PROPOSTA DE UM SISTEMA DE BANCO DE DADOS DISTRIBUÍDO E REPLICADO UTILIZANDO SERVIÇO RMI-JAVA

André Luis Castro de Freitas[1]

dmtalcf@furg.br

[1] FURG – Fundação Universidade Federal do Rio Grande

Rio Grande – RS - BRASIL

RESUMO

Os sistemas de bancos de dados são projetados para gerenciar grandes grupos de informações. Portanto, o gerenciamento dos dados envolve a definição de estruturas para armazenamento de informação e o fornecimento de mecanismos para manipulá-las. Analisando sob esta ótica, verifica-se a existência de uma série de modelos disponíveis onde o modelo relacional tradicional, com arquitetura cliente/servidor, ocupa um alto grau de utilização. Modelos avançados como os bancos de dados distribuídos ainda representam área de interesse e estudo. Este trabalho tem como proposta a definição e implementação de um banco de dados distribuído replicado que utilize várias bases de dados relacionais idênticas em diferentes servidores. A conversação, via rede, entre as estruturas clientes e os diversos servidores que manipulam as bases de dados foi construída utilizando o serviço RMI - *Remote Method Invocation,* disponível na linguagem de programação Java. RMI permite a troca de informações, entre máquinas virtuais Java, através da execução de métodos em objetos remotos, de tal forma, que cliente e servidor possam se comunicar através de um protocolo padrão.

Palavras-chave: Sistema de Banco de Dados Distribuído, Linguagem de Programação Java, RMI – Invocação de Métodos Remotos.

ABSTRACT

The databases systems are designed for manage great groups of informations. Therefore, the administration of the data involves the definition of structures for storage of information and the supply of mechanisms to manipulate them. Analyzing in this way, the existence of a series of available models is verified where the relational model, with client/server architecture, occupies a high use degree. Advanced models as the distributed databases they still represent area of interest and study. This work has as proposal the definition and implementation of a distributed and replied database that uses several identical bases of data in different servers. The conversation, through net, among the structures client and the several servers that manipulate the databases was built using the RMI (Remote Method Invocation) service, available in the Java programming language. RMI allows the change of information, among Java virtual machines, through the execution of methods in remote objects, in such a way, that client and server can communicate through a standard protocol.

Keywords: Distributed Database Systems, Java Programing Language, RMI – Remote Method Invocation.

1 Introdução

Uma base de dados distribuída caracteriza-se como uma coleção de diversas bases de dados interligadas logicamente através de uma rede de computadores (nós). O sistema de software que possibilita a gerência desta base de dados distribuída, tornando a distribuição transparente para o cliente, denomina-se sistema de bases de dados distribuída. Portanto, um sistema de banco de dados distribuído representa a combinação das bases com o sistema, ou seja, consiste de uma coleção de nós cada qual podendo participar na execução de transações que fazem acesso a dados em um ou diversos nós (ÖZSU, 1999).

Considerando o projeto de um sistema de banco de dados distribuído é importante ressaltar alguns princípios básicos (KORTH, 1995):

- duplicação – o sistema mantém diversas réplicas idênticas da relação. Cada réplica é armazenada em um nó diferente resultando na duplicação de dados;
- fragmentação – a relação é dividida em diversos fragmentos sendo que cada fragmento é armazenado em um nó diferente;
- duplicação e fragmentação – esta é uma combinação das duas noções anteriores. A relação é dividida em diversos fragmentos e o sistema armazena diversas réplicas idênticas de cada fragmento.

Atualmente existem alguns bancos de dados distribuídos comerciais que apresentam estas características. Entre eles pode-se citar o *Oracle* 9i, o qual apresenta fragmentação horizontal primária de tabelas e índices. Essa versão demonstra total transparência na manipulação dos dados. Um segundo exemplo é o SQL *Server* 2000 cuja redução de fragmentos horizontais ainda é limitada. Por fim o IBM *Informix Dynamic Server* que utiliza também fragmentação primária através de um servidor cuja responsabilidade é estender a distribuição. Mas todos esses bancos de dados são softwares proprietários e apresentam grande complexidade de desenvolvimento e alto custo de manutenção.

