3
O Arcabouço de Serviços de Acesso a Dados Científicos

O objetivo principal da arquitetura apresentada neste trabalho, como já foi dito anteriormente, é permitir que aplicações científicas ofereçam seus dados através de uma representação estruturada e de maneira eficiente, de modo que essas aplicações sejam integradas através do compartilhamento de seus dados. Tipicamente, um dado científico é representado de maneira complexa devido, principalmente, aos algoritmos que o processa, aumentando significativamente a dificuldade de se implementar esse compartilhamento. Portanto, é fundamental que as aplicações possam acessar visões do dado que sejam as mais adequadas ao seu domínio e ao processamento que pretendem executar, e que essas visões sejam transferidas eficientemente entre as aplicações.

A arquitetura foi concebida através de uma abordagem baseada em componentes de *software* [14] de acordo com o modelo de componentes SCS (Sistema de Componentes de *Software*) [15]. Esse modelo define que um *componente* de *software* deve prover as suas funcionalidades através de serviços. Um serviço é definido através de uma interface e esta pode servir de base para uma ou mais implementações. Uma interface de serviço em conjunto com uma de suas implementações formam uma *faceta*.

Baseado no modelo SCS, o *servidor de dados* proposto nesta arquitetura é um componente que provê suas funcionalidades através de facetas de *serviços de dados*. Um serviço de dados tem a responsabilidade de gerenciar os dados oferecidos por seu servidor, permitindo que as aplicações possam criar, excluir e obter dados.

A obtenção de um dado é iniciada através de uma navegação oferecida pelo serviço de dados. Os descritores dos dados, ou *metadados*, são dispostos em uma hierarquia cujo nível inicial é formado por várias raízes. A *navegação hierárquica* consiste, portanto, na obtenção dos metadados que representam as raízes dessa hierarquia e, na sequência, percorrendo os metadados descendentes até que se chegue ao dado desejado. A partir do metadado que descreve o dado do seu interesse, a aplicação pode, então, solicitar a transferência de uma visão desse dado.

Uma *visão* é uma representação do dado composta por um subconjunto

de seus atributos e operações. A transferência da visão é o que representa de fato a obtenção do dado por parte das aplicações.

A navegação hierárquica e o uso de múltiplas visões são os mecanismos pelos quais a arquitetura oferece a *obtenção parcial* do dado. Representando o dado como uma hierarquia, as aplicações transferem apenas a parte de seu interesse. Da mesma forma, uma visão pode oferecer apenas uma parte do dado, parte essa que é a de real interesse para uma determinada aplicação ou domínio de aplicação.

Na seção seguinte, é apresentada a visão geral da arquitetura, descrevendo as funcionalidades de cada um dos elementos que a compõem e as relações entre eles. Em seguida, na seção 3.2, a interface de cada um desses elementos é apresentada em detalhes.

3.1 Visão Geral da Arquitetura

O modelo de componentes SCS, utilizado como base para a definição dos componentes da arquitetura proposta, define que um componente oferece as suas funcionalidades através de facetas. Esse modelo permite que um componente ofereça a mesma interface por meio de mais de uma faceta, e essas facetas seriam implementadas, possivelmente, de maneiras diferentes. Nesse contexto, o *servidor de dados* definido aqui nessa arquitetura pode oferecer a mesma interface de *serviço de dados* através de mais de uma faceta. Em outras palavras, o servidor de dados pode oferecer mais de um serviço de dados, permitindo que os dados por ele oferecidos sejam separados ou categorizados através de mais de um serviço. Por exemplo, se um servidor oferece dados de diferentes domínios, é natural que haja um serviço de dados para cada um desses domínios. O servidor pode, ainda, ter a necessidade de separar os dados de modo que diferentes visões de um dado sejam oferecidas por diferentes serviços.

Os dados oferecidos por um servidor possuem uma *chave* que os identifica univocamente. Essa chave é composta pelo identificador do servidor de dados e por um identificador do dado dentro desse servidor. Cada servidor é responsável por garantir que o identificador do dado seja único. Como os identificadores dos servidores também devem ser únicos, esse conjunto que compõe a chave é, garantidamente, único mesmo que vários servidores de dados sejam utilizados. A chave do dado está presente em seus metadados e nas suas visões.

