REPLICAÇÃO DE ARQUIVOS COM JAVA

Rodrigo Werlang
Aluno pesquisador
Universidade Regional Integrada do Alto Uruguai e das Missões – URI – Campus Frederico Westphalen
e-mail: inf00536@pluton.urifw.tche.br
Áreas de interesse: tolerância a falhas, banco de dados, redes de computador.

Emerson Rogério de Oliveira Júnior
Professor orientador
Universidade Regional Integrada do Alto Uruguai e das Missões – URI – Campus Frederico Westphalen
e-mail: emerson@inf.uri.br, emerson@inf.ufrgs.br
Doutorando em Ciências da Computação PPGC/UFRGS
Áreas de interesse: tolerância a falhas, sistemas distribuídos, processamento paralelo e distribuído.

ABSTRACT

The objective of this work is to supply one more contribution to the scientific community presenting techniques that will be used in distributed systems that need file replications in real time guaranteeing the permanent availability of the system. As the replication model is based on Java, portability tests and performance were also accomplished in the Linux and Windows platforms, allowing the use of methods for multithreading to manage the connections. To accomplish the replication in real time, it was necessary to develop new classes for insert, removal and updating of bytes in files, once they didn't exist in Java, updating only the modified bytes and putting back dynamically the others.

Keywords: file replication, fault tolerance, distributed systems, primary/backup, availability, Java, bytes insertion, multi-platform.

1. INTRODUÇÃO

Nos dias atuais, com o crescimento desenfreado das tecnologias de comunicação, juntamente com o rápido avanço da informática, os limites impostos pelas distâncias físicas estão cada vez menores, o que permite que grandes e complexos sistemas sejam desenvolvidos e espalhados globalmente com interação total entre os nodos. Tais empreendimentos, logicamente, trazem inúmeros problemas do ponto de vista da segurança, consistência dos dados, rapidez no acesso as informações, disponibilidade do sistema e eficiência com que ele atende a suas especificações.

Em sistemas distribuídos, o emprego de mecanismos de replicação de dados pode ser dito praticamente indispensável para permitir a constante disponibilidade de acesso às informações e pode ser utilizado em pequenas aplicações até gigantescos bancos de dados geograficamente espalhados. Para tal, técnicas de tolerância a falhas resolverão problemas de disponibilidade e confiabilidade do sistema e mecanismos de replicação de arquivos, utilizando-se das técnicas de tolerância a falhas, podem garantir que se um determinado ponto do sistema global não puder dispor das informações desejadas, outro possa. Porém, um dos grandes problemas enfrentados é garantir a consistência dos dados replicados utilizando métodos de sincronização que sejam estáveis e confiáveis.

Para permitir que a replicação dos dados seja realizada transparentemente e recuperada de forma equivalente, o modelo cliente/servidor primário/*backup* [1] foi utilizado em conjunto com a linguagem Java devido à flexibilidade alcançada com esta união. Isto possibilita a utilização da replicação em qualquer plataforma operacional sem a necessidade de adaptações no tratamento das informações ou nos algoritmos empregados, o que diminui consideravelmente a possibilidade de falhas humanas [2], já que o mesmo *bytecode* é utilizado em qualquer plataforma e a Máquina Virtual Java (JVM – *Java Virtual Machine*) preocupa-se em adaptar-se às peculiares de cada plataforma [4].

Neste projeto, a utilização das técnicas citadas e o futuro próximo onde elas serão empregadas em grande escala com um custo cada vez menor fazem dele uma contribuição para chegarmos a maturidade destas tecnologias e conseguir, a partir daí, desenvolver técnicas e métodos cada vez mais aprofundados e tolerantes a falhas.

2. TRABALHOS RELACIONADOS

Mecanismos de sincronização são necessários ao utilizar tolerância a falhas em sistemas distribuídos com o objetivo de garantir a disponibilidade constante do sistema. Assim, como forma de construir rápida e facilmente aplicações deste porte, muitos autores sugerem o uso da programação orientada a grupos com o emprego de ferramentas que forneçam suporte a comunicação *multicast* [7].

