arquivo. A parametrização do método CORBA é feita de forma estática. Ou seja, o tamanho do bloco é definido antes do início da cópia e é utilizado durante toda a operação. A implementação de métodos CORBA dinâmicos, onde o tamanho do bloco varia durante a realização da cópia, é possível mas não foi implementada.

Considerando as operações de *marshalling* e *unmarshalling*, um grande número de chamadas remotas tem uma tendência a provocar uma sobrecarga no protocolo de comunicação em relação à quantidade de dados úteis enviados. Por outro lado, blocos muito grandes podem gerar a utilização de *buffers* muito grandes em memória, e eventualmente diminuir o desempenho devido à má utilização de memória das máquinas. Após algumas experiências (conforme ilustrado na figura 4.6), optamos por fixar o tamanho de bloco padrão em 32 KBytes, pois ele obteve o melhor desempenho em relação as configurações onde haviam várias transferências simultâneas.

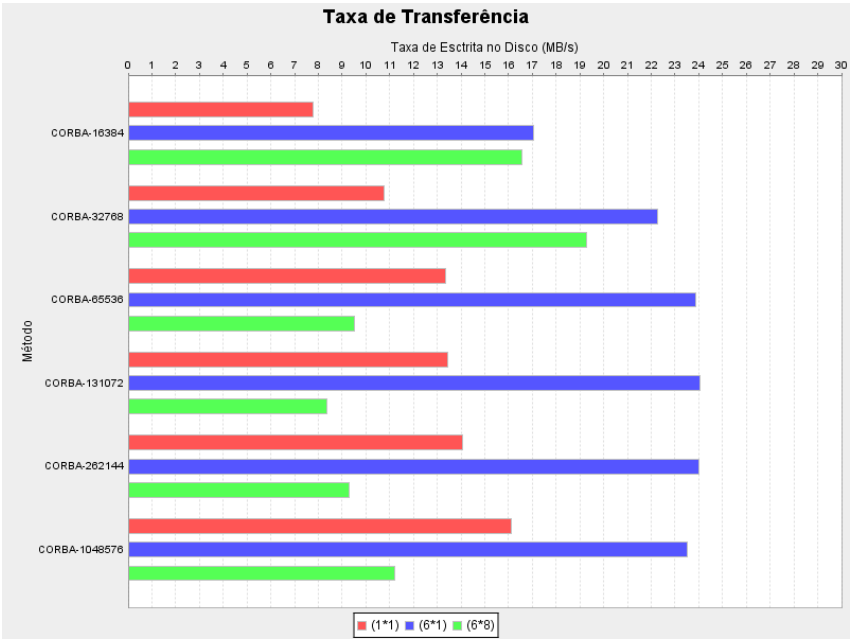

Figura 4.6: Variação do Tamanho do Bloco CORBA Utilizado nas Chamadas Remotas

Nas situações de um único cliente acessando o servidor, o método CORBA obteve um desempenho mais baixo que os métodos anteriores devido ao não paralelismo da leitura e escrita de dados pelas máquinas. Enquanto a máquina de origem lê os dados do disco, o servidor de destino permanece ocioso, e no momento em que a chamada remota é realizada, os dados são transferidos pela rede e o servidor de origem permanece ocioso enquanto ele não

recebe uma resposta do servidor de destino. A variação na taxa de transferência em relação aos tamanhos de bloco se deve à maior quantidade de carga útil, por chamada remota, para blocos maiores. Na seção 6.1 apresentamos alguns mecanismos para aumentar o desempenho do método CORBA, considerando o uso de chamadas assíncronas para a transferência de dados.