O identificador do servidor pode ser utilizado também para se obter uma referência para o mesmo. Com essas informações, uma aplicação qualquer que esteja de posse da chave pode contactar o servidor de origem do dado para

obter mais informações sobre esse dado. É possível, por exemplo, persistir a chave em algum meio de armazenamento qualquer e depois verificar se o dado sofreu algum tipo de atualização. A chave unívoca pode ser utilizada também quando uma aplicação repassa o dado obtido para uma terceira aplicação. Nesse caso, essa terceira aplicação não sabe de onde veio o dado e pode não possuir ainda uma referência para seu servidor de origem.

O servidor de dados é o ponto de entrada da arquitetura. As aplicações devem, de alguma forma, obter uma referência para o componente do servidor de dados e, a partir dele, obter a faceta correspondente ao serviço de dados do seu interesse. A arquitetura aqui apresentada não impõe nenhum mecanismo para que essa comunicação inicial entre uma aplicação e um servidor de dados seja iniciada.

A partir do *serviço de dados* é que a aplicação pode, de fato, iniciar a navegação pelos dados oferecidos por um servidor. A *navegação hierárquica* é iniciada com a obtenção dos metadados que representam as raízes da hierarquia. Em seguida, a aplicação pode descer pela hierarquia solicitando ao serviço de dados os metadados descendentes de um dado qualquer. É possível ainda obter um metadado qualquer oferecido pelo serviço, bastando para isso informar a sua chave.

Os *metadados* utilizados na arquitetura são descrições dos dados que eles representam. Os metadados trazem consigo os nomes de todas as interfaces das visões oferecidas para o dado, permitindo que a aplicação descubra se há visões do dado compatíveis com as visões com as quais deseja trabalhar. Mais uma vez, o objetivo é tornar desnecessária a solicitação, ou a própria transferência, do dado inutilmente. Os metadados são utilizados na arquitetura para permitir que as aplicações façam a navegação pela hierarquia sem a necessidade de transferir dados que não sejam de seu interesse. Esse tipo de navegação torna a arquitetura bastante eficiente, pois elimina a necessidade de se transferir dados que não serão efetivamente utilizados pelas aplicações.

Uma vez que uma aplicação encontrou o dado do seu interesse e que possui a lista de visões oferecidas pelo dado, essa aplicação pode solicitar ao serviço de dados a transferência de uma visão do dado. Uma *visão* é uma representação do dado formada por um subconjunto dos atributos e operações oferecidos pelo dado. Cada visão é, portanto, uma representação diferente para um mesmo dado e essas representações podem inclusive contemplar domínios de aplicação diferentes. Naturalmente, quando apenas os atributos necessários são utilizados, há um aumento na eficiência da transferência do dado.

O mecanismo de múltiplas visões para se representar um dado permite ainda que se implemente o *versionamento* dessas visões. Quando for necessária

a alteração de uma visão, com a inclusão, exclusão ou alteração de atributos ou operações, e essa alteração puder trazer impactos negativos nas aplicações, é possível criar uma nova visão, baseada na que seria alterada, já com todas as mudanças necessárias. Com isso, as novas aplicações que venham a usar o servidor de dados utilizarão a nova versão, enquanto que as aplicações antigas podem continuar usando a versão original da visão, até que chegue o momento de serem atualizadas.

A materialização de uma visão é a tarefa de carregar um dado a partir de alguma origem e, a partir desse dado, gerar uma representação da visão. Essa tarefa pode ser realizada tanto pelo servidor de dados quanto pelos seus serviços. Quando o dado for utilizado apenas por um serviço, é natural que este seja o responsável pela materialização. Caso o dado seja compartilhado por mais de um serviço, é preferível que essa tarefa fique a cargo do servidor para evitar duplicações desnecessárias. Cada implementação deve decidir quem será o responsável pela materialização e em que momento esta será realizada. Uma visão pode ser materializada sempre que for solicitada, no início do funcionamento do servidor ou através de um *cache* que implemente algum tipo de política para decidir quando uma visão deve ser descartada.

O serviço de dados oferece também a possibilidade de se criar ou excluir dados. A criação do dado pode ser feita tanto a partir de um conjunto de metadados quanto usando um outro dado qualquer como origem, ocasionando a cópia desse dado. O dado de origem pode ser oriundo de um serviço de dados diferente e, até mesmo, de um servidor de dados diferente. Tanto a criação quanto a exclusão de um dado são funcionalidades opcionais que podem ou não ser implementadas por um servidor de dados.

A figura 3.1 apresenta a arquitetura do servidor de dados estruturados. O servidor acessa os dados armazenados em alguma base e os apresenta em uma estrutura hierárquica de múltiplas raízes. Para cada um dos dados providos pelo servidor há um conjunto de metadados e um conjunto de 1 a N visões. Os clientes navegam pela estrutura hierárquica de metadados e, quando chegam ao dado de seu interesse, obtém uma ou mais visões com as quais desejam trabalhar.

