Universidade Federal de Campina Grande

Centro de Ciências e Tecnologia
Curso de Mestrado em Informática
Coordenação de Pós-Graduação em Informática

Usando Replicação Ativa para Prover Tolerância a Falhas de Forma Transparente a uma Implementação da Plataforma J2EE

André Andrade Costa

Campina Grande - PB
Novembro - 2002

Universidade Federal de Campina Grande

Centro de Ciências e Tecnologia
Curso de Mestrado em Informática
Coordenação de Pós-Graduação em Informática

Usando Replicação Ativa para Prover Tolerância a Falhas de Forma Transparente a uma Implementação da Plataforma J2EE

André Andrade Costa

Dissertação submetida à Coordenação de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal de Campina Grande como requisito parcial para a obtenção do grau de Mestre em Ciências (MSc)

Prof. Dr. Francisco Vilar Brasileiro
(Orientador)

Campina Grande - PB
Novembro - 2002

Ficha Catalográfica

COSTA, André Andrade

C837U

Usando Replicação Ativa para Prover Tolerância a Falhas de Forma Transparente a uma Implementação da Plataforma J2EE

Dissertação (mestrado), Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Novembro de 2002.

119 p. Il.

Orientador: Prof. Dr. Francisco Vilar Brasileiro

Palavras-chaves:

1. **Tolerância a Falhas**
2. **Replicação Ativa**
3. **J2EE**
4. **JAVA**
5. **Sistemas Distribuídos**

CDU - 681.3.066D

"USANDO REPLICAÇÃO ATIVA PARA PROVER TOLERÂNCIA A FALHAS DE FORMA TRANSPARENTE A UMA IMPLEMENTAÇÃO DA PLATAFORMA J2EE"

ANDRÉ ANDRADE COSTA

DISSERTAÇÃO APROVADA EM 06.12.2002

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador

PROF. WALFREDO DA COSTA CIRNE FILHO, Ph.D
Examinador

PROFª THAIS VASCONCELOS BATISTA, Drª
Examinadora

CAMPINA GRANDE – PB

“Testing can show the presence of bugs, but not their absence”.

- Dijkstra

Agradecimentos

A Deus, em primeiro lugar, por permitir concluir mais esta etapa da minha vida e por me encorajar e me fortalecer nos momentos mais difíceis desse trabalho.

A minha mãe Láice Andrade e a meu pai Rivaldo Machado Costa por terem me tornado a pessoa que sou hoje.

A meu irmão Allex pelo carinho e pela compreensão nos momentos em que não estive presente.

A minhas "irmãs" Karine e Vanilde, que sempre me deram muito apoio e pelo companheirismo ao longo de todo esse tempo.

A meu orientador, Prof. Francisco Vilar Brasileiro, pela paciência e por acreditar que esse trabalho daria certo. Foi uma grande oportunidade tê-lo como orientador.

Aos professores Jacques, Sampaio e Peter por terem me dado grandes lições tanto dentro, quanto fora da sala de aula.

Enfim, a todos aqueles que, direta ou indiretamente, contribuíram, de alguma forma, para a conclusão desse trabalho.

Sumário

Capítulo 1: Introdução 1
Capítulo 2: A Plataforma J2EE e Limitações em Relação ao Provimento de Confiança no Funcionamento 9
2.1 Introdução 9
2.2 A Plataforma Java 2, Enterprise Edition 9
2.2.1 Modelo de Programação de Aplicação J2EE 10
2.2.2 Plataforma J2EE 11
2.2.3 Suíte de Testes de Compatibilidade J2EE 12
2.2.4 Implementação de Referência 13
2.3 Java Servlets 13
2.3.1 Contêiner do Servlet 15
2.3.2 Arquitetura dos Servlets 17
2.3.3 Componentes Web 19
2.3.4 Limitações 20
2.4 Java Naming and Directory Interface 20
2.4.1 Conceito de Nomes e Registros 20
2.4.2 O Conceito de Contexto 21
2.4.3 Arquitetura do JNDI 21
2.4.4 Limitações 23
2.5 Java Message Service 23
2.5.1 Arquitetura 25
2.5.2 Modelos de Envio de Mensagem 30
2.5.3 Limitações 34
2.6 Java Remote Method Invocation 35
2.6.1 Arquitetura 35
2.6.2 Limitações 38
2.7 Enterprise JavaBeans 38
2.7.1 Desenvolvimento Baseado em Componentes 40
2.7.2 Arquitetura de um Enterprise JavaBean 42

2.7.3 O Entity Bean 44
2.7.4 O Session Bean 48
2.7.5 Limitações 51
2.8 Conclusão 52
Capítulo 3: Trabalhos Relacionados 54
3.1 Introdução 54
3.2 O Servidor de Aplicações BEA WebLogic Server 6.0 54
3.3 O Servidor de Aplicações Borland AppServer 4.5 58
3.4 O Servidor de Aplicações Sybase EAServer 3.6 61
3.5 O Servidor de Aplicações Persistence PowerTier for J2EE 6.0 62
3.6 O Servidor de Aplicações Allaire Jrun 3.1 66
3.7 Conclusão 67
Capítulo 4: Projeto de uma Plataforma J2EE Tolerante a Falhas e Transparente 70
4.1 Introdução 70
4.2 Visão Geral 71
4.3 Java Servlet Tolerante a Falhas 72
4.4 JNDI Tolerante a Falhas 74
4.5 JMS Tolerante a Falhas 75
4.6 Group Method Invocation 76
4.7 EJB Tolerante a Falhas 77
4.8 Conclusão 77
Capítulo 5: JBossFT 78
5.1 Introdução 78
5.2 O Servidor de Aplicações JBoss 78
5.3 O JavaGroups 79
5.4 Implementação do *Group Proxy* 83
5.5 Alteração das Bibliotecas Clientes dos Serviços 83
5.5.1 ServletFT 85
5.5.2 JNDIFT 86
5.5.3 GMI 88
5.6 Testes e Avaliações de Desempenho 89
5.7 Conclusão 93
Capítulo 6: Conclusões 95
6.1 Discussão 95

6.2 Direções para Trabalhos Futuros 96
7 - Referências Bibliográficas 98

Lista de Abreviaturas ou Siglas

API	*Application Programming Interface*
BMP	*Bean-Managed Persistence*
CGI	*Common Gateway Interface*
CMP	*Container-Managed Persistence*
CORBA	*Common Object Request Broker Architecture*
DNS	*Domain Name System*
EJB	*Enterprise JavaBeans*
FIFO	*First In First Out*
GMI	*Group Method Invocation*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IIOP	*Internet Inter-Orb Protocol*
IIS	*Internet Information Services*
IP	*Internet Protocol*
J2EE	*Java 2, Enterprise Edition*
JDBC	*Java Database Connectivity*
JMS	*Java Message Service*
JMX	*Java Management Extensions*
JNDI	*Java Naming and Directory Interface*
JSP	*Java Server Pages*
JTA	*Java Transaction API*
JTS	*Java Transaction Service*
LDAP	*Lightweight Directory Access Protocol*
MFC	*Microsoft Foundation Classes*
OCA	*Optimistic Control Attribute*
PTP	*Point-to-Point*
Pub/Sub	*Publish-and-Subscribe*

RMI	*Remote Method Invocation*
SPI	*Service Provider Interface*
TCP	*Transmission Control Protocol*
UDP	*User Datagram Protocol*
URI	*Uniform Resource Identifier*
VCL	*Visual Component Library*
WAN	*Wide Area Network*
XML	*Extensible Markup Language*

Lista de Figuras

Figura 2.1 Arquitetura da Aplicação J2EE 10
Figura 2.2 Funcionamento de uma aplicação *Web* 14
Figura 2.3 Arquitetura do *Servlet* 17
Figura 2.4 Arquitetura JNDI 22
Figura 2.5 Exemplo de comunicação entre sistemas utilizando envio de mensagem 24
Figura 2.6 Envio e recebimento de mensagens com contexto transacional 28
Figura 2.7 Modelo de envio de mensagem PTP 31
Figura 2.8 Exemplo de cliente JMS utilizando o domínio PTP 31
Figura 2.9 Modelo de envio de mensagem Pub/Sub 32
Figura 2.10 Exemplo de cliente JMS utilizando o domínio Pub/Sub 32
Figura 2.11 Registro não durável e registro durável 33
Figura 2.12 Exemplo de interface remota 36
Figura 2.13 Exemplo de implementação do objeto remoto 36
Figura 2.14 Comunicação entre o cliente e o objeto remoto 37
Figura 2.15 Exemplo de invocação do objeto remoto 37
Figura 2.16 Modelo de aplicação multicamada 39
Figura 2.17 Modelo multicamada utilizando a arquitetura EJB 41
Figura 2.18 Controle transacional entre servidores 41
Figura 2.19 Criação de uma instância de um componente EJB 43
Figura 2.20 *Entity bean* da tabela CONTAS 44
Figura 2.21 Interface remota do *entity bean* Conta 45
Figura 2.22 Chave primária do *entity bean* Conta 45
Figura 2.23 Interface *home* do *entity bean* Conta 46
Figura 2.24 Implementação do *entity bean* Conta 47
Figura 2.25 Funcionamento do *session bean* 48
Figura 2.26 Interface *home* do *session bean* Operação 49
Figura 2.27 Interface remota do *session bean* Operacao 49
Figura 2.28 Implementação do *session bean* Operação 50
Figura 2.29 Aplicação cliente 51

Figura 4.1 Arquitetura do projeto 72
Figura 4.2 Funcionamento do balanceador de carga 73
Figura 4.3 Funcionamento do JNDI Tolerante a Falhas 74
Figura 4.4 Funcionamento do provedor JMS replicado 75
Figura 4.5 Funcionamento do GMI 76
Figura 5.1 Arquitetura de uma aplicação utilizando o JavaGroups 81
Figura 5.2 Exemplo de utilização do *PullPushAdapter* 82
Figura 5.3 Exemplo de utilização do *Request Dispatcher* 84
Figura 5.4 Implantação do JBossFT para a camada *Web* 86
Figura 5.5 Arquitetura do JNDI Tolerante a Falhas 87
Figura 5.6 Criação dinâmica do *Group Stub* 88
Figura 5.7 Tempo médio de resposta sem tolerância a falhas 91
Figura 5.8 Tempo médio de resposta com tolerância a falhas 92
Figura 5.9 Tempo de Recuperação do JBossFT e do WebLogic 93

Lista de Tabelas

Tabela 3.1 Resumo das características dos servidores estudados..... 68

Resumo

Um grande número de aplicações distribuídas tem seu projeto e implementação sustentados por plataformas de desenvolvimento. Estas plataformas provêm uma série de serviços especializados, permitindo assim que os programadores possam se concentrar mais nas regras de negócio das aplicações que desenvolvem. Atualmente a plataforma J2EE (Java 2 Enterprise Edition) da SUN Microsystems é uma das mais populares para este fim. Infelizmente, tolerância a falhas, um requisito não funcional cada vez mais presente nas aplicações, não é diretamente suportada pela especificação J2EE. Aplicações desenvolvidas sobre essa plataforma devem, elas mesmas, implementar os mecanismos para tolerância a falhas requeridos, ou usar implementações da plataforma que possuam características de tolerância a falhas. Nesta dissertação nós apresentamos o projeto e a implementação de um servidor de aplicações J2EE que implementa esses mecanismos. Diferentemente de outras soluções disponíveis, que usam replicação passiva, o nosso sistema usa replicação ativa para prover alta confiabilidade de forma totalmente transparente para as aplicações.

Abstract

The use of developing platforms to support the implementation of distributed applications has become a trend. These platforms provide a number of specialized services that help programmers to focus on the business logic of the applications they develop, instead of wasting precious time with the implementation of infrastructure services. J2EE (Java 2 Enterprise Edition) is a platform backed up by SUN Microsystems that has lately gain a lot of attention. Unfortunately, the J2EE specification does not provide any support for fault tolerance, a non-functional requirement more and more necessary for distributed applications. Developers of such applications must themselves provide the necessary mechanisms to fulfill the requirements of the applications. Alternatively, they can use implementations of the platform that are themselves fault tolerant. In this dissertation we present the design and implementation of such a platform. Unlike other implementations available, that use passive replication, our approach uses active replication to provide a solution that is highly reliable and totally transparent to the application.

Capítulo 1: Introdução

Com o objetivo de obter vantagens competitivas e responder de forma mais hábil às mudanças do mercado, as empresas vêem constantemente adotando novas tecnologias. Quando os sistemas de computador começaram a ser utilizados nas organizações, a ênfase da tecnologia de informação era basicamente o gerenciamento de dados. Graças aos sistemas de gerenciamento de banco de dados de grande escala, as empresas puderam reunir, analisar e interpretar dados para obter vantagem estratégica. Na atual economia em rede, a ênfase da tecnologia da informação passa a ser a aplicação [Kassem et al. 00].

O desenvolvimento de sistemas distribuídos é a solução que permite a reutilização de dados pré-existentes dos sistemas legados e o acesso a novos dados. Através desse tipo de aplicação, a organização pode estabelecer elos seguros e consistentes com seus clientes, fornecedores e parceiros. Dessa forma, torna-se muito importante para a organização desenvolver, distribuir e manter aplicações inovadoras de maneira rápida e produtiva sempre que exista necessidade [Cattel-Inscore 01].

Devido ao aumento da complexidade dos sistemas distribuídos, a tendência atual de desenvolvimento tem feito amplo uso de arquiteturas baseadas em componentes servidores. Nesse modelo de desenvolvimento, além de se perceber uma melhor reutilização de software, o desenvolvimento torna-se bastante simplificado, uma vez que os serviços mais complexos requeridos pela aplicação estão implementados no ambiente de execução. Dessa forma, o desenvolvedor da aplicação precisa levar em consideração em seu projeto apenas a lógica do negócio, enquanto que serviços como controle transacional e gerenciamento de conexões com o banco de dados podem ficar sob responsabilidade do servidor de aplicações que está hospedando a aplicação.

A especificação Java 2, Enterprise Edition (J2EE) [Cattel-Inscore 01], criada pela SUN Microsystems em parceria com um número de empresas, é considerada uma das mais promissoras tecnologias de componentes servidores do mercado. Ela permite que sejam

desenvolvidos sistemas distribuídos de maneira mais simples além de oferecer um padrão para o desenvolvimento e distribuição de aplicações necessárias para tirar proveito do alcance da economia em rede [Cattel-Inscore 01]. Quando se utiliza um padrão de desenvolvimento como esse oferecido pela J2EE, as empresas podem focar seus esforços na solução dos problemas das regras de negócio em vez de tentar resolver problemas técnicos de infra-estrutura.

A plataforma J2EE permite que o desenvolvimento de sistemas distribuídos possa fazer uso de tecnologias recém surgidas além de simplificar o desenvolvimento através de um modelo de aplicação baseado em componentes. O modelo J2EE suporta desde aplicações cliente/servidor instaladas em *intranets* coorporativas até aplicações de comércio eletrônico para a Internet [Kassem et al. 00]. De um modo geral, uma aplicação J2EE é composta de componentes *Web* (para que a aplicação possa ser utilizada a partir de um navegador *Web*) e componentes de negócio (contendo a lógica de processamento ou representando uma entidade do projeto) em execução em um servidor de aplicação J2EE. Este servidor irá oferecer serviços como envio de mensagem, acesso ao banco de dados e controle de concorrência de maneira transparente para o desenvolvedor. Os usuários da aplicação J2EE poderão utilizá-la tanto pelo navegador *Web* quanto por uma aplicação com interface gráfica sendo que, em todos os casos, o processamento da aplicação será sempre realizado no servidor de aplicações.

Em vários segmentos do mercado (financeiro, médico, industrial, etc.), existe a necessidade de garantir que determinados sistemas, considerados críticos para o negócio da empresa, atendam a requisitos de confiança no funcionamento, além dos requisitos de seu projeto. De um modo geral, isso significa que a aplicação deve estar sempre em funcionamento e deve executar o serviço especificado corretamente. Para que os serviços permaneçam disponíveis aos clientes mesmo ocorrendo uma falha no *hardware* ou no *software*, normalmente é preciso introduzir mecanismos de tolerância a falhas [Jalote 94]. Quando se utiliza uma arquitetura baseada em componentes servidores como a J2EE, a disponibilidade do serviço ficará comprometida caso o servidor de aplicação que hospeda os componentes apresente algum tipo de falha. Como os serviços básicos e o processamento da aplicação são de responsabilidade do servidor J2EE, os usuários não poderão acessar o serviço (mesmo que suas estações estejam em funcionamento) caso o meio de comunicação entre seu ponto de rede e o do servidor torne-se indisponível ou a máquina do servidor de aplicações apresente um defeito.

Os serviços da plataforma J2EE não são especificados de forma a dar suporte à implementação de aplicações com requisitos de confiança no funcionamento. Assim, os mecanismos de tolerância a falhas devem ser implementados pela própria aplicação ou pelos serviços que compõem o servidor de aplicações. Em ambos os casos os componentes da aplicação devem ser instalados em mais de um servidor de aplicação em máquinas distintas para que, na falha de uma instância do servidor, as outras instâncias operacionais possam dar continuidade ao serviço.

A primeira alternativa consiste em incluir no projeto da aplicação soluções para tratar a eventual falha do servidor de aplicações. Nessa abordagem, a aplicação cliente deve identificar que os componentes ou serviços da instância do servidor J2EE que utilizava não estão mais disponíveis e ela mesma deve passar a acessar outra instância que esteja operacional. É de responsabilidade da aplicação cliente refazer o processamento até o ponto da falha na instância substituta para que o usuário não perceba a indisponibilidade do servidor. O problema dessa alternativa é que os mecanismos de tolerância a falhas não são transparentes para o desenvolvedor, o que faz com que a complexidade da aplicação aumente de forma significativa. Além disso, torna-se obrigatório que toda aplicação inclua esses mecanismos.

A segunda alternativa consiste em implementar os serviços que compõem a plataforma de tal forma que as instâncias do servidor J2EE em execução possam cooperar entre si para que a falha de uma instância possa ser mascarada pelas instâncias operacionais. Enquanto que na primeira alternativa as instâncias do servidor J2EE operam de forma independente umas das outras, nesta abordagem há o conceito de grupo de replicação onde cada instância "conhece" quais são as outras instâncias (réplicas) que fazem parte de seu grupo. A utilização do grupo de replicação tem como objetivo principal oferecer o maior grau possível de transparência para o desenvolvedor da aplicação J2EE. Para a aplicação cliente que utiliza os componentes e serviços, o grupo de réplicas deve ser transparente, ou seja, caso uma instância apresente uma falha, uma réplica deve assumir o processamento sem que a camada cliente ou os componentes da aplicação J2EE precisem realizar nenhum tipo de tratamento. Essa alternativa mostra-se mais adequada uma vez que libera o desenvolvedor da tarefa de implementar os mecanismos para tratamento de falhas simplificando o desenvolvimento de aplicações J2EE robustas.

Este trabalho apresenta o projeto e a implementação de um servidor J2EE onde os mecanismos para tolerância a falhas são utilizados pelas aplicações de forma totalmente transparente. Utilizando esse servidor, o desenvolvedor poderá se concentrar na lógica de negócio e não na infra-estrutura requerida para tolerar falhas. Por oferecer suporte a tolerância a falhas de forma totalmente transparente, a plataforma também permite que aplicações já desenvolvidas também sejam executadas com maior robustez, sem qualquer modificação no código fonte original.

Como explicado anteriormente, é necessário haver um grupo de réplicas do servidor J2EE em execução para que a falha de uma das instâncias não seja percebida pelo usuário. Entretanto, para que o grupo de réplicas seja transparente para a aplicação cliente, é preciso adotar uma estratégia de replicação a fim de coordenar as réplicas preparando-as para que, no momento que ocorrer uma falha, uma réplica possa dar continuidade ao processamento do sistema. [Jalote 94]. As duas estratégias mais usadas para coordenar a computação replicada são: Replicação Ativa e Replicação Passiva [Jalote 94]. Esses modelos de replicação irão realizar o processamento do erro e, conseqüentemente, permitir que a comunicação e a computação possam proceder apesar da falha de uma ou mais réplicas.

Cada estratégia de replicação é utilizada para tratar determinados tipos de falhas de réplicas. Isso implica que é necessário identificar qual o comportamento da réplica quando ela apresenta uma falha. De uma maneira geral, uma réplica pode falhar por tempo, por omissão ou ainda apresentar falha na sua resposta [Cristian 91]. A falha por tempo é aquela na qual a réplica responde muito cedo ou muito tarde a uma requisição. Já a falha por omissão ocorre quando a réplica não responde a algumas requisições enviadas a ela. Quando a réplica falha por omissão e deixa de responder indefinidamente, considera-se que a semântica de falha é silenciosa [Cristian 91]. Essa semântica é considerada mais restritiva por considerar que as respostas que as réplicas enviam estão sempre corretas. Num outro extremo, está a semântica da falha arbitrária onde não é feita nenhuma restrição quanto ao comportamento da réplica quando esta falha. Nessa semântica, a réplica pode deixar de responder, atrasar ou adiantar o envio de respostas, enviar respostas extras, enviar respostas erradas, enviar respostas diferentes para diferentes destinos, recusar o recebimento de requisições ou ainda, maliciosamente, fazer se passar por outra réplica [Jalote 94]. Esse tipo de falha, também conhecida como falha Bizantina, pode ocorrer devido a erros na implementação ou concepção

do *software*, sendo assim, os mecanismos empregados para tolerá-las consistem em empregar diversidade de projeto. Essa solução implica em utilizar várias implementações de uma réplica para comparar as respostas por elas emitidas e obter a resposta correta. Já a semântica da falha silenciosa ocorre por faltas de componentes do *hardware* no qual a réplica está em execução. Para mascarar o erro, basta replicar a mesma réplica em unidades de processamento distintas e empregar um mecanismo de atualização de estado e controle de processamento para que uma réplica possa substituir uma instância que falhou.

O modelo de replicação ativa consiste em realizar o processamento das réplicas em paralelo e comparar os resultados destas de modo a utilizar a decisão majoritária. Sendo assim, essa estratégia pode tanto tolerar falhas Bizantinas, uma vez que será feita uma votação entre os resultados das diferentes implementações, como também falhas com semântica silenciosa poderão ser suportadas (nesse caso, sem a necessidade de votação, já que qualquer resultado enviado será válido) [Jalote 94]. Uma grande vantagem desta técnica é que, em caso de falha de parte das réplicas, o resultado correto pode ser obtido no mesmo instante, uma vez que as réplicas que não apresentam falhas já realizaram o processamento.

Um sistema usando o modelo de replicação ativa deve especificar o grau de replicação baseado na semântica de falha que irá suportar. Define-se o sistema como t-resistente se ele pode tolerar falhas de até um número t de seus componentes [Jalote 94]. Nesse contexto, para suportar falhas Bizantinas no modelo de replicação ativa, um componente deve possuir pelo menos $2t + 1$ réplicas para que o resultado da maioria das réplicas possa ser considerado válido [Jalote 94]. Por outro lado, se apenas falhas silenciosas precisarão ser suportadas, apenas $t + 1$ réplicas serão necessárias, uma vez que bastaria que apenas uma única réplica enviasse a resposta [Jalote 94].

Quando se planeja tolerar apenas falhas silenciosas, o mecanismo de replicação ativa torna-se mais simples por não exigir nenhuma técnica de votação. Neste contexto, é necessário algum tipo de protocolo que faça com que apenas um resultado seja utilizado. Esse processamento pode ser feito de duas formas [Powell 92]:

- para cada resposta gerada, um protocolo é executado entre as réplicas com objetivo de decidir qual delas enviará a resposta para o cliente; ou
- todas as réplicas enviam suas respostas e o cliente trata de selecionar uma e ignorar as outras.

Com a replicação ativa, a recuperação quase instantânea dos erros detectados pode ser conseguida desde que seja possível garantir que todas as réplicas livres de falha produzam as mesmas mensagens de saída na mesma ordem [Powell 92]. Para que essa condição seja atendida é preciso garantir:

- Consistência de entrada: o conjunto de mensagens de entrada deve ser idêntico para todas as réplicas livres de falha; e
- Determinismo do grupo de réplica: começando de estados iniciais idênticos e processando o mesmo conjunto ordenado de mensagens de entrada, todas as réplicas livres de falha produzem mensagens de saída idênticas e na mesma ordem.

Consistência de entrada implica que qualquer mensagem enviada para um componente de software seja entregue a todas ou a nenhuma das réplicas livres de falha. O principal requisito exigido para o sistema de comunicação é que ele implemente um protocolo de comunicação em grupo que garanta unanimidade [Powell 92] entre os receptores livres de falha. Já o determinismo do grupo de réplicas pode ser conseguido garantindo que as réplicas livres de falha processem de forma determinísitica e recebam as mesmas mensagens de entrada numa mesma ordem, isto é, numa ordem total [Powell 92]

Uma outra opção para tratar falhas silenciosas é a utilização do modelo de replicação passiva. Neste modelo, apenas uma réplica (primária) responderá pela requisição e, caso esta venha a falhar, uma réplica suplente assumirá a função da primária [Jalote 94]. Para que a réplica suplente possa assumir o processamento da réplica primária, é necessário que o estado desta seja propagado por todas as suas suplentes. Na replicação passiva, o único processamento realizado pelas réplicas suplentes na ausência de falhas é a atualização de seus estados com a réplica primária numa operação denominada salvaguarda (*checkpointing*) [Jalote 94].

Apesar de diminuir a carga de processamento do sistema na ausência de falhas, esse modelo requer uma capacidade maior de comunicação devido às operações repetidas de salvaguarda. Além disso, na ocorrência de uma falha, haverá uma carga extra de processamento, uma vez que a réplica suplente deve re-executar, a partir do último ponto de salvaguarda, as ações já realizadas pela réplica primária antes da falha. A salvaguarda consiste de uma fotografia do estado interno da réplica primária, incluindo seu espaço de dados e

demais informações específicas da instância que serão úteis para que uma outra réplica possa assumir o processamento [Powell 92].

No momento que uma falha de uma réplica primária é detectada, uma suplente deve ser escolhida dentre as demais réplicas para executar a recuperação a partir do último ponto de salvaguarda. Essa seleção pode ser realizada por meio de eleição dinâmica ou pode seguir uma ordem preestabelecida entre as réplicas suplentes [Jalote 94].

De uma maneira geral, as soluções existentes para prover tolerância a falhas na plataforma J2EE [BEA, Borland 01-1, Sybase, Persistence, Allaire] são baseadas em um modelo de replicação passiva. Os servidores realizam a salvaguarda dos estados dos componentes periodicamente ou a cada requisição processada. Nessas soluções percebe-se que podem ocorrer situações em que a última alteração no estado pode ser perdida sem que a aplicação cliente perceba, levando a uma inconsistência no processamento do sistema. Além disso, o grupo de replicação não é transparente para o cliente. Em todos os casos, quando se deseja utilizar um dos serviços da plataforma J2EE, deve-se especificar qual instância do grupo de replicação deverá responder pelo serviço. É também de responsabilidade da aplicação cliente tratar a falha quando esta acontece durante o processamento da requisição, uma vez que, em quase todos os casos, o servidor só consegue mascarar a falha se ela ocorrer entre requisições. Em resumo, para desenvolver aplicações J2EE com requisitos de confiança no funcionamento utilizando esses servidores de aplicação é necessário que o projeto do sistema inclua alguns mecanismos para complementar o tratamento de falhas oferecido pelo servidor, fazendo com que a complexidade do sistema aumente.

A nossa estratégia para um servidor J2EE tolerante a falhas e transparente consiste em incluir mecanismos de replicação ativa em um servidor J2EE de código aberto. Este modelo de replicação ativa será totalmente transparente para o desenvolvedor de forma que bastará instalar os componentes da aplicação nas instâncias do servidor para que cada réplica possa processar as requisições da camada cliente. Como as réplicas estarão sincronizadas, qualquer uma das respostas emitidas pelas réplicas poderá ser utilizada o que eliminará as “janelas de vulnerabilidade” presentes nas soluções atuais.

Esta dissertação está organizada da seguinte forma: o Capítulo 2 apresenta a plataforma J2EE explicando o funcionamento dos seus serviços e suas respectivas limitações.

Este capítulo visa dar suporte ao restante da dissertação onde esses conceitos serão necessários ao entendimento da solução proposta. No Capítulo 3, analisamos as principais soluções disponíveis para prover confiança no funcionamento para a plataforma J2EE. Também neste capítulo, discutimos as falhas e limitações dessas soluções, como também as implicações decorrentes para o desenvolvedor utilizá-las. O Capítulo 4 descreve a nossa solução sob a forma de um projeto onde são explicados os requisitos necessários a serem implementados. No Capítulo 5, é apresentado o servidor J2EE escolhido como base para nossa solução e os detalhes da implementação da replicação ativa. Neste capítulo também é realizada uma avaliação de desempenho da nossa solução. Essa avaliação tem como objetivo medir o impacto da introdução da replicação ativa no servidor e comparar a nossa solução com outras soluções disponíveis. O Capítulo 6 traz nossas conclusões, com ênfase na relevância dessa dissertação, e discute possíveis direções para trabalhos futuros.

Capítulo 2: A Plataforma J2EE e Limitações em Relação ao Provimento de Confiança no Funcionamento

2.1 Introdução

A plataforma J2EE foi criada por um consórcio de empresas liderado pela SUN Microsystems com o objetivo de oferecer um padrão de desenvolvimento que simplificasse a criação, distribuição e manutenção de aplicações distribuídas baseadas em componentes servidores. Utilizando essa plataforma de desenvolvimento, podem ser criados desde aplicações de comércio eletrônico para a Internet, até sistemas corporativos de grande escala para *intranets* de maneira bastante simplificada, uma vez que a própria plataforma garante a existência dos serviços que irão formar a infra-estrutura da aplicação. Dessa forma, o desenvolvedor da aplicação precisa se concentrar apenas na lógica de negócio, visto que a plataforma irá gerenciar os serviços necessários de forma transparente.

Apesar de oferecer transparência para o desenvolvimento de aplicações baseadas em componentes servidores, a especificação dos serviços da plataforma J2EE não foi concebida de forma a dar suporte ao desenvolvimento de aplicações robustas. Este capítulo apresenta a plataforma J2EE com ênfase nos seus principais serviços e suas respectivas limitações.

