MAC 440/5759 - Sistemas de Objetos Distribuídos - Primeiro Semestre de 2006

EPs 1 e 2: Entidades e Serviços em CORBA

1 Serviços, sessões, processos e entidades

Quatro grandes categorias de objetos remotamente acessíveis têm sido usadas em ambientes de objetos distribuídos:

- *Serviço* — objeto sem estado cujos métodos prestam algum serviço. Exemplo: cálculo do valor da prestação mensal dada a taxa de juros, o total devido e o número de prestações.
- *Sessão* — objeto que tem estado não persistente e representa a interação de um certo processo cliente com a aplicação servidora. Exemplo: carrinho de compras.
- *Processo* — objeto que representa um "processo de negócios" (*business process*) com estado persistente e não vinculado a um determinado processo cliente. Exemplo: processo para obtenção de um financiamento.
- *Entidade* — objeto representativo de uma entidade (no sentido do modelo E/R) persistente que, do ponto de vista dos usuários, possui um identificador (como número da conta corrente, código de produto, número USP, etc.) e cujas informações tipicamente residem num banco de dados.

O que distingue um processo de uma entidade é a ausência de um identificador externo que seja relevante para os usuários. Como a avaliação dessa relevância não é objetiva, nem sempre é clara a distinção entre processo e entidade. Já as diferenças entre sessão e processo são bem nítidas: (1) o estado da sessão não é persistente, o do processo é; (2) a sessão é vinculada a um processo cliente, o processo não é.

Como a importância dessas categorias de objetos é muito grande, elas aparecem nos dois modelos de componentes para objetos distribuídos usados em ambientes não Microsoft: o *Enterprise JavaBeans* (EJB) e o *CORBA Component Model* (CCM). O EJB 2.1 não tem a categoria processo, mas tem as categorias serviço (*stateless session beans*, na terminologia EJB), sessão (*stateful session beans*) e entidade (*entity beans*). O CCM tem quatro categorias de componentes que correspondem exatamente às vistas acima.

2 A controvérsia sobre entidades

Neste trabalho você implementará e usará objetos CORBA das categorias serviço e entidade. Um dos objetivos é entender (e possivelmente formar uma opinião sobre) uma discussão bem atual sobre a categoria entidade. Será que entidades devem ser objetos remotamente acessíveis? Como o acesso remoto a entidades geralmente traz problemas sérios de desempenho, muitos defendem a eliminação da categoria entidade[1].

A discussão sobre acesso remoto a entidades se refletiu na evolução da arquitetura EJB. Na primeira versão da especificação que define essa arquitetura, todos os *entity beans* eram remotamente acessíveis. A versão 2.0 introduziu o conceito de *entity bean* com interfaces locais, que não recebem chamadas de outros processos. A próxima versão da especificação EJB (a versão 3.0) está prestes a ser publicada. Ela contém mudanças tão significativas que, na prática, definem um modelo de componentes completamente novo (o chamado EJB3), no qual não existem *entity beans* remotamente acessíveis. O novo modelo de componentes não elimina seu antecessor, que continua existindo para que se mantenha a compatibilidade com aplicações escritas para as versões anteriores de EJB, mas se soma a ele.

3 As interfaces de uma entidade

Tanto no EJB (até a versão 2.1) como no CCM, os componentes tipo entidade representam uma coleção homogênea de objetos persistentes. Tal coleção pode ser um conjunto de contas correntes, ou de produtos,

[1] Aqui estamos falando de categorias de objetos *remotamente acessíveis*. É claro que entidades continuarão existindo dentro de servidores, mas não como objetos cujos métodos podem ser invocados por outros processos.

funcionários, clientes, etc. Nesses dois modelos, um componente tipo entidade tem duas interfaces: a interface das instâncias e a interface *home*.

A interface das instâncias é a interface comum a todos os elementos da coleção de objetos. Exemplo: interface `ContaCorrente`, no caso de uma coleção de contas correntes.

A interface *home* tem um papel duplo: de fábrica (*factory*) que sabe criar novos elementos para a coleção de objetos (exemplo: criação de uma nova `ContaCorrente`) e de buscador (*finder*) capaz de encontrar elementos com determinadas propriedades (exemplo: buscar `ContaCorrente` pelo nome do titular). Tanto o EJB como o CCM usam o termo *home* para a interface usada para fabricação e busca de elementos da coleção (instâncias do componente). Pode-se considerar que a interface *home* é a interface de uma coleção de objetos e que a interface das instâncias é a interface de cada elemento da coleção.