Com o objetivo de atenuar a complexidade de desenvolvimento do sistema proposto optou-se pela utilização de réplicas, considerando os nós disponíveis, não levando em consideração a possibilidade de fragmentação. Acredita-se que a utilização de réplicas, apesar de resultar na duplicação de informações, contribua positivamente com o paralelismo, quando a

exigência maior do sistema for para leituras e com a disponibilidade das informações pois o sistema pode continuar o processamento apesar da identificação de falhas em alguns nós.

Quando se trata um banco de dados distribuído replicado torna-se complicado assegurar a atomicidade de uma transação, uma vez que diversos nós podem estar participando de sua execução, em se tratando de atualizações. Portanto, a falha de um destes nós, ou a falha de uma linha de comunicação que conecte estes nós, deve resultar em ações que assegurem a confiabilidade na atomicidade das informações (CERI, 1984).

Considerando a afirmação anterior, verifica-se a necessidade de gerenciar a execução das transações que fazem acesso aos dados armazenados em um nó local. É importante ressaltar que cada transação pode ser tanto uma transação local como parte de uma transação global. Partindo deste pressuposto criou-se um serviço, disponível em cada nó, implementado a partir da linguagem Java, utilizando o mecanismo RMI – *Remote Method Invocation*.

RMI disponibiliza uma família de objetos colaborativos localizados em qualquer lugar permitindo a comunicação através de protocolos padrão através da rede. O método permite a comunicação entre máquinas virtuais Java executadas em computadores distintos (NIEMEYER, 1997).

Portanto, o objetivo deste trabalho é utilizar o modelo RMI de comunicações entre aplicações Java para gerenciar e coordenar as transações de um sistema de banco de dados distribuído e replicado. Conforme a figura 1, cada nó contém uma cópia do banco de dados que permanece conectado ao serviço RMI. O cliente acessa o serviço e estabelece a conexão com um dos nós, em caso de consultas, ou com todos os nós, em caso de atualizações.

O cliente tenta conexão com todos os nós disponíveis, através de várias linhas de execução diferentes. Em caso de consultas, a linha de execução que retornar primeiro fornece as informações solicitadas, cancelando as outras. Em caso de atualizações todas as linhas de execução devem ser finalizadas e verificadas para manter a atomicidade das informações.

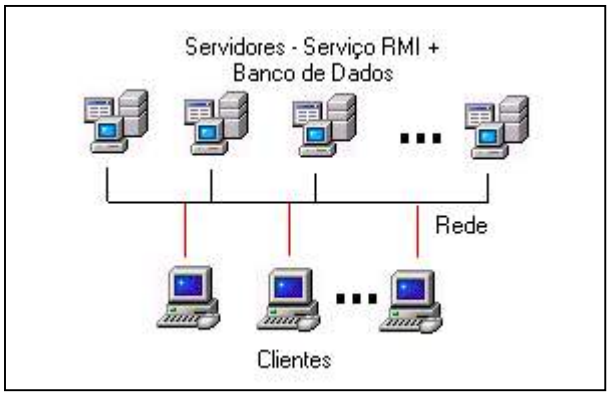

Figura 1: Sistema de Banco de Dados Distribuído Replicado / RMI-Java

Este trabalho está organizado nas seções 2, 3 e 4. A seção 2 define e caracteriza o modelo e o ambiente da estrutura proposta. Na seção 3 é demonstrada a arquitetura do sistema, onde são verificados os lados do servidor e do cliente identificando as estruturas de conexão e controle. Por fim, na seção 4 faz-se à demonstração de um exemplo utilizando a estrutura proposta. Após procede-se a conclusão e trabalhos futuros.

