UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

Projeto e Implementação de uma Ferramenta para Gerência de Servidores *Web* com Suporte a Recursos Espelhados

por

Raquel Meneses da Costa

Campina Grande, julho de 1998

UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Raquel Meneses da Costa

Projeto e Implementação de uma Ferramenta para Gerência de Servidores *Web* com Suporte a Recursos Espelhados

Dissertação apresentada ao curso de Mestrado em Informática da Universidade Federal da Paraíba, em cumprimento às exigências parciais para obtenção do grau de Mestre

Área de Concentração: Ciência da Computação
Sub-Área: Redes de Computadores e Sistemas Distribuídos
Orientador: Francisco Vilar Brasileiro

Campina Grande, julho de 1998

C837p Costa, Raquel Meneses da.
Projeto e implementação de uma ferramenta para gerência de servidores web com suporte a recursos espelhados / Raquel Meneses da Costa. - Campina Grande, 1998.
85 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1998.
"Orientação : Prof. Dr. Francisco Vilar Brasileiro".
Referências.

1. Web - Servidores. 2. Servidores - Gerência. 3. Ferramenta. 4. Dissertação - Informática. I. Brasileiro, Francisco Vilar. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.738.5(043)

PROJETO E IMPLEMENTAÇÃO DE UMA FERRAMENTA PARA GERÊNCIA DE SERVIDORES WEB COM SUPORTE A RECURSOS ESPELHADOS

RAQUEL MENESES DA COSTA

DISSERTAÇÃO APROVADA EM 21.07.1998

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador

PROF. MARCUS COSTA SAMPAIO, Dr.
Examinador

PROF. ROGÉRIO DRUMMOND B.P.DE MELLO FILHO, Ph.D
Examinador

CAMPINA GRANDE - PB

O temor do SENHOR é o princípio da sabedoria; bom entendimento têm todos os que cumprem os seus mandamentos; o seu louvor permanece para sempre.
Salmos 111:10

Agradecimentos

A Deus, em primeiro lugar, a quem dedico este trabalho, por ter-me ajudado e em quem encontrei tranqüilidade nos tantos momentos de dificuldades encontrados durante o desenvolvimento deste trabalho.

Aos meus pais, pela compreensão nos muitos momentos em que não estive disponível devido à falta de tempo.

Às minhas irmãs e, principalmente, a meu irmão, Rinaldo, pelas palavras de ânimo nos tantos momentos de dificuldade.

A Alessandro, pela sua paciência e compreensão.

Ao meu orientador, Prof. Francisco Vilar Brasileiro, por ter-me confiado esta responsabilidade, e a todos do Laboratório de Sistemas Distribuídos: Érica, Luiza, Claúdio, Tati, Felipe, principalmente a Tércio por ter dividido comigo as dificuldades.

Aos funcionários e professores do DSC e da COPIN, por estarem sempre prontos para ajudar.

A Lili da COPEX e a Gerluce do Gênesis, pelas palavras de apoio e incentivo.

Enfim, a todos que, direta ou indiretamente, contribuíram para a concretização deste trabalho.

Resumo

A forma pouco controlada que propicia o enorme crescimento apresentado pela *Web* nos últimos anos é responsável por alguns problemas enfrentados por seus usuários, como a quebra de ligação em um hipertexto. Para solucionar esse problema, esquemas de espelhamento de recursos têm sido propostos, de modo que a probabilidade de indisponibilidade dos recursos seja reduzida. Por outro lado, esses esquemas se preocupam apenas com o acesso para leitura, enquanto que as atualizações sobre os recursos não são tratadas.

Entretanto, ao utilizar a técnica de espelhamento de recursos, é necessário pensar em como manter a consistência dos recursos espelhados, uma vez que inconsistências podem ser geradas durante operações de atualização sobre os recursos espelhados. Nossa proposta apresenta um esquema de gerência de recursos *Web* espelhados que permite a utilização das estratégias de replicação otimista e pessimista para manter a consistência dos recursos *Web* espelhados.

