4
Avaliação

A flexibilidade e a extensibilidade da arquitetura dos servidores de dados estruturados foram avaliadas de duas maneiras distintas: com a implementação de estudos de caso e através de testes de desempenho. Para avaliar a flexibilidade da arquitetura foram utilizadas diferentes tecnologias na implementação dos servidores de dados como, por exemplo, as interfaces abstratas e o *custom marshal* com o uso de *Protocol Buffers*. A extensibilidade da arquitetura, por sua vez foi avaliada através da utilização de diferentes tipos de dados estruturados, inclusive com dados utilizados em aplicações científicas reais. Essas avaliações da arquitetura são detalhadas nas próximas seções.

4.1
Estudos de Caso

A arquitetura de servidores de dados foi utilizada em dois estudos de caso com aplicações desenvolvidas pelo Tecgraf/PUC-Rio em parceria com a Petrobras. O primeiro estudo de caso apresentado utiliza uma simplificação de um tipo de dado de poço de petróleo que é utilizado por duas dessas aplicações. Esse estudo de caso ilustra o uso de múltiplas visões sobre um dado pois uma das aplicações utiliza apenas atributos de nível gerencial para a geração de relatórios, enquanto que a outra utiliza o dado em processamentos através de algoritmos científicos. No segundo estudo de caso, o *middleware* de integração de aplicações científicas *OpenBus* foi utilizado para orquestrar a comunicação entre aplicações que trocam dados entre si. A arquitetura foi implementada para permitir que as aplicações pudessem compartilhar os seus dados uma com a outra. As subseções seguintes apresentam esses dois estudos de caso em detalhes.

4.1.1
Dados de Poço

Um poço de petróleo é uma perfuração na superfície terrestre utilizada para produzir petróleo. Durante o ciclo de vida de um poço, que vai desde o projeto e perfuração até o seu abandono, passando pela produção, são obtidos

muitos dados que são processados de alguma forma para gerarem informações sobre esse poço e seu reservatório. Esses dados são armazenados para que sejam trabalhados ou verificados posteriormente e para esse armazenamento podem ser utilizados diversos meios como, por exemplo, um banco de dados relacional, fitas ou arquivos.

A partir dos dados de um poço, é possível extrair diversos tipos de informação de diversas grandezas físicas. Por exemplo, um geofísico se interessa por questões relacionadas à permeabilidade ou resistividade do poço. Já uma pessoa de um departamento financeiro ou jurídico pode se interessar pela estimativa de produção do poço ou pelos *royalties* que devem ser pagos por essa produção.

Como essas informações são muito diversificadas e usadas por diferentes tipos de usuário, é razoável que exista a mesma diversidade em relação às aplicações que venham a processar esses dados. Uma aplicação voltada para o departamento financeiro ou jurídico, por exemplo, pode usar os dados de poço para gerar relatórios ou planilhas de custos, apenas usando atributos que venham de um banco de dados relacional. Um geólogo, por sua vez, pode utilizar uma aplicação que processe o dado obtido de uma fita ou de um arquivo, gerando alguma saída em memória ou em disco. É bastante razoável imaginar que, apesar de a entidade ser a mesma, poço de petróleo, os usuários utilizam visões bem diferentes dos dados dessa entidade.

Em suma, aplicações distintas têm visões distintas sobre os dados de poço. Se todo o dado relativo a um poço for transferido para cada uma dessas aplicações, existe a chance de boa parte desse dado ser desperdiçada, pois seriam utilizadas apenas algumas partes que são do interesse real do usuário de uma dessas aplicações. Portanto, esse é um cenário onde o uso do mecanismo de múltiplas visões sobre um mesmo dado se aplica fortemente.

A idéia deste estudo de caso é utilizar a arquitetura para que apenas as partes do dado que sejam realmente do interesse das aplicações, e de seus diferentes usuários, sejam transferidas. As visões aqui apresentadas são, em verdade, representações simplificadas de definições utilizadas por duas aplicações desenvolvidas pelo Tecgraf/PUC-Rio em um convênio com a Petrobras[1].

A primeira visão representa o dado de poço na forma utilizada por funcionários do departamento jurídico ou financeiro e é chamada de *corporativa*. Essa visão é composta, principalmente, por atributos do tipo texto que são utilizados diretamente para exibição na tela ou mesmo para a geração de relatórios. A listagem 4.1 apresenta a definição desta visão.

[1] A IDL completa deste estudo de caso encontra-se no apêndice A.3

Listagem 4.1: Visão corporativa de um poço

```
1 valuetype EnterpriseWell supports sdeacidl::DataView {
2   public Header fHeader;
3   public sdeacidl::types::Point fLocation;
4   public string fName;
5   public Situation fSituation;
6   public unsigned long fANPCode;
7   public Classification fClassification;
8 };
```

Já a segunda visão, chamada de *científica*, representa a visão utilizada, por exemplo, por geólogos e geofísicos, para processamento por meio de algoritmos científicos. Essa visão possui atributos que representam vetores de vetores de números em ponto flutuante. Como esses vetores armazenam uma grande quantidade de dados, a transferência dessa visão é mais demorada do que a da visão corporativa. A listagem 4.2 apresenta a definição desta visão.