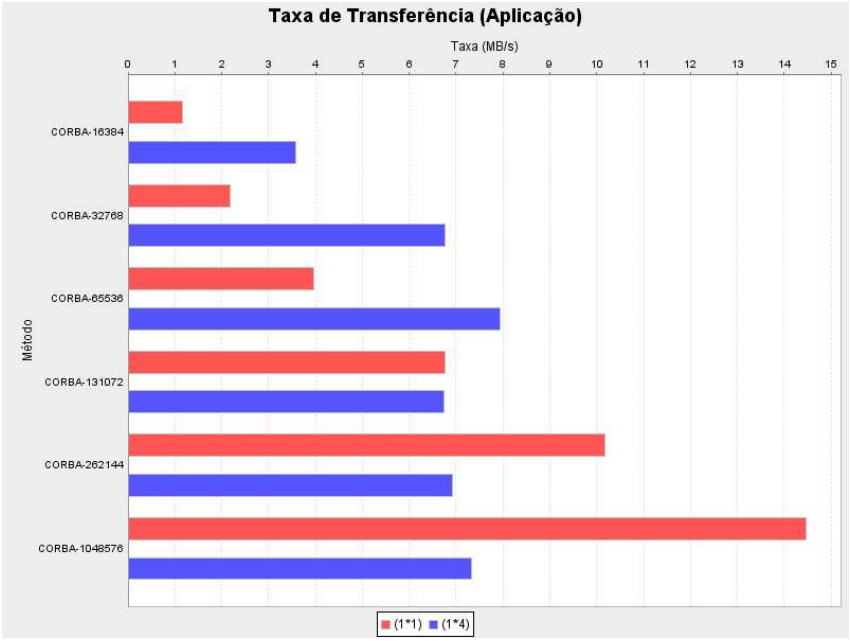

Figura 4.7: Variação do Tamanho do Bloco CORBA na Rede GIGA

A figura 4.7 apresenta a taxa de transferência obtida pelo método CORBA na Rede GIGA. O menor número de chamadas remotas, ocasionado pelo uso de blocos maiores, tem uma tendência a aumentar a taxa de transferência. O limite atingido pela configuração do bloco em 1MB se assemelha ao limite atingido na rede local do Tecgraf, e também deve estar relacionado com a capacidade de leitura dos discos. Logo, em redes de longa distância, devemos priorizar o uso de blocos maiores, aumentando a quantidade de informação útil por chamada remota. Porém, vale lembrar que um bloco excessivamente grande pode trazer problemas de falta de memória nos servidores e uma avaliação entre o comprometimento da taxa de transferência e do uso de memória deve ser realizada.

De forma semelhante aos testes de variação do tamanho do *buffer* TCP na Rede GIGA, a leitura paralela de vários arquivos fez com que a taxa agregada máxima caisse para a faixa de 7,5MB/s. Em situações onde a taxa de transferência se encontra abaixo desse valor, podemos observar que um maior paralelismo faz com que a taxa agregada aumente, como nas situações vistas pelos blocos de 64, 32 e 16 KB.

4.1.4
Comparação entre os métodos

As figuras 4.8, 4.9, 4.10 apresentam uma visão geral com todos os métodos de transferências disponíveis. Além da taxa de escrita no disco, apresentamos a utilização da CPU e a carga média da máquina. O mecanismo de cálculo da carga média, no linux, leva em consideração o tamanho da fila de processos que aguardam a disponibilidade da CPU. Como temos um conjunto de atividades limitado pela capacidade de entrada e saída do servidor, apesar de não termos uma utilização máxima da CPU, temos uma carga média relativamente alta para os mecanismos que criam *threads* baseados no número de transferências concorrentes. A carga alta não necessariamente resulta em uma queda substancial das operações de transferência, por outro lado, caso o servidor seja compartilhado por outros processos, limitados por CPU, acreditamos que uma carga mais baixa permitirá uma melhor alocação dos recursos para esses processos. De uma forma geral, recomendamos o método NIO não bloqueante como método padrão pra transferência de dados, pois ele possui uma boa taxa de transferência em todas as configurações de clientes, e ele mantém a carga da máquina em níveis mais baixos do que os demais métodos.

