ALCIDES ZANAROTTI JUNIOR

ANALISE COMPARATIVA DE ARQUITETURAS PARA GERENCIAMENTO DE OBJETOS EM AMBIENTES COMPUTACIONAIS DISTRIBUÍDOS.

Monografia apresentada à Escola Politécnica da Universidade São Paulo, para conclusão do Curso de MBA em Engenharia de Software.

Área de Concentração:
Ambientes Computacionais Distribuídos.

Orientadora:
Prof.ª Jussara Pimenta Mattos

São Paulo
2002

AGRADECIMENTOS

A amiga e orientadora, Jussara Pimenta Matos, pelas diretrizes seguras e permanente incentivo.

Aos meus familiares que me apoiaram em todos momentos.

Aos colegas que, direta e indiretamente, colaboraram na elaboração deste trabalho.

RESUMO

Este trabalho tem como objetivo apresentar e comparar as arquiteturas para gerenciamento de objetos em ambientes computacionais distribuídos, que integram as principais plataformas comercias, servindo de base para seleção de uma delas.

O desenvolvimento de sistemas para ambientes distribuídos representa um fator importante na escolha de uma arquitetura, sem um tratamento adequado, esses sistemas costumam apresentar resultados pouco produtivos. Portanto, a adoção de padrões no desenvolvimento de sistemas é recomendável, de forma que as metas das empresas possam ser atingidas.

Neste trabalho é referenciado um padrão para construção desse tipo sistema, e com base na sua especificação é possível desenvolver sistemas atendendo requisitos de portabilidade e de interoperabilidade.

De forma a auxiliar na escolha de uma arquitetura adequada para o gerenciamento de objetos em ambientes distribuídos, também são apresentadas as características das plataformas que as integram.

ABSTRACT

This work has as objective to present and to compare the architectures for object management in distributed computational environments, that integrate the main platforms deal, serving of base for election of one of them.

The development of systems for distributed environments represents an important factor in the choice of an architectures, without an adjusted treatment, these systems costumam to present little resulted productive. Therefore, the adoption of standards in the development of systems is recommendable, of form that the goals of the companies can be reached.

In this work a standard for construction of this type is referenciado system, and on the basis of its specification is possible to develop interoperabilidade and portabilidade systems being taken care of requisite.

Of form to assist it in the choice of an architecture adjusted for the object management in distributed environments, also the characteristics of the platforms are presented that integrate them.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

API - Application Program Interface

BLC - Basic Class Library

BMP - Bean Manager Persistence

CGI -Common Gateway Interface

CLS - Common Language Specification

CLR - Common Language Runtime

CMP - Container Bean Persistence

CORBA - Common Object Request Broker Architecture

CTS - Common Type System

DII - Dynamic Invocation Interface

DLL - Dynamic Link Library

EJB - Enterprise JavaBeans

ENC - Enviroment Naming Context

GC - Garbage Collector

HTTP - Hypertext Transfer Protocol

IDL - Interface Description Language

IIOP - Internet Inter -ORB Protocol

IL - Intermediate Language

IR - Interface Repository

JNDI - Java Name and Directory Interface

JIT - Just in time

JPS - Java Pet Store

JRE - Java Runtime Enviroment

JVM - Java Virtual Machine

J2EE - Java 2 Plataform, Enterprise Edition

OMA - Object Management Architecture

OMG - Object Management Group

ORB - Object Request Broker

PDS - Persistent Data Services

PE - Portable Executable

PO - Persistent Object

POM - Persistent Objects Manage

POS - Persistent Object Service

RMI - Remote Method Invocation

SOAP - Simple Object Access Protocol

TCP - Transmission Control Protocol

VES - Virtual Execution System

XML - Extensible Markup Language

LISTA DE TABELAS

LISTA DE FIGURAS

1 - INTRODUÇÃO

A computação em ambientes distribuídos é complexa e sujeita à falhas. A escolha da arquitetura adequada para o gerenciamento dos diferentes tipos de sistemas, que podem funcionar nesses ambientes, representa um fator importante para uma corporação.

Para suprir as necessidades decorrentes da natureza dos ambientes distribuídos, as arquiteturas para gerenciamento de objetos devem atender algumas características, como por exemplo: flexibilidade de comunicação em múltiplos protocolos; portabilidade; gerenciamento de objetos; infra-estrutura para implementação; escalabilidade.

Durante uma declaração do diretor de internet do banco Itaú (PINTO, A . P., 2001), ele cita: "Uma de nossas principais metas é oferecer serviços e soluções que utilizem tecnologia de ponta, agilizando e trazendo mais funcionalidades e modernidade para os nossos clientes. Estaremos utilizando a plataforma .NET para dar maior flexibilidade aos processos de comércio eletrônico". Nesta declaração, a plataforma .NET é citada como solução para problemas de flexibilidade aos processos de comércio eletrônico.

As características de uma plataforma são refletidas nas especificações de suas arquiteturas, algumas vezes, e as arquiteturas podem servir de base para a criação de uma plataforma.

1.1 - Considerações Iniciais

Atualmente, as duas principais plataformas que possuem arquiteturas com suporte para gerenciamento de objetos ambientes computacionais distribuídos são: a plataforma J2EE (Java 2 Plataform, Enterprise Edition) criada pela Sun Microsystems e a plataforma .NET fornecida pela Microsoft. A apresentação dessas

plataformas esta concentrada na especificação da arquitetura de cada uma delas, desde o funcionamento até a infra-estrutura para construção de sistemas.

Atualmente, a construção de sistemas para ambientes distribuídos representa um grande desafio para os arquitetos. Desenvolver um sistema não significa apenas solucionar o problema para o qual foi concebido, é necessário prever, por exemplo, necessidades de expansão, que podem surgir durante o período de sua utilização, de acordo com o tipo de aplicação. Algumas características e funções necessárias a um sistema são complexas e de difícil percepção durante o período de definição e desenvolvimento de um sistema.

A aderência a padrões para construção deste tipo de sistema é uma das formas de assegurar que as necessidades sejam supridas. Como conseqüência disto, o padrão CORBA (Common Object Request Broker Architecture) surgiu para especificar (CORBA,2002) como construir objetos para ambientes distribuídos, garantindo uma implementação segura de necessidades como, por exemplo, a interoperabilidade com outros objetos. Objeto é um termo usado dentro do contexto da técnica Orientada a Objetos (BOOCH et al., 1998) para representar um componente ou sistema, capaz de realizar operações ou fornecer serviços.

1.2 - Objetivo do Trabalho

Este trabalho tem como objetivo apresentar e comparar as arquiteturas para gerenciamento de objetos em ambientes computacionais distribuídos, que integram as duas plataformas comercias mais discutidas atualmente. Para tanto, são apresentados os principais critérios que podem servir de base para seleção de uma delas.

1.3 - Motivação do Trabalho

As empresas devem escolher cuidadosamente as arquiteturas disponíveis no mercado, em especial com suporte para ambientes distribuídos, porque a

dependência em relação a uma arquitetura pode causar um impacto significativo no desenvolvimento ou integração de sistemas ou aplicações em uma corporação.

A principal motivação para a elaboração deste trabalho é contribuir, através do conhecimento e experiência profissional do autor, adquiridos no desenvolvimento de sistemas para ambiente Internet, apresentando uma análise comparativa entre arquiteturas, de tal forma a auxiliar aqueles que desenvolvem sistemas com características similares.

1.4 - Estrutura do Trabalho

O capítulo 1 apresenta o objetivo e a motivação para a elaboração desta monografia, indicando os aspectos considerados importantes ao longo deste trabalho.

O capítulo 2 apresenta a especificação do padrão CORBA considerando algumas características relevantes como a arquitetura de comunicação entre objetos, a implementação de objetos e sua interoperabilidade com outros objetos.

Os capítulos 3 e 4 apresentam as plataformas J2EE e .NET, respectivamente. Estas plataformas utilizam o padrão CORBA, e a descrição de cada uma delas está concentrada na arquitetura para gerenciamento de objetos em ambientes distribuídos.

O capítulo 5 é apresenta uma análise comparativa entre a arquitetura EJB da plataforma J2EE e a arquitetura da plataforma .NET, onde é elaborado um quadro comparativo indicando como as principais características dessas arquiteturas estão relacionadas.

O capítulo 6 apresenta a conclusão final, as contribuições e sugestões para trabalhos futuros.

O processo de elaboração deste trabalho é apresentado no digrama de atividades da figura 1.1, localizada na próxima página.

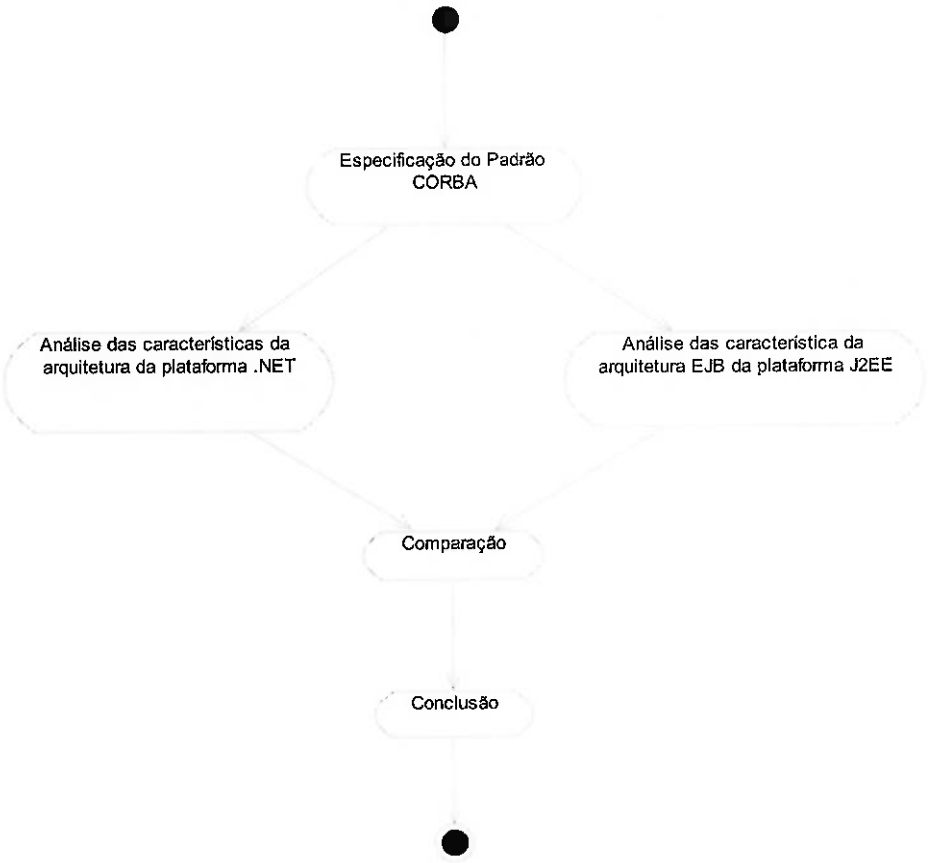

Figura 1.1 – Diagrama de atividades do processo de elaboração deste trabalho.

2 - ESPECIFICAÇÃO DO PADRÃO CORBA

2.1 - Histórico

Como uma forma de tratar e simplificar a computação em sistemas distribuídos heterogêneos, o consórcio OMG (Object Management Group) propôs a especificação do padrão CORBA (CORBA, 2002).

A OMG é um dos vários consórcios existentes, atualmente é composta de aproximadamente de 800 empresas, tendo como um dos objetivos a especificação de padrões a serem utilizados na construção de objetos para ambientes distribuídos, incentivando a reutilização de componentes lógicos através da técnica de orientação a objetos. Tendo como base os objetivos da OMG surgiu a OMA (Object Management Architecture) que define em um conjunto de padrões, a serem aplicados no projeto de uma arquitetura, sobre a qual objetos para ambientes distribuídos são construídos. O padrão CORBA faz parte da OMA (OMA,2002).

Normalmente, as redes de computadores apresentam características heterogêneas. Essa heterogeneidade ocorre tanto em relação aos dispositivos computacionais quanto em sistemas, sendo essa afirmação válida tanto em ambientes acadêmicos quanto comerciais. Na teoria, essa diversidade de componentes possibilitaria um maior número de configurações, cada uma das quais mais adequada ao desempenho de uma determinada tarefa ou para uma determinada situação. Porém, na prática nota-se que o controle dessas diferentes configurações é bastante complexo.

O padrão CORBA especifica como construir objetos, permitindo que se realizem requisições a outros objetos de forma transparente, independente da plataforma que esteja sendo usada, dessa forma é possível desenvolver produtos ou sistemas, de forma a atender requisitos de portabilidade e interoperabilidade, que devem estar refletidos na implementação através de qualquer linguagem de programação. O uso dessa especificação possibilita a construção rápida de sistemas para ambientes distribuídos, além de melhorar sua manutenção, devido às características aderentes ao padrão CORBA(CORBA, 2002)..

2.2 - Arquitetura de Comunicação entre Objetos

O padrão CORBA é mais do que um conjunto de regras para construção de objetos. Através de sua adoção, é possível utilizar uma infra-estrutura de serviços e utilitários que podem facilitar a construção e integração de sistemas em ambientes distribuídos (CORBA, 2002).

Em geral, nos ambientes distribuídos o cliente possui acesso a referências do objeto a ser requisitar, conhece a estrutura lógica do objeto, de acordo com sua interface. No padrão CORBA, a IDL(Interface Description Language) é responsável por informar aos clientes quais operações estão disponíveis e como elas podem ser requisitadas em um objeto remoto, construído baseado na especificação. Esta linguagem define o tipo do objeto pela especificação de sua interface, que consiste de uma série de operações e de parâmetros. Através da definição da IDL, é possível mapear objetos em qualquer linguagem de programação.

Diferentes linguagens de programação, orientadas a objetos ou não, podem requisitar os objetos que seguem o padrão CORBA. É importante ressaltar que o mapeamento feito com a IDL tem que ser o mesmo para todas as implementações do objeto, esse mapeamento inclui a definição específica dos tipos de dados e das regras para requisitar objetos.

Para o mapeamento de linguagens não orientadas a objeto, existe uma interface especializada, de acordo com o tipo da linguagem, denominada IDL stub. O IDL stub define operações identificáveis pelos desenvolvedores, uma vez que eles são familiarizados com a linguagem de programação que implementa a interface. Em alguns casos, por exemplo, rotinas de linguagens específicas podem ser usadas para sincronizar as linhas de controle do programa, como a requisição de componentes (HARKEY, D.; ORFALI, R, 1997).

O gerenciamento da comunicação entre objetos, em sistemas distribuídos que seguem o padrão CORBA é feito pelo ORB (Object Request Broker), fluxo de comunicação apresentado na figura 2.1, objeto responsável por:

- Prover os mecanismos necessários para encontrar a implementação de um objeto para o pedido de um cliente;
- Preparar a implementação do objeto para receber o pedido;
- Comunicar os dados para permitir a realização do pedido.