Segundo Silva [7], classes desenvolvidas em Java, orientadas a grupos, podem auxiliar no desenvolvimento de sistemas que utilizam tais modelos em aplicações distribuídas.

Acompanhando os mecanismos de sincronização, a replicação vem sendo empregada com base em vários modelos.Um destes modelos, sugere que a replicação seja realizada em *clusters* (agrupamentos), onde os *clusters* são estações de trabalho e um ou mais servidores de arquivos em uma rede local. Visto que as réplicas são criadas e mantidas dinamicamente nos *clusters* que estiverem compartilhando determinados arquivos, um dos problemas enfrentados é localizar uma réplica válida e manter a consistência entre estas [8].

Tendo em vista a agilidade da replicação, portabilidade e atualização somente da parte do arquivo que foi alterada, bem como a atualização das réplicas no instante em que as alterações ocorrerem, o modelo pesquisado neste trabalho sugere a criação de uma classe – visto que esta não existe na linguagem Java – que permita a atualização dinâmica dos dados de um arquivo, ou seja, a atualização somente dos *bytes* modificados, realocando os demais de forma que estes fiquem dispostos de maneira correta, de acordo com a operação que foi efetuada (inserção, remoção).

3. MODELO DA REPLICAÇÃO DE ARQUIVOS

Tendo em vista que todo o sistema está embasado em conceitos multi-plataforma, todos os objetos do sistema rodam sob a linguagem Java, possibilitando a realização de testes nas plataformas Linux e Windows. Além disso, toda a comunicação é feita através de *sockets* [5], o que torna possível o desenvolvimento de clientes e mesmo servidores em qualquer linguagem que implemente este modelo de comunicação. Toda essa flexibilidade e portabilidade entre plataformas operacionais, no entanto, tem um preço: Java não gera código nativo, mas sim um código intermediário denominado *bytecode* [4], que é um código independente de plataforma que é lido e interpretado pela Máquina Virtual Java.

Qualquer aplicação Java é executada sob os olhos de uma máquina virtual, que faz a alocação de memória, coleta de lixo automática da memória, trata da segurança e dezenas de outras funções úteis com as quais o programador não precisa se preocupar durante o desenvolvimento de sistemas [6].

É comum imaginar que replicação de arquivos esteja associada a objetos distribuídos, porém, uma das principais diferenças entre objetos distribuídos e a replicação de arquivos deste trabalho é a necessidade de constantes atualizações para garantir a sincronização dos dados em todos os servidores que compõe o sistema ao utilizar a replicação. Isto trouxe a necessidade do desenvolvimento de classes não existentes na linguagem Java para inserção, remoção e atualização de *bytes* em arquivos de forma ágil e eficaz para não comprometer o desempenho do sistema mesmo com replicação em tempo real. Outro aspecto que deve ser considerado é que em objetos distribuídos, mesmo com a utilização de serialização para o armazenamento dos objetos (que então se tornam arquivos) estes sempre são atualizados de forma atômica, não divisível, ao passo que a replicação de arquivos somente atualiza os *bytes* alterados recolocando os demais, para que fiquem dispostos da maneira correta dentro do arquivo.

Pressupondo que um arquivo possua a frase: *Java é uma linguagem multiplataforma que gera bytecodes.*Ao escrever a informação *orientada a objetos* após *linguagem ...*

1. utilizando uma das classes já existentes na linguagem a frase ficaria assim: *Java é uma linguagem orientada a objetos gera bytecodes;*
2. utilizando a classe implementada a partir de uma já existente a frase ficaria assim: *Java é uma linguagem orientada a objetos multiplataforma que gera bytecodes.*

O algoritmo utilizado, além de realocar dinamicamente as informações em um arquivo, também avalia se é necessário realizar tal trabalho. Para que o processo de troca funcione corretamente, devem ser utilizados dois *buffers*, um para a leitura dos *bytes* da posição atual até o tamanho do *buffer* e outro para a leitura dos *bytes* a partir da posição onde o outro terminou. Além destes membros, o método necessita ainda de dois atributos que guardem a posição de gravação e de leitura no arquivo.