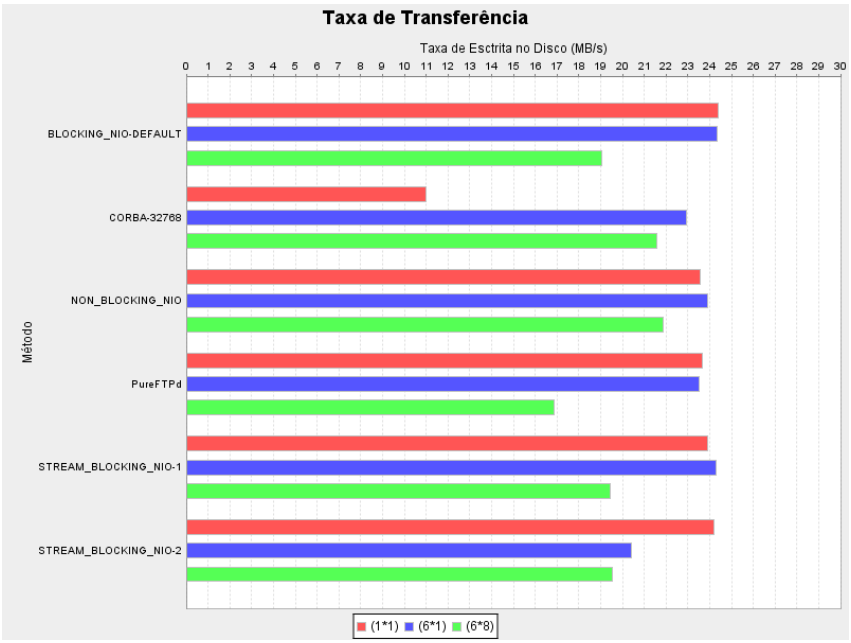

Figura 4.8: Comparação Geral dos Métodos (Disco)

A forma de realização dos testes de cópia foi definida após a execução de diversas interações sobre experimentos preliminares. Após a realização de um experimento, uma avaliação do resultado era realizada e um novo teste

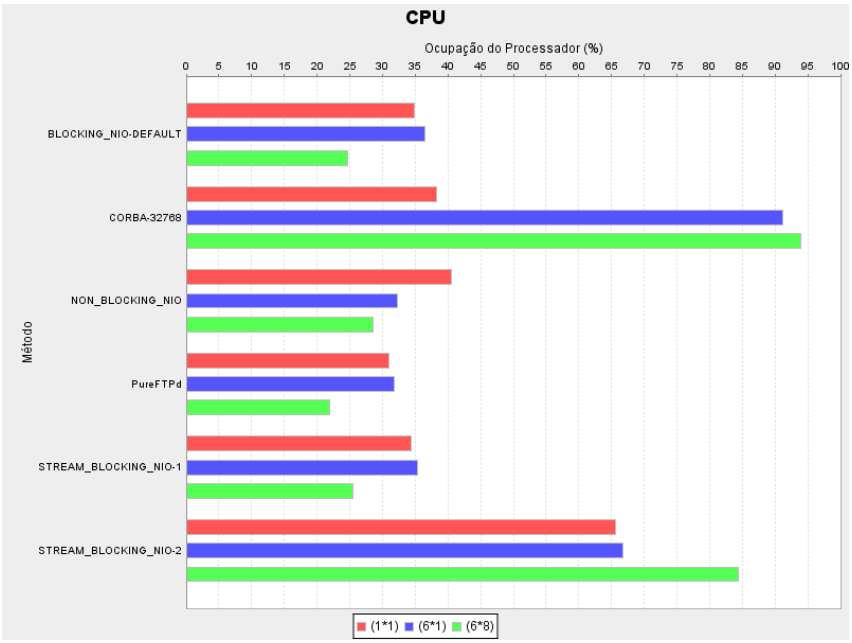

Figura 4.9: Comparação Geral dos Métodos (CPU)

