UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANIBAL MANTOVANI DINIZ

CONTROLE DE RÉPLICAS UTILIZANDO A PLATAFORMA JAVA 2 ENTERPRISE EDITION

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Prof. Rogério Cid Bastos, Dr.

Florianópolis, março de 2003

CONTROLE DE RÉPLICAS UTILIZANDO A PLATAFORMA JAVA 2 ENTERPRISE EDITION

ANIBAL MANTOVANI DINIZ

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação na Área de Concentração de Sistemas de Conhecimento e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Estado de Santa Catarina.

_____

Coordenador: Prof. Fernando Alvaro Ostuni Gauthier, Dr.

Banca Examinadora

_____

Prof. Rogério Cid Bastos, Dr.

_____

Prof. Raul Sidnei Wazlawick, Dr.

_____

Prof. Álvaro G. Rojas Lezana, Dr.

Este trabalho dedico para:
Arlete, Amanda, Alan, Assis, Alaíde
Olívia, Arnaldo, Antonio, Alessandro e Janete
que colaboraram com a realização
deste, incentivando-me.

SUMÁRIO

ÍNDICE DE FIGURAS

RESUMO

DINIZ, A. M. **Controle de Réplicas Utilizando a Plataforma Java 2 Enterprise Edition**. Florianópolis, 2002. 65p. Dissertação (mestrado) – Universidade Federal de Santa Catarina.

Um dos desafios para os sistemas computacionais modernos é a implantação destes em projetos que utilizam as técnicas de distribuição, tanto de dados quanto da execução de processos em máquinas distribuídas geograficamente em localidades diferentes. Para atender estes requisitos explora-se a capacidade de plataformas baseadas em componentes, utilizadas para prover estas características às aplicações corporativas. Da utilização destes componentes em aplicações distribuídas surgem estudos e técnicas para garantir melhor desempenho, transparência, segurança e disponibilidade garantida. Traz um estudo da plataforma J2EE onde, em um ambiente distribuído, quer se garantir a disponibilidade de componentes implantados nos servidores de aplicação através das técnicas de replicação. A abordagem tratada é não adaptar a Máquina Virtual Java e deixar transparente, para a aplicação, o controle dos objetos replicados. Utiliza-se a característica de alterar arquivos descritores dos Enterprise Java Beans - EJB implantados no servidor e acrescentam-se novos componentes responsáveis pelo gerenciamento das réplicas. Para a construção da metaclasse do componente gerenciador de réplicas utiliza-se reflexão computacional. Aplicam-se as técnicas de Replicação Ativa e Passiva em um estudo de caso e mede-se o desempenho destas técnicas.

Palavras chaves: Sistemas Distribuídos, Tolerância a Falhas, Replicação e J2EE.

ABSTRACT

DINIZ, A. M. **Controle de Réplicas Utilizando a Plataforma Java 2 Enterprise Edition**. Florianópolis, 2002. 65p. Dissertação (mestrado) – Universidade Federal de Santa Catarina.

One of the challenges for the modern computational systems is the implantation of these in projects that use the distribution techniques, as much of data how much of the execution of processes in machines distributed geographically in different localities. To take care of these requirements it is explored the capacity of platforms based on components, used to provide these characteristics to the corporative applications. From the use of these components in distributed applications appear studies and techniques to guarantee a better performance, transparency, security and guaranteed availability. It brings a study of platform J2EE where, in a distributed environment, the availability of components implanted in the servers of application through the response techniques wants to guarantee itself. The treated boarding is not to adapt the Virtual Machine Java and to leave transparent, for the application, the control of talked back objects. It is used characteristic to modify describing archives of the Enterprise Java Beans - EJB implanted in the server and adds new components responsible for the management of the rejoinders. For the construction of it metaclasse of the gerenciador component of rejoinders is used computational reflection. The techniques of Active and Passive Response in a case study are applied and measure the performance of these techniques.

Keywords: Distributed Systems, Fault Tolerant, Replication and Java 2 Enterprise Edition.

1 INTRODUÇÃO

A popularidade da Internet abriu espaço para a utilização das redes de computadores e veio alavancar novos negócios ou, pelo menos, fazer negócios antigos de uma nova forma atingindo um mercado consumidor disperso geograficamente e muito maior. Comércio eletrônico, Internet *banking*, entretenimento virtual dentre outros são bons exemplos de novos negócios viabilizados pelo desenvolvimento das redes. Entretanto, o surgimento destas aplicações foi acompanhado por um proporcional incremento na complexidade e heterogeneidade dos sistemas de redes de computadores. Isto levou a comunidade acadêmica de computação a desenvolver e melhorar os sistemas de administração e gerenciamento de redes e aplicações, criando e aplicando técnicas para o desenvolvimento de sistemas distribuídos. Segundo JALOTE (1994), no funcionamento de um sistema distribuído a palavra-chave deve ser transparência, ou seja, a complexidade do sistema não deve transparecer para o usuário. O não cumprimento das atividades do sistema de acordo com suas especificações pode comprometer significativamente a transparência do sistema e levar a erros irrecuperáveis. Então, uma necessidade importante desses sistemas é o desenvolvimento de aplicações tolerantes a falhas. Estas aplicações visam garantir que as propriedades essenciais, ou serviços, sejam preservados mesmo na presença de falhas em alguns componentes físicos do sistema. Fundamentalmente, a tolerância a falhas é obtida por redundância. No caso de mecanismos de tolerância por software, esta redundância significa basicamente replicação de dados ou processos. Assim, a facilidade de disponibilizar bibliotecas de componentes para aplicações especiais incentiva a busca de soluções para problemas recorrentes na forma de componentes genéricos e reusáveis. O fato de encapsular estes serviços faz com que os projetistas ou programadores de aplicações passem apenas a utilizá-los e, de forma particular, para tecnologias específicas, alguns serviços tais como a replicação de componentes de software podem ser oferecidos de forma independente do domínio específico da aplicação.

1.1 OBJETIVOS DO TRABALHO

1.1.1 Objetivo Geral

Possibilitar a replicação de componentes de software em servidores de aplicação que utilizam a plataforma Java 2 Enterprise Edition - J2EE gerenciando-os através de meta-componentes.

1.1.2 Objetivos Específicos

a) Desenvolver meta-componente para prover controle de réplicas na plataforma J2EE;
b) garantir a transparência ao usuário. O código fonte dos componentes do usuário não deverá sofrer alteração, quando este utilizar o controle de réplicas;
c) garantir a portabilidade, o controlador de réplicas deve ser construído sem estender a Máquina Virtual Java;
d) comparar o desempenho, através da medição de tempos de resposta, dos métodos de replicação ativa, passiva e sem replicação.

1.2 LIMITAÇÕES DO TRABALHO

a) Seguir o padrão MVC (Model View Controller);
b) aplicar o estudo apenas à camada de negócios e destes, os objetos *SessionBean* do tipo S*tateless*;
c) utilizar apenas *container* padrão J2EE.

1.3 ESTRUTURA DO TRABALHO

Este trabalho é constituído de seis capítulos com os conteúdos distribuídos da seguinte forma:

a) no Capítulo 1 apresenta-se de forma resumida a proposta de trabalho, caracterizando a sua relevância, os seus objetivos e limitações;

b) o Capítulo 2 aborda o assunto Tolerância a Falhas em Sistemas Distribuídos, discorrendo sobre a necessidade de estabelecer-se técnicas de controle para tornar os sistemas distribuídos empresariais mais robustos;
c) no Capítulo 3 apresenta-se uma revisão bibliográfica sobre J2EE, ferramenta principal onde será aplicada a técnica de replicação;
d) o Capítulo 4 estabelece abordagens sobre os modelos de Controle de Réplicas.
e) o Capítulo 5 apresenta o modelo proposto necessário para o gerenciamento das réplicas em um estudo de caso proposto e sua performance;
f) o Capítulo 6, respaldado no estudo supracitado, apresenta as conclusões e recomendações para trabalhos futuros.

2 TOLERÂNCIA A FALHAS[1] EM SISTEMAS DISTRIBUÍDOS

Para entender-se a relevância deste estudo, discute-se alguns termos básicos sobre o assunto.

2.1 CONCEITOS BÁSICOS SOBRE TOLERÂNCIA A FALHAS

2.1.1 Falha, Erro e Defeito

Estando interessados no sucesso de determinado sistema de computação no atendimento da sua especificação. Um defeito (*failure*) é definido como um desvio da especificação. Defeitos não podem ser tolerados, mas deve ser evitado que o sistema apresente defeito. Define-se que um sistema está em estado errôneo, ou em erro, se o processamento posterior a partir desse estado pode levar a um defeito. Finalmente define-se falha ou falta (*fault*) como a causa física ou algorítmica do erro. Falhas são inevitáveis. Componentes físicos envelhecem e sofrem com interferências externas, sejam ambientais ou humanas. O software, e também os projetos de software e hardware, são vítimas de sua alta complexidade e da fragilidade humana em trabalhar com grande volume de detalhes ou com deficiências de especificação. Defeitos são evitáveis usando técnicas de tolerância à falhas. Alguns autores nacionais traduzem as palavras inglesas *failure* como falha e *fault* como falta. Para ser coerente com essa última tradução a área deveria se chamar tolerância a faltas, pois *failures* não podem ser toleradas (TAYSE, 2003).

---

[1] Segundo Tayse (2002), "aos poucos o termo dependabilidade vem substituindo tolerância à falhas no meio acadêmico. Em 2000, o Fault Tolerant Computing Symposium, FTCS, foi rebatizado Dependable Systems and Networks. Em 2003, o SCTF vai passar a se chamar LADC, Latin America Dependable".

2.1.2 O Modelo de Três Universos

Pode-se associar os conceitos de falha, erro e defeito a três universos conforme mostrado na figura 1. Falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

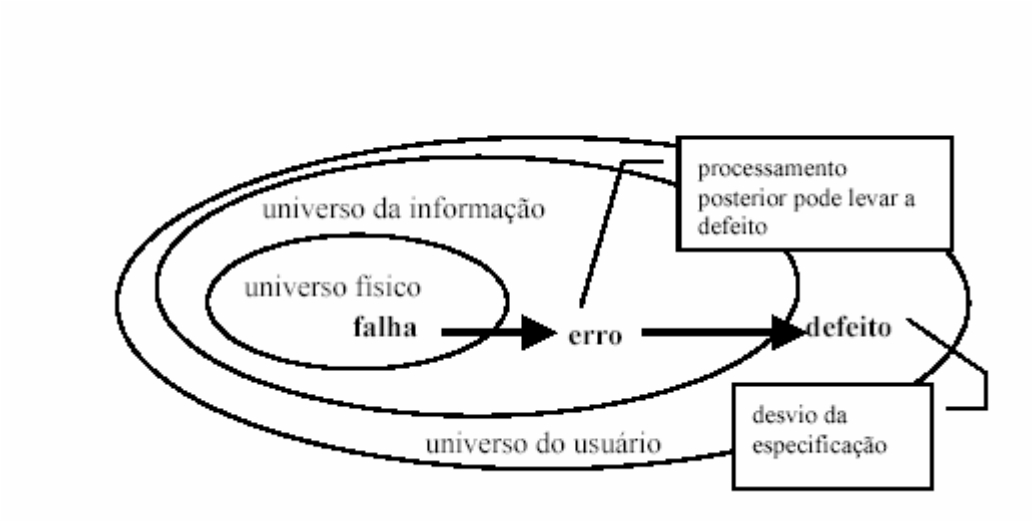

Figura 1: O mundo dos três erros. (PRADHAN, 1996)

Por exemplo: um chip de memória, que apresenta uma falha do tipo grudado-em-zero (stuck-at-zero) em um de seus bits (falha no universo físico), pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro no universo da informação) e como resultado o sistema pode negar autorização de embarque para todos os passageiros de um vôo (defeito no universo do usuário). É interessante observar que uma falha não necessariamente leva a um erro (aquela porção da memória pode nunca ser usada) e um erro não necessariamente conduz a um defeito (no exemplo, a informação de vôo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura).

Define-se latência de falha como o período de tempo desde a ocorrência da falha até a manifestação do erro devido àquela falha. Define-se latência de erro como o período de tempo desde a ocorrência do erro até a manifestação do defeito devido aquele erro.

Baseando-se no modelo de 3 universos, o tempo total desde a ocorrência da falha até o aparecimento do defeito é a soma da latência de falhas e da latência de erro.

2.1.3 Classificação de Falhas

As falhas aparecem geralmente classificadas em falhas físicas, aquelas de que padecem os componentes, e falhas humanas. Falhas humanas compreendem falhas de projeto e falhas de interação.

As principais causas de falhas são problemas de especificação, problemas de implementação, componentes defeituosos, imperfeições de manufatura, fadiga dos componentes físicos, além de distúrbios externos como radiação, interferência eletromagnética, variações ambientais (temperatura, pressão, umidade) e também problemas de operação.

Falhas de software e também de projeto são consideradas atualmente o mais grave problema em computação crítica. Sistemas críticos, tradicionalmente, são construídos de forma a suportar falhas físicas. Assim é compreensível que falhas não tratadas, e não previstas, sejam as que mais aborreçam, pois possuem uma grande potencial de comprometer a confiabilidade e disponibilidade do sistema.

2.2 DEPENDABILIDADE

O objetivo de tolerância a falhas é alcançar dependabilidade. O termo dependabilidade é uma tradução literal do termo inglês *dependability*, que indica a qualidade do serviço fornecido por um dado sistema e a confiança depositada no serviço fornecido.

Segundo Pradhan apud Tayse (2003), os principais atributos de dependabilidade são confiabilidade, disponibilidade, segurança de funcionamento (*safety*), segurança (*security*), mantenabilidade, testabilidade e comprometimento do desempenho (*performability*). Pode-se defini-los assim:

a) dependabilidade (*dependability*) é a qualidade do serviço fornecido por um dado sistema;
b) confiabilidade (*reliability*) é a capacidade de atender a especificação, dentro de condições definidas, durante certo período de funcionamento e condicionado a estar operacional no início do período;

c) disponibilidade (*availability*) é a probabilidade de o sistema estar operacional num instante de tempo determinado; alternância de períodos de funcionamento e reparo;
d) segurança (*safety*) é a probabilidade do sistema ou estar operacional e executar sua função corretamente ou descontinuar suas funções de forma a não provocar dano a outros sistema ou pessoas que dele dependam;
e) segurança (*security*) é a proteção contra falhas maliciosas, visando privacidade, autenticidade, integridade e irrepudiabilidade dos dados.

2.3 SISTEMAS DISTRIBUÍDOS

Segundo Tanenbaum (1995), um sistema distribuído é uma coleção de computadores independentes que parecem ao usuário como um único computador. A definição anterior implica em hardware formado por máquinas autônomas e software fornecendo a abstração de uma única máquina, já Lamport (1978) de uma forma jocosa definiu: “um sistema distribuído é aquele em que somos impedidos de trabalhar devido à falha de um computador de que nunca ouvimos falar.” Tayse (2003) escreveu que “sistemas distribuídos são construídos por vários nodos de processadores independentes”. Ou seja, os computadores de um sistema distribuído não apresentam memória comum, como nos computadores paralelos, e a troca de mensagens através de canais de comunicação é a responsável pelo acoplamento entre os elementos do sistema. Sistema distribuído não é sinônimo de redes de computadores. Uma rede de computadores pode fornecer a infra-estrutura computacional para um sistema distribuído, mas nem toda aplicação de rede é necessariamente distribuída.