Abstract

The World Wide Web huge growth on the past years is responsible for some problems faced by its users, the broken-link in a hipertext being one of them. Resource mirroring schemes have been proposed to solve this problem, by decreasing the propability of unavailability of the resources. However, this schemes solve only the problem of reading mirrored resources, the problem of updating mirrored resources consistently are not treated.

We propose a management scheme that uses optimistic and pessimistic strategies to maintain the mirrored resources consistency.

Índice

Agradecimentos ... *i*

Resumo ... *ii*

Abstract ... *iii*

Índice ... *iv*

Lista de Figuras ... *vii*

Capítulo 1: Introdução ... 1

1.1. Estrutura da Tese ... 3

Capítulo 2: Espelhando Recursos na *Web* ... 5

2.1. Introdução ... 5

2.2. Componentes da *Web* ... 6

2.3. Aspectos do Protocolo HTTP ... 6

2.4. Projeto de um Servidor *Web proxy* com Suporte a Recursos Espelhados ... 13

2.5. Atualização dos Recursos Espelhados ... 16

Capítulo 3: Mantendo a Consistência da Informação Replicada ... 19

3.1. Introdução ... 19

3.2. Protocolos de Consistência ... 20

3.2.1. Estratégias Pessimistas ... 21

3.2.2. Estratégias Otimistas ... 31

3.3. Discussão sobre as Abordagens ... 37

Capítulo 4: Projeto de um Suporte para Atualização de Recursos Espelhados na

Web 39

4.1. Introdução 39

4.2. Gerência de Recursos *Web* Espelhados 40

4.2.1. Modelo dos Recursos Web Espelhados 41

4.2.2. Características dos Pedidos de Atualização 42

4.2.3. Atualização dos Recursos Web Espelhados 43

4.2.3.1. Atualizações na Presença de Falhas 48

4.2.3.1.1. Atualizações Otimistas na Presença de Falhas 48

4.2.3.1.2. Atualizações Pessimistas na Presença de Falhas 53

4.3. Autenticação de Gerentes de Recursos 55

4.4. Projeto de uma Ferramenta de Atualização 57

Capítulo 5: Aspectos da Implementação 58

5.1. Introdução 58

5.2. Implementação do Esquema de Gerência de Recursos *Web* Espelhados 59

5.2.1. Ferramenta de Atualização 60

5.2.1.1. Bases de Dados do Resolvedor 62

5.2.1.1.1. Atualização das Base de Dados do Resolvedor 63

5.2.1.2. Módulo de Atualização Pessimista 64

5.2.1.3. Módulo de Atualização Otimista 66

5.2.1.4. Execução em Segundo Plano 68

5.2.2 Daemon 68

5.2.2.1. Base de Dados dos Recursos Espelhados 70

5.2.2.2. Módulo de Processamento de Atualização Pessimista 70

5.2.2.3. Módulo de Processamento de Atualização Otimista 72

5.3. Módulos de Recuperação 73

5.3.1. Módulo de Recuperação da Ferramenta 74

5.3.2. Módulo de Recuperação do Daemon 75

Capítulo 6: Conclusões 77

6.1. Contribuições 78

6.2. Direções para Trabalhos Futuros 79

6.3. Considerações Finais 80

Referências Bibliográficas 82

Lista de Figuras

Figura 1 - Interação entre um Cliente e um Servidor *Web* 6

Figura 2 - Cadeia de Comunicação com Componentes Intermediários 8

Figura 3 - Espelhamento de recursos com auxílio de um *proxy* 15

Figura 4 - Particionamento na Rede 17

Figura 5 - Conflito detectado por vetores de versão incompatíveis 34

Figura 6 - Arquitetura para Atualizações de Recursos Web Espelhados 41

Figura 7 - Atualização Atômica Finalizada 47

Figura 8 - Atualização Atômica Cancelada 48

Figura 9 - Conflito de Transações em Servidores Espelhados 49

Figura 10 - Exemplo de Colisão em Atualizações Concorrentes 51

Figura 11 - Arquitetura Implementada para Atualizações de Recursos *Web* Espelhados 60