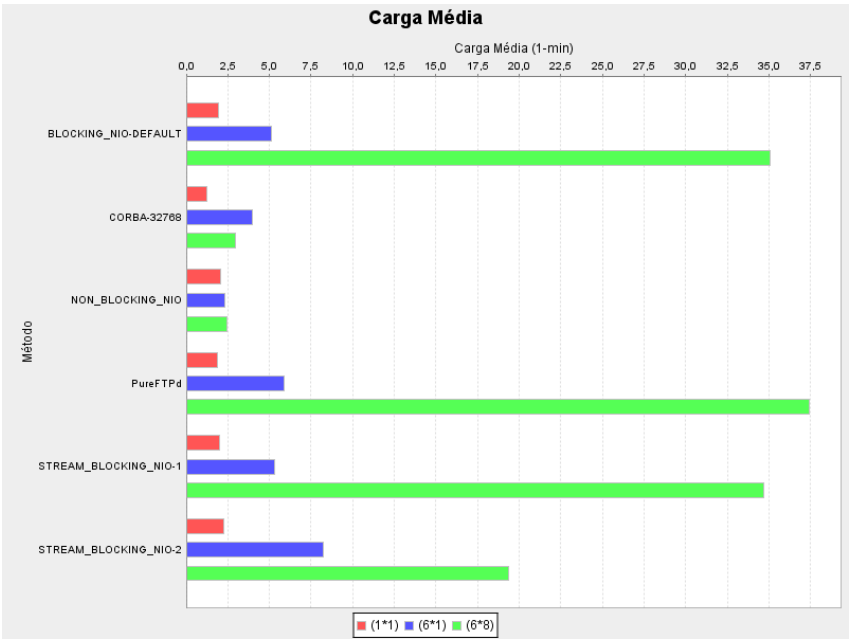

Figura 4.10: Comparação Geral dos Métodos (Carga Média)

era definido visando avaliar alguma anomalia nas informações obtidas. Por exemplo, nos testes de várias cópias simultâneas, todas as operações de cópia eram disparadas em um curto intervalo de tempo, fazendo com que o servidor recebesse todas as solicitações praticamente no mesmo instante e que as transferências ocorressem em paralelo.

Uma das anomalias observadas foi a existência de um pico na carga da máquina em um momento próximo ao final da transferência dos arquivos. Ampliando a quantidade de informação transferida, com o objetivo de avaliar o comportamento do pico na carga, percebemos que, independentemente do arquivo utilizado, a máquina sempre gerava esse pico no final da transferência. De posse dessa informação, uma nova iteração de experimentos foi realizada, fazendo com que o início das operações de cópia se dessem em um maior intervalo de tempo. Dessa forma, nós estávaamos simulando um uso mais comum do sistema, onde diversos clientes solicitam as operações de transferência de uma forma não sincronizada.

De acordo com os novos experimentos, percebemos que o pico na carga da máquina foi controlado. Por outro lado, visando garantir que, em um determinado instante de tempo, a máquina estaria efetivamente tratando um determinado número de cópias simultâneas, definimos uma função que calcula o atraso no início da transferência em função do arquivo a ser transmitido. Ou seja, durante um certo intervalo de tempo, situado próximo ao centro do período completo de transferência dos arquivos, nós tinhamos um número conhecido de operações de cópia simultâneas.

Ainda devido à presença do pico de carga no final das operações de transferência, além de distribuir o início das operações de cópia em um maior intervalo de tempo, também era comum que fizéssemos a cópia de dois arquivos de tamanhos diferentes no mesmo experimento. Assim, poderíamos tentar observar se o comportamento da máquina era influenciado pela quantidade de dados transferidas absolutamente, ou se o comportamento possuía relação com o início e fim da operação de cópia.

4.1.5
Limites no Número de Transferências Simultâneas

Os diversos métodos de cópia possuem características próprias e utilizam variações na forma de uso da rede, de gerenciamento de conexões e de uso da

CPU. As diferentes implementações são regidas por configurações distintas e o número máximo de cópias simultâneas pode variar entre os diversos métodos.

Os métodos baseados no NIO, por exemplo, tentam abrir uma conexão TCP para cada cópia sendo realizada. O método baseado em CORBA, por sua vez, normalmente reutiliza a conexão entre um mesmo processo cliente e servidor, multiplexando as diversas solicitações sobre um mesmo soquete TCP. Da mesma forma, o controle sobre os *timeouts* na tentativa de conexão e nas operações de entrada e saída podem ser feitos de forma diferente entre os diversos métodos. Após a tentativa de se atingir o limite do número de cópias simultâneas de cada método, algumas exceções foram lançadas pelo sistema, indicando falhas na operação de cópia. Assim, resolvemos evoluir o código fonte do sistema, de forma a tratar as exceções adequadamente e permitir a configuração de *timeouts* na aplicação.

