Uma Implementação Tolerante a Faltas e Transparente da Plataforma J2EE

André Andrade Costa, Francisco Vilar Brasileiro*

Universidade Federal da Campina Grande
Coordenação de Pós-Graduação em Informática
Av. Aprígio Veloso, s/n, Bodocongó
58109-970, Campina Grande, PB, Brasil
Tel: (+55) 83 310 1123 Fax: (+55) 83 310 1124

andrec@infonet.com.br,fubica@dsc.ufcg.edu.br

Abstract. *The use of developing platforms to support the implementation of distributed applications has become a trend. These platforms provide a number of specialised services that help programmers to focous on the bussiness logic of the applications they develop, instead of wasting precious time with the implementation of infra-structure 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-fucntional requirement more and more necessary for distributed applications. Developers of such applications must themselves provide the necessary mechanisms to fulfil the requirements of the applications. Alternatively, they can use implementations of the platform that are themselves fault-tolerant. In this paper we present the design and implementation of such a platform. Unlike other implementations available, our implementation provides a solution that is highly reliable and totally transparent to the application.*

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 faltas, 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 faltas requeridos, ou usar implementações da plataforma que possuam características de tolerância a faltas. Neste artigo nós apresentamos o projeto e a implementação de um servidor de aplicações J2EE que implementa esses mecanismos. Diferentemente de outras soluções disponíveis, o nosso sistema provê alta confiabilidade de forma totalmente transparente para as aplicações.*

1 Introdução

Devido ao aumento da complexidade dos sistemas de informação, 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

*Este trabalho é parcialmente financiado pelo CNPq (processo 300.646/96).

torna-se bastante simplificado uma vez que os serviços mais complexos requeridos pelo sistema estão implementados no ambiente de execução. Atualmente, uma das mais promissoras tecnologias de componentes servidores é a especificação *Java 2, Enterprise Edition* (J2EE), criada pela SUN Microsystems em parceria com um número de empresas [4]. A especificação J2EE reúne outras especificações como *Java Servlets*, *Enterprise JavaBeans* (EJB), *Java Naming and Directory Interface* (JNDI), *Java Remote Method Invocation* (RMI), *Java Database Connectivity* (JDBC), *Java Transaction API* (JTA), *Java Transaction Service* (JTS), etc, para oferecer uma maneira transparente de desenvolvimento baseado em componentes servidores.

Entretanto, 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 para tolerância a faltas podem ser implementados de duas maneiras: pela própria aplicação, ou pelos serviços que compõem a plataforma J2EE. A primeira alternativa, além de tornar o desenvolvimento da aplicação mais complexo, exige que todas as aplicações incluam, normalmente de forma *ad hoc*, esses mecanismos ao seu código. A segunda alternativa é mais adequada, pois libera o desenvolvedor da tarefa de implementar os mecanismos para tolerância a faltas. Além disso, dependendo da concepção dos servidores que implementam os serviços, pode permitir que os mecanismos para tolerância a faltas executem de forma transparente para as aplicações.

Neste artigo nós apresentamos o projeto de um servidor J2EE no qual os mecanismos para tolerância a faltas são utilizados pelas aplicações de forma totalmente transparente. Diferentemente das outras soluções existentes, nossa abordagem utiliza replicação ativa. O modelo de replicação ativa implementado é totalmente transparente para o desenvolvedor, de forma que basta 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 qualquer "janela de vulnerabilidade". Por oferecer suporte a tolerância a faltas totalmente transparente, a plataforma também permite que aplicações legadas executem com maior robustez, sem qualquer modificação no código fonte original. A nossa implementação se baseia em software aberto (o servidor de aplicações JBoss[1] e a ferramenta de comunicação em grupo JChannel[2]) e se consitui em uma alternativa às soluções proprietárias existentes.