A prática usual em EJB e CCM é dar à interface *home* o nome da interface das instâncias, com o sufixo `Home`. Este é um exemplo bem simples, em IDL, de interface de instância (`Customer`) e sua correspondente interface *home* (`CustomerHome`):

```
interface Customer {
    long getId();
    string getName();
    string getPhone();
    void setPhone(in string phone);
};

interface CustomerHome {
    Customer create(in long id, in string name, in string phone) raises(CreateException);
    void remove(in long id) raises(RemoveException);
    Customer findById(in long id) raises(NotFoundException);
    CustomerList findByName(in string name);
    CustomerList findByPhone(in string phone);
};
```

Embora as interfaces acima não sejam de componentes EJB (que têm suas interfaces definidas em Java) nem de componentes CCM (pois elas não satisfazem certas condições que não vêm ao caso aqui), elas ilustram o conceito de entidade nesses modelos de componentes. Ambos os modelos dão suporte à geração automática de código que faz acesso a um banco de dados para leitura ou atualização do estado persistente das entidades. Além disso, ambientes EJB e CCM cuidam automáticamente de certos "aspectos de sistema" como segurança, transações e controle de concorrência, além de manter e gerenciar *pools* de *threads* e de conexões com servidores de bancos de dados. Lidar com esses aspectos dificulta bastante a construção de aplicações servidoras a partir do zero. O EJB e o CCM resolvem esse problema oferecendo um "servidor genérico e extensível" que cuida dos aspectos de sistema e permite que nele sejam implantados componentes voltados especificamente para as aplicações. Esse tipo de servidor EJB ou CCM é denominado servidor de aplicações.

4 O padrão serviço/sessão de fachada

O padrão "serviço (ou sessão) de fachada" é bastante usado, em ambientes EJB, para evitar que clientes façam chamadas remotas a componentes tipo entidade. Nesse padrão de projeto, todos os acessos remotos aos dados das entidades devem ser mediados por componentes tipo serviço ou sessão residentes no mesmo servidor. Os componentes tipo serviço ou sessão recebem as chamadas remotas dos clientes e fazem chamadas locais a componentes tipo entidade. Tal arranjo tem o objetivo de minimizar o número de chamadas remotas e o tráfego de dados entre os clientes e o servidor.

No caso de componentes EJB, o padrão serviço/sessão de fachada é conhecido simplesmente como *session facade*, pois na terminologia EJB o nome *session* é usado tanto para componentes tipo serviço (*stateless session beans*) como para componentes tipo sessão (*stateful session beans*). Para mais informações sobre esse padrão de projeto no contexto de EJB, veja o livro *EJB Design Patterns*, de Floyd Marinescu (Wiley, 2002).

O arquivo `VideoRental.idl`, incluído ao final deste texto, mostra o uso de um serviço de fachada numa aplicação servidora voltada para vídeo-locadoras. Há três tipos de entidades no servidor da vídeo-locadora:

- filme (interfaces `Movie` e `MovieHome`),
- exemplar de filme (interfaces `MovieCopy` e `MovieCopyHome`) e
- usuário (interfaces `Customer` e `CustomerHome`).

O uso de tipos de entidades distintos para `Movie` e `MovieCopy` capturam a diferença entre um "filme abstrato" (o conteúdo de um DVD ou de uma fita VHS) e um exemplar (cópia física) de filme, em DVD ou VHS. O que você aluga e leva para casa é o exemplar, não o filme.

Veja, no arquivo `VideoRental.idl`, que há um relacionamento 1 para N entre `Movie` e `MovieCopy` (podem haver muitos exemplares de um filme). No lado `Movie`, esse relacionamento se manifesta através das operação `getCopies`. No lado do `MovieCopy` ele se manifesta através da operação `getMovie`.

Há também um relacionamento 1 para N entre `Customer` e `MovieCopy` (um usuário pode alugar muitos CDs ou fitas VHS). Note que a data de devolução de um ítem alugado é, conceitualmente, um atributo desse relacionamento. No lado do `Customer`, o relacionamento se manifesta através da operação `getTakenMovieCopies` e no lado do `MovieCopy` através das operações `isRented`, `rentTo`, `unrent`, `getTaker` e `getReturnDate`.