Listagem 4.2: Visão científica de um poço

```
1 struct Curve {
2   string fName;
3   Header fHeader;
4   sdeacidl::types::NumberList fValues;
5 };
6 typedef sequence<Curve> CurveList;
7
8 valuetype ScientificWell supports sdeacidl::DataView {
9   public Header fHeader;
10  public sdeacidl::types::Point fLocation;
11  public CurveList fCurves;
12 };
```

A navegação hierárquica tem como raízes as bacias sedimentares onde poços são perfurados. A partir de uma bacia podem ser obtidos os seus descendentes que são os poços de petróleo propriamente ditos. Os poços de petróleo não possuem descendentes; a partir deles o usuário já pode escolher uma visão e solicitar a transferência desta.

Uma abordagem alternativa para a navegação nos poços que oferecem visão científica seria representar as curvas que compõem um poço como descendentes deste. As curvas de um poço representam valores obtidos por instrumentos para uma determinada propriedade do poço. Por exemplo, existem curvas de porosidade, permeabilidade, densidade, etc. Essas curvas são processadas em alguns algoritmos por geólogos ou geofísicos e, por isso, é necessária apenas na visão científica. Sendo assim, a hierarquia teria um nível a mais para os dados de poço que oferecessem a visão científica; aqueles que oferecessem apenas a visão corporativa ficariam inalteradas. Desse modo, poderíamos transferir apenas a visão representando uma única curva e a visão científica do poço seria menor. A listagem 4.3 apresenta a visão científica de um poço

sem as suas curvas; cada curva é definida por um novo *valuetype* conforme apresentado na listagem 4.4.

Listagem 4.3: Visão científica de poço sem o atributo representando suas curvas

```
1 valuetype ScientificWell supports sdeacidl::DataView {
2   public Header fHeader;
3   public sdeacidl::types::Point fLocation;
4 };
```

Listagem 4.4: Visão de uma curva de poço

```
1 valuetype Curve supports sdeacidl::DataView {
2   public string fName;
3   public Header fHeader;
4   public sdeacidl::types::NumberList fValues;
5 };
```

Tratada a questão da hierarquia, era necessário decidir a quantidade de serviços de dados a ser utilizada. A primeira solução seria o uso de um único serviço de dados que oferecesse todas as visões do dado de poço. Essa solução facilita bastante o trabalho das aplicações que trabalham com mais de uma dessas visões, pois todas elas estariam acessíveis nesse único serviço.

Há certos casos, porém, em que é desejável o uso de mais de um serviço de dados. Por exemplo, quando as visões do dado são originadas de fontes diferentes seria possível criar um serviço de dados para cada uma dessas visões. A visão corporativa do poço poderia ser construída a partir de informações provenientes de um banco de dados relacional. Já a visão científica, poderia ser materializada a partir de algum tipo de arquivo que seja lido pela aplicação por meio de bibliotecas especializadas. Seriam criados, então, um serviço para acessar o banco de dados e outro que utilizasse uma essas bibliotecas especializadas.

Conforme dito anteriormente, todo dado é representado por uma chave unívoca. Para o dado de poço, existe uma informação que é bem conhecida por seus usuários e que identifica um poço de maneira única. Essa informação é a *sigla* do poço que é uma informação curta, textual e que não é reaproveitada por mais nenhum outro poço, mesmo que um poço seja excluído.

Para os casos em que se deseja utilizar poços sem necessariamente utilizar a sua sigla, as aplicações devem investigar os metadados procurando pelas informações de que necessita. Por exemplo, os metadados podem conter o estado do poço e aplicações gerenciais poderiam gerar relatórios apenas de poços que já estejam em produção. Para isso, a aplicação deveria solicitar apenas a visão corporativa dos poços que contenham em seus metadados a informação de estado de produção.

As visões do dado de poço apresentadas possuem informações comuns, que são compartilhadas por ambas. Por exemplo, a localização geográfica de um poço é interessante tanto para se descobrir a quem pagar os *royalties* pela produção quanto para se obter informações sobre a formação do solo onde o poço se localiza, através de informações da bacia sedimentar na qual o poço se localiza. Desse modo, tanto a visão *corporativa* quanto a visão *científica* devem possuir um tipo comum que ofereça os atributos e operações que sejam comuns a esses dados.