O restante desse artigo está organizado da seguinte forma. A Seção 2 apresenta de forma abreviada a plataforma J2EE, enfatizando algumas vunelrabilidades da mesma. Na Seção 3 são discutidos os mecanismos para tolerância a faltas usados em nosso projeto. A Seção 4 apresenta a nossa implementação (maiores detalhes sobre a implementação podem ser encontrados em [5]). A Seção 5 conclui o artigo com nossos comentários finais[3].

2 A Plataforma J2EE

A plataforma J2EE foi projetada para oferecer um padrão de desenvolvimento e um ambiente de execução de aplicações distribuídas baseadas em componentes. Através de um *Servidor de Aplicações* que implemente os serviços dessa plataforma, pode-se fazer uso de tecnologias como transações distribuídas e objetos remotos de maneira simples e transparente.

A seguir discutiremos alguns dos principais serviços oferecidos pela plataforma, colocando enfâse nas possíveis vulnerabilidades dos mesmos.

[1] Disponível em http://www.jboss.org/.

[2] Disponível em http://www.sourceforge.net/projects/javagroups/.

[3] Comparação com trabalhos relacionados e uma avaliação de desempenho detalhada podem ser encontradas na versão estendida desse artigo em http://www.lsd.dsc.ufpb.br/publications/JBossFT.ps.

2.1 *Java Servlets*

Para permitir que a aplicação possa ser utilizada através de *browsers* (clientes *Web*) via Internet ou em *intranets*, a plataforma J2EE inclui a especificação *Java Servlets* [1]. Esse serviço foi projetado para estender as funcionalidades de um servidor *Web* para que, além do conteúdo estático provido através de páginas no formato HTML (*HyperText Markup Language*), conteúdo dinâmico também pudesse ser oferecido. Um *servlet* pode ser definido como um componente *Web* gerenciado por um processo para gerar conteúdo dinâmico [1]; este processo é denominado *container*. O *servlet* consiste de uma classe Java que será carregada e executada a partir de invocações do servidor *Web*. Os *servlets* interagem com os *browsers* através de um protocolo de requisição/resposta baseado no HTTP (*Hypertext Transfer Protocol*) implementado pelo *container* do *servlet* [1].

O protocolo HTTP foi projetado para ser um protocolo sem estado [1]. Entretanto, muitas aplicações *Web* necessitam que uma seqüência de requisições originadas de um mesmo cliente possam ser associadas umas com as outras, por exemplo, quando um cliente deve ser autenticado antes de utilizar um serviço. Depois de o cliente informar a sua identificação e sua senha, a aplicação deve "lembrar" que toda requisição proveniente daquele *browser* está associada ao cliente que foi autenticado anteriormente. Assim, a aplicação deve manter um estado (sessão) em nome do cliente. A especificação *Java Servlet* define que o *container* deve implementar algum mecanismo de acompanhamento da sessão do cliente de forma que isso fique transparente para o desenvolvedor do *servlet* [1]. Através de uma interface única, o *servlet* pode salvar objetos na sessão associando-os com um nome para que o mesmo *servlet* ou um outro em execução no mesmo *container* possa recuperar esse objeto para utilizar em seu processamento.

Para garantir um nível maior de disponibilidade das aplicações *Web*, faz-se necessário que os *servlets* passem a estar em execução em mais de uma instância do servidor J2EE. 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 aquele tiver acesso às informações das sessões que estavam ativas antes da falha deste.

2.2 *Java Naming and Directory Interface*

Um serviço de grande importância na plataforma J2EE é o JNDI [7]. 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. Através desse serviço é possível registrar objetos com um determinado nome para que aplicações distribuídas pela rede possam ter acesso aos mesmos a partir desse nome [6]. Os serviços de nomes e diretórios desempenham um papel muito importante em *intranets* e na Internet uma vez que provêm compartilhamento de informações por toda a rede.

A utilização de apenas uma instância do servidor que implementa o serviço JNDI pode provocar a perda das informações do serviço. Portanto, para que faltas 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 permanecerão 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.