2 Modelo e Ambiente

O mecanismo RMI permite o acesso a um objeto em uma máquina diferente o qual é chamado de objeto remoto. Quando o cliente quer executar um método remoto ele simplesmente invoca um método normal da linguagem Java que será encapsulado em um objeto substituto chamado *stub*. O *stub* empacota, como um bloco de bytes, os parâmetros utilizados no método remoto. O *stub*, então, envia estas informações para o servidor. No lado do servidor, um objeto receptor separa os parâmetros, localiza o objeto chamado, invoca o método e, por fim, envia um pacote composto dos dados de retorno para o cliente. O *stub* cliente separa o valor de retorno (HORSTMANN, 2001). Portanto, através do mecanismo de RMI é feita à conexão entre o cliente e os nós disponíveis.

A conexão local entre o banco de dados e o serviço RMI é feita através da API *Java DataBase Connectivity* (JDBC). Utilizando JDBC é possível fazer acesso a um banco de dados independentemente do verdadeiro mecanismo de banco de dados que está sendo utilizado para armazenamento das informações (REESE, 2001).

O conjunto de classes que implementam as interfaces JDBC para um determinado mecanismo de banco de dados é chamado *driver* JDBC. A figura, a seguir, demonstra o modelo caracterizando, justamente, a existência de uma estrutura RMI para cada nó que disponibiliza o serviço de acesso ao banco de dados. Cada serviço conecta-se, localmente, a cada cópia do banco via um *driver* JDBC.

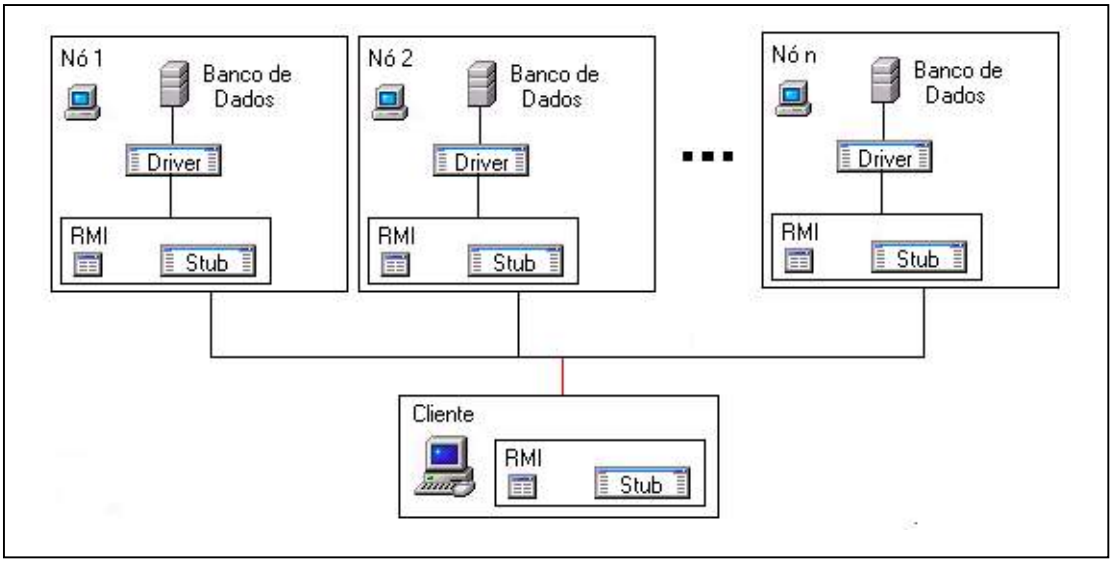

Figura 2: Modelo Interno do Sistema de Banco de Dados Distribuído Replicado

O cliente, por sua vez, também dispõe de um serviço RMI para acesso a objetos remotos. Conseqüentemente, o cliente via RMI faz conexão com os nós para consulta e/ou atualização das informações.

Com relação ao ambiente, o trabalho está sendo efetuado com testes sobre dois bancos de dados: MySQL e PostgreSQL, utilizando plataformas Windows e Linux. O objetivo é que cada nó possua uma configuração diferenciada, ou seja, o nó 1, por exemplo, contenha plataforma Windows com MySQL, o nó 2 plataforma Windows com PostgreSQL, o nó 3 plataforma Linux com MySQL e assim sucessivamente. Tem-se por meta avaliar, futuramente, a performance, velocidade e segurança destes nós, em se tratando de que estes apresentam características bem diferenciadas.