2.3.1 Vantagens

Podem-se enumerar algumas vantagens dos sistemas distribuídos tais como, por exemplo, as relacionadas abaixo:

a) econômica: aproveita máquinas potencialmente ociosas; mais barato do que ter vários processadores interconectados do que ter um supercomputador;
b) velocidade: a capacidade de interconectar processadores possibilita que se obtenha performances que apenas um sistema composto é capaz de atingir;

c) distribuição inerente: algumas distribuições já o são por natureza. Ex.: aplicação para uma cadeia de supermercados, alguns tipos de jogos, etc.;
d) tolerância à falhas: quando uma máquina falha, o sistema como um todo pode continuar funcionando, apenas apresentando uma diminuição no seu desempenho;
e) crescimento incremental: pode-se aumentar o poder computacional através da inclusão de novos equipamentos;
f) flexibilidade os sistemas distribuídos são mais flexíveis do que as máquinas isoladas, por essa razão são muitas vezes utilizados até mesmo sem a necessidade de maior desempenho. Esta flexibilidade permite que vários usuários compartilhem dados e periféricos.

2.3.2 Desvantagens

Todo o conjunto de benefícios que cerca os sistemas distribuídos acaba por acarretar um grande número de desvantagens, algumas delas são:

a) no mercado ainda existem poucos softwares de prateleira que utilizam esta tecnologia;
b) dificuldade para evitar acesso indevido ao sistema;
c) a rede de interconexão pode causar problemas ou não dar a vazão exigida pela demanda.

2.4 UTILIZAÇÃO DE SISTEMAS DISTRIBUÍDOS

É crescente a necessidade de se construir sistemas computacionais distribuídos tolerantes a faltas, haja vista o uso destes sistemas em aplicações que exigem um bom nível de dependabilidade. Nesse caso, o custo de falhas no sistema, seja nos componentes de hardware ou software, pode ser alto, trazendo conseqüências indesejáveis (FRAGA, 2001).

As pesquisas em tolerância a faltas em sistemas distribuídos têm um importante papel no desenvolvimento de aplicações confiáveis. O objetivo primário desta área de pesquisa é a proposição de soluções que permitam aplicações terem seus serviços oferecidos continuamente, mesmo na presença de falhas parciais no sistema. Por outro lado, nos últimos anos, a área de sistemas distribuídos tem avançado também na direção

da padronização aberta. Esta tendência vem no sentido de combater soluções proprietárias devido aos altos custos de desenvolvimento de sistema que o uso das mesmas traz — por exemplo, custos provenientes da dificuldade de manutenção de software, interoperabilidade e portabilidade (FRAGA, 2001).

Muitos estudos sobre este tema tratam dos mecanismos para tolerância a faltas implementando-os no nível da aplicação, aumentando a complexidade no desenvolvimento da mesma e não reaproveitando o custo e o esforço gasto na implementação dos mecanismos.

De fato, o ideal seria disponibilizar estes mecanismos na forma de serviços que pudessem ser utilizados pelo projetista da aplicação sem que este se preocupasse com questões de implementação. Isto facilitaria a tarefa do projetista e ainda permitiria que os serviços fossem utilizados por diferentes aplicações.

Segundo Sampaio (2000), a fim de reduzir a complexidade no desenvolvimento de aplicações distribuídas com requisitos de confiança no funcionamento, os projetistas de tais aplicações têm empregado duas abordagens complementares, quais sejam:

a) Utilização de ferramentas de apoio à construção de sistemas tolerantes a faltas, por exemplo, *toolkits* de programação, bibliotecas de funções e serviços especializados do sistema operacional;
b) Restrição da semântica de falha dos componentes que formam a infra-estrutura de processamento e comunicação sobre a qual a aplicação será executada.

Aplicações com requisitos de confiança no funcionamento são, em maior ou menor graus, críticos. Desta maneira, a implementação dos serviços sobre os quais tais aplicações se apóiam deve ser validada.

No sentido de facilitar esta tarefa, normalmente utiliza-se um modelo de sistema conhecido para descrever a infra-estrutura de execução dos serviços.

Na literatura, ainda segundo Sampaio (2002), podem ser encontrados diferentes modelos de sistema, tais como: modelo síncrono, modelo assíncrono, modelo assíncrono temporizado e o modelo quase-síncrono. Estes modelos diferem, essencialmente, em relação a uma característica particular, as garantias de sincronismo providas por cada um deles. Tais garantias são expressas através de suposições feitas acerca dos atrasos na transmissão das mensagens e escalonamento de tarefas. Numa

escala de sincronismo, os modelos síncrono e assíncrono estão em extremidades opostas. Enquanto no modelo síncrono os atrasos na transmissão das mensagens e escalonamento de tarefas são limitados e conhecidos, no modelo assíncrono não é imposta nenhuma restrição em relação aos referidos atrasos.

Como pôde-se observar neste capítulo, garantir que sistemas distribuídos funcionem adequadamente, dispende para as pessoas e empresas um custo elevado, pois leva tempo para desenvolver-se sistemas confiáveis e tolerantes a falhas.

Para as empresas que utilizam servidores de aplicação baseados na plataforma J2EE a disponibilização de serviços que implementem replicação seria então da maior importância. No próximo capítulo apresenta-se uma revisão sobre a plataforma, seu funcionamento e os mecanismos que permitirão a aplicação e gerenciamento de réplicas de componentes.

3 J2EE – JAVA 2 ENTERPRISE EDITION

Hoje, mais e mais desenvolvedores querem escrever aplicações com transações distribuídas para as empresas e aumentar a velocidade, segurança e confiabilidade de tecnologias *server-side*. Sabe-se que no mundo aumenta velozmente a demanda por *e-commerce* e tecnologias de informação e que aplicações empresariais tem sido definidas, construídas e mantidas por menos dinheiro e com maior velocidade.

Propondo uma solução para atender estas necessidades a Sun Microsystems apresentou ao mercado empresarial a plataforma Java 2 Enterprise Edition.

3.1 HISTÓRICO

No início dos anos 90, os provedores de sistemas de informação começaram a responder às demandas de aplicação através do modelo de aplicação multicamadas, que basicamente separa a lógica do negócio dos serviços do sistema e da interface do usuário, colocando-a em uma camada intermediária, de *middle-tier*. A evolução de novos serviços de *middle-tier* deu ímpeto adicional a essa nova arquitetura. Ainda, o crescente uso da Internet e Intranets em aplicações corporativas contribuiu para uma maior ênfase em clientes mais leves e mais fáceis de instalar.

O projeto multicamadas simplifica o desenvolvimento, a implantação e a manutenção de aplicações, possibilitando que os desenvolvedores concentrem-se nas especificidades da programação da lógica de negócio, pois contam com o apoio dos serviços de infra-estrutura e de aplicações cliente (*standalone* ou em *browsers* Web).

Uma vez desenvolvida, a lógica de negócio pode ser implantada nos servidores mais apropriados às necessidades de uma organização. Mas o fato do modelo multicamada ter sido implementado até esta época em uma variedade de padrões divergentes limitava a habilidade dos desenvolvedores de eficientemente construir aplicações de componentes padronizadas (capazes de serem implantadas em uma larga variedade de plataformas), ou de prontamente escalar aplicações para adequá-las às condições de negócios em constante modificação.

Na divisão JavaSoft da Sun Microsystems, diversos esforços de desenvolvimento apontavam para o que se tornaria a tecnologia EJB, dentre estes esforços pode-se enumerar pelo menos três:

a) a tecnologia dos servlets mostrou que os desenvolvedores estavam ávidos para criar ambiente tipo CGI (Common Gateway Interface) que pudessem executar em qualquer servidor Web com suporte à plataforma Java;
b) a tecnologia JDBC (Java *Database Connectivity*) forneceu um modelo que casava as características WORA (Write Once, Run Anywhere) da linguagem de programação Java com os sistemas de gerenciamento de banco de dados;
c) a arquitetura de componentes *JavaBeans* demonstrou a grande utilidade de encapsular conjuntos completos de funções em componentes, prontamente reutilizáveis e fáceis de configurar, no lado cliente.

A convergência desses três levou ao padrão EJB (SOUZA, 2002a).

Aplicações multicamadas baseadas em componentes estão definindo a evolução do futuro da computação. E a tecnologia dos *Enterprise JavaBeans* oferece componentes com benefícios de portabilidade e escalabilidade para uma vasta gama de servidores, juntamente com desenvolvimento, implantação e manutenção simplificados.

Em abril de 1997, a Sun Microsystems anunciou sua iniciativa de desenvolver uma Plataforma Java para Corporações (Java *Platform for the Enterprise*). Para tanto, encorajou o desenvolvimento de uma coleção de Extensões Java Padrão (Java *Standard Extensions*), conhecida como APIs *Enterprise* Java, ou APIs Java Corporativas. Essas APIs tinham o objetivo de prover interfaces de programação independentes do fornecedor de *middle-tier*, sendo que sua pedra fundamental seria a API EJB (THOMAS, 1998).

Observa-se que a arquitetura EJB definiu um padrão de componentes servidores e uma interface independente de fornecedor para aplicações servidoras Java. No entanto, apenas o modelo EJB (Enterprise Java Bean) não é suficiente para garantir portabilidade, interoperabilidade e consistência de plataforma, pois não especifica detalhes de implementação – tais como comunicação e protocolos de transação, apenas os indica. Ao mesmo tempo, as APIs *Enterprise* Java ainda eram classificadas como extensão padrão da plataforma Java, ou seja, não havia garantias que essas APIs estariam instaladas em um sistema específico.

Esses problemas foram contornados pela definição da plataforma J2EE. Através dela, a Sun especificou todo um mecanismo capaz de validar um sistema como apto a suportar um ambiente Java completo, com todas as APIs *Enterprise* Java. Isso significa

que uma plataforma J2EE validada provê, com certeza, um ambiente de execução (runtime) Java integrado e consistente, que garante uma certa qualidade de serviço e assegura portabilidade e interoperabilidade para aplicações (SOUZA, 2001).

3.2 A PLATAFORMA

A Plataforma Java 2, Enterprise Edition (J2EE) define o padrão para desenvolvimento de aplicações empresarial multicamadas, simplifica aplicações empresariais baseando-as na padronização de componentes modulares, provendo um conjunto de serviços para estes componentes e tratando muitos detalhes da aplicação automaticamente sem programação complexa. Utiliza as vantagens de muitas das características da plataforma Java 2, Standard Edition, tais como portabilidade "Write Once, Run Anywhere", API JDBC para acesso a banco de dados, tecnologia CORBA para interação com os recursos existentes na empresa, e um modelo de segurança que protege os dados mesmo em aplicações para a Internet.

O padrão J2EE inclui uma especificação completa e testes de compatibilidade para garantir a portabilidade de aplicações através da grande variedade de sistemas empresariais existentes capazes de suportar J2EE. Abaixo, na figura 2, pode-se ver uma aplicação empresarial segundo o modelo utilizado nesta plataforma.

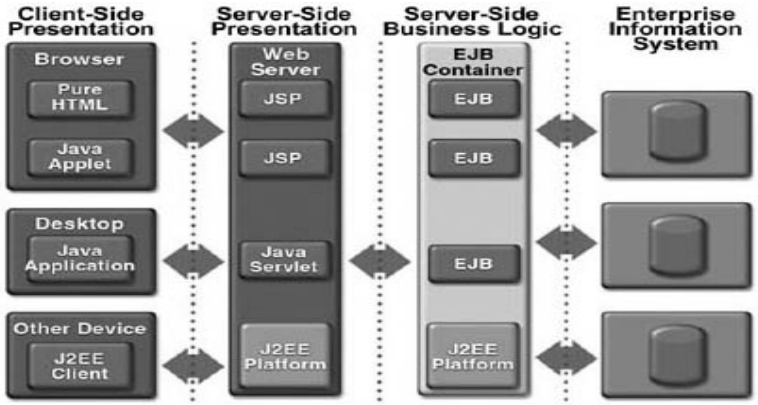

Figura 2: Modelo de aplicação empresarial (SUN, 2003b)

Neste modelo tem-se a aplicação construída totalmente baseada em componentes e gerenciada por *containers* e os *connectors* são responsáveis pela conexão com os softwares legados da empresa como se pode ver na figura 3 abaixo.

Figura 3: Fundamentos do modelo da plataforma (SUN, 2003b)

3.3 J2EE *CONTAINERS*

Normalmente, aplicações multicamadas para pequenos clientes são difíceis para escrever porque elas envolvem muitas linhas de intrincados códigos para manipular transações e gerenciamento de estado, *multi-threading*, recursos de *pooling*, e outros detalhes complexos de baixo nível. A arquitetura J2EE baseada em componentes e independente de plataforma torna fácil escrever aplicações porque a lógica de negócios está organizada dentro de componentes reutilizáveis. Além disso, o servidor J2EE fornece suporte na forma de *container* para todo tipo de componentes. Não é preciso desenvolver estes serviços eles já estão disponíveis e pode-se concentrar apenas no desenvolvimento para resolver o problema do negócio.

3.3.1 Serviços do Container

Segundo Sun (2003a), containers são a interface entre um componente e as funcionalidades de baixo nível de uma plataforma específica que suportam este componente. Para que enterprise bean, ou componentes de aplicações cliente possam ser executados precisam estar montados dentro de uma aplicação J2EE e implantada dentro do *container* dela.

O processo de montagem envolve um conjunto de especificações no *container* para cada componente da aplicação J2EE e para a própria aplicação J2EE. Os ajustes do *container* customizam a sustentação subjacente fornecida pelo servidor J2EE, que inclui

serviços tais como a segurança, a gerência da transação, busca de nomes (JNDI- Java Naming Directory Interface), e conectividade remota. Estão aqui alguns dos destaques:

a) o modelo de segurança J2EE permite a você configurar um componente Web ou um *enterprise bean* para que os recursos do sistema sejam acessados somente por usuários autorizados;
b) o modelo da transação do J2EE deixa-o especificar relacionamentos entre os métodos que realizam uma transação de modo que todos os métodos em uma transação sejam tratados como uma única unidade;
c) os serviços de busca do JNDI fornecem uma interface unificada para serviços de nomes e de diretórios na empresa de modo que os componentes da aplicação possam alcançar estes nomes e diretórios;
d) o modelo de conectividade remota do J2EE controla comunicações de baixo nível entre clientes e *enterprise beans*. Depois que um *enterprise bean* é criado, um cliente invoca métodos nele como se estivesse na mesma máquina virtual.

O fato de que a arquitetura J2EE fornece serviços configuráveis significa que os componentes da aplicação dentro da mesma aplicação J2EE podem se comportar diferentemente dependendo de onde são implantados. Por exemplo, um *enterprise bean* pode ter os ajustes de segurança que lhe permitem um determinado nível de acesso aos dados da base de dados em um ambiente e um outro nível de acesso à base de dados em um outro ambiente de execução.

O *container* controla também serviços não configuráveis tais como ciclos de vida de *enterprise bean* e de *servlets*, *pooling* do recurso da conexão com base de dados, persistência dos dados, e acesso às APIs da plataforma. Embora a persistência dos dados seja um serviço não configurável, a arquitetura J2EE deixa desativar manualmente, no *container,* o gerenciador de persistência, incluindo o código apropriado em sua implementação do *enterprise bean* quando se quer mais controle do que este fornece. Por exemplo, pode-se usar seus próprios métodos de busca ou criar um *cache* customizado da base de dados.

3.3.2 Tipos de Container

O processo da distribuição instala componentes da aplicação J2EE nos *containers* do J2EE ilustrados na figura 4.