Visto que todos os sistemas atuais são multi-processados e/ou multi-programados, os objetos implementados utilizam-se de *threading* e gerenciadores de conexão. Os gerenciadores de conexão, em especial, são os responsáveis por cada conexão estabelecida com um servidor do sistema, ou seja, para cada nova conexão estabelecida, um novo gerenciador de conexão é criado. Isto permite ter inúmeros processos clientes acessando o servidor e cada qual ter um tratamento individual para suas requisições, conforme ilustrado na figura 3.1.

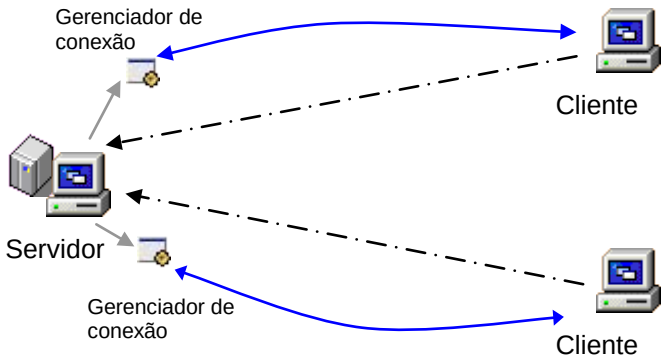


Figura 3.1: Gerenciador de conexões

Ao iniciar os objetos (tanto servidores quanto clientes), todos devem conhecer a lista de servidores disponíveis na rede de comunicação para ser possível realizar a sincronização dos arquivos, bem como realizar novas conexões a servidores *backup* se por acaso o servidor primário vier a falhar. Especificamente no momento em que o servidor primário é iniciado, este se conecta com todos os *backups* contidos em sua lista para saber quais estão ativos e quais não.