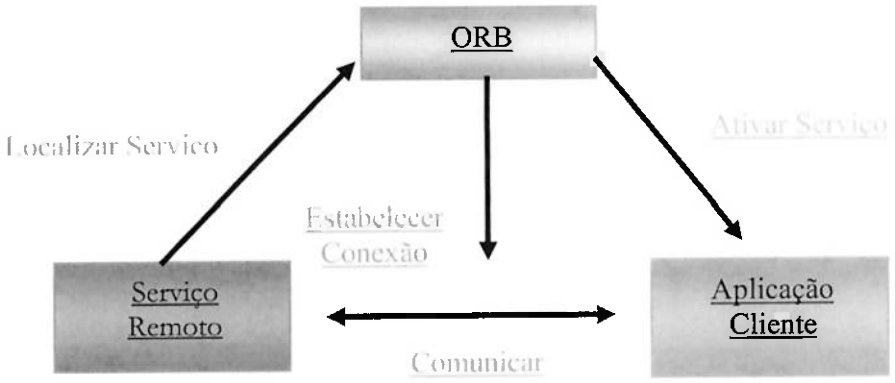

Figura 2.1- Representação da solicitação de um serviço através do ORB

Nota-se que o cliente realiza requisições, através do ORB que provê um gerenciamento de comunicação, a um nível de abstração que permite a utilização de um objeto independente da localização da sua implementação.

Algumas implementações de ORB possuem diferentes representações para objetos e diferentes maneiras de realizar requisições, isto é possível para um cliente que possui acesso simultâneo a duas referências de objetos gerenciadas pelo mesmo ORB. A especificação da interface do ORB está organizada em 3 categorias:

- Operações comuns para todas implementações do ORB;
- Operações específicas para tipos particulares de objetos;

- Operações específicas para estilos particulares de implementação de objeto.

A Figura 2.2 apresenta a estrutura individual de um ORB. As setas indicam se o ORB é chamado ou faz uma chamada. O atendimento das requisições de um cliente são realizadas de duas formas:

- Estática: é possível definir uma interface de objeto estaticamente através de uma IDL stub, definindo a propriedade do objeto como estática;
- Dinâmica: é possível adicionar uma interface dinamicamente no serviço de IR (Interface Repository) através de uma DII (Dynamic Invocation Interface), permitindo acesso a esses componentes durante o tempo de execução

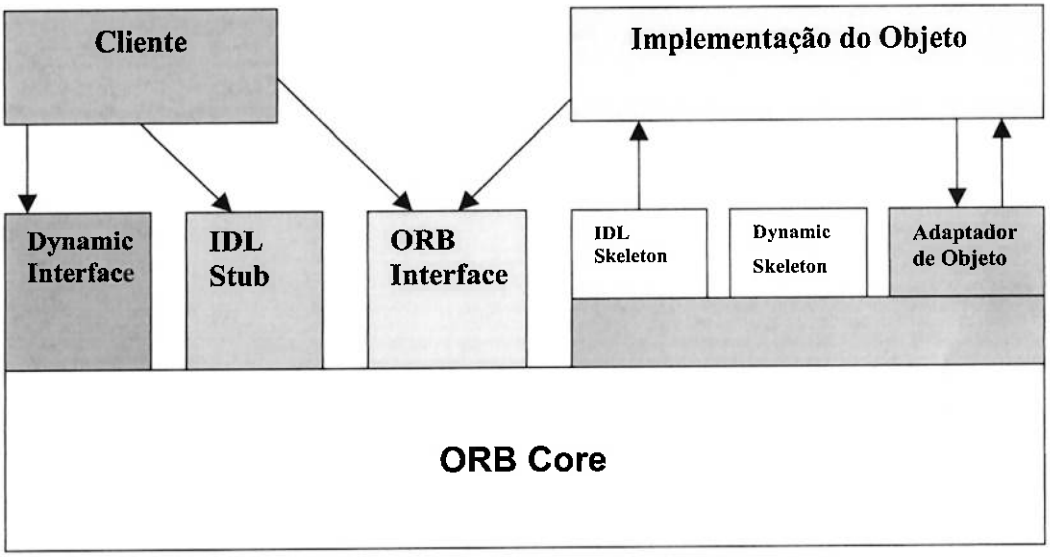

Figura 2.2 - Estrutura individual de um ORB (Adaptada da referência (CORBA,2002)).

O IR constitui o serviço responsável por fornecer informações da IDL, disponíveis em tempo de execução. As informações fornecidas pelo IR são usadas para realizar requisições. Utilizando estas informações, é possível que o programa encontre

objetos em que a interface é desconhecida no processo de compilação, mas desta forma, é possível determinar as operações válidas para este objeto e invocá-lo.

O DII representa a interface que permite a implementação dinâmica de requisições a objetos, isto é, ao invés de chamar uma rotina específica de uma operação particular de um objeto, o cliente pode especificar no objeto a ser invocado uma operação através da requisição. Assim, o código cliente pode conter informações sobre as operações a serem realizadas e os tipos de parâmetros a serem passados (estes são obtidos da IR ou de outra fonte em tempo de execução).

O gerenciamento da comunicação entre objetos por um ORB acontece da seguinte forma:

- O cliente faz uma requisição tendo acesso a uma referência do objeto;
- A requisição encontra um IDL stub especifico para aquele objeto;
- A requisição cria uma DII.

Este processo é apresentado na figura 2.3, tanto o IDL stub quanto a DII satisfazem a mesma semântica de requisição.

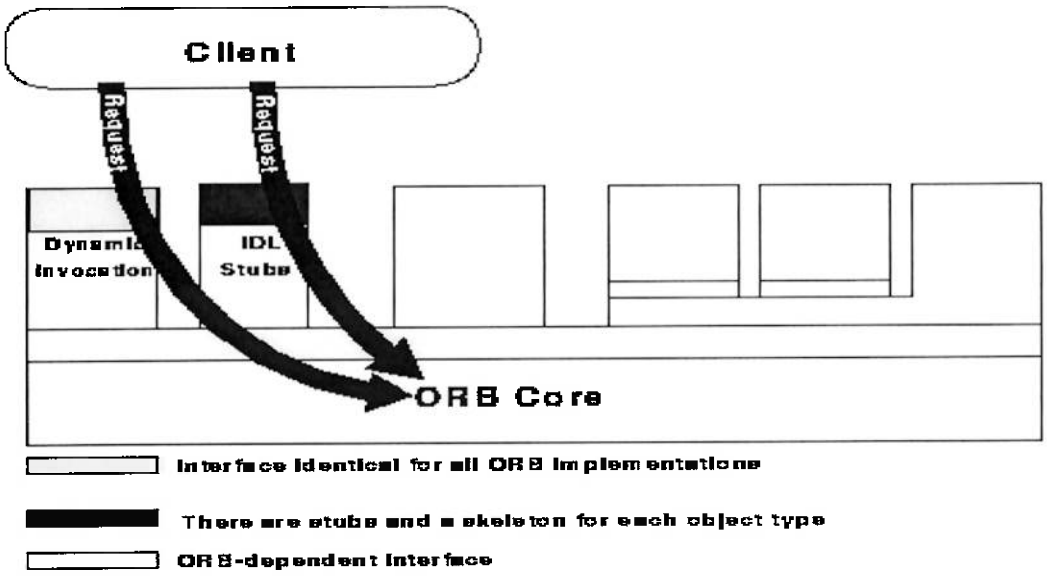

Figura 2.3 - Requisição de um cliente (CORBA,2002).

Em seguida, o ORB localiza a implementação do código desejado, transmite os parâmetros e transfere o controle da implementação do objeto através do adaptador de objeto, fluxo apresentado na figura 2.4.

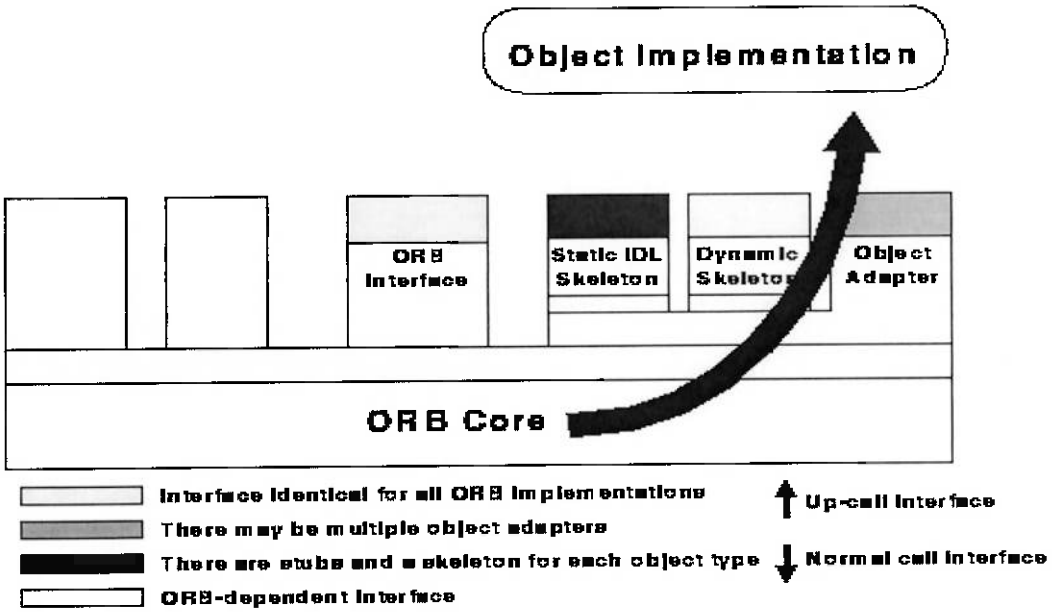

Figura 2.4 - ORB transferindo a requisição do cliente (CORBA,2002).

O adaptador de objeto é o componente responsável por exportar a interface para a implementação do objeto, tendo as seguintes funções:

- Geração e interpretação de referência de objetos;
- Métodos de invocação;
- Segurança das interações;
- Registro das implementações;
- Ativação/desativação de objetos;
- Mapeamento da referência de objetos para a correspondente implementação de objeto;

O adaptador de objetos define vários serviços do ORB, que a implementação de objeto requisitado necessita. Diferentes ORBs fornecem diferentes níveis de serviços

e diferentes ambientes de operação que tem algumas propriedades implícitas e necessitaram que outras sejam adicionadas pelo adaptador de objetos. Por exemplo, é comum para a implementação do objeto a necessidade de salvar certos valores de referência de objetos, para obtenção de uma fácil identificação do objeto invocado. Se o adaptador de objeto permite em uma implementação especificar certos valores quando um novo objeto é criado, ele pode ser capaz de guardá-los em uma referência de objeto para o ORB.

Não é necessário que todos os adaptadores de objetos forneçam a mesma interface e funcionalidade. Algumas implementações de objetos têm necessidades especiais, por exemplo, uma base de dados orientada a objeto pode registrar implicitamente vários objetos sem fazer requisições individuais ao adaptador de objeto. Neste caso, é desnecessário para o adaptador de objetos manter qualquer objeto por estado.

Existem vários tipos de adaptadores de objetos, a maioria é projetada para atender diferentes formas de implementação de objetos, assim, somente são considerados novos adaptadores de objetos, quando a implementação requisitar diferentes serviços ou interfaces.

Através dessas especificações, é possível desenvolver implementações de objetos localizados em qualquer lugar, encapsulados como componentes que clientes remotos podem acessar através de requisições de métodos.

2.3 - Tipos de Implementação de Objetos

Geralmente, as implementações de objetos não dependem do ORB, elas podem selecionar interfaces para serviços independentes do ORB, através da escolha do adaptador de objetos. A implementação do objeto provê o estado atual e o comportamento com do objeto. Além da definição de métodos para suas próprias operações, uma implementação usualmente define procedimentos para ativar e desativar objetos, e usará outros objetos ou estruturas para tornar o estado do objeto persistente para controle de acesso. A implementação do objeto, como é apresentado

na figura 2.5, interage com o ORB de diversas maneiras para criar novos objetos e para obter serviços dependentes do ORB.

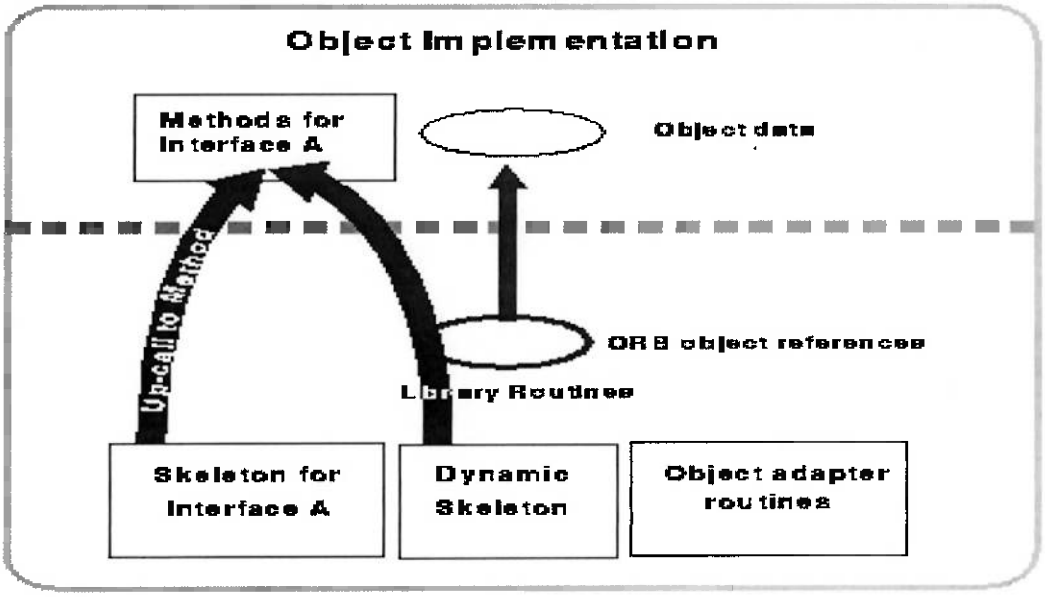

Figura 2.5 - Interação da implementação do objeto com o ORB (CORBA,2002).

Devido a grande variedade de implementações de objetos, é difícil definir um padrão. Há quatro tipos de implementações principais (OMG ,2002):

- Serviços (CORBA services) - são especificações de objetos que provêm a assistência fundamental e de baixo nível para todas as aplicações Se existir um mecanismo de comunicação adequado, o ORB pode ser implementado com rotinas residentes no cliente e implementações. O cliente pode acessar direto à localização do serviço para estabelecer comunicação com as implementações;
- Utilitários (CORBA facilities) - um conjunto de especificações de alto nível que fornece um conjunto de serviços exigidos por muitas aplicações Para centralizar o controle do ORB, todos clientes e implementações podem comunicar-se com um ou mais servidores cujo trabalho é rotear as requisições dos clientes para as implementações;

- Interfaces de Domínios de Aplicação (CORBA domains) - especificações para atender uma área específica de aplicações. Para realçar os requisitos de segurança, robustez e desempenho, o ORB pode utilizar serviços básicos de um sistema operacional. Referências de objeto podem ser feitas de forma a não serem corrompidas, reduzindo assim o custo de autenticidade de cada requisição;
- Interfaces de Aplicação (CORBA applications) - são interfaces específicas de aplicações do usuário e, portanto não são padronizadas. Para objetos simples e que a implementação pode ser dividida, a implementação utiliza bibliotecas. Neste caso, o IDL stub pode ter os métodos atuais, isto é, assume que é possível para o programa cliente acessar os dados do objeto e desta forma, a implementação garante que o cliente não modificará os dados.