2.2 A Plataforma Java 2, Enterprise Edition

Com o objetivo de uniformizar o desenvolvimento como também garantir a compatibilidade da aplicação desenvolvida, a plataforma J2EE define uma arquitetura padrão composta pelos seguintes elementos [SUN 99-3]:

- Modelo de Programação de Aplicação (J2EE *Application Programming Model*): Um modelo padrão de programação para o desenvolvimento de aplicações multicamadas com clientes leves (*thin-clients*);

- Plataforma J2EE (J2EE *Platform*): Uma plataforma padrão para hospedar aplicações J2EE sob a forma de uma especificação de serviços gerenciados por um servidor de aplicações;
- Suíte de Testes de Compatibilidade (J2EE *Compatibility Test Suit*): Um conjunto de testes de compatibilidade usados para verificar se um determinado servidor de aplicações segue rigorosamente a especificação J2EE; e
- Implementação de Referência (J2EE *Reference Implementation*): Uma implementação da especificação J2EE para servir de referência para implementação de servidores de aplicação e para demonstrar as possibilidades da plataforma J2EE.

2.2.1 Modelo de Programação de Aplicação J2EE

A plataforma J2EE foi projetada para dar suporte ao desenvolvimento de aplicações potencialmente complexas que necessitam acessar dados de várias origens distintas e devem ser distribuídas para vários tipos de clientes. O modelo de aplicação J2EE define uma arquitetura para implementação de serviços sob a forma de aplicações multicamadas com o objetivo de oferecer escalabilidade, acessibilidade e fácil gerenciamento [SUN 99-3]. A Figura 2.1 ilustra as camadas que irão compor uma aplicação seguindo o modelo proposto pelo Modelo de Programação de Aplicação.

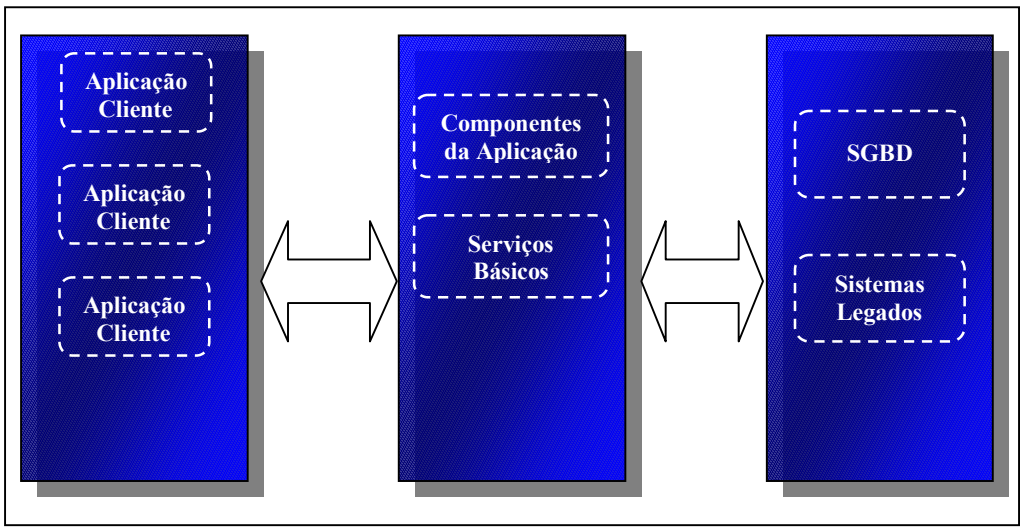

Figura 2.1 Arquitetura da Aplicação J2EE

Esse modelo de desenvolvimento permite particionar o trabalho necessário para implementar uma aplicação multicamada em duas partes: as lógicas de negócio e de apresentação que ficam sob responsabilidade do desenvolvedor final e os serviços de infra-estrutura providos pela plataforma J2EE. O desenvolvedor pode deixar a cargo da plataforma (servidor de aplicações) a responsabilidade de prover as soluções para os problemas de infra-estrutura existentes no desenvolvimento dos serviços básicos da camada intermediária (*middle-tier*) [SUN 99-3]. Com a plataforma J2EE, o processamento da aplicação será realizado na camada intermediária através de componentes Enterprise JavaBeans (EJB), e a apresentação da aplicação para o usuário é realizada pelos componentes *Web* (compostos por JavaServer Pages e Java Servlets) também em execução na camada intermediária [SUN 99-3].

Além de permitir que a interface da aplicação siga o padrão Internet através de páginas HTML (*HyperText Markup Language*) geradas pela camada intermediária, o modelo de aplicações J2EE suporta outros tipos de clientes na camada de interface. É possível, por exemplo, que a camada de interface com o cliente seja um *applet*, uma aplicação gráfica Java ou um cliente implementado com outra linguagem de programação [SUN 99-3].

2.2.2 Plataforma J2EE

A especificação J2EE define uma plataforma padrão composta de vários serviços que serão gerenciados por um servidor de aplicações com o objetivo de oferecer um ambiente para a execução das aplicações [SUN 99-2]. Uma vez que os servidores de aplicações J2EE seguem uma única especificação, uma mesma aplicação poderá ser executada da mesma forma em qualquer produto disponível.

O ambiente de execução da plataforma J2EE é estruturado da seguinte forma [SUN 99-2]:

- Componentes de Aplicação (*Application Components*): Uma aplicação J2EE é formada por vários componentes: aplicações cliente, *applets*, componentes *Web* e Enterprise JavaBeans;
- Contêineres: Na plataforma J2EE, os contêineres provêm o suporte para execução dos componentes que formam o sistema. Essa camada de software se interpõe entre os componentes da aplicação e os serviços J2EE para que os

componentes possam fazer uso desses serviços de forma transparente. Um servidor J2EE deve prover um contêiner específico para a aplicação cliente e o *applet*, para os componentes *Web* e para os componentes EJB. O contêiner permite que o desenvolvedor especifique de que forma este irá utilizar determinados serviços como controle transacional, segurança, *pooling* de recursos e gerenciamento de estado, apenas definindo de forma apropriada as propriedades do componente; e

- Serviços padrões: Os serviços que um servidor J2EE deve suportar são: HTTP – definido no lado cliente por interfaces a esse protocolo e no lado servidor pelas tecnologias JSP e Java Servlets; HTTPS; Java Transaction API (JTA) – API (*Application Programming Interface*) utilizada pelas aplicações para demarcar transações; RMI-IIOP – protocolo de comunicação entre objetos remotos; Java IDL – serviço que provê interoperabilidade com a tecnologia CORBA; JDBC – API para acesso a bancos de dados relacionais; Java Message Service (JMS) – API padrão para intercâmbio de mensagens; Java Naming and Directory Interface (JNDI) – API padrão para serviços de nomes; e JavaMail – API padrão para acesso a serviços de correio eletrônico.

É importante destacar que é permitido ao servidor J2EE incluir outras extensões que sejam necessárias. Os fabricantes têm liberdade para implementar outras tecnologias da plataforma Java assim como outros serviços e protocolos não definidos na especificação.

2.2.3 Suíte de Testes de Compatibilidade J2EE

Com o objetivo de permitir que os fabricantes possam verificar se suas implementações estão de acordo com a especificação da plataforma J2EE, a SUN Microsystems criou a Suíte de Testes de Compatibilidade J2EE, um conjunto de ferramentas e aplicações de teste que irão atestar a funcionalidade do servidor J2EE. Essas ferramentas irão verificar se os serviços estão implementados de forma correta, e se os componentes estão funcionando de forma integrada. Já as aplicações de teste irão verificar se o servidor será capaz de hospedá-las de forma consistente [SUN 99-3].

2.2.4 Implementação de Referência

Além de especificar o que os produtos J2EE devem atender, a plataforma J2EE inclui uma implementação da especificação para servir de referência. Trata-se de um servidor amplamente funcional que serve tanto para que os fabricantes possam comparar a funcionalidade de seus produtos quanto para os desenvolvedores da aplicação final verificarem a portabilidade dos seus sistemas.

2.3 Java Servlets

Com a popularização da Internet, além de conteúdo estático, serviços dinâmicos começaram a ser oferecidos cada vez mais. Enquanto que em um primeiro momento as informações disponibilizadas na *World Wide Web* eram produzidas de forma manual através de páginas HTML, o aumento do volume de informação que deveria estar disponível aos usuários da *Web* incentivou o desenvolvimento de aplicações que disponibilizassem esses dados de forma automática, uma vez que a produção de páginas estáticas se tornaria inviável. Como a Internet oferece uma alta demanda e uma grande escala, passou a ser decisivo para muitas empresas permitir que seus serviços pudessem ser acessados pela *Web*. Assim, foi necessário desenvolver um tipo de aplicação que interagisse com o navegador *Web* (*browser*) para receber deste os dados de entrada fornecidos pelo usuário e retornar o resultado do processamento sob a forma de uma página HTML.

Pode-se citar como exemplo deste tipo de aplicação (aplicação *Web*) um sistema de comércio eletrônico onde o usuário, através do seu navegador *Web*, pesquisa os produtos que deseja comprar e em seguida informa seus dados para efetuar o pedido. Quando o usuário pesquisa os produtos disponíveis, ele está fornecendo parâmetros de consulta para a aplicação de comércio eletrônico, de forma que esta possa acessar a base de dados da empresa e recuperar os dados necessários. Similarmente, na confirmação do pedido, o usuário preenche seus dados em um formulário exibido pelo navegador *Web* que envia estes dados para a aplicação a fim de registrá-los no banco de dados. A Figura 2.2 ilustra esse processo.

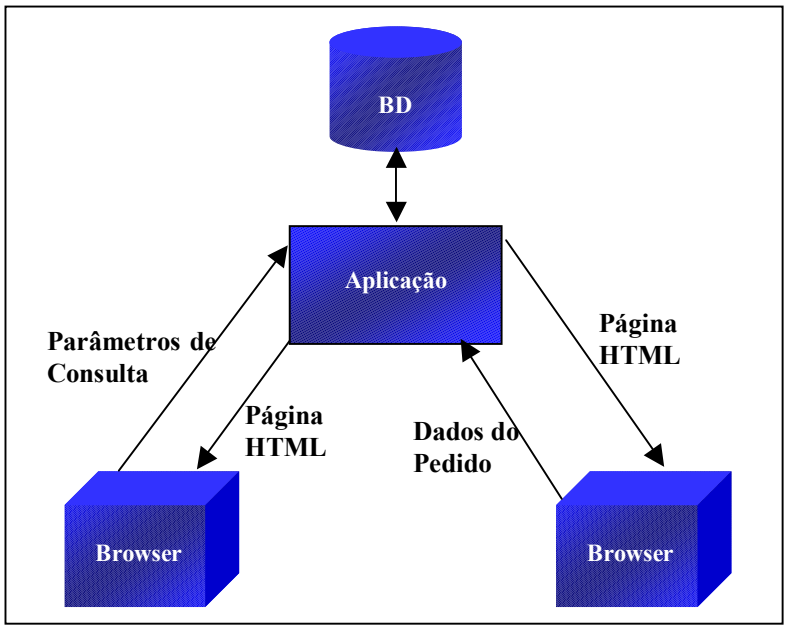

Figura 2.2 Funcionamento de uma aplicação *Web*

Na Internet, as páginas no formato HTML são hospedadas em um tipo de software chamado servidor *Web* (*Web server*). Quando o usuário solicita uma página, o seu navegador *Web* envia uma requisição HTTP (*HyperText Transfer Protocol*) para o servidor *Web*, o que faz com que este retorne a página HTML requisitada para o navegador.

Para dar suporte ao desenvolvimento de aplicações para a Internet, muitos fabricantes de servidores *Web* passaram a incluir em seus produtos APIs para acessar as facilidades do servidor *Web*. Com essas APIs, o desenvolvedor poderia estender as funcionalidades do servidor de páginas para que ele passasse a atender às suas necessidades. Essa solução permitiria, por exemplo, que o servidor *Web* interagisse com o banco de dados ao receber determinadas requisições e gerasse uma página HTML como reposta do processamento. Entretanto, esse tipo de solução não foi bem aceita por obrigar que a aplicação desenvolvida tivesse de ser executada em conjunto com um produto específico.

Com o objetivo de desenvolver aplicações *Web* que fossem independentes do servidor de páginas, passou a ser adotado a CGI (*Common Gateway Interface*) [Kassem et al. 00]. Através da CGI, os desenvolvedores poderiam usar qualquer linguagem que suportasse essa tecnologia para receber as requisições do servidor *Web* e enviar o resultado do processamento sob a forma de uma página HTML. Apesar do uso de CGI ser suportado por praticamente todos os servidores, essa abordagem possui diversas limitações que a impedem de ser

utilizada para desenvolver serviços que exigem alta demanda. O modelo de aplicações da CGI oferece baixo desempenho uma vez que, a cada requisição do cliente, um novo processo da aplicação CGI é criado para atender a essa requisição. Da mesma forma, o processo é encerrado quando a aplicação envia a resposta da requisição. Caso o serviço venha a ter um grande número de requisições em um curto espaço de tempo, o servidor que hospeda a aplicação poderá ficar sem recursos suficientes devido à quantidade de processos em execução. Além disso, cada vez que o processo é criado, geralmente ele deve obter conexões com banco de dados ou outros recursos para atender à requisição, o que pode levar a uma limitação de usuários que o serviço irá atender.

Diante desses problemas, a SUN Microsystems criou a especificação Java Servlets [SUN 99-4], uma tecnologia independente de plataforma que permite que o desenvolvedor possa estender as funcionalidades do servidor *Web* para, por exemplo, acessar bancos de dados e sistemas legados. Além de poder ser executado em conjunto com qualquer servidor *Web*, o *servlet* resolve o problema de desempenho da aplicação CGI, uma vez que há apenas um processo em execução atendendo a várias requisições por meios de múltiplas linhas de execução (*threads)*.

Um *servlet* pode ser definido como um componente *Web* gerenciado por um contêiner para gerar conteúdo dinâmico [SUN 99-4]. Ele consiste de uma classe Java que será carregada e executada a partir de invocações do servidor *Web*. Os *servlets* interagem com clientes *Web* (navegadores, dispositivos móveis, etc.) através de um protocolo de requisição/resposta baseado no HTTP e implementado pelo contêiner do *servlet* [SUN 99-4].

2.3.1 Contêiner do Servlet

O contêiner do *servlet*, em conjunto com um servidor *Web*, provê a infra-estrutura necessária para a execução dos *servlets*. O contêiner será responsável por [Bodoff 01]:

- redirecionamento de requisições;
- segurança;
- concorrência;
- gerenciamento do ciclo de vida dos *servlets*;
- decodificação de requisições; e

- formatação das respostas.

O contêiner do *servlet* pode ser tanto implementado pelo servidor *Web* quanto como um *software* a parte [SUN 99-4]. No primeiro caso, o contêiner estará em execução no mesmo processo do servidor *Web*, ou seja, o próprio servidor de páginas irá gerenciar a execução dos *servlets*. Na segunda situação, o contêiner é um processo que interage com o servidor *Web* recebendo as requisições HTTP e retornando as páginas geradas pelos *servlets*. Nesse último caso, é possível que o contêiner esteja em execução tanto na mesma máquina do servidor *Web*, quanto em um ponto de rede distinto.

Quando o navegador *Web* realiza uma requisição HTTP, essa requisição é processada pelo servidor *Web* e, se for identificada como uma requisição a um *servlet*, a requisição é repassa para o contêiner. A maneira como o servidor *Web* decide repassar a requisição HTTP varia de acordo com o servidor *Web* utilizado, mas, de uma maneira geral, é especificado no servidor de páginas que requisições possuindo determinado padrão deverão ser processadas por outro *software* (o contêiner). De forma similar, o contêiner, a partir da requisição e de seus parâmetros de configuração, determina qual *servlet* deverá ser invocado e realiza a chamada a sua classe, passando objetos representando a requisição e a resposta.

O *servlet* pode utilizar o objeto referente à requisição para descobrir informações sobre o cliente remoto, quais os parâmetros que o usuário está informando à aplicação e outros dados relevantes. Com essas informações, o *servlet* irá executar a lógica de negócio e retornar o resultado desse processamento (a maioria das vezes uma página HTML) através do objeto que representa a resposta [SUN 99-4]. Após o *servlet* ter processado a requisição, o contêiner irá formatar a resposta de maneira que ela possa ser recebida pelo servidor *Web* e, conseqüentemente, enviada para o navegador *Web* do usuário. A Figura 2.3 ilustra a interação dos componentes que compõe a arquitetura.

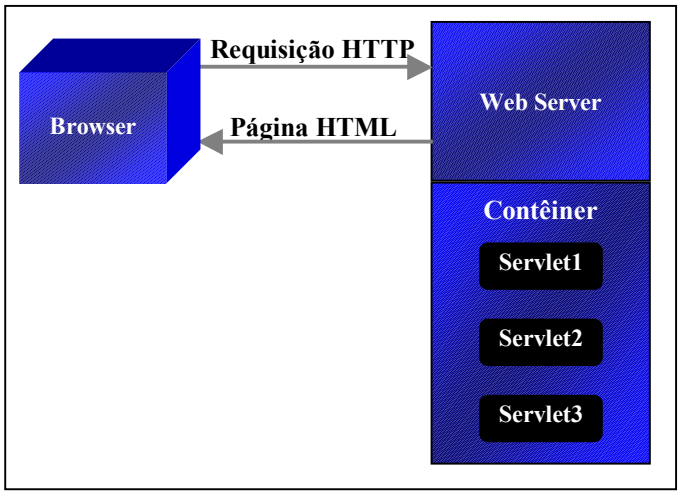

Figura 2.3 Arquitetura do *Servlet*

A especificação Java Servlet não define de que maneira o servidor *Web* e o contêiner irão se comunicar. Assim, da mesma maneira que o fabricante do contêiner deverá prover o ambiente de execução, fica sob sua responsabilidade conhecer o protocolo do servidor de páginas para que o desenvolvimento do *servlet* seja independente da plataforma sobre a qual ele irá executar [SUN 99-4].

2.3.2 Arquitetura dos Servlets

O principal componente da arquitetura dessa tecnologia é a interface `javax.servlet.Servlet` que deve ser implementada por qualquer *servlet* que se deseja desenvolver. A implementação dessa interface consiste em codificar os métodos que irão ser invocados pelo contêiner à medida que as requisições forem sendo recebidas. A cada requisição, é criada uma linha de execução na máquina virtual Java do contêiner para realizar o processamento dessa requisição. A especificação define que o contêiner deve suportar todos os tipos de requisição da versão 1.1 do protocolo HTTP [SUN 99-4]: GET, POST, PUT, DELETE, HEAD, OPTIONS e TRACE.

É de responsabilidade do contêiner gerenciar o ciclo de vida dos *servlets*. Antes de começar a processar as requisições dos clientes, o *servlet* deve ser instanciado e inicializado. O contêiner poder instanciar o *servlet* durante seu próprio processo de inicialização ou quando ocorre a primeira requisição para esse *servlet*. Após ser instanciado, o *servlet* poderá realizar o seu processo de inicialização que pode consistir, por exemplo, em obter conexões com o

banco de dados ou em alocar outros recursos necessários para processar as requisições que serão feitas. Com a instância do *servlet* criada, o contêiner irá utilizá-la para processar as requisições. O contêiner pode decidir destruir a instância de um *servlet* caso perceba, por exemplo, que esse *servlet* não está sendo usado por um determinado período de tempo [SUN 99-4].

Outro componente importante dessa arquitetura é o objeto que representa o contexto do *servlet* (`javax.servlet.ServletContext`). Esse objeto define a visão que o *servlet* possui da aplicação *Web* da qual ele faz parte [SUN 99-4]. Além de permitir o acesso a recursos disponíveis, o contexto pode ser usado pelos *servlets* para fazer um registro (*log*) de eventos ou definir atributos para serem utilizados por outros *servlets* da sua aplicação. Esses atributos podem ser qualquer objeto, sendo que cada um possuirá um nome que o identificará. Dessa forma, os *servlets* de uma aplicação *Web* poderão compartilhar dados para serem usados ao longo do processamento e o contêiner irá garantir o controle de concorrência aos dados do contexto.

Quando o *servlet* recebe uma requisição, ele a recebe na forma de um objeto definido pela sua API. O objeto `javax.servlet.http.HttpServletRequest` irá encapsular toda a informação da requisição do cliente [SUN 99-4]. No protocolo HTTP, essa informação é transmitida do cliente para o servidor sob a forma de cabeçalhos e do corpo da mensagem da requisição HTTP. Os dados mais importantes da requisição para um *servlet* são os parâmetros, que são valores em formato texto, enviados pelo cliente para o contêiner como parte da requisição. Esses valores podem ser fornecidos pelo cliente através de um formulário HTML ou podem estar embutidos no URI (*Uniform Resource Identifier*) que invocará o *servlet* [SUN 99-4]. Esses parâmetros da requisição irão compor os dados de entrada necessários para que a aplicação possa realizar o processamento, como por exemplo: dados de um pedido, condições de uma consulta, etc.

Após o navegador *Web* realizar uma requisição, ele irá aguardar por uma resposta do servidor indicando o resultado do processamento. Para auxiliar na criação dessa resposta, é passado para o *servlet* o objeto `javax.servlet.http.HttpServletResponse` que irá encapsular toda a informação que deve ser retornada do servidor para o cliente [SUN 99-4]. Esse objeto possui métodos que permitem que o *servlet* crie a resposta de seu processamento

(uma página HTML com a confirmação de um pedido ou resultado de uma consulta, por exemplo) como se estivesse gerando um arquivo em disco.

O protocolo HTTP foi projetado para ser um protocolo sem estado [SUN 99-4]. Entretanto, muitas aplicações *Web* necessitam que uma série de diferentes requisições, originadas de um mesmo cliente, possam ser associadas umas com as outras. Um exemplo disso seria quando um cliente deve ser autenticado antes de utilizar um serviço. Depois de ele informar a sua identificação e sua senha, a aplicação deve "lembrar" que toda requisição proveniente daquele navegador *Web* está associada ao cliente que foi autenticado. Assim, a aplicação deve manter um estado (sessão) em nome do cliente. A especificação Java Servlet define a interface `javax.servlet.http.HttpSession` para permitir que o contêiner implemente algum mecanismo de acompanhamento da sessão do cliente de forma que fique transparente para o desenvolvedor do *servlet* [SUN 99-4]. Com o objeto `HttpSession`, o *servlet* pode salvar objetos na sessão associando-os com um nome. Assim, o mesmo *servlet* ou um outro da mesma aplicação poderá recuperar esse objeto para utilizar em seu processamento. Uma vez que nenhum evento é gerado informando que o cliente não está mais ativo, o administrador ou o desenvolvedor pode especificar um período de tempo (*timeout*) de validade da sessão [SUN 99-4]. Caso a sessão não seja utilizada pelo período de tempo especificado, ela será inutilizada, o que fará que os objetos armazenados sejam destruídos impedindo que a máquina virtual passe a ficar sem recursos.

2.3.3 Componentes Web

Numa aplicação J2EE, a camada de interface com o cliente pode ser desenvolvida através de componentes *Web*, que podem ser *servlets* ou JavaServer Pages (JSP) [Bodoff 01]. As páginas JSP podem ser consideradas como uma maneira simplificada de desenvolver *servlets*. O desenvolvedor, em vez de codificar a classe do *servlet*, constrói uma página HTML incluindo código Java no trecho da página onde deve ser exibido conteúdo dinâmico. Na primeira requisição feita à página, o contêiner gera automaticamente um *servlet* que irá receber as requisições, gerar o código HTML referente à página construída e executar código Java incluído pelo desenvolvedor. Dessa forma o processo de execução do JSP é totalmente baseado na especificação Java Servlet.

2.3.4 Limitações

À medida que os *servlets* foram se tornando mais populares, surgiu um número cada vez maior de aplicações *Web* de grande porte baseadas nessa tecnologia. Esse tipo de aplicação caracteriza-se por apresentar uma grande demanda, tráfego intenso e por necessitar de alta disponibilidade [Hunter-Crawford 01]. Para atender a esses requisitos, é necessário que uma mesma aplicação *Web* esteja em execução em servidores distintos para haver uma distribuição de carga entre eles e para que a falha de um servidor não leve à indisponibilidade do sistema. Entretanto, uma vez que a aplicação pode armazenar diversas informações na sessão, um segundo servidor só poderá continuar o processamento de um primeiro que falhou se o segundo tiver acesso às informações das sessões que estavam ativas e do contexto (`ServletContext`) antes da falha.

2.4 Java Naming and Directory Interface

Com a utilização cada vez maior de sistemas distribuídos nas corporações de médio e grande porte, surgiu a necessidade de prover uma maneira para que diferentes tipos de aplicações pudessem compartilhar informações entre si. Assim, surgiram os serviços de nomes que, de uma maneira geral, facilitam o acesso a recursos dos sistemas. Através desses serviços, uma aplicação ou o administrador podem registrar um recurso ou informação associando-o a um nome para que uma outra aplicação possa recuperar o recurso a partir desse nome [SUN 99-5]. Como exemplo, pode-se citar o Internet *Domain Name System* (DNS) que faz o mapeamento de nomes de máquina para endereços IP [RFC1034] e os sistemas de arquivo de um sistema operacional responsáveis por fazer o mapeamento de um nome de arquivo para o identificador interno do arquivo.

2.4.1 Conceito de Nomes e Registros

Para localizar um objeto (recurso, informação, etc.) em um serviço de nomes, o cliente deve informar o nome do objeto. A forma como o nome deve ser composto irá depender exclusivamente do serviço de nomes. No Unix, por exemplo, um dos formatos dos nomes em seu sistema de arquivos especifica que o nome do arquivo é composto do diretório raiz até o seu diretório relativo e cada componente do caminho deve ser separado por "/". Já no LDAP

(*Lightweight Directory Access Protocol*) o formato do nome deve ser da direita para a esquerda e cada componente do nome (composto do nome do campo e de seu valor) deve ser separado por "," [RFC2251]. Por exemplo, o nome: "cn=Maria da Silva,o=Banco X,c=BR" corresponde ao registro LDAP "cn=Maria da Silva" que é relativo a "o=BancoX" que, por sua vez é relativo a "c=BR".

A associação de um nome com um objeto é chamada de registro. O DNS, por exemplo, possui registros que mapeiam nomes do tipo "www.bancox.com.br" com um endereço IP.

2.4.2 O Conceito de Contexto

Nos serviços de nomes, um contexto corresponde a um conjunto de registros. Através do contexto, é possível obter um determinado objeto (realizando a consulta através do nome) como também criar e remover registros ou recuperar todo os nomes registrados. Em um contexto, é possível criar um registro tanto para um objeto quanto para um outro contexto (chamado subcontexto).

No sistema de arquivos do Unix, por exemplo, pode-se afirmar que o nome "/usr/bin/ls" identifica o arquivo "ls" que está contido no subcontexto (diretório) "bin" que, por sua vez, está contido no subcontexto "usr" do contexto "/".

2.4.3 Arquitetura do JNDI

O JNDI (Java Naming and Directory Interface) é um serviço essencial da plataforma J2EE [Roman 99]. Ele consiste de uma especificação para que aplicações clientes em Java possam interagir com sistemas de nomes e diretórios, provendo uma interface comum para as várias implementações existentes no mercado. Através desse serviço, é possível registrar objetos com um determinado nome para que aplicações distribuídas pela rede possam acessá-los a partir desse nome de forma independente da implementação do serviço de nomes [Kassem et al. 00].

A arquitetura JNDI é composta de dois elementos principais [SUN 99-5]:

- API JNDI, um conjunto de interfaces utilizado pelas aplicações que necessitam acessar um serviço de nomes; e
- SPI (*Service Provider Interface*) JNDI, pacote de classes que permite que as várias implementações de serviços de nomes possam ser utilizadas pelas aplicações clientes de forma transparente.

A implementação das interfaces da API JNDI é de responsabilidade do serviço de nomes e irá formar a biblioteca cliente JNDI de acesso ao serviço de nomes. Essa biblioteca cliente contém o protocolo de acesso ao serviço de nomes e será utilizada pelas aplicações clientes JNDI como ilustrado na Figura 2.4.

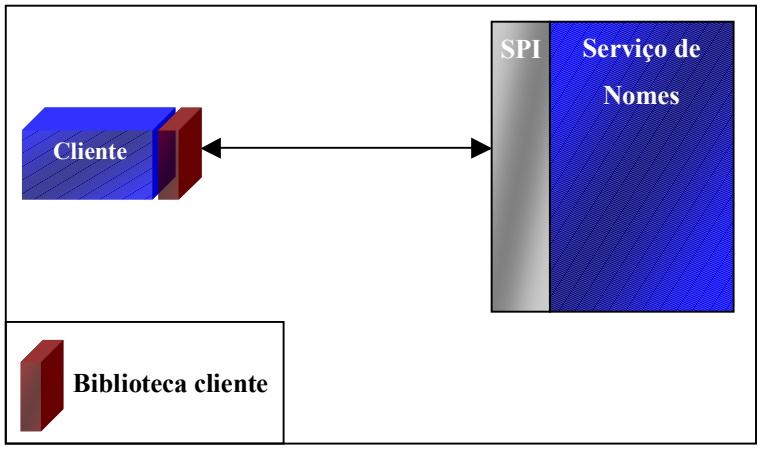

Figura 2.4 Arquitetura JNDI

A principal vantagem o JNDI é que ele permite que as aplicações clientes permaneçam independentes do serviço de nomes utilizado. A aplicação cliente obtém o acesso inicial ao serviço de nomes através da classe `javax.naming.InitialContext`. Através de um parâmetro passado a esta classe em tempo de execução, será utilizada a biblioteca cliente correspondente do serviço de nomes a acessar [SUN 99-5]. A classe `InitialContext` irá invocar o "construtor" (*factory*) de contextos da biblioteca cliente (classe que implementa `javax.namin.spi.InitialContextFactory`) e este, por sua vez, irá instanciar o contexto correspondente ao serviço de nomes [SUN 99-5]. Por meio desse contexto (interface `javax.naming.Context`) a aplicação cliente poderá consultar (`lookup`), registrar (`bind`), remover registros (`unbind`) ou obter todos os registros desse contexto (`list`).

2.4.4 Limitações

A utilização de apenas uma instância do servidor que implementa o serviço JNDI pode provocar a perda das informações do serviço. Portanto, para que falhas sejam toleradas, deve-se optar pela utilização de mais de uma instância do servidor. Entretanto, fica a cargo da aplicação garantir que as instâncias do serviço JNDI permaneçam atualizadas uma vez que a especificação desse serviço não determina que isso deva ser feito pela plataforma J2EE. Além disso, quando uma aplicação deseja consultar o serviço JNDI, ela deve implementar o tratamento de eventuais falhas, isso inclui a detecção da falha e a localização de outra instância do serviço para realizar a consulta.

2.5 Java Message Service

À medida que as organizações foram desenvolvendo novas aplicações, a necessidade de integrá-las para formar um sistema de informação mais amplo e ágil tornou-se maior. Passou a ser comum em muitas empresas implementar um sistema independente para atender a cada departamento, o que levou à distribuição dos dados e a uma maior dificuldade de combiná-los para extrair as informações necessárias. Sendo assim, sistemas de envio de mensagens começaram a ser usados para permitir que aplicações isoladas pudessem trocar dados e possibilitar que componentes do negócio separados fossem combinados para formar um sistema mais flexível e confiável.