3 Arquitetura

A arquitetura de uma aplicação focaliza o processamento do sistema de particionamento, ou seja, a arquitetura decide em que máquina e em que espaço de processo um

determinado código deve executar (REESE, 2001). Qualquer aplicação de banco de dados é uma aplicação cliente/servidor se ela manipular armazenamento e recuperação de dados no processo de banco de dados e manipulação e apresentação de dados em qualquer outro lugar. O objetivo desta proposta é utilizar uma arquitetura de três camadas (arquitetura que utiliza um servidor de aplicação entre cliente e bando de dados) com vários servidores conectados a bancos de dados idênticos (replicados). Para que esta estrutura funcione corretamente a seguir serão demonstradas algumas características estabelecidas no lado do cliente e do servidor.

3.1 Cliente

A camada de apresentação, a qual caracteriza o cliente, deve fornecer uma interface de usuário para utilização do sistema. Conforme as informações solicitadas pelo usuário, o sistema ativa várias linhas de execução (*threads*), no objetivo de consultar ou atualizar os diversos servidores. Através do envio da mensagem *start* a aplicação ativa cada *thread* para uma lista de servidores disponíveis pré-estabelecidos.

```
private static String[] servidores = {"200.132.78.158","200.132.78.18","200.132.78.19"};
```

Cada linha é colocada em execução através do método *run* cuja primeira solicitação é a execução do comando *PING*. O comando *PING* tem por função verificar se o servidor solicitado encontra-se ativo ou não. Caso o servidor solicitado encontre-se ativo é feita à tentativa para conexão com o serviço remoto. O primeiro passo é a localização do serviço através da mensagem *lookup* identificando um serviço RMI de nome *DataBase*. Por fim, se a conexão for bem sucedida é enviada a mensagem *executaComando* passando como parâmetro o comando SQL requerido. Neste momento o controle operacional está de posse do servidor. É importante salientar que a ativação das *threads* é feita para todos os servidores disponíveis.

No caso de um comando SQL do tipo *SELECT* a primeira *thread* que atender a requisição do usuário tem por objetivo prover o cancelamento das outras que ainda estão tentando localizar a informação. No caso de comandos de atualização como *INSERT*, *UPDATE* ou *DELETE* cada *thread* deve terminar naturalmente.