Infelizmente, não conseguimos parametrizar o mecanismo de cópia baseado em NIO para utilizar *timeouts* diferentes do valor padrão. Apesar da API do NIO permitir que esses valores sejam especificados, não constatamos nenhum efeito dos valores atribuídos na execução dos experimentos. Assim, após a realização dos testes, observamos que pelos valores padrão de *timeout* e, de acordo com a utilização que damos aos nossos soquetes TCP (com um tráfego intenso de dados), os métodos baseados em NIO suportaram a utilização de 96 conexões simultâneas para um mesmo servidor, na configuração "(6x16)". Porém, ao realizar o experimento com a criação de 192 conexões, na configuração "(6x32)", o sistema passou a lançar exceções indicando falha nas operações. Vale salientar que o método `STREAM_BLOCKING_NIO-<STREAMS_NUMBER>`, por realizar a criação de vários canais paralelos, tem o seu limite de operações dividido pelo número de fluxos, pois cada fluxo atua como se fosse uma cópia independente.

Por outro lado, ao aumentar o número de transferências utilizando o método CORBA, o sistema passou a lançar exceções que indicavam que o limite de chamadas remotas simultâneas havia sido atingido. O JacORB[6], ORB utilizado durante os nossos testes, utiliza apenas uma conexão entre os processos cliente e servidor. Essa conexão é reutilizada por diversas *threads* para fazer com que as chamadas remotas sejam feitas de forma a compartilhar a mesma conexão entre o cliente e servidor. Algumas propriedades são capazes de controlar o comportamento do servidor nas situações onde muitas chamadas remotas estão sendo realizadas simultaneamente. Basicamente, as proprieda-

[6]http://www.jacorb.org

des `jacorb.poa.thread_pool_max` e `jacorb.poa.queue_max` determinam o número máximo de *threads* responsáveis por tratar as solicitações, bem como o tamanho máximo da fila de requisições. O fator crucial para determinar o comportamento do sistema na presença de muitas solicitações simultâneas é a propriedade `jacorb.poa.queue_wait`. Fazendo com que ela seja definida para "on", o servidor passa a fazer com que os clientes sejam bloqueados quando a fila de requisições está cheia, e aguardem pelo processamento das requisições pendentes. Caso essa propriedade esteja definida para "off", o sistema lança uma exceção `CORBA TRANSIENT` indicando que aquela requisição não pode ser tratada naquele momento.

Na configuração padrão do ORB, o limite da fila está definido com o valor 100, e a propriedade `queue_wait` está definida como "off". Ao executar um experimento com 192 clientes, na configuração "(6x32)", o sistema lançou algumas exceções `CORBA TRANSIENT` indicando a impossibilidade se de realizar as operações de cópia e fazendo com que o experimento não fosse concluído com sucesso. Após a configuração do ORB, definindo o valor de `queue_wait` em "on" e as propriedades `jacorb.poa.thread_pool_max` e `jacorb.poa.queue_max` em 80 e 160, respectivamente, o teste foi novamente executado e o experimento foi concluído com sucesso.

Visando verificar o comportamento com um maior número de clientes, um experimento na configuração "(6x128)" foi realizado e pudemos constatar a evolução no número de bytes escritos para 480 arquivos simultaneamente. Eventualmente, quando algumas transferências eram concluídas, os demais arquivos começavam a ser efetivamente copiados. O valor limite, 480, está relacionado ao número máximo de operações tratadas simultaneamente pelo JacORB. Como temos 6 máquinas atuando como clientes, cada uma delas inicia a transferência de 80 arquivos (valor definido pelo `jacorb.poa.thread_pool_max`) e apenas quando alguma *thread* é liberada uma nova transmissão tem início. Os valores das propriedades `jacorb.poa.thread_pool_max` e `jacorb.poa.queue_max` podem ser ajustados no arquivo de configuração do ORB. Vale salientar que após o término do experimento, todos os 768 arquivos haviam sido efetivamente transferidos e validados.