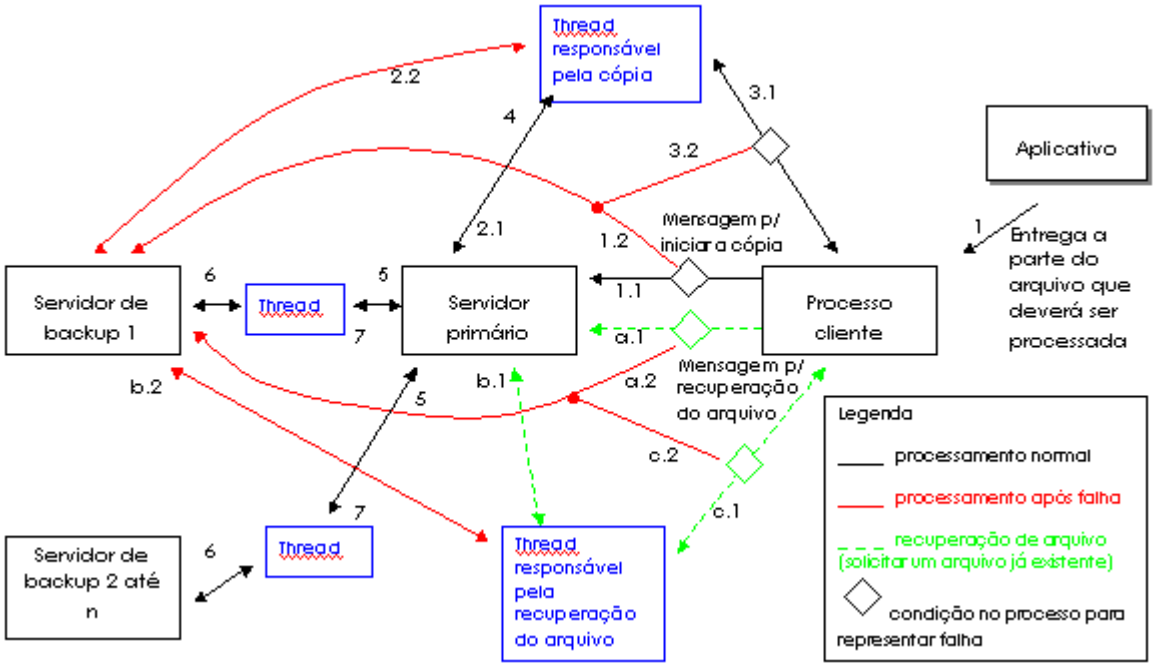


Figura 3.2: Diagrama lógico do sistema de replicação

Baseando-se na figura 3.2 e seguindo cada passo conforme descrito nas conexões entre os nodos que compõe o sistema, vejamos as operações realizadas por cada um deles em particular, passando primeiramente a situação onde o processo cliente atualiza um arquivo já aberto:

1. O *software* aplicativo entrega a parte do arquivo que deseja copiar ao servidor para a camada cliente, que ficará bloqueada durante todo o processo de atualização.
   1.1. Uma vez com os dados necessários, o processo cliente irá enviar uma mensagem ao servidor primário requisitando o início da transferência das informações.
   1.2. Se ocorrer falha, o primeiro servidor *backup* disponível será eleito como novo servidor primário e a mensagem será reenviada [1].
2. O servidor que se conectou com o cliente atende a requisição.
   2.1. É criado um processo contendo o gerenciador de conexão para realizar toda a comunicação com o *socket* cliente para que o servidor seja liberado e possa ficar aguardando novas requisições de outros clientes.
   2.2. Se a falha ocorrer no passo 1.1, no passo 1.2, o cliente conecta-se com o próximo servidor *backup* da lista e este então lança um novo gerenciador de conexão para realizar a comunicação com o cliente.
3. A *thread* denominada gerenciador de conexão inicia a comunicação:
   3.1. A *thread* e o cliente se comunicam e a transferência dos *bytes* modificados ocorre.
   3.2. Se neste momento ocorrer uma falha, o cliente ainda deverá ser capaz de desviar o processamento para o primeiro *backup* disponível, elegendo-o como novo servidor primário e o processo retornará ao ponto 1.2 para recomeçar [1].
4. Uma vez que o gerenciador de conexão – que é responsável pela comunicação entre o servidor e o cliente – tiver concluído seu processamento, ele irá entregar novamente o processamento ao servidor para que então se inicie o processo de sincronização dos *bytes* modificados para os arquivos contidos nos demais servidores *backup*.
5. Para isso, o processo servidor irá criar uma conexão para cada servidor e replicar as informações de maneira eficiente sem comprometer todo o sistema.
6. Cada conexão irá transferir a mesma parte do arquivo para cada um dos servidores *backup* disponíveis.
7. Terminado o trabalho, o servidor envia uma mensagem ao processo cliente que somente então é desbloqueado.

Este tipo de processamento descreve claramente um tratamento de comunicação síncrona entre os nodos do sistema. Neste tipo de situação, também conhecida como bloqueante, no

momento em que o processo emissor enviar uma mensagem ao processo receptor, o primeiro ficará bloqueado até que o receptor (que pode ser um processo remoto) receba e processe a informação recebida, liberando então o emissor para que este possa dar continuidade ao seu trabalho [3].

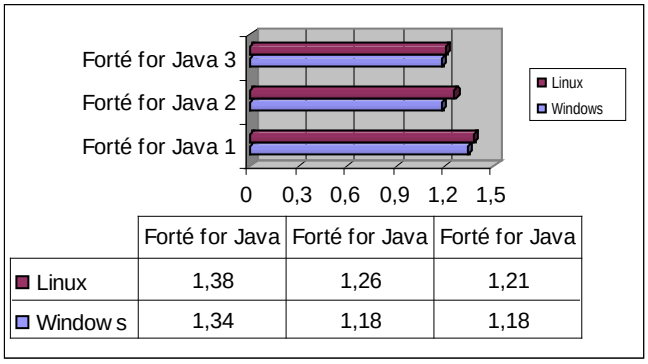
A seguir, o processo utilizado para abrir um arquivo guardado em um servidor:

a. O *software* aplicativo entrega à camada cliente a requisição para recuperar determinado arquivo para visualização, impressão, etc.
   a.1. Uma vez com os dados necessários, o processo cliente irá enviar uma mensagem ao servidor primário requisitando o início da transferência e este por sua vez replica a mesma para os demais servidores de forma que estes também abram o arquivo para futuras modificações.
   a.2. Se ocorrer falha, a mensagem deve ser reenviada ao primeiro servidor de *backup* contido na lista conhecida pela camada cliente.
b. O servidor que se conectou com o cliente atende a requisição:
   b.1. É criada uma *thread* para realizar toda a comunicação com o *socket* cliente para que o servidor seja liberado e possa ficar aguardando novas requisições de outros clientes.
   b.2. Se a falha ocorrer no passo a.1, no passo a.2, o processo cliente conecta-se com o próximo servidor *backup* da lista e este então lança um novo gerenciador de conexão para realizar a comunicação com o cliente.
c. O gerenciador de conexão inicia a comunicação:
   c.1. O cliente recebe o arquivo enviado pelo gerenciador de conexão.
   c.2. Se neste momento ocorrer uma falha, o cliente ainda deverá ser capaz de desviar o processamento para o primeiro *backup* disponível e o processo retornará ao ponto a.2 para recomeçar.

4. RESULTADOS OBTIDOS E CONCLUSÃO

Ao longo do desenvolvimento deste trabalho, vários testes foram realizados baseando-se sempre na linguagem Java, com o objetivo de saber sua capacidade em suportar sistemas baseados em tolerância a falhas, sua flexibilidade entre ambientes operacionais heterogêneos, sua portabilidade e velocidade. Para tal, foram utilizadas as plataformas Windows e Linux. Além disso, testes de performance e eficiência mostraram como a replicação de arquivos em tempo real se comporta com arquivos de tamanhos diferenciados.

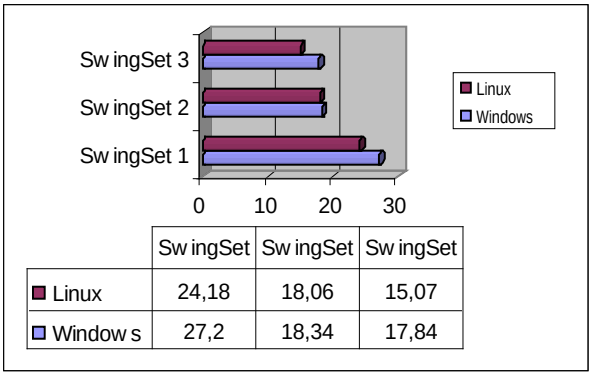
Para realizar os testes, foi utilizado um computador equipado com um processador Intel Pentium II Celeron 333 MHz, 96MB de memória principal, HD Quantum de 6 GB, placa de vídeo Trident Tgui9680 com 1MB de memória e as plataformas operacionais Windows 98 SE e Conectiva Linux 4.0 com ambiente gráfico KDE. A versão Java utilizada em ambos os sistemas operacionais foi a 1.2.2. No ambiente Windows, no entanto, o acelerador de performance *HotSpot Performance Engine v1.0.1* fornecido pela própria Sun Microsystems foi utilizado para comparar a velocidade em Windows com acelerador de performance e em Linux sem acelerador. Dois aplicativos – o **Swingset.jar** (que acompanha o produto) e o ambiente de desenvolvimento **Forté for Java CE Release Candidate 1** – escritos inteiramente em Java e que consomem recursos do sistema de forma constante, foram executados para medir o desempenho da linguagem.