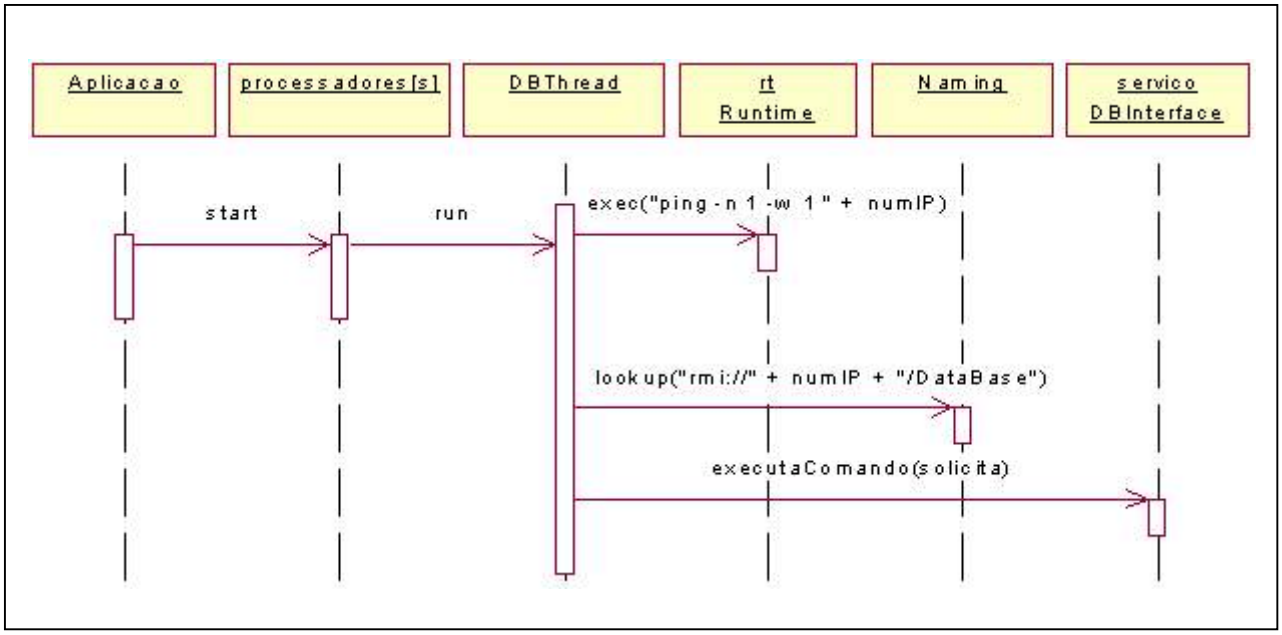

Figura 3: Conexão Cliente - Serviço

Para os comandos de atualização são verificados quais os serviços que não estão disponíveis. Para cada serviço ativo será enviado o comando SQL a ser executado e o número IP da máquina que não oferece, temporariamente, o serviço. Estas informações são armazenadas no arquivo de *log* de cada servidor ativo.

A ativação das linhas de execução para envio das informações de falha ocorre de maneira semelhante às consultas. A única diferença no diagrama da figura 3 é a modificação do método *executaComando(solicita)* pelo método *atualizaLog(numIPLog, solicita).* Através do envio da mensagem *start* a aplicação, agora, ativa cada *thread* para uma lista de servidores que estão ativos no momento. Cada servidor recebe, portanto, a mensagem *atualizaLog* a qual é responsável pelo armazenamento destas informações no lado do servidor. Estas informações são armazenadas até o momento em que a máquina de número IP determinada voltar as suas atividades normais e permitir a atualização de sua base de dados.

3.2 Servidor

O lado servidor tem por responsabilidade gerenciar, fornecer e manipular as informações junto ao banco de dados. Os métodos remotos são definidos no lado do servidor sendo que o cliente só dispõe das interfaces dos mesmos. Portanto, os métodos *executaComando*

e *atualizaLog* citados anteriormente são implementados no lado do servidor. Para cada solicitação de comando SQL recebida pelo servidor este precisa acessar a base local disponível. O acesso é feito via API JDBC que promove a conexão de uma aplicação Java a uma base de dados.

A figura 4, a seguir, mostra a seqüência do envio de mensagens para o método *executaComando*. Para cada acesso realizado ao serviço a aplicação faz uma nova conexão à base de dados. O método *getConnection* tem por objetivo atualizar as informações a respeito da base de dados que será utilizada. É feita a leitura de um arquivo texto o qual possui informações a respeito do *driver*, *url*, *username* e *password*. Esta facilidade foi inserida no intuito de propor ao usuário a utilização de qualquer tecnologia de base de dados no lado servidor desde que este possua os *drivers* de gerenciamento para tal.

```
Properties props = new Properties();
String fileName = arquivo + ".properties";
FileInputStream in = new FileInputStream(fileName);
props.load(in);
String driver = props.getProperty("jdbc.driver");
String url = props.getProperty("jdbc.url");
String username = props.getProperty("jdbc.username");
String password = props.getProperty("jdbc.password");
```

Após a identificação das informações, quando uma classe *Driver*, reconhece sua URL, ela cria uma conexão de banco de dados utilizando as propriedades especificadas. Ela fornece a classe *DriverManager* uma implementação *java.sql.Connection* representando esta conexão de banco de dados. *DriverManager* retorna este objeto *Connection* para a implementação.