O tipo *Well* foi criado para conter as informações comuns à todas as visões do dado de poço. Como a utilidade do tipo é evitar a duplicação das informações em mais de uma visão, não faz sentido instanciar uma visão desse tipo. Desse modo, o tipo *Well* foi definido como um *valuetype* abstrato, contendo apenas operações que são comuns às visões de um dado de poço. As operações retornam informações comuns como um cabeçalho que descreve o poço e a lista de pontos da geometria do poço. A sigla do poço, utilizada como chave, não precisa ser incluída neste tipo, pois todas as visões de dados precisam implementar a interface *DataView* e esse tipo já oferece uma operação para se obter a chave unívoca do dado.

Listagem 4.5: Definição de um super-tipo para um poço

```
1 abstract valuetype Well {
2   Header getHeader();
3   sdeacidl::types::Point getLocation();
4 };
```

A visão *corporativa* oferece as informações que são usadas tipicamente para a geração de relatórios gerenciais ou exibição na tela. Essa visão estende o tipo *Well* e ainda implementa o tipo que representa uma visão de dados, *DataView*. As informações contidas nesse tipo, chamado de *EnterpriseWell*, são o nome do poço, a situação (ou estado) do poço, sua classificação e um código atribuído pela ANP (Agência Nacional do Petróleo, Gás Natural e Biocombustíveis).

Listagem 4.6: A visão corporativa do poço

```
1 valuetype EnterpriseWell : Well supports sdeacidl::DataView {
2   public string fName;
3   public Situation fSituation;
4   public unsigned long fANPCode;
5   public Classification fClassification;
6 };
```

Assim como a visão corporativa, a visão *científica* também estende o tipo *Well* e implementa o tipo *DataView*. Essa visão, definida pelo tipo *ScientificWell*, é composta por um atributo que representa as curvas do poço e,

cada curva, é composta por vetores de vetores de números em ponto flutuante. Sendo assim, a transferência de uma visão dessas é bem mais lenta do que a de uma visão corporativa.

Listagem 4.7: A visão científica do poço

```
1 valuetype ScientificWell : Well supports sdeacidl::DataView {
2   public CurveList fCurves;
3 };
```

Este estudo de caso foi importante para se ilustrar o uso de um dado que é representado por diferentes visões de acordo com as aplicações e com os usuários que o utiliza. A IDL completa com todos os dados utilizados neste estudo de caso encontra-se no apêndice A.3. A visão científica descrita aqui foi utilizada em todos os testes de desempenho que serão descritos adiante.

4.1.2 OpenBus

Este estudo de caso apresenta um uso da arquitetura no compartilhamento de dados entre aplicações através do OpenBus. O OpenBus é um barramento para a integração de aplicações científicas construído através de uma arquitetura orientada a serviços (SOA). Seguindo essa arquitetura de serviços, as funcionalidades do barramento são providas através de três serviços principais, chamados de *serviços básicos*. Dentre essas funcionalidades estão autorização e autenticação, registro e busca de ofertas de serviço e troca de mensagens entre aplicações. A figura 4.1 ilustra o barramento e seus respectivos componentes.

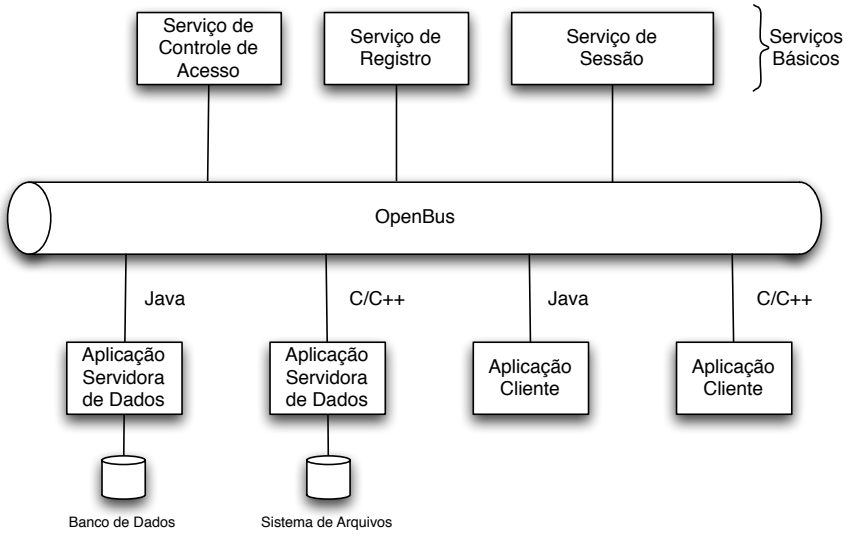

Figura 4.1: Arquitetura do OpenBus

A integração das aplicações ao barramento também é realizada através de serviços. Uma aplicação que queira ser acessada por outras deve registrar algum tipo de serviço no barramento, de acordo com a natureza da integração pretendida. Por exemplo, uma aplicação que queira servir dados no barramento deve registrar algum tipo de serviço de dados.