Através desses tipos de implementações, é possível construir objetos seguindo a especificação do padrão CORBA e que atendam a diferentes finalidades, interagindo em um ambiente distribuído heterogêneo.

2.4 - Interoperabilidade de Objetos

Como previsto pela OMG(OMG,2002), existe atualmente uma grande variedade de produtos que obedecem à especificação do padrão CORBA. Entretanto, devido à grande flexibilidade permitida, implementações de ORB que possuem a mesma regra de negócio diferem, refletindo em soluções peculiares em função dos fabricantes, não só pela utilização de diferentes mecanismos para a obtenção das mesmas funcionalidades, como também através do acréscimo de novas funções consideradas importantes para o seu usuário final. Isso corresponde às decisões técnicas relativas, como por exemplo: o tempo despendido por uma requisição; os níveis de segurança ; uso de determinados protocolos de comunicação.

A interoperabilidade entre objetos está basicamente associada com uma mudança transparente de domínio. Considera-se domínio como um contexto em que certas características e/ou regras comuns são preservadas. Há uma tendência de que esses

domínios sejam, basicamente, de cunho administrativo (nomes, grupos, gerenciamento de recursos, segurança, etc) e/ou tecnológico (protocolos, sintaxes, redes, etc), sendo que os mesmos não correspondem, necessariamente, aos limites de um ORB instalado.

Domínios possibilitam o particionamento de um contexto em grupos de objetos que tenham características em comum. Um determinado objeto pode, portanto, fazer parte de mais de um domínio, desde que satisfaça aos seus requisitos. Considera-se o limite de um domínio, como um contexto, no qual uma determinada característica tem algum significado. Como exemplos de domínios, pode-se citar os contextos de:

- Uma referência de objetos;
- Uma sintaxe de transferência de mensagens;
- Um endereço;
- Uma mensagem de rede;
- Uma política de segurança;
- Um identificador de tipos;
- Um serviço de transações.

A Interoperabilidade só é possível através de uma conexão entre domínios. Para atender ao requisito, é preciso traduzir como fazer para que um objeto Y, no ORB B, apareça como um objeto X, no ORB A, de maneira que esta última seja capaz de utilizar X da mesma forma que faria com um objeto qualquer que fosse, de fato, implementado por ela, nem sempre de fácil implementação. Além disso, todas as funcionalidades fornecidas pelo ORB B devem ser acessíveis pelo ORB A através de X. Com isso, uma requisição em X deve ser transformada numa requisição em Y e, para tanto, é preciso criar X através da passagem de Y para o ORB A. O objeto X será, então, um representante de Y no ORB A, recebendo a denominação de Proxy de Y.

Durante a conversão da requisição pode ser necessário o mapeamento de outros domínios, além do definido pela referência de objeto. Múltiplos domínios podem

estar sendo acessados simultaneamente e cada conversão será igualmente necessária para o completo entendimento pela ORB destino.

Esta interoperabilidade entre diferentes ORBs é provida pela IIOP(Internet Inter - ORB Protocol). O IIOP corresponde a um protocolo de comunicação, com mapeamento TCP (Transmission Control Protocol), permitindo que requisições sejam enviadas para objetos distribuídos gerenciados por outros ORBs em outros domínios (MOWBRAY, T.J.; ZAHAVI R., 1995). A Figura 2.6, abaixo, apresenta um esquema utilizado pelo IIOP para comunicação entre ORBs.

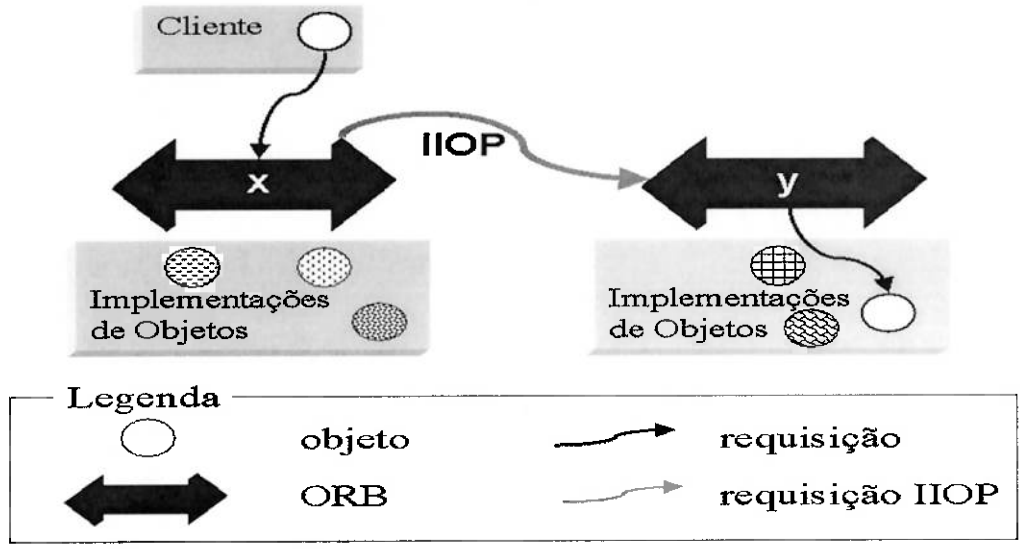

Figura 2.6 - IIOP - Comunicação entre ORBs.

O TCP é um protocolo orientado a conexão que fornece um serviço confiável de transferência de dados fim a fim. O TCP foi projetado para funcionar com base em um serviço de rede sem conexão e sem confirmação, interagindo de um lado com processos de aplicações e de outro lado com protocolos da arquitetura Internet (SOARES, L.F.G; LEMOS, G;COCHER, S, 1995)

2.5 - Persistência de Objetos

O estado de um objeto é armazenado em um meio não volátil (MOWBRAY, T.J.; ZAHAVI R., 1995) como, por exemplo, em um banco de dados ou um sistema de

arquivos. O estado de um objeto pode ser considerado como composto por duas partes: um estado dinâmico, que existe tipicamente em memória e não precisa necessariamente existir durante o tempo de execução objeto, e um estado persistente, que o objeto poderia usar para reconstruir um estado dinâmico. É esse estado persistente que deve ser armazenado e gerenciado em um meio de armazenamento persistente.

O POS (Persistent Object Service), definido no padrão CORBA, permite que um objeto continue existindo ao término da aplicação que o criou ou do cliente que o utilizou, pois o estado do objeto pode ser salvo em um meio de armazenamento persistente e restaurado, se necessário. O objetivo do POS é prover interfaces comuns para os mecanismos usados no armazenamento e gerenciar estados persistentes de objetos, portanto, tem como uma das responsabilidades principais, o armazenamento do estado persistente de objetos, com outros serviços provendo outras capacidades.

O ORB possui a habilidade de manter a referência a um objeto de forma persistente, mas isso não garante que um determinado objeto esteja disponível apenas porque a sua referência ainda é válida. O POS fornece:

- Suporte para base de dados coorporativas, incluindo bancos de dados e bases de dados baseados em sistemas de arquivo;
- Independência de base de dados, isto é, uma única API(Application Program Interface) cliente independente de uma base de dados particular e um mecanismo único para armazenamento ou restauração de objetos a serem usados no lado do servidor.

Buscando a possibilidade de implementação da persistência em diferentes meios de armazenamento, o POS estabelece uma interface única de objetos para múltiplas bases de dados apresentados na figura 2.7.

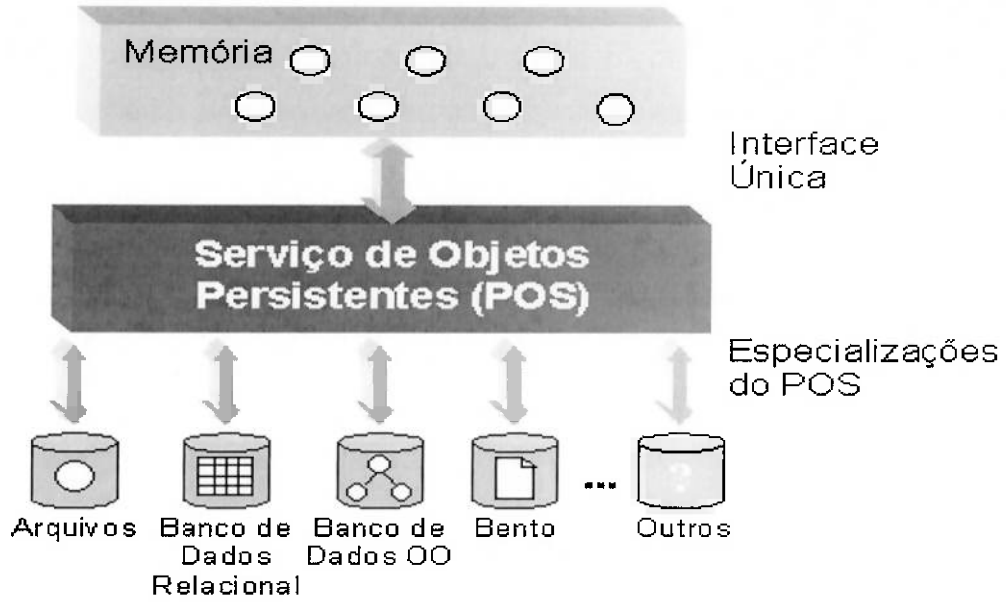

Figura 2.7 - Implementação de persistência em diferentes meios

De forma mais detalhada, apresentada na figura 2.8, o POS é formado por quatro elementos:

- PO (Persistent Object): são os objetos cujos estados são armazenados de forma persistente. Um objeto pode tornar-se persistente, utilizando a propriedade de herança do comportamento da classe PO, via IDL. Um objeto persistente deve herdar ou prover um mecanismo para externalizar seu estado quando o mecanismo de armazenamento solicitar. Essa solicitação é feita através de um protocolo específico. Cada PO possui um identificador persistente que descreve a sua localização dentro da base de dados utilizando uma string de identificação. Clientes tipicamente interagem com a interface de objetos persistentes para controlar a persistência do objeto;
- POM (Persistent Objects Manage): é uma interface independente da implementação para operações de persistência. O POM provê um acesso uniforme aos diferentes tipos ou instâncias de serviços de persistência de dados;

- PDS (Persistent Data Services): são interfaces para implementações particulares de bases de dados. O PDS realiza a tarefa de mover dados entre um objeto e uma base de dados;
- Datastores: são implementações que armazenam um dado persistente de um objeto, independente do espaço de endereçamento que contém o objeto.

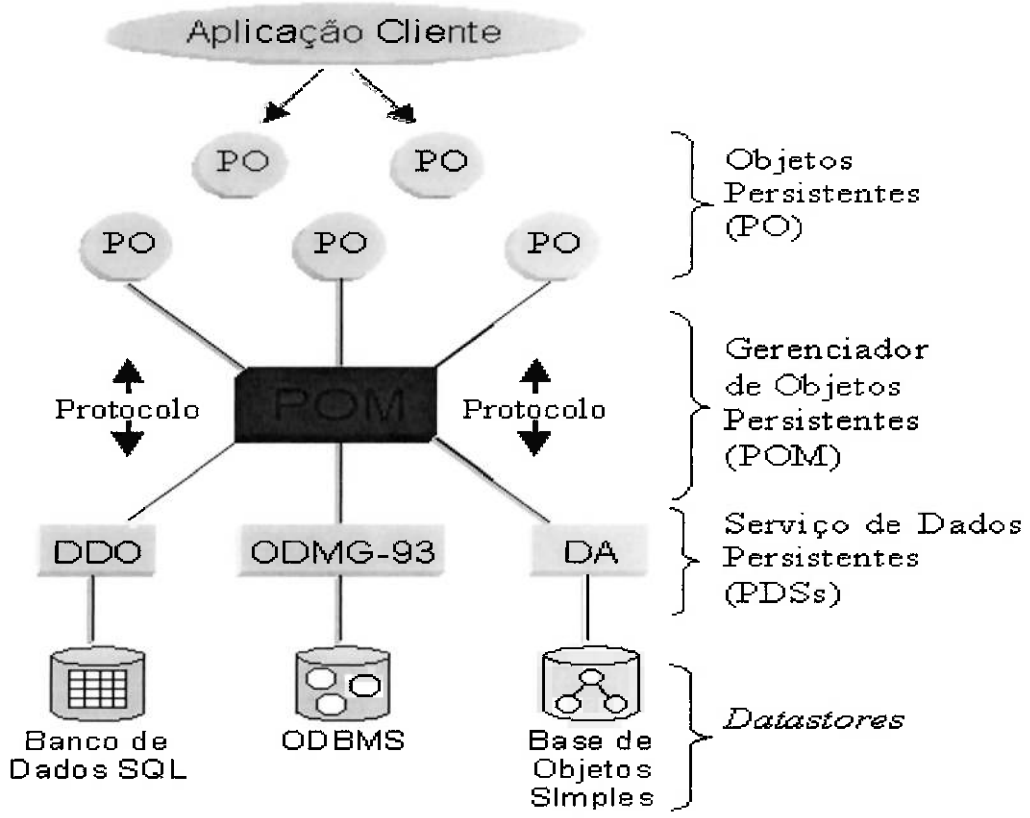

Figura 2.8 - Elementos do POS

O objetivo desse serviço é guardar o estado de um objeto de forma persistente. Este serviço não garante que um objeto continue disponível e sim que seu estado possa ser restaurado, caso seja necessário. O POS é um padrão relativamente recente que ainda possui poucas implementações. Esse serviço é importante quando são implementadas aplicações tolerantes à falhas ou na migração de sistemas legados. Uma das principais vantagens dessa especificação é a possibilidade de guardar o estado do objeto em diferentes meios de armazenamento. Por outro lado, o controle dessa diversidade não está muito bem tratado no padrão. Existe a possibilidade de um

objeto persistente se comunicar tanto como Gerenciador de Objetos Persistente quanto diretamente com o Serviço de Dados Persistentes. A comunicação com o último se daria através do componente Protocolo que não está completamente especificado.

2.6 - Replicação de Objetos

A replicação de componentes em sistemas distribuídos normalmente é utilizada para torná-lo mais confiável. A replicação envolve a manutenção da consistência entre as múltiplas cópias, isto é, é preciso garantir que todas as cópias possuam o mesmo estado. As duas políticas de replicação mais conhecidas são replicação ativa, onde todas as cópias do objeto replicado tratam uma requisição e retornam um resultado e replicação passiva, onde apenas uma cópia, geralmente denominada primária, executa os pedidos e atualiza as outras cópias.

Uma possibilidade de implementar replicação é a utilização de um serviço de comunicação em grupo, existindo três abordagens possíveis de replicações utilizando a especificação CORBA:

- Integration Approach: consiste na integração de um sistema de comunicação de grupo existente dentro de um ORB. Essa abordagem é fácil de desenvolver, pois não é preciso desenvolver uma nova camada de comunicação em grupo, sendo transparente para o cliente;
- Interception Approach: consiste na interceptação de mensagens relacionadas a um ORB existente e o mapeamento dessas mensagens para um pacote de comunicação em grupo. A principal vantagem, é que essa abordagem não necessita de qualquer modificação no ORB;
- Service Approach : consiste em prover a comunicação em grupo como um serviço de CORBA ao lado do ORB. Essa é a abordagem com maior conformidade ao padrão CORBA, ela modela a replicação no mesmo nível de outras funcionalidades, como persistência e transação, incluídas como serviços.