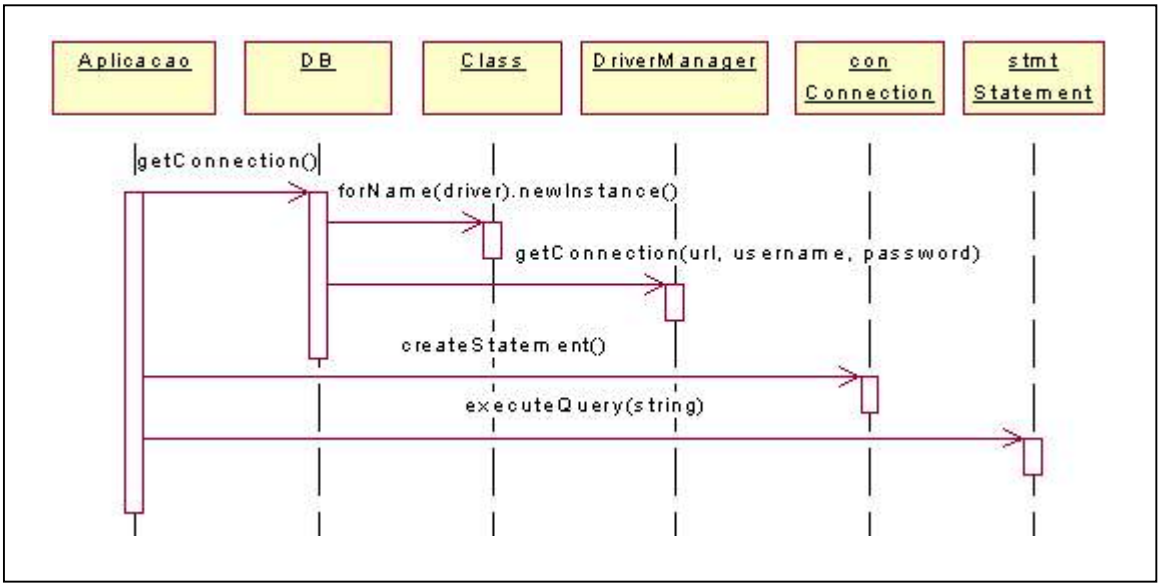

Figura 4: Conexão Servidor

A partir do objeto *Connection* é criado um conjunto de procedimentos através de *createStatement* para que se possa fazer o direcionamento do comando SQL desejado. Por fim, o método *executeQuery* é responsável pela execução do comando desejado. É importante observar que para comandos do tipo *SELECT* o método retornará informações que serão encaminhadas ao lado cliente via serviço RMI. Para comandos de atualização que não geram informações de retorno para visualização, faz-se o retorno do status para gerenciamento das atividades.

O lado servidor comporta-se como uma aplicação com uma base de dados local. Caso a base de dados não esteja disponível no momento, pelo fato do banco de dados não estar ativo, o serviço RMI deste servidor mantém um arquivo de *log* com as operações de atualização. Toda vez que for proposta alguma verificação ou atualização de informações no banco e existirem informações no arquivo de *log* este é automaticamente atualizado.

3.3 Falha de Serviço

O grande problema a ser administrado pelo sistema é quando um determinado serviço não está disponível e comandos SQL de atualização são solicitados. Conforme mencionado na seção 3.1 o cliente responsabiliza-se em detectar quais serviços não estão disponíveis enviando para os servidores ativos informações sobre os IPs com problemas bem como informando, também, os respectivos comandos SQL. Esta parte já está implementada e atende as necessidades previstas para o perfeito funcionamento do sistema.

Em fase de implementação está, portanto, a atualização das informações quando um determinado serviço retorna ao perfeito funcionamento. É importante salientar que este servidor não pode prover informações até que venha a ser totalmente atualizado. A idéia é que o serviço ao retornar a funcionar envie solicitações aos outros servidores verificando se estes outros detém comandos em seus arquivos de *logs*. O pior caso é quando todos os outros servidores não fornecem o serviço. Neste caso o sistema deve interromper as suas atividades de atualização, fornecendo, apenas, informações sobre consultas, apresentando ao usuário que está atualizado até uma determinada data. Caso contrário cada servidor disponível retornará as informações que detém para atualizar a base em questão.