As tecnologias para envio de mensagens permitem que um determinado componente possa enviar mensagens para um destino de onde serão recuperadas pelo receptor. Uma das principais características de uma aplicação que utiliza envio de mensagens é a possibilidade de oferecer comunicação distribuída com baixo acoplamento, ou seja, o emissor não necessita conhecer quem será o receptor e este não precisa conhecer nada a respeito do emissor. Estes componentes precisam saber apenas como interpretar as mensagens e qual destino usar para enviar e receber estas mensagens [Haase 01]. Além disso, a comunicação entre o emissor e o receptor pode ser feita de forma assíncrona: o emissor não precisa esperar que o receptor receba a mensagem para continuar seu processamento.

Desde a versão 1.1, a plataforma Java oferece o RMI (Remote Method Invocation), uma tecnologia que permite a comunicação entre aplicações hospedadas em máquinas virtuais distintas [Campione et al. 98]. Entretanto, por obrigar que o emissor tenha de conhecer o receptor previamente (alto acoplamento) e por não oferecer uma maneira de envio de mensagem de forma assíncrona, o RMI não se mostrou adequado para muitos tipos de situações. Pode-se tomar como exemplo uma empresa de revenda que possui um sistema de comércio eletrônico e um sistema de controle de estoque desenvolvidos utilizando linguagens/plataformas diferentes. Para cada venda realizada, o sistema de comércio eletrônico deve atualizar o sistema de controle de estoque e este, por sua vez, realiza o pedido para o fornecedor do produto toda vez que a loja precisa ser reabastecida. O intercâmbio de informações pode ser feito através de envio de mensagens assíncronas como ilustrado na Figura 2.5.

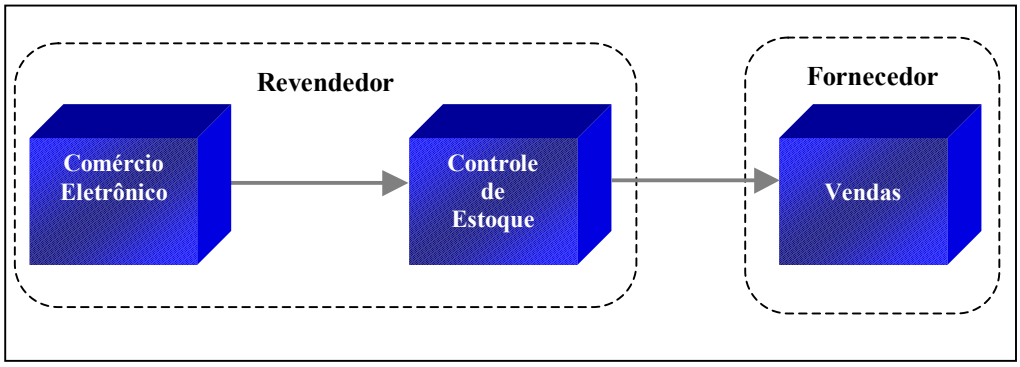

Figura 2.5 Exemplo de comunicação entre sistemas utilizando envio de mensagem

Nesse cenário, cada sistema é independente do outro, ou seja, o sistema de comércio eletrônico não precisa esperar por uma resposta do sistema de controle de estoque e o sistema de vendas do fornecedor não precisa necessariamente estar em funcionamento no momento que for feito o pedido. Uma vez que os sistemas de envio de mensagem não requerem alto acoplamento entre os componentes envolvidos, não haverá problema caso estes tenham sido desenvolvidos utilizando linguagens ou plataformas distintas.

Muitos fabricantes de *software* criaram suas próprias soluções para permitir comunicação entre aplicações através de envio de mensagens (*Message Oriented Middleware*) e, para permitir que aplicações e serviços escritos em Java pudessem acessar esses sistemas de envio de mensagens de uma maneira comum, foi criada a especificação Java Message Service (JMS) [SUN 01]. Essa especificação consiste de um conjunto de interfaces e

de semânticas associadas que definem como um cliente JMS acessa as facilidades oferecidas pelo sistema de envio de mensagem [SUN 01]. Para o desenvolvedor, o JMS aparece sob a forma de uma extensão da plataforma Java para criar, enviar e receber mensagens de forma assíncrona [Haase 01]. Com o JMS, o emissor pode continuar seu processamento após o envio da mensagem mesmo que esta ainda não tenha sido entregue ao receptor. Dessa forma o JMS permite integrar aplicações corporativas com baixo acoplamento (ou até mesmo sem acoplamento) e possibilitar que estas aplicações não dependam de um produto em particular [Flores 01]. O JMS tornou-se um componente importante da plataforma J2EE por que, além de permitir a comunicação entre componentes distintos de forma assíncrona, possibilita a integração com sistemas legados da organização.

Uma aplicação JMS consiste basicamente de um conjunto de mensagens definidas pela aplicação e um conjunto de clientes que irão enviar e receber estas mensagens [SUN 01]. Um fabricante que deseja produzir um produto baseado na especificação JMS deve desenvolver um provedor (JMS *Provider*) implementando as interfaces do JMS. Nessa tecnologia, as mensagens são requisições assíncronas ou eventos que serão consumidos por um componente de *software* [SUN 01]. Estas mensagens são constituídas de dados em um formato especificado descrevendo uma ação do negócio e o intercâmbio destas ações gera o progresso do sistema como um todo.

2.5.1 Arquitetura

Para entender o funcionamento dessa tecnologia é necessário conhecer o que constitui uma aplicação JMS. De um modo geral as aplicações JMS podem ser divididas nos seguintes componentes [SUN 01]:

- Clientes JMS: Aplicações escritas em linguagem Java que recebem e enviam mensagens;
- Outros clientes: Aplicações escritas em outras linguagens que acessam o serviço de envio de mensagens através de bibliotecas nativas que o próprio produto disponibiliza;
- Mensagens: As aplicações definem um conjunto de mensagens que serão usadas para estabelecer comunicação entre seus clientes.

- Provedor JMS: Sistema de envio de mensagens que implementa tanto a especificação JMS quanto os demais requisitos necessários; e
- Objetos auxiliares: Objetos definidos pelo JMS criados e pré-configurados pelo administrador do sistema com o objetivo de serem usados pelos clientes JMS para acessar as facilidades do provedor JMS.

A especificação JMS permite que os provedores JMS possam diferir quanto à tecnologia adotada na infra-estrutura dos seus produtos. É esperado também que diferenças significantes ocorram na configuração e administração desses produtos. Para que os clientes JMS possam ser portáveis, eles devem permanecer isolados desses aspectos proprietários do provedor. Isso é obtido através dos objetos auxiliares que serão criados e configurados pelo administrador e depois usados pelos clientes JMS. Esses objetos são dispostos no serviço JNDI pelo administrador para que os clientes JMS possam obtê-los de forma direta e única. Os clientes usam esses objetos por meio das interfaces JMS que são únicas e usadas por todos os provedores. Os objetos auxiliares é que irão interagir diretamente com os recursos específicos do provedor JMS. Há dois tipos de objetos auxiliares [SUN 01]:

- Construtor de Conexão (*Connection Factory*): Objeto utilizado pelo cliente JMS para obter uma conexão com o provedor JMS. Este objeto engloba um conjunto de parâmetros de configuração definidos pelo administrador [Haase 01]; e
- Destino (*Destination*): Objeto utilizado pelo cliente JMS para especificar tanto o destino das mensagens que ele envia quanto a origem das mensagens que ele deve receber. Uma vez que o JMS não define uma sintaxe padrão para endereçamento, foi criado este objeto que engloba o endereçamento específico do provedor [SUN 01].

De uma maneira geral, uma aplicação JMS consiste de clientes JMS enviando e recebendo mensagens utilizando um provedor JMS. Antes de iniciar o processo de comunicação, o cliente deve implementar os seguintes passos [SUN 01]:

1. Através do JNDI, obter uma referência a um Construtor de Conexão;
2. Através do JNDI, obter um ou mais objetos do tipo Destino;
3. Através do Construtor de Conexão, criar uma conexão JMS com o provedor JMS;
4. Utilizar a conexão obtida para criar uma sessão JMS;

5. Utilizar a sessão e o Destino para criar produtores e consumidores de mensagens; e
6. Ativar a conexão para que o envio de mensagens possa ocorrer.

A conexão criada pelo Construtor de Conexão representa uma conexão virtual com o provedor JMS e pode, por exemplo, consistir de um *socket* TCP/IP estabelecido entre o cliente e a instância do provedor JMS em execução [Haase 01]. Além disso, a conexão será responsável por alocar recursos destinados à comunicação fora da máquina virtual Java [SUN 01]. A conexão criada também serve para guardar a identificação do cliente que está acessando o provedor JMS. Esta identificação é gerada pelo Construtor de Conexão no momento que o cliente solicita a criação de uma conexão [SUN 01] e serve para associar uma conexão e seus objetos com um estado mantido pelo provedor JMS em nome do cliente. O provedor JMS deve garantir que esse estado seja usado por apenas um cliente para evitar perda ou duplicação de mensagens [SUN 01].

A sessão JMS criada a partir da conexão consiste de um contexto no qual mensagens poderão ser enviadas, recebidas e confirmadas, além de funcionar como um fabricante de mensagens, produtores e consumidores de mensagens [Haase 01]. Quando uma aplicação cliente deseja enviar mensagens, ela deve utilizar a sessão para criar um produtor de mensagens que irá se encarregar de acessar o provedor JMS e enviar a mensagem para o destino desejado [SUN 01]. De forma similar, o cliente JMS deve criar um consumidor de mensagens que irá recuperar as mensagens de um determinado destino. O cliente JMS também utiliza a sessão JMS para criar as mensagens que deseja enviar. Após obter da sessão a nova mensagem criada, a aplicação cliente pode preencher seu conteúdo com as informações necessárias para a aplicação, antes de enviá-la [SUN 01].

A sessão também é responsável por manter a ordem das mensagens enviadas e recebidas e armazená-las até que elas sejam confirmadas. O desenvolvedor do cliente JMS pode optar por especificar a sessão como transacional. Nesse caso, a sessão irá suportar séries de transações e cada transação irá agrupar um conjunto de mensagens produzidas e um conjunto de mensagens consumidas em unidades atômicas. De um modo geral, as transações irão organizar o fluxo de mensagens de entrada e o fluxo de mensagens de saída da sessão em unidades indivisíveis [SUN 01]. Caso uma das operações da transação falhe, a transação é desfeita e as operações poderão ser refeitas do começo. Quando uma transação é efetivada,

todas as mensagens produzidas são enviadas e todas as mensagens consumidas são confirmadas. Entretanto, se a transação é desfeita, todas as mensagens produzidas são destruídas e as mensagens consumidas são recuperadas e reenviadas [Haase 01].

É importante destacar que não existe um contexto transacional entre os pontos extremos da aplicação (emissor e receptor). Esse contexto irá existir entre o emissor e o destino ou entre o destino e o receptor [Haase 01] como ilustra a Figura 2.6.

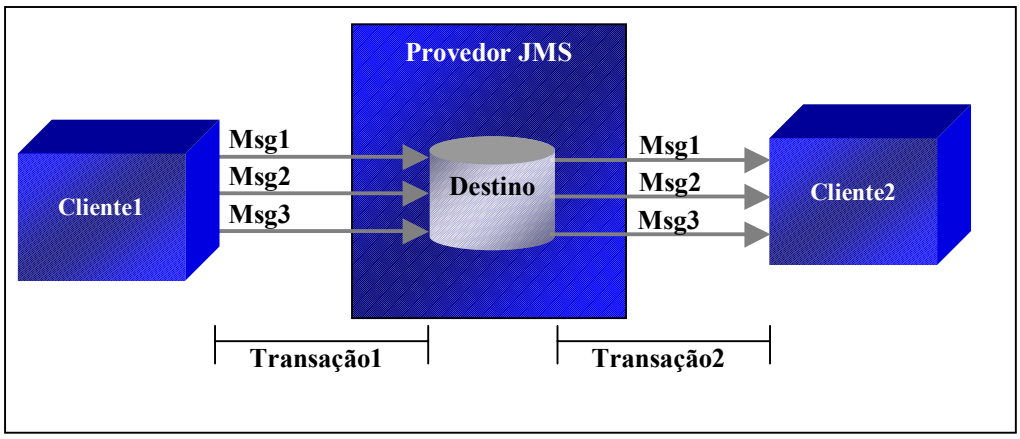

Figura 2.6 Envio e recebimento de mensagens com contexto transacional

Portanto, é desaconselhável agrupar envios e recebimentos de mensagens na mesma transação. Caso o objetivo da aplicação seja implementar um protocolo de requisição/resposta (enviar uma mensagem e esperar por uma outra mensagem em resposta à primeira na mesma transação) o cliente será bloqueado porque o envio não será feito até a transação ser efetivada [Haase 01].

Os clientes JMS irão consumir as mensagens em uma ordem serial garantida pela sessão [SUN 01]. Essa ordem é importante tanto para a lógica do negócio quanto para garantir o funcionamento correto do mecanismo de confirmação de mensagens. Entretanto, a sessão intercala as mensagens dos seus diversos consumidores para que um não fique bloqueado esperando pela finalização do outro. A especificação JMS garante que as mensagens enviadas para um destino deverão ser recebidas na mesma ordem com a qual foram enviadas, definindo, assim, uma ordenação parcial [SUN 01]. O JMS não define a ordem de recebimento de mensagens provenientes de destinos distintos como também não há ordenação de mensagens de um mesmo destino enviadas a partir de sessões diferentes [SUN 01].

A sessão JMS também irá definir o modo como as mensagens recebidas serão confirmadas. Uma mensagem só é considerada plenamente consumida quando é confirmada e este processo se dá da seguinte forma [Haase 01]:

1. A aplicação cliente recebe a mensagem;
2. A aplicação cliente processa a mensagem; e
3. O recebimento da mensagem é confirmado.

A confirmação do recebimento pode ser iniciada tanto pelo provedor JMS de forma automática, quanto pela aplicação cliente que recebeu a mensagem. A escolha de quem deve iniciar a confirmação é especificada na sessão JMS pelo desenvolvedor. Caso a sessão seja transacional, a confirmação da mensagem é feita automaticamente quando a transação é efetivada e o processo de recuperação (reenvio das mensagens) é iniciado caso a transação seja desfeita. Entretanto, se a sessão não for especificada como transacional, a recuperação deve ser feita manualmente e há três maneiras de confirmação [SUN 01]:

- AUTO_ACKNOWLEDGE: Essa opção faz com que a sessão realize a confirmação automaticamente no momento que o cliente receber a mensagem.
- CLIENT_ACKNOWLEDGE: Neste modo, o cliente deve explicitamente confirmar o recebimento de uma mensagem ao recebê-la. A confirmação de uma mensagem implica na confirmação das mensagens previamente recebidas.
- DUPS_OK_ACKNOWLEDGE: Essa opção indica à sessão para confirmar o recebimento de mensagens automaticamente, mas não de imediato. Essa abordagem tem por objetivo diminuir a carga extra, resultante da confirmação das mensagens, uma vez que a confirmação se dá após o recebimento de um determinado número de mensagens. Entretanto, essa opção pode ocasionar entrega múltipla de mensagens.

Quando a sessão deve realizar uma recuperação (em virtude de uma transação não efetivada ou porque o cliente explicitamente iniciou este processo), a sessão irá reenviar todas as mensagens após a última mensagem confirmada [SUN 01]. As mensagens reenviadas devem conter um campo indicando que se trata de um reenvio para que o cliente possa fazer algum tipo de tratamento.

Uma outra função da sessão JMS consiste em criar consumidores e produtores de mensagens. Os consumidores são utilizados para receber mensagens de um determinado

destino e podem fazer isso de forma síncrona ou assíncrona. No modo síncrono, o consumidor permanece bloqueado a espera de uma mensagem enquanto no modo assíncrono o consumidor é notificado assim que uma mensagem é recebida no destino. O produtor é o meio pelo qual uma aplicação cliente envia mensagens para um destino. Através do produtor, pode-se especificar qual o modo de envio de mensagem que o provedor JMS irá utilizar [SUN 01]:

- Persistente: Nesse modo, todas as mensagens são armazenadas em um meio persistente para que, caso o serviço do provedor JMS venha a falhar, as mensagens não sejam perdidas. Dessa forma, as mensagens são entregues uma e apenas uma vez, o que significa que nenhuma mensagem pode deixar de ser entregue e que nenhuma mensagem será entregue em duplicidade; e
- Não-persistente: Para diminuir a carga extra decorrente do acesso a disco, as mensagens podem ser entregues sem serem armazenadas previamente. Entretanto, uma falha no provedor JMS irá ocasionar a perda de mensagens ainda não recebidas. Nesse modo, o provedor JMS irá transmitir as mensagens no máximo uma vez, ou seja, a mensagem pode não ser enviada, mas o cliente não irá receber a mensagem mais de uma vez.

2.5.2 Modelos de Envio de Mensagem

De uma maneira geral, os sistemas de envio de mensagem podem ser classificados como “um-para-um” e “um-para-muitos”. Na primeira abordagem, há apenas um destinatário para uma dada mensagem, enquanto que na segunda, vários destinatários poderão receber uma mesma mensagem. A especificação JMS aceita os dois estilos e, como é possível que um provedor implemente apenas um modelo, o JMS oferece um domínio separado pra cada modelo: *Point-to-Point* (PTP) e *Publish-and-Subscribe* (Pub/Sub) [SUN 01].

Nas aplicações PTP, o destino (objeto *Destination*) dos clientes JMS consiste de uma fila de mensagens. Esse modelo caracteriza-se como “um-para-um” porque uma mensagem enviada para uma fila só poderá ser recebida por um cliente. Entretanto, é possível haver vários produtores e consumidores usando a mesma fila. As mensagens são armazenadas nas filas até serem consumidas ou expirarem e são removidas sempre do início da fila (*first in first out*) [Flores 01]. A Figura 2.7 abaixo ilustra uma aplicação JMS seguindo o modelo PTP.

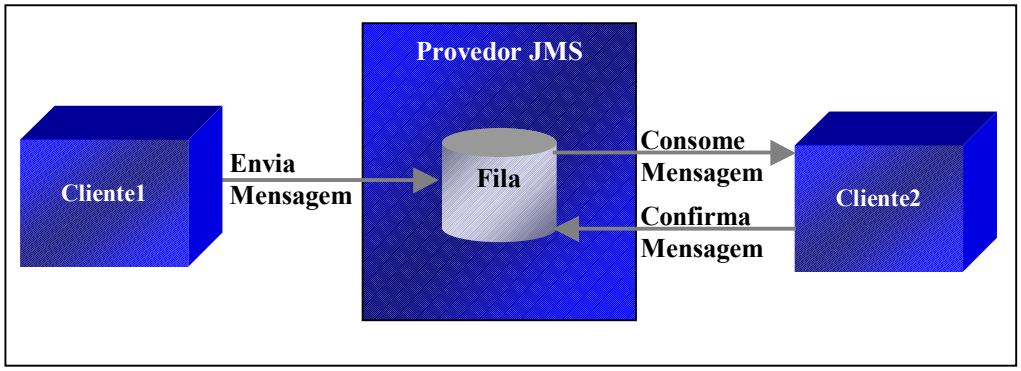

Figura 2.7 Modelo de envio de mensagem PTP

A implementação de clientes JMS seguindo o modelo PTP deve seguir o que foi descrito anteriormente, usando as interfaces do domínio *Point-to-Point*. A Figura 2.8 mostra um cliente JMS iniciando a conexão com o provedor assim como o envio e o recebimento de uma mensagem.

```
...
 QueueConnectionFactory qcf;
 Queue                  destination;
 QueueConnection        conn;
 QueueSession           session;
 QueueSender            producer;
 QueueReceiver          consumer;
 TextMessage            message;

 qcf = (QueueConnectionFactory)jndi.lookup("QConnFactory");
 destination = (Queue)jndi.lookup("Fila1");
 conn        = qcf.createQueueConnection();
 session     = conn.createQueueSession(true, 0);
 producer    = session.createSender(destination);
 message     = session.createTextMessage();
...
 //Envio de mensagem
 message.setText("Hello World!");
 producer.send(message);
...
 //Recebimento de mensagem
 consumer    = session.createReceiver(destination);
 message     = (TextMessage)consumer.receive();
...
```

Figura 2.8 Exemplo de cliente JMS utilizando o domínio PTP

Caso a aplicação JMS necessite que uma mensagem enviada por um cliente possa ser recebida por vários destinos, pode-se fazer uso do domínio *Publisher-and-Subscriber*. Nesse modelo, o emissor (*Publisher*) envia mensagens para um tópico para que os destinatários

registrados nesse tópico (*Subscribers*) possam receber essas mensagens [Flores 01]. A Figura 2.9 ilustra o funcionamento do modelo *Pub/Sub*.

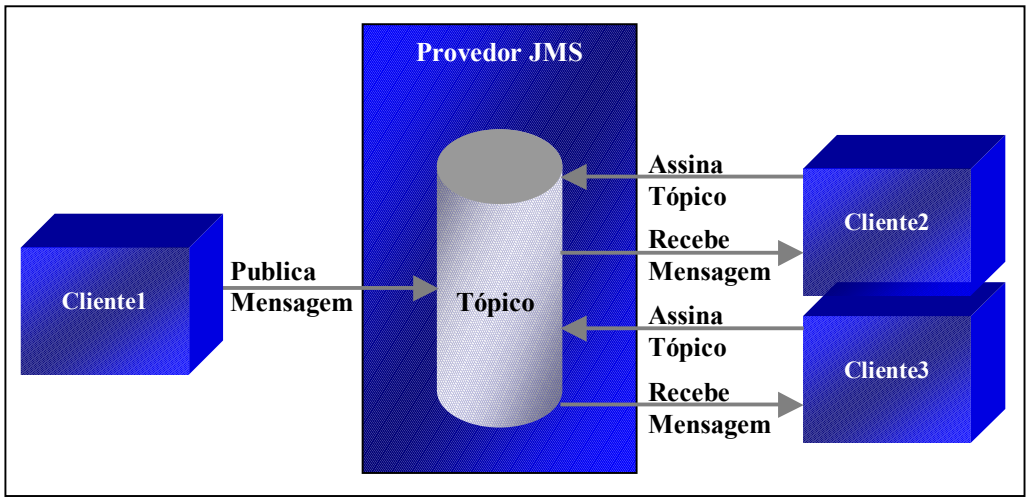

Figura 2.9 Modelo de envio de mensagem Pub/Sub

A implementação de clientes JMS seguindo o modelo *Pub/Sub* deve seguir o que foi descrito anteriormente, usando as interfaces do domínio *Publish-and-Subscribe*. A Figura 2.10 mostra um cliente JMS iniciando a conexão com o provedor assim como o envio e o recebimento de uma mensagem.

```
...
TopicConnectionFactory tcf;
Topic                  destination;
TopicConnection        conn;
TopicSession           session;
TopicPublisher         producer;
TopicSubscriber        consumer;
TextMessage            message;

tcf = (TopicConnectionFactory)jndi.lookup("TConnFactory");
destination = (Topic)jndi.lookup("Topico1");
conn        = tcf.createTopicConnection();
session     = conn.createTopicSession(true, 0);
producer    = session.createPublisher(destination);
message     = session.createTextMessage();
...
//Envio de mensagem
message.setText("Hello World!");
producer.publish(message);
...
//Recebimento de mensagem
consumer    = session.createSubscriber(destination);
message     = (TextMessage)consumer.receive();
```

Figura 2.10 Exemplo de cliente JMS utilizando o domínio Pub/Sub

Por padrão, no modelo *Pub/Sub*, uma mensagem só permanece no tópico o tempo suficiente para que seja entregue aos clientes [SUN 01]. Dessa forma, um cliente só irá receber as mensagens que forem enviadas ao tópico após o registro no tópico (*subscription*) ter sido efetuado. Isso implica que as mensagens enviadas a um tópico que não possui consumidores registrados serão perdidas. Para garantir que as mensagens enviadas serão realmente recebidas pelos clientes no modelo *Pub/Sub*, o produtor pode especificar o modo de envio como persistente e o consumidor pode criar um registro durável com o tópico. Com o custo de aumentar a carga extra da comunicação, o desenvolvedor pode criar um consumidor durável (*durable subscriber*). Esse consumidor cria um registro durável com um identificador único que é mantido pelo provedor JMS. Registros subseqüentes utilizando esse mesmo identificador poderão dar continuidade com o registro criado inicialmente utilizando o estado deixado pelo consumidor anterior. Caso não exista nenhum consumidor ativo no momento, o provedor irá armazenar as mensagens que estão sendo enviadas para o tópico até que elas sejam consumidas ou expirem. Enquanto que em um registro não durável o registro é terminado quando o consumidor é desconectado, com o registro durável, o registro é mantido até que um consumidor encerre o registro explicitamente [Haase 01] como ilustra a Figura 2.11.

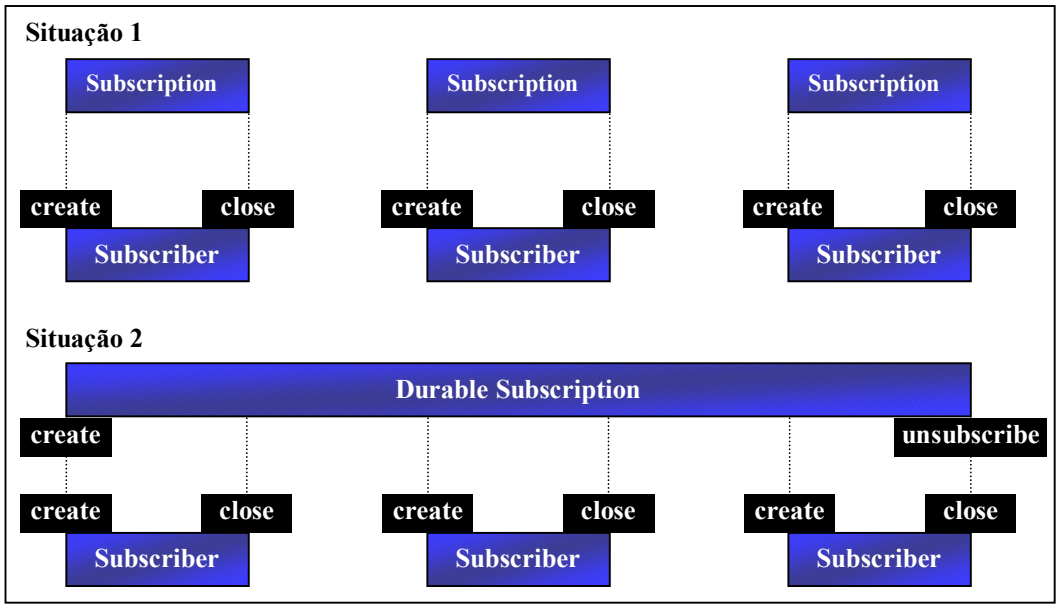

Figura 2.11 Registro não durável e registro durável

Na Figura 2.11, enquanto que na primeira situação são criados vários registros (um para cada consumidor), a segunda situação ilustra um registro durável sendo compartilhado por vários consumidores ao longo do tempo. Com o registro durável, o terceiro consumidor irá receber as mensagens que foram enviadas ao tópico antes de seu registro.

2.5.3 Limitações

Em muitos casos, a aplicação JMS não pode tolerar que mensagens sejam perdidas ou que uma determinada mensagem seja entregue mais de uma vez. Para isso, a especificação permite que uma mensagem possa ser armazenada no destino antes de ser recebida pelos clientes (modo persistente de envio) e, para o caso do modelo *Publish-and-Subscribe*, o próprio provedor irá garantir que todos os destinatários irão receber a mensagem se for feito um registro durável. Além disso, pode-se fazer uso de sessões transacionais para que os procedimentos de reenvio sejam efetuados de forma transparente para os clientes. Entretanto, a disponibilidade da aplicação ficará comprometida caso o serviço do provedor JMS torne-se indisponível. Mesmo sendo possível que mais de uma instância do provedor possa estar em execução em máquinas diferentes, a coordenação de utilização dos serviços das várias instâncias fica sob responsabilidade dos clientes JMS, uma vez que a especificação não inclui mecanismos para que as instâncias do provedor possam cooperar para agir como um único serviço [SUN 01].

Como foi explicado anteriormente, o primeiro passo que o cliente JMS deve executar é obter referências para um Destino e um Construtor de Conexão de um provedor JMS através do serviço JNDI. A partir daí, o cliente irá obter uma conexão e utilizar os serviços do provedor do qual ele obteve os objetos auxiliares. Caso a instância do provedor JMS torne-se indisponível, o cliente receberá um erro quando acessar os objetos obtidos anteriormente e deverá explicitamente localizar outra instância em execução em algum servidor da rede. Entretanto, mesmo com esse tratamento por parte da aplicação, mensagens poderão ser perdidas caso o meio persistente que o provedor utilizava não possa ser acessado. Isso pode ocasionar inconsistência para aplicação, uma vez que o cliente pode enviar várias mensagens com sucesso para o destino e o provedor pode falhar antes de iniciar a entrega para os consumidores. Conseqüentemente, os clientes passarão a receber mensagens de outro provedor o qual não poderá acessar as mensagens armazenadas no primeiro.

Todas as mensagens do JMS possuem um campo chamado JMSMessageID que serve para identificar as mensagens enviadas pelo provedor [SUN 01]. É de responsabilidade do provedor garantir que cada mensagem possua um valor diferente para esse campo, para que a consistência das aplicações seja mantida. Entretanto, caso uma segunda instância do provedor seja usada para substituir uma instância em uso que falhou, a nova instância poderá gerar identificações repetidas uma vez que ela não tem acesso às identificações que foram geradas.

2.6 Java Remote Method Invocation

Através do RMI, uma aplicação cliente em Java pode utilizar um objeto instanciado em uma máquina virtual diferente da sua e possivelmente executando em um outro computador da rede. Na especificação RMI, qualquer objeto cujos métodos podem ser invocados de uma outra máquina virtual é chamado de "objeto remoto" [Roman 99]. A localização física dos objetos remotos e dos clientes que os utilizam é irrelevante nessa tecnologia. Essa característica permite que o objeto remoto possa ser utilizado da mesma maneira tanto por objetos clientes da sua própria máquina virtual quanto por objetos em execução em um ponto distinto da rede. O RMI surgiu para oferecer um nível maior de abstração para aplicações que precisam trocar informações pela rede. No lugar de enviar mensagens utilizando diretamente o protocolo de rede, a aplicação pode invocar métodos de um objeto localizado em uma máquina virtual diferente da sua da mesma maneira que invoca qualquer outro objeto.