A implementação de replicação de objetos é importante para garantir requisitos de tolerância à falhas e a disponibilidade dos objetos. Existem várias propostas de implementação, porém considera-se a implementação da replicação como um serviço CORBA (CORBA, 2002). Existe atualmente um grupo na OMG buscando propostas de padronização de serviços para tolerância á falhas, onde, possivelmente, a replicação será de alguma forma tratada.

3 - ARQUITETURA EJB DA PLATAFORMA J2EE

3.1 - Histórico

Há oito anos que a plataforma Java foi criada, e uma é das plataformas de desenvolvimento de sistemas mais discutidas da atualidade. Em seu lançamento, os programas construídos, Applets (APPLETS, 2002), tinham foco de execução nos navegadores de Internet.

Applets são programas carregados a partir de um servidor de Internet e são executados em um navegador no computador cliente (requisitante). Os Applets, em geral, dependem dos navegadores que os suportam para serem executados, mas podem ser executados também com uma ferramenta chamada AppletViewer(APPLETS, 2002). Como são executados através de um navegador, consequentemente, também tem acesso às mesmas capacidades deste: gráficos sofisticados, desenho e pacotes de processamento de imagens, elementos de interface de usuário, serviço de rede e tratamento de eventos.

Executando de forma completamente diferente e dependendo dos navegadores, os Applets foram alvo de várias críticas contra a adoção da plataforma Java. Alguns fabricantes de plataformas concorrentes não haviam percebido é que enquanto os códigos executados em clientes não correspondiam às respectivas funcionalidades, a plataforma Java estava sendo utilizada, de forma eficaz, os servidores de aplicações, principalmente os servidores de Internet.

Inicialmente, a plataforma Java utilizada em servidores de Internet estava limitada ao Servlet Engine(SERVLET,2002), mecanismo que permite a construção de sistemas executados em servidores de Internet utilizando CGI (Common Gateway Interface).

O CGI é um protocolo de comunicação através do qual o servidor de Internet gerencia a transferência de informações entre um programa residente no servidor e

um navegador de Internet utilizado no computador cliente.O CGI é um protocolo e não um programa executável capaz de realizar alguma coisa.

Originalmente, o modelo de componentes da plataforma Java foi representado pelo JavaBeans, porém com o crescimento e utilização da plataforma surgiu a necessidade de uma arquitetura para componentes em ambientes distribuídos, surgindo a arquitetura EJB. O modelo JavaBeans foi a agregado a EJB, que passou a representar a arquitetura de componentes para o desenvolvimento e implantação em ambientes distribuídos para plataforma Java. Desde sua criação, há dois anos, vários fornecedores vêm se interessando por ela, isto porque (EJB, 2001):

- Os servidores de EJB fornecem sustentação automática para serviços tais como transações, segurança e conectividade da base de dados;
- O Código é escrito uma única vez e executado em qualquer sistema operacional (Write Once, Run AnywhereTM);
- Simplifica o desenvolvimento dos componentes;
- Permite acesso compartilhado por múltiplos usuários.

Para entender melhor este interesse, pode-se tomar como exemplo, o gerenciamento de uma transação. No passado, os desenvolvedores tinham que escrever e manter o código de gerência de transações, ou confiar em um sistema de gerência de transações adquirido. A arquitetura EJB permite que os componentes simplesmente participem nas transações, apenas especificando objetos e métodos que são transacionais. Os servidores de EJB abstraem os detalhes de gerência das transações, assim os desenvolvedores podem concentrar seus esforços no desenvolvimento das regras de negócio.

A arquitetura EJB é uma das muitas especificações de arquiteturas contidas na plataforma J2EE. É normal confundir as duas, porque uma grande parte dos conceitos da plataforma J2EE origina-se da arquitetura EJB. A figura 3.1, na próxima página, apresenta a plataforma J2EE representada no modelo de três camadas (apresentação, negócio e dados).

O modelo de três camadas foi proposto, principalmente, para melhorar a eficiência de busca e recuperação de informação Este modelo pode ser visto como um caso particular de um tipo de coordenação com funções determinadas, ou seja, cada camada tem seu papel pré-determinado na solução do problema.

A plataforma J2EE é um conjunto de especificações e um guia de práticas, que juntos permitem o desenvolvimento, instalação, execução e gerenciamento de aplicações com n-camadas. A plataforma J2EE expande a plataforma Java, sendo completa, robusta, estável, segura e de alto desempenho, voltada para o desenvolvimento de soluções corporativas.

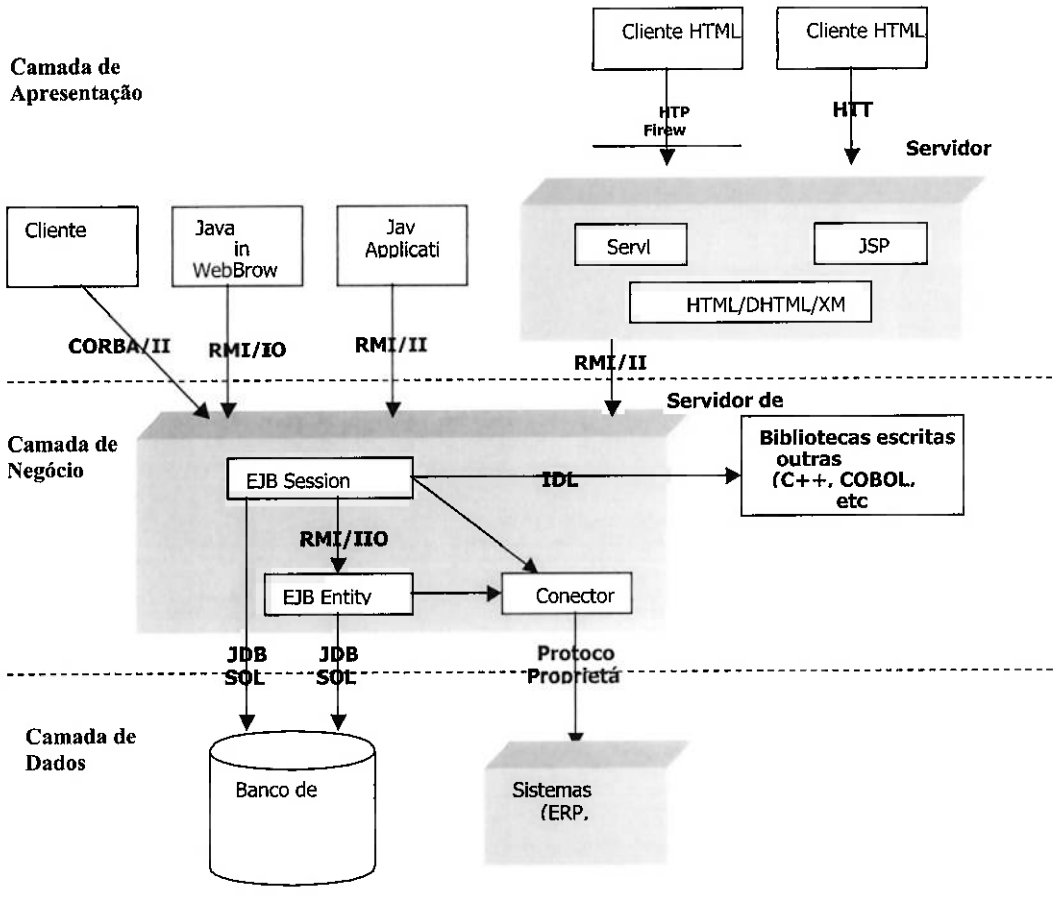

Figura 3.1 - Plataforma J2EE em função do modelo de três camadas(MATOS, J. P., 2002)

Na figura 3.1, pode ser observado que a arquitetura EJB é direcionada para a camada de negócio, fornecendo recursos de infra-estrutura e serviços necessários para um ambiente distribuído. Basicamente, a camada de negócios é responsável entre outros serviços por controlar e mediar as outras duas camadas (apresentação e dados).

3.2 - ARQUITETURA EJB

A figura 3.2 apresenta os componentes da arquitetura EJB mais detalhadamente.

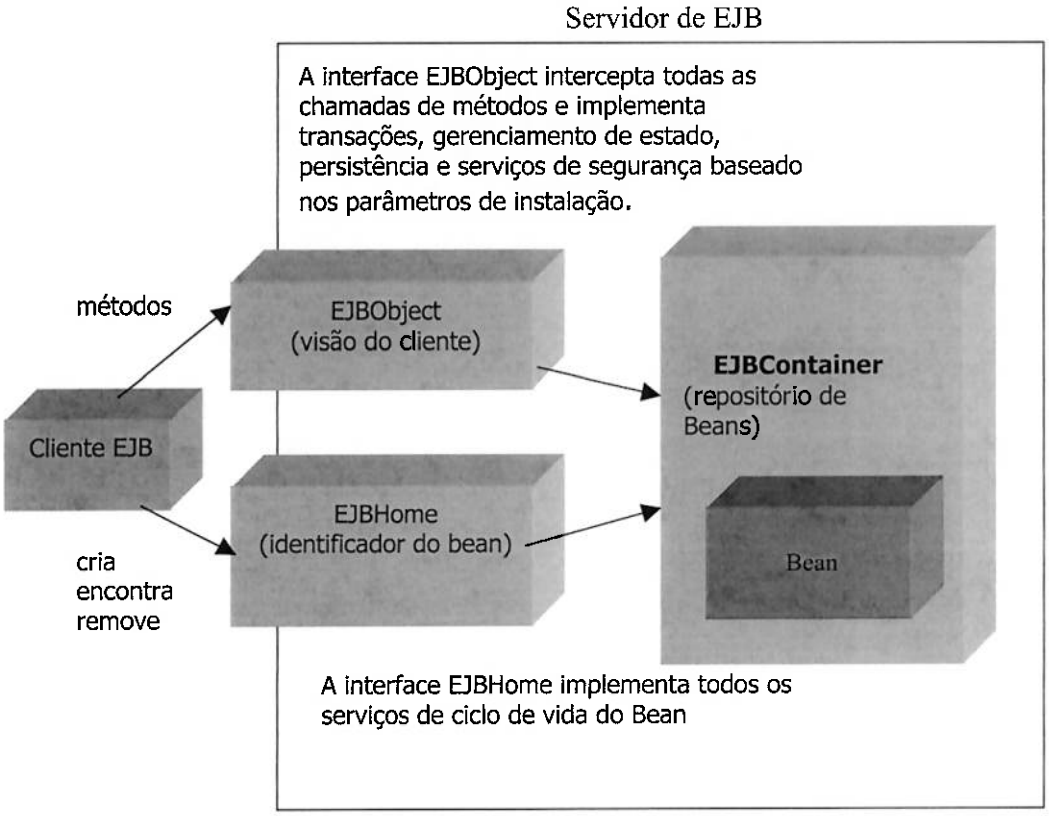

Figura 3.2 - Componentes da arquitetura EJB(MATOS, J. P., 2002)

3.2.1 - Cliente EJB

O Cliente EJB é o responsável por encontrar no Servidor EJB o recipiente (EJBContainer), que contém o componente de uma determinada operação. Através de JNDI (Java Name and Directory Interface), apresentado a seguir, o recipiente faz o mapeamento deste componente, omitindo os detalhes de execução. A seguir, é

apresentado um exemplo de parte de código em linguagem Java, que implementa uma classe cliente realizando um mapeamento através do JNDI.

```
public class Client
{
 public static void main(String [] args)
 {
  try
  {
   Context jndiContext = getInitialContext();
   Object obj = jndiContext.lookup("java:env/ejb/user");
   User user = (User)javax.rmi.PortableRemoteObject.narrow(obj,SiteHome.class);
   Info info = user.create(1);
   site.setName("Joao");

   UserPK pk = new UserPK();
   pk.id = 1;
   Info info_2 = user.findByPrimaryKey(pk);
   System.out.println(info_2.getName());

  }
  catch (java.rmi.RemoteException re){re.printStackTrace();}
  catch (javax.naming.NamingException ne){ne.printStackTrace();}
  catch (javax.ejb.CreateException ce){ce.printStackTrace();}
  catch (javax.ejb.FinderException fe){fe.printStackTrace();}
 }
}
```

A principal finalidade do JNDI é fornecer um serviço de nomes, permitindo a associação de um nome, ou uma outra representação alternativa mais simples, a recursos computacionais tal como, endereços de memória, de rede e referências códigos em geral. As duas funções básicas são:

- Associar um nome a um recurso;
- Localizar um recurso a partir de seu nome.

Como exemplo, suponha um sistema de arquivos, indicando a ligação do caminho que associa a árvore de diretórios, onde se localiza um arquivo, a um bloco de memória.

c:\temp\dados.txt	16A0:0C00

O JNDI é uma extensão da plataforma Java, que fornece infra-estrutura para construção de sistemas, que necessitam da associação de nomes a recursos computacionais (J2EE, 2002).

3.2.2 - Servidor EJB

O Servidor de EJB é o responsável pela infra-estrutura de gerenciamento, tal como, controle de transações, persistência de objetos, segurança, etc. Além disso, outros requisitos importantes para um ambiente distribuído são suportados, tal como, escalabilidade, portabilidade, incorporados da plataforma Java.

No Servidor EJB encontram-se os Beans, componentes desenvolvidos ou de fornecedores, que ficam armazenados especificamente no EJBContainer, podendo ser definido como recipiente e/ou repositório de Beans.

As funções principais do EJBContainer são de assegurar que a persistência, transação e segurança sejam aplicadas corretamente a toda operação que um Cliente EJB faça a um determinado Bean.

A figura 3.3, localizada na próxima página, apresenta o funcionamento do EJBContainer.

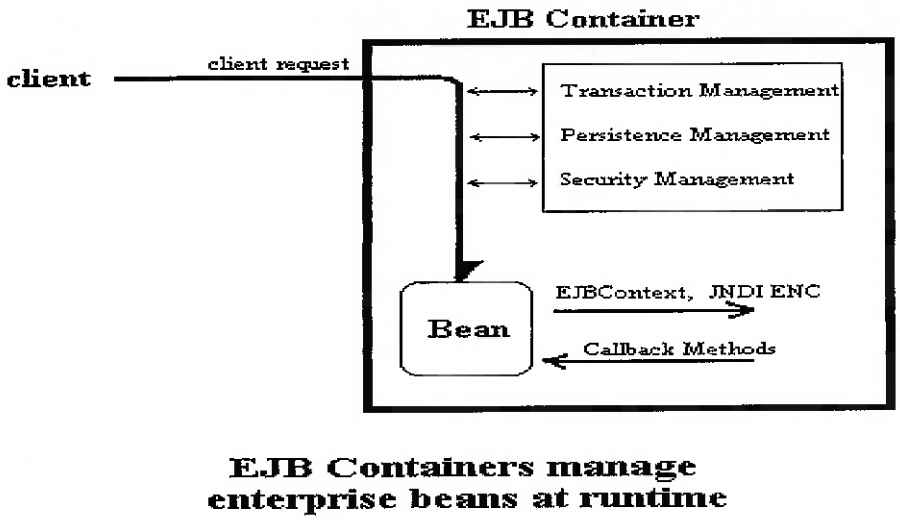

Figura 3.3 - EJBContainer (EJB,2001)

Para reduzir a utilização de processador e memória da aplicação, o EJBContainer cria um pool de recursos e gerencia o ciclo de vida de todos os Beans, sendo que, mais de um Bean pode ser gerenciado simultaneamente. Quando um Bean não está sendo utilizado, o EJBContainer o coloca no pool para ser usado por outro cliente, e somente o carrega novamente quando for necessário.