2.3 *Enterprise JavaBeans*

Outro serviço presente na plataforma J2EE é o EJB [8]. Nesse serviço, os componentes podem ser customizados sem a necessidade de alteração do código fonte, possibilitando que o desenvolvimento da aplicação se torne bastante simples. Os serviços providos pela infra-estrutura isentam

o desenvolvedor de lidar com aspectos mais complicados da implementação, como gerenciamento de transações, ciclo de vida de objetos, concorrência e segurança. Os componentes EJB ficam armazenados em um ou mais servidores e constituem o *framework* de serviços do desenvolvedor, podendo ser utilizado 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 mudar o comportamento do componente. O serviço EJB provê um *container* que oferece para os componentes servidores um contexto de execução, além de gerenciamento e controle de serviços [8]. Com o EJB é possível criar componentes persistentes que representam os dados (*entity beans*) e componentes que contêm apenas lógica de negócio (*session beans*). Dessa forma, os desenvolvedores das aplicações apenas têm de criar a interface com o usuário que passaria os dados do usuário para os *session beans* que manipulariam os *entity beans* quando necessário.

Para tornar o serviço EJB tolerante a faltas, é necessário que cada componente esteja hospedado em mais de um servidor de aplicação para que, em caso de falha de um dos servidores, uma réplica do componente hospedada em um servidor livre de faltas possa assumir o processamento. Uma vez que o componente EJB caracteriza-se por manter um estado interno ao longo do seu ciclo de vida [7], é preciso que os estados das réplicas de um componente estejam sincronizados para que o fluxo do processamento possa continuar de forma consistente.

3 Projeto de uma Plataforma J2EE Tolerante a Faltas e Transparente

A solução adotada teve como base incluir em um servidor J2EE de código aberto mecanismos de replicação ativa. Como cada requisição a um serviço realizada por uma aplicação cliente será executada 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 ou por nenhuma delas.

Os serviços da plataforma J2EE são utilizados pelas aplicações 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 mecanismo de comunicação em grupo possa ser utilizado de forma transparente pelas aplicações clientes, as bibliotecas clientes tiveram de ser alteradas para que enviem a requisição feita pelo cliente para o grupo de replicação e não mais para uma instância. Em cada instância do servidor J2EE deverá haver um serviço em execução, denominado *GroupProxy*, que será responsável por interceptar as requisições enviadas ao grupo e repassá-las ao servidor J2EE. Após a requisição ser processada pelo servidor, o *GroupProxy* devolverá a reposta da requisição para a aplicação cliente que fez a requisição. A biblioteca cliente, que permanece esperando a resposta de requisição, retorna a primeira resposta recebida para o cliente que fez a requisição. A Figura 1 ilustra de maneira geral a arquitetura da solução.

Cada serviço da plataforma J2EE oferece uma maneira específica de acesso. A seguir estão descritas as alterações necessárias para cada serviço.

3.1 *Java Servlet* Tolerante a Faltas

Como foi explicado anteriormente, os *servlets* são acessados através de *browsers*. Uma solução que implique na alteração do *browser* para que estes passem a enviar uma requisição para o grupo de replicação ao invés de enviar a requisição para um servidor específico não seria viável. Entretanto, como muitas organizações já possuem um servidor *Web* para o conteúdo estático, os servidores J2EE geralmente oferecem um *plug in* para o servidor *Web* para que este redirecione as

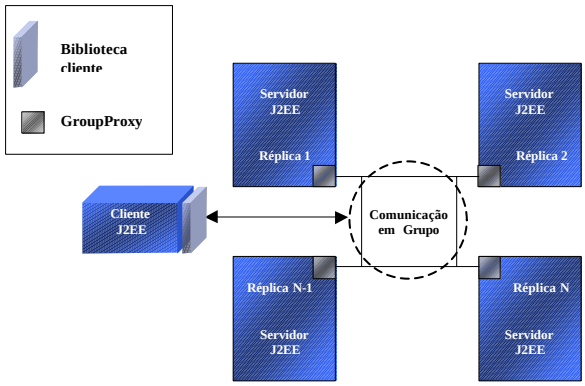

Figura 1: Arquitetura da solução

requisições ao conteúdo dinâmico para o *container* do *servlet*. Assim, este *plug in* teve de ser alterado para que, ao receber a requisição originada do *browser*, envie esta requisição para o grupo de replicação através do mecanismo de comunicação em grupo. O *plug in* será responsável também por filtrar as respostas geradas pelas réplicas para que apenas a primeira retorne ao *browser*.

