Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação
MAC-5759 – Sistemas de Objetos Distribuídos
Prof. Fabio Kon

Implementação de Interceptadores de Compressão para o JacORB

Alexey Antônio Villas Bôas <alexey@ime.usp.br>
Edgard Pevidor de Miranda <edgard@ime.usp.br>
Flávio Regis de Arruda <regis@ime.usp.br>

São Paulo, 4 de julho de 2002

Sumário

Resumo

Interceptadores são mecanismos através do qual Serviços CORBA podem interceptar o fluxo normal de execução do ORB, adicionando ou modificando a comunicação entre o cliente e o servidor ou modificando o comportamento do ORB. Atualmente interceptadores são muito utilizados para operações de log e de redirecionamento e distribuição de carga. Também foram utilizados como mecanismo de segurança, para criptografar os dados trafegados entre cliente e servidor, antes da integração de SSL nos ORBs atuais.

Nosso trabalho consiste na implementação de interceptadores que realizem compressão dos dados trafegados entre cliente e servidor, no caso do ORB JacORB. Faz parte do trabalho uma análise de performance dessa implementação, indicando os ganhos ou perdas obtidos com tal implementação. No que segue descrevemos nossas experiências durante o processo de implementação, as soluções encontradas e os dados e conclusões obtidos com as medidas feitas.

Capítulo 1

Implementação

Este capítulo tem o propósito de descrever como foi realizada a implementação de nosso projeto, exibindo os problemas que surgiram durante tal e as soluções encontradas para eles. Procuramos relatar aqui nossa experiência com o desenvolvimento e apresentar alguns fatos que tivemos a oportunidade de aprender nesse período.

As duas modalidades de interceptadores que surgem quando se fala de compressão de requisições com o `JacORB` são os de requisição (*Request Interceptors*) e os de mensagem (*Message Interceptors*). Em seguida tratamos cada um deles.

1.1 Request Interceptors no `JacORB`

Os interceptadores de requisição fazem parte do padrão desenvolvido pela `OMG` e são implementados no `JacORB`. Pelo que pudemos perceber ao estudar a seu respeito, eles são muito utilizados para `logs`, bem como para redirecionamento de chamadas ao servidor (através do lançamento de uma exceção: `ForwardRequest`).

Por serem portáveis, sua utilização foi bastante cogitada por nosso grupo. Entretanto, eles logo se mostraram inadequados para a compressão da dados, pois não permitem que os argumentos enviados em uma chamada de função sejam modificados. Na realidade, para o caso de chamadas via `stubs` e `skeletons` estáticos, não é possível nem mesmo ter acesso à lista de argumentos. Isso ocorre, pois os interceptadores são chamados após o processo de `marshaling` no cliente e antes do processo de `unmarshaling` no servidor, como nos foi explicado por e-mail por Nicolas Noffke [8], um dos desenvolvedores dos interceptadores no `JacORB`:

```
Hello Alexey,
the request arguments are not available if you use the stream-based API
(i.e.
the generated stubs/skeletons instead of DSI/DII). This is so because
the
arguments have already been marshaled to the stream *before* the
interceptors
are invoked on the client side, and will not be unmarshaled until the
interceptors have been invoked on the server side. Unmarshaling can
only be done
by the generated helper classes which aren't available in the context
where the
```

```
interceptors are invoked.

(...)
```

Porém, mesmo com o uso de `DSI/DII` não é possível alterar a lista de argumentos sendo enviada para o servidor, tornando inviável o uso de interceptadores de requisição para a compressão de dados.

1.2 Message Interceptors no JacORB

Tecnicamente, os interceptadores de mensagem são ideais para a implementação da compressão de dados desejada, uma vez que eles permitem acesso (de leitura e escrita) ao *stream* `GIOP` enviado do cliente para o servidor e vice-versa. Por outro lado, sua especificação e implementação são proprietárias do `JacORB`, tornando-os não portáveis. De fato, eles estão presentes nas primeiras versões do `JacORB` (como `1.0` e `1.1`), mas não nas mais recentes. Isso nos levou a utilizar as seguintes ferramentas durante o desenvolvimento:

- `JacORB1.0 beta 15`
- `JDK 1.2.2`
- Jakarta Ant 1.2

Vale ressaltar que devido às características apresentadas dos interceptadores de requisição, a implementação de nosso projeto foi toda feita utilizando interceptadores de mensagem. Em seguida apresentamos uma breve explicação sobre o funcionamento desse tipo de interceptador. Uma explicação mais detalhada pode ser obtida em [9].

1.2.1 Programação de Message Interceptors no JacORB

A implementação de um interceptador de mensagem para o cliente consiste da criação de uma classe que implemente a interface `jacorb.orb.ClientMessageInterceptor`, da implementação dos métodos declarados por essa interface e do registro dessa classe como um interceptador de mensagem[1]. A interface `jacorb.orb.ClientMessageInterceptor` define os seguintes métodos a serem implementados:

```
public  byte [] post_invoke( org.omg.CORBA.Object t, byte [] buf )

public  byte [] pre_invoke( org.omg.CORBA.Object t, byte [] buf )
```

onde o `buf` é o buffer contendo a requisição a ser enviada para servidor (no caso de `pre_invoke`) ou o buffer de resposta recebido do servidor (no caso de `post_invoke`). O parâmetro `t` recebido não é utilizado na nossa implementação e é usado para registros de interceptadores por objeto (não tratado neste texto). O buffer modificado pelo interceptador deve ser o parâmetro retornado – esse novo buffer é o que será enviado para o servidor (no caso de `pre_invoke`) ou passado para o cliente (no caso de `post_invoke`). Um interceptador que não faça nada simplesmente retornaria o parâmetro `buf` recebido.