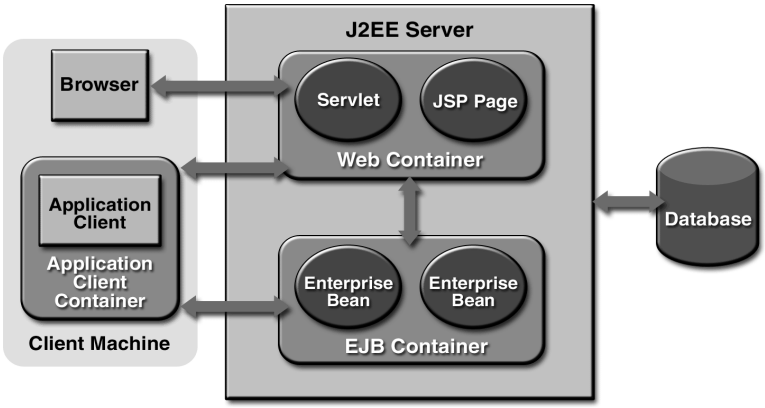

Figura 4: Servidor J2EE e Containers (SUN, 2003a)

Abaixo apresenta-se as funcionalidades do ambiente representado na figura 4:

a) o J2EE *Server* é a parte programa do produto J2EE, fornece *containers* EJB e Web;
b) *o Container Enterprise* JavaBeans (EJB) controla a execução de *enterprise beans* para aplicações J2EE. Os *enterprise beans* e seu container funcionam no servidor J2EE;
c) *o Container* Web controla a execução de JSP (Java *Server Pages*) e componentes *servlet* da aplicação J2EE. Componentes Web e seu *container* funcionam no servidor J2EE;
d) *o Container* Aplicação Cliente controla a execução de componentes da aplicação cliente. A aplicação cliente e seu *container* executam no cliente.
e) *o Container* Applet controla a execução de applets. Consiste de um navegador Web e um *plug-in* rodando junto no cliente.

3.3.3 Empacotamento

Os componentes J2EE são empacotados separadamente e re-empacotados para constituir uma aplicação J2EE a ser distribuída. Cada componente e seus arquivos relacionados tais como arquivos GIF e HTML ou classes de serviço público do lado

servidor, e um descritor da distribuição são montados em um módulo e adicionados à aplicação J2EE. Uma aplicação J2EE é composta de um ou mais *enterprise beans*, componente Web e o componente do cliente da aplicação. A solução final da empresa pode usar uma única aplicação J2EE ou ser composta de duas ou mais aplicações J2EE, dependendo das exigências do projeto.

Uma aplicação J2EE e cada um de seus módulos têm seu próprio descritor de implantação. Um descritor de implantação é um documento XML (com extensão xml) que descreve ajustes da distribuição de um componente. Um descritor de implantação do módulo *enterprise bean*, por exemplo, declara atributos da transação e autorizações de segurança para um *enterprise bean*. A informação do descritor de implantação é declarativa, portanto pode ser mudada sem modificar o código fonte do *bean*. Durante o funcionamento, o usuário J2EE lê o descritor da distribuição e age em cima do componente conforme sua configuração.

Uma aplicação J2EE com todos os seus módulos é entregue em um arquivo *Enterprise Archive* (EAR). Um arquivo EAR é um arquivo Java Archive (JAR) padrão com uma extensão ear. Na versão GUI do DeployTool da distribuição do J2EE SDK, cria-se um arquivo EAR primeiramente e adiciona-se o JAR e os arquivos Web Archive (WAR) ao EAR. Se utilizar-se as ferramentas do *packager* pela linha de comando, entretanto, cria-se os arquivos JAR e WAR primeiramente e só depois então cria-se o EAR. Observa-se que:

a) cada arquivo JAR EJB contem um descritor de implantação, o *enterprise bean*, e arquivos relacionados;
b) cada arquivo JAR do cliente da aplicação contem um descritor de implantação, os arquivos da classe do cliente da aplicação, e arquivos relacionados;
c) cada arquivo WAR contem um descritor de implantação, os componentes Web, e arquivos relacionados.

Usando módulos EAR torna possível montar várias aplicações diferentes de J2EE usando alguns dos mesmos componentes. Nenhuma codificação extra é necessária.

3.3.4 Papéis no Desenvolvimento

Os módulos reusáveis tornam possível dividir o processo do desenvolvimento e da distribuição da aplicação em papéis distintos de modo que diferentes pessoas ou companhias possam executar partes diferentes do processo.

Os primeiros dois papéis envolvem comprar e instalar o produto e ferramentas J2EE. Uma vez que o software é comprado e instalado, componentes J2EE podem ser desenvolvidos por fornecedores de componentes da aplicação, ser montados por montadores de aplicação, e ser implantados por implantadores de aplicação.

Em uma grande organização, cada um destes papéis pode ser executado por indivíduos ou por equipes diferentes. Esta divisão de trabalho é feita porque cada um dos papéis que estejam mais adiantados pode enviar um arquivo portátil para ser a entrada de um papel subseqüente. Por exemplo, na fase de desenvolvimento dos componentes da aplicação, um colaborador do software de *enterprise bean* entrega arquivos JAR de EJB. No papel do conjunto da aplicação, um outro colaborador combina estes arquivos JAR de EJBs em uma aplicação de J2EE e conserva-as em um arquivo EAR. No papel da distribuição da aplicação, um administrador de sistema no local do cliente usa o arquivo EAR para instalar a aplicação J2EE em um servidor J2EE.

Os papéis diferentes não são sempre executados por pessoas diferentes. Se for uma pequena companhia, por exemplo, ou se for uma prototipação, a mesma pessoa realizará todos os papéis.

3.3.4.1 Provedor de produto J2EE

O fornecedor de produto J2EE é uma companhia que projeta e torna disponíveis para a compra a plataforma J2EE, APIs, e outras características definidas na especificação de J2EE. Os fornecedores de produto são tipicamente sistemas operacionais, sistemas de banco de dados, servidores de aplicação, ou vendedores de servidores Web que implementam a plataforma J2EE.

3.3.4.2 Provedor de componente de aplicação

O fornecedor de componente de aplicação é a companhia ou pessoa que cria componentes Web, *enterprise beans*, *applets*, ou aplicações cliente para usar na aplicação J2EE.

Segundo Souza (2001), produz os blocos de construção de uma aplicação J2EE. Geralmente é especialista no desenvolvimento de componentes reutilizáveis, e tem conhecimento no domínio do negócio da aplicação. Não precisa conhecer o ambiente operacional onde seus componentes serão utilizados. Podem criar documentos HTML, JSP, enterprise beans, etc. Define os atributos declarativos dos componentes nos descritores de implantação, e empacota componentes em arquivos JAR e WAR.

3.3.4.3 Montador de aplicação

O montador de aplicação é a companhia ou pessoa que recebe os componentes da aplicação através dos arquivos JAR e monta-os dentro de uma aplicação J2EE no arquivo EAR. O montador ou implantador pode editar o descritor de implantação diretamente ou usar ferramentas que corretamente adicionam marcadores XML de acordo com as seleções interativas. É o responsável por prover instruções de montagem, descrevendo dependências externas da aplicação que o implantador resolverá no processo de implantação.

3.3.4.4 Implantador e administrador da aplicação

É um especialista em um ambiente operacional, responsável pela implantação de componentes e aplicações J2EE em tal ambiente. Resolve as dependências externas das aplicações declaradas pelo Provedor de Componente de Aplicação e pelo Montador de Aplicação.

O processo de implantação geralmente envolve dois estágios: primeiro, o implantador (via ferramenta de implantação) gera as classes e interfaces adicionais que habilitam o *container* a gerenciar os enterprise beans em tempo de execução.

Essas classes são específicas do *container* em questão. Em seguida, o implantador executa a instalação dos *Enterprise Beans* e de suas classes e interfaces adicionais no container EJB.

Em certas situações, um implantador qualificado pode customizar a lógica do negócio dos enterprise beans no momento de sua implantação. Tipicamente, ele usa as ferramentas do *container* para escrever um código de aplicação relativamente simples que envolve (wrap) os métodos de negócio do bean.

É o responsável pela configuração e administração de uma infra-estrutura de rede e de computação de uma empresa. Também é o responsável pela supervisão da boa execução de aplicações J2EE (SOUZA, 2001).

3.3.4.5 Provedor de ferramenta

É uma companhia ou pessoa que cria, monta, e empacota ferramentas usadas por fornecedores de componentes, montadores, e implantadores.

3.4 SOFTWARES DA IMPLEMENTAÇÃO DE REFERÊNCIA

O J2EE SDK não é comercial por definição e é distribuído livremente pela Sun Microsystems para demonstrações, prototipação e uso educacional. Vem com um servidor de aplicação J2EE, servidor Web, banco de dados relacional, APIs J2EE, e um conjunto completo de desenvolvimento e ferramentas de implantação (SUN, 2003a).

A finalidade do J2EE SDK é permitir que os fornecedores de produtos testem as condições em que seus produtos serão entregues e que funcionem seguindo rigorosamente a especificação da plataforma. Isto garantirá a portabilidade dos seus produtos.

3.4.1 Acesso a Banco de Dados

Os bancos de dados relacionais fornecem armazenamento persistente para os dados da aplicação.

Uma implementação J2EE não é obrigada a suportar um tipo particular de banco de dados isto significa que o banco de dados suportado pode variar.

3.4.2 APIs J2EE

A plataforma Java 2, Standard Edition (J2SE) SDK é necessária para rodar o J2EE SDK e fornecer o núcleo de APIs para escrever componentes J2EE, núcleo para desenvolver ferramentas, e a máquina virtual Java. O J2EE SDK fornece as seguintes APIs para serem usadas em aplicações J2EE (SUN, 2003a):

3.4.2.1 Enterprise JavaBeans technology 2.0

Um *enterprise bean* é o corpo do código com os campos e os métodos para executar os módulos da lógica do negócio. Pode-se pensar em um *enterprise bean* como um bloco de edifício que possa ser usado sozinho ou com outros *enterprise bean* para executar a lógica do negócio no servidor J2EE.

Há três tipos de *enterprise bean*: *session beans*, *entity beans*, e message-driven beans. Os *enterprise bean* interagem freqüentemente com as bases de dados. Um dos benefícios do *enterprise bean* é que não se tem que escrever nenhum código de SQL ou usar o JDBC API diretamente, executar operações de acesso à banco de dados; o *container* de EJB faz isto.

3.4.2.2 JDBC API 2.0

A API JDBC permite invocar comandos SQL através de métodos da linguagem de programação Java. Usa-se a API JDBC em um *enterprise bean* quando se sobrescreve o gerenciador de persistência *default*, operações de acesso ao banco de dados são manipulados pelo *container* e seu *enterprise bean* não contem código JDBC ou comandos SQL. Pode-se também usar a API JDBC de um servlet ou página JSP (Java Server Pages) para acessar diretamente o banco de dados sem ser através de um *enterprise bean*.

A API JDBC tem duas partes: uma interface em nível de aplicação usada pelos componentes da aplicação para acessar o banco de dados, e uma interface provedora de serviço para ligar um *driver* para a plataforma J2EE.

3.4.2.3 Tecnologia java servlet 2.3

A tecnologia Java Servlet permite definir classes servlets específicas HTTP. Uma classe servlet estende a capacidades de servidores que são acessados por aplicações que seguem o modelo de programação requisição-resposta. Ainda que servlets possam responder a qualquer tipo de solicitação, eles são comumente usados para estender as aplicações nos servidores Web.

3.4.2.4 JavaServer pages technology 1.2

A tecnologia JavaServer Pages permite colocar código diretamente dentro de um documento texto. Uma página JSP é um documento texto que contem dois tipos de

texto: modelo de dados estáticos, que podem ser expressos como texto, tal como o formato do HTML, do WML, do XML, e do JSP, que determinam como a página constrói o conteúdo dinâmico.

3.4.2.5 Java message service 1.0

O JMS é um serviço de mensagens padrão que permite componentes de aplicação J2EE criar, receber e ler mensagens. Ele habilita comunicação distribuída de forma livre, confiável e assíncrona.

3.4.2.6 Java naming and directory interface 1.2

O JNDI fornece a funcionalidade de nomes e diretórios, pode associar atributos a objetos e objetos usando seus atributos. Aplicações podem acessar múltiplos nomes e serviços de diretório incluindo nomes existentes e serviços de diretórios tais como LDAP, NDS, DNS, e NIS. Isto permite que aplicações J2EE coexistam com aplicações e sistemas legados.

3.4.2.7 Java transaction API 1.0

A API Java Transaction ("JTA") fornece uma interface padrão para demarcar transações. A arquitetura J2EE fornece um *auto commit default* para manipular *commits* e *rollbacks*.

3.4.2.8 JavaMail API 1.2

Aplicações J2EE podem usar a API JavaMail para enviar e receber e-mail. A API JavaMail possui duas partes: uma interface em nível de aplicação usada pelos componentes da aplicação e para enviar e-mail, e uma interface provedora de serviços.

3.4.2.9 Java API for XML processing 1.1

A API JAXP permite processar documentos XML dentro de aplicações escritas na linguagem Java. JAXP usa os padrões SAX (Simple API for XML Parsing) e DOM (Document Object Model) para o processamento do documento, dando ao usuário a possibilidade de escolha entre a representação do XML como um objeto ou a manipulação dos dados de forma seqüencial.Quando um documento XML é processado com a API SAX, tem-se uma leitura ou gravação seqüencial dos elementos. Por outro lado, a API DOM fornece uma estrutura em árvore para a representação dos elementos.

A construção do DOM precisa ler todo o documento XML e montar a hierarquia na memória, isto faz com que o tempo e o uso da memória para o DOM sejam maiores do que para o SAX. O principal pacote da API JAXP é o javax.xml.parser. Dentro do pacote pode-se encontrar duas implementações que são representadas pelas classes: SAXParserFactory e DocumentBuilderFactory. Esta última é a que permite lidar com o documento XML como um objeto no padrão DOM (MENÉNDEZ, 2002).

3.4.2.10 J2EE connector architecture 1.0

A arquitetura Connector J2EE é usada por vendedores de ferramentas J2EE e integradores de sistema para criar adaptadores de recurso que realizem a conexão dos sistemas legados da empresa com os componentes J2EE da aplicação. Um adaptador de recurso é um componente de software que permite a aplicações J2EE acessar e interagir com o gerenciador de recursos base.

3.4.2.11 Java authentication and authorization service 1.0

O Java Authentication and Authorization Service ("JAAS") fornece um caminho para a aplicação autenticar e autorizar um usuário ou grupo de usuários específicos a executá-lo. JASS é uma versão da linguagem Java do framework padrão Pluggable Authentication Module (PAM) que estende a arquitetura de segurança da plataforma para suportar autorização baseada no usuário.

3.5 ENTERPRISE JAVABEANS

"Enterprise JavaBeans é uma arquitetura de computação distribuída baseada em componentes. *Enterprise beans* são componentes de aplicações empresariais distribuídas orientadas a transação" (DEMICHIEL, 2002).

Os *enterprise beans* representam o conceito central da plataforma J2EE, pode-se dizer que a maioria das tecnologias e APIs da plataforma J2EE servem para criar uma plataforma onde seja possível a utilização dos EJBs.

3.5.1 Características do componente

As características essenciais de um *enterprise bean* segundo Demichiel (2002) são:

a) um *enterprise bean* tipicamente contem a lógica de negócios que opera nos dados da empresa;
b) as instâncias dos *enterprise beans* são criadas e gerenciadas durante a execução por um *container;*
c) um *enterprise bean* pode ser customizado em tempo de implantação editando seu ambiente de entrada;
d) várias informações de serviço, tais como uma transação e atributos de segurança, são separados da classe do *enterprise bean*. Isto permite que informações de serviço sejam gerenciadas por ferramentas durante o processo de montagem e implantação;
e) acesso ao cliente é mediado pelo *container* no qual o *enterprise bean* está implantado;
f) se um *enterprise bean* usa somente os serviços definidos pela especificação EJB, o *enterprise bean* pode ser implantado em qualquer *container* EJB compatível. *Containers* especializados podem fornecer serviços adicionais além daqueles definidos pela especificação EJB. Um *enterprise bean* que dependa daquele serviço somente poderá ser implantado em um container que suporta o serviço;
g) um *enterprise bean* pode ser incluído em uma aplicação montada sem requerer mudanças no código fonte ou recompilação do *enterprise bean.*

3.5.2 Contratos Enterprise JavaBeans

Basicamente, existem dois tipos de contratos EJB: o contrato da Visão de Cliente e o contrato do Componente (SOUZA, 2001), como descrito nas subseções a seguir.