3.2 JNDI Tolerante a Faltas

Um cliente JNDI acessa este serviço através de uma classe Java chamada *Context*, onde é especificado qual servidor que hospeda o serviço JNDI. Dessa forma, essa classe foi alterada para interagir com o mecanismo de comunicação em grupo de forma que, quando um cliente requisite a criação de um novo registro, a requisição seja 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 *Context* repassará ao cliente apenas a primeira resposta recebida, descartando as demais.

3.3 EJB Tolerante a Faltas

Um componente EJB hospedado em um servidor J2EE é utilizado pelas aplicações clientes através de um mecanismo de RMI. Na especificação RMI, qualquer objeto cujos métodos podem ser invocados de uma outra máquina virtual é chamado de *objeto remoto*. A localização física dos objetos remotos e dos clientes que os utilizam é irrelevante. 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 outra máquina virtual (possivelmente em outra máquina).

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 [3]. Um desses objetos auxiliares é o *stub* que age como uma *referência remota* para a 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 reposta para a invocação do método.

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 [1]. Esse registro consiste em associar um nome com a 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. 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.

A nossa abordagem para prover confiança no funcionamento para componentes EJBs consiste em implementar replicação ativa alterando o comportamento do serviço RMI. Sendo assim, para garantir que a replicação ativa será transparente tanto para o desenvolvedor da aplicação cliente quanto para os componentes EJBs que estarão fazendo uso do serviço RMI, o *stub* do objeto remoto deverá ser responsável por enviar a requisição para o grupo de instâncias do objeto remoto, utilizando o mecanismo de comunicação em grupo, e receber a primeira resposta da invocação, filtrando as demais.

4 JBossFT

Seguindo o projeto apresentado na seção anterior, foram implementados os mecanismos para prover replicação ativa para os serviços *Servlet*, JNDI e EJB. O servidor J2EE escolhido como base para implementação desses mecanismos foi o JBoss, esse servidor de aplicações é estruturado de tal forma que novos módulos podem ser adicionados sem que o seu código precise ser alterado. Basta que o novo módulo implemente algumas interfaces do *framework* do JBoss para que ele seja executado como mais um serviço desse servidor.

O mecanismo de comunicação em grupo adotado foi o JChannel, uma ferramenta implementada em Java que acompanha a biblioteca JavaGroups. Essa biblioteca oferece vários *padrões de projeto* para serem utilizados sobre qualquer ferramenta de comunicação em grupo [2]. Como o JavaGroups permite que a ferramenta possa ser desacoplada sem a necessidade de alteração da aplicação, outro mecanismo mais eficiente pode ser escolhido.

Por ser um serviço essencial para todo o projeto, o *GroupProxy* foi o primeiro a ser implementado. Ele consiste de um serviço em execução no JBoss aguardando as mensagens que estão sendo enviadas ao grupo de replicação. Através da mensagem recebida, o *GroupProxy* recupera a requisição da mensagem e a redireciona para o serviço apropriado da sua instância. Ele foi implementado utilizando um bloco básico fornecido pelo JavaGroups chamado *Listener* que notifica um determinado objeto quando alguma mensagem deve ser recebida pela réplica. Após o serviço processar a requisição, o *GroupProxy* envia a reposta do processamento para o integrante do grupo de replicação que originou a requisição.