3.2 Detalhamento das Interfaces

A arquitetura de servidores de dados tem como um forte requisito a independência de plataforma e de linguagem de programação. Essa característica é fundamental para que a arquitetura possa ser implementada pelo maior número possível de aplicações científicas. Para atender a esse requisito, os tipos que

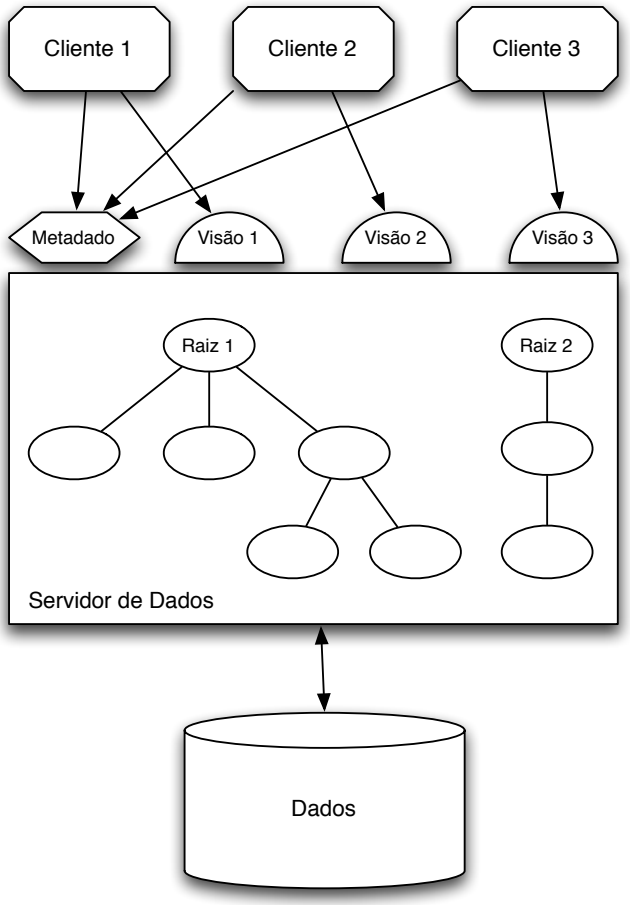

Figura 3.1: A arquitetura do servidor de dados estruturados

compõem a arquitetura foram todos definidos através da *linguagem de definição de interfaces (IDL)* de CORBA[1].

Como já foi dito, os servidores de dados utilizam o modelo de componentes distribuídos *SCS (Sistema de Componentes de Software)*. O SCS possui o mesmo requisito de independência de plataforma e de linguagem de programação que a arquitetura de servidores de dados e, por esse motivo, os tipos que compõem esse modelo também foram definidos através da IDL de CORBA[2].

Um componente SCS oferece as suas funcionalidades através de facetas, que são definidas através de interfaces CORBA e que, portanto, são implementadas por objetos remotos. Uma faceta é definida pelo nome de sua interface CORBA, por um nome simbólico e por um objeto CORBA que implementa a interface. Uma vez definida uma faceta, a mesma pode ser identificada através da sua interface ou do seu nome simbólico.

[1] A IDL completa dos servidores de dados encontra-se no apêndice A.1
[2] A IDL completa do SCS encontra-se no apêndice A.2

Um componente SCS deve prover obrigatoriamente uma faceta que implemente a interface *IComponent*. A interface *IComponent* define um tipo "componente". Através dessa faceta é possível, dentre outras coisas, obter todas as outras facetas oferecidas pelo componente. Ou seja, é através de uma faceta *IComponent* que as aplicações obtêm as facetas dos serviços de dados. A figura 3.2 apresenta um componente SCS modelado em um diagrama de componentes seguindo a notação UML 2.x. Neste diagrama há um componente SCS padrão que oferece a faceta obrigatória *IComponent*.