2.6.1 Arquitetura

Para permitir que um objeto possa ser referenciado a partir de uma outra máquina virtual, o desenvolvedor deve criar uma interface que herde de `java.rmi.Remote`. Nessa interface, devem ser declarados todos os métodos que poderão ser invocados remotamente. Na Figura 2.12, está definida a interface remota para um objeto remoto que irá somar dois valores.

```
import java.rmi.Remote;
import java.rmi.RemoteException

public interface Calculador extends Remote
{
  public int adicionar(int n1, int n2) throws RemoteException;

}
```

Figura 2.12 Exemplo de interface remota

Definida a interface remota, o desenvolvedor deve criar uma classe que implemente essa interface, provendo o código dos métodos nela definidos. Essa classe, normalmente, herda algumas funcionalidades básicas do serviço RMI de outras classes, como por exemplo da classe `java.rmi.server.UnicastRemoteObject`. A implementação do objeto remoto para o exemplo proposto é ilustrada na Figura 2.13.

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class CalculadorImpl extends UnicastRemoteObject
                            implements Calculador
{
  public int adicionar(int n1, int n2) throws RemoteException
  {
    return n1 + n2;
  }
}
```

Figura 2.13 Exemplo de implementação do objeto remoto

Uma das grandes vantagens da tecnologia RMI é que todo o processo de comunicação entre os objetos fica transparente para os desenvolvedores do objeto remoto e do cliente. Dessa forma, a implementação do objeto remoto consiste apenas na lógica da aplicação. Para que isso seja possível, a infra-estrutura do RMI oferece objetos auxiliares que irão permitir que uma invocação de método feita pelo objeto cliente seja recebida pelo objeto remoto através do protocolo de rede [Campione et al. 98]. Um desses objetos auxiliares é o *stub* que age como uma "referência remota" à instância do objeto remoto. Este objeto (gerado a partir da compilação da interface remota e da implementação do objeto remoto) residirá na máquina virtual do cliente e irá interceptar as chamadas aos objetos remotos. O *stub* será responsável por interagir diretamente com o protocolo de rede para enviar as requisições (contendo a invocação do método) para a implementação do objeto remoto e receber desta a resposta do método com ilustrado na Figura 2.14.

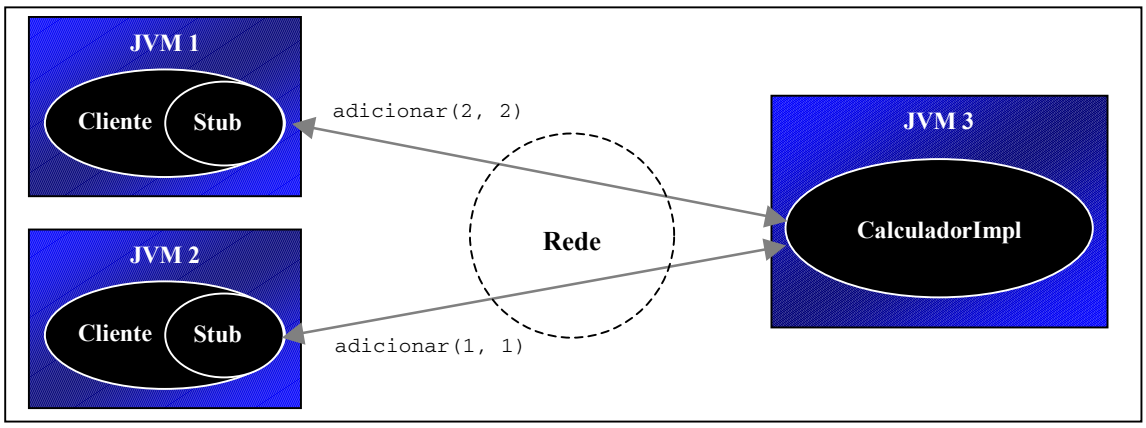

Figura 2.14 Comunicação entre o cliente e o objeto remoto

Para que o objeto remoto possa ser invocado por objetos clientes em execução em outras máquinas virtuais, ele deve ser registrado em um serviço de nomes como o RMIRegistry ou JNDI [SUN 99]. Esse registro consiste em associar um nome à instância do objeto remoto, para que os objetos clientes possam obter uma referência remota para esse objeto a partir desse nome. Quando um objeto cliente deseja utilizar um objeto remoto, em primeiro lugar é estabelecida uma sessão com o serviço de nomes. Em seguida, a aplicação cliente consulta o serviço de nomes informando o nome do objeto remoto desejado e o resultado dessa consulta é o *stub* do objeto remoto. A partir daí, os métodos que o cliente invocar serão enviados pelo *stub* para a instância real do objeto remoto para serem processados. Um exemplo de invocação ao objeto remoto criado está ilustrado na Figura 2.15.

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class Cliente
{
  public static void main(String[] args) throws Exception
  {
    //Conexão com o serviço de nomes JNDI
    Context ctx = new InitialContext();

    //Obtém uma referência remota ao objeto Calculador com
    //base no nome com o qual ele foi registrado
    Calculador objetoRemoto = (Calculador)ctx.lookup("Calc");

    //Invocação do método remoto
    int resultado = objetoRemoto.adicionar(2, 2);
  }
}
```

Figura 2.15 Exemplo de invocação do objeto remoto

Como o *stub* obtido é um objeto que implementa a mesma interface remota implementada pelo objeto remoto, o objeto cliente tem a "ilusão" de estar referenciado o objeto remoto, o que torna a utilização do *stub* transparente para o cliente.

2.6.2 Limitações

Através dos *stubs*, as aplicações clientes podem acessar os objetos remotos instanciados em máquinas virtuais diferentes da sua como se fossem objetos locais. Entretanto, de acordo com a especificação RMI, estes objetos auxiliares não possuem mecanismos para detectar a falha das instâncias referenciadas de forma automática e transparente. De uma maneira geral, é possível que, após o cliente obter uma referência remota para o objeto, a máquina virtual do objeto remoto apresente uma falha. Na próxima vez que o cliente invocar um método, ele receberá do *stub* uma exceção indicando um problema de comunicação com o objeto remoto. Isso irá obrigar a aplicação cliente a recuperar novamente uma referência a um objeto remoto em outra máquina virtual. Este tratamento extra por parte do cliente é devido ao fato de que os objetos *stubs* só podem referenciar uma instância em uma determinada máquina virtual, mesmo havendo outras instâncias em pontos distintos da rede.

2.7 Enterprise JavaBeans

Com a utilização cada vez maior dos sistemas cliente/servidor, percebeu-se que esse paradigma não seria viável para o ambiente corporativo de grande porte e para a Internet, devido à dificuldade de atualização de versão dos aplicativos e à limitação de escalabilidade. Nesse modelo, há apenas duas camadas de *software*: a aplicação, contendo a interface com o usuário, e a lógica de negócio, constituindo a primeira camada; e o banco de dados, representando a segunda. Isso faz com que sempre que for preciso fazer uma alteração na lógica do sistema, a aplicação deve ser redistribuída para todas as estações. Além disso, por mais sofisticado que seja o sistema gerenciador de banco de dados, chega-se a um ponto em que o servidor não suporta atender a todas as conexões dos clientes simultaneamente.

Devido à necessidade de atender a uma escala maior de usuários e ao aumento da complexidade dos sistemas, foi necessário particionar as aplicações em três ou mais camadas de *software* independentes. Esse novo paradigma permitiu que os módulos da interface, da lógica de negócio e do acesso a dados fossem desenvolvidos e testados independentemente e reaproveitados em vários sistemas. Além disso, as aplicações puderem atender a um número maior de usuários, uma vez que, nas máquinas clientes, estaria em execução apenas um programa extremamente leve e pequeno, responsável apenas pela apresentação e validação dos dados. Nos servidores é que residiriam as rotinas que executam um processamento maior, tornando possível uma melhor gerência de balanceamento de carga. A Figura 2.16 ilustra o modelo multicamada.

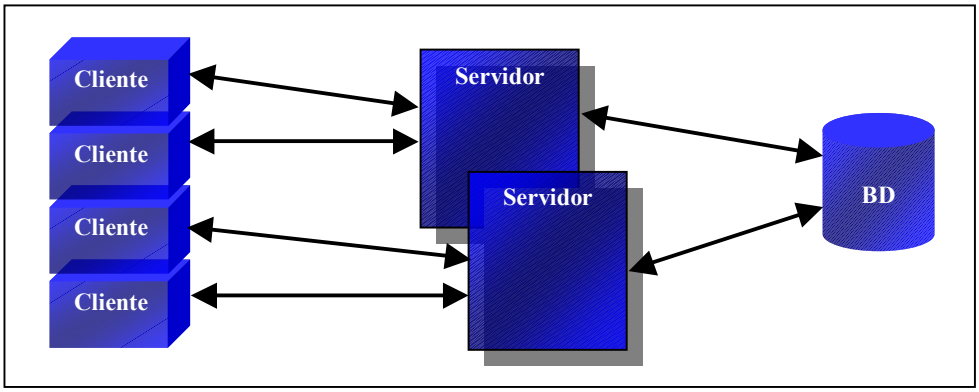

Figura 2.16 Modelo de aplicação multicamada

Uma vez que esse modelo requer tanto a distribuição de *software* entre os servidores quanto a comunicação destes com as aplicações da interface, surgiram várias tecnologias para permitir que fossem desenvolvidos sistemas que não necessitassem interagir diretamente com as mensagens do protocolo de rede, o que implicaria num desenvolvimento com um nível de abstração muito baixo. Dentre elas, destacam-se o Java RMI e o CORBA, onde é possível referenciar um objeto de uma outra máquina como se ele fosse local [Campione et al. 98]. Apesar de essas tecnologias oferecerem uma base confiável para a comunicação entre objetos, tornou-se necessária uma plataforma que lidasse com outros aspectos como processamento de transações distribuídas, balanceamento de carga, gerenciamento de conexões com o banco de dados, ciclo de vida dos objetos, concorrência e segurança, oferecendo esses recursos de forma transparente para o desenvolvedor. Nesse contexto é que está o Enterprise JavaBeans (EJB), uma especificação da SUN Microsystems que reúne várias tecnologias já usadas como JNDI, RMI/CORBA, JDBC (Java Database Connectivity), JTS (Java Transaction Service) e

JTA (Java Transaction API), para oferecer um modelo de desenvolvimento baseado em componentes servidores [Thomas 98]. Utilizando essa tecnologia, percebe-se uma melhor adaptabilidade, uma vez que os componentes podem ser configurados sem a necessidade de alteração do código fonte. Além disso, o desenvolvimento da aplicação torna-se mais simples por que o desenvolvedor não precisa lidar com aspectos mais complicados da infra-estrutura.

2.7.1 Desenvolvimento Baseado em Componentes

Devido à semelhança entre os nomes, costuma-se confundir os termos JavaBeans e Enterprise JavaBeans como se fossem a mesma tecnologia. O JavaBeans refere-se ao modelo de componentes do Java, ou seja, é uma especificação para criação de "pedaços de aplicação" com suas propriedades e eventos que serão combinados para desenvolver aplicações gráficas [Campione et al. 98] da mesma forma que o MFC (Microsoft Foundation Classes) da Microsoft e o VCL (Visual Component Library) da Borland. Já o EJB também é uma especificação que define um modelo de criação de componentes, mas não visuais e sim componentes servidores [Thomas 98]. Os componentes EJB ficam armazenados em um ou mais servidores de aplicações e constituem o *framework* de serviços da empresa, podendo ser utilizados por qualquer sistema. Com essa tecnologia de componentes, pode-se fazer uso de uma programação declarativa onde o componente possui propriedades que, quando modificadas, podem alterar seu comportamento. Da mesma forma que os componentes gráficos como botões e campos de texto são encapsulados ou agrupados em estruturas maiores como formulários, páginas HTML ou outros componentes, a plataforma EJB provê uma estrutura chamada contêiner que também oferece para os componentes servidores um contexto de execução, além de gerenciamento e controle de serviços [Thomas 98].

Com o Enterprise JavaBeans é possível criar componentes persistentes que representam as entidades do projeto (*entity beans*) e componentes que contêm apenas lógica de negócio (*session beans*). Enquanto que os *entity beans* oferecem uma visão orientada a objetos das tabelas relacionais do projeto, cada *session bean* representa uma ação do sistema. Assim, os desenvolvedores que irão compor as aplicações finais apenas têm de criar a interface com o usuário (que pode ser um *applet*, uma aplicação gráfica em Java ou outra linguagem, *servlet* ou JSP) que passam os dados de entrada para os *session beans* que manipula os *entity beans* quando necessário como ilustrado na Figura 2.17.

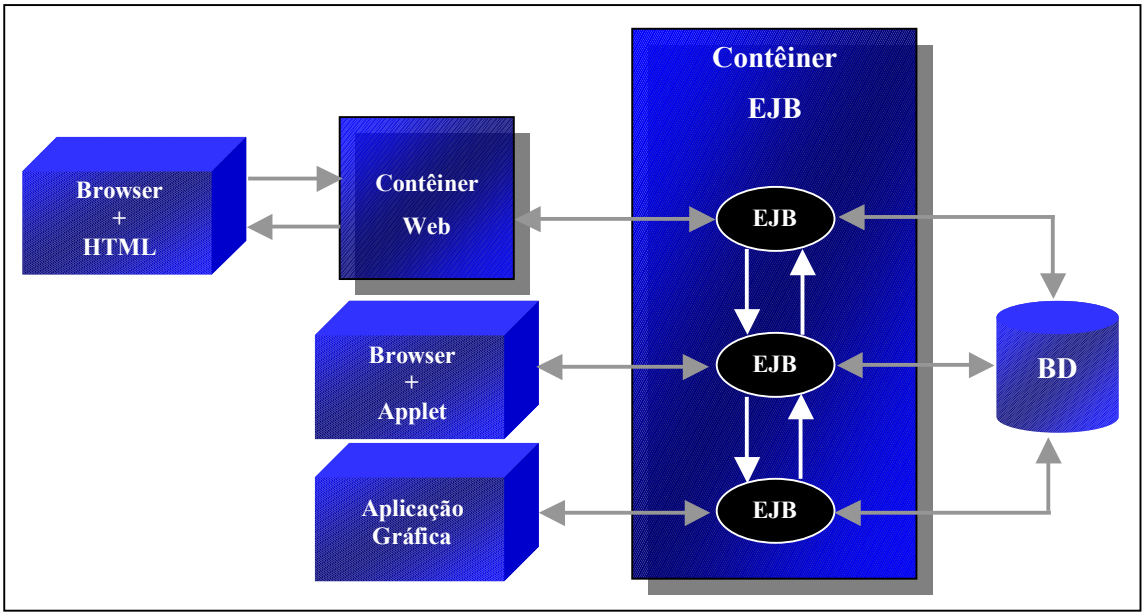

Figura 2.17 Modelo multicamada utilizando a arquitetura EJB

Além de permitir um alto nível de reaproveitamento, essa tecnologia oferece uma completa transparência no que se refere a controle transacional [SUN 99-1]. Pode-se, por exemplo, utilizar um componente que invoca outro componente hospedado em um segundo servidor numa mesma transação sem que o cliente crie explicitamente o contexto transacional. Como ilustrado na Figura 2.18, fica sob responsabilidade das instâncias do servidor EJB efetivar ou desfazer a transação de forma transparente para a aplicação cliente [Thomas 98].

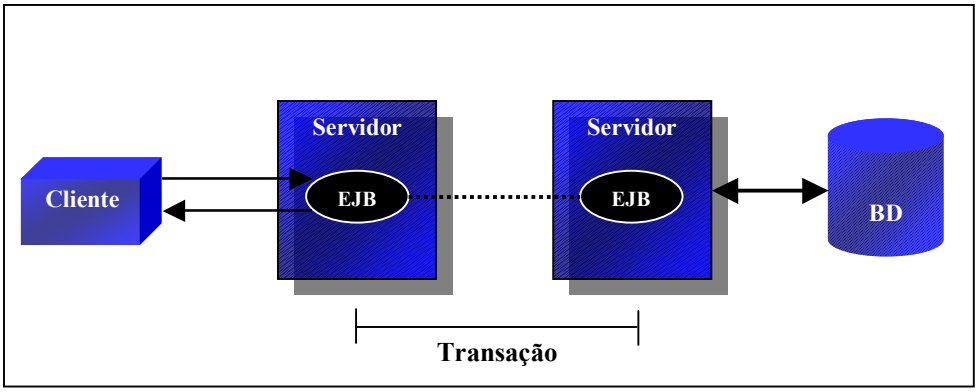

Figura 2.18 Controle transacional entre servidores

Da mesma forma, há o controle automático do ciclo de vida e do estado do componente entre as várias requisições. Além disso, a especificação garante total compatibilidade entre servidores, ou seja, um mesmo componente pode ser instalado em qualquer servidor que siga a especificação.

2.7.2 Arquitetura de um Enterprise JavaBean

A primeira etapa na criação de um EJB (seja um *session bean* ou um *entity bean*) é definir como será sua interface [Roman 99], ou seja, quais são os métodos que o componente poderá executar. Através da interface, o cliente tem conhecimento dos serviços que o componente está disponibilizando. Na definição desses métodos, pode-se usar todos os tipos primitivos da linguagem Java como também tipos definidos pelo usuário [SUN 99-1].

O passo seguinte é a criação da interface *home* [Roman 99]. Essa interface servirá para que o cliente consiga uma referência remota para o EJB, ou seja, ela fará o papel de "fabricante" do componente. Os métodos dessa interface irão criar novas instâncias e devem obrigatoriamente se chamar `create` [SUN 99-1]. Esses métodos podem ser definidos para receber parâmetros que servirão para prover o componente com um estado inicial e devem ter como retorno a interface do componente. Com isso, já é possível implementar o *bean* propriamente dito, ou seja, implementar os métodos declarados na sua interface.

De acordo com a especificação EJB, a comunicação entre os componentes e os clientes que os acessam deve ser feita através das tecnologias RMI ou RMI-IIOP (Remote Method Invocation – Internet Inter-ORB Protocol) sendo que este último seria utilizado para prover interoperabilidade com o padrão CORBA [SUN 99-1]. Como a especificação EJB é uma arquitetura baseada em componentes servidores, faz-se necessário o uso da tecnologia RMI para permitir que os componentes hospedados nos servidores de aplicações possam ser utilizados da mesma maneira, e de forma transparente, tanto por aplicações cliente, quanto por outros componentes [Roman 99]. Para cada componente EJB em execução, há um objeto remoto para o *home* e um ou vários objetos remotos para a classe que implementa o *bean* desse componente [SUN 99-1]. Quando um componente é instalado em um servidor, automaticamente é disponibilizado um objeto remoto para o *home* no serviço JNDI, para que a aplicação cliente possa, por meio desse serviço, obter uma referência remota (*stub*) para o *home* desse componente [SUN 99-2]. Como a instância real do objeto *home* encontra-se no servidor EJB e não na máquina virtual do cliente, a aplicação cliente irá utilizá-lo para obter referências remotas (*stubs*) aos *beans* que estarão instanciados no servidor EJB [Kassem et al. 00]. Quando o cliente requisita a criação de um novo componente, o objeto *home* cria uma nova instância do *bean* e retorna o seu *stub* correspondente. Para os componentes *entity*

beans, é possível também localizar um conjunto de instâncias. Nesse caso, o objeto *home* instancia objetos com base nos estados já armazenados no banco de dados e retorna um conjunto de referências para essas instâncias [SUN 99-1]. Através das referências remotas obtidas do objeto *home*, a aplicação cliente poderá utilizar o *bean* desejado como se estivesse invocando métodos de um objeto local. O *stub* do *bean* será responsável por enviar todas as requisições à instância no servidor EJB para que lá elas possam ser processadas.

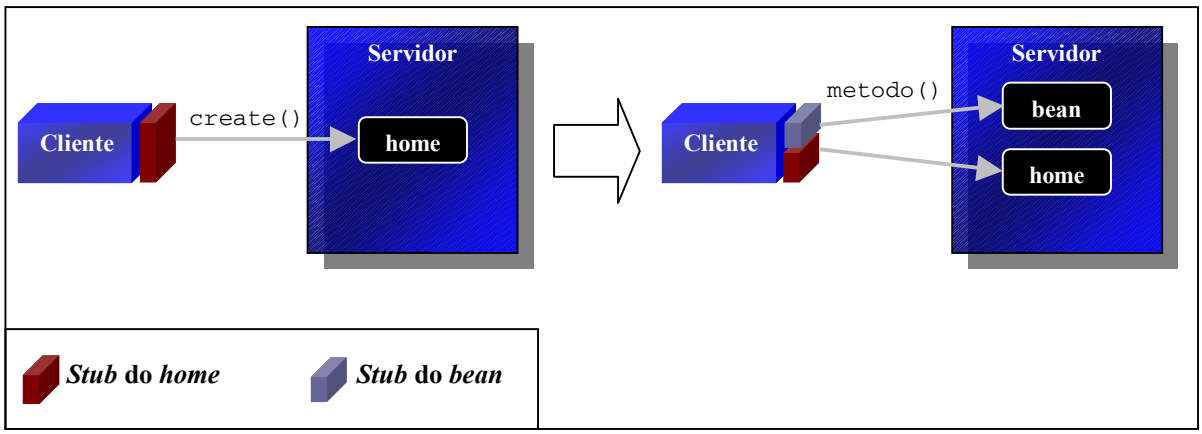

Figura 2.19 Criação de uma instância de um componente EJB

A Figura 2.19 ilustra o processo de obtenção de uma referência a um *bean*. No primeiro passo, o cliente, por meio da *stub* do *home*, invoca o método `create` para que seja criada uma nova instância do *bean* no servidor. A partir daí, o cliente irá utilizar a *stub* do *bean* obtida para invocar métodos desse componente.

Após implementar as classes do componente, é necessário criar ainda o seu descritor de instalação (*deployment descriptor)*, que consiste em um arquivo no formato XML (*Extensible Markup Language*) que contem a identificação do componente e todas as propriedades que ele vai necessitar durante o seu processamento no servidor [Roman 99]. Com o descritor de instalação, é possível alterar o comportamento do componente (contexto transacional, banco de dados a utilizar, etc.) em tempo de instalação [Thomas 98]. Geralmente, os servidores EJB disponibilizam ferramentas gráficas para gerar o descritor de instalação que pode conter propriedades adicionais além das que obrigatoriamente devem estar presentes [SUN 99-1].

2.7.3 O Entity Bean

Do ponto de vista da aplicação cliente, o *entity bean* é um tipo de componente que representa uma visão orientada a objetos de uma ou mais entidades que são armazenadas em um meio persistente [Thomas 98]. Esse tipo de componente é bastante usado para representar uma linha de uma tabela em um banco de dados. Como exemplo, para uma tabela CONTAS que armazena as contas dos correntistas de um banco, é possível criar um *entity bean* correspondente, em que cada instância desse componente representa uma determinada conta corrente nessa tabela, como ilustrado na Figura 2.20.

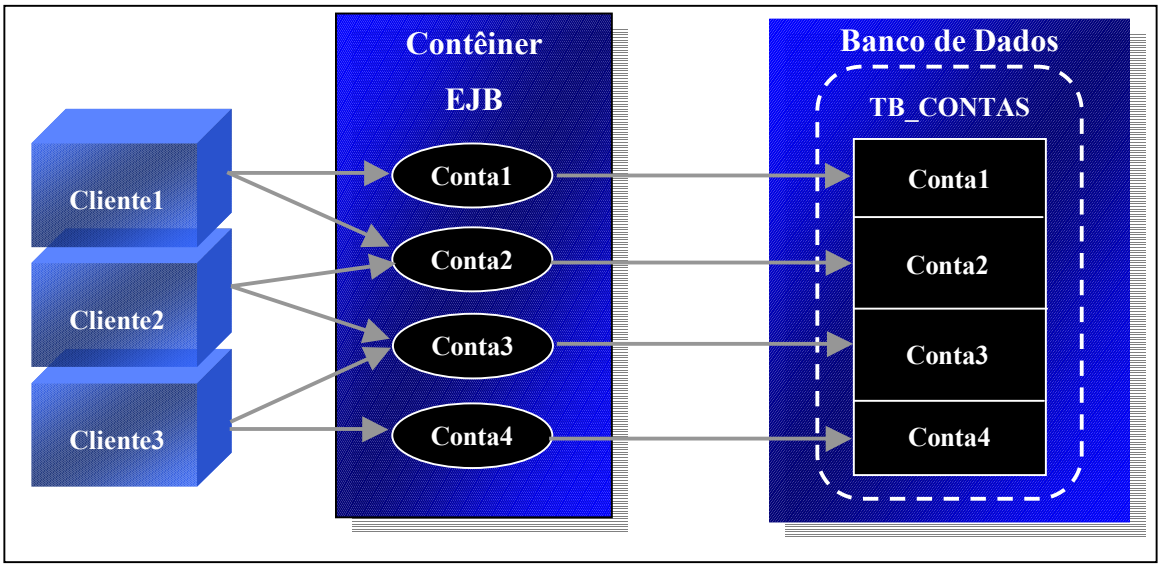

Figura 2.20 *Entity bean* da tabela CONTAS

Para implementar um *entity bean* deve-se seguir o que foi explicado anteriormente. Em primeiro lugar é necessário criar a sua interface. É recomendável que na interface desse componente conste métodos que encapsulem a lógica de alteração dos atributos do EJB. Por exemplo: levando em consideração que a tabela CONTAS tenha os campos NUMERO(char(5)) e SALDO(float), a interface do componente correspondente poderia ser como ilustrado na Figura 2.21.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Conta extends EJBObject
{
  public void saque(float quantia)
    throws RemoteException, SaqueException;

  public void deposito(float quantia)
    throws RemoteException, DepositoException;

  public String obterNumero()throws RemoteException;

  public float obterSaldo() throws RemoteException;
}
```

Figura 2.21 Interface remota do *entity bean* Conta

Nesse exemplo, a alteração do valor do saldo só poderá ser feita através dos métodos "`saque`" e "`deposito`" o que garante que as validações do projeto serão efetuadas antes dos dados serem alterados oferecendo uma maior funcionalidade para a aplicação. Feito isso, deve-se criar a chave primária do *entity bean* que consiste em uma classe contendo os atributos que compõe a chave primária da entidade na tabela [Roman 99]. Essa classe será utilizada tanto pelo cliente quanto pelo servidor para identificar uma determinada instância do componente [SUN 99-1]. Para o exemplo proposto, a chave poderia ser como ilustrado na Figura 2.22.

```
public class ContaPK implements java.io.Serializable
{
  //Os atributos que compõe a chave devem ser públicos
  public String numero;

  //Deve sempre haver o construtor padrão
  public ContaPK(){}

  public ContaPK(String numero)
  {
    this.numero = numero;
  }
}
```

Figura 2.22 Chave primária do *entity bean* Conta

O próximo passo na criação de um *entity bean* consiste na definição da interface *home* do componente. Quando se trata de um *entity bean*, essa interface pode conter tanto métodos que localizam uma instância quanto os que irão criar uma nova [Roman 99]. Para os métodos que servirão para criar uma instância, devem ser definidos parâmetros que servirão para

inicializar o estado do componente e criar um novo registro na tabela correspondente. Já os métodos que localizarão uma instância, devem receber os valores necessários para o critério de pesquisa definido para o método. Esses métodos podem tanto retornar uma única instância do componente quanto uma coleção deles [SUN 99-1]. A interface *home* do exemplo poderia ser definida como na Figura 2.23.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException
import java.rmi.RemoteException;
import java.util.Collection;

public interface ContaHome extends EJBHome
{
  public Conta create(String numero)
      throws CreateException, RemoteException;

  public Conta findByPrimaryKey(ContaPK pk)
      throws FinderException, RemoteException;

  public Collection findSaldoMaiorQue(float saldoParam)
      throws FinderException, RemoteException;
}
```

Figura 2.23 Interface *home* do *entity bean* Conta

Com a interface *home* dessa forma, as aplicações clientes podem fazer duas consultas: uma pela chave primária, e outra que devolve todos os registros (*entity beans*) que possuem um saldo maior que o que foi passado como parâmetro para o método `findSaldoMaiorQue(float saldoParam)`.

Após a definição das interfaces, já é possível implementar o *bean* propriamente dito. Para isso deve-se escolher como será feita a gerência de sua persistência: CMP (*Container-Managed Persistence*) ou BMP (*Bean-Managed Persistence*) [Thomas 98]. No primeiro caso, o desenvolvedor do componente não precisa implementar a lógica para acessar o banco de dados. Toda a implementação JDBC fica a cargo do servidor por meio do contêiner [SUN 99-1]. Já no BMP, o desenvolvedor é que deve prover o acesso ao banco de dados. Para o componente do exemplo proposto, usando CMP, o desenvolvedor necessitaria implementar apenas o trecho ilustrado na Figura 2.24.

```
import javax.ejb.EntityBean;
import javax.ejb. CreateException
import java.rmi.RemoteException;

public class ContaBean implements EntityBean
{
.
.
.
  //Atributos do bean. Devem corresponder aos campos da tabela
  public String numero;
  public float saldo;
.
.
.
  //Chamado quando o cliente invoca o create da home.
  public void ejbCreate(String numero)
    throws CreateException, RemoteException
  {
    this.numero = numero;
    this.saldo = 0;
  }
.
.
.
  //Método declarado na interface do bean
  public float obterSaldo() throws RemoteException
  {
    return saldo;
  }

  //Método declarado na interface do bean
  public void saque(float quantia)
     throws RemoteException, SaqueException
  {
    if (quantia > saldo)
      throw new SaqueException("Saldo insuficiente");
    saldo -= quantia;
  }

  //Método declarado na interface do bean
  public void deposito(float quantia)
     throws RemoteException, DepositoException
  {
    if (saldo < 0)
      throw new DepositoException("Valor de depósito inválido");
    saldo += quantia;
  }
}
```

Figura 2.24 Implementação do *entity bean* Conta

Com o componente implementado, já é possível instalá-lo no servidor definindo seu descritor de instalação [Roman 99]. Nele será especificado o seu nome, que o identificará no serviço de nomes (JNDI), o comportamento transacional, o nível de isolamento para tratar da concorrência, além das características de sua persistência: a tabela à qual o *entity bean* se

refere, os atributos dele que serão gerenciados automaticamente pelo servidor e o mapeamento destes com os campos da tabela [SUN 99-1]. Quando o *entity bean* for CMP, também é necessário incluir os critérios dos métodos de pesquisa de forma declarativa. A criação do descritor de instalação varia de acordo com o servidor EJB. Alguns providenciam ferramentas gráficas que funcionam como um assistente na criação do arquivo e, ao final, geram as classes que formarão o contêiner do componente.