O registro do interceptador é feito através de um método do `ORB`, como exemplificado abaixo:

```
ClientMessageInterceptor cmi = new ClientMessageInterceptor();
orb.addInterceptor (cmi);
```

[1] Uma etapa importante para utilização e interceptadores no `JacORB` 1.0 é sua habilitação no arquivo de configuração (tipicamente `jacorb.properties`). Através desse recurso é possível "desativar" os interceptadores, mesmo que eles estejam corretamente implementados e registrados.

Os interceptadores de mensagem do servidor são registrados utilizando o método `orb.addServerInterceptor`, em vez de `orb.addInterceptor`. Além disso, a classe que implementa o interceptador deve implementar a interface `jacorb.orb.ServerMessageInterceptor`, que declara os seguintes métodos:

```
public  byte [] post_invoke( byte [] buf )
public  byte [] pre_invoke( byte [] buf )
```

O funcionamento desses métodos é análogo ao descrito para o caso de clientes. Exemplos de implementação de interceptadores podem ser encontrados nos fontes do projeto. Para uma descrição detalhada da organização desses arquivos, consulte o Apêndice B.

Com isso, a ordem de chamada dos interceptadores é a seguinte:

1. `pre_invoke` do cliente
2. `pre_invoke` do servidor
3. `post_invoke` do servidor
4. `post_invoke` do cliente

Um cuidado importante ao se alterar o buffer `buf` é o de não corromper o header da requisição. Esse header fica localizado nos 12 primeiros bytes do buffer e corresponde à seguinte estrutura na linguagem `IDL`:

```
struct MessageHeader {
  char           magic [4];
  Version        GIOP_version;
  boolean        byte_order;
  octet          message_type;
  unsigned long  message_size;
};
```

1.2.2 A Implementação de Message Interceptors no JacORB

Pudemos perceber que a implementação dos interceptadores de mensagem no `JacORB` não estava completamente correta quando tentamos executar o exemplo de uso de interceptadores de mensagem fornecidos com o `JacORB` 1.0. Esse exemplo "simula" criptografia dos dados, invertendo os bytes do buffer a ser enviado para o servidor . O código toma apenas o cuidado de realizar tal inversão a partir da posição 12 do buffer, mantendo o header íntegro (ou seja, ele troca o byte da posição x com o byte da posição buf.length $+ 11 - x$)[2]. Ao se executar o código fornecido, entretanto, obtém-se a exceção `java.lang.OutOfMemoryError`.

Com uma inspeção na lista de discussão do `JacORB` ([8]) tivemos a impressão de que o desenvolvimento dos interceptadores de mensagem tiveram seu desenvolvimento descontinuado, em virtude de a especificação da `OMG` acerca de interceptadores ter sido terminada (e não incluir interceptadores de mensagem). Com isso, a equipe do `JacORB` parece ter deixado de lado os interceptadores de mensagem para se voltar para (os então portáveis e "*standardized*") request interceptors. Tal fato é evidenciado

[2]Nota-se também que neste caso não é necessário alterações no header, uma vez que o tamanho do buffer não é modificado pela inversão de bytes. Tal situação não ocorre no caso da compressão e uma alteração no campo `message_size` se faz necessária, como será visto oportunamente.

pelo seguinte diálogo, ocorrido em Fevereiro/Março 2000, onde Timothy Wall encontra um problema com os interceptadores, que de fato constitui a raiz do erro que obtivemos.

```
1) there is code in jacorb/orb/giop/Messages that accesses the contents
of a message before the interceptor for that message is invoked.
Messages.getRequestId tries to extract a service context length from the
buffer before the interceptor has been called.  In my case, I'm
encrypting the buffer contents, so anything that tries to read the
message buffer prior to decryption will get garbage.

....
```

Eis a resposta de Gerald Brose aos comentários de Wall:

```
Yes, that should not be done.

Let me take the opportunity to outline our future plans with regard
to interceptors:

The OMG has published the Portable Interceptors Specification (maybe
it's still a revised submission). We intend to adopt and implement
this spec. which will ensure that even code that uses interceptors
is portable between ORBs - which is obviously not the case now.

(...)

The gist of this is that those of you who use interceptors for
encryption should migrate to using SSL instead when we release
that. Also, I would advise you to rethink your interceptor
applications if you need to modify request data. While these
points may seem to be restrictive, the advantages outweigh this
by far, IMO.

Regards, Gerald Brose.
```

Posteriormente recebemos a confirmação de que havia um erro na implementação dos interceptadores de mensagem (em um mail de Nicolas Noffke):

```
Hello Alexey,
if I remember correctly (can't tell fo sure), message interceptors
where
somewhat broken. The problem was that they would not be invoked until
after the
request header had been demarshaled on the server side.

(...)
```

O problema apontado por Wall pôde ser verificado por nós ao fazermos um acompanhamento do fluxo de execução de uma requisição quando ela chega ao servidor. Uma requisição é recebida pelo servidor através de uma thread da classe `RequestReceptor`. No método `run` de `RequestReceptor` a classe `ServerRequest` é instanciada (recebendo como parâmetro o buffer vindo do cliente, e ainda não modificado pelo interceptador de mensagem do servidor). O construtor dessa classe acabará por chamar métodos que tentam ler e interpretar os dados presentes além da posição 12 do buffer, o que causará um erro. A chamada para o método `pre_invoke` do servidor só é feita bem adiante no fluxo de execução, na classe `RequestProcessor`.