Listagem 3.1: A faceta de componente

```
1 interface IComponent {
2   void startup() raises (StartupFailed);
3   void shutdown() raises (ShutdownFailed);
4
5   ComponentId getComponentId ();
6
7   Object getFacet (in string facet_interface);
8   Object getFacetByName (in string facet);
9 };
```

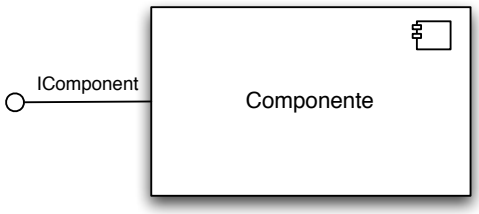

Figura 3.2: Um componente SCS

A implementação de um *servidor de dados* consiste na criação de um componente, que representa esse servidor, e da criação de suas facetas. É tarefa do servidor exportar as suas facetas como objetos remotos e, com isso, colocá-las prontas para as solicitações provenientes das aplicações. A figura 3.3 apresenta o servidor de dados, que é um componente SCS que oferece, além da faceta *IComponent*, uma ou mais facetas de serviço de dados.

O componente do servidor e as suas facetas são executados em *um único processo* e, por causa disso, as facetas podem se comunicar umas com as outras diretamente através de referências locais. Como o componente e suas facetas compartilham o mesmo espaço de memória, o servidor pode ser utilizado para armazenar informações utilizadas por mais de uma faceta, evitando duplicação. O servidor pode, por exemplo, armazenar os dados oferecidos pelos seus serviços em um *cache* único.

As facetas de *serviços de dados* são definidas pela interface *IDataService*. Essa interface define as operações necessárias para a realização da navegação

Figura 3.3: O componente do Servidor de Dados

hierárquica pelos descritores dos dados oferecidos. Uma aplicação que queira navegar pelos dados de um servidor deve, como dito antes, obter as raízes da hierarquia. A operação responsável por servir os metadados das raízes da hierarquia é a *getRoots*. Em seguida, a aplicação obtém os descendentes desses metadados obtidos usando, para isso, a operação *getChildren*. Além dessas duas, há uma outra operação para a obtenção de metadados, chamada *getDataDescription*, que é responsável por servir um metadado qualquer a partir da chave de um dado.

Listagem 3.2: A faceta de serviço de dados

```
1  interface IDataService {
2    DataDescriptionList getRoots();
3    DataDescriptionList getChildren(in DataKey fKey);
4
5    DataDescription getDataDescription(in DataKey fKey);
6    DataView getDataView(in DataKey fKey, in string fViewInterface);
7    DataViewList getDataViewList(in DataKeyList fKeyList,
8        in string fViewInterface);
9    sdeacidl::types::StringSeq getViewInterfaces(in DataKey fKey);
10
11   DataKey createData(in DataKey fParentKey, in MetadataList fMetadata)
12     raises (OperationNotSupported);
13   DataKey createDataFrom(in DataKey fParentKey, in DataKey fSourceKey)
14     raises (OperationNotSupported, UnknownType);
15   boolean deleteData(in DataKey fKey) raises (OperationNotSupported);
16 };
```

Os *metadados* oferecidos pelos serviços de dados são objetos do tipo *DataDescription*. Este tipo é definido como um *valuetype* concreto para poder ser instanciado diretamente e para permitir a criação de tipos estendidos através de herança. O *DataDescription* possui a chave unívoca do dado e uma sequência com os nomes das visões oferecidas para o dado descrito; estas informações estão armazenadas nos atributos *key* e *views*, respectivamente.

A inclusão de novos metadados em um *DataDescription* pode ser feita através de herança, como citado no parágrafo anterior, ou da utilização de um atributo chamado *metadata*. Este atributo é representado por um sequência de pares chave-valor; tanto a chave quanto o valor são do tipo *string*. A decisão de se estender o tipo *DataDescription* ou de se utilizar o atributo *metadata* vai

depender da natureza do metadado a ser representado. Caso não seja possível acomodar o metadado em uma *string* ou no caso de ser necessária uma tipagem mais forte deste metadado, a opção a ser utilizada será a herança. Em caso contrário, o atributo *metadata* deve ser utilizado, evitando que se crie extensões ao *DataDescription*.

Listagem 3.3: Os metadados de um dado

```
1  struct Metadata {
2    string fName;
3    string fValue;
4  };
5  typedef sequence<Metadata> MetadataList;
6
7  valuetype DataDescription {
8    public DataKey fKey;
9    public sdeacidl::types::StringSeq fViews;
10   public MetadataList fMetadata;
11 };
12 typedef sequence<DataDescription> DataDescriptionList;
```

A *chave* unívoca do dado é representada pelo tipo *DataKey*. O tipo *DataKey* define dois atributos: *service_id* e *actual_data_id*. O primeiro atributo traz o identificador do tipo do componente implementado pelo servidor de origem do dado. Já o segundo, representa um identificador unívoco do dado dentro do seu servidor de dados. A partir do momento em que uma aplicação encontra o dado do seu interessa é possível solicitar a transferência de uma *visão* desse dado. As visões suportadas por um dado encontram-se descritas em seu metadado (atributo *views*). Além disso, uma aplicação pode verificar as visões suportadas por um dado através da operação *getViewInterfaces* do serviço de dados.