O servidor FTP utilizado possuía a limitação de aceitar apenas 50 conexões simultâneas. Dessa forma, ao realizar um experimento na configuração "(6x128)", apenas 50 cópias aconteciam simultaneamente. Os demais clientes FTP não conseguiam se conectar ao servidor e permaneciam tentando até que eventualmente conseguissem se conectar e enviar os dados. Ao término

do experimento, semelhantemente ao método CORBA, todos os 768 arquivos haviam sido copiados com sucesso.

Dessa forma, o método CORBA possui uma boa escalabilidade comparado aos demais métodos, e pode ser eficientemente configurado e utilizado para permitir a cópia simultânea de vários arquivos. Devido ao seu desempenho relativamente baixo nas situações onde apenas um cliente acessa o servidor, os métodos baseados em NIO podem ser utilizados para garantir um maior desempenho. O método baseado em FTP pode ser usado como um mecanismo alternativo, caso algum problema no método NIO impeça a conclusão da cópia com sucesso. Por fim, a existência dos diversos métodos permite uma ampla faixa de possibilidades para a transferência de arquivos e, de acordo com a rede e configuração de máquinas utilizadas, novos experimentos podem ser realizados com o objetivo de se obter o máximo desempenho. Um sistema de mais alto nível, aos moldes do Stork [19], pode ser construído para escolher dinamicamente o método de cópia a ser utilizado, ou mesmo as propriedades do método escolhido, como o tamanho do bloco CORBA, o número de canais paralelos, ou o tamanho do *buffer* TCP a ser utilizado.

4.2 Escalabilidade no Número de Referências Remotas e Canais de Acesso

Visando avaliar a eficácia do uso da política *DefaultServant* do *middleware* CORBA para o fornecimento de referências remotas para os arquivos, realizamos um experimento para observar a memória utilizada pelo sistema após o fornecimento de milhares de referências para os arquivos. A figura 4.11 apresenta o consumo de memória para o fornecimento de 100 a 100000 referências para os arquivos no servidor. Percebemos uma leve variação na quantidade de memória alocada, que corresponde apenas a uma flutuação da memória no servidor e, como esperávamos, o fornecimento de referências aos clientes não gera um maior uso de memória no servidor.

Como discutido anteriormente, no caso da criação de referências para os canais de acesso remoto, realizamos a alocação de memória referente ao descritor de arquivo mantido no servidor. O uso de memória não é significativo, pois o fator limitante para a escalabilidade do mecanismo de acesso remoto está relacionado ao número de descritores de arquivo abertos pelo processo servidor. O número máximo de descritores de arquivos abertos siultaneamente é definido pelo sistema operacional, e, atualmente, uma variável no arquivo