Além disso, o EJBContainer pode gerenciar conexões a banco de dados e acesso a outros Beans. Um Bean pode interagir com o EJBContainer de três formas diferentes:

- Métodos CallBack: todo Bean implementa uma interface a EJBHome que define diversos métodos chamados callback. Cada método callback é invocado em diferentes momentos no ciclo de vida do bean, tal como ativação de um bean, gravação de dados, remoção de um bean da memória, entre outros. Deve-se saber quando cada método é chamado e qual a sua utilização para implementar uma aplicação de maneira correta e eficiente.
- EJBContext: todo Bean deve instanciar um objeto do tipo EJBContext, que é a referência direta para o EJBContainer. Esse objeto possui métodos

para obter informações de uma requisição, tal como a identidade do cliente, o estado de uma transação ou obter referências remotas para si mesmo.

- JNDI: todo Bean automaticamente tem acesso a um sistema especial de nomes, denominado ENC(Enviroment Naming Context). O ENC é gerenciado pelo EJBContainer e acessa os Beans usando JNDI. Com isso, o Bean pode ter acesso a outros Beans e a propriedades específicas de um Bean.

Os Beans são desenvolvidos através da linguagem Java, e são responsáveis por realizar determinadas operações dentro do seu contexto. Para o desenvolvimento de um Bean é necessário seguir um padrão, que tem como objetivo manter compatibilidade de versões futuras do JavaBeans.

No servidor, o Bean tem seu tempo de execução limitado pelo cliente que o requisitou ou então, quando o servidor deixa de funcionar. Caso esteja sendo compartilhado por dois clientes, o seu tempo de execução limitado a requisição do último cliente solicitante. Os Beans com o comportamento acima, são também denominados Beans de sessão, as principais características desses Beans são:

- Encapsular operações complexas com as entidades do sistema;
- Retirar o gerenciamento do negócio do cliente;
- Simplificar ao máximo o código do cliente.A seguir, é apresentado um exemplo de código em linguagem Java, que define uma interface remota para um Bean de sessão.

```
public interface TravelAgent extends javax.ejb.EJBObject
{
 public void setCruiseID(int cruise)
  throws RemoteException, FinderException;
 public int getCruiseID()
  throws RemoteException, IncompleteConversationalState;
 public void setCabinID(int cabin)
  throws RemoteException, FinderException;
 public int getCabinID()
  throws RemoteException, IncompleteConversationalState;
 public int getCustomerID( )
```

```
    throws RemoteException, IncompleteConversationalState;
  public Ticket bookPassage(CreditCard card, double price)
    throws RemoteException,IncompleteConversationalState;
  public String [] listAvailableCabins(int bedCount)
    throws RemoteException,IncompleteConversationalState;
}
```

Existem dois tipos de sessão que podem ser implementadas para esses Beans:

- Stateful: usada quando é preciso que a sessão armazene algum valor. Este tipo tem a seguinte característica, apenas um único cliente acessa um único Bean e durante a requisição, os dados do Bean são armazenados na sessão. Os dados serão apagados somente quando o cliente remover o Bean. Essa abordagem deve ser implementada com cuidados extras, esta implementação afeta diretamente o desempenho do sistema;
- Stateless: não mantém as informações do Bean em sessão, pode ser compartilhado entre diversos clientes, nesta abordagem o desempenho é mais otimizado.

Assim, o Bean de sessão pode ser comparado a verbos porque normalmente executam uma ação qualquer, que pode ser: chamar um outro Bean; finalizar uma compra; calcular um desconto e etc.

Existem também os Beans persistentes, são usados geralmente para controlar transações, seja com um banco de dados ou com outros Beans, as principais características são:

- Possuir um tempo de execução longo;
- Permitir acesso compartilhado por múltiplos usuários;
- Fornecer uma visão em forma de objeto dos dados de um repositório.

A seguir, é apresentado um exemplo de parte de um código que demonstra a construção de um Bean persistente. A classe ProductEJB implementa as regras impostas pela interface EntityBean que define os métodos básicos de um Bean persistente.

```
public class ProductEJB implements EntityBean
{
 public String productId;

 private EntityContext context;
 private double price;
 private String description;

 public void setPrice(double price)
 {
  this.price = price;
 }

 public String ejbCreate(String productId, String description,
  double price) throws CreateException
 {

  if (productId == null)
  {
   throw new CreateException();
  }

  this.productId = productId;
  this.description = description;
  this.price = price;

  return null;
 }

 public void ejbRemove() { ... }
 public void ejbLoad() { ... }
 public void ejbStore() { ... }
 public void unsetEntityContext() { ... }
 public void ejbPostCreate(String productId, String description,
   double balance) { ... }
}
```

Existem dois modos de implementação de persistência para esses Beans:

- BMP (Bean Manager Persistence): o desenvolvedor escreve o código necessário, por exemplo, para acessar o banco de dados. Este modo de implementação pode acarretar alguns inconvenientes, tal como, o aumento da possibilidade de erros e de tempo de desenvolvimento;

- CMP (Container Bean Persistence): o EJBContainer gera automaticamente, por exemplo, o código de acesso ao banco de dados. Este modo de implementação pode permitir um aumento da produtividadeporém, o programador deve estar atento ao número de requisições que o EJBContainer faz ao banco de dados, pois é comum realizar requisições que não são necessárias.

Vários clientes podem acessar um Bean persistente simultaneamente, porém todas as operações de escrita devem ter o suporte de transação garantindo a integridade.

Resumindo, no servidor EJB, o controle das requisições do cliente é realizado pela implementação da interface EJBObject, que intercepta as requisições do cliente e chama os métodos corretos nas instâncias dos Beans específicos. O acesso as instâncias dos Beans é feito através da implementação da interface EJBHome, que representa uma interface para cada Bean residente no EJBContainer. Um detalhe importante é que a EJBHome de todo Bean tem que satisfazer as regras impostas pela interface Remote do pacote RMI (Remote Method Invocation) (RMI,2002), o que possibilita que qualquer Bean possa ser acessado por qualquer cliente. Na figura 3.4, localizada na próxima página, é apresentado um diagrama de seqüência que demonstra o funcionamento dos componentes da arquitetura EJB.

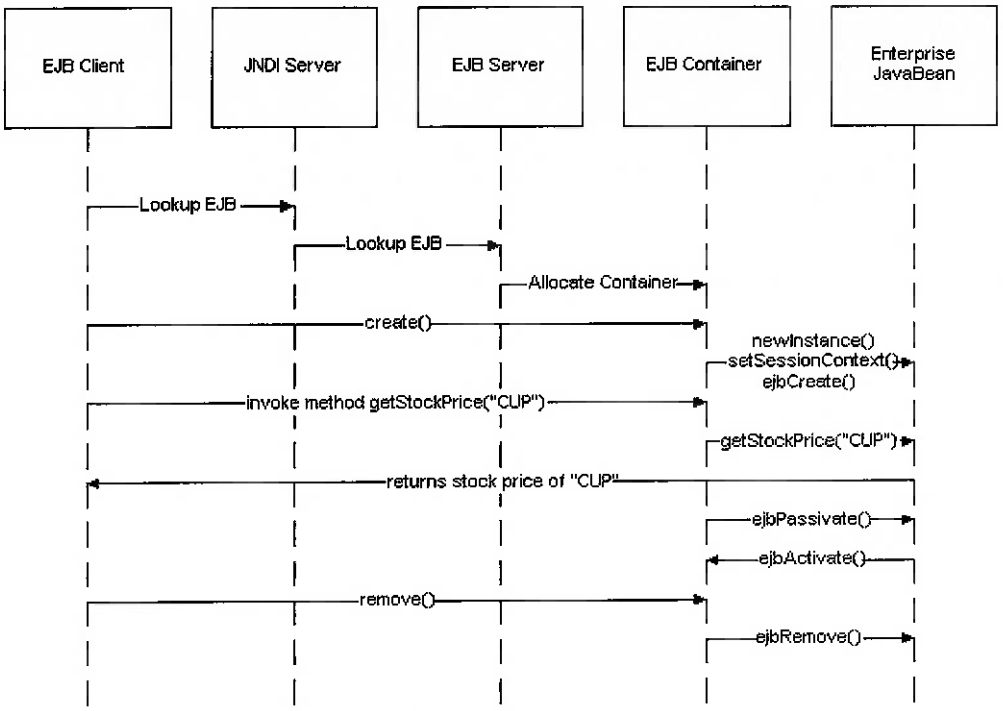

Figura. 3.4 – Diagrama de Seqüência do funcionamento dos componentes EJB

Para melhor entendimento do diagrama, este é descrito passo a passo:

1º passo: o Cliente EJB solicita serviço para o Servidor JNDI;

2º passo: o Servidor de JNDI faz o mapeamento do Bean responsável pelo serviço solicitado pelo cliente;

3º passo: o Servidor EJB referencia o Container, onde se encontra o Bean para o Servidor de JNDI;

4º passo: o cliente inicia uma nova sessão invocando o método create();

5º passo: no Container uma nova instância do Bean é criada, e a partir desta, é invocado o método newInstance(). Em paralelo, é realizada a invocação do método setSessionContext(), que inclui informações sobre o recipiente, o ambiente e a identidade do cliente que realizou a chamada. Finalizando, é feita a invocação do método ejbCreate(), utilizando os parâmetros da entrada emitidos pelo cliente, concluindo a requisição;

6º passo: o Cliente invoca o método getStockPrice("CPU") ;

7º passo: a interface EJBObject invoca o método getStockPrice("CPU") no Bean;

8º passo: o Bean realiza o processamento e retorna a requisição;

9º passo: no Container o EJBObject invoca o método ejbPassivate() para o Bean, esse método tem como implementação a liberação do Bean para novas requisições;

10º passo: o Bean invoca o método ejbActivate() na EJBObject, esse método é responsável por emitir uma sinalização para o EJBObject, informando que o Bean esta a disposição de novas requisições;

11º passo: o Cliente invoca o método remove() no Container, destruindo a instância do Bean que foi alocado com o comando create();

12º passo: a EJBObject invoca o método ejbRemove() para o Bean, destruindo a instância do Bean.

O RMI é um conjunto de regras de comunicação, implementadas em um pacote de classes (java.rmi.*),que facilitam o desenvolvimento de aplicativos distribuídos em linguagem Java (J2EE, 2002). O desenvolvimento de aplicações distribuídas utilizandoRMI é mais simples que o desenvolvimento através do padrão emissor/receptor dos dispositivos de rede, conhecido usualmente como Socket, uma vez que não é necessário projetar um protocolo de comunicação entre o emissor e o receptor.

A seguir, é apresentada uma forma de se construir uma aplicação utilizando RMI. Neste exemplo, o aplicativo permite um cliente transferir qualquer tipo de arquivo texto de uma máquina cliente para um servidor. Os passos desta construção são apresentados a seguir:

1º passo: definir a interface remota, especificando os métodos fornecidos pelo servidor e acessados pelo cliente remoto.

```
public interface FileInterface extends Remote
{
 public byte[] downloadFile(String fileName)
```

```
    throws RemoteException;
}
```

A interface FileInterface.java define um único método, o downloadFile, que recebe uma String como argumento (o nome do arquivo no servidor remoto) e retorna os dados contidos neste arquivo em um Array de Bytes.

As características da interface FileInterface são:

- Deve ser declarada como pública para que os clientes sejam capazes de carregar objetos remotos que a implementam;
- Deve entender a interface Remote do pacote java.rmi, que define os métodos de acesso a objetos remoto;
- Os métodos dessa interface devem ser capazes de transmitir uma exceção do tipo java.rmi.RemoteException, possibilitando que o cliente possa tratar e tomar a decisão adequada em função da mensagem da exceção.

2º passo: implementar a interface remota;

Nesse passo, é implementada a interface FileInterface, que extende a classe UnicastRemoteObject, do pacote Java.rmi.*, indicando que a classe FileImpl.java é utilizada para criar um simples e não-replicado objeto remoto.

```
public final class FileImpl extends UnicastRemoteObject
        implements FileInterface
{
  public FileImpl() throws RemoteException
  {
    super();
  }

  public byte[] downloadFile(String fileName)
  {
    try
    {
      File file = new File(fileName);
      byte buffer[] = new byte[(int)file.length()];
      BufferedInputStream input = new
        BufferedInputStream(new FileInputStream(fileName));
```

```
        input.read(buffer,0,buffer.length);
        input.close();

        file = null;
        input = null;

        return buffer ;
      }
      catch(Exception e)
      {
        e.printStackTrace();
        return null;
      }
    }
}
```

3º passo: desenvolver o servidor;

```
public class FileServer
{
   public static void main(String argv[])
   {
     if(System.getSecurityManager() == null)
     {
       System.setSecurityManager(new RMISecurityManager());
     }

     try
     {
       FileInterface fi = new FileImpl("FileServer");
       Naming.rebind("//127.0.0.1/FileServer", fi);
     }
     catch(Exception e)
     {
       e.printStackTrace();
     }
   }
}
```

A classe FileServer.java representa o servidor, a chamada do método rebind ("//127.0.0.1/FileServer",fi), na classe Naming, assume que o registro RMI está sendo executado sobre a porta padrão 1099. Caso seja necessário executar o registro RMI sobre uma porta diferente, não há restrições, desde que a essa esteja disponível, por exemplo, se o registro RMI é executado na porta 4500 a declaração se torna:

```
Naming.rebind ("//127.0.0.1:4500/FileServer",fi)
```

Porém, para que o servidor funcione é necessário:

- Criar uma instância do RMISecurityManager e instala-la;
- Criar uma instância do objeto remoto FileImp, neste exemplo;
- Registrar o objeto criado com o registro RMI.

4º passo: desenvolver o cliente;

O cliente requesita os métodos especificados na interface remota FileInterface. Para isto, o cliente deve primeiro obter uma referência para o objeto remoto do registro RMI, uma vez que esta referência é obtida, o método downloadFile é chamado. A implementação do cliente é apresentada a seguir, na qual o cliente aceita dois argumentos na linha de comando: o primeiro é o nome do arquivo remoto a ser transferido e o segundo é o endereço do computador a partir da qual o arquivo será copiado, ou seja, da máquina onde se encontra o servidor.

```
public class FileClient
{
  public static void main(String argv[])
  {
    if(argv.length != 2)
    {
     System.exit(0);
    }

    try
    {
      String name = "//" + argv[1] + "/FileServer";
      FileInterface fi = (FileInterface) Naming.lookup(name);
      byte[] filedata = fi.downloadFile(argv[0]);
      File file = new File(argv[0]);
      BufferedOutputStream output = new
        BufferedOutputStream(new FileOutputStream(file.getName()));
      output.write(filedata,0,filedata.length);
      output.flush();
      output.close();
    }
    catch(Exception e)
    {
```

```
        e.printStackTrace();
    }
  }
}
```

3.2.3 - Desenvolvimento de componentes para EJB

As características de uma arquitetura estão relacionadas às linguagens de desenvolvimento que podem ser utilizadas na construção de seus componentes. No caso da arquitetura EJB, muitas funcionalidades e padrões são incorporados da linguagem Java, que é utilizada na construção dos Beans. A seguir são apresentadas algumas dessas características.