A solução encontrada por nós para resolver tal problema foi fazer com que os interceptadores de mensagem fossem chamados "antes" no fluxo de execução, tanto do lado do servidor quanto do lado do cliente. Para o servidor, isso consistiu em fazer a chamada para os interceptadores no método `run` de `RequestReceptor` e tirá-la da classe `RequestProcessor` (a fim de evitar que o interceptador fosse chamado mais de uma vez).

A seguinte alteração foi feita no método `run`, da classe `RequestProcessor`
(no arquivo `src/jacorb/orb/BasicAdapter.java`).

```
(...)

while( !done )
  {
    byte [] buf = connection.readBuffer();

    // debug:
    // System.out.println("BasicAdapter: got Buffer");

    /* let the message-level interceptors do their job */

    int msg_type = buf[7];

    /** Aletracao **/

    if( jacorb.util.Environment.serverInterceptMessages())
buf = ((jacorb.orb.ORB)orb).server_messageIntercept_pre( buf );

    /** Alteracao **/

    switch( msg_type )
      {

(...)
```

O seguinte foi alterado no método `invoke_operation`, em `src/jacorb/poa/RequestProcessor.java`.

```
(...)

try {
   if( Environment.serverInterceptMessages()) {

      byte [] buf = request.getBuffer();
```

```
        /*** original:
        request.updateBuffer(controller.getORB().server_messageIntercept_pre(
        buf ));
    /*** alterado*/
    request.updateBuffer(buf);
        }
```

Com relação ao cliente, a mesma estratégia foi utilizada. Neste caso, porém foi necessário alterar as classes `Connection` e `ReplyInputStream`. Segue a alteração no método `receiveReply` em `src/jacorb/orb/Connection.java` para fazer a chamada ao interceptador:

```
(...)

if( !connected())
    throw new EOFException();
byte [] buf = readBuffer();
int msg_type = buf[7];

/**** Original: (vazio)
*/
if( jacorb.util.Environment.serverInterceptMessages())
    buf = ((jacorb.orb.ORB)orb).client_messageIntercept_post( null, buf );

// ***

switch ( msg_type )
{

(...)
```

Já em `ReplyInputStream.java` (método `wakeup`), foi necessário comentar a chamada par ao interceptador:

```
(...)

if( jacorb.util.Environment.clientInterceptMessages())
{

    /*** Original:
 buffer = ((jacorb.orb.ORB)orb).client_messageIntercept_post(target,  buffer);
    */
}
```

Com essas alterações, os interceptadores passaram a ser chamados no momento correto[3], permitindo que nossa implementação de compressão funcione.

[3]Aqui cabe uma observação com relação à implementação das chamadas para os interceptadores de mensagem. Nas

Vale apenas mais uma observação quanto à implementação de exemplo do `JacORB` (que criptografa a mensagem invertendo os bytes). Nela existe mais um erro (além do comentado acima) com respeito ao tamanho do buffer. O que ocorre é que o buffer que o servidor recebe (e conseqüentemente o método `pre_intercept` do servidor) pode ter tamanho maior que o original do cliente (nos exemplos que analisamos, o buffer foi preenchido com zeros). Em testes que fizemos, o cliente criava um buffer com tamanho 99 e o tamanho do buffer ao chegar no servidor era 128. Isso pode indicar que o `ORB` faz um alinhamento do buffer para que seu tamanho seja um múltiplo de 2. Mas de qualquer forma, é um fato que variações no tamanho do buffer podem ocorrer. Com isso, o interceptador do servidor deveria obter o tamanho do buffer (por exemplo, para reverter a inversão dos bytes) do header da mensagem e não da propriedade `length` de `buf`. No entanto, o exemplo de criptografia do `JacORB` utiliza tal propriedade, incluindo na reversão os bytes zero inseridos durante o processo de transmissão da mensagem e gerando um buffer corrompido.

Nota-se que tal problema não surge no processo de compactação, dado que em nenhum momento nossa implementação confia no valor da propriedade `length` de `buf`, mas somente no valor do campo `message_size` do header.

1.3 Interceptadores de Compressão

Nesta seção, procuramos descrever de forma simplificada a estrutura dos interceptadores de compressão implementados, destacando os pontos chave dessa implementação.

As classes `ClientMessageInterceptor` e `ServerMessageInterceptor` contém as implementações dos interceptadores em si. Elas recebem o parâmetro `buf` (conforme explicado anteriormente) e passam esse buffer para a classe `MessageUtil` para compactação ou descompactação, conforme o caso. A operação de compressão (compactação ou descompactação) realizada por cada interceptador é como indicado abaixo.

Client pre_invoke Realiza compressão dos dados a serem enviados para o servidor.

Server pre_invoke Realiza a descompressão dos dados recebidos do cliente.

Server post_invoke Realiza a compressão da resposta a ser enviada para o cliente.

Client post_invoke Realiza a descompressão da resposta recebida do servidor.