3.5.2.1 Contrato da visão de cliente

É o contrato entre o cliente e um container, que provê um modelo de desenvolvimento uniforme para aplicações que usam e*nterprise beans*, e também habilita o uso de ferramentas de desenvolvimento e a reutilização de componentes. O cliente de um e*nterprise bean* pode ser um outro e*nterprise bean* implantado no mesmo ou em outro *container*, ou ainda pode ser um programa Java arbitrário (um *applet* ou um *servlet*). Essa visão de cliente também pode ser mapeada para ambientes não-Java, tais como clientes CORBA escritos em outras linguagens de programação. O contrato de

visão de cliente é implementado através de um conjunto específico de interfaces, definidas pelo desenvolvedor do EJB. O container EJB invoca essas interfaces que envolvem o EJB (*wrappers*) em certos momentos da execução. Isso significa que o cliente não interage diretamente com o *enterprise bean*, mas sim com essas interfaces, cujas classes são geradas automaticamente pelo próprio container. A visão de cliente, ilustrada na Figura 7, inclui as interfaces *Home* e *Remote*, como descrito a seguir (THOMAS ,1998):

a) a interface *Home* provê acesso aos serviços de ciclo de vida do *bean*. Os clientes podem usá-la para criar e destruir instâncias do *bean*. O *container* automaticamente registra esta interface para cada classe de *bean* instalada no mesmo, através da API Java Naming and Directory Interface (JNDI). Isso permite que o cliente localize a interface *Home* de uma classe de *bean* para criar uma nova instância do mesmo. Quando um cliente cria ou localiza um *bean*, o *container* retorna a sua interface *Remote*. A Figura 5 apresenta um exemplo de codificação da interface *Home* que o fornecedor de e*nterprise bean* deve prover, juntamente com o *enterprise bean* propriamente dito;

```
1: package Beans;
2: import java.rmi.RemoteException;
3: import javax.ejb.CreateException;
4: import javax.ejb.EJBHome;
5: public interface CalcHome extends EJBHome {
6: Calc create() throws CreateException, RemoteException;
7: }
```

Figura 5: Exemplo de codificação da interface *Home* (SOUZA, 2001).

b) interface *Remote* provê acesso aos métodos de negócio do *enterprise bean*. Permite que o *container* intercepte todas as operações feitas no *enterprise bean*. Cada vez que o cliente invoca um método, sua requisição passa pelo *container* antes de ser delegada ao *enterprise bean*, e é dessa forma que o *container* implementa o gerenciamento de estado, o controle de transação, os serviços de segurança e de persistência transparentemente, tanto para o cliente quanto para o *enterprise bean*, atuando como elemento de ligação entre o *enterprise*

bean e o servidor EJB. A interface *Remote* estende a interface javax.ejb.EJBObject. A Figura 6 apresenta um exemplo de codificação da interface *Remote* que o fornecedor de *enterprise bean* também deve fornecer.

```
1: package Beans;
2: import javax.ejb.EJBObject;
3: import java.rmi.RemoteException;
4: public interface Calc extends EJBObject {
5:     public double calcBonus(int multiplier, double bonus)
6:     throws RemoteException;
7: }
```

Figura 6 : Exemplo de codificação da interface *Remote* (SOUZA, 2001).

3.5.2.2 Contrato do componente

É o contrato entre um *enterprise bean* e seu c*ontainer*. Basicamente determina qual os requisitos que devem ser seguidos pelos (SOUZA, 2001):

a) desenvolvedores do EJB devem implementar os métodos de negócio na classe do *enterprise bean* e especificar as interfaces *Home* e *Remote;*
b) desenvolvedores do *container* devem garantir que *container* delegue a invocação feita pelo cliente aos métodos do EJB e que o *container* implemente as classes das interfaces *Home* e *Remote*.

A Figura 7 ilustra as principais interdependências entre o *enterprise bean*, o *container*, o servidor EJB e o cliente.

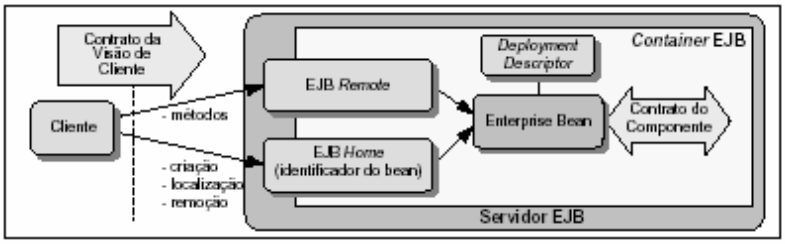

Figura 7 : A visão dos contratos (SOUZA, 2001).

3.5.3 Arquivo ejb-jar

Um arquivo ejb-jar é um formato padrão usado por ferramentas de EJB para empacotar *enterprise beans* com sua informação declarativa. O arquivo ejb-jar é destinado para ser processado por ferramentas de montagem e de implantação.

O arquivo ejb-jar é um contrato usado tanto com o fornecedor do *bean* e o montador da aplicação como entre o montador da aplicação e o implantador.

O arquivo ejb-jar inclui:

a) aquivos de classes Java dos *enterprise beans* e seus *home*, componentes, e interfaces *web service endpoint*;

b) um descritor de implantação XML. O descritor de implantação fornece duas informações: a estrutural e a de montagem sobre os enterprise beans no arquivo ejb-jar. A informação de montagem da aplicação é opcional.

Um exemplo de descritor de implantação seria o da Figura 8, abaixo:

```
1:  <?xml version="1.0" encoding="Cp1252"?>

2:  <!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
      JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

3:  <ejb-jar>
4:    <description>Session Bean para cálculo de multiplicação</description>
5:    <display-name>CalcJar</display-name>
6:    <enterprise-beans>
7:      <session>
8:        <description>no description</description>
9:        <display-name>CalcEJB</display-name>
10:       <ejb-name>CalcEJB</ejb-name>
11:       <home>Beans.CalcHome</home>
12:       <remote>Beans.Calc</remote>
13:       <ejb-class>Beans.CalcEJB</ejb-class>
14:       <session-type>Stateless</session-type>
15:       <transaction-type>Bean</transaction-type>
16:     </session>
17:   </enterprise-beans>
18: </ejb-jar>
```

Figura 8: Exemplo de descritor de implantação (SOUZA, 2001).

3.5.4 Objetos Session, Entity, e Message-Driven

A arquitetura *Enterprise JavaBeans* define três tipos de objetos *enterprise bean*, *session*, *entity* e *message-driven* (DEMICHIEL, 2002).

3.5.4.1 Um objeto *session*

Um componente tipo *session*, ou de sessão, é um componente servidor que visa suprir os seguintes requisitos de aplicação:

a) representar uma sessão com um único cliente, mantendo o estado conversacional (*stateful*) das múltiplas invocações de métodos realizadas pelo cliente, ou representar um serviço sem estado (*stateless*) para múltiplos clientes;
b) poder estar ciente de uma transação e atualizar dados de um banco de dados;
c) ter uma identificação única gerada e gerenciada por seu *container*;
d) ser eliminado quando seu *container* sofrer uma parada: o cliente deve então restabelecer a conexão (criar um novo componente *session*) para continuar a computação.

A Figura 9 apresenta um exemplo de codificação de *session bean*. Para esse exemplo, as interfaces *Home* e *Remote* correspondentes são as apresentadas nas Figura 5 e 6 – observa-se que todos os espaços de nomes das classes e interfaces nesses

exemplos seguem o mesmo padrão (package Beans). O descritor de implantação correspondente é o da Figura 8.

```
2: package Beans;
3: import java.rmi.RemoteException;
4: import javax.ejb.SessionBean;
5: import javax.ejb.SessionContext;
6: public class CalcEJB implements SessionBean {
7:     public double calcBonus(int multiplier, double bonus) {
8:            double calculo = (multiplier * bonus);
9:            return calculo;
10:    }
11:    public void ejbCreate() { }
12:    public void setSessionContext(SessionContext ctx) { }
13:    public void ejbRemove() { }
14:    public void ejbActivate() { }
15:    public void ejbPassivate() { }
16:    public void ejbLoad() { }
17:    public void ejbStore() { }
18: }
```

Figura 9 :Exemplo de codificação de um componente session bean.

3.5.4.2 Um objeto *entity*

Um componente tipo *entity*, ou entidade é um componente servidor que visa suprir os seguintes requisitos de aplicação:

a) eepresentar um objeto de negócio, que pode ser compartilhado entre múltiplos clientes;
b) ter um ciclo de vida diretamente atrelado a um banco de dados – o identificador único de um *entity* é a chave primária de uma entidade em um banco de dados;
c) sobreviver a uma parada do *container* EJB – o componente *entity*, sua chave primária e sua referência remota são mantidas. Se o estado de um componente *entity* tiver sido atualizado por uma transação no momento da parada do *container*, o estado do *entity* é automaticamente reiniciado com o estado existente após o último *commit*. Essa falha, porém, não é totalmente transparente ao cliente, que recebe uma exceção se chamar esse *entity* após a parada.

3.5.4.3 Um objeto *message-driven*

Um típico objeto *message-driven* tem as seguintes características:

a) executa recebimento de uma única mensagem do cliente;

b) é invocado de forma assíncrona;
c) pode estar submetido a controle de transações;
d) pode atualizar dados em uma base de banco de dados;
e) não representa dados diretamente compartilhados na base de dados, embora possa alcançar e atualizar tais dados;
f) seu ciclo de vida é relativamente curto ;
g) é *stateless*;
h) é removido quando o *container* EJB fica fora de operação. O *container* tem que restabelecer um novo objeto message-driven para continuar a computação.

Um típico *container* EJB fornece um ambiente de execução escalável para executar um grande número de objetos message-driven concorrentemente.

3.5.5 Regras para Criação de Enterprise JavaBeans 2.0

Para a criação de Enterprise JavaBeans é necessário seguir a especificação, onde esta define algumas regras.

3.5.5.1 Regra 1

Todo EJB deve possuir interface *Home*, *Remote* e a classe abstrata do *bean*, onde o nome da interface *Home* deve ser formado pelo conceito que o *bean* representa, mais a palavra *Home*, o nome da interface *Remote* deve ser formado pelo conceito que o *bean* representa e o nome da classe abstrata deve ser formado pelo conceito que o *bean* representa mais a palavra *Bean*.

Quando o EJB for um *EntityBean* e possuir classe para a chave primária, o nome da classe para a chave primária deve ser formado pelo conceito do *EntityBean* mais a palavra PK. A interface remota possui exatamente o nome do conceito, pelo fato dela representar o EJB para a camada cliente.

3.5.5.2 Regra 2

A interface *Home* deve ser pública e estender a interface javax.ejb.EJBHome. A interface *Remote* deve ser pública e estender a interface javax.ejb.EJBObject. A classe abstrata do bean deve ser pública e implementar javax.ejb.EntityBean quando for um *EntityBean* e javax.ejb.SessionBean quando for um *SessionBean*.

3.5.5.3 Regra 3

Todos os métodos das interfaces *Home* e *Remote* devem possuir a exceção java.rmi.RemoteException na sua cláusula *throws*, justamente para indicar para a camada cliente quando ocorrer uma exceção no acesso remoto.

3.5.5.4 Regra 4

A interface *Home* deve possuir os métodos (*create*) para a criação do *Bean*, o qual deve ter como retorno a interface *Remote* do *Bean* e lançar a exceção javax.ejb.CreateException. No caso de *EntityBeans* é possível ainda ter métodos conhecidos como "*finders*", ou seja métodos para localizar *EntityBeans* através de um comando de seleção escrito em EJBQL, onde esse comando ficará no descritor de implantação do *EntityBean*, os métodos *finders* devem possuir na sua cláusula *throws* a exceção javax.ejb.FinderException, a qual será lançada quando houver erro na busca do *EntityBean*, sendo que a exceção javax.ejb.ObjectNotFoundException, a qual é subclasse de FinderException, será lançada quando o objeto não for encontrado. O tipo de retorno dos métodos *finders* deve ser a interface remota do *Bean* ou java.util.Collection, este quando a busca retornar mais de um *EntityBean*.

É obrigatório a definição do método findByPrimaryKey(BeanPrimaryKey pk), o qual deve retornar a interface *Remote* do *Bean*.

3.5.5.5 Regra 5

A interface *Remote* deve possuir os métodos que serão ofertados à camada cliente. Sendo que no tipo mais comum de *EntityBean*, aquele que possui a persistência gerenciada automaticamente pelo container (CMP -Container-Manager Persistence) é necessário que a interface *Remote* possua métodos abstratos *get* e *set* (seguindo o modelo *JavaBeans*) para cada atributo que será ofertado à camada cliente. Um outro modelo de *EntityBeans* é o BMP (Bean-Manager Persistence), onde o próprio Bean é responsável por persistir os seus atributos, sendo que não se abordará esse tipo de EntityBean, pois a sua implementação exige a implementação de todo o código para tratar a persistência, retirando muito a produtividade e aumentando a complexidade do desenvolvimento, sendo necessário apenas quando a fonte de dados não permitir o uso de *EntityBeans* CMP.

3.5.5.6 Regra 6

A classe abstrata do *Bean* deve possuir a implementação dos métodos que representam as regras de negócios a serem executadas pelo EJB. Sendo que no caso dos *EntityBeans* são obrigatórios os métodos set e get para todos os seus atributos, isso no caso de *EntityBeans* CMP.

Os EJB ainda precisam possuir um atributo privado do tipo javax.ejb.SessionContext para *SessionBean* e javax.ejb.EntityContext para *EntityBean*, onde este será utilizado no relacionamento entre o EJB e seu *container*, mais precisamente para representar o EJB no *pool* de EJBs do *container*.

Os *SessionBeans* implementam a interface javax.ejb.SessionBean, assim o mesmo deve implementar seus métodos, sendo eles: ejbActivate(), ejbPassivate(), ejbRemove(), setSessionContext(SessionContext p0) todos levantando as exceções EJBException e RemoteException.

Os Entity Beans implementam a interface javax.ejb.EntityBean, assim o mesmo deve implementar seus métodos, sendo eles: ejbActivate(), ejbLoad(), ejbPassivate(), ejbRemove(), ejbStore(), setEntityContext(EntityContext p0) e unsetEntityContext() todos levantando as exceções EJBException e RemoteException.

Esses métodos dizem respeito ao ciclo de vida do EJB e do seu relacionamento com o *container*. Nos *SessionBeans* e em *EntityBeans* CMP somente é necessário implementar os métodos setEntityContext e unsetEntityContext, os quais podem ser implementados sempre da mesma forma. Outro detalhe para os *EntityBeans* CMP é que a classe do *Bean* deve possuir um construtor padrão e um par de métodos (ejbCreate, e ejbPostCreate) para cada método *create* da interface *Home*. Onde o método ejbCreate deve ter como tipo de retorno a chave primária do *Bean*, mas na sua implementação deve ser retornado o valor *void*.