3.2.3.1 – Linguagem Java

A linguagem Java originou-se como parte de um projeto de pesquisa que visava a criação de um software avançado, para atendera uma extensa variedade de máquinas conectadas através de redes e sistemas distribuídos. O objetivo inicial do projeto era desenvolver um ambiente operacional pequeno, confiável, portável, distribuído e que operasse em tempo real. Inicialmente, a linguagem escolhida foi C++. Porém, com o passar do tempo, as dificuldades encontradas com C++ aumentaram até o ponto em que as restrições só poderiam ser resolvidas criando uma linguagem completamente nova.

A linguagem Java foi criada tendo como base a linguagem C++, foi projetada para atender a vários requisitos desejáveis em uma linguagem de programação, como por exemplo, confiabilidade, em função do gerenciamento de memória que resulta em um ganho de eficiência, e redigibilidade, por eliminar alguns conceitos do C++ que dificultavam a reutilização de código.

A documentação da linguagem Java fornecida pela Sun Microsystems, empresa responsável pela criação e desenvolvimento da linguagem, utiliza as seguintes palavras para defini-la (JAVA,2002):

- Simples: uma característica marcante da linguagem Java é a simplicidade, como conseqüência pode ser utilizada sem um treinamento intenso, ou larga experiência anterior. Desenvolvedores especializados na linguagem C++ tem uma rápida compreensão de Java devido à sua semelhança. Java omite muitos termos poucos usados e operações confusas em C++ que trazem mais complicações que benefícios;
- Orientada a Objeto: os programadores podem reutilizar código, utilizar bibliotecas de terceiros com proteção e encapsulamento, e adicionar funcionalidades às existentes;
- Robusta e Segura: durante o processo de desenvolvimento da linguagem Java, buscou-se a construção de uma linguagem para escrita de programas confiáveis. A linguagem enfatiza a verificação de possíveis erros, em tempo de compilação, e realiza verificação dinâmica, em tempo de execução, eliminando situações que podem gerar erros;
- Arquitetura Neutra: a linguagem é projetada para suportar aplicações distribuídas em diversos ambientes em rede. Em tais ambientes, é possível a execução de aplicações em diferentes tipos de máquinas. Em várias plataformas de hardware, as aplicações podem ser executadas utilizando recursos de diferentes sistemas operacionais e operar interagindo com outras linguagens de programação. Isto é possível devido a arquitetura concebida, onde a linguagem gera um código binário, utilizando uma JVM (Java Virtual Machine), ou seja, um sistema operacional neutro e portável, que não requer alterações;
- Portável: como o tamanho dos tipos numéricos é fixado, são eliminadas algumas restrições, porque os dados binários são armazenados de acordo com a definição de seu tipo;
- Interpretada: o interpretador da linguagem pode executar o código binário Java, diretamente, em qualquer sistema operacional para o qual exista uma JVM compatível.

A plataforma em linguagem Java utiliza um conjunto de pacotes de classes básicas tal como, controle de entrada e saída de dados, gráficos, conexões e etc. Outro ponto

importante, dentro do contexto proposto para facilitar o desenvolvimento de sistemas, é o recurso de controle de memória proporcionado pela JVM, conhecida no contexto da plataforma J2EE como JRE(Java Runtime Enviroment). A partir do momento que o sistema é executado, o GC (Garbage Collector) da JVM, assume todo o controle da memória usada pelo sistema, e assim, o desenvolvedor não precisa despender tempo implementando-o.

O GC é um mecanismo seguro de liberação e alocação de memória incorporado na JVM. Por exemplo, à medida que uma área de memória é necessária, o GC assume a tarefa de liberá-la. O gerenciamento de memória, quando efetuado diretamente pelo desenvolvedor, torna o programa mais eficiente em termos de desempenho, mas ao mesmo tempo existem restrições, obrigando-o a alocar e liberar memória quando assim é solicitado. Esse tipo de abordagem aumenta o risco de execução do sistema, sendo chamado de código inseguro.

A linguagem Java utiliza uma ferramenta externa chamada Javadoc(JAVADOC,2002), que pode gerar páginas HTML com documentação, a partir de marcações especiais colocadas no código Java. A documentação de código é muito importante no processo de desenvolvimento de software, pois é utilizada como um guia para consulta rápida.

A figura 3.5, localizada na próxima página, apresenta o processo de compilação e execução de uma aplicação Java Stand Alone (isolada).

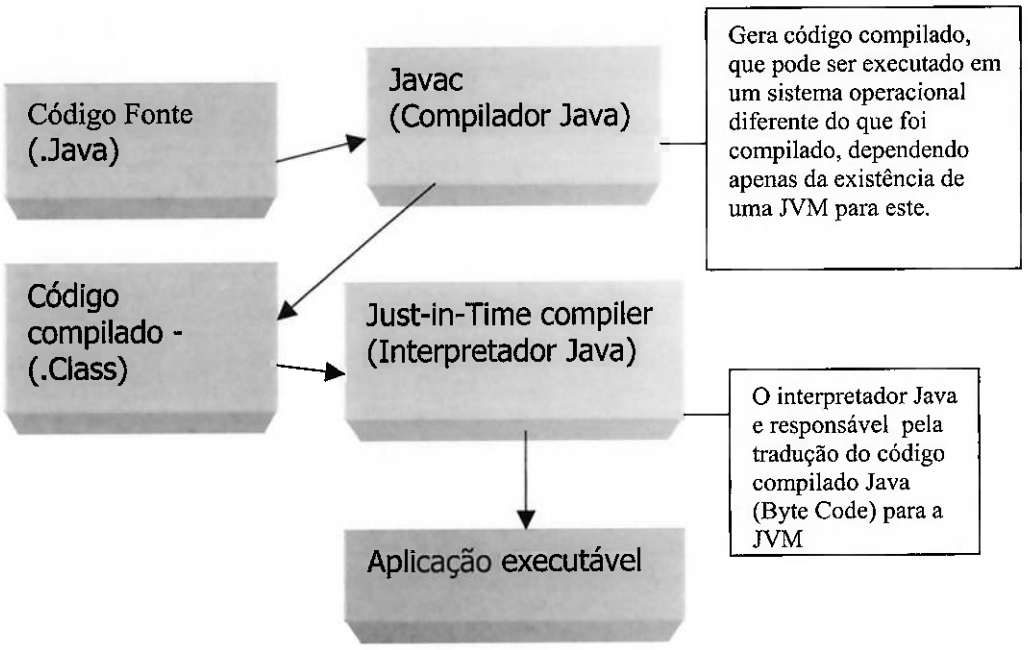

Figura 3.5 – Processo de compilação e execução de aplicações Java Stand Alone

A seguir, são apresentados alguns exemplos de implementações de componentes para arquitetura EJB através da linguagem Java.

1º exemplo: interface Object;

```
public interface Converter extends EJBObject
{
 public double dollarToYen(double dollars)
   throws RemoteException;
 public double yenToEuro(double yen)
   throws RemoteException;
}
```

2º exemplo: interface Home;

```
public interface ConverterHome extends EJBHome
{
 Converter create()
   throws RemoteException, CreateException;
}
```

3º exemplo: classe Bean;

```
public class ConverterEJB implements SessionBean
```

```
{

 public double dollarToYen(double dollars)
 {
  return dollars * 121.6000;
 }
 public ConverterEJB() {}
 public void ejbCreate() {}
 public void ejbRemove() {}
 public void ejbActivate() {}
 public void ejbPassivate() {}
 public void setSessionContext(SessionContext sc) {}
}
```

4º exemplo: classe Cliente;

```
...
 import Converter;
 import ConverterHome;
...
 Context initial = new InitialContext();
 Object objref = initial.lookup("MyConverter");

 ConverterHome home =
  (ConverterHome)PortableRemoteObject.narrow(objref,
  ConverterHome.class);

 Converter currencyConverter = home.create ();
 double amount = currencyConverter.dollarToYen(100.00);
```

A simplicidade das construções, que pode ser observada nos exemplos acima apresentados, onde as implementações de controle, basicamente, estão prontas e através de uma extensão é possível incorporá-las ao código que esta sendo desenvolvido. Desta forma, o desenvolvedor pode concentrar-se na construção das regras de negócio do sistema. Assim, pode-se dizer que a linguagem Java constitui um diferencial importante, em função das vantagens que são proporcionadas durante a construção e nas possíveis manutenções de componentes e serviços.

3.2.3.2 - Web Services

Pode-se definir um Web Service como um software, que conhece a comunicação entre tipos diferentes de software em um ambiente distribuído. Um Web Service deve seguir as seguintes características:

- Descrição: expor e descrever a si mesmo para outras aplicações, possibilitando a compreensão do serviço que executa;
- Localização: pode ser localizado por outras aplicações via um diretório remoto, se o serviço estiver registrado neste diretório;
- Requisição Padronizada: pode ser invocado pelas aplicações utilizando-se de protocolos padrão;
- Resposta Padronizada: deve retornar uma resposta pelo mesmo protocolo sob o qual foi invocado o serviço.

Enfim, deve fornecer facilidades de localização, compreensão, utilização e comunicação. Estas facilidades podem ser implementadas através de APIs especificas da plataforma J2EE e como conseqüência, utilizadas na arquitetura EJB.

A plataforma J2EE possui um conjunto completo de APIs, para o desenvolvimento de Web Services. Esse conjunto permite aos desenvolvedores, colaboradores e vendedores de ferramentas construir aplicações e produtos com segurança, gerência distribuída de transação e gerência de conexão, que são essenciais para esse tipo de serviço(WEBSERVICE,2002).

A API JAXP, por exemplo, permite a construção de aplicações que utilizam XML (Extensible Markup Language) na execução de seus serviços. Todo processamento de XML pode ser realizado através dessa API, agilizando o processo de desenvolvimento dos Web Services.

O XML é uma linguagem de marcação de dados que provê um formato para descrever dados estruturados, facilitando declarações mais precisas do conteúdo e resultados mais significativos de busca através de múltiplas plataformas (XML,2002).

Na figura 3.6 é apresentado um exemplo de interação entre emissor, intermediário e receptor utilizando-se a linguagem XML.

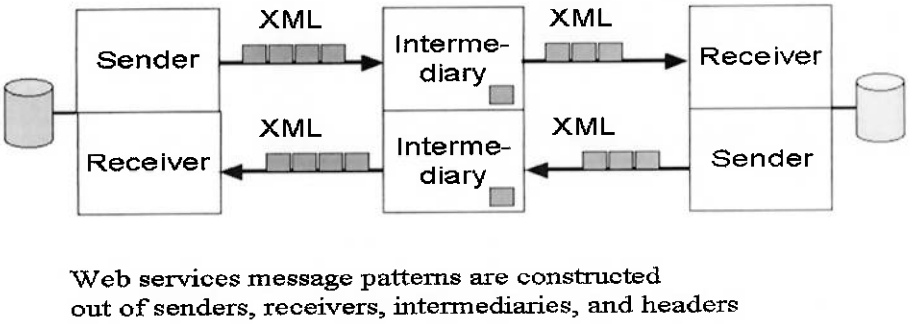

Figura 3.6 - Interação entre emissor, intermediário e receptor através de XML (WEBSERVICES,2002)

Outra API importante é a JAXM, que permite enviar mensagens com SOAP(Simple Object Access Protocol) em ambientes distribuídos.

SOAP representa um protocolo de invocação remota de métodos de objetos baseado em XML (SOAP,2002). O SOAP é a base dos Web Services, é um protocolo que permite que empresas troquem informações em XML de uma forma simples, segura e robusta.

Assim, com base nas APIs da plataforma J2EE, é possível construir serviços para um ambiente distribuído de maneira simples, atendendo os padrões de interoperabilidade com outras plataformas.

4 - ARQUITETURA DA PLATAFORMA .NET

4.1 - Histórico

Em 22 de junho de 2000 a Microsoft anunciou a plataforma .NET (pronuncia-se "dót NET"), geração de softwares e aplicativos para a Internet. Este projeto é considerado pela Microsoft como o mais ambicioso desde o lançamento do sistema operacional Windows, há mais de uma década.

O projeto é de tal importância que Bill Gates afastou-se da presidência da Microsoft, dedicando-se tempo integral a esse projeto, na posição de principal arquiteto de software da Microsoft. Uma significativa parcela dos recursos profissionais, tecnológicos e financeiros da Microsoft estão sendo direcionados para o desenvolvimento dessa plataforma(MICROSOFT,2002).

A .NET é uma plataforma de desenvolvimento para múltiplos ambientes baseados em XML. Essa plataforma tem como fundamento permitir o desenvolvimento de diferentes tipos de aplicação, permitindo a utilização de uma variedade de linguagens de programação, além de integrar componentes desenvolvidos em linguagens diferentes.

A seguir, as figuras exemplificam os componentes da plataforma .NET e a hierarquia entre eles.

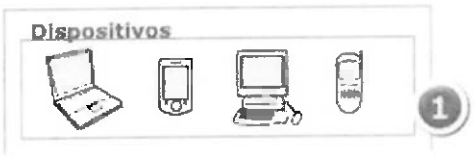

Figura 4.1 - Dispositivos (NET,2002).

No topo da hierarquia, encontram-se os múltiplos dispositivos que acessam as informações de dispositivos clientes, tais como microcomputadores e telefones celulares inteligentes.

.

Figura 4.2 - Web Services (NET,2002).

No segundo nível da hierarquia, figura 4.2, encontram-se os Web Services, software desenvolvidos pela Microsoft.

Figura 4.3 – Passport / Segurança (NET,2002).

No terceiro nível da hierarquia, figura 4.3, encontram-se os serviços de segurança e o serviço de autenticação de usuários, de acordo com o padrão da Microsoft, o Passport.

Figura 4.4 – Web Services Enabler (NET,2002).

No quarto nível da hierarquia, figura 4.4, encontram-se os serviços específicos para aplicações corporativas, desenvolvidos para atender requisitos de cobrança, de acordo com as necessidades do mercado, de forma a permitir maior flexibilidade aos sistemas.

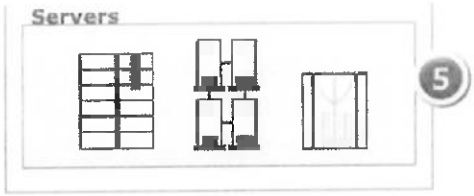

Figura 4.5 – Servers (NET,2002).

No quinto nível da hierarquia, figura 4.5, encontram-se diversos softwares que formam a infra-estrutura básica de suporte aos Web Services e interligam a plataforma .NET com o sistema legado das empresas.