Os serviços básicos do OpenBus são *componentes de software* construídos sobre o modelo de componentes distribuídos SCS, assim como a arquitetura proposta neste trabalho. Também como os serviços de dados propostos, um serviço do OpenBus é uma faceta de um componente SCS. Sendo assim, um servidor de dados que siga a arquitetura proposta, pode ser integrado ao OpenBus diretamente, sem a necessidade de ajustes.

Para este estudo de caso, a aplicação científica utilizada para servir seus dados foi o WebSintesi. O WebSintesi é uma instância do *framework* CSBase [16] e é um ambiente integrador de aplicações de geologia e geofísica; os dados oferecidos pelo WebSintesi, portanto, são pertencentes a esse domínio.

O WebSintesi provê facilidades para o gerenciamento de recursos e execução de aplicações em um ambiente de computação heterogêneo e distribuído. O usuário pode organizar os seus dados através de arquivos que estão organizados em projetos. Os projetos dos usuários são abstrações de um sistema de arquivos remoto e, em consequência disso, a navegação hierárquica é bastante natural neste domínio. Esses projetos ficam armazenados em um servidor centralizado e, assim, um usuário pode acessar os seus dados independente de sua localização; basta acessar o servidor em que seus dados estão armazenados.

A integração realizada nesse estudo de caso tinha como objetivo compartilhar os projetos dos usuários com outras aplicações, de modo que estas pudessem obter os dados dos usuários e também criar novos dados nos projetos. Como os dados dos projetos dos usuários estão todos centralizados no servidor WebSintesi, foi criado um servidor de dados, seguindo a arquitetura proposta neste trabalho, que é responsável por acessar os projetos pertencentes a um servidor WebSintesi. O servidor de dados foi criado com um único serviço de dados que é responsável por acessar todos os projetos.

Uma aplicação que deseja ler ou gravar dados em um projeto de um usuário utiliza os mecanismos fornecidos pelo barramento para encontrar o serviço de dados em questão e, através dos mecanismos providos pela arquitetura de servidores de dados, fazer a transferência do dado. Como os dados oferecidos são os projetos do WebSintesi e estes possuem naturalmente uma hierarquia, a navegação hierárquica foi implementada de maneira direta. Os metadados dependem do domínio da instância do CSBase que está servindo

seus dados. Por exemplo, os tipos de arquivo oferecidos dependem do domínio da instância. Alguns metadados, entretanto, são comuns e são definidos pelo próprio CSBase como por exemplo o dono do arquivo, as datas de criação e de última alteração do arquivo, as permissões e o tamanho.

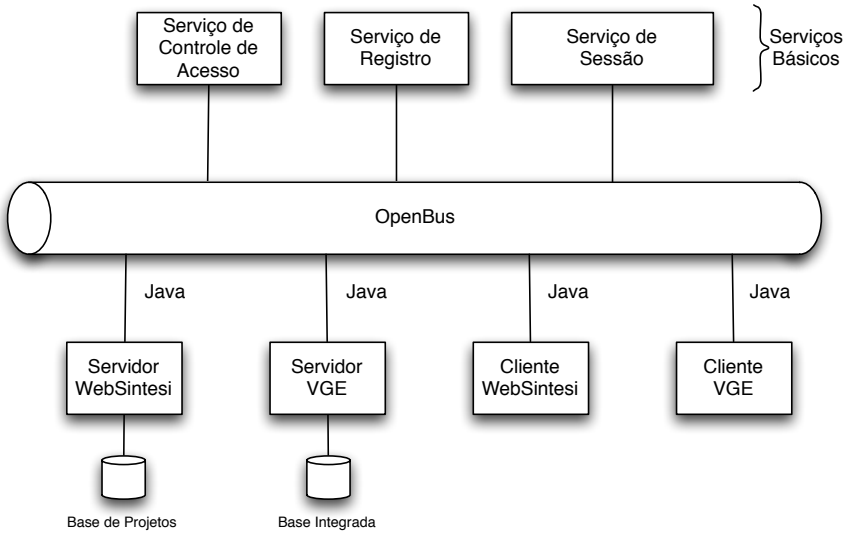

Figura 4.2: Aplicações integradas através do OpenBus

Uma particularidade nesta integração foi o fato de serem realizadas transferências de dados não estruturados. Apesar de não ser o foco da arquitetura proposta, foi necessário transferir um dado integralmente do servidor para o cliente através de vetores de *bytes* e o uso de uma visão estruturada acarretaria uma perda significativa de desempenho. Por exemplo, se fosse utilizada uma visão estruturada, mesmo que composta apenas de um atributo que fosse um vetor de *bytes*, ainda assim seria necessário passar pelos mecanismos de empacotamento e desempacotamento de CORBA, além da necessidade de carregar esses dados em memória para a criação da visão, antes de efetivamente transferí-los.