3.5.5.7 Regra 7

Os EntityBeans podem possuir uma classe para representar a chave primária, onde essa classe deve ser serializável (implementar a interface *Serializable*), possuir um construtor padrão e redefinir os métodos: *public int hashCode*() e *public boolean equals* (*Object* obj) herdados de java.lang.Object.

Para cada instância de *enterprise bean*, o *container* EJB gera uma instância de um objeto de contexto que mantém informações sobre o gerenciamento das regras e do estado corrente da instância, obtido originalmente do descritor de implantação. Esse objeto de contexto é utilizado tanto pelo *container* EJB quanto pelo *enterprise bean* para coordenar transações, segurança e persistência entre outros serviços.

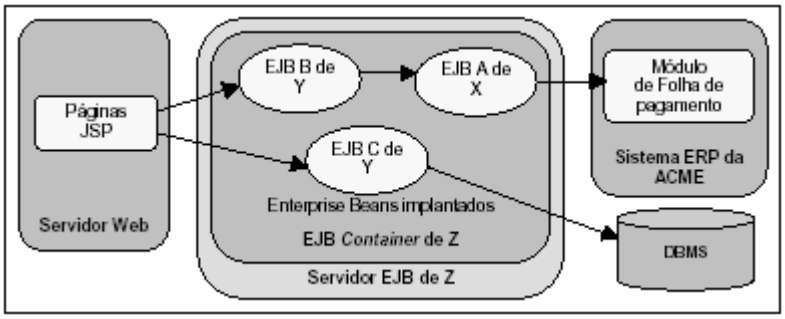

Figura 10 :Cenário: Aplicação Web para empregados de uma empresa (SOUZA, 2001).

3.5.6 Distribuição de Serviços

A especificação EJB determina o uso da API de Invocação de Método Remota Java - Remote Method Invocation (RMI) - para prover acesso aos *enterprise beans*. A RMI também suporta comunicação através do protocolo de comunicação padrão do CORBA, o Internet InterORB Protocol – IIOP. Os *enterprise beans* que se baseiam no subconjunto RMI-IIOP da RMI têm interoperabilidade com uma infra-estrutura de componentes distribuída multi-linguagem, pois qualquer cliente CORBA pode acessar *enterprise beans* e estes também podem acessar qualquer servidor CORBA (THOMAS apud SOUZA, 2001).

Quando o protocolo RMI-IIOP é usado, o cliente comunica-se com o *enterprise bean* usando *stubs* para os objetos do lado servidor. Os *stubs* implementam, ou realizam, as interfaces *Home* e *Remote* (Figura 11).

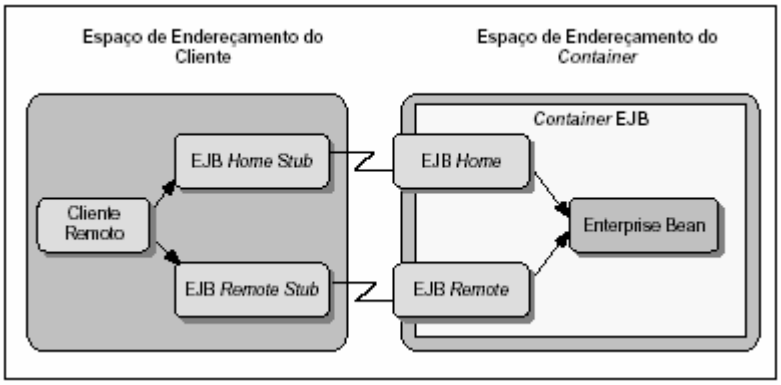

Figura 11: Localização dos stubs EJB cliente (SOUZA, 2001).

Os *stubs* de comunicação usados no lado cliente são gerados, em tempo de implantação do *enterprise bean*, pelas ferramentas de desenvolvimento do fornecedor de *container* EJB. Esses *stubs* são padrão, se o *container* usa o RMI-IIOP como protocolo de distribuição; caso contrário, são específicos do *container* em questão.

3.5.7 Restrições de Programação

O fornecedor de *Bean* deve respeitar algumas restrições de programação para garantir a portabilidade do EJB. Em resumo, um EJB NÃO deve (SUN, 2002c):

a) tentar gerenciar ou usar *threads*, pois são funções exclusivas do container EJB;
b) usar as funcionalidades da AWT (*input* / *output*), pois muitos servidores EJB não permitem interação direta entre o EJB e a console / teclado do sistema;
c) usar o pacote java.io para tentar acessar arquivos ou diretórios no sistema de arquivos – as APIs de sistema não são as mais adequadas para manipulação de dados, devendo-se preferir APIs como a JDBC;
d) tentar ouvir ou aceitar conexões em um *socket*, ou usar um *socket* para *multicast* – não é permitido que instâncias de EJB sejam servidoras de rede, pois as funções de rede são da competência do *container* EJB;
e) tentar criar um *classloader*, obter o *classloader* corrente, estabelecer o contexto do classloader, assinalar ou criar um gerente de segurança, parar a JVM (Java Virtual Machine), ou alterar os *streams* de entrada, de saída e de erro, pois essas são funcionalidades do *container* EJB;
f) tentar carregar uma biblioteca nativa (código executável específico para uma plataforma), pois essa funcionalidade é da competência do container EJB;

g) tentar acessar ou modificar os objetos de configuração de segurança – *Policy*, *Security*, *Provider*, *Signer* e *Identity*, também são funções do *container* EJB;
h) tentar passar o apontador *this* como um argumento ou resultado de um método – ao invés, deve-se usar o getEJBObject() do SessionContext ou do *EntityContext*;
i) tentar usar a API de Reflexão Java para acessar informação que as regras de segurança da linguagem de programação Java tornam indisponíveis.

Essas regras de programação garantem a portabilidade das implementações dos *enterprise* JavaBeans entre diferentes *containers*. Sua adoção não deve comprometer componentes que possuam apenas métodos referentes às regras de negócio de uma aplicação. Se o comprometimento ocorrer, a lógica do componente deve ser revista.

Concluída a revisão sobre o ambiente J2EE e EJBs, o próximo capítulo apresenta o conceito de controle de réplicas em sistemas distribuídos e os métodos que possibilitam sua implementação.

4 CONTROLE DE RÉPLICAS

4.1 REPLICAÇÃO

Um serviço de replicação de objetos permite a existência de cópias de um objeto em múltiplos nós de um sistema distribuído. A primeira motivação para a replicação é o incremento da acessibilidade a um objeto, replicando-o em todos aqueles nós em que existem seus clientes. Uma segunda motivação é a tolerância à falhas. Replicando-se um objeto em vários nós, seus serviços serão mais robustos diante de possíveis falhas em algum nó. Em ambos os casos, uma questão fundamental a resolver é a manutenção da consistência entre réplicas, quer dizer, assegurar que todas as réplicas mantêm os mesmos valores, mantendo tempos de resposta razoáveis. A motivação final para a replicação, como não podia ser de outra maneira, é a melhora de rendimento do sistema.

O requisito principal de qualquer esquema de replicação é a transparência. Os clientes não têm que saber da existência de múltiplas cópias, de tal forma que utilizarão um único nome para referir-se ao objeto replicado, independentemente da réplica concreta que vá servir suas mensagens. Outro requisito muito importante é a consistência, que já se comentou.

Na figura seguinte se mostra um modelo geral de um sistema de replicação. Os objetos cliente realizam uma série de pedidos, que podem ser de leitura ou podem atualizar informações das réplicas. Todas os pedidos dos clientes são gerenciados inicialmente por objetos *front-end*, encarregados da comunicação com as réplicas em nome dos clientes. Dependendo do esquema de replicação utilizado, os objetos *front-end* se comunicam com as réplicas de diferentes formas.

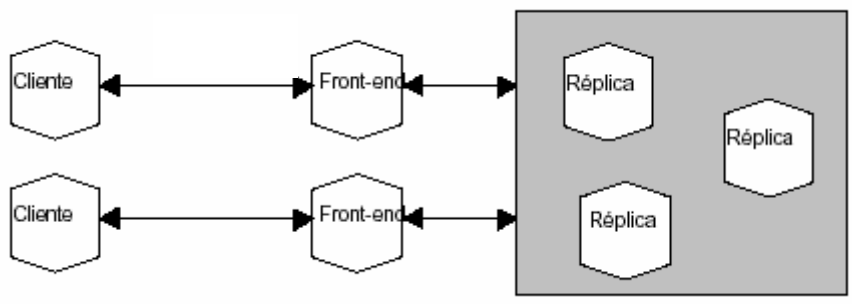

Figura 12 :Arquitetura básica de um sistema de gestão de réplicas.

O estudo de réplicas apresenta dois tipos de gerenciamento/controle de réplicas, o ativo e o passivo, a seguir ver-se-á estas duas formas.

4.2 REPLICAÇÃO ATIVA

Na replicação ativa, todas as réplicas de um componente de software trabalham em paralelo processando os valores de entrada recebidos e, na ausência de faltas, passam pelas mesmas transições de estado, produzindo saídas idênticas e na mesma ordem. As saídas de todas as réplicas são submetidas a um elemento votador, que realiza votação majoritária para decidir o resultado final do processamento. Nesse caso, é possível mascarar falhas arbitrárias (falhas Bizantinas) dos processadores. Em contrapartida, se a semântica de falha dos processadores é por parada, a replicação ativa pode ser utilizada sem votação, porque qualquer saída produzida será um valor correto. Vale salientar que, para utilizar esta estratégia de replicação faz-se necessário garantir as seguintes condições (SCHNEIDER, 1990):

a) a consistência de entrada que garanta que o conjunto de valores de entrada deve ser idêntico para todas as réplicas do mesmo objeto;
b) o determinismo do grupo de réplicas, ou seja, Partindo do mesmo estado inicial e processando o mesmo conjunto ordenado de valores de entrada, todas as réplicas corretas produzem o mesmo conjunto ordenado de valores de saída.

A primeira condição é garantida incorporando-se ao sistema de comunicação um protocolo de comunicação que assegure a atomicidade entre os receptores livres de falhas, tal como os protocolos de difusão atômica apresentada em (CHANG & MAXEMCHUK, 1984).

Já o determinismo do grupo de réplicas pode ser conseguido forçando-se o determinismo de cada réplica. Desta maneira, a estratégia de replicação ativa é baseada na abordagem da máquina de estado (SCHNEIDER, 1990).

A abordagem da máquina de estado é um método genérico para suportar tolerância a faltas através de replicação. Uma máquina de estado consiste de variáveis de estado e comandos, os quais, respectivamente, guardam e transformam o estado da máquina. Cada comando é implementado por um programa determinístico, além disso, a execução de um comando é atômica em relação a outros comandos e modificam

variáveis de estado e/ou produz alguma saída. Um cliente da máquina de estado requisita a execução de um comando.

As respostas para requisições de processamento são direcionadas a um ativador (processo de controle), periférico (terminal, disco rígido) ou ao cliente que enviou a requisição.

Uma versão tolerante a faltas de uma máquina de estado (no caso, um componente de software baseado neste modelo) pode ser implementada replicando-se aquela máquina em processadores distintos de um sistema distribuído. Se as réplicas corretas, isto é, as réplicas executando em processadores livres de falhas, partem do mesmo estado inicial e processam o mesmo conjunto ordenado de requisições, então, serão produzidas saídas idênticas para o processamento realizado por cada réplica. Quando faltas arbitrárias são consideradas, um grupo de réplicas implementando uma máquina de estado tolerante a faltas, deve conter, pelo menos, réplicas, onde qualquer número de réplicas podem falhar no sistema. Nesse caso, o resultado do processamento realizado no grupo será a saída produzida pela maioria dos seus membros. Por outro lado, se as réplicas falham, apenas, em decorrência de faltas por parada, então, é suficiente que o grupo seja composto por réplicas. Desta maneira, o resultado do processamento será a saída produzida por qualquer membro do grupo.

Para implementar uma máquina de estado tolerante a faltas faz-se necessário garantir os seguintes requisitos:

a) o acordo que todas as réplicas corretas que implementam uma máquina de estado recebam as requisições de clientes para esta máquina;
b) a ordem, todas as réplicas corretas que implementam uma máquina de estado processam as requisições recebidas na mesma ordem.

Tais requisitos determinam a consistência dentro do grupo de réplicas e devem ser atendidos por protocolos específicos, executados por cada réplica.

4.3 REPLICAÇÃO PASSIVA

Na replicação passiva, um componente de software é implementado por um grupo de réplicas em que um único elemento, a réplica primária, recebe, processa e responde a todas as requisições de clientes. Os demais membros do grupo, as réplicas suplente, só serão ativadas no caso de falha da réplica primária, com o intuito de dar continuidade ao

serviço interrompido. De acordo com esta estratégia, as réplicas devem apresentar semântica de falha por parada, desse modo, todas as saídas produzidas por uma réplica serão corretas, não havendo necessidade de utilizar nenhum mecanismo para validação de resultados.

As réplicas suplentes de um componente de software guardam uma cópia de alguns estados anteriores deste componente, a partir dos quais a execução pode ser continuada quando a réplica primária falhar. Nesse sentido, a réplica primária emite, periodicamente, uma salvaguarda para todas as réplicas suplentes a fim de atualizar os estados internos dessas réplicas.

Note que, na ausência de faltas, as réplicas suplentes não executam qualquer processamento, exceto a atualização dos seus estados internos.

Quando ocorre uma falha da réplica primária, as suplentes escolhem entre si uma réplica para assumir o papel de primária e esta reinicia a execução interrompida a partir do último ponto de salvaguarda estabelecido, ou seja, a partir do estado interno atual. Isto significa um retrocesso no estado do sistema, conseqüentemente, as ações realizadas pela réplica primária, antes de sua falha, serão executadas novamente.

Em relação à emissão de salvaguardas, se as réplicas não são determinísticas, então, a réplica primária deve emitir uma salvaguarda sempre que houver mudanças no seu estado interno. Tal procedimento não acarreta uma excessiva sobrecarga de comunicação no sistema, pois, as mudanças ocorridas na réplica primária, entre pontos de salvaguarda consecutivos, são pequenas. Por outro lado, se o determinismo das réplicas pode ser assumido, a freqüência na emissão de salvaguardas diminui. Nesse caso, as réplicas suplentes guardam as mensagens enviadas e recebidas pela réplica primária, desde o último ponto de salvaguarda estabelecido, para que, quando assumirem o papel de primária, processem as mensagens de entrada diretamente, alcançando o mesmo estado da antiga réplica primária, antes da sua falha.

Utilizando replicação passiva não se faz necessário garantir que o grupo de réplicas seja determinístico, além disso, é possível economizar a capacidade de processamento do sistema.

Porém, observam-se as seguintes restrições:

a) carga extra de comunicação (permanente), gerada pela necessidade de enviar salvaguardas para as réplicas suplentes;

b) carga extra de processamento (temporária), gerada quando uma réplica suplente assume o papel de réplica primária, tendo que executar o procedimento para recuperação de erros, já discutido. Devido à recuperação de erros, existirá um atraso no fornecimento do serviço e este atraso pode não ser compatível com os requisitos de tempo real de muitas aplicações.

5 APLICAÇÃO DA PROPOSTA

A partir da técnica que se propos neste trabalho e ainda, das tecnologias exploradas anteriormente, considerar-se-á a aplicação do serviço de replicação baseando-se nos seguintes tópicos:

a) neste trabalho utilizou-se o servidor de aplicação JBoss;
b) o servidor deverá possuir EJBs (Enterprise JavaBeans) implantados (*deployment*) e estes serão nomeados EJBs Base;
c) a replicação deverá acontecer apenas nos EJBs do tipo *Stateless* (não armazenam estados);
d) o administrador do servidor de aplicação deverá conhecer quais são os EJBs que devam ser replicados;
e) o serviço JNDI (Java Naming Directory Interface) deverá estar ativado;
f) a replicação poderá ser realizada no mesmo servidor de aplicação ou em quantos servidores forem adequados.
g) serão descritos o código e comentários das implementações;
h) aplicam-se duas técnicas de replicação, a Ativa e a Passiva.