Em conjunto com a plataforma .NET, a Microsoft criou a linguagem C# (pronuncia-se "C Sharp"), tornando-se uma linguagem de referência e um diferencial para a plataforma. Embora exista a possibilidade de utilização de diferentes linguagens através da .NET,deve-se considerar a adoção do C# como linguagem principal de desenvolvimento de um sistema, devido as seguintes razões:

- A linguagem foi criada especificamente para a plataforma .NET;
- O compilador C# foi desenvolvido para dar suporte a linguagem;
- A maior parte das classes do .NET Framework foram desenvolvidas em C#.

A seguir, são apresentadas as características da plataforma .NET, descrevendo sua arquitetura.

4.2 – Arquitetura .NET

Neste item, a arquitetura da plataforma .NET é descrita através dos componentes que a compõe e em seguida é apresentada como uma aplicação .NET é executada. Os componentes da arquitetura .NET são apresentados na figura 4.6.

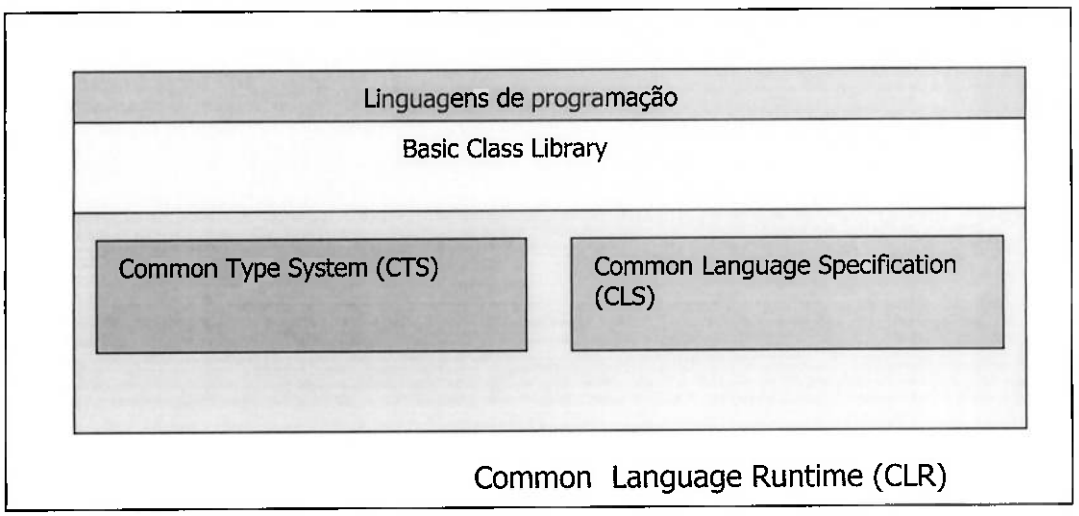

Figura 4.6 - Arquitetura da plataforma .NET (LIMA, E.; REIS,E., 2002).

O CLR (Common Language Runtime) representa o ambiente de execução das aplicações .NET. As aplicações .NET não são aplicações Win32 propriamente ditas, apesar de executarem no sistema operacional Windows, . O Win32, ao identificar uma aplicação .NET ativa o runtime .NET que, a partir desse momento, assume o controle total da aplicação, sendo compartilhado e, portanto, não existe um runtime para cada uma das linguagens, ele é comum a todas as linguagens suportadas pela plataforma .NET.

No CLR há o CTS (Common Type System), responsável pela definição e características dos tipos suportados pela .NET. As linguagens que podem ser usadas na .NET , necessariamente, suportam esses tipos. Apesar da especificação não demandar que todos os tipos definidos no CTS sejam suportados pela linguagem, esses tipos podem ser um subconjunto do CTS, ou ainda um superconjunto. Por exemplo: um tipo Enum é derivado da classe System.Enum e todas as linguagens implementam o tipo Enum da mesma forma. Na arquitetura proposta na plataforma

.NET, todo tipo deriva da classe Object, e com isso, os diversos tipos nas diferentes linguagens são implementados, obedecendo às regras definidas no CTS.

As regras do CTS são definidas pela CLS (Common Language Specification), que constitui um subconjunto do CTS, e qualquer linguagem que a implemente na .NET, devem atender estas regras, de modo que, o código gerado resultante da compilação na referida linguagem seja perfeitamente entendido pelo runtime .NET, isto é um imperativo porque, caso contrário, as vantagens da plataforma .NET, requisitos como independência da linguagem de programação e interoperabilidade ficam comprometidos.

Assim, dizer que uma linguagem é compatível com o CLS significa dizer que, mesmo quando esta é sintaticamente diferente de qualquer outra linguagem que implemente a .NET, semanticamente ela é igual, porque durante a compilação é gerado um código intermediário, denominado IL (Intermediate Language), equivalente as duas partes iguais de códigos, porém escritas em linguagens diferentes.

A plataforma .NET, como a plataforma Java, também tem como um dos objetivos, simplificar o desenvolvimento de sistemas, e para tanto possui um conjunto de classes básicas denominado BLC (Basic Class Library). Na BLC encontram-se bibliotecas de controle de entrada e saída de dados, gráficos, sockets, gerenciamento de memória (CHAPPELL, DAVID, 2002). Essas bibliotecas de classes estão organizadas hierarquicamente em uma estrutura denominada "namespace".

Considerando os componentes descritos acima, pode-se observar que a plataforma .NET, apesar de inicialmente estar sendo concebida para o ambiente Windows, possui uma arquitetura portável tanto em termos de linguagem de programação quanto no nível da arquitetura do processador, visto que o código gerado pode ser interpretado para uma linguagem assembly da plataforma, em tempo de execução, sem a necessidade de recompilação do código-fonte.

Para exemplificar, é apresentado um trecho código tanto em VB.NET quanto em C#.

- Trecho de código em VB.NET

```
Import System
Public Module AppNet
        Sub Main()
                Console.Writeline("VB.NET")
        End Sub
End Module
```

- Trecho de código em C#.

```
Using system;

Public class AppNet
{
        public class static void  Main()
        {
                Console.Writeline("C#");
        }
}
```

Os dois trechos de código acima, apesar de serem semanticamente diferentes, quando traduzidos para IL têm como resultado o mesmo código intermediário (LIMA, E.; REIS,E, 2002).

Para um entendimento mais adequado de como uma aplicação .NET é executada, é preciso o esclarecimento de alguns conceitos:

- Tempo de compilação: entende-se por tempo de compilação a parte do processo de compilação que diz respeito a geração de código IL e de informações específicas da aplicação necessárias para sua correta execução;

- Metadados: são conjuntos de instruções geradas no processo de compilação de uma aplicação .NET, junto com IL, que contém as seguintes informações específicas do sistema:
    - A descrição dos tipos, usados na aplicação, podendo esta ter sido gerada em forma de executável;
    - A descrição dos membros de cada tipo (propriedades, métodos, eventos).
    - A descrição de cada unidade de código externo usado na aplicação e que é requerida, para que esta execute adequadamente;
    - Resolução da chamada de métodos;
    - Resolução de versões diferentes de uma aplicação.

Pode-se dizer que uma aplicação .NET é auto-explicativa, dispensando a utilização do registro do sistema operacional Windows, para armazenamento de informações adicionais a seu respeito. O CLR vai pesquisar nos Metadados a versão correta da aplicação a ser executada. Esta é uma característica adicional importante no que diz respeito à implementação e manutenção de sistemas em produção;

- Assembly: toda aplicação .NET, quando compilada, é armazenada fisicamente em uma unidade de código denominada Assembly. Uma aplicação pode ser composta de um ou mais Assemblies, os quais são representados nos sistemas de arquivos em forma de executável, ou de biblioteca de ligação dinâmica conhecida como DLL (Dynamic Link Library);
- PE (Portable Executable): quando um aplicativo é compilado, são geradas instruções em IL, os Metadados com informações do sistema também são gerados e armazenados. Diz-se portável porque ele pode ser executado em qualquer plataforma que suporte a .NET, sem a necessidade de recompilação, operação que é efetuada automaticamente pelo runtime, quando da execução do sistema;
- Compilação JIT (Just in time): um compilador JIT, também conhecido como JITTER, converte instruções IL para instruções específicas das

arquitetura do processador onde a aplicação .NET está sendo executada. Na plataforma .NET existem três diferentes tipos de JITTER:

1. Pré – JIT: compila de uma só vez todo código da aplicação .NET que está sendo executada e o armazena no cachê para uso posterior;
2. Econo – JIT: é usado em dispositivos onde o limite de memória é baixo. Sendo assim, o código é compilado sob demanda, e a memória alocada, que não esta em uso, é liberada;
3. Normal- JIT: compila o código sob demanda e coloca o código resultante no cachê, de forma que, esse código não precise ser recompilado quando houver uma nova invocação do mesmo método.

- VES (Virtual Execution System): o processo de compilação acontece no VES, e é aqui onde o JITTER é ativado quando uma aplicação .NET é chamada. O JITTER é ativado a partir do runtime do sistema operacional Win32, que transfere o controle para o runtime .NET, após isso, a compilação do PE é efetuada e só então o código assembly, próprio da arquitetura do processador, é gerado para que a aplicação possa ser executada.

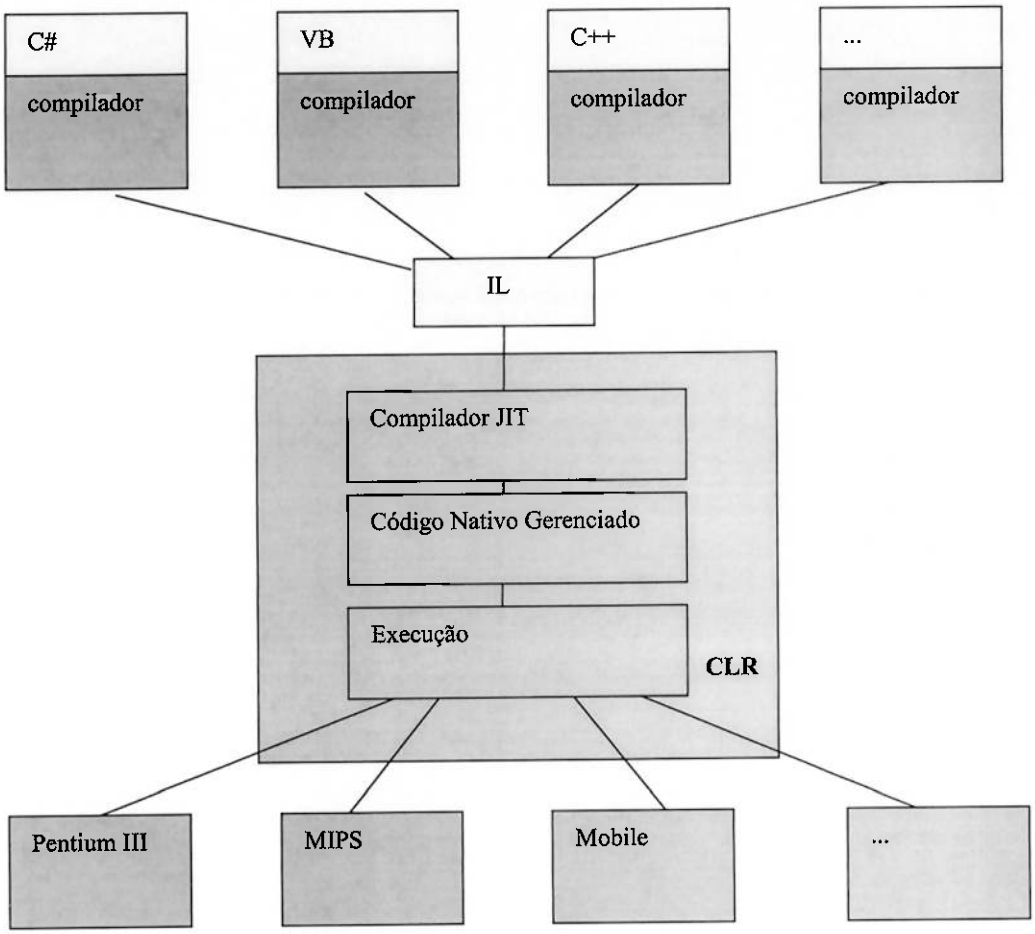

Figura 4.7 – Processo de execução de uma aplicação .NET(LIMA, E.; REIS,E,2002).

A figura 4.7 ilustra o processo de execução de uma aplicação .NET, desde a geração de instruções em IL em tempo de compilação, até a geração do código assembly específico da plataforma de execução.

O gerenciamento de memória é efetuado pelo runtime .NET, permitindo que o desenvolvedor se concentre na resolução das regras de negócio. Á medida que uma área de memória é necessária para alocar um objeto, o GC (como na plataforma Java) assume todo o controle de memória que o sistema venha a precisar.

A plataforma .NET contém muitos recursos que visam não só impedir diversos erros de programação, como também viabilizar a implantação de um ambiente gerenciador. Estes recursos são conhecidos como .Net Managed Components. Neste ambiente o código não pode danificar o sistema de execução nem causar perda de

dados, isto é especialmente importante na Internet, tanto do ponto de vista de um usuário que transfere um aplicativo da rede de algum local não confiável, como do lado dos servidores, que não podem parar.

A base deste esquema é a seguinte:

- O sistema de tipos não pode ser violado, todo executável contém informações dos tipos usados e implementados e estas informações permitem validações em tempo de execução.
- As permissões de acesso a um determinado recurso, tal como arquivos, conexões TCP são validadas em tempo de execução em função de um conjunto de atributos, tais como, permissões do usuário, permissões do executável, sua origem e assinaturas digitais. Este sofisticado sistema de segurança recebe o nome de Evidence Based Security.

No item a seguir, apresenta-se o desenvolvimento de componentes para aplicações .NET, que pode diferir em diversos tipos: aplicações clientes; aplicações gráficas; aplicações para Internet; aplicações para banco de dados; aplicações multitarefa.

4.3 - Desenvolvimento de aplicações para a .NET

Na plataforma .NET existe a flexibilidade de desenvolvimento em diversas linguagens, que são aderentes às especificações CLS da CTS, para que possam ser compatíveis. Dentre as linguagens que suportam a .NET podem ser citadas: C# , C++, Perl, Pascal, Cobol, Phyton, Visual Basic, Small Talk (LIMA, E.; REIS,E,2002). Para a linguagem Java, vêm sendo desenvolvido pela Microsoft uma versão suportada pela plataforma .NET, e que irá se chamar J#(leia-se J-sharp).

A seguir, é apresentado o desenvolvimento de aplicações .NET através da linguagem C#, como citado anteriormente, construída especialmente para a plataforma .NET.

4.3.1 - Linguagem C#

As características principais da linguagem C# são:

- Simplicidade: os desenvolvedores de C# costumam dizer que essa linguagem é tão poderosa quando o C++ e tão simples quando o Visual Basic;
- Orientada a Objetos: em C#, qualquer variável é parte de uma classe;
- Definição de Tipos: isso ajuda a evitar erros de manipulação imprópria de tipos, atribuições incorretas e etc;
- Controle de Versões: cada assembly gerado tem informações sobre a versão de código, permitindo a coexistência de dois assemblies homônimos, mas de versões diferentes no mesmo ambiente.
- Flexibilidade: se o desenvolvedor precisar usar ponteiros, C# permite, mas ao custo de desenvolver código não-gerenciado.