Um cliente remoto poderia desempenhar todas as suas tarefas (consultar as informações sobre um filme, alugar um exemplar de filme para um usuário, etc.) fazendo chamadas às operações dos objetos CORBA que representam as entidades acima descritas. Alternativamente, pode-se fazer com que o servidor disponibilize um "serviço de fachada" e escrever um cliente remoto que faça acesso apenas ao objeto CORBA que representa a fachada. O arquivo `VideoRental.idl` define também a interface `RentalService`, que provê o serviço de fachada. Note que esse serviço usa a estrutura auxiliar `MovieInfo` para passar ao cliente remoto, de uma só vez, todas as informações sobre um filme. O uso de estruturas auxiliares, que carregam um monte de informações de uma vez, tem o objetivo de reduzir o número de interações entre servidor e seus clientes remotos. Esse é outro padrão de projeto, conhecido como *data transfer object* (DTO) e bastante usado em sistemas distribuídos. O livro *EJB Design Patterns* (citado anteriormente) contém mais informações sobre esse padrão no contexto de EJB.

5 Tarefas

Seu trabalho consiste em:

- escrever um servidor CORBA que implemente as interfaces definidas no arquivo VideoRental.idl;
- escrever um cliente que chame as operações do objeto de fachada `RentalService`;
- escrever outro cliente que, em vez de chamar operações do objeto de fachada, faça seqüências de chamadas remotas às entidades do servidor, de modo a obter, com cada seqüência de chamadas às entidades, as mesmas informações ou o mesmo efeito que obteria se chamasse uma das operações da fachada;
- comparar os tempos de execução das chamadas remotas ao objeto de fachada com os tempos correspondentes para as seqüências de chamadas às entidades. Esses tempos devem ser medidos no lado do cliente.

Bônus: Meça os tempos de execução com o cliente e o servidor rodando na mesma máquina e com o cliente e o servidor em máquinas diferentes, porém na mesma rede local. Se tiver chance, meça também os tempos com o cliente na sua casa e o servidor aqui no IME.

6 Etapas

O trabalho será dividido em duas etapas, que corresponderão ao EP1 e ao EP2 de MAC 440/5759:

- Na primeira etapa não será implementada a persistência das entidades no servidor. Você pode assumir que a quantidade de instâncias das entidades é sufcientemente pequena para que todas elas fiquem ativas, em memória, ao mesmo tempo. Quando o servidor encerrar sua execução, todas as informações sobre entidades serão perdidas.
- Na segunda etapa, as entidades terão seu estado armazenado num banco de dados. Para isso, deverá ser usado o servidor de bancos de dados MySQL disponível na rede do IME. Assuma que a quantidade de instâncias das entidades pode ser grande a ponto de inviabilizar a ativação simultânea de todas as instâncias. Resolva esse problema de uma das seguintes maneiras:
  - fazendo instanciação dinâmica de serventes para as instâncias das entidades (através de um `ServantLocator` para cada tipo de entidade) e implementando o *evictor pattern*, ou
  - usando um `DefaultServant` para todas as instâncias de entidades de um mesmo tipo.

Ao final de cada etapa você deverá entregar também um relatório contendo os resultados das medições de tempo efetuadas e uma avaliação comparativa entre os dois esquemas testados (acesso remoto à entidades e acesso remoto apenas ao serviço de fachada).

7 Requisitos

- Este trabalho deve ser feito em equipes de até duas pessoas.
- Cada equipe deve se registrar comigo (Reverbel) até o dia 13 de abril, através de uma mensagem informando os nomes dos integrantes da equipe. A mensagem deve ser enviada para meu email e deve ter *subject* "registro de equipe de SOD".
- Linguagens de programação aceitas: Java e C++.
- ORBs que você deve utilizar: JacORB (ORB para Java) e MICO (ORB para C++).
- A implementação não deve ser toda em Java ou toda em C++. Pelo menos uma das partes (cliente ou servidor) de uma das duas etapas deve ser escrita numa linguagem diferente das demais. Minha recomendação é usar C++ no lado do cliente, numa das duas etapas, e escrever todo o resto em Java.
- Você deverá entregar um arquivo tar.gz contendo os seguintes ítens:
  - um arquivo tipo txt, pdf ou ps com o seu relatório;
  - os arquivos-fonte em Java e C++;
  - um arquivo `build.xml`, para geração automatizada dos programas Java pelo utilitário `ant`;
  - um `makefile`, para geração automatizada dos programas C++ pelo utilitário `make`;
  - um arquivo `README` descrevendo o brevemente o conteúdo e a estrutura de subdiretórios do arquivo tar.gz entregue e contendo quaisquer informações adicionais que sejam importantes para gerar e executar o servidor e o(s) cliente(s).