Os métodos de compactação fazem uso das classes `GZIPInputStream` e `GZIPOutputStream` para comprimir ou descomprimir os dados. Além disso, elas fazem uso da classe `MessageHeaderUtil`, que cuida da preservação do cabeçalho da mensagem (algo necessário, conforme descrito na seção 1.2.1). Essa classe possui métodos para ler o cabeçalho de um buffer, para alterar a propriedade `message_size` do cabeçalho lido e para escrever o cabeçalho novamente em um buffer (com o novo valor de `message_size`, caso este tenha sido alterado). Ela também possui um método para ler o valor de `message_size`, a fim de não gerar o mesmo problema de inconsistência de tamanho do buffer existente no exemplo de criptografia do `JacORB`, conforme visto na seção 1.2.2. Vale ainda ressaltar

alterações que foram exibidas acima, não há a pretensão de corrigir o problema com interceptadores de mensagem de forma definitiva e completa. Na realidade, não pudemos analisar todos os aspectos relacionados à alteração feita, para saber se isso causa algum efeito inesperado no `JacORB` ou não. As alterações foram testadas para nosso propósito em específico: criar um interceptador de compactação, mas não dispomos nem de tempo nem de conhecimento do código do `JacORB` suficiente para julgar se efeitos colaterais podem ocorrer. Um efeito sabido das alterações mostradas acima é a impossibilidade de acessar o argumento `t` (para o caso de interceptadores de cliente) quando o interceptador é registrado por objeto.

que a estrutura exata do cabeçalho para criação desses métodos de leitura e escrita foi obtida através da classe `org.omg.GIOPMessageHeaderHelper`, do código fonte do `JacORB`.

Capítulo 2

Análises de Performance

Esta seção descreve os testes de performance que foram realizados com os interceptadores de compressão, a fim de avaliar os benefícios (ou prejuízos) trazidos por eles quando comparados com as mesmas operações realizadas sem a compressão. É objetivo desta seção analisar esses benefícios e indicar em que situações ele ocorre e em que situações chega-se mesmo a ter prejuízo.

2.1 Procedimento Experimental

Descrevemos aqui as técnicas e procedimentos utilizados na obtenção dos dados.

2.1.1 O Programa de testes

O programa de testes faz uso da seguinte `IDL` para realização dos testes:

```
module tests {
  interface Tests {

    typedef sequence<octet> OctetSeq;

    string toUpper (in string s);
    string retString (in long i);
    OctetSeq doNothing (in OctetSeq i);

  };
};
```

Com isso, conseguimos testar variantes significativas para o processo de compressão (como, por exemplo tipos de arquivos diferentes, tais quais imagens e arquivos texto). Segue uma descrição mais detalhada do que faz cada método:

toUpper Recebe uma string de entrada, converte-a para caixa alta e a devolve para o cliente. A string de entrada é lida de um arquivo no cliente.

retString Recebe um inteiro e devolve uma string lida de um arquivo. Aqui vale ressaltar que o tempo de execução deste método pode ser retardado pela leitura do arquivo contendo a string. Entretanto isso não representa um problema, dado que esse atraso estará presente tanto para as medidas que utilizem compressão quanto para as que não.

doNothing Recebe um array de bytes do cliente (lido pelo cliente de um arquivo) e o retorna, sem alterações. Este método será utilizado para testar arquivos com diferentes taxas de compressão.

O cliente (classe `Client`) faz apenas chamadas para esses métodos no servidor e cronometra seu tempo de execução. Para a cronometragem, foi implementada a classe `Crono`. Nota-se que o cronômetro é sempre disparado imediatamente antes de a chamada remota ser feita e parado imediatamente depois que ela retorna.

2.1.2 Procedimento

Os experimentos foram realizados utilizando três máquinas diferentes, cujas configurações seguem:

Máquina sabio Pentium III de 1 Ghz (Dell Dimension 4100) com 512 MB de memória RAM 133Mhz. Placa de rede Intel Adapater 10/100. O sistema sendo executado nessa máquina era Debian/GNU Linux (woody), com kernel 2.4.18.

Máquina groo Pentium III de 133 MHz com 32 Mb de memória RAM. Placa de rede Realtech 10/100. O sistema sendo executado nessa máquina era Debian/GNU Linux (potato).

Máquina Mafalda A máquina mafalda das ET's do IME.

A conexão entre as máquinas `sabio` e `groo` é feita por um switch 10/100. A conexão entre `sabio` (ou `groo`) e `mafalda` é pela internet, com um link de 256Kb (supostamente mais lenta).

Foram realizados 4 tipos de testes, conforme abaixo.

Teste 1 Uma chamada para o método `doNothing`, passando como parâmetro uma imagem `JPEG`. Seu tamanho original é de 202271 bytes e seu tamanho compactado de 200304 bytes, apresentando uma taxa baixa de compactação: $1,0\%$.

Teste 2 Uma chamada para `toUpper`, passando como parâmetro uma string de texto grande. A string tem tamanho original de 357426 bytes e tamanho compactado de 42032 bytes, apresentando uma alta taxa de compressão: $88,2\%$.

Teste 3 Uma chamada para `toUpper`, passando como parâmetro uma string de texto pequena. Seu tamanho original é de 13 bytes e seu tamanho compactado é de 43 bytes, apresentando uma taxa de compressão de $-15,4\%$.

Teste 4 Uma chamada para `retString`, recebendo como parâmetro uma string de texto grande. Seu tamanho original é de 357426 bytes e compactado de 42032 bytes, apresentando uma alta taxa de compressão de $88,2\%$.