A seguir é apresentada a metodologia de implantação da proposta:

5.1 SERVIDOR DE APLICAÇÃO

O Servidor de Aplicação é o elemento principal da aplicação distribuída J2EE, portanto é de fundamental importância o domínio desta. Para este experimento utilizou-se o JBoss.

JBoss é uma execução da especificação de EJB 1.1 (e partes de 2.0), isto é, é um servidor e um *container* para *Enterprise JavaBeans*. Nisto é similar à edição do J2SDK *Enterprise Edition* (J2EE) da empresa Sun, mas o núcleo do JBoss fornece somente um servidor de EJB. O núcleo de JBoss não inclui um *container* Web para páginas de servlets/JSP, embora haja os pacotes disponíveis que incluem Tomcat ou Jetty, que são *containers* Web.

Por causa do uso de pouca memória, JBoss inicializa aproximadamente 10 vezes mais rapidamente do que J2EE.

Há um servidor interno de base de dados SQL para assegurar *beans* persistentes, e este inicializa automaticamente com o servidor.

Uma das melhores características do JBoss é seu suporte para a implantação à quente (*hot deployment*). Isto significa que distribuir um *bean* é simplesmente copiar o arquivo JAR, WAR ou EAR que contem o *bean* no diretório da distribuição. Se isto for feito quando o bean já estiver carregado, JBoss descarrega-o automaticamente e carrega então a versão nova.

JBoss é distribuído sob a LGPL (Lesser General Public License)[2], que significa que está livre, padrão para o trabalho comercial, e o LGPL se assegura de que permaneça dessa maneira (JBOSS, 2003a).

Uma das informações relevantes para utilizar-se este servidor é, segundo JBOSS (2003b), de que no primeiro dia de janeiro de 2003 o número acumulado de *downloads* registrados em 2002 foi de 2.064.141. Isto mostra a tendência deste tornar-se o padrão entre os servidores de aplicação.

O fato de utilizar-se o JBoss não descaracteriza o recurso de replicação em outros servidores, apenas os nomes dos arquivos descritores ou o DTD (Document Type Definition) "podem" ser diferentes para cada caso em particular e também porque o seu *container* é totalmente compatível com padrão.

5.2 O DIRETÓRIO DE DIST. E OS ARQ. DE IMPLANTAÇÃO

Cada servidor de aplicação possui sua própria forma de gerenciar a implantação de beans, mas todos eles apresentam em comum um diretório onde as aplicações distribuídas são implantadas. No JBoss 3.0 o diretório de implantação pode ser <*JBOSS_DIST*>*server****all***deploy* ou <*JBOSS_DIST*>*server****default***deploy* ou ainda, <JBOSS_DIST>\server**minimal**\deploy sendo utilizada a configuração na qual o servidor foi inicializado. Este servidor pode ser inicializado em uma das três formas: *all*, *default* ou *minimal*. O modo de inicialização está ligado diretamente aos serviços a serem disponibilizados. Para se inicializar o servidor executa-se o comando run –c <all>/<default>/<minimal>.

[2] A licença LGPL está disponível para consulta em http://www.gnu.org/licenses/lgpl.html .

Uma aplicação J2EE é composta por módulos de componentes do cliente da aplicação, *enterprise beans* e ainda componentes Web. Nesta, cada um de seus módulos possuem seus próprios descritores de implantação. (BODOFF, 2002)

Um descritor de implantação é um documento XML (*Extensible Markup Language)* com uma extensão xml que descreve as configurações de implantação de um componente. Uma aplicação J2EE com todos os seus módulos é entregue em um arquivo EAR (Enterprise ARchive), este padrão de arquivo é compactado. Um arquivo EAR normalmente é composto de um arquivo JAR (Java Archive), um arquivo WAR (Web Archive) e uma pasta interna chamada META-INF. É nesta pasta que está o arquivo application.xml o qual é o descritor do conteúdo do arquivo EAR. Abaixo, na figura 13, apresenta-se um exemplo de código da aplicação primerEJB:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application>
  <display-name>Primer EJB</display-name>
  <module>
    <web>
      <web-uri>primerEJB.war</web-uri>
      <context-root>/primerEJB</context-root>
    </web>
  </module>
  <module>
    <ejb>primerEJB.jar</ejb>
  </module>
</application>
```

Figura 13 : Código fonte do arquivo application.xml

No exemplo acima pode-se observar a descrição da aplicação chamada Primer EJB, ela é composta por dois módulos, os arquivos primerEJB.war e o arquivo primerEJB.jar. Chama-se de módulo porque estes são arquivos que contem outros arquivos e também estão fisicamente compactados.

Dentro dos módulos JAR e WAR existem pastas de descritores e outros arquivos como demonstrado na figura 14 abaixo:

```
primerEJB.jar
        META-INF
                jboss.xml
                ejb-jar.xml
        pacote_classes
primerEJB.war
        WEB-INF
                pacote_classes
        index.jsp
```

Figura 14 : Estrutura dos arquivos ear, composto por jars e wars.

Os arquivos descritores jboss.xml e ejb-jar.xml para o exemplo, são também apresentados nas figuras 15 e 16 abaixo:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
        "http://www.jboss.org/j2ee/dtd/jboss.dtd">
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>Primer</ejb-name>
      <jndi-name>ejb/Primer</jndi-name>
    </session>
  </enterprise-beans>
</jboss>
```

Figura 15 : Código fonte do arquivo jboss.xml

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems,
   Inc.//DTD Enterprise JavaBeans 2.0//EN"
        "http://java.sun.com/dtd/ejb-jar_2_0.dtd">
<ejb-jar>
  <description></description>
  <enterprise-beans>
    <session>
      <display-name>PrimerEJB</display-name>
      <ejb-name>Primer</ejb-name>
      <home>pacote.PrimerHome</home>
      <remote>pacote.Primer</remote>
      <ejb-class>pacote.PrimerEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

Figura 16 : Código fonte do arquivo ejb-jar.xml

Verifica-se a partir da figura 16 que o arquivo ejb-jar.xml descreve o *enterprise bean*, indicando um nome para ele, a classe *home*, a classe *remote* e também a classe deste. O marcador <*session-type*> indica ao servidor que tipo de EJB é este.

Na figura 15, o arquivo jboss.xml providencia ao *bean* um nome de entrada no JNDI.

Para disponibilizar então uma aplicação J2EE coloca-se o arquivo primerEJB.ear do exemplo, no diretório de implantação adequado e, automaticamente, o servidor reconhece e instala a aplicação.

Então, para que se saiba quais são os EJB *Stateless* que se possa replicar basta selecionar no diretório de implantação os arquivos EAR e dentro deles investigar o arquivo ejb-jar.xml.

5.3 SERVIÇO JNDI

No JBoss 3.0, quando o servidor é inicializado no modo *all* ou *default* o serviço já estará disponível, no modo *minimal*, isto não ocorre. Portanto, deve-se fazer a opção por um destes dois modos.

5.4 FERRAMENTA DE IMPLANTAÇÃO DE RÉPLICAS

Este aplicativo foi desenvolvido em Java e fará a replicação dos EJBs conforme a configuração feita pelo montador da aplicação, que deverá fornecer a informação dos EJBs que irão ser replicados, quantas réplicas devam ser criadas e em quais servidores da rede serão implantados.

Através deste aplicativo será gerada uma metaclasse responsável pelo controle das réplicas utilizando um *template* desenvolvido para esta tarefa em conjunto com a API de reflexão computacional disponibilizada pela Sun. Além disto, distribui cópias do EJB a ser replicado nos servidores designados pelo montador da aplicação. Caso o montador da aplicação escolha replicar mais de uma vez o EJB no mesmo servidor, este irá gerar cópias desta classe com nomes diversos, um novo nome para cada cópia.

Para que o cliente da aplicação possa utilizar a nova metaclasse, que possibilita o gerenciamento das réplicas, será então feita uma alteração no nome JNDI do componente implantado fazendo com que o cliente quando chame o EJB Base, na verdade, localize a metaclasse deste.

5.5 CONVERSORMOEDAS UM ESTUDO DE CASO

Para avaliar o comportamento, no ambiente J2EE, desta proposta de implementação, se utiliza um EJB do tipo *SessionBean* com três métodos apenas, este é um EJB que realiza a conversão de moedas: Real para Dólar, Real para Euro e Dólar para Real. Nas figuras 17, 18, 19, 20, 21 e 22 abaixo se apresenta a estrutura e o código fonte de um EJB implantado no servidor:

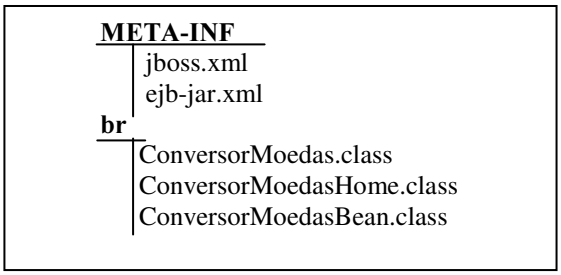

Figura 17 : Estrutura do arquivo ConversorMoedas.jar .

Na estrutura da figura 17 pode-se verificar que a aplicação possui: o EJB, com o *bean* ConversorMoedasBean.class, sua interface *remote* ConversorMoedas.class e a interface *home* ConversorMoedasHome.class além dos descritores jboss.xml e ejb-jar.xml .

```
<?xml version="1.0"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN"
"http://java.sun.com/dtd/ejb-jar_2_0.dtd">
        <ejb-jar>
          <description></description>
          <enterprise-beans>
            <session>
              <display-name>ConversorMoedasBase</display-name>
              <ejb-name>ConversorMoedas</ejb-name>
              <home>br.anibal.ConversorMoedasHome</home>
              <remote>br.anibal.ConversorMoedas</remote>
              <ejb-class>br.anibal.ConversorMoedasBean</ejb-class>
              <session-type>Stateless</session-type>
              <transaction-type>Bean</transaction-type>
            </session>
          </enterprise-beans>
</ejb-jar>
```

Figura 18 : Conteúdo do arquivo descritor ejb-jar.xml .

A figura 18 informa ao servidor de aplicação as classes que compõe esta aplicação e na figura 19 abaixo se mostra como configurar o JNDI (*Java Naming Diretory Interface*) atribuindo um nome válido “ejb/ConversorMoedas” para o *bean*.

```
1:<?xml version='1.0' encoding='UTF-8' ?>
2:<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
"http://www.jboss.org/j2ee/dtd/jboss.dtd">
3:      <jboss>
4:              <enterprise-beans>
5:                      <session>
6:                              <ejb-name>ConversorMoedas</ejb-name>
7:                              <jndi-name>ejb/ConversorMoedas</jndi-name>
8:                      </session>
9:              </enterprise-beans>
10:     </jboss>
```

Figura 19 : Conteúdo do arquivo descritor jboss.xml .

Na figura 20 será apresentada a interface *remote* e na figura 21 a interface *home*.

```
package br.anibal;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface ConversorMoedas extends EJBObject {
  public Float real_dolar(Float real) throws RemoteException;
  public Float real_euro(Float real) throws RemoteException;
  public Float dolar_real(Float real) throws RemoteException;
}
```

Figura 20 : Conteúdo da interface *remote*, ConversorMoedas.java .

```
package br.anibal;
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
public interface ConversorMoedasHome extends EJBHome {
   ConversorMoedas create() throws RemoteException, CreateException;
}
```

Figura 21 : Conteúdo da interface *home*, ConversorMoedasHome.java .

```
package br.anibal;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
public class ConversorMoedasBean implements SessionBean {
    Float taxa_dolar = new Float("3.9000");
    Float taxa_euro = new Float("4.2000");
  public Float real_dolar(Float real) {
    return new Float(real.floatValue()/taxa_dolar.floatValue());
  }
  public Float real_euro(Float real) {
    return new Float(real.floatValue()/taxa_euro.floatValue());
  }
  public Float dolar_real(Float real) {
    return new Float(taxa_dolar.floatValue()*real.floatValue());
  }
  public ConversorMoedasBean() {System.out.println("Construtor");}
  public void ejbCreate() {System.out.println("Construtor EJB");}
  public void ejbRemove() {System.out.println("Remove EJB");}
  public void ejbActivate() {System.out.println("Activate EJB");}
  public void ejbPassivate() {System.out.println("Passivate EJB");}
  public void setSessionContext(SessionContext sc) {System.out.println("Contexto");}
}
```

Figura 22 : Código fonte do EJB - arquivo ConversaoMoedasBean.java

Além do arquivo ConversorMoedas.jar, que contem a camada EJB ou de negócios, para que o cliente possa executar remotamente esta aplicação foi criado o arquivo ConversorMoedas.war, que contem a camada WEB e apresenta a seguinte estrutura:

Figura 23 : Estrutura do arquivo ConversorMoedas.war .

Os arquivos com extensão *class,* da figura 23, são as mesmas interfaces que também estavam disponíveis no arquivo ConversorMoedas.jar, através delas é que o cliente faz chamadas remotas ao EJB. A figura 24 é o código fonte instalado no container web que o cliente acessa a partir de seu navegador.

```
1: <%@ page import="br.anibal.*,javax.ejb.*, java.math.*, javax.naming.*, javax.rmi.PortableRemoteObject,
java.rmi.RemoteException" %>
2: <%!
3:   private ConversorMoedas conversor_moedas_jsp = null;
4:   public void jspInit() {
5:     try {   InitialContext ic = new InitialContext();
6:       Object objRef = ic.lookup("ejb/ConversorMoedas");
7:       ConversorMoedasHome home = (ConversorMoedasHome)PortableRemoteObject.narrow(objRef,
ConversorMoedasHome.class);
8:       conversor_moedas_jsp = home.create();
9:     } catch (RemoteException ex) {
10:          System.out.println("Não pude criar o bean !!!!!!!!!!!!!!!!!!!."+ ex.getMessage());
11:    } catch (CreateException ex) {
12:          System.out.println("Não pude criar o bean !!!!!!!!!!!!!!!!!!!."+ ex.getMessage());
13:    } catch (NamingException ex) {
14:          System.out.println("Não encontrei o nome: "+ "ConversorMoedas "+ ex.getMessage());
15:    } catch (Exception ex) {
16:         ex.printStackTrace();  }  }
17:  public void jspDestroy() {
18:       conversor_moedas_jsp = null;
19:  }  %>  <html>
20: <head>  <title>Conversor de moedas</title>  </head>
21: <body bgcolor="white">  <h1><b><center>Conversor de Moedas</center></b></h1>
22: <hr><p>Entre com um valor para conversão:</p>
23: <form method="get">
24:         <input type="text" name="quantia" size="25"> <br><p>
25:         <input type="submit" value="Calcular">
26:         <input type="reset" value="Limpar">
27: </form>
28: <%    String quantia = request.getParameter("quantia");
29:   if ( quantia != null && quantia.length() > 0 ) {   Float d = new Float (quantia);   %>
30:   <p>   <%= quantia %> Reais são  <%= conversor_moedas_jsp.real_dolar(d) %>  Dólares.
31:  <p>   <%= quantia %> Reais são <%= conversor_moedas_jsp.real_euro(d) %>  Euros.
32:  <p>   <%= quantia %> Dolares são <%= conversor_moedas_jsp.dolar_real(d) %>  Reais.   <%    }   %>
33:  </body>
34:  </html>
```

Figura 24 : Código fonte do programa index.jsp .