Uma outra questão que vem a acontecer é a existência de duplicidades de informações nos arquivos de *logs* disponíveis. O gerenciamento desta etapa propõe o armazenamento no lado do servidor dos comandos já atualizados, com respectivas datas e horários de solicitação. Caso um servidor forneça as informações e outro esteja fora de serviço, e também contenha as informações, em uma próxima atualização estes comandos já estarão atualizados e serão descartados.

Este estudo, sobre atualizações durante falhas, apresenta uma complexidade inerente e ainda encontra-se em fase de implementação e testes.

4 Estudo de Caso

Esta seção tem por finalidade mostrar, resumidamente, um pequeno conjunto de interfaces que utilizam os serviços distribuídos demonstrados nas seções anteriores. A base de dados apresenta duas tabelas uma de autores e a outra de livros que se relacionam através do código do autor.

Figura 5: Janela Principal – Lado Cliente

A interface principal, conforme figura 5, demonstra as opções disponíveis para o sistema. É importante salientar que o cliente da aplicação não percebe a estrutura de serviços disponíveis que operam o sistema.

As interfaces de consulta conforme demonstradas na figura 6, propõe a consulta de autores (A) e livros (B) existentes respectivamente. Conforme citado anteriormente quando é solicitada uma consulta o serviço que responder primeiro é o que fornecerá as informações necessárias.

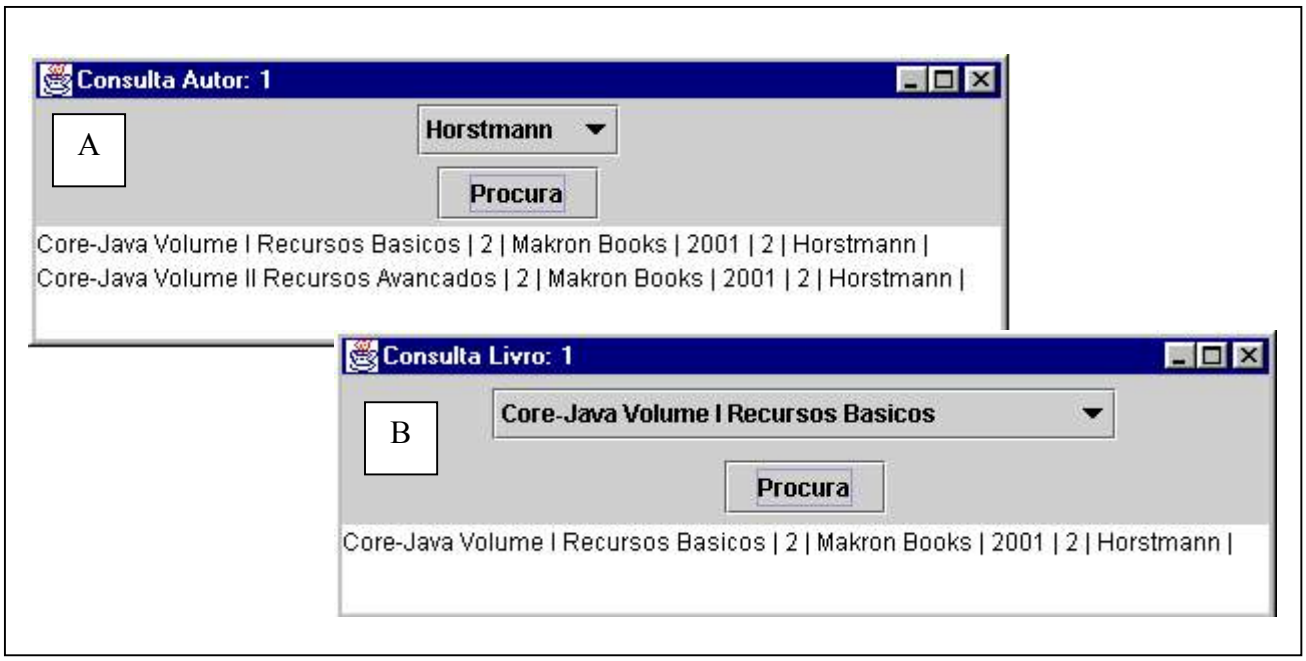

Figura 6: Consulta Autores – Livros

Por fim, apresenta-se na figura 7 as interfaces para atualização de autores (A) e livros (B). As opções disponíveis para o cliente são as mesmas de uma base de dados convencional e conforme já abordado é imperceptível à implementação dos serviços propostos.