Para cada um desses testes foram tomadas 6 baterias de medidas, da seguinte forma. Para cada teste, tomamos medidas na mesma máquina (na máquina `sabio`), em máquinas diferentes, mas ligadas por uma rede muito rápida (a máquina `sabio` como servidora e a máquina `groo` como cliente) e em máquinas diferentes ligadas por uma rede mais lenta (a máquina `mafalda` como servidora e `sabio` como cliente). Para cada configuração de máquinas, tomamos 10 medidas para a versão com os interceptadores ativos (com compactação) e 10 medidas com os interceptadores inativos (sem compactação).

Todas as medidas foram feitas em milisegundos.

2.2 Resultados Obtidos

Nesta seção apresentamos os resultados obtidos para cada um dos 4 testes realizados. Para cada conjunto de dados, descartamos os *outliers* e calculamos sua média e seu desvio padrão (já sem levar em conta os *outliers*). As fórmulas utilizadas para esses cálculos são as seguintes:

$$Media = \frac{\sum_{i=1}^{n} x_i}{n}$$

$$DesvioPadrao = \sqrt{\sum_{i=1}^{n} \frac{(x_i - Media)^2}{n}}$$

Ainda, para cada teste, apresentamos um gráfico comparativo das médias calculadas.

Teste 1 Os dados obtidos para este teste foram como segue.

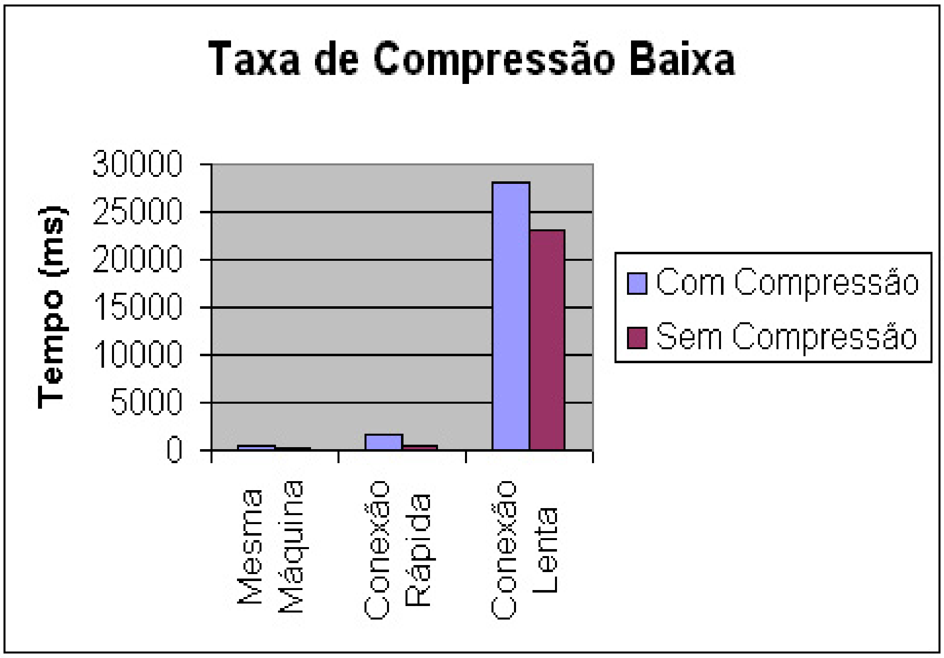
- Com compressão:

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	496	1737	28733
2	455	1709	27768
3	469	1669	29221
4	465	1688	27092
5	441	1677	29004
6	451	1650	27386
7	463	1666	31370
8	445	1664	27038
9	465	1660	29025
10	436	1637	27036
Média	458,6	1675,7	28033,67
Desvio Padrão	17,27	29,25	947,80

- Sem compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	128	564	22948
2	139	536	22987
3	144	536	23169
4	140	535	22909
5	147	534	23193
6	145	555	23016
7	138	682	26770
8	145	536	22954
9	147	539	22913
10	145	542	22901
Média	141,8	541,83	2998,88
Desvio Padrão	5,83	10,51	109,94

A partir desses dados, podemos construir o seguinte gráfico:



Teste 2 Os dados obtidos para este teste foram como segue.

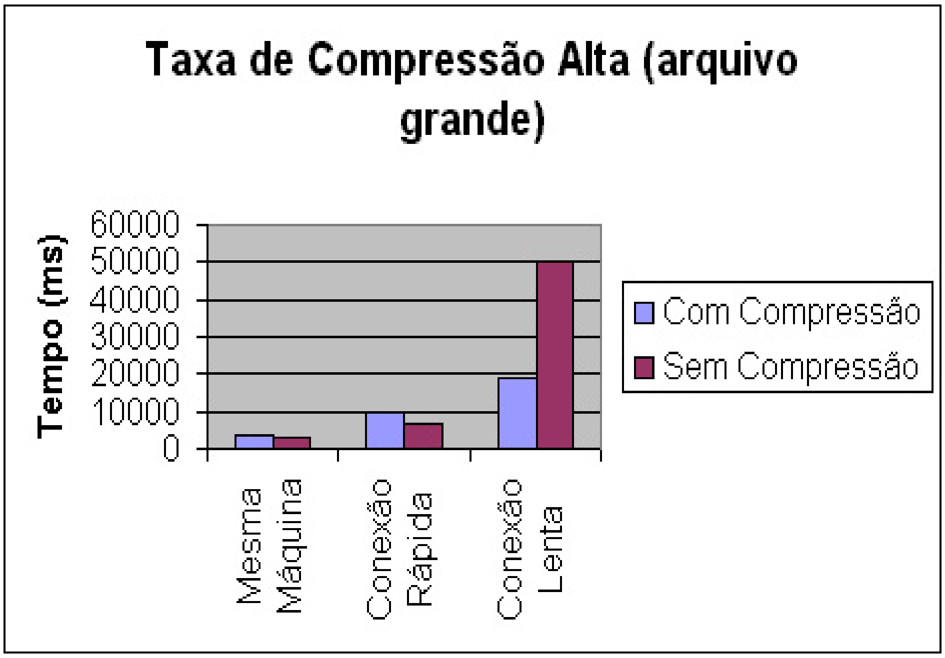
- Com compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	4452	10451	23218
2	6307	9331	19281
3	4533	10614	19011
4	2382	8957	17818
5	4468	7757	18554
6	2952	9826	19622
7	2256	9653	17698
8	4558	9204	18507
9	2362	9823	18884
10	3274	9776	21023
Média	3470,78	9737,22	18933,11
Desvio Padrão	1028,71	543,16	1004,12