2.7.4 O Session Bean

O outro tipo de EJB é o *session bean*, componente que realiza o processamento da lógica do negócio no servidor EJB em nome do cliente. Diferente do *entity bean* onde uma mesma instância pode ser acessada por vários clientes, cada instância desse tipo de componente está associada a um cliente [Thomas 98]. De uma maneira geral, o *session bean* contem apenas lógica de negócio e possivelmente um estado. Isso torna possível implementar clientes que tratem apenas da interface com o usuário contendo chamadas a *session beans* que manipulariam os dados (*entity beans*). Um exemplo desse modelo poderia ser um *session bean* responsável por fazer a transferência entre duas contas utilizando o *entity bean* apresentado anteriormente como ilustrado na Figura 2.25.

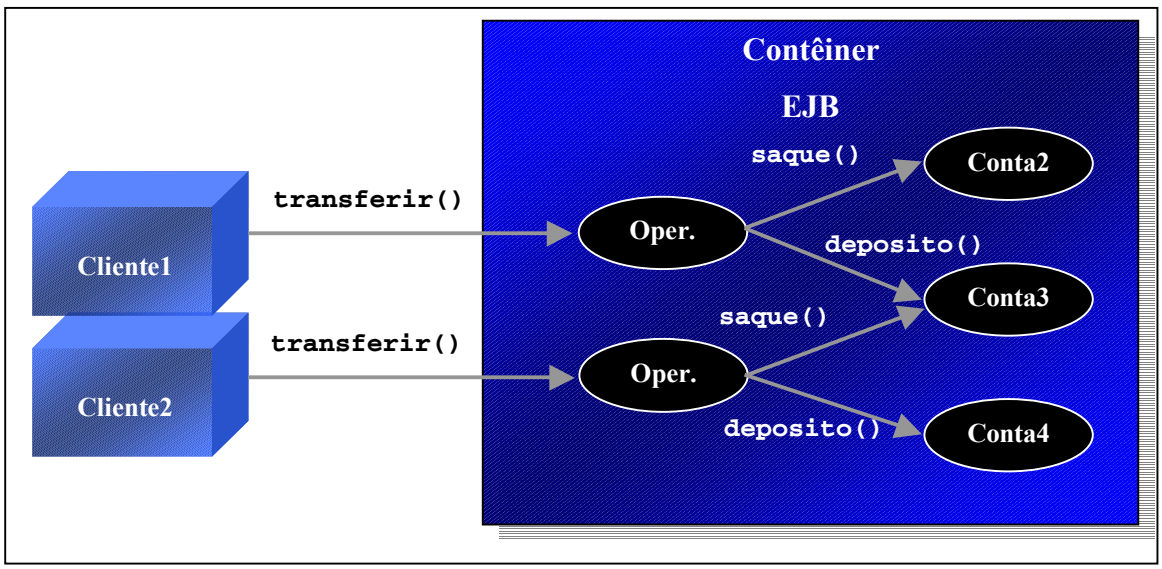

Figura 2.25 Funcionamento do *session bean*

Antes de iniciar a implementação de um *session bean* deve-se escolher qual será seu tipo: *stateful* ou *stateless* [Roman 99]. No primeiro caso, o *session bean* possui um estado

(dados internos) que serão preservados ao longo do seu ciclo de vida, enquanto que no *stateless*, não há estado há ser preservado [SUN 99-1]. Quando o cliente invoca mais de uma vez métodos de uma mesma referência de um *bean* do tipo *stateless*, é bastante provável que não seja a mesma instância no servidor que realizará o processamento para cada método invocado [Roman 99]. Como o *stateless* não possui nenhum estado, não haverá nenhum problema se diferentes instâncias estiverem respondendo para um cliente. Essa diferença de tipos existe apenas para aproveitar melhor os recursos da máquina, uma vez que não é necessário ter um número tão alto de objetos *stateless* no *pool* do servidor EJB.

É importante destacar que constitui um erro se a interface *home* de um *session bean stateless* possuir um método "`create`" com parâmetros [SUN 99-1], porque não há nenhuma garantia que esses dados passados serão preservados quando for chamado um método dessa instância criada. Para o exemplo proposto a interface *home* do *session bean* poderia ser definida como na Figura 2.26.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface OperacaoHome extends EJBHome
{
  public Operacao create() throws CreateException, RemoteException;
}
```

Figura 2.26 Interface *home* do *session bean* Operação

Já a interface do *bean* poderia ser a ilustrada na Figura 2.27.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Operacao extends EJBObject
{
  public void transferir(String numContaOrg,
                         String numContaDest,
                         float quantia)
      throws RemoteException, TransferenciaException;
}
```

Figura 2.27 Interface remota do *session bean* Operacao

Nesse exemplo, o método que irá executar a lógica já recebe os números das contas e o valor a ser transferido, o que dispensa a necessidade de um estado interno para o *session*

bean. A implementação desse método consiste em localizar as contas (*entity beans*) e invocar os métodos "`saque`" e "`deposito`" com está ilustrado na Figura 2.28.

```
import javax.ejb.SessionBean;
import java.rmi.RemoteException;
import javax.naming.Context;

public class OperacaoBean implements SessionBean
{
...
  //Método declarado na interface do bean
  public void transferir(String numContaOrg,
                         String numContaDest,
                         float quantia)
     throws RemoteException, TransferenciaException
  {
    try
    {
      //Localiza um objeto home do bean Conta
      ContaHome cHome = (ContaHome)jndiCtx.lookup("Conta");
      ContaPK pkOrig = new ContaPK(numContaOrg);
      Conta cOrig = cHome.findByPrimaryKey(pkOrig);
      ContaPK pkDest = new ContaPK(numContaDest);
      Conta cDest = cHome.findByPrimaryKey(pkDest);
      cOrig.saque(quantia);
      cDest.deposito(quantia);
    }
    catch (Exception e)
    {
      throw new TransaferenciaException("Falha na operação de
transferência");
    }
  }
}
```

Figura 2.28 Implementação do *session bean* Operação

Como neste caso, o uso do componente consiste em apenas um método e não há nenhum dado que necessite ser guardado, o seu descritor de instalação deve indicar que ele é do tipo *stateless*.

Nesses exemplos, é possível destacar, além da facilidade de acesso a dados, a transparência em relação a controle transacional. Uma possível configuração no descritor de instalação dos componentes criados poderia especificar que os métodos do *entity bean* devem ser executados sempre em um contexto transacional, ou seja, com uma transação já iniciada, enquanto que no método do *session bean* poderia ser especificado que este sempre iniciaria uma nova transação ao ser invocado. Com isso, quando a aplicação cliente chamar o método "`transferir`", será criada uma transação automaticamente, e esta será usada pelos métodos

"saque" e "deposito" do *entity bean*. Assim, caso ocorra uma exceção durante o processamento (no método "deposito", por exemplo), tudo que havia sido processado será desfeito.

A implementação da aplicação cliente consiste apenas em localizar a interface *home* através do JNDI, criar um *session bean* e chamar seu método como ilustrado na Figura 2.29.

```
import javax.naming.Context;
import javax.naming.InitialContext;

public class Cliente
{
  public static void main(String[] args) throws Exception
  {
    Context ctx = new InitialContext();
    OperacaoHome opHome = (OperacaoHome)ctx.lookup("Operacao");
    Operacao op = opHome.create();
    op.transferir("111", "222", 100);
  }
}
```

Figura 2.29 Aplicação cliente

2.7.5 Limitações

A especificação EJB oferece suporte para que sejam desenvolvidos sistemas multicamadas distribuídos onde a complexidade da infra-estrutura fica a cargo do servidor de aplicações. Apesar de permitir a criação de aplicações com um alto grau de transparência em relação a acesso a banco de dados, controle transacional, comunicação entre objetos etc, essa arquitetura de desenvolvimento não oferece suporte para que sejam desenvolvidos sistemas com requisitos de confiança no funcionamento de forma transparente.

Como explicado anteriormente, por meio de objetos *stubs* as aplicações clientes podem acessar os componentes remotamente como se fossem objetos locais. Entretanto, de acordo com a especificação EJB, estes objetos auxiliares não possuem mecanismos para detectar a falha das instâncias referenciadas de forma automática e transparente. De uma maneira geral, é possível que, após o cliente obter uma referência remota ao objeto *home* ou ao *bean*, o servidor EJB torne-se indisponível. Na próxima vez que o cliente invocar um método, ele receberá do *stub* uma exceção indicando problema de comunicação com o servidor. Isso irá obrigar a aplicação cliente a recuperar novamente uma referência a um

objeto *home* em outro servidor e, com ele, obter uma nova referência para o *bean*. Este tratamento extra por parte do cliente é devido ao fato de que os objetos *stubs* só podem referenciar uma instância em um determinado servidor mesmo havendo outra instância em um servidor distinto. Além disso, o objeto *home* não pode criar automaticamente uma referência para o *bean* de outro servidor diferente daquele no qual ele foi criado.

Além de ter de encontrar um servidor livre de falha, a aplicação deve implementar o tratamento para prover as referências recém obtidas com um estado inicial e re-executar a transação que foi interrompida [SUN 99-1]. Quando o componente acessado é um *session bean stateless*, a lógica será mais simples uma vez que este tipo de EJB não possui estado interno bastando que os métodos sejam invocados novamente na nova referência. No caso do *session bean stateful*, haverá a perda do seu estado interno uma vez que este é armazenado em memória [SUN 99-1]. Já para o *entity bean*, o estado poderá ser recuperado caso a falha tenha ocorrido após a efetivação da transação na qual ele participava [SUN 99-1]. Uma vez que o estado desse componente é atualizado para o banco de dados apenas no final de cada transação, as alterações que foram feitas no estado podem ser perdidas caso uma falha ocorra no servidor EJB antes da finalização da transação.

2.8 Conclusão

Para o desenvolvimento de sistemas distribuídos, a plataforma J2EE provê um modelo de desenvolvimento baseado em componentes que oferece ao desenvolvedor um aumento de produtividade e um nível maior de abstração. Uma vez que a complexidade de implementação dos serviços utilizados pela camada intermediária fica a cargo do fabricante do servidor J2EE, o foco do desenvolvimento pode se voltar para a lógica de negócio. Além disso, a plataforma oferece total portabilidade e interoperabilidade por permitir que uma mesma aplicação J2EE possa ser instalada em qualquer servidor que esteja de acordo com a especificação, sem a necessidade de alteração, e por garantir que a comunicação entre componentes de servidores de fabricantes diferentes seja completamente transparente [Thomas 98]. Vale destacar também que o modelo de desenvolvimento baseado em componentes aumenta sensivelmente a reusabilidade de *software*, visto que um mesmo componente pode ser reutilizado com um comportamento diferente, bastando apenas que suas propriedades sejam alteradas.

Como explicado neste capítulo, os serviços e componentes da aplicação são hospedados em um *software* chamado servidor de aplicações. Assim, para garantir a disponibilidade da aplicação, deve haver mais de uma instância do servidor de aplicações executando em máquinas distintas da rede. Entretanto, para que o usuário não perceba a falha de uma das instâncias, a camada cliente deve ser responsável por reiniciar o processamento em uma instância livre de falhas. Uma vez que a plataforma J2EE não especifica que os serviços e componentes que compõem a aplicação devem sincronizar seus estados entre as instâncias do servidor, a complexidade da camada cliente tende a aumentar consideravelmente visto que o desenvolvedor, e não o servidor de aplicações, será o responsável por manter os estados internos atualizados.

No capítulo seguinte, é feita uma análise dos principais servidores J2EE que disponibilizam algum tratamento para as limitações apresentadas. Além de explicarmos o funcionamento dos mecanismos para tolerar falhas, discutiremos a eficácia dos mesmos e o nível de transparência oferecido ao desenvolvedor.

Capítulo 3: Trabalhos Relacionados

3.1 Introdução

Devido às exigências do mercado, diversos servidores de aplicações começaram a implementar alguns mecanismos para oferecer requisitos de confiança no funcionamento, mesmo não havendo nenhum tipo de padronização na especificação J2EE. Muitas organizações começaram a sentir a necessidade de garantir que os seus sistemas continuassem operacionais mesmo após a ocorrência de uma falha de hardware ou, em alguns casos, era desejável torná-lo disponível a uma escala maior de usuários. Para que esses objetivos fossem alcançados, esses servidores adotaram mecanismos para replicar os componentes da aplicação em instâncias distintas do servidor de aplicações. Assim, seria oferecida uma cópia da aplicação em um servidor livre de falhas e seria permitida a distribuição do processamento em diversas máquinas, possibilitando que mais usuários pudessem acessar o sistema. Apesar desses servidores diferirem em vários aspectos como mecanismos de tratamento de falhas disponíveis, forma como estes são implementados e grau de transparência oferecida à camada cliente no momento em que uma falha ocorre, eles se assemelham pelo fato de terem por objetivo tratar apenas falhas silenciosas através de replicação passiva. Nesses servidores, pode-se perceber que o enfoque dado às soluções priorizou o desempenho em detrimento da confiança no funcionamento.

Este capítulo apresenta os principais servidores que oferecem mecanismos de replicação e faz uma análise das soluções adotadas por cada servidor para tratar falhas.

3.2 O Servidor de Aplicações BEA WebLogic Server 6.0

Desenvolvido pela BEA Systems [BEA], o WebLogic Server é um servidor de aplicações baseado na arquitetura J2EE, que se propõe a dar suporte ao desenvolvimento de aplicações corporativas e de *e-business* e, uma vez que ele está escrito 100% em Java, é possível usá-lo em uma grande variedade de plataformas. Além disso, o WebLogic Server

suporta vários tipos de clientes: desde navegadores *Web* (ou qualquer outro cliente HTTP) até dispositivos móveis via WAP e aplicações Java, através dos protocolos RMI ou IIOP (*Internet Inter-ORB Protocol*) [BEA 01-1]. Por ter um grande respaldo comercial, há atualmente várias ferramentas de desenvolvimento que oferecem suporte para a instalação de componentes neste servidor.

Além dos serviços J2EE como EJB, JMS, Java Servlet, JDBC, JNDI, JTS e JTA, este servidor de aplicações possui um servidor *Web* integrado que, além de hospedar documentos estáticos, é responsável pela execução das páginas dinâmicas e da replicação da sessão HTTP entre os vários servidores do grupo de replicação [BEA 01-1]. O WebLogic Server permite que uma instância do servidor (servidor primário) possa replicar as informações das sessões HTTP utilizadas pelos *servlets* com um segundo servidor (servidor secundário) evitando que o cliente necessite reiniciar a sessão caso o servidor o qual estava utilizando venha a falhar [BEA 01-2]. Entretanto, se ocorrer a falha simultânea dos servidores primário e secundário, as sessões estarão perdidas uma vez que o grau de replicação é fixo em um [BEA 01-2] mesmo havendo mais de dois servidores fazendo parte do grupo de replicação.

Além dos servidores do grupo de replicação, é necessário haver uma terceira instância chamada de *proxy server* que será o único servidor "visível" aos clientes e será responsável por direcionar as requisições HTTP destes aos servidores do grupo de replicação que irão responder efetivamente à requisição [BEA 01-2]. Cada vez que o navegador *Web* submete uma requisição, o *proxy server* escolhe uma instância do grupo de replicação (baseado numa política de balanceamento de carga) e envia a requisição para que esta instância possa processá-la. Caso esta instância não responda ao fim de um tempo definido, o *proxy server* reenvia a requisição para uma outra instância. Dessa forma, a disponibilidade do sistema ficará comprometida caso o *proxy server* deixe de responder, uma vez que os clientes não têm conhecimento dos servidores do grupo de replicação.

Para tratar a vulnerabilidade do serviço JNDI, cada servidor do grupo de replicação mantém, além de sua própria base de registros, uma cópia da base das demais instâncias do servidor fazendo com que os clientes tenham a visão de uma única árvore JNDI contendo todos os registros [BEA 01-2]. A atualização dessas informações entre os servidores é feita constantemente através de mensagens de IP *multicast*, permitindo que o grupo de replicação possa tomar conhecimento quando uma instância torna-se inoperante ou quando uma nova

instância passa a fazer parte do grupo de replicação. Uma vez que o IP *multicast* não é um meio de comunicação confiável, há uma possibilidade das bases de registros entre os servidores tornarem-se inconsistentes umas com as outras. Além disso, fica sob responsabilidade da aplicação cliente que acessa este serviço identificar que a instância do servidor tornou-se indisponível e localizar outra instância.

O WebLogic Server também oferece alguns mecanismos para prover tolerância a falhas para os componentes EJB. É possível especificar que o componente (seja ele *session bean* ou *entity bean*) é replicado fazendo com que as referências para esses componentes "conheçam" todas as réplicas existentes para o componente [BEA 01-2]. Dessa forma, como o cliente acessa o componente por meio dessa referência replicada, esta pode direcionar a requisição para uma réplica do componente de forma transparente caso perceba que a instância que vinha sendo usada tornou-se indisponível. Entretanto, caso um dos servidores venha a falhar após o cliente ter obtido a referência replicada, esta não terá conhecimento da falha do servidor e poderá tentar acessar o servidor indisponível ou deixar de utilizar um servidor recém adicionado ao grupo de replicação. Além disso, a referência replicada somente irá utilizar uma segunda réplica do componente na próxima invocação a esse componente. Isso significa que, caso o componente falhe durante a invocação, a aplicação não irá perceber que houve um problema.

Quando o cliente invoca a criação de um *session bean* sem estado através da interface *home*, a *stub* retornada traz consigo referências para todas as réplicas desse componente [BEA 01-2]. Com isso, a *stub* teria capacidade de interceptar as requisições do cliente verificando se a instância que irá processar a requisição está operacional e, caso contrário, redirecionar a chamada para uma réplica. Como nesse caso não há estado a ser preservado, qualquer réplica poderia responder à requisição de imediato. Entretanto, uma vez que o WebLogic não tem controle sobre o que foi processado pelo componente até o momento da falha, a *stub*, por padrão, não re-executa o método em uma outra réplica porque haveria o risco de que determinados dados já alterados antes da falha fossem alterados novamente. Assim, o WebLogic impõe que o desenvolvedor "marque" no descritor de instalação do *session bean* a ser replicado se ele é idempotente ou não [BEA 01-3]. Nessa concepção, se o componente é indicado como idempotente significa que ele não irá realizar nenhuma alteração persistente o que permitirá que seus métodos possam ser chamados várias vezes sem risco de ocasionar inconsistências. Caso o componente seja definido como idempotente, a sua *stub* poderá

refazer automaticamente a requisição para uma réplica em um outro servidor caso perceba que a instância requisitada tenha falhado. Caso o *session bean* não seja idempotente, ficará sob responsabilidade da aplicação cliente decidir se invoca novamente o método ou não [BEA 01-3].

Para os *sessions beans stateful*, este servidor oferece um serviço de replicação de estado para garantir a continuidade do processamento após a ocorrência de uma falha [BEA 01-2]. Quando o cliente solicita a criação de uma nova instância, também será criada uma réplica secundária em uma outra instância do servidor, com o objetivo de armazenar as alterações feitas no estado da instância primária [BEA 01-3]. Cada vez que uma transação é finalizada corretamente, o estado do componente na réplica secundária é atualizado. Quando o servidor que hospeda a instância primária falhar, a próxima requisição do cliente ao componente será repassada pela *stub* à réplica secundária que passará a ser a instância primária desse componente [BEA 01-2]. A partir daí, uma nova réplica será escolhida para ser a instância secundária e irá receber as atualizações do estado da mesma forma que antes [BEA 01-3]. Entretanto, pode acontecer que, em algumas situações, as alterações feitas no estado não sejam atualizadas na réplica secundária. Caso a transação envolvendo um EJB *stateful* seja finalizada corretamente, e o servidor primário apresente uma falha antes de replicar o estado para o servidor secundário, o cliente não irá perceber a falha e irá utilizar a réplica secundária com um estado antigo o que levará o sistema a um comportamento incorreto [BEA 01-3]. Numa outra situação, quando a primeira transação é finalizada e o estado não puder ser replicado por falha do servidor primário, a próxima requisição do cliente tentará utilizar a réplica secundária que não possui estado algum. Assim, o cliente deve estar preparado para receber uma exceção do componente e criar novamente uma instância do *session bean* através da interface *home* [BEA 01-3]. Além disso, a aplicação cliente deve estar preparada para o caso da falha simultânea dos servidores primário e secundário uma vez que o grau de replicação é fixo em um.

No WebLogic Server, os *entity beans* podem ser usados de duas formas: para ler e gravar dados no banco (modo *read-write*) e apenas para obter os dados (modo *read-only*) [BEA 01-3]. Se um componente desse tipo for especificado como *read-write* no seu descritor de instalação, significa que ele irá tanto obter quanto alterar os dados na tabela que ele representa. Já os componentes *read-only*, permitem que os dados sejam lidos do banco de dados, mas não permitem que seu estado seja alterado. Nesse servidor não há replicação de

estado para os componentes *read-write* [BEA 01-2]. Isso significa que, caso uma aplicação cliente esteja acessando um *entity bean* e o servidor que o hospeda torne-se indisponível, uma instância em um outro servidor deve ser localizada e a transação deve ser refeita pelo cliente. Esse mesmo comportamento é válido para os componentes *read-only*, mesmo havendo replicação de estado para esse tipo de componente [BEA 01-3]. Esta replicação existe apenas para que possa ser feita uma distribuição das requisições ao componente entre os vários servidores do grupo de replicação.

3.3 O Servidor de Aplicações Borland AppServer 4.5

Para tentar diminuir as dificuldades do desenvolvimento, da implantação, e da gerência de aplicações distribuídas para a plataforma *Web*, a Borland Software Corporation desenvolveu o Borland AppServer, um servidor de aplicações que, além de implementar os serviços da plataforma J2EE, oferece ferramentas visuais para auxiliar tanto o desenvolvedor quanto o administrador do sistema [Borland 01-3]. Por usar o protocolo IIOP para comunicação entre objetos remotos, é possível que aplicações clientes escritas em outras linguagens além do Java acessem os componentes instalados nesse servidor.

Apesar de oferecer um servidor *Web* integrado, este servidor de aplicações não oferece nenhum tipo de mecanismo para garantir a utilização contínua e correta do sistema para os usuários que o acessam através de um navegador *Web* [Borland 01-1]. Como muitas aplicações *Web* utilizam a sessão do *servlet*, o fato de não haver suporte para que esta possa ser preservada caso o servidor apresente uma falha pode causar a indisponibilidade do sistema.

Já para preservar as informações do serviço JNDI, o Borland AppServer permite que seja usado um meio persistente como um banco de dados. Essa solução consiste em executar até duas instâncias do serviço de nomes JNDI em máquinas diferentes especificando que todas estarão compartilhando o mesmo banco de dados [Borland 01-2]. Dessa forma, caso uma instância do serviço de nomes torne-se indisponível, a outra continuará atendendo aos clientes uma vez que as instâncias sempre obtêm as informações do banco de dados. Entretanto, a disponibilidade do sistema como um todo irá depender do servidor de banco de

dados. Caso ocorra uma falha neste serviço, as instâncias do JNDI não poderão recuperar os registros previamente armazenados. Além disso, como o grau de replicação do serviço JNDI é fixo em um [Deshpande 00], essa solução suporta que apenas uma instância falhe.

O Borland AppServer foi criado com um conjunto de módulos independentes que implementam os serviços requeridos pela plataforma J2EE [Borland 01-4]. Isso permite que cada instância que faça parte do grupo de replicação possa ser customizada para responder apenas a um ou por alguns serviços. Os principais módulos são [Borland 01-4]:

- Contêiner *Web* – responsável pela execução dos *servlets*;
- Serviços CORBA – serviços utilizados para realizar a comunicação entre os objetos remotos e entre estes e as aplicações clientes;
- Serviço JNDI – implementação da especificação JNDI da SUN;
- Contêiner EJB – responsável por executar e gerenciar os componentes EJB;
- Gerenciador de transações – implementação das especificações JTS e JTA da SUN; e
- Serviço de armazenamento de sessão – banco de dados interno utilizado para armazenar estados dos componentes.

Dos serviços acima, os únicos que não permitem replicação são o contêiner *Web* e o serviço de armazenamento de sessão. Assim, além de ser possível executar mais de uma instância do serviço JNDI, pode-se ter várias instâncias de contêineres EJB, cada uma possuindo um conjunto diferente de componentes [Borland 01-2].

Como o Borland AppServer permite que várias instâncias do servidor em um grupo de replicação hospedem um contêiner EJB, é possível replicar todos os tipos de componentes EJB. Uma vez que cada tipo de componente possui características e comportamentos próprios, o procedimento empregado por este servidor de aplicações no momento da falha de uma instância vai variar conforme o tipo de componente [Borland 01-2]. Se um determinado *session bean stateless* for instalado em mais de um contêiner EJB de um grupo de replicação, o protocolo de comunicação será responsável por direcionar as requisições das aplicações clientes para uma instância livre de falhas de forma transparente para o cliente [Deshpande 00]. Uma vez que esse tipo de componente não possui estado, uma réplica é escolhida aleatoriamente para responder a uma requisição. Entretanto, se a falha ocorrer durante o

processamento do método do componente, a aplicação cliente que invocou o método irá perceber a falha e ela deverá ser responsável por refazer a requisição de forma explícita [Borland 01-2].

Da mesma forma que o *stateless*, os componentes EJB *stateful* também são replicados de forma automática no instante em que são instalados nas instâncias do contêiner EJB do grupo de replicação [Borland 01-1]. Além disso, o comportamento do protocolo de comunicação no momento em que ocorre uma falha em uma instância é semelhante ao que ocorre no *stateless*: a cada requisição efetuada, o protocolo de comunicação confere se a instância do componente que estava sendo usada continua operacional e, caso não esteja, uma réplica desse componente é localizada para responder pela requisição. Se a falha ocorrer durante o processamento do método, a aplicação que o invocou deverá tratar a exceção e invocar novamente o método [Deshpande 00]. Neste servidor de aplicações, em vez do estado do EJB ser mantido na instância do componente, ele é preservado no serviço de armazenamento de sessão. Assim, no momento que uma réplica é escolhida pra substituir uma instância falha, ela é inicializada com o último estado salvo do componente que falhou [Borland 01-2]. Entretanto, é possível que a réplica seja inicializada com um estado que não reflete a última modificação feita por um cliente. Como o estado das instâncias é passado para o serviço de armazenamento de sessão periodicamente [Borland 01-4], pode acontecer que uma falha ocorra depois que o estado foi modificado e antes dele ser salvo. Além disso, como não é possível replicar o serviço de armazenamento de sessão, todos os estados dos componentes serão perdidos caso ocorra uma falha na instância do servidor que hospeda esse serviço.

O Borland AppServer também permite que *entity beans* do mesmo tipo instalados em instâncias do contêiner EJB do mesmo grupo de replicação atuem como réplicas uns dos outros [Borland 01-1]. Como nos outros casos, o protocolo de comunicação será responsável por enviar a requisição para um servidor livre de falhas. Uma vez que o contêiner EJB faz com que a instância do *entity bean* leia o seu estado da tabela o qual ele representa antes de cada transação ser iniciada e requer que esse componente salve as alterações antes da transação ser finalizada [Borland 01-4], há a garantia de que sempre haverá um estado consistente do componente a ser usado. Porém, caso a falha ocorra durante o processamento da transação, não é possível continuar a transação em uma réplica de um outro contêiner EJB.

Uma vez que a transação é desfeita quando uma falha ocorre, fica sob responsabilidade da aplicação cliente reiniciar toda a transação.

3.4 O Servidor de Aplicações Sybase EAServer 3.6

Criado pela Sybase, o EAServer é a integração de dois produtos já desenvolvidos por esta empresa: o PowerDynamo e o Jaguar CTS [Sybase 00]. O primeiro é um servidor de conteúdo *Web* dinâmico, enquanto que o segundo é um serviço para gerenciamento de transações entre componentes. Com essa integração, a Sybase oferece uma solução completa para o desenvolvimento de aplicações *Web* como portais e sistemas multicamadas baseados na plataforma J2EE que precisem interagir com sistemas legados [Sybase]. Uma vez que este servidor implementa tanto a especificação EJB quanto a CORBA, é possível usá-lo para integrar sistemas escritos em diferentes linguagens [Sybase 00].

O EAServer implementa o conceito de grupo de replicação em que várias instâncias do servidor executando em diferentes máquinas compartilham da mesma configuração, fazendo com que as aplicações clientes tenham a visão de apenas uma unidade [Wolf 99]. Para isso, uma das instâncias é escolhida como primária e será a partir dela que as outras instâncias desse grupo irão obter os parâmetros de configuração no momento em que forem inicializadas [Wolf 99]. Como essa sincronização não ocorre dinamicamente, haverá uma inconsistência no grupo de replicação se algum tipo de alteração for feita no servidor primário após a inicialização das outras instâncias. Além disso, a escolha do servidor primário sempre é feita manualmente pelo administrador do sistema. Isso implica que, na falha do servidor primário, um novo não será escolhido automaticamente entre as outras instâncias [Sybase 99].

Apesar das instâncias serem inicializadas com a configuração do servidor primário, cada uma pode ser customizada para oferecer serviços diferentes das outras [Wolf 99]. Entretanto, mesmo que um determinado serviço possa estar em execução em mais de uma instância do servidor, não implica que haverá replicação de suas informações entre as instâncias, como acontece, por exemplo, com o PowerDynamo [Sybase 99]. Não há suporte no EAServer para que as sessões do *servlets* sejam replicadas entre as instâncias fazendo com que o usuário que acesse o sistema pelo navegador *Web* seja obrigado a utilizar um outro

servidor em caso de falha. Assim, a possibilidade de existir um determinado serviço em execução em mais de uma instância do EAServer significa, em alguns casos, apenas uma maior disponibilidade do mesmo, deixando o tratamento para a continuidade do serviço a cargo da aplicação cliente.

Em um grupo de replicação é possível haver mais de uma instância executando o serviço JNDI [Wolf 99]. Para isso basta habilitar esse serviço nas instâncias desejadas e, a partir daí, elas irão sincronizar entre si as suas bases de registros. A tarefa de especificar os servidores que irão hospedar o serviço JNDI é de responsabilidade do administrador do sistema, assim como incluir servidores adicionais caso os servidores previamente configurados deixem de responder [Wolf 99]. Essa abordagem não oferece uma solução plenamente transparente para as aplicações clientes uma vez que estas deverão especificar de qual instância do servidor desejam utilizar o serviço JNDI.