Figura 7: Atualização de Autores - Livros

5 Conclusões

O objetivo deste trabalho foi demonstrar as características básicas de uma proposta para implementação de um sistema de banco de dados distribuído e replicado utilizando serviço RMI-Java.

Os mecanismos de suporte ao serviço RMI permitem a conexão entre diversas aplicações Java em equipamentos diferentes. As implementações propostas até o momento demonstram a eficácia e aplicabilidade da distribuição dos serviços remotos. Unindo, portanto, os serviços remotos aos serviços disponíveis para bancos de dados foram demonstrados, neste trabalho, o gerenciamento distribuído das atividades de consulta e atualização a bases de dados.

Preliminarmente, percebe-se que o estado atual do sistema atende aos requisitos no que se refere a consultas e atualizações. Com relação as atualizações, em caso de falha nos servidores, os trabalhos de implementação ainda encontram-se em testes no sentido de manter atualizadas todas as bases disponíveis.

Acredita-se que a validação da consistência lógica apresentada pelo sistema proposto será aumentada gradualmente com novos testes, portanto mais estudos de caso, envolvendo aplicações reais, estão sendo preparados para certificar o modelo.

Uma outra questão importante, ainda em avaliação, é a utilização das interfaces via *browse* de navegação *web*. Atualmente as interfaces são construídas a partir de aplicações java pois as *applets* apresentam limitações quanto à comunicação com servidores diferentes. Portanto, um estudo sobre comunicação via *applets* entre servidores está sendo realizado.

6 Trabalhos Futuros

Alguns problemas ainda devem ser estudados. Com relação ao crescimento da rede são raros os estudos de desempenho de um SGBD distribuído. Faz-se importante detalhar a rede de comunicação observando o comportamento de protocolos de comunicação e de algoritmos à medida que os sistemas crescem. A maioria dos trabalhos nesta área utiliza modelos

simplificados, cargas de trabalho artificiais, premissas conflitantes ou ainda consideram poucos algoritmos específicos.

Para que sejam plenamente alcançados os objetivos da replicação (disponibilidade e desempenho), é necessário que seja avaliada a integração dos três tipos de replicação que ocorrem: replicação dos dados, replicação do processamento e replicação da comunicação.

7 Referências Bibliográficas

CERI, S.; PELAGATII, G. **Distributed Database Systems - Principles and Systems**, MacGraw Hill, 1984.

ELMASRI, R; NAVATHE, S. **Fundamentals of Database Systems**, Benjamin/Cummings, 2ª. edição, 1994.

HORSTMANN, C; CORNELL, G. **Core Java 2: Volume I, Fundamentals**. [S.l.]: Prentice Hall, 2000.

HORSTMANN, C; CORNELL, G. **Core Java 2: Volume II, Advanced Features**. [S.l.]: Prentice Hall, 2001.

KORTH, H; SILBERSCHATZ, A. **Sistema de Banco de Dados**. Makron Books, São Paulo, 1995.

NIEMEYER, P; PECK, J. **Exploring Java**. [S.l.]: O'Reilly & Associates, 1997.

ÖZSU, M.T; VALDURIEZ, P. **Principles of Distributed Database Systems**, Prentice Hall, 2ª. edição, 1999.

REESE, G. **Programação para Banco de Dados. JDBC e Java**. Ed. Bercley, São Paulo, 2001.

Agradecimentos

Agradeço a FAPERGS – Fundação de Amparo a Pesquisa do Estado do Rio Grande do Sul por proporcionar apoio financeiro necessário para a realização deste trabalho.