- Sem compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	2144	5891	52038
2	3827	6328	48999
3	3743	5210	49124
4	2756	7348	52748
5	2387	7826	49453
6	2639	6830	50227
7	2356	6329	51559
8	2093	7593	50138
9	3287	7293	49591
10	3083	6837	49368
Média	2831,5	6748,5	50324,5
Desvio Padrão	630,44	816,47	1324,48

O gráfico construído é o que segue:



Teste 3 Os dados obtidos para o terceiro teste são os seguintes.

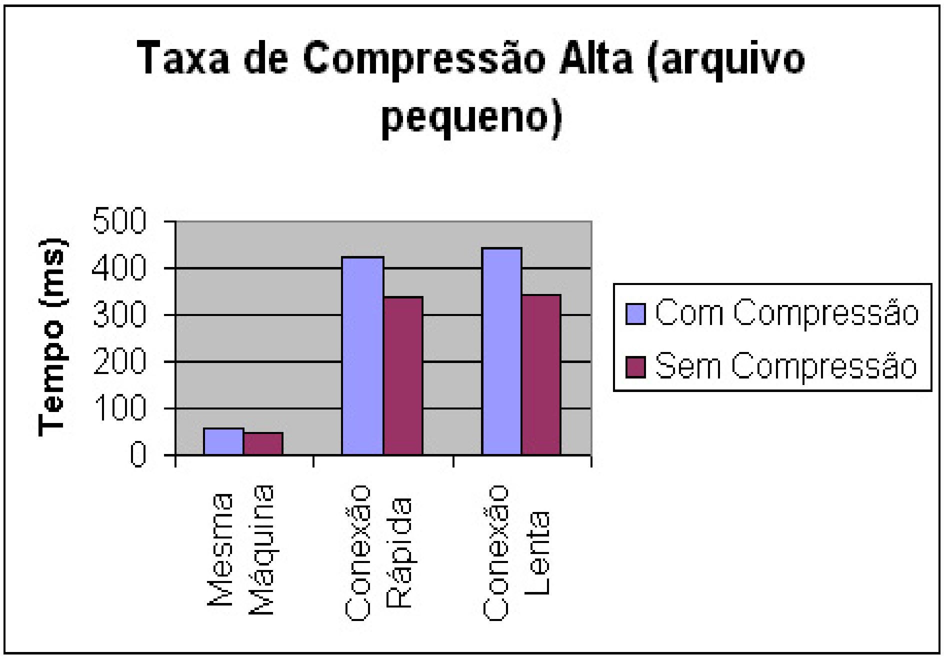
- Com compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	80	442	1923
2	63	418	579
3	52	408	417
4	51	409	473
5	53	469	400
6	52	405	405
7	59	407	431
8	52	467	475
9	69	403	436
10	67	414	370
Média	57,56	424,2	442,89
Desvio Padrão	7,14	25,62	61,15

- Sem compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	62	355	1352
2	40	333	297
3	40	339	291
4	59	343	320
5	56	331	285
6	40	338	301
7	40	334	332
8	56	333	455
9	41	336	428
10	57	334	372
Média	49,1	337,6	342,33
Desvio Padrão	9,54	7,06	62,45

Seu gráfico é apresentado a seguir:



Teste 4 Para o quarto teste obtivemos os seguintes resultados.