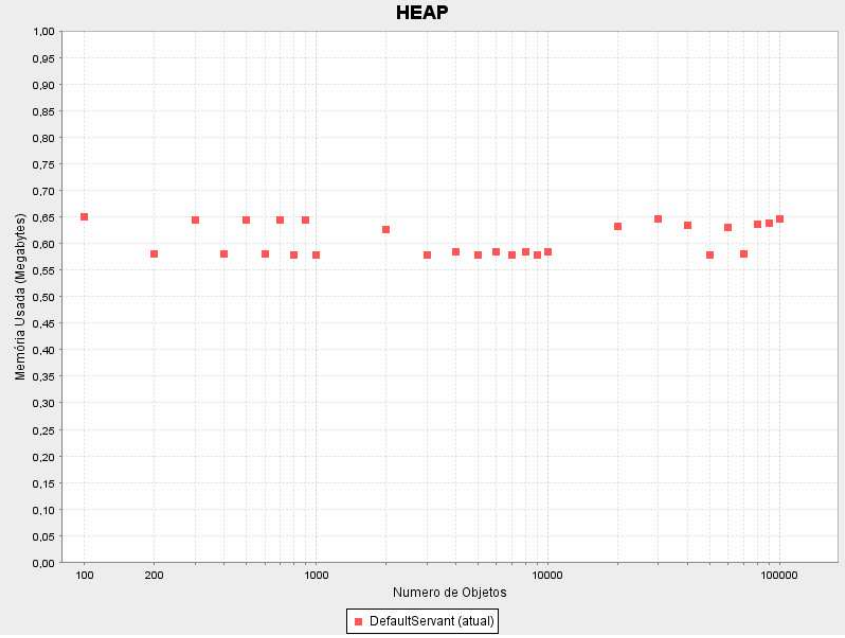

Figura 4.11: Uso de Memória com a Política *DefaultServant*

de configuração do GridFS permite que um valor máximo de canais abertos seja definido, fazendo com que a tentativa de abertura de novos canais lançe uma exceção informando que o limite foi atingido. A seção 6.1 trata algumas formas de se aumentar a escalabilidade do servidor, no que se refere aos canais de acesso remoto.

4.3
Testes em Sistemas Legados (NFS x FUSE)

Visando comparar o desempenho da nossa interface FUSE (seção 3.4.2) em relação ao NFS, utilizamos o comando `dd` no Linux para realizar a cópia de arquivos com um tamanho de bloco configurável. O arquivo de origem estava armazenado em uma área local no disco e o arquivo de destino consistia em um diretório remoto acessível via NFS e FUSE. No caso da cópia via NFS, obtivemos a taxa de 24,54MB/s, que é equivalente ao desempenho obtido pelas operações de cópia entre os servidores. No caso do uso do FUSE, a taxa de transferência foi de apenas 1,41MB/s, um valor de aproximadamente 6% da taxa comumente encontrada nas operações de cópia. Acreditamos que esse baixo desempenho esteja relacionado ao *binding* Java para o FUSE (via JNI), visto que o desempenho obtido nas operações de cópia via CORBA, mesmo considerando um único cliente e um único servidor, atinge valores na faixa de 15MB/s. De qualquer modo, a interface com o FUSE permite o uso

Protocolo	Tamanho do Bloco	Taxa (MB/s)
NFS	512 KB	20,33
GridFS	512 KB	15,14
NFS	32 KB	3,14
GridFS	32 KB	2,43

Tabela 4.5: Comparação entre o NFS e o GridFS (Acesso Remoto)

de aplicações legadas em cenários onde não é possível uma modificação da aplicação para acessar os arquivos pela interface CORBA, ou para situações onde também não é possível copiar o arquivo para uma área local. Uma implementação otimizada com o FUSE, toda em C++, também pode ser implementada para ampliar o desempenho.

4.4 Testes de Desempenho no Acesso Remoto (NFS x Canais CORBA)

Dado o baixo desempenho da nossa implementação com o FUSE, realizamos novos experimentos para comparar o desempenho do acesso randômico pelo uso do NFS e de um `RandomAccessChannel`, definido pelo GridFS. Um arquivo de 1 GB estava disponível no nó mestre do cluster do Tecgraf e era disponibilizado tanto por um servidor NFS quanto por um GridFS daemon executando no mestre. Dois programas Java acessavam esse arquivo: 1) utilizando um `RandomAccessFile` (classe definida na API da Java), e 2) utilizando um `RandomAccessChannel` (definido pela interface remota do GridFS). Os programas faziam uma série de *seeks* e *reads* para acessar várias partes do arquivo. Variamos o tamanho do bloco em dois valores: 32KB e 512KB.

A tabela 4.5 sumariza o resultado dos experimentos. Podemos observar que o tamanho do bloco também apresenta uma grande influência nas taxas de transferência obtidas e que o GridFS possui um desempenho aproximadamente 25% inferior em relação ao NFS. Acreditamos que o desempenho inferior está relacionado a necessidade da alocação e gerenciamento de memória pela máquina virtual Java.

Por fim, os diversos mecanismos de acesso aos dados oferecem uma ampla gama de possibilidades e as aplicações não legadas podem optar pelo uso dos canais de acesso remoto, ou pela cópia de arquivos entre as diversas máquinas envolvidas.