Utilizando o JavaGroups, foi criado um novo bloco básico chamado *RequestDispatcher* que implementa um protocolo de requisição/resposta para ser usado nos outros componentes do projeto. Através dele, uma classe pode enviar uma mensagem que será recebida por todos os integrantes do grupo de replicação e receber apenas a primeira resposta referente à sua requisição. Esse bloco básico realiza a conexão com o grupo de replicação no momento de sua criação, e, quando recebe uma mensagem do seu cliente para ser enviada, ele adiciona um cabeçalho a esta mensagem contendo um identificador de requisição. Através das primitivas do JavaGroups, o *RequestDispatcher* envia essa mensagem como um *multicast* para o grupo ao qual ele pertence. Quando o *RequestDispatcher* 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. Utilizando o *RequestDispatcher*, o *plug in* que interage com o servidor *Web* foi alterado para enviar a requisição do *browser* para o grupo de replicação e não mais para uma instância.

Para facilitar a implementação do cliente JNDI (*Context*), foi implementado o bloco básico *GroupMethodCall* 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 *GroupMethodCall* monta uma mensagem e a envia utilizando *RequestDispatcher* 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 *GroupMethodCall* permitindo que a invocação ao método fosse replicada para todas as instâncias do objeto remoto no grupo de replicação.

A Figura 2 ilustra as modificações feitas no servidor de aplicação JBoss para implementar o JBossFT, que incorpora os mecanismos para tolerância a faltas descritos acima. Como pode-se perceber, o cliente J2EE continua utilizando as mesmas interfaces ao serviço JNDI (classe *Context*). Entretanto, essa classe foi alterada para que as requisições fossem direcionadas para o grupo por meio do JChannel. Em cada instância do JBossFT, o *GroupProxy* realiza a intercepatção das mensagens para serem processadas pelo serviço J2EE implementado pelo JBoss.

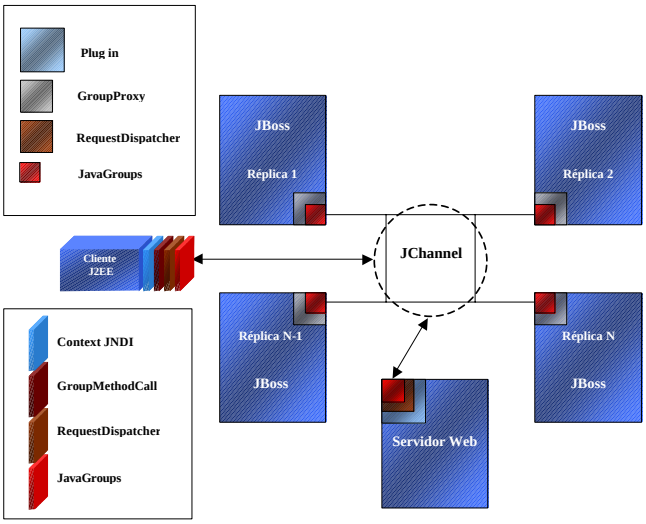

Figura 2: Arquitetura do JBossFT

Já a implentação de tolerância a faltas dos componentes EJB se baseia na implementação de um mecanismo transparente de invocação remota de método em um grupo de réplicas (*Group Method Invocation*, ou GMI). Para entender seu funcionamento é necessário, primeiro, entender como um objeto remoto não replicado é implementado.

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. No exemplo mostrado na Figura 3, está definida a interface remota para um objeto remoto que irá somar dois valores.

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

Figura 3: Interface de um objeto remoto

Definida a interface remota, o desenvolvedor deve criar uma classe que implemente essa interface (provendo a implementação dos métodos nela definidos). Essa classe normalmente herda de outras classes pré-definidas (ex. *java.rmi.server.UnicastRemoteObject*) algumas funcionalidades básicas do serviço RMI. A Figura 4 mostra a implementação do objeto remoto cuja interface foi definida na Figura 3.