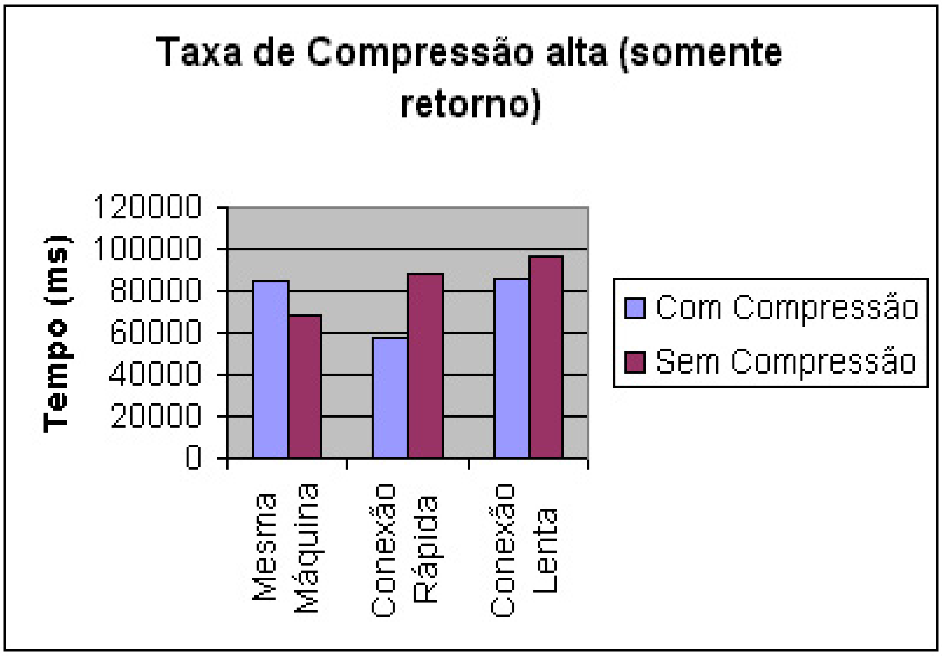
- Com compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	82753	48112	87432
2	83058	97319	78248
3	81254	80470	49853
4	70885	89028	93842
5	86797	48421	92387
6	90668	51562	97367
7	134351	47923	87438
8	130618	49873	71232
9	92719	49648	84327
10	92719	50283	83742
Média	85106,63	57257,78	86223,89
Desvio Padrão	7334,04	15774,03	8071,95

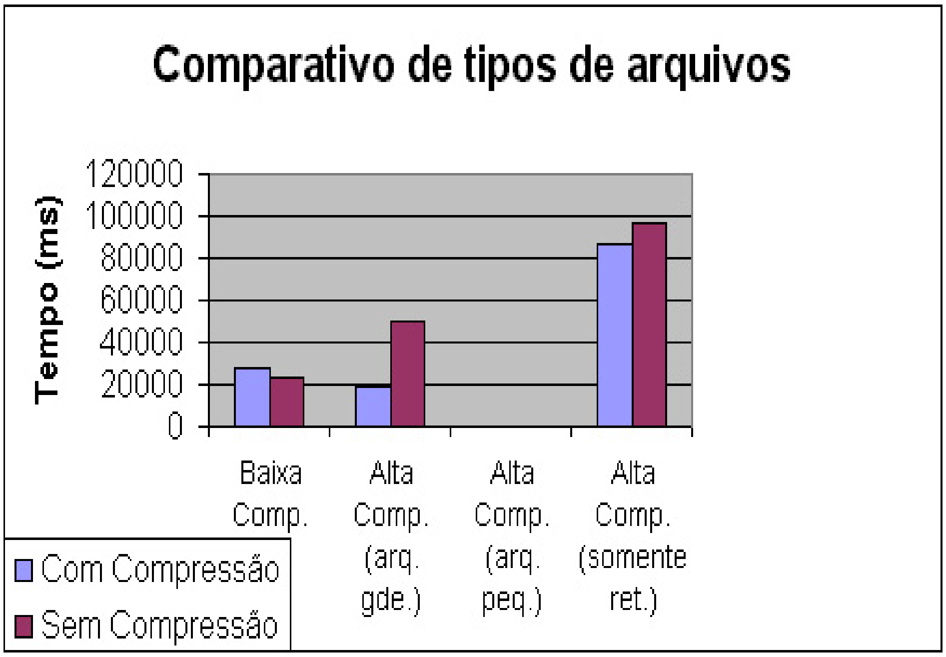
- Sem compressão

Medida	Mesma Máquina	Conexão rápida	Conexão Lenta
1	60285	87324	100276
2	92987	89265	101837
3	47696	90273	97376
4	76832	79203	96439
5	64383	82387	98032
6	73595	90237	94387
7	56644	96254	92384
8	83156	83644	93845
9	52426	98547	93527
10	80050	84759	97438
Média	68805,4	88189,3	96554,1
Desvio Padrão	14751,43	6051,71	3055,13

O gráfico construído a partir desses dados é o que segue.



Os resultado obtidos acima nos sugerem que o uso de compactação só é de fato eficiente no caso em que a conexão é lenta. Isso nos leva a apresentar um comparativo entre os diferentes testes (contendo dados com diferentes taxas de compressão) apenas para o caso de conexão "lenta"(entre `mafalda` e `sabio`):



2.3 Conclusões

A análise dos dados obtidos nos mostra que há uma grande variação da performance fornecida pelo uso de compressão nas mensagens trocadas entre cliente e servidor. Primeiramente, nota-se que a compressão traz um prejuízo no caso de execução na mesma máquina, ou mesmo para a vasta maioria dos casos em que havia uma rede muito rápida (exceto no Teste 4). Esse resultado se mostra conforme esperado, pois nessas situações a comunicação dos dados não é "gargalo" de execução do sistema. Por outro lado, a processamento para a compactação se faz presente e seu tempo de execução não é desprezível, mesmo em máquinas potentes (como a máquina `sabio`). Com isso, pode-se concluir que, em geral a compactação trouxe prejuízo para o caso de execução na mesma máquina e em redes de alta velocidade.

Outro cenário já bastante distinto é o que envolve uma rede de velocidade inferior (um link de 256 Kb com a Internet). Nesse cenário, o desempenho das requisições que tiveram suas mensagens compactadas se mostra mais vantajoso. Porém, o gráfico intitulado "Comparativo de tipos de arquivos" nos mostra claramente que mesmo nos casos de rede mais lenta, a compactação só se mostra vantajosa para certos tipos de dados. Para arquivos com boa taxa de compactação (como arquivos texto com um tamanho razoável) a compactação se mostrou ser de excelente resultado. Porém, para arquivos com taxa baixa de compactação (como a imagem `JPEG` utilizada ou arquivos muito pequenos) a compactação ainda se mostrou ineficiente.