Um dos dados que deveriam ser transferidos possuía algumas características que inviabilizavam a utilização de visões estruturadas. O SEG-Y pode ser incluído na categoria de *bulk data*, pois um arquivo nesse formato tem, corriqueiramente, dezenas de *gigabytes*. O próprio arquivo traz em seu cabeçalho os detalhes do seu formato binário como, por exemplo, se os números em ponto flutuante estão no formato IBM ou IEEE ou se os *bytes* estão ordenados como *little endian* ou *big endian*. Desse modo, é comum existirem aplicações que processam um dado SEG-Y linearmente, ou seja, o processamento é efetuado ao mesmo tempo em que a leitura está sendo realizada. Os dados processados vão sendo descartados, pois este processamento é *oneway*, ou seja, não é

necessário voltar para uma parte do dado já processada. A saída desse processamento pode ficar tanto em memória, no caso de resultados de tamanho pequeno, quanto em disco, também escrevendo a saída linearmente.

Um outro tipo de dado que poderia ser transferido de maneira não-estruturada é o dado de poço no formato LAS. Embora não possa ser incluído na categoria de *bulk data*, este dado também pode ser processado linearmente, como o SEG-Y. Ainda como o SEG-Y, esse tipo possui um cabeçalho que descreve o seu formato binário e, portanto, a aplicação que o recebe tem todas as informações para processá-lo corretamente.

Para atender à necessidade da transferência de dados não-estruturados foi criada uma visão chamada *UnstructuredData* que é formada por um conjunto de atributos que informam à aplicação sobre um *socket* no servidor de dados. De posse dessa informação, a aplicação pode abrir uma conexão com esse *socket* e solicitar a transferência de todo o conteúdo do dado.

Listagem 4.8: Definição da visão de um dado não-estruturado

```
1 valuetype UnstructuredData supports sdeacidl::DataView {
2   public string fHost;
3   public unsigned long fPort;
4   public boolean fWritable;
5 };
```

A transferência do dado através de um *socket* deve seguir um protocolo bem definido. O OpenBus oferece um protocolo e uma implementação deste através de uma biblioteca chamada *File Transfer Channel (FTC)*. Essa implementação, como o próprio nome já diz, é limitada ao uso de arquivos; dados em memória, por exemplo, não podem ser transferidos diretamente. O FTC provê o esqueleto de uma implementação de um *socket* servidor que deve ser utilizada pelo serviço de dados. Do lado do cliente, ou seja, das aplicações interessadas em obter o dado, o FTC oferece um objeto que representa um arquivo com operações de leitura e escrita como se este arquivo estivesse sendo utilizado localmente. Existem implementações da biblioteca FTC para as principais linguagens de programação utilizadas pelo OpenBus: Lua, Java e C++.

Esse estudo de caso, utilizando aplicações reais, mostrou que a flexibilidade da arquitetura é bastante satisfatória. Foi possível realizar a transferência de dados não-estruturados sem que fosse necessária nenhuma mudança na arquitetura, mesmo esse tipo de representação não sendo o foco desta. A utilização desta visão não-estruturada pode ser utilizada como uma alternativa para a integração de aplicações legadas, pois não seria necessário alterá-las para tratar dos dados na forma estruturada. A arquitetura permitiu que a implementação criada obtivesse o alto desempenho necessário para a transferência dos seus dados.

4.2 Desempenho

A avaliação do desempenho proporcionado pela arquitetura foi realizada através da implementação de *sete experimentos*. Em cada experimento foi criado um servidor de dados e uma aplicação que solicita a transferência dos dados oferecidos. Os servidores são compostos por uma única faceta de serviço de dados, que oferece uma única visão de dado, que é a visão *científica* do dado de poço que foi apresentada na seção 4.1.

As aplicações iniciam o processo de transferência do dado pela navegação hierárquica. Para simplificar os experimentos, todos os dados oferecidos pelos servidores encontram-se como raízes da hierarquia. Ao executar a operação *getRoots*, então, uma aplicação está de posse de todos os descritores dos dados oferecidos. Como esses descritores contém a chave unívoca dos dados, a aplicação pode obter a visão destes imediatamente.

A transferência da visão dos dados oferecidos é solicitada duas vezes seguidas pela aplicação. Na primeira, a aplicação solicita a visão oferecida para cada uma das chaves obtidas sucessivamente, através da operação *getDataView*. Em seguida, a aplicação solicita a visão passando a lista completa de chaves, através de uma única chamada à operação *getDataViewList*. Os tempos de transferência são registrados logo após a sequência de chamadas à operação *getDataView* e logo após a chamada única à *getDataViewList*.

Tanto os servidores de dados quanto as aplicações foram desenvolvidos na linguagem de programação Java. A lista completa de versões do *software* utilizado nos experimentos encontra-se na tabela 4.1.