  O desempacotamento do seu arquivo tar.gz deverá produzir um <u>diretório</u> contendo esses ítens. O diretório deve ter nome da forma ep2-membros-da-equipe (exemplo: ep2-joao-maria).

Bom trabalho!

Apêndice: arquivo VideoRental.idl

```
module VideoRentalStore {

    enum MediaType {VHS, DVD};
    struct Date { short day; short month; short year; };

    // Forward interface declarations

    interface Movie;
    interface MovieCopy;
    interface Customer;

    // Auxiliary sequence definitions

    typedef sequence<string> StringList;
    typedef sequence<Movie> MovieList;
    typedef sequence<MovieCopy> MovieCopyList;
    typedef sequence<Customer> CustomerList;

    // Exception definitions

    exception CreateException { string detail; };
    exception RemoveException { string detail; };
    exception NotFoundException { };
    exception AlreadyRentedException { };
    exception NotRentedException { };

    // Definitions of entity interfaces (Movie, MovieCopy and Customer)
    // and their corresponding home (factory and finder) interfaces

    interface Movie {
        long getId();
        string getName();          // movie name
        string getDirector();      // name of director
        StringList getGenre();     // list of genre names (comedy, action, etc)
        StringList getCast();      // list of actor/actress names
        short getYear();           // production year
        short getDuration();       // movie duration in minutes
        MovieCopyList getCopies(); // the copies of this movie
    };

    interface MovieHome {
        Movie create(in long id,
                     in string name,
                     in string director,
                     in StringList genre,
                     in StringList cast,
                     in short year,
                     in short duration)
            raises(CreateException);
        void remove(in long id)
            raises(RemoveException);
        Movie findById(in long id)
            raises(NotFoundException);
        MovieList findByName(in string name);
        MovieList findByDirector(in string director);
        MovieList findByGenreAndYear(in string genre, in short year);
    };
```

```
interface MovieCopy {
    long getId();
    Movie getMovie();
    MediaType getMediaType();
    boolean isRented();
    void rentTo(in Customer customer,
                in Date returnDate)
        raises(AlreadyRentedException);
    void unrent()
        raises(NotRentedException);
    Customer getTaker()
        raises(NotRentedException);
    Date getReturnDate()
        raises(NotRentedException);
};

interface MovieCopyHome {
    MovieCopy create(in long id,
                     in Movie movie,
                     in MediaType mediaType)
        raises(CreateException);
    void remove(in long id)
        raises(RemoveException);
    MovieCopy findById(in long id)
        raises(NotFoundException);
    MovieCopyList findByMovieIdAndMediaType(in long movieId,
                                            in MediaType mediaType);
};

interface Customer {
    long getId();
    string getName();
    string getPhone();
    void setPhone(in string phone);
    MovieCopyList getTakenMovieCopies();
};

interface CustomerHome {
    Customer create(in long id,
                    in string name,
                    in string phone)
        raises(CreateException);
    void remove(in long id)
        raises(RemoveException);
    Customer findById(in long id)
        raises(NotFoundException);
    CustomerList findByName(in string name);
    CustomerList findByPhone(in string phone);
};

// Additional structure and sequence definitions for service interface

struct RentedMovieCopyInfo {
    long movieCopyId;
    string customerName;
    string customerPhone;
    Date returnDate;
};
```

```
    typedef sequence<RentedMovieCopyInfo> RentedMovieCopies;
    typedef sequence<long> InStoreMovieCopies;  // sequence of MovieCopy ids

    struct MovieInfo {
        long id;
        string name;
        string director;
        StringList genre;
        StringList cast;
        short year;           // production year
        short duration;       // movie duration in minutes
        InStoreMovieCopies inStoreCopies;
        RentedMovieCopies rentedCopies;
    };

    typedef sequence<MovieInfo> MatchingMovies;

    // Additional exception definitions for service interface

    exception InvalidMovieCopyIdException  { };
    exception InvalidCustomerIdException  { };

    // Definition of service interface

    interface RentalService {
        MovieInfo findMovieById(in long id)
            raises(NotFoundException);
        MatchingMovies findMovieByName(in string name);
        MatchingMovies findMovieByDirector(in string name);
        MatchingMovies findMovieByGenreAndYear(in string genre,
                                               in short year);
        void startRental(in long movieCopyId,
                         in long customerId,
                         in Date returnDate)
            raises(InvalidMovieCopyIdException,
                   AlreadyRentedException,
                   InvalidCustomerIdException);
        void endRental(in long movieCopyId)
            raises(InvalidMovieCopyIdException,
                   NotRentedException);
    };

};
```