O único tipo de componente EJB sobre o qual o EAServer implementa mecanismos de tolerância a falhas é o *session bean* [Sybase 99]. Um mesmo componente pode ser instalado em todas as instâncias do grupo de replicação e o protocolo de comunicação irá escolher uma réplica dentre as operacionais para responder à requisição.Caso o protocolo perceba que a invocação não foi completada corretamente, uma outra réplica será escolhida para executar a invocação de forma transparente para a aplicação cliente. Entretanto, uma vez que é possível que o componente faça modificações permanentes em bancos de dados, a re-execução da invocação poderá causar inconsistências no sistema. Assim, esse comportamento só será apresentado pelo protocolo de comunicação caso o administrador o habilite [Wolf 99]. É importante destacar que o EAServer não realiza a replicação do estado interno dos componentes, o que impede de usar o mecanismo descrito acima para *session beans* com estado e para *entity beans*.

3.5 O Servidor de Aplicações Persistence PowerTier for J2EE 6.0

Considerado como principal produto da Persistence Software Inc, o PowerTier for J2EE (chamado anteriormente de PowerTier for EJB) é um servidor de aplicações que, além de seguir a especificação J2EE, caracteriza-se por oferecer ferramentas de desenvolvimento

como um gerador de esquemas relacionais e de código fonte, mecanismos de *cache* de dados e pelo suporte transacional utilizando padrões do CORBA [Persistence]. Este servidor se propõe a facilitar o desenvolvimento dos componentes EJB por permitir que estes sejam gerados a partir de um esquema do banco de dados, com também é possível definir os componentes com seus atributos para que ele gere tanto o código fonte como o esquema relacional correspondente [Persistence 99-1]. Além disso, o PowerTier é indicado pelo fabricante como uma solução para aplicações de comércio eletrônico que dependem de um grande número de transações e de acesso intenso a banco de dados uma vez que seu mecanismo de *cache* transacional compartilhado permite armazenar os *entity beans* já utilizados em memória para que as próximas requisições não necessitem acessar o banco de dados [Hess 00]. Além de permitir que os relacionamentos desses componentes também sejam armazenados para uma maior otimização da navegação pela informação, os objetos em *cache* podem ser acessados por usuários distintos [Persistence 99-1]. Uma vez que essa solução de *cache* implementa mecanismos de controle transacional, há a garantia de haver o isolamento entre aplicações clientes distintas que estejam acessando o mesmo componente para que a consistência da informação seja preservada [Persistence 99-1].

A solução do PowerTier para replicação dos componentes EJB é baseada no seu mecanismo de *cache* transacional compartilhado aliado ao seu serviço de sincronização denominado PowerSync [Persistence 99-1]. Essa sincronização de *cache* utiliza um serviço confiável de envio de mensagem que garante a comunicação correta entre os servidores do grupo de replicação. O funcionamento da sincronização consiste em replicar entre as instâncias do grupo de replicação as alterações no *cache* que foram efetivadas [Persistence 99-2]. Quando uma transação é finalizada corretamente, a instância do componente é colocada no canal de comunicação para que ela possa ser entregue para as outras instâncias do servidor de forma confiável [Persistence 99-1].

Neste servidor de aplicações, é possível replicar apenas *entity beans*. Para isso, o desenvolvedor deve alterar a implementação dos componentes que deseja replicar para incluir um atributo de controle de ordenação chamado *Optimistic Control Attribute* (OCA) [Persistence 99-2]. Esse atributo é utilizado pelo serviço de sincronização de *cache* para determinar se uma determinada mensagem de sincronização que está sendo recebida contém um dado mais atualizado do que o que está em *cache* [Persistence 99-2]. Uma vez que nesse sistema de envio de mensagens não é possível garantir a ordem com que as mensagens de

sincronização irão chegar nas instâncias do grupo de replicação, a falta deste atributo no componente poderia implicar em inconsistência de informação. O PowerTier utiliza o OCA em um algoritmo para decidir o que fazer quando o serviço de sincronização recebe uma mensagem fora de ordem [Persistence 99-2]. A depender do resultado do algoritmo, a instância do servidor que está recebendo uma mensagem de sincronização com ordem incorreta poderá:

- atualizar seu *cache* com o que foi recebido;
- ignorar a mensagem; ou
- invalidar o objeto que está em seu *cache* para que a próxima requisição faça com que ele seja lido do banco de dados.

A atualização do *cache* entre os servidores do grupo de replicação ocorre sempre que uma instância do PowerTier finaliza uma transação da aplicação cliente, impossibilitando que alterações feitas até o momento da falha sejam perdidas caso a transação ainda estivesse sendo processada. No momento em que a transação é finalizada, a instância do servidor faz a atualização do seu próprio *cache* a partir das operações ocorridas na transação. Em seguida essas operações são passadas para o serviço confiável de envio de mensagem que irá montar uma mensagem contendo estas operações e a enviará para todos os servidores do grupo de replicação de forma assíncrona. Cada servidor do grupo de replicação, ao receber a mensagem, aplica o algoritmo para decidir se aceita ou não a mensagem de sincronização recebida [Persistence 99-2]. O algoritmo de sincronização de *cache* permite que cada instância do grupo de replicação possa detectar problemas de ordenação de mensagens e resolver esses conflitos de forma apropriada [Persistence 99-2]. Esse algoritmo compara o OCA contido na mensagem de sincronização com o valor que está em seu *cache* para determinar se este deve ser atualizado. O administrador do sistema deve especificar o valor do *Optimism Level*, um parâmetro usado pelo algoritmo de sincronização durante o cálculo da diferença entre a versão contida na mensagem e a que está em *cache*. O *Optimism Level* indica o delta máximo entre as versões da instância antiga e da nova que o administrador está disposto a permitir para que a sincronização seja efetuada [Persistence 99-2].

O algoritmo de sincronização reduz de forma significativa o risco de existir dados obsoletos no *cache* devido a problemas de ordenação de mensagens. Entretanto, uma vez que este algoritmo depende do valor do OCA, ele não consegue lidar com algumas seqüências de eventos envolvendo exclusões de registros. Uma situação típica, por exemplo, é quando uma

instância é excluída, outra transação recria o objeto e essas mensagens são recebidas fora de ordem. O OCA de um objeto que é recriado irá conter o valor inicial (zero) fazendo com que o valor do OCA antigo seja considerado mais novo. Além disso, quando o valor do OCA alcançar o limite máximo do tipo definido pelo desenvolvedor (2^{64} caso o OCA seja definido como inteiro) ele será reinicializado com valor zero fazendo com que as mensagens de sincronização sejam ignoradas [Persistence 99-2].

Apesar de garantir a replicação dos estados dos *entity beans* entre os servidores do grupo de replicação, o PowerTier não oferece um mecanismo transparente para a aplicação cliente continuar o processamento da transação caso ocorra uma falha na instância do servidor utilizada. Usando este servidor, a aplicação que acessa os componentes deve implementar a lógica para detectar que um servidor não está mais operacional e para localizar uma outra instância desse grupo de replicação. Além disso, fica sob responsabilidade da aplicação identificar em que ponto sua transação foi interrompida e re-executar esta transação de forma correta.

O PowerTier permite que várias instâncias do grupo de replicação possuam cada uma um servidor *Web* e um contêiner *servlet*. Com isso, há um balanceamento de carga entre os servidores do grupo de replicação apesar de não ser possível garantir a continuidade do serviço por não haver nenhum mecanismo de replicação de sessão.

O PowerTier implementa a especificação JNDI por meio do serviço de nomes do CORBA chamado COSNaming [Persistence 99-1]. Nesta implementação, é usado um meio persistente de armazenamento para preservar as informações sobre os componentes instalados. Assim, num grupo de replicação, é possível haver várias instâncias do serviço JNDI onde cada uma irá obter os dados do meio persistente durante a inicialização [Persistence 99-2]. Apesar desta solução tolerar que alguns servidores venham a falhar, o serviço ficará comprometido caso o meio persistente de armazenamento torne-se indisponível. Como cada servidor pode responder pelo serviço JNDI, o PowerTier oferece também um serviço de balanceamento de carga que irá tornar a falha de um determinado servidor transparente para a aplicação cliente [Persistence 99-2]. Para isso, as aplicações devem especificar que irão acessar o serviço de nomes através do serviço de balanceamento de carga no momento que forem obter o contexto do serviço JNDI. Cada vez que for feita uma consulta, o serviço de balanceamento de carga intercepta a requisição de consulta e, por meio

de uma política de balanceamento, escolhe um servidor JNDI do grupo de replicação [Persistence 99-2]. Escolhido o servidor, o serviço de balanceamento irá acessar o serviço JNDI desse servidor para obter a referência ao componente desejado pelo cliente [Persistence 99-2]. Caso a instância do servidor escolhido não possua uma referência para o componente, o serviço de balanceamento escolhe um outro servidor. Dessa forma as aplicações clientes terão o serviço de balanceamento como ponto de contato com o grupo de replicação.

3.6 O Servidor de Aplicações Allaire Jrun 3.1

Concebido a princípio para ser puramente um contêiner *servlet*, atualmente o JRun consiste em um conjunto de aplicações para desenvolvimento de aplicações servidoras para a plataforma Java [Allaire]. Este conjunto é formado pelos seguintes componentes:

- JRun Studio – ferramenta visual para desenvolvimento de aplicações para a plataforma *Web* como páginas JSP e *servlets*;
- Kawa – editor de código fonte voltado para a linguagem Java com recursos para criação de componentes EJB; e
- JRun Server – servidor de aplicações que segue o padrão J2EE e serve de ambiente para a execução das aplicações servidoras.

Por estar implementado de forma modular, é possível configurar o JRun Server para atender a necessidades específicas da aplicação. Ou seja, pode-se tanto personalizá-lo para ser um simples contêiner *servlet* quanto um ambiente completo para hospedar componentes EJB [Allaire 01-2]. Além disso, o administrador do sistema tem a opção de utilizar o servidor *Web* integrado do JRun Server ou configurá-lo para interagir com algum outro servidor de páginas já existente.

Em relação a provimento de confiança no funcionamento, o JRun Server permite que apenas a camada *Web* possa ser replicada. Sua solução para criação de grupos de replicação é feita através do Allaire ClusterCATS, um serviço de balanceamento de carga e de replicação de sessão que acompanha este servidor de aplicações [Allaire 01-2]. Este serviço permite também que o administrador possa gerenciar os servidores do grupo de replicação e identifique servidores superutilizados [Allaire 00].

Para garantir a continuidade de uma aplicação, o ClusterCATS utiliza um mecanismo de replicação de sessão dos *servlets* entre os servidores do grupo de replicação [Allaire 00]. Cada vez que uma aplicação *Web* é inicializada, além da sessão ser criada no servidor que respondeu a requisição, as informações que ela contém são copiadas para os outros servidores. Assim, caso o servidor acessado torne-se indisponível, a aplicação cliente poderá continuar sua execução uma vez que as informações guardadas na sessão poderão ser recuperadas.

Quando é utilizado um servidor *Web* independente, pode-se instalar um módulo do ClusterCATS que irá redirecionar as requisições ao conteúdo dinâmico para uma ou várias instâncias de um grupo de replicação do JRun Server [Allaire 00]. Com isso, além de se obter um balanceamento de carga, não haverá risco do navegador *Web* do cliente tentar acessar uma instância não operacional. Como o ClusterCATS possui um serviço que testa periodicamente cada instância do grupo de replicação [Allaire 01-1], ele conhece previamente quais servidores estão operacionais e quais não estão. Quando uma instância deixa de responder ao seu teste, ela fica isolada do grupo de replicação até que o ClusterCATS detecte que o servidor voltou a estar operacional.

3.7 Conclusão

Como apresentado, as soluções existentes para prover tolerância a falhas na plataforma J2EE são baseadas em um modelo de replicação passiva. Nesses servidores, é realizada a salvaguarda do estado do serviço periodicamente ou a cada requisição processada. Percebe-se que podem ocorrer situações em que a última alteração no estado pode ser perdida sem que a aplicação cliente perceba, levando a uma inconsistência no processamento do sistema. Além disso, o grupo de replicação não é transparente para o cliente. Em todos os casos, quando se deseja utilizar um dos serviços da plataforma J2EE, deve-se especificar qual instância do grupo de replicação deverá responder pelo serviço. É também de responsabilidade da aplicação cliente tratar a falha quando esta acontece durante o processamento da requisição, uma vez que, em quase todos os casos, o servidor só consegue mascarar a falha se ela ocorrer entre requisições. No período em que essa pesquisa foi realizada, nenhum dos servidores

provia qualquer mecanismo para tolerância a falhas para o serviço JMS. A Tabela 3.1 resume as deficiências encontradas nos servidores J2EE apresentados nesse capítulo.

Servidor	**Serviço**		
	Servlet	**JNDI**	**EJB**
BEA WebLogic Server 6.0	- Grau de replicação igual a um.	- Baseado em IP *multicast*; - Grupo de replicação não é transparente.	- Grau de replicação igual a um; - Pode ocorrer inconsistência de estado; - Não oferece replicação de estado para *entity beans*
Borland AppServer 4.5	- Não oferece replicação de estado para *servlets*.	- O meio de armazenamento persistente externo ao serviço representa um único ponto de falha.	- Mascara a falha apenas se esta ocorrer entre requisições; - O meio de armazenamento persistente do *session bean stateful* representa um único ponto de falha; - Pode ocorrer inconsistência de estado.
Sybase EAServer 3.6	- Não oferece replicação de estado para *servlets*.	- Grupo de replicação não é transparente.	- Tolerância a falhas apenas para *session bean stateless*
Persistence PowerTier 6.0	- Não oferece replicação de estado para *servlets*.	- O meio de armazenamento persistente externo ao serviço representa um único ponto de falha.	- Tolerância a falhas apenas para *entity bean*; - Necessita alteração do código dos componentes; - Pode ocorrer inconsistência de estado; - Grupo de replicação não é transparente.
Allaire JRun 3.1	- Grupo de replicação não é transparente.	- Não oferece replicação de estado para JNDI.	- Não oferece replicação de estado para EJB.

Tabela 3.1 Resumo das características dos servidores estudados

De uma maneira geral, para desenvolver aplicações J2EE com requisitos de confiança no funcionamento utilizando os servidores de aplicações existentes, é necessário que o projeto do sistema inclua alguns mecanismos para complementar o tratamento de falhas oferecido pelo servidor, fazendo com que a complexidade do sistema aumente consideravelmente. Além disso, por se basearem em replicação passiva, nenhum desses servidores oferece suporte para a replicação do serviço básico de RMI.

Com o objetivo de eliminar as vulnerabilidades do RMI, foram desenvolvidos diversos trabalhos que se baseiam na implementação de um mecanismo de invocação remota de método em grupo também conhecido como GMI (*Group Method Invocation*). Entretanto, todos os trabalhos pesquisados requerem algum tipo de alteração no objeto remoto ou no objeto cliente, ou ainda perdem características de portabilidade do Java. No sistema Aroma [Melliar-Smith 00], são oferecidas duas abordagens: a primeira exige que o lado cliente e o servidor informem explicitamente ao serviço RMI as classes que irão realizar a comunicação entre o objeto cliente e o objeto remoto. Na segunda, as bibliotecas da máquina virtual que irão acessar os serviços nativos de rede são modificadas. Essas duas abordagens têm como objetivo interceptar as invocações remotas realizadas pelos objetos clientes para replicar as requisições para todas as réplicas do grupo. Enquanto que a primeira abordagem faz com que se perca transparência (o desenvolvedor deve ativar explicitamente a replicação), a segunda não é portável, uma vez que, para cada plataforma, as bibliotecas deverão ser alteradas. Em outras soluções como [Kielmann 00] e [Hagiont-Boyer 01], o objeto remoto deve utilizar uma API específica fornecida pela solução no lugar da API RMI padrão. Já o Filterfresh [Baratloo et al. 98] exige a utilização de um serviço de nomes proprietário no lugar de um serviço JNDI.

O próximo capítulo apresenta a nossa abordagem para um servidor J2EE tolerante a falhas e transparente, que inclui também um mecanismo de GMI transparente. A solução está descrita sob a forma de um projeto onde são explicados os requisitos necessários a serem implementados.

Capítulo 4: Projeto de uma Plataforma J2EE Tolerante a Falhas e Transparente

4.1 Introdução

Como visto no capítulo anterior, as soluções apresentadas pelos servidores de aplicações J2EE existentes não são completamente transparentes para os desenvolvedores, além de permitirem que ocorram situações em que o processamento do sistema torne-se inconsistente. O projeto de servidor de aplicações J2EE proposto nesse trabalho tem como objetivo principal oferecer transparência para o desenvolvedor e não permitir que ocorra nenhuma "janela de vulnerabilidade" devido a uma falha de uma das instâncias do servidor.

Da mesma forma que as implementações existentes, a solução apresentada nesse capítulo também tem como base a replicação do servidor de aplicações. Isso significa que deverá haver mais de uma instância do servidor J2EE em execução, e os componentes da aplicação (*servlets*, componentes EJB, etc) estarão instalados em todas as instâncias do servidor. Diferentemente da replicação passiva utilizada pelos outros servidores, onde apenas uma instância realiza o processamento e a sincronização com as outras réplicas, o nosso projeto irá utilizar replicação ativa como mecanismo básico para a introdução de tolerância a falhas. Nesse modelo de replicação, todas as réplicas realizam o processamento fazendo com que os seus estados mantenham-se naturalmente sincronizados. Para tal, o processamento das réplicas precisa ser determinístico e as requisições solicitadas pelos clientes precisam ser processadas na mesma ordem por todas as réplicas operacionais [Schneider 90].

O nosso projeto tem como base incluir em um servidor J2EE de código aberto um serviço de replicação ativa com o objetivo de tratar falhas silenciosas. Como cada requisição a um serviço realizada por uma aplicação cliente será processada por todas as instâncias do grupo de replicação, a falha de uma instância não irá implicar na indisponibilidade da aplicação. A sincronização entre as réplicas será mantida através de um mecanismo de comunicação em grupo que provê as seguintes propriedades:

- Ordem: as requisições são processadas por todas as réplicas na mesma ordem; e
- Atomicidade: uma requisição ou é processada por todas as réplicas corretas ou por nenhuma delas.

Através desse mecanismo, uma requisição feita por uma aplicação cliente poderá ser enviada de forma transparente e confiável ao grupo e não mais a uma instância. Por exemplo, uma requisição a um *servlet* será processada por todas as instâncias do *servlet* nos servidores em execução, da mesma forma que uma requisição de registro no serviço JNDI ou uma invocação a um método de um componente EJB.

Esse capítulo apresenta o projeto de um servidor J2EE tolerante a falhas e transparente explicando a abordagem adotada, assim como os requisitos necessários a serem implementados.

4.2 Visão Geral

Como explicado no capítulo 2, os serviços da plataforma J2EE são utilizados pelas aplicações clientes (camada da interface) por meio de bibliotecas clientes que se encarregam de realizar a comunicação entre a aplicação e a instância do servidor. Dessa forma, para que o serviço de comunicação em grupo possa ser utilizada de forma transparente pelas aplicações clientes, as bibliotecas clientes deverão ser alteradas para passar a interagir com este serviço. Assim, a requisição feita pelo cliente será enviada para o grupo de replicação e não mais para uma instância permitindo que a aplicação cliente não precise conhecer quais servidores que fazem parte do grupo de replicação.

Será necessário desenvolver um serviço adicional que deverá estar em execução em cada instância do servidor J2EE. Este serviço, denominado *Group Proxy*, será responsável por interceptar as requisições enviadas ao grupo pelas aplicações clientes para repassá-las ao servidor J2EE. Após a requisição ser processada pelo servidor, o *Group Proxy* devolverá a reposta para a aplicação cliente que fez a requisição. Uma vez que serão geradas várias respostas para uma requisição, a biblioteca cliente, que permanece esperando a resposta da

requisição, retorna apenas a primeira resposta recebida para o cliente que fez a requisição. A Figura 4.1 ilustra, de maneira geral, a arquitetura da solução.

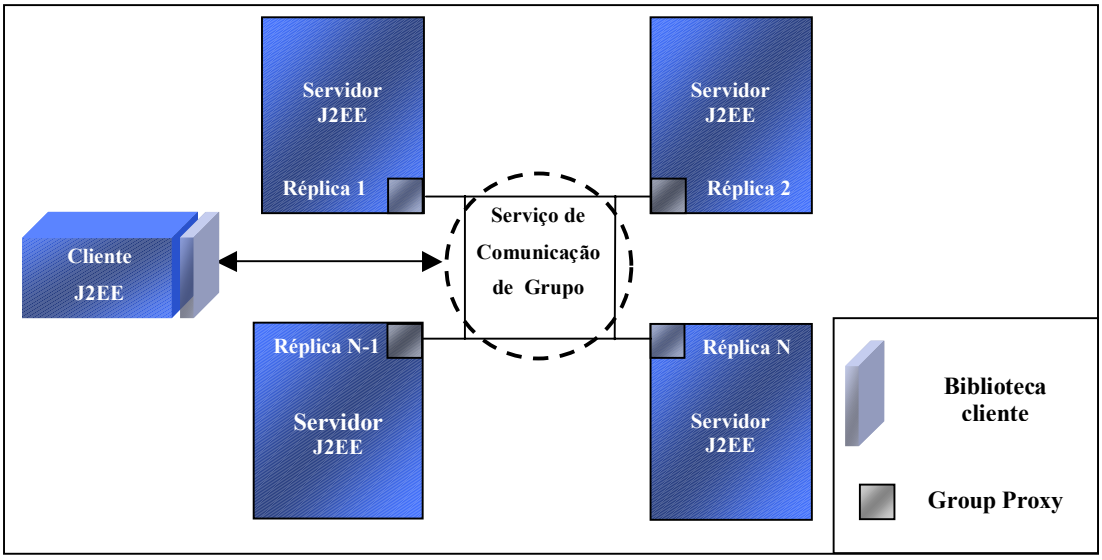

Figura 4.1 Arquitetura do projeto

Dessa forma, não será necessário alterar nem a aplicação cliente nem os componentes da aplicação hospedados no servidor para que o serviço de comunicação em grupo seja utilizada no grupo de replicação.

Cada serviço da plataforma J2EE possui um conjunto de objetos específicos para acessá-los. A seguir estão descritas as alterações necessárias nas bibliotecas clientes para cada serviço.

4.3 Java Servlet Tolerante a Falhas

A adoção da replicação ativa resolve as vulnerabilidades apresentadas pelos *servlets* uma vez que, como todas as réplicas dos *servlets* em execução processam as requisições, as informações mantidas nas sessões permanecem sincronizadas em todas as instâncias do contêiner do *servlet*.

Como foi explicado anteriormente, os *servlets* são acessados através de navegadores *Web* e não seria possível alterá-los para que enviem a requisição para o grupo de replicação

(através do serviço de comunicação em grupo) em vez de enviar a requisição para um servidor específico. Entretanto, como muitas organizações já possuem um servidor *Web* responsável pelo conteúdo estático, os servidores J2EE oferecem um *plug in* para que o servidor de páginas passe a redirecionar as requisições ao conteúdo dinâmico para o contêiner do *servlet*. Assim, a utilização do serviço de comunicação em grupo deverá ser feita através do *plug in* permitindo que o navegador *Web* continue a enviar as requisições para o servidor *Web*. O *plug in* deverá ser alterado para que, ao receber a requisição originada do navegador *Web*, envie esta requisição para o grupo de replicação através do serviço de comunicação em grupo. O *plug in* será responsável também por filtrar as respostas geradas pelas réplicas para que apenas a primeira retorne ao navegador *Web*.

Como se pode perceber, a disponibilidade da aplicação estará dependente do servidor de páginas. Caso este venha a falhar, o usuário não poderá acessar a aplicação mesmo que exista alguma instância operacional do servidor J2EE. Assim, é recomendável replicar também o servidor de páginas em máquinas diferentes. Entretanto, para que estes servidores possam ser utilizados pelo navegador *Web*, é necessário instalar um tipo de *hardware* denominado balanceador de carga (*load balancer*). Este dispositivo irá se encarregar de distribuir as requisições feitas pelo navegador *Web* entre os servidores *Web*. O balanceador de carga possui um endereço IP virtual que será utilizado pelos navegadores *Web* para enviar a requisição. Ao receber uma requisição destinada ao IP virtual, o balanceador de carga irá enviar para um dos servidores *Web* e, se for detectado que o servidor *Web* não está operacional, o próprio balanceador de carga reenvia a requisição para outro servidor de páginas. A Figura 4.2 ilustra a introdução do balanceador de carga na solução proposta.

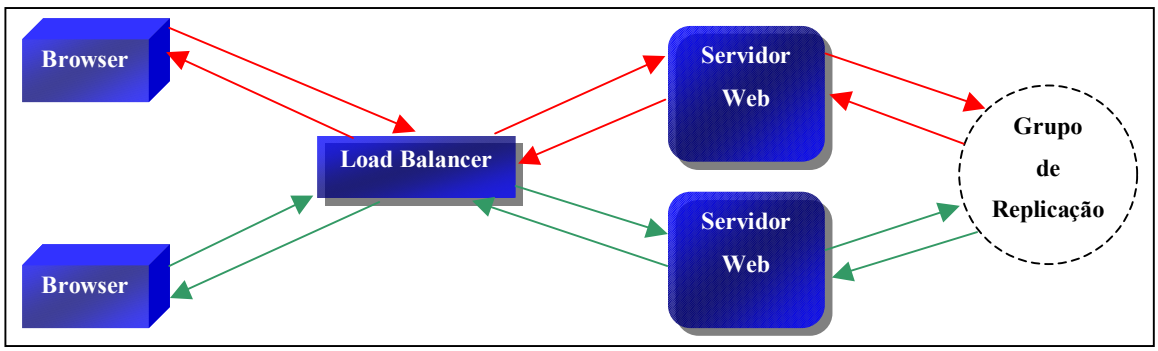

Figura 4.2 Funcionamento do balanceador de carga

Com o balanceador de carga, além de se poder distribuir as requisições HTTP dos usuários entre os servidores *Web*, haverá a garantia de que a aplicação continuará disponível mesmo que um desses servidores venha a falhar. Deve-se levar em consideração que, dependendo do fabricante, pode-se utilizar mais de um balanceador de carga para evitar que uma falha no balanceador não torne a aplicação indisponível.

4.4 JNDI Tolerante a Falhas

Como explicado no capítulo 2, é necessário replicar o serviço JNDI em mais de uma instância do servidor J2EE para garantir que as informações dos registros não sejam perdidas após a eventual falha de uma das instâncias do servidor. Com a introdução da replicação ativa, todas as instâncias do servidor receberiam as requisições de registro e de consulta, garantindo a sincronização das bases de registros.

Uma vez que o acesso ao serviço JNDI é realizado pelos clientes através da classe que representa o contexto JNDI (`javax.naming.Context`), essa classe deverá ser alterada para interagir com o serviço de comunicação em grupo. Dessa forma, quando um cliente requisitar a criação de um novo registro, a requisição será recebida por todas as instâncias que formam o grupo de replicação. De forma similar, a consulta a um nome será realizada por todas as instâncias e o contexto receberá apenas a primeira resposta para repassá-la ao cliente como ilustrado na Figura 4.3.

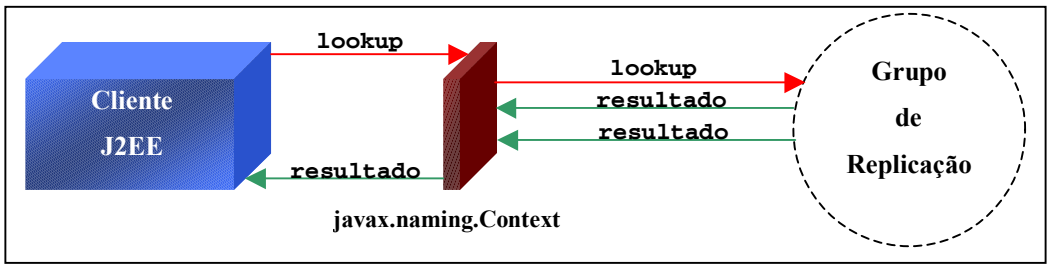

Figura 4.3 Funcionamento do JNDI Tolerante a Falhas

4.5 JMS Tolerante a Falhas

Utilizando o provedor JMS replicado proposto nesse projeto, não será necessário que os clientes JMS realizem nenhum tipo de sincronização entre as instâncias do provedor JMS. Cada mensagem enviada pelo cliente JMS será recebida nos destinos de todas as instâncias do provedor, de forma que, caso uma das instâncias falhe, outra instância poderá realizar a entrega da mensagem.

Para que o grupo de replicação permaneça transparente para os clientes JMS, os objetos produtores e consumidores deverão interagir diretamente com o serviço de comunicação em grupo. Dessa forma, a mensagem enviada pelo produtor será recebida por todas as instâncias do grupo de replicação e os consumidores serão responsáveis por filtrar mensagens repetidas como ilustrado na Figura 4.4.

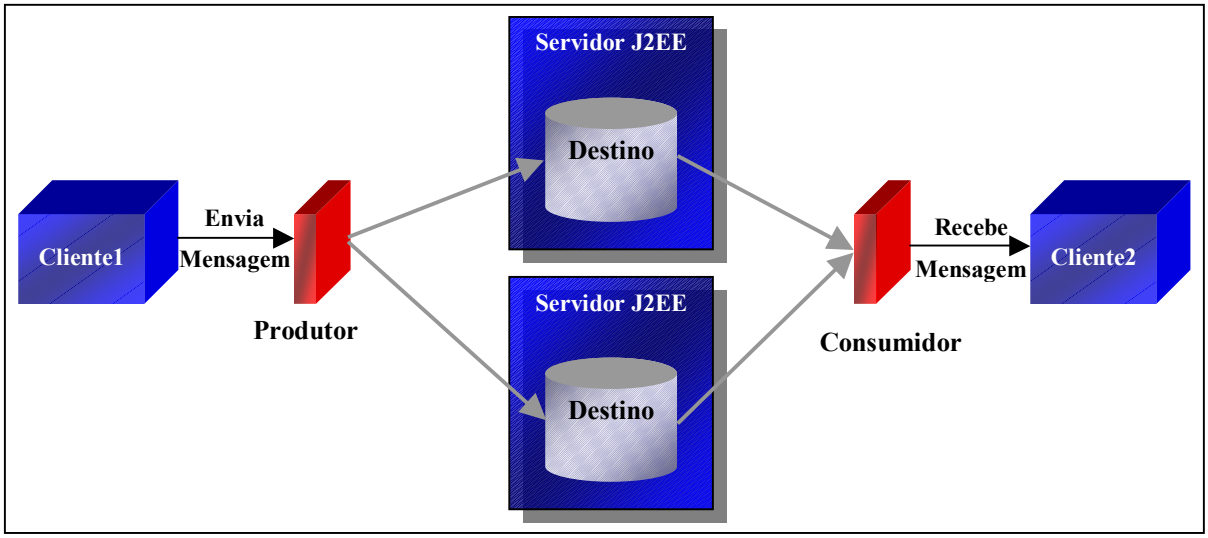

Figura 4.4 Funcionamento do provedor JMS replicado

O serviço JMS é utilizado pelos clientes inicialmente através dos objetos auxiliares Construtor de Conexão e Destino. Assim, o Construtor de Conexão deverá ser alterado para criar uma conexão com o grupo de replicação (através do serviço de comunicação em grupo) e não mais com uma instância específica. De forma similar, o Destino irá representar os destinos de todas as instâncias do provedor JMS.