O sistema apresentado na figura 25 demonstra bem o funcionamento da aplicação distribuída no ambiente proposto. Qualquer estação de trabalho que esteja conectada à rede e contenha um navegador web poderá ser cliente da aplicação, apenas fornecendo ao navegador a URL *(Uniform Resource Locator)* do servidor e o contexto da aplicação no container web onde a aplicação está hospedada. Como exemplo de URL, no caso da situação apresentada na figura 25, pode-se apresentar as seguintes possibilidades: http://A/ConversorMoedas onde o "A" é o nome do servidor na rede, ou ainda, poder-

se-ia indicar a URL com o IP (Internet Protocol) do servidor. No caso do cliente acessar de fora da rede local, e existir um domínio válido na Internet, este domínio também poderá ser fornecido.

Na máquina "servidora A" será executado o arquivo index.jsp, o qual através do serviço de *lookup*, disponibilizado pelo pacote javax.naming, irá construir e fazer chamadas ao *bean* através das suas interfaces *home* e *remote* acessando assim seus recursos disponíveis.

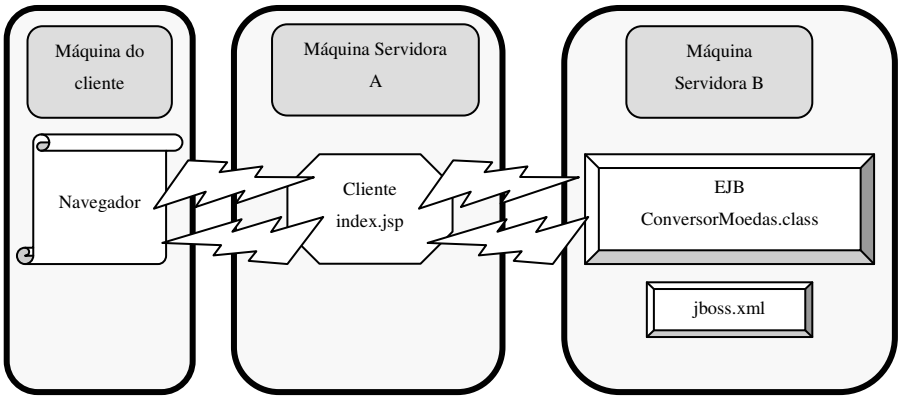

Figura 25 : Esquema de Funcionamento da Aplicação Base a ser Replicada.

5.6 GERENCIAMENTO DAS RÉPLICAS

Para que a implementação acima possa atender a especificação desejada e ainda conte com o serviço de replicação, na seqüência abaixo aplicam-se as duas formas estudadas anteriormente.

5.6.1 Replicação Ativa

Neste modelo de gerenciamento, como visto anteriormente, todas as réplicas recebem a mensagem de entrada de forma paralela, processam o serviço solicitado e passam o resultado gerado a um objeto moderador ou árbitro, que receberá estes resultados e votará qual dos resultados deverá ser devolvido àquela solicitação efetuada no início.

O modelo de implementação de replicação proposto, utilizando a técnica de replicação ativa é representado na figura 25 abaixo:

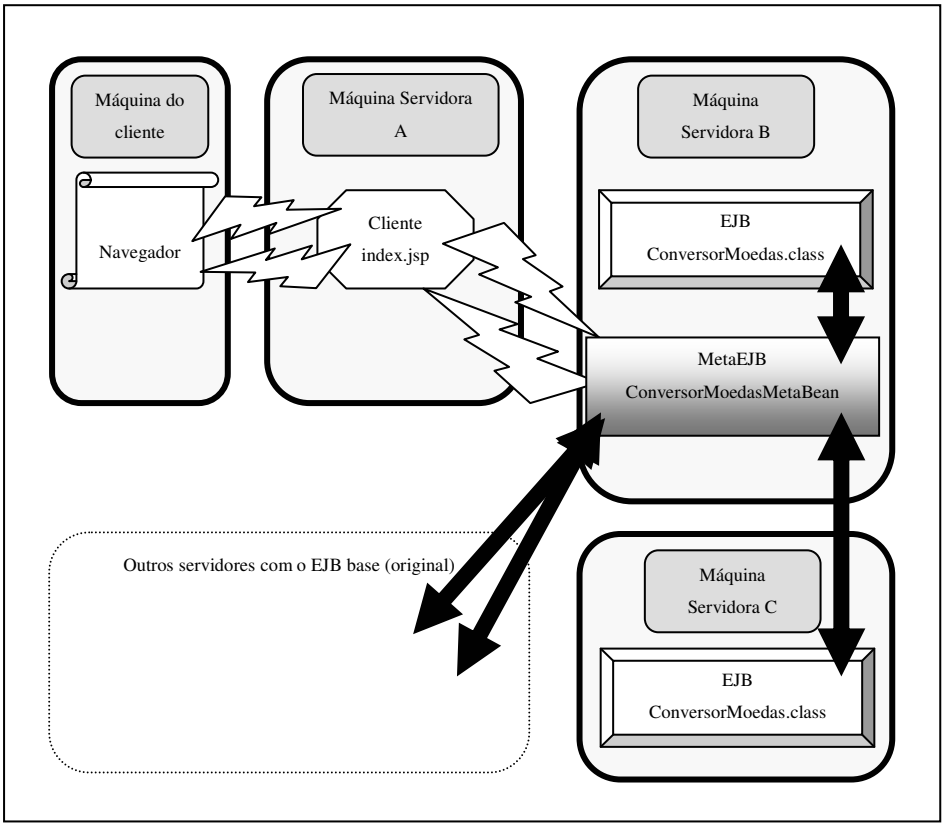

Figura 26 : Implantação de réplicas em diversos servidores, replicação ativa.

Levando-se em consideração que o EJB tem associado a ele um arquivo jboss.xml e que este referencia o EJB para o serviço JNDI como “ejb/ConversorMoedas”, linha7 da figura 19, e este não deva ser mais alcançado pelo cliente web, altera-se neste arquivo o nome JNDI, ou seja, na linha 7 da figura 19 substitui-se o nome que era referenciado pelo cliente, linha 6 da figura 24, “ejb/ConversorMoedas” por “ejb/ConversorMoedasBase” (linha 7 da figura 27) ou seja, o cliente continuará a fazer “lookup” ao mesmo nome não requerendo assim nenhum tipo de alteração em seu código fonte.

A metaclasse gerada para coordenar as réplicas então é que terá no seu arquivo jboss.xml referenciado seu nome como “ejb/ConversorMoedas” como se pode ver na figura 28, logo abaixo.

A partir do EJB e através da reflexão computacional, gera-se a metaclasse com todas as definições de seus métodos, e na implementação destes é que serão feitas as chamadas às réplicas.

```
1:<?xml version='1.0' encoding='UTF-8' ?>
2:<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
"http://www.jboss.org/j2ee/dtd/jboss.dtd">
3:      <jboss>
4:              <enterprise-beans>
5:                      <session>
6:                              <ejb-name>ConversorMoedas</ejb-name>
7:                              <jndi-name>ejb/ConversorMoedasBase</jndi-name>
8:                      </session>
9:              </enterprise-beans>
10:     </jboss>
```

Figura 27: Arquivo jboss.xml do EJB que estava disponível no servidor.

```
1:<?xml version='1.0' encoding='UTF-8' ?>
2:<!DOCTYPE jboss PUBLIC "-//JBoss//DTD JBOSS//EN"
"http://www.jboss.org/j2ee/dtd/jboss.dtd">
3:      <jboss>
4:              <enterprise-beans>
5:                      <session>
6:                              <ejb-name>ConversorMoedasMeta</ejb-name>
7:                              <jndi-name>ejb/ConversorMoedas</jndi-name>
8:                      </session>
9:              </enterprise-beans>
10:     </jboss>
```

Figura 28: Arquivo jboss.xml do MetaEJB que será implantado no servidor.

Em cada método da metaclasse serão executadas chamadas ao método de mesmo nome dos EJBs, distribuídos como réplicas nos outros servidores. E assim, a cada chamada, haverá o retorno armazenado em uma variável local. Se houver falha no retorno, por queda do servidor de aplicação ou então por falha de comunicação, a chamada será "cancelada" por *timeout*. Como podem existir diversas réplicas no servidor, o critério para determinar qual é a resposta correta, entre as que retornaram, é a do maior número de respostas iguais, e esta então retorna ao cliente.

Na figura 29 abaixo se apresenta o código fonte gerado através da reflexão computacional e *templates* para a metaclasse do Conversor de Moedas.

```
package br.anibal;
import java.rmi.RemoteException;   import javax.ejb.SessionBean;
import javax.ejb.SessionContext;   import javax.naming.*;
import javax.rmi.PortableRemoteObject;   import br.anibal.*;   import java.util.*;
public class ConversorMoedasMetaBean implements SessionBean {
     Float taxa_dolar = new Float("3.9000");
     Float taxa_euro = new Float("4.2000");
  public Float real_dolar(Float real) {
     Float retorno_server1 = new Float("0");
     Float retorno_server2 = new Float("0");
     try {
               ConversorMoedasHome home=Conexao("jnp://A:1099","ejb/ConversorMoedasBase");
               ConversorMoedas conversor_moedas_server1 = home.create(); //acesso remoto ao bean
               retorno_server1 = conversor_moedas_server1.real_dolar(real);
     }catch (Exception e){
               retorno_server1.valueOf("0");
     }
     try {
               ConversorMoedasHome home=Conexao("jnp://B:1099","ejb/ConversorMoedasBase");
               ConversorMoedas conversor_moedas_server2 = home.create(); //acesso remoto ao bean
               retorno_server2 = conversor_moedas_server2.real_dolar(real);
     }catch (Exception e){
               retorno_server2.valueOf("0");
     }
    if (retorno_server1.equals(retorno_server2))
                  return retorno_server1;
    else {
       if (retorno_server1.floatValue() != 0.0f)
               return retorno_server1;
       if (retorno_server2.floatValue() != 0.0f)
               return retorno_server2;
       return new Float("0"); }   }
  // outros métodos não estão representados aquí
  //
  public ConversorMoedasHome Conexao(String servidor, String nome_jndi) {
               Properties props = new Properties();
               props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
               try{
               //instanciando o EJB Base
               props.put(Context.PROVIDER_URL, servidor); //aqui é adicionado o nome do servidor
               InitialContext ic = new InitialContext(props);
               Object objRef = ic.lookup(nome_jndi);
               return (ConversorMoedasHome)PortableRemoteObject.narrow(objRef, ConversorMoedasHome.class);
               }catch(Exception ex){
               System.out.println("O servidor: "+servidor+" ou o nome: "+nome_jndi+" Não foram encontrados.");
                        return null;    }
               }
               public ConversorMoedasMetaBean() {System.out.println("Construtor Meta");
               }
               public void ejbCreate() {System.out.println("Construtor Meta EJB");}
               public void ejbRemove() {System.out.println("Remove Meta EJB");}
               public void ejbActivate() {System.out.println("Activate Meta EJB");}
               public void ejbPassivate() {System.out.println("Passivate Meta EJB");}
               public void setSessionContext(SessionContext sc) {System.out.println("Contexto Meta");
            }
}
```

Figura 29: Código fonte da metaclasse gerada para ser o gerenciador de réplicas ativa.

Então, através do código gerado acima e também da adição dos arquivos descritores jboss.xml e ejb-jar.xml, constrói-se um novo arquivo com a extensão EAR que deve ser implantado no servidor base, ou seja, neste exemplo o servidor B e nos outros servidores se implantada a réplica do EJB.

5.6.2 Replicação Passiva

Como visto no item 4.3 o gerenciamento, na replicação passiva, consiste em tornar uma delas a "réplica primária" e as outras somente serão executadas caso esta não possa ser acessada e, neste caso, se deve eleger uma nova réplica primária. Como esta proposta trata apenas de replicar EJBs do tipo *SessionBean Stateless*, a obrigatoriedade de atualização dos estados armazenados no EJB, de todas as outras réplicas, não é implementada.

Para gerenciar estas réplicas a metaclasse gerada deverá implementar um método responsável pela eleição da réplica ativa. A eleição ocorre de forma simples, é seqüencial (linear), a cada falha de *lookup* de uma réplica um índice é incrementado fazendo com que o gerenciador assuma que a réplica primária é a próxima da lista até que alcançando novamente o final da lista retorne ao primeiro elemento e assim por mais uma vez e, se não for encontrada nenhuma réplica retorna-se ao cliente um valor *default*.

Acessando-se a réplica primária através da chamada dos métodos desta pelo gerenciador, obtém-se então a execução solicitada pelo cliente. Caso a execução não tenha retorno em função de uma falha por parada durante a execução, uma nova chamada à outra réplica deverá acontecer, já que o *timeout* programado no *container* irá interromper esta espera .

Na figura 30 a seguir é mostrado o código fonte do gerenciador de réplicas para o EJB apresentado como estudo de caso.

```
package br.anibal;
import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import br.anibal.*;
import java.util.*;
public class ConversorMoedasMetaBean implements SessionBean {
  Float taxa_dolar = new Float("3.9000");
  Float taxa_euro = new Float("4.2000");
  String nome_jndi = new String("ejb/ConversorMoedasBase");
  static Vector lista_servidor = new Vector();
  static int indice = 0;
  public Float real_dolar(Float real) {
              int contador_tentativas_local = 0;
    do {    try {
                    ConversorMoedasHome home=Conexao(indice);
                    ConversorMoedas conversor_moedas_server = home.create(); //acesso remoto ao bean
                    Float resultado = conversor_moedas_server.real_dolar(real);
                    return  resultado;
            }catch (Exception e){
                    if (++indice > lista_servidor.size()) indice = 0;
                    if (++contador_tentativas_local <= 1) continue;
              return new Float("0");
      }
    } while (true);
  }
  public Float real_euro(Float real) {// código equivalente ao do método anterior – só a chamada ao método é que muda
  }
  public Float dolar_real(Float real) {// código equivalente ao do método anterior – só a chamada  ao método é que muda
  }
  public ConversorMoedasHome Conexao(int indice) {
   int contador_tentativas = lista_servidor.size()*2;
   Properties props = new Properties();
   props.put(Context.INITIAL_CONTEXT_FACTORY, "org.jnp.interfaces.NamingContextFactory");
   do {
      try{
          props.put(Context.PROVIDER_URL, lista_servidor.elementAt(indice));
          InitialContext ic = new InitialContext(props);
          Object objRef = ic.lookup(nome_jndi);
          return (ConversorMoedasHome)PortableRemoteObject.narrow(objRef, ConversorMoedasHome.class);
       }catch(Exception ex){
          if (--contador_tentativas==0) break;
          if (++indice > lista_servidor.size()) indice = 0;
       }
   } while (true);
    return null;
  }
  public ConversorMoedasMetaBean() {
              lista_servidor.addElement("jnp://A:1099");
              lista_servidor.addElement("jnp://B:1099");
  }
  public void ejbCreate() {System.out.println("Construtor Meta EJB");}
  public void ejbRemove() {System.out.println("Remove Meta EJB");}
  public void ejbActivate() {System.out.println("Activate Meta EJB");}
  public void ejbPassivate() {System.out.println("Passivate Meta EJB");}
  public void setSessionContext(SessionContext sc) {System.out.println("Contexto Meta");}
}
```

Figura 30: Código fonte da metaclasse gerada para ser o gerenciador de réplicas passivo.

5.6.3 Observações Sobre a Implementação

Este tópico discute algumas citações no referencial teórico e sua devida implementação no ambiente J2EE.