Tabela 4.1: *Software* utilizado nos testes de desempenho

Software	**Versão**
Java Virtual Machine	1.6.0
JacORB	2.3.0
SCS	1.0.0
Protocol Buffers	2.0.3

Os servidores de dados foram sempre executados em máquinas distintas das aplicações clientes, para que o processamento de um não influenciasse no do outro. Essas máquinas utilizadas, entretanto, possuíam a mesma configuração, que é descrita na tabela 4.2. A única exceção foi a execução da aplicação do experimento *future* que, como será apresentado adiante neste capítulo, faz uso de duas *threads* de execução e, por isso, foi utilizada uma máquina com dois processadores. A configuração dessa máquina é descrita na tabela 4.3. A comunicação entre as aplicações era realizada através de uma rede com uma

taxa de transferência efetiva de, aproximadamente, 94000 kBps (*kilobytes* por segundo).

Tabela 4.2: Configuração das máquinas usadas nos testes de desempenho

Processador	Intel(R) Pentium(R) 4 HT 3.0 GHz
Memória	1 GB
Sistema Operacional	CentOS 4.4
Versão do *Kernel*	2.6

Tabela 4.3: Configuração da máquina usada nos testes de desempenho do experimento *future*

Processador	Intel(R) Pentium(R) D 3.40 GHz (Dual Core)
Memória	2 GB
Sistema Operacional	CentOS 4.4
Versão do *Kernel*	2.6

Os tempos obtidos com a transferência dos dados em cada experimento são apresentados na tabela 4.4. A coluna *Um A Um* apresenta os tempos obtidos com a transferência dos dados um a um, através de sucessivas chamadas à operação *getDataView*. Já a coluna *De Uma Vez* apresenta os tempos obtidos com a transferência dos dados de uma vez, através de uma única chamada à operação *getDataViewList*. Esses tempos representam a média aritmética de 10 execuções de cada experimento. Em cada um destes, foram transferidas 10000 visões científicas de poço, todas já carregadas em memória, cada uma com 440 *bytes*, em um total de 4,2 *megabytes* transferidos.

Tabela 4.4: Tempos obtidos nos testes de desempenho

Implementação / Tempo (segundos)	**Um A Um**	**De Uma Vez**
valuetype	84	83
abstractinterface	84	82
custommarshal	118	117
protobuf	1656	1656
protobuf-speed	543	543
custommarshal-protobuf	1123	910
future	484	484

Estes mesmos tempos também estão expostos no gráfico 4.3. Essa visão dos tempos facilita a comparação entre os diversos experimentos realizados.

O primeiro experimento foi criado para se testar o uso de *valuetypes* e, por causa disso, foi denominado *valuetype*. O tipo *DataView*, que é uma interface abstrata na definição da arquitetura, foi alterado, nesse experimento,