Listagem 3.4: A chave unívoca de um dado

```
1  struct DataKey {
2    scs::core::ComponentId fServerID;
3    URI fActualDataID;
4  };
5  typedef sequence<DataKey> DataKeyList;
```

Uma visão é definida na arquitetura como um tipo que estenda a *interface abstrata DataView*. Por se tratar de uma interface abstrata, um *DataView* pode ser implementado tanto por um objeto remoto (interface) quando por um objeto por valor (*valuetype*), ficando a cargo dos serviços de dados decidir qual a melhor forma de representar os seus dados. Caso o dado seja definido como um *valuetype* concreto, o serviço de dados poderá redefinir o empacotamento e o desempacotamento desse dado. Essa tarefa, chamada de *custom marshal*, consiste na implementação de um algoritmo para a conversão dos atributos de dados em uma representação própria para transferência e vice-versa. A

utilização ou não do mecanismo de *custom marshal* é decidida no momento da definição do *valuetype*, indicando se este é do tipo *custom*. A implementação do tipo do dado é quem contém as operações responsáveis pelo seu empacotamento e desempacotamento. Essa implementação estará presente portanto no servidor e em cada aplicação que tenha a necessidade de transferir esse tipo de visão.

Listagem 3.5: A visão de um dado

```
1 abstract interface DataView {
2   DataKey getKey();
3   service::IDataService getDataService();
4   string getViewInterface();
5 };
6 typedef sequence<DataView> DataViewList;
```

O mecanismo de interfaces abstratas, entretanto, não é de todo transparente. As aplicações devem saber de que forma o dado foi representado para poder fazer a conversão do tipo *DataView* recebido para o tipo real da visão, que é necessariamente um tipo estendido dessa interface. No caso de objetos remotos a aplicação poderá fazer um *narrow* para o tipo apropriado; já no caso de objetos por valor, será necessário um *cast*.

As visões possuem a chave unívoca do dado e a operação *getKey* é a responsável por informar essa chave. Um *DataView* oferece também uma operação, *getViewInterface*, que é usada para se obter o nome da interface implementada pela visão. Através dessa informação é que se deve fazer o *narrow* ou o *cast* descritos anteriormente. Por fim, há ainda a operação *getDataService* que dá à aplicação uma referência ao serviço de dados de onde a visão se originou.

De posse do nome da interface de uma visão e da chave do dado, é possível solicitar a transferência dessa visão ao serviço de dados, através da operação *getDataView*. Caso uma aplicação queira transferir a mesma visão de mais de um dado ao mesmo tempo deve utilizar a *gatDataViewList*. Nessa operação ao invés de apenas uma chave, a aplicação deve informar uma sequência de chaves de todos os dados de seu interesse.

A criação de um novo dado é solicitada pelas aplicações através de operações disponibilizadas na interface do serviço de dados. Quando se deseja criar um dado em um serviço, a partir apenas de seus metadados, utiliza-se a operação *createData*. Essa operação recebe a chave do dado ascendente do que está sendo criado e um conjunto de propriedades que farão parte de seus metadados. É possível também criar um dado a partir de um outro dado já existente, efetivamente realizando uma cópia. A operação *createDataFrom*, responsável por essa cópia, também recebe a chave do dado ascendente, mas, ao invés de receber uma lista de propriedades, recebe a chave do dado que

será usado como origem para a cópia. Se o dado de origem não oferecer visões suportadas pelo serviço onde a cópia está sendo gerada, o serviço pode lançar uma exceção do tipo *UnknownType*.

Como essa arquitetura se propõe a atender servidores de dados de uma forma geral, alguns servidores serão apenas servidores de consulta e não serão capazes de criar e excluir dados. É importante haver uma forma oficial de informar que uma determinada operação não está disponível para certos servidores. As operações de criação e exclusão de um dado são opcionais, ou seja, nem todo serviços de dados oferecerá estas funcionalidades. Caso um serviço não permita a criação ou exclusão do dado, poderá lançar uma exceção do tipo *OperationNotSupported*, indicando para as aplicações que essas operações não estão disponíveis.