Apêndice A

Como Executar os Testes

Este capítulo descreve o processo de instalação e execução dos testes utilizados neste trabalho.

A.1 Instalação

Conforme mencionado no Capítulo 1, as ferramentas que foram utilizadas durante o desenvolvimento foram as seguintes:

- `JacORB1.0 beta 15` (pode ser obtido em `http://www.jacorb.org/releases/ftp/`).
- `JDK 1.2.2` (pode ser obtido em `http://java.sun.com`).
- Jakarta Ant 1.2 (pode ser obtido em `http://jakarta.apache.org/builds/jakarta-ant/release/`).

O projeto foi desenvolvido e testado em Debian/GNU Linux.

Deve-se seguir as instruções de instalação de cada uma dessas ferramentas antes de utilizar o programa de testes fornecido. Com relação ao `JacORB` é necessário recompilá-lo, substituindo os seguintes fontes pelos fornecidos:

- `${JACORB_HOME}/src/jacorb/poa/RequestProcessor.java`
- `${JACORB_HOME}/src/jacorb/orb/BasicAdapter.java`
- `${JACORB_HOME}/src/jacorb/orb/Connection.java`
- `${JACORB_HOME}/src/jacorb/orb/giop/ReplyInputStream.java`

onde `${JACORB_HOME}` denota o diretório base da instalação do `JacORB`.

Para a compilação dos testes, basta ir para o diretório `compact` e executar o `ant`:

```
$ cd ${HOME}/compact
$ ant
```

onde `${HOME}` denota o diretório raiz da instalação do projeto.

A compilação das ferramentas utilitárias se dá da mesma forma:

```
$ cd ${HOME}/util
$ ant
```

A.2 Execução dos Testes

A execução dos programas de testes se dá de uma forma bastante simples. Para executar o servidor, basta utilizar os seguintes comandos[1].

```
$ cd ${HOME}/compact/classes
$ jaco tests.Server <ior>
```

onde `ior` é o arquivo onde será gravada a `IOR` gerada pelo Servidor. A execução do cliente se dá de forma semelhante, com `ior` sendo o mesmo arquivo fornecido para execução do servidor:

```
$ cd ${HOME}/compact/classes
$ jaco tests.Client <ior>
```

Após a execução do cliente, surge um menu modo texto com opções para execução dos diferentes métodos de teste, conforme descrito na seção 2.1.1.

[1] Nota-se que devido ao uso de interceptadores de mensagem não é possível utilizar o serviço de nomes, de forma que não é preciso iniciá-lo neste ponto

Apêndice B

Organização dos Arquivos Fornecidos

Este Apêndice procura descrever a estrutura de diretórios fornecida. Esta é como segue.

docs Contém a documentação do projeto.

jacorb Contém os arquivos fonte do `JacORB` que foram modificados para a implementação dos interceptadores de compressão, conforme descrito na seção 1.2.2. A estrutura de diretórios em que se encontra cada arquivo reflete a estrutura original em que o arquivo se encontra na distribuição dos fontes do `JacORB`.

compact Contém os fontes da implementação dos interceptadores e das classes de teste. São eles:

- **Client.java** O cliente.
- **ClientMessageInterceptor.java** A implementação do interceptador de mensagem do cliente.
- **Crono.java** A classe de cronômetro.
- **MessageHeaderUtil.java** A classe de manipulação do header.
- **MessageUtil.java** A classe de compressão.
- **Server.java** O servidor.
- **ServerMessageInterceptor.java** A implementação do interceptador de mensagem do servidor.
- **TestsImpl.java** A implementação da interface `tests`.
- **tests.idl** O arquivo `IDL`.

util Contém uma classe que realiza a compressão de um arquivo, utilizando o mesmo algoritmo implementado nos interceptadores. Ela foi utilizada em testes internos.

Referências Bibliográficas

[1] BEA. Developing java interceptors. http://edocs.bea.com/wle/wle50/rli/javadevp.htm, 2002.

[2] Luigi Verde Carlo Marchetti. Portable interceptors. http://www.dis.uniroma1.it/irl/pi.htm, 2002.

[3] R.Baldoni C.Marchetti, L.Verde. Corba request portable interceptors: A performance analysis. In *nd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 208–217, September 2001.

[4] Andreas Vogel Gerald Brose and Keith Duddy. *Java Programming with CORBA*. John Wiley & Sons, 2001.

[5] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[6] OMG. Interceptors published draft with corba 2.4+ core. http://cgi.omg.org/cgi-bin/doc?ptc/2001-03-04, 2001.

[7] Sun. Java idl and java rmi-iiop technologies: Using portable interceptors (pi). http://java.sun.com/j2se/1.4/docs/guide/idl/PI.html, 2002.

[8] JacORB team. Jacorb develper mailing list. http://lists.spline.inf.fu-berlin.de/mailman/listinfo/jacorb-developer.

[9] JacORB team. Jacorb programming guide 1.0. http://www.jacorb.org/documentation.html, 2000.

[10] JacORB team. Jacorb programming guide 1.4. http://www.jacorb.org/documentation.html, 2002.