4.6 Group Method Invocation

No RMI, o *stub*, por meio do protocolo de rede, envia a invocação do método para a implementação do objeto remoto, estando este em execução no mesmo ponto de rede do *stub* ou não. A nossa abordagem para prover confiança no funcionamento para aplicações baseadas em RMI consiste em implementar um mecanismo de invocação remota de método em um grupo de réplicas (*Group Method Invocation* ou GMI) através da alteração do comportamento dos *stubs* dos objetos remotos. Na nossa solução, o *stub* do objeto remoto irá interagir com o serviço de comunicação em grupo, de maneira que a invocação de método realizada pelo cliente seja enviada para o grupo de instâncias do objeto. Após enviar a requisição ao grupo, o *stub* permanecerá esperando pela primeira resposta do processamento do método para que o resultado da invocação (um objeto ou uma exceção) seja devolvido ao cliente. A Figura 4.5 ilustra o funcionamento do nosso projeto de GMI.

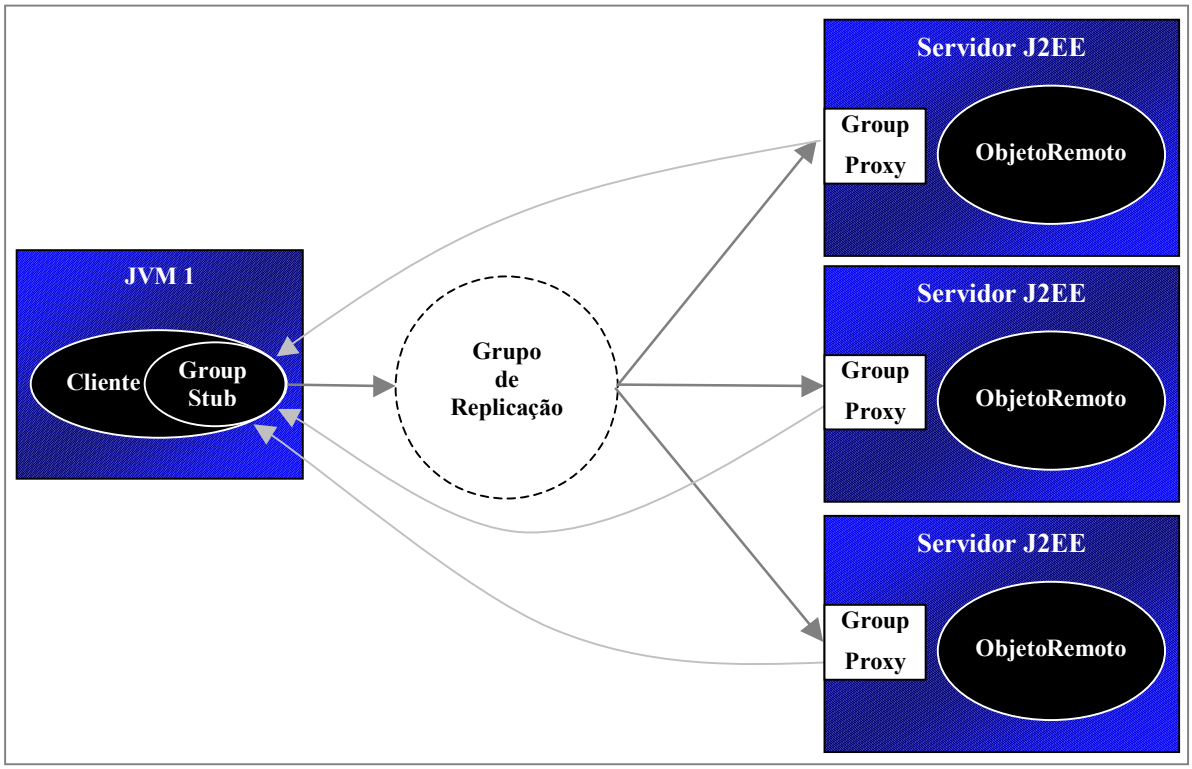

Figura 4.5 Funcionamento do GMI

Para utilizar essa implementação de GMI não é necessária nenhuma alteração no cliente uma vez que o *Group Stub* será responsável por enviar a invocação ao grupo de réplicas. Também não é preciso incluir nenhum tratamento na implementação do objeto

remoto, visto que o *Group Proxy* será responsável por receber as mensagens enviadas ao grupo de replicação.

4.7 EJB Tolerante a Falhas

Um componente EJB hospedado em um servidor J2EE é utilizado pelas aplicações clientes através de sua referência remota. A aplicação cliente primeiro obtém uma referência para um objeto remoto correspondente à interface *home* do componente para, com este objeto, poder obter uma referência a um objeto remoto correspondente ao componente EJB. Uma vez que a infra-estrutura de comunicação utilizada pelos componentes EJB é o RMI, este projeto fará uso do GMI apresentado na seção anterior para replicar as invocações aos componentes para todas as instâncias do servidor J2EE. Assim, no momento que a aplicação cliente obtiver uma referência ao objeto *home* via JNDI, estará, na verdade, obtendo um objeto *Group Stub* que fará com que as requisições para criar (métodos `create`) ou localizar (métodos `find`) as instâncias do EJB sejam enviadas ao grupo de replicação. De forma similar os retornos dos métodos do *home* serão objetos da classe *Group Stub* referentes aos componentes EJB.

Essa abordagem permitirá que os estados dos componentes permaneçam sincronizados, requisito necessário para que as réplicas possam dar continuidade no processamento da aplicação mesmo ocorrendo falha de uma ou mais instâncias.

4.8 Conclusão

Este capítulo apresentou um projeto de um servidor J2EE tolerante a falhas e transparente cuja abordagem para prover confiança no funcionamento é a introdução de replicação. Esta solução oferece uma completa transparência para o desenvolvedor, uma vez que não requer nem a alteração da camada cliente nem dos componentes da camada intermediária para fazer uso do mecanismo de replicação. Além disso, a falha de uma ou mais instâncias do servidor J2EE não irá implicar em atraso no tempo de resposta do sistema uma vez que as instâncias operacionais já processaram a requisição.

Capítulo 5: JBossFT

5.1 Introdução

Seguindo o projeto descrito no capítulo anterior, foi desenvolvido o JBossFT, que consiste da introdução dos mecanismos propostos para tolerar falhas apresentados no capítulo anterior, no servidor J2EE JBoss [JBoss].

Uma vez que a implementação desses mecanismos exige que as requisições aos serviços sejam replicadas por todas as instâncias na mesma ordem e de forma atômica, foi preciso incorporar uma ferramenta de comunicação em grupo que atendesse a esses requisitos.

Este capítulo apresenta brevemente o servidor JBoss e a ferramenta de comunicação em grupo utilizada, além de descrever os detalhes de implementação da replicação ativa para os serviços Java *Servlet*, JNDI e RMI. Uma avaliação dessa implementação é feita no final do capítulo para medir o desempenho do servidor utilizando a replicação ativa.

5.2 O Servidor de Aplicações JBoss

O JBoss é um produto *open source* que, além de seguir a especificação J2EE 1.2, é estruturado de tal forma que serviços adicionais podem ser incluídos sem que seu código precise ser alterado. Para tal, basta que o novo serviço implemente algumas interfaces do *framework* do JBoss para que ele seja executado como mais um serviço desse servidor [JBoss 01]. Esse foi o principal motivo que nos levou a utilizar o JBoss como a base da implementação de um servidor J2EE tolerante a falhas.

Desenvolvido inicialmente em 1999 para ser apenas um servidor para componentes EJB, a sua versão atual (2.4) está implementada totalmente em Java e oferece um ambiente completo para desenvolvimento e distribuição de aplicações J2EE. Este servidor utiliza a

tecnologia Java Management Extensions (JMX) [JBoss 01] com o objetivo de oferecer uma interface padronizada para gerenciamento de seus serviços, além de servir de base para sua arquitetura baseada em módulos. No JBoss, cada serviço (EJB, JNDI, etc.) constitui um módulo baseado na especificação JMX. O módulo central do JBoss se encarrega de gerenciar esses serviços através do JMX. Assim, é possível personalizar uma instância do servidor desabilitando alguns módulos para atender a um número reduzido de requisitos. É possível também incluir um novo serviço ou substituir a implementação de algum módulo existente sem a necessidade de alteração da lógica do servidor.

Um exemplo dessa facilidade de personalização pode ser constatado na própria distribuição do JBoss que permite que o desenvolvedor possa escolher entre o Jetty e o Tomcat 3.2 como contêiner para os *servlets* [JBoss]. Considerado como um dos mais populares ambientes para desenvolvimento de aplicações *Web*, o Tomcat faz parte do projeto Jakarta [Tomcat] e, além de ter seu código fonte liberado, oferece interoperabilidade com os principais servidores *Web* do mercado. Por esses motivos e por apresentarem estabilidade de execução, a plataforma escolhida para desenvolver os mecanismos de replicação ativa foi o JBoss em conjunto com o Tomcat.

5.3 O JavaGroups

O JavaGroups é um *toolkit* para desenvolvimento de aplicações que necessitam de comunicação *multicast* confiável [JavaGroups]. Ele pode ser utilizado para criar grupos de processos cujos membros podem trocar mensagens entre si. Através dessa ferramenta, é possível tanto enviar uma mensagem para o grupo quanto para um membro específico, sendo que o gerenciamento de afiliação do grupo é realizado pela própria ferramenta.

O projeto JavaGroups é constituído de duas partes: o JChannel e uma biblioteca de padrões de projeto. O JChannel é uma ferramenta de comunicação em grupo implementada em Java e se caracteriza por ser bastante flexível, uma vez que possibilita que as propriedades de comunicação entre os processos possam ser parametrizadas em tempo de execução. Isso é possível devido a sua pilha de protocolos que permite que os desenvolvedores possam adaptá-la aos requisitos da aplicação e às características da rede. Dessa forma, a carga extra de

comunicação poderá ser minimizada uma vez que somente os protocolos necessários estarão em uso. O JChannel disponibiliza um conjunto com vários protocolos (além de permitir que sejam criados novos) que podem ser combinados para atender a várias situações. Os principais protocolos oferecidos pelo JChannel são [JavaGroups]:

- Protocolo de transporte – UDP (IP *multicast*) e TCP – interfaces que utilizam diretamente o protocolo de rede;
- Protocolo de fragmentação de mensagens – utilizado para transmitir mensagens grandes em partes menores;
- Protocolo de envio de mensagem *unicast* e *multicast* confiável – realiza a retransmissão de mensagens perdidas;
- Protocolo de detecção de falha – utilizado para remover membros do grupo que apresentaram falhas;
- Protocolos de ordenação de mensagens – atômica (entrega para todos ou para nenhum), FIFO (*First In First Out* – ordenação de mensagens de um emissor), ordenação total (ordenação de mensagens independente do emissor) e causal (ordenação de mensagens que apresentam dependência);
- Protocolo para gerenciamento de afiliação de grupo; e
- Protocolo para criptografia de mensagens.

Quando um processo (emissor) necessita enviar uma mensagem para um grupo de processos através do JChannel, o emissor deve, primeiro, estabelecer uma conexão com o canal de comunicação do grupo. Essa conexão é realizada instanciando um objeto que representa esse canal (especificando opcionalmente o protocolo) para, em seguida, invocar o método `connect`, onde deve ser informado o nome do canal/grupo que se deseja conectar. Feito isso, o emissor pode enviar uma mensagem (um objeto Java) pelo canal, fazendo com que todos os processos conectados ao canal recebam a mensagem. O envio da mensagem é feito através do método `send` onde são informados o destino (nenhum para enviar a todos os processos do grupo) e o objeto a ser enviado. Para receber a mensagem, o processo invoca o método `receive` do canal que faz com que o processo permaneça bloqueado até que uma mensagem seja recebida.

Quando o canal recebe uma requisição de envio de mensagem, a mensagem a ser enviada é passada para a pilha de protocolos que, por sua vez, a repassa para o protocolo de

maior nível. Este protocolo processa a mensagem e a repassa para o protocolo seguinte e este procedimento continua até que o último protocolo da pilha disponibilize a mensagem na rede [JavaGroups]. O recebimento da mensagem ocorre de forma similar: o protocolo de mais baixo nível permanece esperando por mensagens na rede e, quando é identificada uma mensagem, ela é passada para o protocolo superior até que ela chegue ao canal. O canal armazena a mensagem em uma fila até que ela seja consumida pela aplicação. A aplicação pode também invocar o método `disconnect` para encerrar a conexão com o grupo de comunicação.

Além de oferecer um mecanismo para comunicação confiável em grupo, o projeto JavaGroups disponibiliza uma biblioteca de padrões de projeto que podem ser utilizados sobre qualquer ferramenta de comunicação em grupo [Ban 98]. Dessa forma, além de disponibilizar alguns blocos básicos (*building blocks*) úteis ao desenvolvimento da aplicação, o JavaGroups permite que a implementação permaneça independente da ferramenta de comunicação em grupo. Os blocos básicos se interpõem entre a aplicação e o canal de forma que a aplicação utilize os blocos básicos em vez do canal. Cada bloco básico representa um padrão de projeto voltado à comunicação em grupo e contém a lógica que o desenvolvedor da aplicação final deveria implementar utilizando o canal. Assim, os blocos básicos oferecem um nível maior de abstração para o desenvolvimento de aplicações que necessitam interagir com um grupo de processos. A Figura 5.1 ilustra como é feita a interação entre a aplicação e o JavaGroups.

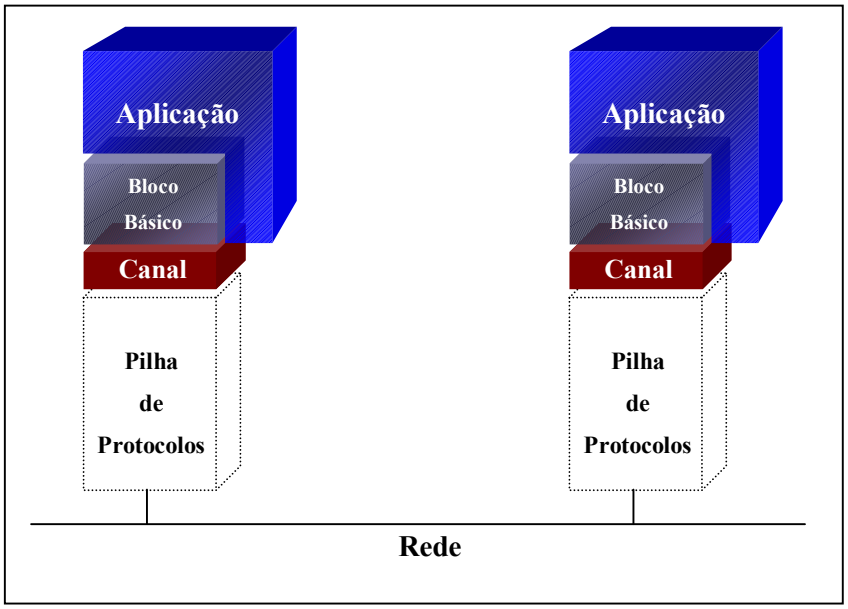

Figura 5.1 Arquitetura de uma aplicação utilizando o JavaGroups

Um exemplo de um bloco básico presente no JavaGroups é o *PullPushAdapter* [Ban 98]. Como foi explicado anteriormente, a aplicação deve ser responsável por verificar constantemente através do canal, se há alguma mensagem a ser recebida. Com o *PullPushAdapter*, a aplicação será notificada automaticamente no momento que uma mensagem for recebida. Basta que seja informado qual o canal e qual objeto deverá ser notificado (um objeto que implemente a interface `JavaGroups.MessageListener`) para que o *PullPushAdapter* invoque o método `Receive` do `MessageListener` informando a mensagem recebida. A Figura 5.2 ilustra a utilização do *PullPushAdapter*.

```
import JavaGroups.Channel;
import JavaGroups.JChannelFactory;
import JavaGroups.MessageListener;
import JavaGroups.PullPushAdapter;
import JavaGroups.Message;

public class Receptor implements MessageListener
{
  Channel         channel;
  PullPushAdapter adapter;

  public static void main(String[] args) throws Exception
  {
    new Receptor().iniciar();
  }

  public void iniciar() throws Exception
  {
    channel = new JChannelFactory().CreateChannel(null);
    channel.Connect("GrupoDeTeste");
    adapter = new PullPushAdapter(channel, this, null);
  }

  //Método definido na interface MessageListener
  public void Receive(Message msg)
  {
    System.out.println("Mensagem Recebida:" + msg.GetObject());
  }
}
```

Figura 5.2 Exemplo de utilização do *PullPushAdapter*

5.4 Implementação do *Group Proxy*

O *Group Proxy* consiste de um serviço em execução no JBoss aguardando pelas mensagens que estão sendo enviadas ao grupo de replicação. Para que o *Group Proxy* fosse executado em conjunto com o JBoss, foi preciso criar a sua interface de serviço. Essa interface herda de `org.jboss.util.ServiceMBean` e define as operações referentes ao novo serviço. A implementação dessa interface (o *Group Proxy* propriamente dito), além de implementar as operações de sua interface, implementa as operações do `ServiceMBean` que são: iniciar, parar, inicializar e finalizar o serviço. Para que o módulo central do JBoss possa gerenciar esse novo serviço através do JMX (invocando as operações de iniciar, parar, etc), basta que seja especificado no arquivo de configuração do servidor qual a classe do serviço.

No momento em que o servidor é iniciado, o JBoss requisita a inicialização de todos os seus serviços. Nessa inicialização, o *Group Proxy* irá estabelecer a conexão com o canal do grupo de replicação e instanciar um *PullPushAdapter*. Dessa forma, o *Group Proxy* será notificado pelo *PullPushAdapter* sempre que for recebida uma mensagem pelo canal. Através da mensagem recebida, o *Group Proxy* recupera a requisição e descobre qual membro do grupo é o emissor da mensagem. Com a requisição obtida, o *Group Proxy* a redireciona para o serviço apropriado da instância do servidor. Após o serviço processar a requisição, o *Group Proxy* envia a reposta do processamento para o integrante do grupo de replicação que originou a requisição.

Na interface de serviço do *Group Proxy* foram definidas algumas operações para permitir que o administrador possa tanto obter informações quanto alterar parâmetros de execução desse serviço. O administrador pode, por exemplo, descobrir o número de membros que fazem parte do grupo ou alterar o nome e os parâmetros do protocolo do canal em tempo de execução. O gerenciamento do *Group Proxy* e dos outros serviços do JBoss é realizado através de uma interface *Web* oferecida pelo próprio servidor.

5.5 Alteração das Bibliotecas Clientes dos Serviços

O envio de mensagens utilizando o canal do JChannel é baseado em um modelo assíncrono: o emissor requisita o envio de uma mensagem e, após a mensagem ser recebida

pelo grupo, seu processamento continua. Vários tipos de aplicações necessitam implementar um protocolo de requisição/resposta, ou seja, enviar uma mensagem e só continuar o processamento após o recebimento de uma outra mensagem em resposta à primeira. Com o JChannel, o emissor deve implementar a lógica para permanecer bloqueado após enviar uma mensagem pelo canal. O emissor deve verificar cada mensagem recebida para continuar seu processamento apenas quando for recebida a resposta da sua requisição.

Na nossa implementação, há várias situações que requerem envio de mensagem de forma síncrona. Assim, foi criado um bloco básico chamado *Request Dispatcher* que implementa a funcionalidade do protocolo de requisição/resposta. Para utilizar esse bloco básico, o emissor precisa apenas instanciar um objeto dessa classe, informando o canal a ser usado, e invocar o método `request` passando a mensagem a ser enviada ao grupo. O retorno desse método é a primeira mensagem de resposta recebida pelo *Request Dispatcher*. A Figura 5.3 ilustra a utilização do *Request Dispatcher*.

```
import JavaGroups.Channel;
import JavaGroups.JChannelFactory;
import jbossft.group.RequestDispatcher;

public class RequestSender
{
  public static void main(String[] args) throws Exception
  {
    Channel channel = new JChannelFactory().CreateChannel(null);
    channel.Connect("GrupoDeTeste");
    RequestDispatcher reqDisp = new RequestDispatcher(channel);
    Message req = new Message("Requisição de Teste");
    Message res = reqDisp.request(req);
    System.out.println("Resposta :" + res.GetObject());
    channel.Disconnect();
  }
}
```

Figura 5.3 Exemplo de utilização do *Request Dispatcher*

Esse bloco básico utiliza o canal especificado para enviar as mensagens ao grupo o qual ele representa. No momento em que o emissor realiza uma requisição, o *Request Dispatcher* adiciona um cabeçalho à mensagem contendo um identificador de requisição. Através do canal do JavaGroups, o *Request Dispatcher* envia essa mensagem como um *multicast* para o grupo ao qual ele pertence e permanece aguardando pelo recebimento de uma mensagem. Quando o *Request Dispatcher* recebe uma mensagem através do grupo de

replicação, ele verifica se a identificação contida no cabeçalho corresponde à identificação de requisição da mensagem enviada. Caso as identificações sejam iguais, essa mensagem é devolvida ao cliente que realizou a requisição, e as demais mensagens não serão mais consideradas. Se as identificações não coincidirem, o *Request Dispatcher* volta a aguardar pela resposta da requisição. Esta última situação pode ocorrer quando alguma réplica atrasa a resposta de uma requisição anterior.

5.5.1 ServletFT

Na versão utilizada, o Tomcat disponibiliza duas versões do *plug in* responsável por redirecionar as requisições a conteúdo dinâmico para os *servlets*: uma para o servidor *Web* Apache do Apache Group e outra para o servidor *Web* Internet Information Services (IIS) da Microsoft Corporation. Cada versão é implementada em código nativo utilizando as interfaces de acesso do produto correspondente, uma vez que o *plug in* deverá estar em execução em conjunto com o servidor *Web*.

O servidor de páginas irá invocar o *plug in* toda vez que for recebida uma requisição a um *servlet* ou JSP. O *plug in*, por sua vez, repassará a requisição ao Tomcat de acordo com o seu protocolo de comunicação utilizando *sockets* TCP/IP. Na instância do Tomcat, há um serviço chamado Conector responsável por receber as requisições e repassá-las ao gerenciador de contêineres do Tomcat, a fim de serem processadas pelos *servlets*. O Tomcat oferece uma versão do *Conector* para receber requisições do *plug in* (*Ajp12ConnectionHandler*) e outra versão para receber requisições no formato HTTP originadas diretamente do navegador *Web* (*HttpConnectionHandler*).

Para que a nossa implementação permanecesse independente de plataforma (tanto do servidor *Web* quanto do sistema operacional), em vez de modificar as versões do *plug in* para interagirem com a ferramenta de comunicação em grupo, foi criado um novo *Conector* utilizando o *framework* do Tomcat. Este novo *Conector* (*GroupConnectionHandler*), da mesma forma que o *Ajp12ConnectionHandler*, recebe as requisições do *plug in*, mas envia as requisições para o grupo de replicação através do JChannel. Através de classes auxiliares fornecidas pelo Tomcat, o *GroupConnectionHandler* recupera a requisição enviada pelo *plug in* e cria um objeto contendo os dados da requisição. Esse objeto é enviado pelo *Request Dispatcher* para ser recebido pelo *Group Proxy* de cada instância do JBossFT. O *Group*

Proxy, ao receber a requisição, repassa-a para o gerenciador de contêineres de sua instância e devolve o resultado (a página no formato HTML) para o *GroupConnectionHandler*.

Para implantar essa solução é necessário apenas executar uma instância do Tomcat na mesma máquina do servidor *Web* e especificar na configuração do Tomcat que o Conector a ser utilizado será o *GroupConnectionHandler*. Dessa forma, nenhuma alteração no código fonte do Tomcat é necessária. A Figura 5.4 ilustra a interação dos componentes descritos.

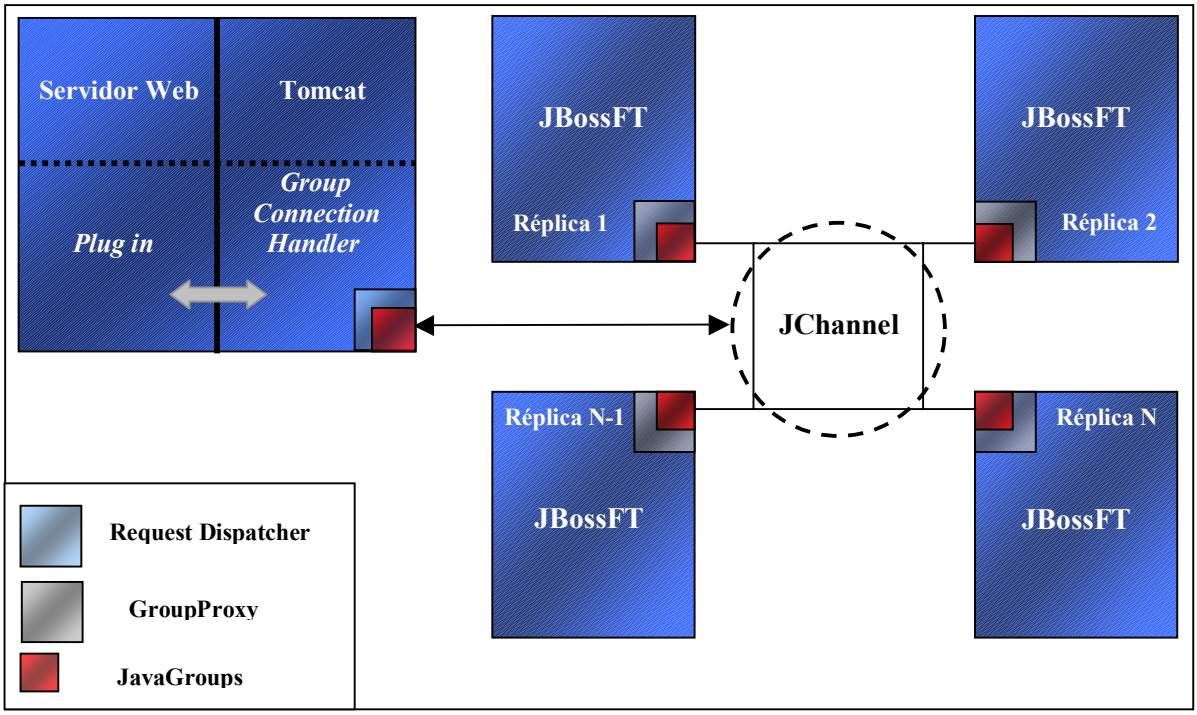

Figura 5.4 Implantação do JBossFT para a camada *Web*

5.5.2 JNDIFT

Para facilitar a implementação do serviço JNDI tolerante a falhas, foi implementado o bloco básico *Group Method Call* que consiste em um modelo de invocação replicada de método. Através dele, uma classe pode invocar um método a ser processado em todas as instâncias do grupo de replicação e receber a primeira resposta gerada. O cliente desse bloco básico precisa apenas informar o nome da classe, qual o nome do método dessa classe que deseja invocar e os argumentos do método. Com isso o *Group Method Call* monta uma mensagem representando o método e seus parâmetros e a envia utilizando *Request Dispatcher* para que esse método seja processado em todas as instâncias do grupo de replicação.

Em sua forma original, o `Context` implementado pelo JBoss é apenas um cliente de um objeto remoto hospedado no servidor: em cada método do `Context`, é feita uma invocação via RMI para esse objeto remoto. Em nossa solução, o `Context` foi alterado para passar a utilizar o *Group Method Call* permitindo que a invocação ao método seja replicada para todas as instâncias do objeto remoto no grupo de replicação. A Figura 5.5 ilustra as modificações feitas no servidor de aplicação JBoss para implementar o JBossFT, que incorpora os mecanismos para tolerância a falhas descritos acima.

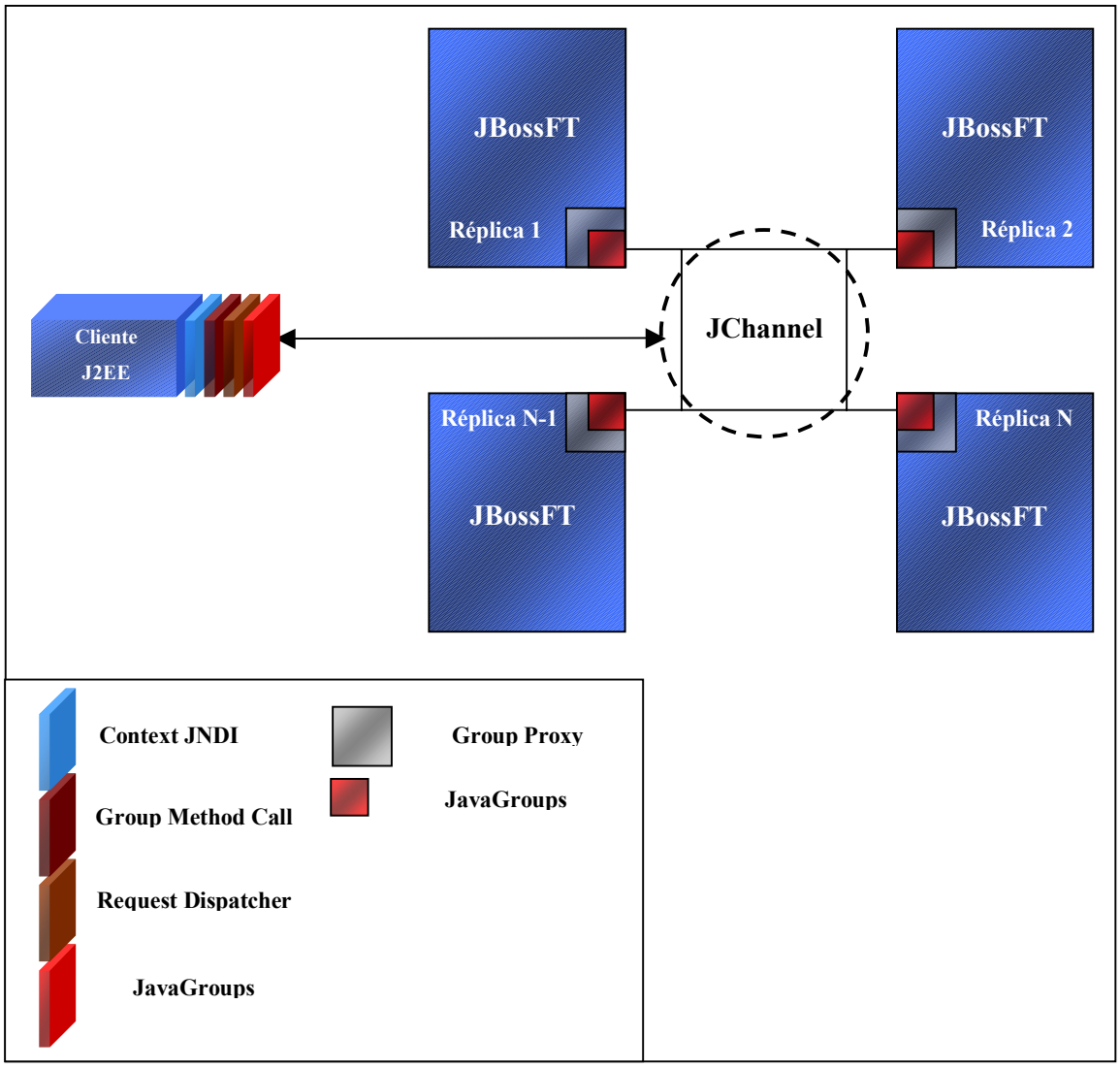

Figura 5.5 Arquitetura do JNDI Tolerante a Falhas

5.5.3 GMI

Como descrito no capítulo 2, quando o objeto cliente deseja utilizar um objeto remoto, ele realiza uma consulta ao serviço de nomes para obter o *stub* desse objeto. Deve-se destacar que o objeto cliente não sabe qual a classe do *stub*, e sim que ele implementa a mesma interface remota implementada pelo objeto remoto. Essa característica permitiu que fosse possível retornar uma outra classe *stub* no lugar da classe do RMI padrão no momento que o objeto cliente consulta o serviço de nomes. Essa outra classe, a *Group Stub*, será responsável por interagir com o JChannel para replicar as invocações de método. A classe do *Group Stub* é criada dinamicamente no momento em que é feita a consulta ao serviço JNDI como ilustrado na Figura 5.6.