A seguir, é apresentado um exemplo de parte de um código em linguagem C#.

```
using System;

class Hello
{

 public static void Main()
 {
   Console.WriteLine("Hello World!!!");
 }

}
```

Algumas considerações pode ser feitas em relação ao exemplo acima:

- A claúsula using referencia as classes a serem utilizadas;
- O "namespace" System contém muitas classes, uma delas é a classe denominada Console;
- O método WriteLine, simplesmente emite o texto no console.

Na plataforma .NET, o uso do "namespace" tem como função estruturar logicamente as aplicações. Fisicamente, é através dos assemblies que o código IL e as informações de como os metadados gerados durante o processo de compilação são armazenados no sistema de arquivos. A figura 4.8, exemplifica a estrutura de uma aplicação .NET.

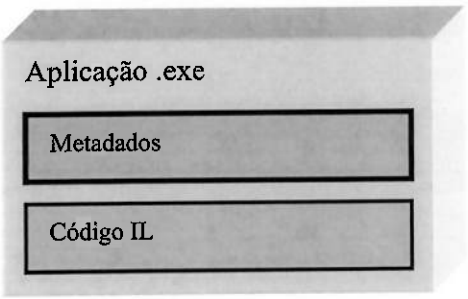

Figura 4.8 – Estrutura básica de uma aplicação .NET

Em C#, um assembly é auto-explicativo e não requer que informações adicionais sejam armazenas no registro do sistema.

O C# agrega o VS.NET, possibilitando documentar o código gerado utilizando a linguagem XML. A linguagem XML facilita a geração automática de documentação, a partir de comentários inseridos no código-fonte.

O VS.NET é um ambiente de desenvolvimento da mesma família das versões do Visual Studio da Microsoft, mas ele é completamente integrado as linguagens compatíveis com a plataforma.

4.3.2 - Web Services

Para o desenvolvedor, um Web Service, no contexto da plataforma .NET pode ser visualizado como uma página ASP.NET, que mapeia automaticamente pedidos via Internet a métodos de uma linguagem de alto-nível.

O ASP.NET constitui uma linguagem de desenvolvimento para aplicações .NET que tem foco de execução na Internet. Possui um sofisticado ambiente de desenvolvimento e de execução, que traduz o modelo de programação de uma forma transparente ao desenvolvedor, fazendo com que este não precise conhecer as variáveis do servidor.

O SOAP representa um papel importante na plataforma .NET e tem um extenso suporte no Visual Studio.NET. Um Web Service no contexto da plataforma .NET é representado por um conjunto de WebMethods. Os WebMethods são funções chamadas remotamente através de SOAP.

Na plataforma .NET os Web Services são implementados sempre em uma classe derivada de System.Web.Services.WebService. Nesta classe derivada, podem ser adicionadas funções que são chamadas via SOAP. A diferença entre um WebMethod e um método comum é a presença de um "atributo WebMethod", uma espécie de diretiva de compilação.

A seguir, é apresentado um exemplo, de parte de um código em linguagem C#, definindo a leitura de um arquivo XML.

```
XMLReader xml = null;

Private void loadXML()
{
        Try
        {
```

```
            xml = new XMLReader(fileName);
            while(xmlRead())
                switch(xmx.name)
                {
                    case "txtName":
                    txtName.Text = xml.ReadString();
                    break;
                    default:
                    break;
                }
            }
        }
        catch(Exception ex)
        {
            MessageBox.Show("Error - load file " + ex.toString();
        }
        finally
        {
            if(xml!=null) xml.close();
        }
}
```

A classe XML usada no código acima esta contida no namespace System.XML. Como na plataforma J2EE, a plataforma .NET contem um conjunto completo de classes (namespace) que podem ser utilizadas na construção de Web Services.

5 – ANÁLISE COMPARATIVA ENTRE A ARQUITETURA EJB DA PLATAFORMA E ARQUITETURA DA PLATAFORMA .NET

A arquitetura EJB da plataforma J2EE e a arquitetura da plataforma .NET apresentam características semelhantes para o gerenciamento de sistemas em ambientes distribuídos, quando analisadas superficialmente, e a princípio, um desenvolvedor pode ser capaz de construir sistemas similares utilizando uma ou outra plataforma. Porém, existem diferenças fundamentais na seleção de uma dessas arquiteturas, conseqüentemente também, para plataformas que as adotam.

As duas arquiteturas utilizam o mesmo conceito de execução de sistemas, onde um código primário, desenvolvido através de linguagens padrões ou suportadas pelas plataformas, é compilado para um código intermediário (IL na .NET e Byte Code na EJB) e executado em um sistema operacional virtual, o CLR na plataforma .NET e JRE na plataforma J2EE.

A JRE é utilizada para executar sistemas, desenvolvidos na linguagem Java, em vários ambientes operacionais. Essa portabilidade permite que um sistema desenvolvido, originalmente para um sistema operacional, possa ser executado sem alterações em outros sistemas operacionais.

Por outro lado, o CLR mapeia diversas linguagens de desenvolvimento, possibilitando maior flexibilidade na escolha da linguagem de programação porém, restrita a uma única plataforma utilizando o sistema operacional Windows. O suporte oferecido a diversas linguagens impõe algumas restrições, obrigando essas linguagens a seguir algumas definições impostas pelo CTS.

Em relação à infra-estrutura para o gerenciamento componentes, na arquitetura EJB existem mais recursos do que os oferecidos pelo Net Managed Components, para os componentes .NET,.como por exemplo, acesso automático de objetos no banco de dados. As linguagens de desenvolvimento para construção de componentes, nas duas

plataformas, possuem diversos pacotes ou bibliotecas que podem ser utilizadas, inclusive para construção de Web Services baseados em SOAP e XML.

A linguagem C# representa a linguagem de referência para plataforma .NET, tal como a linguagem Java para a plataforma .NET. A linguagem C# possui mais recursos do que a linguagem Java, em especial os recursos para manipulação de ponteiros. Porém, esses recursos devem ser utilizados com critério, pois não são controlados pelo Net Managed Components. Os códigos gerados com esses recursos são conhecidos como códigos inseguros (LIMA, E.; REIS,E,2002).

A linguagem ASP.NET da plataforma .NET , que possui um avançado ambiente de desenvolvimento e de execução para sistemas no ambiente da Internet, pode ser citada como um diferencial, pois possui vários recursos que visam facilitar o desenvolvimento. Em contrapartida, a arquitetura EJB possui recursos como Beans persistentes, usados para controlar transações, seja com um banco de dados ou com outros Beans, que também visam facilitar o desenvolvimento diminuindo a complexidade de construção dos sistemas.

Para integração de sistemas em ambientes distribuídos, nas duas arquiteturas, pode-se utilizar o padrão CORBA, garantindo-se que as necessidades de interoperabilidade dos sistemas sejam supridas.

A escalabilidade oferecida pelos servidores de aplicação da arquitetura EJB, representa outra característica muito importante para expansão de sistemas. Porém, na plataforma .NET, para os sistemas críticos existe a restrição de serem executados somente sobre a plataforma Windows.

Em relação aos padrões de plataforma, apenas a linguagem C# foi padronizada na plataforma .NET, o restante da plataforma permanece como produto proprietário da plataforma Windows. Em contrapartida, a arquitetura EJB incorpora os padrões da plataforma J2EE, ou seja, aberto (JAVA,2002)

A plataformas J2EE e .NET não possuem compatibilidade nem mesmo de código fonte, está sendo aguardado o lançamento do J#. Portanto, um mesmo programa não pode ser testado em ambas arquiteturas.

A Microsoft reescreveu a aplicação JPS(Java Pet Store) (JPS,2002), tutorial da plataforma J2EE para plataforma .NET, mostrando que o desempenho da arquitetura chega ser 28 vezes superior ao da arquitetura EJB. Esse porte foi criticado por ser muito diferente do original, não preservando os princípios básicos das arquiteturas. O JPS é escrito em três camadas, enquanto a versão da plataforma .NET utiliza inclusive, stored procedures.

A EJB tem presença marcante no mercado corporativo, em servidores de aplicação. Em compensação, a Microsoft domina o mercado dos computadores pessoais, esse mercado pode estar relativamente saturado, mas ainda é muito lucrativo e pode ser usado para alavancar iniciativas na plataforma .NET.

A tabela 5.1, apresenta um quadro comparativo entre as características das plataformas J2EE e .NET.

Característica	J2EE	.NET
Tipo de Tecnologia	Aberta e padronizada	Proprietária e semi - padronizada
Sistema operacional virtual (Interpretador)	JRE	CLR
Linguagem Padrão	Java	C#
Portabilidade	Sim	Não
Escalabilidade	Sim	Não
Interoperabilidade	Padrão CORBA	Padrão CORBA
Suporte a várias linguagens de desenvolvimento	Não	Sim
Gerenciamento de componetes	EJB	Net Managed Components

Tabela 5.1 - Quadro comparativo entre as plataforma J2EE e .NET

A princípio, ambas arquiteturas irão coexistir e ter boas representações no mercado, possibilitando a escolha da solução mais adequada para cada corporação.

6 - CONCLUSÃO

A escolha de uma arquitetura adequada é um desafio para um arquiteto de sistemas, cada uma tem seus pontos fortes e fracos. Portanto, vários aspectos devem ser considerados, como por exemplo, requisitos de escalabilidade e desempenho, utilização de padrões, maturidade da arquitetura, tendências do mercado, entre outros. Assim sendo, para uma tomada de decisão, é necessária a definição de estratégias, em relação ao desenvolvimento de sistema, a médio e em longo prazo, levando em conta o perfil da equipe e o legado existente na corporação.

Considerando o momento atual e baseando-se nas características analisadas nesse trabalho, pode-se afirmar que a arquitetura EJB representa a melhor solução para gerenciamento de objetos em sistemas distribuídos. A portabilidade e escalabilidade podem ser citadas como o maior diferencial da arquitetura EJB, pois são requisitos indispensáveis neste tipo de ambiente.

Em relação ao gerenciamento de objetos, pode-se dizer, que os servidores EJB são tecnologicamente mais avançados que os servidores .NET. Além disso, ela segue um padrão, que garante a compatibilidade de diferentes versões do produto, podendo ser portável para diferentes plataformas e possui um grupo de fornecedores a altura da própria Microsoft, que no caso da plataforma .NET representa um único fornecedor de servidores.

Por outro lado, dependendo da perspectiva e da estratégia adotada na empresa, a .NET pode ser considerada como um ambiente de uma próxima geração. A linguagem de programação C #, a adoção do XML e SOAP apresentam uma estrutura nova de intercâmbio de dados. A arquitetura .NET poderá vir a dominar o mercado, devido ao grande número de usuário de produtos da Microsoft. Porém, para que isso aconteça, a evolução dos servidores da plataforma .NET é essencial. Além disso, no mundo Microsoft, cada nova versão de um produto, geralmente exige alterações em todos os sistemas.

As especificações atuais do EJB possuem limitações em relação ao controle e acesso a objetos, não oferecendo suporte à segurança necessária as regras de negócio de uma determinada aplicação, que normalmente são codificadas diretamente nos métodos de negócio do EJB. O JBoss (JBOSS, 2002) possui uma arquitetura específica para assegurar que as implementações relativas à segurança de regras de negócio sejam separadas em objetos distintos.

Por adotar um padrão aberto, o JBoss depende extensivamente de terceiros para o desenvolvimento de ferramentas. Contudo, com a tendência crescente de adeptos à comunidade de desenvolvimento de soluções para padrões abertos, isto possibilita uma ascensão rumo ao desenvolvimento de tais ferramentas. Este trabalho pode servir de base para futuros pesquisas e um estudo aprofundado sobre os servidores de aplicação da arquitetura EJB, em especial o JBoss, em função dos seus princípios.

REFERÊNCIA BIBLIOGRAFICA

COAD, P.; YOURDON, E. **Object-Oriented Analysis**. Ed. Prentice Hall, 1991.

PINTO, A. P.; Depoimento extraído da obra **Microsft. NET - Benefícios e Oportunidades**. São Paulo, v.1, p.10, 2001.

CORBA; **CORBA 2.6.1 Specification**; s.l., v. 2.6.1, 2002.

OMA; **Object Management Group, Inc**. Disponível em: http://cgi.omg.org/oma/. Acesso em 10, jul. 2002.

HARKEY, D. ; ORFALI, R., **Client/Server Programming With Java and CORBA,** Ed. John Wiley & Sons, 1997.

MOWBRAY, T.J.; ZAHAVI R.; **The Essential CORBA**, John Wiley & Sons and, 1995.

SOARES, L.F.G; LEMOS, G; COCHER, S.; **Redes de Computadores**. Ed. Campus, 1995.

MATOS, J. P.; Figura extraída do material do módulo **Aplicações Cliente Servidor**, do MBA em Tecnologia da Informação da Faculdade de Economia e Administração da Universidade de São Paulo, 2002

HAEFEL, R. M, **Enterpise JavaBeans,** Ed. O´Reilly, 1998.

EJB; **Enterprise JavaBeans Technology**. Disponível em: http://java.sun.com/ejb/index.htm. Acesso em 2, dez. 2001.

JAVA; **Essential Java Classes** .Disponível em: http://java.sun.com/docs/books/tutorial/essential/index.html. Acesso 23, jan 2002

XML; **Extensible Markup Language**.Disponível em http://www.w3.org/XML/. Acesso em 14, set. 2002

SOAP; **SOAP 1.2 spec**. . Disponível em: http://www.w3.org/2000/xp/Group/. Acesso em 02, nov. 2002

WEBSERVICES; **Web Services Activity**. Disponível em: http://www.w3.org/2002/ws/. Acesso em 10, nov. 2002.

MICROSOFT; **Microsft. NET - Benefícios e oportunidades**. S.n. 2002

NET; **Componentes da plataforma .NET**. Disponível em <http://www.microsoft.com/brasil/net/componentes/default.asp>.
Acesso em 15, de set. de 2002.

J2EE; **Documentation**. Disponível em: http://developer.java.sun.com/developer/infodocs/. Acesso em 10, set. 2002.

BOOCH, G; RUMBAUCH, J; JACOBSON, I. **The Unified Modelling Language User Guide**. Ed. Addilson Wesley, 1999

LIMA, E.; REIS,E; **C# e .NET – Guia do Desenvolvedor**. Ed. Campus, 2002

CHAPPELL, DAVID; **Undertanding .NET – Tutorial and Analysis**. Ed. Addilson-Weslley,2002.
JPS; **Java Pet Store**.Disponível em: 2002; Acesso em 10, set. 2002.

APPLETS; **Applets**. Disponível em: http://java.sun.com/applets/. Acesso em 19, out. 2002

RMI;Java **Remote Method Invocation**. Disponível em http://java.sun.com/products/jdk/rmi/index.html. Acesso em 10, ago. 2002

SERVLET; **Servlet**. Disponível em: http://java.sun.com/products/servlet/index.html. Acesso em 10, set. 2002.

JAVADOC;**Javadoc Tool Home Page.** Disponível em: http://java.sun.com/j2se/javadoc/index.html. Acesso em 20, ago. 2002.