	Forté for Java	Forté for Java	Forté for Java
Linux	1,38	1,26	1,21
Windows	1,34	1,18	1,18

Tempo medido em minutos (minutos, segundos)

Gráfico 4.1: Resultado do teste de performance do aplicativo Forte for Java



	SwingSet	SwingSet	SwingSet
Linux	24,18	18,06	15,07
Windows	27,2	18,34	17,84

Tempo medido em segundos (segundos, milésimos/seg)

Gráfico 4.2: Resultado do teste de performance do aplicativo SwingSet

Conforme os gráficos 4.1 e 4.2, chega-se a conclusão que a linguagem possui um desempenho muito semelhante nos ambientes onde foi testada. No entanto, deve-se levar em consideração que na plataforma Windows foi utilizado um acelerador de performance e na plataforma Linux não. Mesmo assim, o sistema operacional Linux obteve resultados muito próximos na execução do Forté for Java e superou o Windows na aplicação Swingset.jar . Outro dado de fator decisório é o tratamento de *multithreading* (multi-tarefa) nos dois sistemas operacionais, onde o Linux se mostrou mais eficiente que o Windows, porém, não obteve o mesmo resultado em operações de atualização de tela *(repaints)*, onde o Windows foi muito superior.

As análises feitas sobre a replicação de arquivos em tempo real, mostraram que o método responsável pelas atualizações é bastante veloz, o que demonstra que a técnica de realocação dos

bytes no arquivo é eficiente e que não deprecia o desempenho do sistema como um todo, permitindo que a replicação em tempo real seja utilizada de maneira praticamente imperceptível aos olhos do usuário. Um dos motivos da velocidade da atualização dos dados no arquivo é o protocolo de comunicação, que envia ao servidor somente a parte do *buffer* que foi modificada, fazendo com que a atualização sempre seja executada pela CPU local onde o servidor está rodando.

Dado o fato de que o algoritmo de atualização de *bytes* sempre os realoca dinamicamente no arquivo, o método mostrou que ao trabalhar com arquivos muito grandes e modificar *bytes* que estão em seu início, o processo demora mais porque os *bytes* posteriores ao ponto de atualização são colocados *n* posições para frente ou para traz – dependendo da modificação realizada.

Um fato é certo; a história da computação está caminhando a cada dia com passos mais largos e utilizando sistemas cada vez mais complexos que necessitam vitalmente de técnicas novas e aperfeiçoadas que permitem a eles estar disponíveis 24 horas por dia, 7 dias por semana e com certeza, mesmo num mundo de incertezas, a união entre Java e tolerância a falhas estará dia após dia mais presente na evolução desta nova era de sistemas distribuídos.

REFERÊNCIAS

[1] Jalote, Pankaj - Fault Tolerance in Distributed Systems. Englewood Cliffs, New Jersey, Ed. Prentice Hall, 1994.

[2] Weber, Taisy Silva et alii – Tolerância a Falhas - Conceitos e técnicas; Aplicações; Arquitetura de sistemas confiáveis. Porto Alegre, RS, Departamento de Informática Aplicada, Instituto de Informática, UFRGS, 1996.

[3] Kirner, Cláudio & Mendes, Sueli B.T. – Sistemas Operacionais Distribuídos; aspectos gerais e análise de sua estrutura. Rio de Janeiro, RJ, Ed. Campus, 1988.

[4] Kramer, Douglas et alii – The Java Platform; A WhitePaper. http://www.java.sun.com, Sun Microsystems, 1996.

[5] Sun Microsystems – "Custom Networking; All About Socket"; The Java Tutorial. http://www.java.sun.com/docs/books/tutorial/networking/sockets/.

[6] Gosling, James & McGilton, Henry – The Java Environment; A WhitePaper. http://www.java.sun.com/docs/white, Sun Microsystems, 1995.

[7] Silva, Robson Soares & Jansch-Pôrto, Ingrid – Implementação de Mecanismos de Sincronização em Java. http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana99/robson/robson.html.

[8] Sandhu, Harjinder S. & Zhou, Songnian – Cluster-Based File Replication in Large-Scale Distributed Systems. http://www.cs.yorku.ca/~hsandhu/papers/sigmetrics92.html.