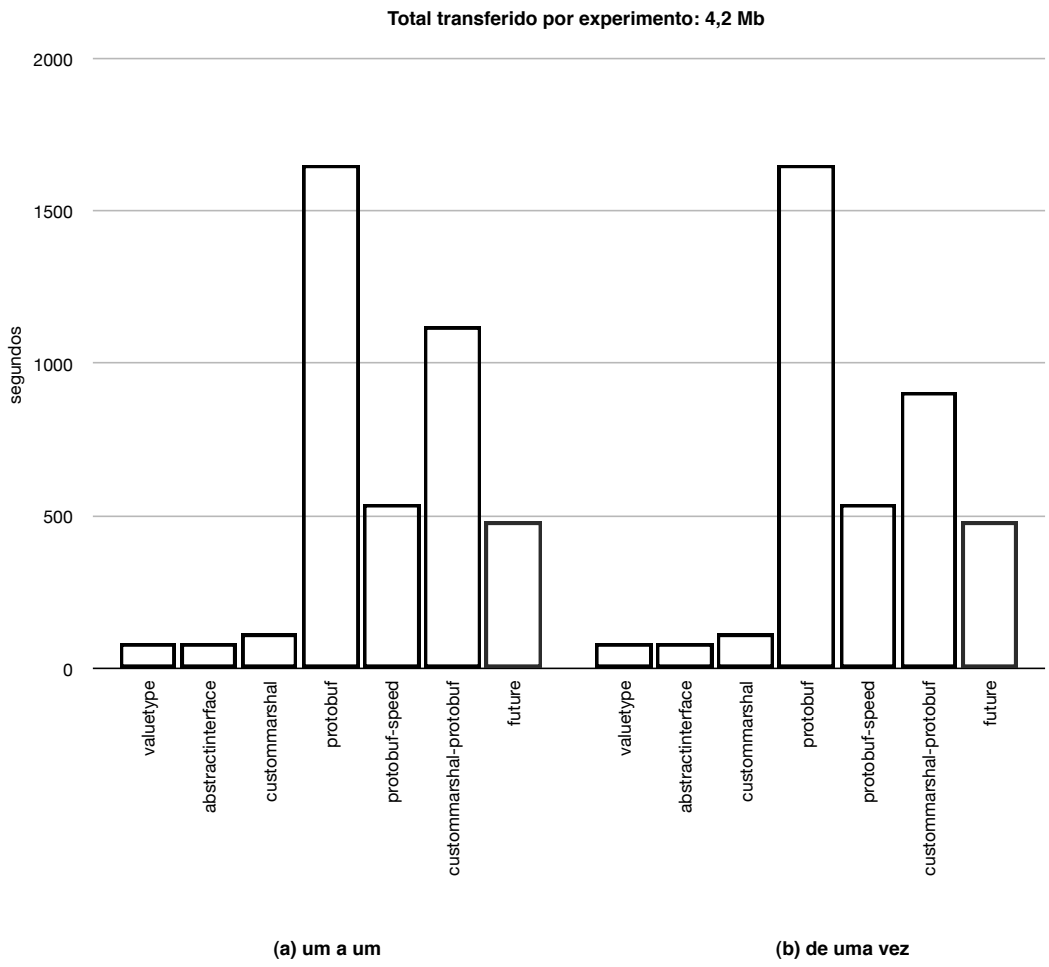

Figura 4.3: Gráfico com os tempos obtidos nos testes de desempenho

para ser um *valuetype* concreto. Ou seja, a visão transferida era formada em toda a sua hierarquia de tipos por *valuetypes*: *ScientificWell*, *Well* e *DataView*. O empacotamento e o desempacotamento das visões foi realizado automaticamente por CORBA e não havia nenhuma intervenção por parte do programador. Os tempos obtidos nesse experimento serão usados como referência e servirão de base nas comparações com os tempos obtidos nos demais experimentos.

Em seguida, foi criado um experimento, chamado de *abstractinterface*, cujo objetivo era verificar se o uso de interfaces abstratas traria algum ônus para a transferência dos dados e, se fosse o caso, avaliar se tal ônus seria aceitável em vista da flexibilidade provida por esse mecanismo. Os tempos obtidos demonstraram que o uso das interfaces abstratas não gera nenhum impacto perceptível na transferência do dado. Sendo assim, o uso das interfaces abstratas foi adotado pela arquitetura como a forma de se representar as visões

de dados, dando maior flexibilidade na implementação dos servidores.

O terceiro experimento, denominado *custommarshal*, tinha como objetivo avaliar o impacto da utilização do mecanismo de *custom marshal*, para permitir que o empacotamento e o desempacotamento de um dado fosse implementado explicitamente. A implementação feita para o dado de poço científico simplesmente lia ou escrevia os atributos nos *streams* disponibilizados sem realizar nenhum tipo de conversão, sendo, portanto, o mais próximo possível do mecanismo implementado por CORBA. A idéia era avaliar se o mecanismo de *custom marshal* era realmente custoso como está salientado na especificação CORBA, ou se esse custo dependia da implementação realizada. Como foi visto através dos tempos obtidos, há, realmente, uma perda de desempenho ao se usar esse mecanismo, algo em torno de 40%.

Como o uso de *custom marshal* se mostrou mais lento do que transferir o *valuetype* diretamente pelos mecanismos de CORBA, ainda havia a alternativa de se utilizar uma representação binária para a transferência que pudesse ser mais adequada do que a utilizada por CORBA. A tecnologia escolhida para essa tarefa foi o *Protocol Buffers* que prometia uma representação dos dados bastante eficiente tanto para o armazenamento quanto para a transferência dos dados. Entretanto, como será detalhado a seguir, os experimentos demonstraram que essa tecnologia, apesar de ser bem superior à tecnologia XML, por exemplo, não possui um desempenho que possa ser comparado ao de CORBA.

Para realizar os experimentos com o *Protocol Buffers* foi necessário criar as mesmas representações da visão científica de dado de poço através de mensagens, utilizando a IDL do *Protocol Buffers*. Os tipos de dados necessários para a execução desses experimentos, estão no apêndice A.4. É importante notar que apenas os tipos realmente necessários para a realização do experimento foram representados na IDL do *Protocol Buffers*.

Antes de tudo, era necessário avaliar o *Protocol Buffers* individualmente, sem a presença da tecnologia CORBA, a fim de atestar que essa tecnologia era realmente eficiente como se anunciava. Foi criado, então, um experimento, chamado de *protobuf*, bastante similar ao uso do serviço de dados, onde um servidor oferecia dados de poço e uma aplicação lia esses dados através de um *socket*. As mensagens, que representavam a visão científica do poço, eram instanciadas, convertidas em vetores de *bytes* e, então, enviadas através de um *socket*. No lado do cliente, o vetor de *bytes* recebido era convertido novamente na mensagem. Esse experimentou mostrou que o tempo gasto no empacotamento e no desempacotamento das mensagens era bastante alto e o desempenho do *Protocol Buffers* ficou muito aquém do desempenho obtido com o uso dos *valuetypes* de CORBA, aproximadamente vinte vezes mais lento.