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

Figura 4: Implementação de um objeto remoto

Quando o objeto cliente deseja utilizar um objeto remoto, ele realiza uma consulta ao serviço de nomes para obter o *stub* desse objeto, como indicado na Figura 5.

```
public class Cliente {
    public static void main(String[] args) throws Exception {
        // Conexão com o serviço JNDI
        Context ctx = new InitialContext();
        // Obtendo uma referência remota para o objeto Calculador
        Calculador objetoRemoto = (Calculador) ctx.lookup("Calculador");
        // Invocando o método remoto
        int resultado = objetoRemoto.adicionar(2, 2);
    }
}
```

Figura 5: Acesso a um objeto remoto

Deve-se destacar que o objeto cliente não sabe qual é a classe do *stub*, e sim que ele implementa a mesma interface remota implementada pelo objeto remoto. Essa característica permitiu que na nossa implementação fosse possível retornar um *stub* de uma outra classe, no momento que o objeto cliente consulta o serviço de nomes. Essa outra classe, a *GroupStub*, é responsável por interagir com o mecanismo de comunicação em grupo para replicar as invocações de método[4]. A classe do *GroupStub* é criada dinamicamente no momento em que é feita a consulta ao serviço JNDI.

Foi necessário implementar um serviço adicional que irá funcionar em conjunto com o serviço JNDI em cada instância do servidor de aplicação. Esse serviço, o *JNDIProxy*, intercepta as consultas para detectar quando o resultado da consulta é um *stub* de um objeto remoto. Caso a consulta seja a um objeto remoto, o *JNDIProxy* descobre qual a interface remota que o *stub* implementa através da API *Java Reflection*, que permite obter meta-informações (nome da classe, métodos, etc) de objetos [3]. Após identificar a interface remota do *stub*, o *JNDIProxy* cria a classe do *GroupStub* para o 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

[4]Esta classe herda boa parte de suas funcionalidades da classe *GroupMethodCall* discutida anteriormente.

essa instância irá implementar. Dessa forma, o *JNDIProxy* irá devolver um *GroupStub* 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.

Também nesse caso, o serviço *GroupProxy* em execução 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 *GroupProxy* devolve a reposta da invocação para o *GroupStub* do objeto cliente que fez a invocação. O *GroupStub*, que permanece esperando a resposta do método, retorna a primeira resposta recebida para o objeto cliente que realizou a invocação.

5 Conclusão

Esse artigo apresentou uma solução para desenvolver aplicações J2EE com requisitos de confiança no funcionamento onde os mecanismos para tolerar faltas estão implementados no servidor de aplicações. A nossa proposta teve como base implementar mecanismos de replicação ativa em um servidor J2EE de código aberto que provê 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 dois novos serviços (*GroupProxy* e *JNDIProxy*) fossem adicionados 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.

Referências

[1] Java Servlet specification, version 2.2. SUN Microsystems, 1999.

[2] BAN, B. JavaGroups - group communication patterns in Java, 1998. http://www.cs.cornell.edu/home/bba/Patterns.ps.gz.

[3] CAMPIONE, M., WALRATH, K., AND HUML, A. *The Java Tutorial*. Addison-Wesley, 2000.

[4] CATTEL, R., AND INSCORE, J. *Criando aplicações comerciais com a plataforma Java 2, Enterprise Edition*. Editora Campus, 2001.

[5] COSTA, A. A. Usando replicação ativa para prover tolerância a falhas de forma transparente a uma implemetação da plataforma J2EE. Dissertação de mestrado, COPIN - Universidade Federal da Paraíba, Campina Grande, dezembro 2002.

[6] KASSEM, N., AND (EDITORS), E. T. *Designing Enterprise Applications with the Java 2 Platform*. Addison-Wesley, 2000.

[7] ROMAN, E. *Mastering Enterprise JavaBeans and the Java 2 Platform*. John Wiley & Sons, 1999.

[8] THOMAS, A. Enterprise JavaBeans technology server: Component model for the Java Platform. Patricia Seybold Group, 1998. http://java.sun.com/products/ejb/white/white_paper.html.