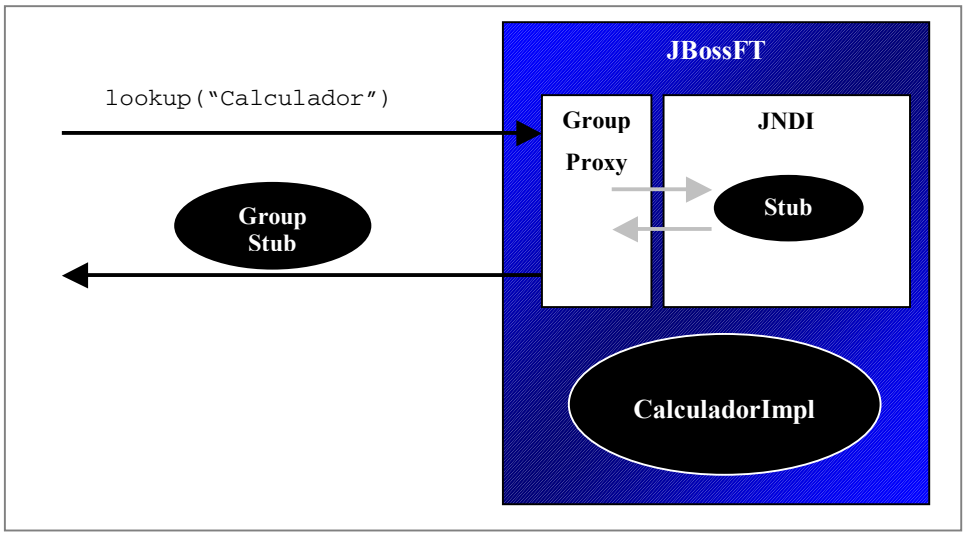

Figura 5.6 Criação dinâmica do *Group Stub*

A criação dinâmica do *Group Stub* é realizada pelo *Group Proxy* após o serviço JNDI processar a requisição de consulta ao objeto remoto. Uma vez que as requisições do JNDI replicado são recebidas pelo *Group Proxy*, este serviço analisa o retorno das consultas JNDI para detectar quando o resultado da consulta é um *stub* de um objeto remoto. Caso a consulta seja a um objeto remoto, o *Group Proxy* descobre qual a interface remota que o *stub* implementa através da API Java *Reflection*. Esta API permite obter informações (nome da classe, métodos, etc) de qualquer objeto [Campione et al. 00] em tempo de execução. Após identificar a interface remota do *stub*, o *Group Proxy* irá criar a classe do *Group Stub* para este objeto remoto utilizando a classe `java.lang.reflect.Proxy`. Esta classe do Java *Reflection* é responsável pela criação de instâncias de classes *Proxy* dinâmicas. No momento

de criação de uma instância de uma *Proxy* dinâmica, é possível especificar quais interfaces essa instância irá implementar. Dessa forma, o *Group Proxy* irá devolver um *Group Stub* que implementa as mesmas interfaces do *stub* do objeto remoto sendo que este processo ficará totalmente transparente para o objeto cliente e não exigirá nenhuma alteração na implementação do objeto remoto.

Após o cliente realizar a consulta ao objeto remoto, a referência obtida (o *Group Stub*) irá estabelecer uma conexão com o canal do grupo de replicação e instanciar o *Group Method Call*. Através desse bloco básico, o *Group Stub* poderá replicar as invocações a métodos do objeto remoto realizadas pelo objeto cliente. Também nesse caso, o serviço *Group Proxy* que executa em cada instância do servidor J2EE é responsável por interceptar as requisições enviadas ao grupo e repassá-las ao objeto remoto. Após a requisição ser processada pelo objeto remoto, o *Group Proxy* devolve a reposta da invocação para o *Group Stub* do objeto cliente que fez a invocação. O *Group Stub*, que permanece esperando a resposta do método, retorna a primeira resposta recebida para o objeto cliente que realizou a invocação.

5.6 Testes e Avaliações de Desempenho

Com o objetivo de validar a implementação e avaliar a perda de desempenho sofrida quando os mecanismos de replicação ativa são utilizados, foi desenvolvida uma aplicação que foi executada tanto no JBossFT quanto nos servidores de aplicação JBoss e WebLogic. A aplicação de teste não foi executada em outros servidores comerciais disponíveis (ex. Borland AppServer e Sybase EAServer) dado que, até onde sabemos, apenas o WebLogic implementa mecanismos para tolerar falhas de *servlets*.

A aplicação consiste de um sistema de comércio eletrônico simplificado, implementado através de alguns *servlets* que acessam uma base de dados e o serviço JNDI.

Os servidores foram executados em uma plataforma composta de processadores Pentium II 400Mhz, com 256Mb de RAM, utilizando a máquina virtual Java 1.3.0 da SUN Microsystems e conectados por uma rede Ethernet 10Mbps.

O usuário inicia o uso da aplicação através do *servlet* que apresenta o catálogo de produtos. Através do catálogo, o usuário adiciona os produtos que deseja comprar em seu "carrinho de compras" e em seguida confirma o pedido passando seus dados pessoais.

Para verificar se a solução atende aos objetivos propostos, foi realizado um experimento no qual foram provocadas sucessivas falhas em uma das instâncias do grupo de replicação enquanto eram realizadas requisições à aplicação de forma contínua. Para este experimento foi desenvolvido um serviço que, periodicamente, suspende a execução da instância do JBossFT da máquina na qual ele está em execução. Este serviço permite que a instância do servidor funcione corretamente por 5 minutos e, decorrido esse tempo, interrompe a execução por um minuto. Para gerar as requisições à aplicação foi utilizado o Web Application Stress Tool[1] que simula um navegador *Web*. Esta ferramenta, além de medir o tempo que leva para uma página ser recebida após a requisição HTTP ser enviada, reporta se alguma requisição enviada deixou de ser respondida. Neste experimento, o Web Application Stress Tool gerou requisições à aplicação por 3 horas e nenhuma delas deixou de ser respondida pelo grupo de replicação formado por quatro instâncias mesmo com falhas sendo provocadas periodicamente em uma das instâncias.

Para realizar as avaliações de desempenho, também foi utilizado o Web Application Stress Tool para enviar requisições para a aplicação por 10 minutos e medir o tempo médio de resposta. O gráfico de barras da Figura 5.7 mostra os tempos médios de resposta para o JBoss, o JBossFT e o WebLogic, sendo que esses últimos foram executados sem a ativação dos mecanismos de tolerância a falhas, i.e. com o número de réplicas igual a 1.

[1]Disponível em http://home.rte.microsoft.com/.

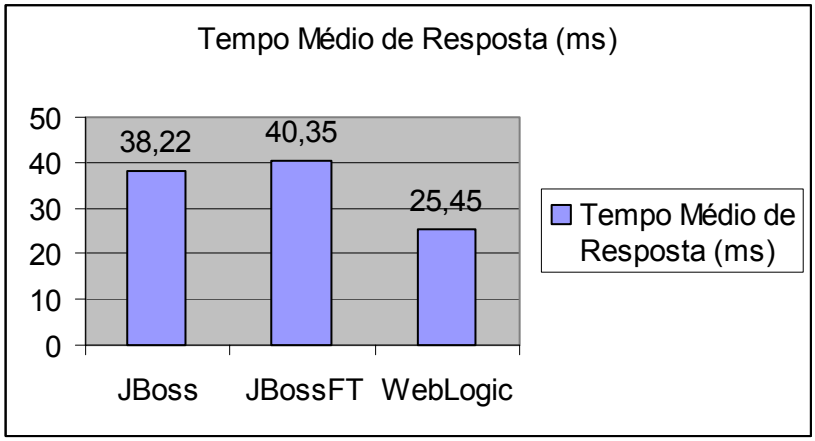

Figura 5.7 Tempo médio de resposta sem tolerância a falhas

Como pode ser visto, mesmo com um grupo formado por apenas uma réplica, o JBossFT é mais lento que o JBoss (aproximadamente 5,6%). Isso ocorre devido à necessidade da comunicação se processar com o auxílio da ferramenta de comunicação em grupo. O WebLogic por sua vez é bem mais rápido que o JBoss e o JBossFT (50,2% e 58,5%, respectivamente). Nós acreditamos que isso se deve ao fato do WebLogic ser um produto comercial que, devido às pressões do mercado, tem requisitos de desempenho bem mais exigentes do que os impostos ao JBoss e ao JBossFT, o primeiro um sistema *open source* e o segundo uma extensão do primeiro apoiado sobre uma ferramenta de comunicação em grupo não comercial.

Nosso segundo experimento objetivou identificar a perda de desempenho sofrida pelos servidores JBossFT e WebLogic quando o número de réplicas no grupo crescia. O gráfico da Figura 5.8 mostra os tempos médio de resposta para grupos variando de 1 a 4 réplicas.

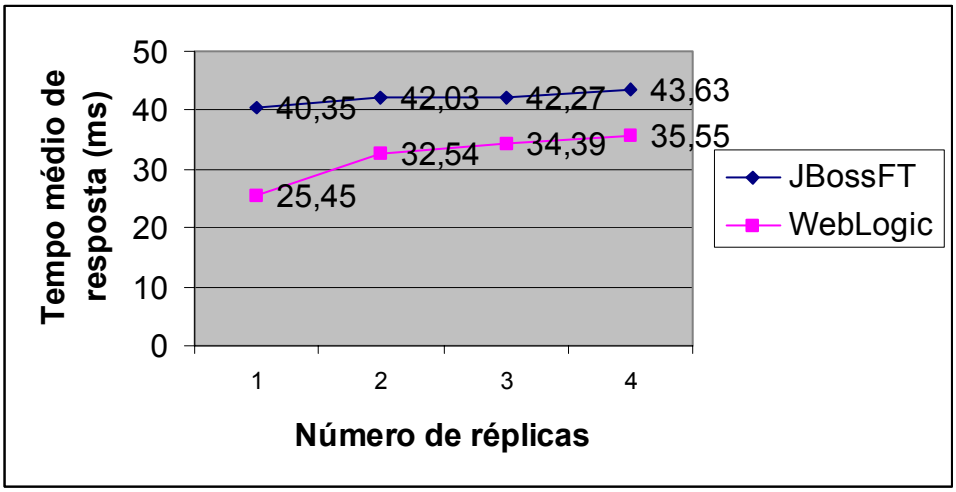

Figura 5.8 Tempo médio de resposta com tolerância a falhas

Como esperado, o desempenho diminui à medida que um número maior de falhas pode ser tolerado. Entretanto, o impacto do aumento do número de réplicas é mais acentuado no caso do WebLogic que usa replicação passiva do que no caso do JBossFT que usa replicação ativa. Por exemplo, a configuração com 3 réplicas do JBossFT é apenas 4,8% mais lenta que a configuração sem tolerância a falhas, enquanto que para o WebLogic o desempenho diminui em 35,1% para a mesma configuração.

Com o objetivo de comparar o tempo de recuperação do JBossFT com o do WebLogic, realizamos um experimento onde uma falha é provocada em uma réplica no mesmo momento em que é feita uma requisição ao grupo. Nesse teste, utilizamos um grupo de quatro réplicas e configuramos o simulador de requisições para gerar 300 requisições, sendo que, a cada 100 requisições, foi provocada a falha de uma das réplicas. O gráfico da Figura 5.9 mostra o tempo de resposta do JBossFT e do WebLogic para cada requisição.

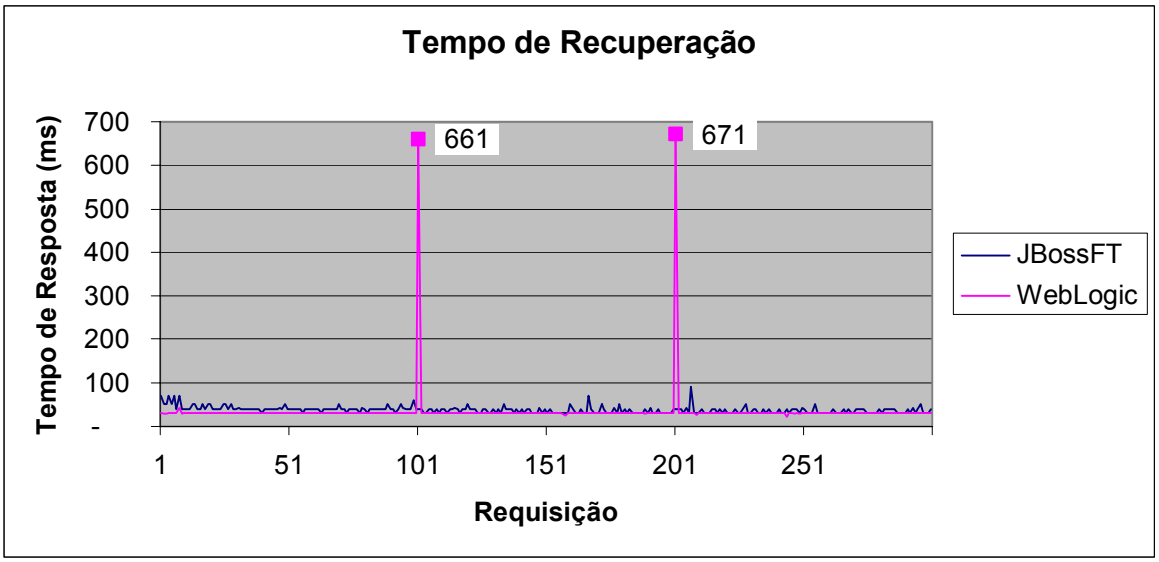

Figura 5.9 Tempo de Recuperação do JBossFT e do WebLogic

Como pode ser constatado, o tempo de resposta do WebLogic aumentou aproximadamente 1.900% quando é feita uma requisição após uma falha ser provocada em uma das réplicas. Já o JBossFT apresentou uma aumento de apenas 30% no seu tempo de resposta após a ocorrência das falhas.

5.7 Conclusão

Por estar implementado com base no JBoss e no Tomcat, o JBossFT irá herdar a estabilidade de execução desses produtos, uma vez que eles já se encontram bastante amadurecidos e contam com um grande número de desenvolvedores e colaboradores. Através da replicação ativa introduzida pelo JBossFT, a aplicação poderá continuar disponível mesmo que f instâncias do servidor venham a falhar num grupo de N réplicas onde $N = f + 1$.

Como foi apresentado neste capítulo, a implementação da replicação ativa não exigiu alteração na lógica dos serviços nem do JBoss nem do Tomcat. Uma vez que os mecanismos foram adicionados sob a forma de novos serviços, é possível introduzi-los facilmente em futuras versões desses produtos.

Com a utilização do JavaGroups e dos blocos básicos representando padrões de projetos, a implementação da replicação ativa não se tornou dependente da ferramenta de comunicação em grupo. Dessa forma, é possível substituir o JChannel por um mecanismo de ordenação de mensagens mais "leve" sem alterar a implementação da solução. Essa substituição poderá melhorar os tempos de resposta do grupo de replicação uma vez que o JChannel realiza gerenciamento de afiliação de grupo, um requisito que não é necessário nesta solução e que introduz um carga extra na comunicação entre as réplicas.

Com o JChannel, o envio de mensagens para o grupo é feito através de *broadcast*, o que impossibilita que a aplicação cliente e as instâncias do JBossFT sejam distribuídas em uma WAN. Para atender a uma situação como esta, seria necessário utilizar uma ferramenta de comunicação em grupo que envie as mensagens diretamente aos integrantes do grupo.

Capítulo 6: Conclusões

6.1 Discussão

Esta dissertação apresentou uma solução para desenvolver aplicações J2EE com requisitos de confiança no funcionamento onde os mecanismos para tolerar falhas são totalmente transparentes para o desenvolvedor final. Dessa forma, o esforço do desenvolvimento pode se concentrar na regra de negócio uma vez que a infra-estrutura da plataforma será responsável por garantir a continuidade do serviço.

Como visto no Capítulo 2, a plataforma J2EE foi criada com o objetivo de simplificar o desenvolvimento de sistemas coorporativos e de aplicações *Web*. Através da especificação de vários tipos de serviços, essa plataforma oferece suporte para criar soluções para as mais variadas situações. Além disso, a distribuição das aplicações torna-se mais dinâmica e portável, visto vez que elas são hospedas no servidor de aplicações que segue a especificação J2EE. Uma vez que o processamento dos componentes da aplicação é realizado no servidor J2EE, a disponibilidade do sistema como um todo torna-se dependente do servidor e do *hardware* no qual ele está executando. Apesar de ser possível instalar os componentes da aplicação em mais de uma instância do servidor, a especificação da plataforma J2EE não oferece nenhum suporte para coordenar as réplicas do servidor J2EE. Assim, caso ocorra uma falha em uma das instâncias, a aplicação deve detectar e tratar a falha para que o usuário possa continuar a utilizar a aplicação. Esse tipo de solução aumenta a complexidade das aplicações de forma significativa, além de obrigar que toda aplicação inclua os mecanismos de tratamento de falhas.

Devido a esses problemas, surgiram várias implementações de servidor J2EE disponibilizando algum mecanismo de tolerância a falhas. Como apresentado no Capítulo 3, esses servidores são baseados em um modelo de replicação passiva em que a salvaguarda do estado do serviço é realizada periodicamente ou a cada requisição processada. De uma maneira geral, essas soluções requerem que seja feito algum tipo de tratamento por parte da

aplicação e, em muitos casos, trocam confiabilidade por desempenho ao definirem pequenas "janelas de vulnerabilidade" durante as quais uma falha pode tornar o sistema inconsistente. Além disso, o grupo de replicação não é transparente para o cliente: em todos os casos, quando se deseja utilizar um dos serviços do servidor de aplicações, deve-se especificar qual instância do grupo de replicação deverá responder pelo serviço. Deve-se destacar também que no período em que essa pesquisa foi realizada, nenhum dos servidores provia qualquer mecanismo para tolerância a falhas para o serviço JMS[2].

A nossa proposta teve como base implementar um serviço de replicação ativa em um servidor J2EE de código aberto com o objetivo de oferecer alta confiabilidade e transparência para as aplicações. Essa implementação não exigiu que a lógica de nenhum serviço J2EE precisasse ser modificada. Foi necessário apenas que um novo serviço (*GroupProxy*) fosse adicionado e que as bibliotecas clientes fossem modificadas. No projeto do servidor J2EE apresentado, o grupo de réplicas permanece totalmente transparente para aplicação e, em caso de falha de um ou mais servidores, o cliente poderá receber a resposta do seu processamento imediatamente uma vez que sua requisição é processada pelas réplicas ainda operacionais.

Nossos experimentos demonstraram que, embora o nosso sistema tenha um desempenho pior do que outras soluções disponíveis, ele acrescenta uma carga extra pequena para o servidor de aplicações sobre o qual ele se baseia (para um grupo de 3 réplicas o JBossFT é apenas 10,6% mais lento que o JBoss).

6.2 Direções para Trabalhos Futuros

Uma das restrições da solução apresentada é a obrigatoriedade de existir replicação de banco de dados. Como cada instância realiza o processamento das requisições aos serviços, em varias situações essas requisições resultam em atualizações no banco de dados o que exige que cada instância atualize sua própria cópia da base de dados. Apesar dessa abordagem oferecer um nível maior de confiança no funcionamento, a solução perde em flexibilidade, por

[2] Dada a importância mercadológica da tecnologia J2EE e o número de empresas de grande porte que provêem produtos comerciais que implementam a plataforma J2EE, essa é uma área em constante e rápida evolução. Portanto, o leitor deve levar esse fato em consideração quando avaliar a comparação entre tecnologias feita no Capítulo 3 dessa dissertação.

não oferecer a alternativa de utilizar apenas um banco de dados para todo o grupo de replicação. Esta alternativa pode ser mais apropriada para determinadas organizações que já utilizam os mecanismos de tolerância a falhas do próprio sistema gerenciados de banco de dados. Para oferecer esse tipo de configuração, poderia ser criado um serviço que atuasse em conjunto com o banco de dados para filtrar as sentenças SQL replicadas, possibilitando que apenas a primeira requisição fosse executada no banco de dados.

Uma importante funcionalidade que deve ser incorporada ao JBossFT é o mecanismo de reconfiguração de grupo. Através dele, novas réplicas podem ser adicionadas no grupo de replicação em substituição de instâncias que venham a falhar. No atual estágio do projeto, isso não é possível uma vez que a instância a ser adicionada não possui os estados atuais dos serviços.

Com o objetivo de melhorar desempenho, pode-se diminuir a quantidade de respostas enviadas ao cliente. Para isso, poderia ser definida uma política de prioridades para que apenas as duas ou três réplicas de maior prioridade enviem suas respostas. Esse mecanismo irá diminuir a carga extra de comunicação caso o grupo de replicação possua um grande número de instâncias.

Além disso, para oferecer uma solução completa para o desenvolvimento de aplicações J2EE com requisitos de confiança no funcionamento, é necessário implementar os mecanismos que irão realizar replicação ativa para os serviços JMS e EJB como descritos no Capítulo 3.

7 - Referências Bibliográficas

[Allaire 00] Allaire Corporation. “Using Allaire ClusterCATS”. set. 2000.

[Allaire 01-1] Allaire Corporation. “Advanced Configuration Guide”. 2001.
http://www.allaire.com/images/objects/news/ACG.pdf

[Allaire 01-2] Allaire Corporation. “Developing Applications with Jrun”.2001.
http://www.allaire.com/documents/jr31/devapp.pdf

[Allaire] Allaire Corporation. “Allaire Jrun”.
http://www.allaire.com/Products/JRun

[Ban 98] B. Ban. “JavaGroups – Group Communication Patterns in Java”. 1998. http://www.cs.cornell.edu/home/bba/Patterns.ps.gz

[Baratloo et al. 98] A. Baratloo, P. E. Chung, Y. Huang, S. Rangarajan, and S. Yajnik. “Filterfresh: Hot replication of Java RMI server objects”. In *Proceedings of the Fourth USENIX Conference on Object-Oriented Technologies and Systems*, pages 65-78, Santa Fé, NM, April 1998.

[BEA 01-1] BEA Systems. “Introduction to BEA WebLogic Server”.
http://edocs.bea.com/wls/docs60/pdf/intro.pdf

[BEA 01-2] BEA Systems. “Using WebLogic Server Clusters”.
http://edocs.bea.com/wls/docs60/pdf/cluster.pdf

[BEA 01-3] BEA Systems. “Programming WebLogic Enterprise JavaBeans”.

http://e-docs.beasys.com/wls/docs60/pdf/ejb.pdf

[BEA] BEA Systems. “BEA WebLogic Server”.
http://www.beasys.com/products/weblogic/server/index.shtml

[Bodoff 01] Stefhanie Bodoff et al, “The J2EE Tutorial”, 2001

[Borland 01-1] Borland Software Corporation, “Clustreing in Borland AppServer”, jan. 2001.

http://www.borland.com/appserver/papers/clustering.html

[Borland 01-2] Borland Software Corporation, “Borland AppServer 4.5 FAQ”, mar 2001.

http://www.borland.com/devsupport/appserver/faq/45/index.html

[Borland 01-3] Borland Software Corporation, “Borland AppServer - An Integrated Solution for Developing, Deploying, and Managing Distributed Multi-tier Applications”, jan. 2001.

http://www.borland.com/appserver/papers/appsrvr/

[Borland 01-4] Borland Software Corporation, “Borland AppServer User’s Guide”, 2001.

http://www.borland.com/techpubs/books/appserver/appserver45/users-guide/bas45users-guide.zip

[Campione et al. 00] M. Campione, K. Walrath, A. Huml, The Java Tutorial: Third Edition, Addison-Wesley, 2000

[Campione et al. 98] M. Campione, K. Walrath, A. Huml et al, The Java Tutorial Continued: The Rest of JDK, Addison-Wesley, 1998

[Cattel-Inscore 01] R. Cattel e Jim Inscore, Criando aplicações comerciais com a plataforma Java 2, Enterprise Edition, Editora Campus, 2001.

[Cristian 91] F. Cristian, "Understanding Fault-Tolerant Distributed System", Communications of the ACM, 34(2):56-78, fev. 1991

[Deshpande 00] Salil Deshpande, "Clustering: Transparent Replication, Load Balncing, and Failover", CustomWare, jan. 2000.

http://www.borland.com/appserver/papers/11501_clustering.pdf

[Ensemble] The Ensemble Distributed Communication System. http://www.cs.cornell.edu/Info/Projects/Ensemble/index.htm

[Flores 01] Flores, Tonia A. "A Java Message Service Primer". mai. 2001.

http://www.oreillynet.com/pub/a/onjava/2001/05/03/jms_primer.html

[Haase 01] Haase, Kim. "Java Message Service Tutorial". mai. 2001.

http://java.sun.com/products/jms/tutorial/1_3-fcs/doc/jms_tutorialTOC.html

[Hagiont-Boyer 01] D. Hagimont, F. Boyer. "A Configurable RMI Mechanism for Sharing Distributed Java Objects". *IEEE Computer Society Press*, 5(1):36-43, 2001.

[Hess 00] D. A. Hess. "PowerTier for Enterprise JavaBeans", Gartner, mar 2000.

http://enterprise.cnet.com/enterprise/0-9563-707-2026476.html

[Hunter-Crawford 01] Jason Hunter e William Crawford, Java Servlet Programming, 2 ed, O'Reilly & Associates, 2001.

[Jalote 94] P. Jalote, Fault Tolerance in Distributed System, Prentice-Hall, 1994.

[JavaGroups] JavaGroups. http://www.javagroups.com

[JBoss 01] JBoss Manual. http://www.jboss.org/online-manual/HTM. 2001

[JBoss] JBoss. http://www.jboss.org/

[Kassem et al. 00] Nicholas Kassem et al, Designing Enterprise Applications with the Java 2 Platform, Enterprise Edition, Addison-Wesley, 2000

[Kielmann 00] J. Maassen, T. Kielmann, H.E. Bal. "Generalizing Java RMI to Support Efficient Group Communication". In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 88-96, 2000.

[Melliar-Smith 00] N. Narasimhan, L.E. Moser and P.M. Melliar-Smith. "Transparent Consistent Replication of Java RMI Objects". In *International Symposium on Distributed Objects and Applications*, 2000.

[Persistence 99-1] Persistence Software Inc. "PowerTier 6.0 for Enterprise JavaBeans, A Technical White Paper".

http://www.persistence.com/sources/download/whitepapers/pt-ejb_tech.pdf

[Persistence 99-2] Persistence Software Inc. "PowerTier for Enterprise JavaBeans Programming Guide", set 1999

[Persistence] Persistence Software Inc. "PowerTier for J2EE"

http://www.persistence.com/powertier/info.html

[Powell 92] D. Powell (Ed.), Delta-4 A Generic Architeture for Dependable Distributed Computing, Springer-Verlag, 1992.

[RFC1034] "Domain Names - Concepts and Facilities". http://www.ietf.org/rfc/rfc1034.txt.

[RFC2251] Lightweight Directory Access Protocol (v3). http://www.ietf.org/rfc/rfc2251.txt.

[Roman 99] Ed Roman, Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition, John Wiley & Sons, 1999

[Schneider 90] F. B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". *ACM Computing Surveys*, 22(4):299-319, dec 1990.

[SUN 01] SUN Microsystems. "Java Message Service Specification v1.0.2". ago. 2001.

[SUN 99-1] SUN Microsystems. "Enterprise JavaBeans Specification version 1.1". 1999

[SUN 99-2] SUN Microsystems. "Java 2 Paltform Enterprise Edition Specification version 1.2". 1999

[SUN 99-3] SUN Microsystems. "Simplified Guide to the Java 2 Platform, Enterprise Edition". 1999

[SUN 99-4] SUN Microsystems. "Java Servlet Specification, v2.2". 1999.

[SUN 99-5] SUN Microsystems. "Java Naming and Directory Interface 1.2 Application Programming Interface", jul. 1999

Application Programming Interface". jul. 1999.

[Sybase 00] Sybase Inc. "Sybase EAServer, The high-performance engine for e-Business success". 2000

http://www.sybase.com/content/1011622/eas_brochure.pdf

[Sybase 99] Sybase Inc. "Understanding Continuous Availability". 1999

[Sybase] Sybase Inc. "Sybase EAServer".
http://www.sybase.com/products/applicationservers/easerver/

[Thomas 98] Anne Thomas. ENTERPRISE JAVABEANS TECHNOLOGY Server Component Model for the Java Platform.
http://java.sun.com/products/ejb/white/white_paper.html

[Tomcat] Tomcat. http://jakarta.apache.org

[Wolf 99] David Wolf, "Load Balancing and Failover Using Sybase Enterprise Application Server", Sybase Inc., 1999