Um experimento análogo ao *protobuf* foi criado, para utilizar um mecanismo de empacotamento e de desempacotamento de mensagens que seria mais rápido do que o mecanismo padrão oferecido pelo *Protocol Buffers*. Quando se define um arquivo de mensagens no *Protocol Buffers*, é possível informar que as mensagens ali contidas devem ser compiladas em representações cujos empacotamento e desempacotamento levem em consideração a velocidade do procedimento e não o tamanho do vetor de *bytes* gerado. O nome da opção que ativa esse modo de empacotamento e desempacotamento é chamada de *SPEED* e, por causa disso, o nome do experimento é *protobuf-speed*. O uso dessa opção torna os procedimentos de empacotamento e de desempacotamento de mensagens, aproximadamente, 3 vezes mais rápidos, mas, ainda assim, aproximadamente 6 vezes mais lentos que o empacotamento e o desempacotamento de *valuetypes*. Sendo assim, os experimentos seguintes utilizaram a opção *SPEED* para as mensagens geradas.

O experimento *custommarshal-protobuf* foi criado para se avaliar o uso do *Protocol Buffers* junto com o mecanismo de *custom marshal* de CORBA. Nesse experimento, a visão era representada como um *valuetype* concreto que tinha o mecanismo de *custom marshal* ativado. Um dos inconvenientes para a realização desse experimento era a necessidade de criar a mensagem referente à visão científica na hora de se fazer o *marshal*, pois era uma atividade que consumia bastante tempo. Sendo assim, unicamente para esse experimento, a visão científica foi alterada para conter apenas operações, ao invés de sua definição com atributos, como segue abaixo, na listagem 4.9.

Listagem 4.9: Visão científica do poço modificada para a execução do experimento *custommarshal-protobuf*

```
1 valuetype ScientificWell : Well supports sdeacidl::DataView {
2   Header getHeader();
3   Point getLocation();
4   CurveList getCurves();
5 };
```

Com essa alteração, a visão era representada internamente com um atributo que era do tipo da mensagem *Protocol Buffers* correspondente à visão. Ou seja, durante o empacotamento a mensagem que seria transferida já existia e esse procedimento consistia apenas em converter este atributo em um vetor de *bytes* próprio para a transferência. O procedimento de desempacotamento era análogo ao de empacotamento, ou seja, o vetor de *bytes* recebido era convertido em uma mensagem que era um atributo da visão. Mesmo com essa implementação, que tornava desnecessária a criação da mensagem no momento da transferência, esse experimento ainda foi de 10 a 13 vezes mais lento do que o experimento *valuetype*, tido como referencial.

Para tentar amenizar a perda de desempenho com o uso do *custom marshal*, foi criado mais um experimento, chamado de *future*, que fazia o desempacotamento das visões na forma de objetos futuro. Um *objeto futuro* [17] é um *proxy* para um objeto que pode não estar pronto para uso, tipicamente porque há alguma computação ainda não completada que está preparando o objeto. Quando algum dos atributos ou métodos do objeto é utilizado, é feito um *lock* nessa chamada até que o objeto esteja completado.

Durante o desempacotamento da visão, logo após a transferência dos *bytes* contendo o seu estado, uma *thread* era obtida de um *pool* para realizar o desempacotamento desses *bytes* na mensagem correspondente à visão do poço. Ou seja, no procedimento de desempacotamento da visão era solicitado o desempacotamento da mensagem representando o dado recebido. A *thread* de desempacotamento do *valuetype* terminava logo após essa solicitação, dando a impressão de que o *valuetype* já estava pronto no cliente, liberando o servidor para outras atividades como, por exemplo, enviar novos *valuetypes*. A visão, cujo desempacotamento ainda não foi completado, utiliza um mecanismo de *lock* para travar uma chamada que precise utilizar os seus atributos, e esse *lock* é liberado apenas quando todo o processo de desempacotamento é finalizado.

Os experimentos realizados demonstraram que a implementação padrão de empacotamento e desempacotamento dos *valuetypes* de CORBA é bastante eficiente em comparação com outras tecnologias de transferência de dados estruturados. Por exemplo, o desempenho de CORBA foi muito superior ao *Protocol Buffers*, uma tecnologia nova e bastante promissora. O *Protocol Buffers* que é de 20 a 100 vezes mais rápido do que XML, chega a ser de 6 a 20 vezes mais lento do que os *valuetypes* de CORBA. Entretanto, a taxa de transferência obtida é muito baixa em relação ao *throughput* da rede.

As interfaces abstratas de CORBA também demonstraram muita eficiência, pois o seu uso em conjunto com os *valuetypes* foi imperceptível, em termos de desempenho. Esse mecanismo deve ser utilizado sempre que for necessário, dada a flexibilidade proporcionada. Ao contrário das interfaces abstratas, o mecanismo de *custom marshal*, conforme salientado pela especificação CORBA, deve ser evitado sempre que possível. A simples declaração de um *valuetype* como customizável, mesmo com uma implementação bastante simples, foi suficiente para reduzir o desempenho da transferência do dado.