Figura 12 - Extrato da base de dados do DNS 63

Figura 13 - Algoritmo do Módulo de Atualização Pessimista 65

Figura 14 - Algoritmo do Módulo de Atualização Otimista 67

Figura 15 - Estrutura de Processos do *Daemon* 69

Figura 16 - Algoritmo do Módulo de Processamento de Atualização Pessimista 71

Figura 17 - Algoritmo do Módulo de Processamento de Atualização Otimista 73

Capítulo 1

Introdução

A *World Wide Web*, ou simplesmente *Web*, surgiu com a finalidade de facilitar o acesso às informações e aos recursos da Internet. As tecnologias utilizadas para suportar a *Web* oferecem a infra-estrutura necessária para que usuários usufruam de ferramentas com interfaces gráficas altamente sofisticadas e fáceis de serem manipuladas. Essas ferramentas permitem que os usuários tenham acesso a uma grande quantidade de informações armazenadas em máquinas distribuídas por todo o mundo.

Ao longo dos últimos anos, a *Web* tem apresentado um crescimento muito acentuado; novos usuários vêm sendo introduzidos a cada dia com o intuito de usufruir de seus recursos. Só para se ter uma idéia, de junho de 1993 a janeiro de 1997 foram disponibilizados aproximadamente 600 mil *Web sites*. Por outro lado, esse rápido crescimento é responsável por alguns problemas enfrentados pelos seus usuários, dentre os quais, a quebra de ligações em um hipertexto é um dos mais evidentes [Ingham et al. 1996].

Uma quebra de ligação ocorre quando não é possível ter acesso a um dado recurso no endereço indicado. Elas são causadas por diversos fatores, dentre os quais podemos citar a falta de integridade referencial, que permite a remoção de recursos que ainda estão sendo referenciados, e a falta de transparência de migração, causada pelo fato da identificação de um recurso normalmente está associada a sua localização física. Em ambos os casos, a quebra de ligação causa muitos transtornos para os usuários dos

recursos, afetando diretamente a credibilidade dos fornecedores da informação.

Outra causa da quebra de ligação é a indisponibilidade de recursos, gerada por possíveis falhas na infra-estrutura computacional da *Web*. A falha de uma máquina, um particionamento[1] na rede de comunicação entre o usuário e o servidor ou até mesmo uma sobrecarga na rede podem tornar um recurso indisponível.

Para que essas falhas sejam toleradas, é necessário introduzir redundância no sistema [Jalote 1994]. No caso da *Web*, está técnica seria aplicada aos seus próprios recursos. A replicação de recursos em localidades diferentes, gerando caminhos alternativos aos já existentes, é também conhecida como espelhamento de recursos. A técnica de espelhamento de recursos foi utilizada em [Fonsêca 1998], onde foi desenvolvido um servidor *Web proxy* com suporte a recursos espelhados. Tal suporte resolve alguns dos principais problemas de quebra de ligação através da utilização de um esquema de endereçamento de recursos baseado em URNs (*Uniform Resource Names*) [Moats 1997]. Através da utilização desse esquema de endereçamento, é possível ter acesso de forma transparente aos recursos espelhados.

O trabalho desenvolvido por Fonsêca atende aos requisitos da maior parte dos recursos atualmente disponibilizados na *Web*, sobre os quais operações de leitura são realizadas numa proporção muito maior do que são realizadas operações de atualização. Entretanto, novas perspectivas de utilização da *Web* têm surgido a cada dia, onde operações de atualização são tão freqüentes quanto operações de leitura, e onde a manutenção da consistência da informação é crucial.