No modelo apresentado segundo Chang e Maxemchuk (1984) deve ser feito um multicast com garantia de entrega para todos os elementos replicados e estes devem recebê-la na mesma ordem temporal. Para isto se envia mensagens com seqüenciadores temporais destinadas às réplicas e aguardam-se *acknowledges* dessas. Realiza-se depois a chamada aos métodos adequados de forma a fazê-lo apenas às réplicas alcançadas. Como esta proposta é controlar as falhas por parada do serviço e, não se deve modificar a classe base e também não se pode utilizar em um EJB *threads*, este modelo apresentado aqui usa o *lookup*, que é gerenciado pelo *container* para garantir a existência da réplica em cada um dos servidores previamente informados à classe do gerenciador e aí é enviada a mensagem a ser executada pela réplica, caso não seja encontrado o servidor o container levanta uma exceção . O *container* garante também que se a mensagem não retornar uma resposta num tempo devidamente pré-programado, levanta uma exceção por *timeout*.

Como o ambiente atende requisições de múltiplos clientes o problema da concorrência deve ser observado. Em um EJB ou servlet, para cada usuário que solicita a execução do componente o *container* cria uma *thread* de execução deste, portanto, para a manipulação de dados através de variáveis estas devem ser criadas localmente ao método.

5.6.4 Experimento e coleta de dados

Para registrar o desempenho do modelo proposto foi criado um software cliente para ser executado em máquinas remotas, que executa 100 (cem) vezes a chamada aos métodos do EJB, medindo assim a latência dessas chamadas. Para tabelar esta informação foi aplicada o cálculo da média aritmética nos valores de latência. Foram feitas avaliações na aplicação original, na aplicação com o gerenciador de réplicas ativa e com o gerenciador de réplicas passivas.

A rede onde foram implementados os testes possui as seguintes características: máquinas servidoras e clientes com CPU Intel Pentium 4 de 1,6MHz com memória de

128M bytes, placas de rede e *Switch* 3Com com portas de 100M bits e sistema operacional Windows NT Workstation 4.0.

A falta simulada no cliente foi o desligamento da energia elétrica da máquina servidora.

Tabela 1: Execução da proposta sem falta nos servidores.

Quantidade de Servidores	**Latência Rep. Ativa**	**Latência Rep. Passiva**
Apenas o servidor original	Só a primeira vez foi 20ms, todas as outras 99 vezes foram menores que 1 ms.	
Base + 1 Réplica	48	45
Base + 2 Réplicas	83	45
Base + 3 Réplicas	121	45
Base + 4 Réplicas	159	45

Tabela 2: Execução da proposta com faltas nos servidores.

Quantidade de Servidores	**Latência Rep. Ativa**	**Latência Rep. Passiva**
Base + 1 Réplica c/ falta	1706	1707
Base + 2 Réplicas c/ 1 falta	1712	1763
Base + 2 Réplicas c/ 2 faltas	3416	1775
Base + 3 Réplicas c/ 1 falta	1733	1745
Base + 3 Réplicas c/ 2 faltas	3451	1762
Base + 3 Réplicas c/ 3 faltas	6920	1780
Base + 4 Réplicas c/ 1 falta	1782	1752
Base + 4 Réplicas c/ 2 faltas	3416	1772
Base + 4 Réplicas c/ 3 faltas	6910	1785
Base + 4 Réplicas c/ 4 faltas	13920	1792

Analisando-se as informações colhidas pela execução do software vê-se claramente que a aplicação do método de replicação passiva se mantém com pouca variação em cada uma das tabelas, não ocorrendo este efeito durante a aplicação do método da replicação ativa.

Para melhor visualizar esta informação, construiu-se o gráfico da figura 31 abaixo.

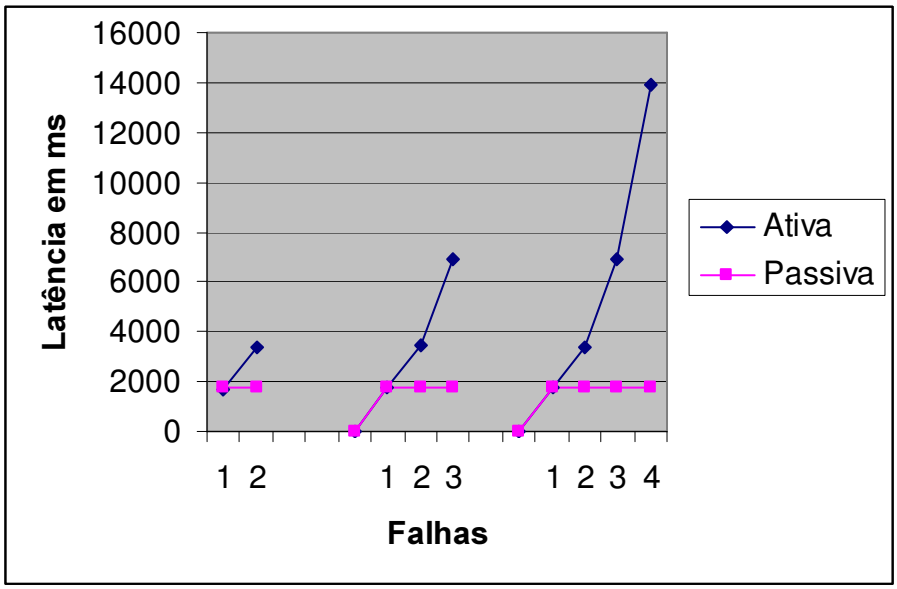

Figura 31: Desempenho das réplicas conforme número de falhas

Para a replicação ativa tem-se que quando aumenta o número de servidores com réplicas do EJB o tempo de latência aumenta de forma geométrica para cada servidor que falha, piorando muito o desempenho, mas dá garantias que a execução acontece.

Na replicação passiva observa-se que independentemente do número de servidores replicados a latência permanece praticamente constante. Observou-se também que a seqüência das máquinas que falhavam alteravam o tempo de latência, ou seja, depende a posição em que elas se encontravam na fila do gerenciador de réplicas.

6 CONCLUSÕES

No capítulo anterior aplicou-se em uma plataforma distribuída dois modelos de controle que eram uma hipótese e a motivação para este trabalho. Neste capítulo se apresenta algumas considerações sobre o desenvolvimento do trabalho e sugere-se complementações de estudos para ampliar o escopo deste.

Quanto ao objetivo principal de aplicar e avaliar o conceito do controle de réplicas neste sistema distribuído, viu-se ser possível e que as teorias clássicas de tolerância a falhas podem ser aplicadas com algumas restrições impostas pela plataforma. A avaliação dos resultados demonstra mais uma vez que estes métodos causam um grande *overhead* ao sistema e que a replicação passiva, para o escopo deste trabalho é a melhor solução, já que este modelo depende totalmente do *container* fornecido pela plataforma.

Foi desenvolvido o meta-componente para prover controle de réplicas ativa e passiva para esta plataforma, garantindo a transparência ao usuário tanto durante seu uso quanto na proteção do seu código fonte pois este não é alterado e ainda, é um *Enterprise JavaBean* que foi construído segundo as regras definidas na especificação própria, isto garante a portabilidade deste componente para execução em todos os *containers* certificados pelo padrão.

Os valores medidos de latência faz surgir a expectativa de novos trabalhos na área para que se traga à luz novos modelos mais eficientes de implementação para este problema nesta plataforma.

Como recomendação para trabalhos futuros, além de propor meta-objetos que melhorem o desempenho do serviço deve-se estender os serviços de replicação para os outros tipos de componentes da plataforma e ainda garantir que o cliente possa alcançar o serviço, já que se o servidor que abriga o meta-componente parar, as outras réplicas não serão alcançadas e portanto o serviço fica indisponível.

Se novos trabalhos forem sendo incorporados a este, o conjunto dessas soluções fará com que se tenha de fato serviços replicados ou seja, tolerante a falhas por parada.

REFERÊNCIAS BIBLIOGRÁFICAS

ARNOLD, Ken; GOSLING, James. Programando em Java. Makron Books, São Paulo, 1997.

BAN, B. JavaGroups user's guide. Department of Computes Science, Cornell University. August 1999. 73p. http://JavaGroups.sourceforge.net/

BASS, L.; PAUL, C.; KAZMAN, R. Software Archithecture in Practice. Addison-Wesley 1998.

BODOFF, Stephanie. J2EE:Tutorial. Rio de Janeiro: Campus, 2002. pg 10.

CHANG, J.;MAXEMCHUK, N.. Reliable Broadcast Protocols. ACM Transactions on Computer Systems 2(3): 251-273, 1984.

COULOURIS G.; DOLLIMORE, J.; KINDBERG, T.. Distributed Systems: Concepts and Design . Addison Wesley, Terceira Edição. 2001.

DEITEL, H. M.; DEITEL, P. J.; STEELE, Laura C.. Java: How to Program. Prentice Hall, New Jersey, 1998.

DEMICHIEL, Linda G.. Enterprise JavaBeansTM Specification,Version 2.1. Sun Mirosystems. 2002. (arquivo ejb-2_1-pfd-spec.pdf)

FABRE, J. V. Nicomette, T. Pérennou, R. J. Stroud and Z. Wu. Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming. Proceedings of the 25th IEEE International Symposium on Fault-Tolerant Computing, Pasadena (CA), June 1995.

FRAGA, Joni; C. Maziero; Lau L. and O. Loques Implementing Replicated Services in Open Systems Using a Reflective Approach, Proceedings of the 3th IEEE International Symposium on Autonomous Descentralized Systems - ISADS 97, Berlin - Germany, April 1997.

FRAGA, Joni, Lung, Lau Cheuk, Westphall, Carla Merkle, Montez, Carlos Barros Livro Texto dos Minicursos SBRC2001, Capítulo 2, Publicado pela SBC/LARC, p.45-88. Maio de 2001.

GUERRAOUI, R.; SCHIPER, A.. Software-Based Replication for Fault Tolerance. IEEE Computer, 40(4), pp. 68-74, April, 1997.

GUERRAOUI, R. Schiper, A. Fault-Tolerance by Replication in Distributed Systems. In Proc. Conference on Reliable Software Technologies (invited paper), p. 38-57. Springer Verlag, LNCS 1088, June 1996.

GUERRAOUI, R.; SCHIPER, A.,. Fault-Tolerance by Replication in Distributed Systems. Reliable Software Technologies - Ada-Europe´96, Springer Verlag, LNCS 1088, p.38-57, 1996.

HAGSAND, O.; H. Herzog, K.P. Birman, R. Cooper. Object-Oriented Reliable Distributed Computing. 2nd IEEE International Workshop on Object-Orientation in Operational Systems, 1992.

HORSTMANN.Cay S., Cornel, Gary. Core Java 2. Makron Books, São Paulo, 2001.

JALOTE, P.. Fault Tolerance in Distribuited Systems. New Jersey: Prentice-Hall, 1994, 432p.

Java Core Reflection: API and Specification. Java Soft, Mountain View, CA, USA, Jan. 1997.

JBOSS. JBoss 2.4+ Documentation, Consulta feita ao site http://www.jboss.org/online-manual/HTML/ch01.html#d0e131 em janeiro de 2003a.

JBOSS. Over 2 Million Downloads In 2002, Consulta feita ao site http://www.jboss.org/ em janeiro de 2003b.

JÚNIOR, José Maria Rodrigues Santos. A Plataforma Java 2 Enterprise Edition e a Especificação Enterprise Java Beans 2.0 - Mestrado em Informática Fora de Sede Universidade Tiradentes Aracaju – Sergipe, 2001.

LAMPORT, L., R. Shostak, M. Pease. The Byzantine Generals Problem. TOPLAS 4, 3 (July 1982) pp. 382-401.

LAMPORT, L., Time, clocks and the ordering of events in a distributed system. CACM 21(7):558-565, ACM, July 1978.

LISBOA, Maria Lúcia Blanck. Arquitetura de Meta-nível. Tutorial apresentado no XI Simpósio Brasileiro de Engenharia de Software, Fortaleza, Out. 1997.

LITTLE, M. C. Object Replication in a Distributed System. PhD. Thesis, University of Newcastle upon Tyne Computing Laboratory, September 1991.

MAIAN, Luis. Wutka, Mark . Java: Técnicas Profissionais. Berkeley, São Paulo, 1997.

MENÉNDEZ, Andrés Ignácio Martinez. Uma ferramenta de apoio ao desenvolvimento de Web Services. Dissertação (mestrado), Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Agosto de 2002, 97 p. Il.

MULLENDER, Sape. Distributed Systems. New York: ACM Press, 1993, 601p.

ORCHARD D. Transaction Procesing with EJB. Abril/1999 Component Strategies, SIGS.

ORFALI, R., HARKEY, D. The Essential Distributed Objects Survival Guide – Orfali. John Wiley & Sons, Inc., 1996.

PRADHAN, D. K., Fault-Tolerant System Design. Prentice Hall, NewJersey, 1996.

ROMAN, Ed. Mastering Enterprise JavaBeansTM and the JavaTM 2 Platform, Enterprise Edition. John Wiley & Sons; 722p, 1999.

SAMPAIO, Lívia Maria Rodrigues. Serviços de Processamento Tolerantes a Faltas para Sistemas Distribuídos Assíncronos. Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande - Pb, Outubro de 2000. 81 p. Il.

SCHIPER, André; RAYNAL, Michel. From Group Communication to Transations in Distribuited Systems. Communications of the ACM, vol 39, nº 4, Abril 1996, pp 84-87.

SCHNEIDER, F. B. Implementing Fault-Tolerant Service Using the State Machine Approach: A Tutorial. ACM Computing Survey, 22(4):299-319, December 1990.

SORIA-RODRIGUEZ, Pedro. Multicast-Based Interactive-Group Object-Replicaton for Fault Tolerance. Worcester Polytechnic Institute, Massachussets, 1998. Dissertação de Mestrado. Disponível por www em http://xfactor.wpi.edu/Works/PSoria/thesis.html

SOUZA, Cristina Verçosa Pérez Barrios de. Uso de Reflexão Computacional em Aplicações da Plataforma J2EE™. Curitiba, 2001. 163 p. Dissertação (Mestrado) – Pontifícia Universidade Católica do Paraná. Departamento de Informática.

SUN. The J2EE Tutorial, Sun Microsystems, Inc., EUA. J2EE Containers. Consulta feita ao site http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Overview3.html em janeiro de 2003a.

SUN. The J2EE Tutorial, Sun Microsystems, Inc., EUA. Distributed Multitiered Applications. Consulta feita ao site http://java.sun.com/j2ee/overview2.html em janeiro de 2003b.

SUN. Enterprise JavaBeans™ Specification, v1.1., Sun Microsystems, Inc., EUA. Consultado o site http://java.sun.com/products/ejb/docs.html em Março de 2002.

TANEMBAUM, Andrew S.. Sistemas Operacionais Modernos. Editora Prentice Hall do Brasil. 1995.

THOMAS, Anne. Enterprise JavaBeans™ Technology – Server Component Model for the Java™ Platform. Patricia Seybold Group, EUA, Dez. 1998.

THOMAS, Anne. Java™ 2 Platform, Enterprise Edition – Ensuring Consistency, Portability, and Interoperability. Patricia Seybold Group, EUA, Jun. 1999.

THOMAS, Michael D., Patel, Patrik R.. Programando em Java para a Internet. Makron Books, São Paulo, 1997.

VEER, Emily A. Vander, Maian, Luis. Java Beans: para Leigos . Berkeley, São Paulo, 1997.

WEBER, Taisy Silva.Um roteiro para exploração dos conceitos básicos de tolerância a falhas. Instituto de Informática – UFRGS. Curso de Especialização em Redes e Sistemas Distribuídos. www.inf.ufrgs.br/~taisy/disciplinas/ textos/Dependabilidade.pdf, 2003.

WUTKA, Mark . Special Edition: Using Java 2. Que. Indianapolis, 2001.