Assim, ao se utilizar a técnica de espelhamento de recursos para aumentar a disponibilidade dos mesmos, é necessário pensar em como manter a consistência dos

1Um particionamento ocorre quando uma falha na comunicação isola dois ou mais sites em partições individuais. Sites pertencentes a uma mesma partição podem se comunicar, entretanto, não existe comunicação entre sites pertencentes a partições distintas.

recursos espelhados, pois as várias instâncias dos recursos precisam estar sempre íntegras e consistentes. O trabalho desenvolvido por [Fonsêca 1998] não cobre esta funcionalidade, uma vez que ele se preocupou apenas com o problema de acesso a recursos para leitura.

Nosso objetivo neste trabalho é estender as funcionalidades do servidor *Web* com suporte a recursos espelhados apresentado em [Fonsêca 1998], fornecendo meios para que ele realize as atualizações dos recursos de forma transparente ao usuário, além de garantir a integridade e a consistência dos recursos espelhados. Garantimos também a autenticidade dos usuários nas atualizações e utilizamos um componente que é responsável pelas atualizações dos recursos espelhados nos servidores *Web*.

Nós ofereceremos um esquema para gerência de recursos *Web* espelhados, que permite a atribuição de dois tipos de semântica de consistência aos recursos. Desta forma, se um recurso possui semântica de consistência otimista, será permitido que atualizações sejam realizadas independente das falhas que possam ocorrer durante a atualização. Por outro lado, recursos com semântica de consistência pessimista só serão atualizados se todos os servidores que mantêm suas cópias estiverem disponíveis.

A principal contribuição deste trabalho é possibilitar a utilização dos recursos disponíveis na *Web* livre dos problemas de quebra de ligações e de possíveis inconsistências geradas por operações de atualizações sobre os recursos espelhados.

1.1. Estrutura da Tese

O restante deste trabalho está estruturado da seguinte forma. No capítulo 2 apresentaremos o modelo da *Web*, seus principais componentes e suas inter-relações; descreveremos também o projeto desenvolvido por Fonsêca, que visa exatamente resolver os problemas de indisponibilidade da informação através do espelhamento de seus recursos; além disso, discutiremos os aspectos relacionados à atualização dos recursos *Web* espelhados, onde apresentaremos de forma sucinta a nossa proposta.

No capítulo 3, serão discutidos os vários protocolos de controle de consistência

existentes e algumas abordagens utilizadas para manter a consistência de informações replicadas. O objetivo deste capítulo é oferecer um embasamento teórico dos vários aspectos de consistência existentes e como algumas abordagens se propõem a resolver as inconsistências.

A proposta de um esquema de gerência de recursos *Web* espelhados será apresentada no capítulo 4. Este esquema oferece um suporte para a manutenção de dois tipos de semânticas de consistência. Além disso, definiremos um esquema de autenticação de recursos para evitar que gerentes não autorizados atualizem determinados recursos espelhados. O capítulo 5, abordará aspectos de implementação de nosso esquema de gerência de recursos *Web* espelhados. Finalizando, no capítulo 6 apresentaremos nossas conclusões e discutiremos possíveis direções para trabalhos futuros.

Capítulo 2

Espelhando Recursos na *Web*

2.1. Introdução

A *Web* é composta por um conjunto de componentes que se comunicam utilizando o protocolo HTTP (*HiperText Transfer Protocol*) [Berners-Lee et al. 1996, Fielding et al. 1997]. Ela oferece recursos necessários para que seus usuários tenham acesso a uma grande quantidade de informações de forma fácil e rápida, criando, com isso, uma variedade de oportunidades para utilização de redes de computadores de abrangência mundial.

O objetivo deste capítulo é apresentar as principais tecnologias utilizadas para suportar a *Web* e alguns aspectos do protocolo HTTP, que serão importantes para o entendimento de nosso trabalho. Além disso, mostraremos alguns detalhes da proposta desenvolvida por Fonsêca [Fonsêca 1998] para fornecer suporte para o espelhamento de recursos, e com isso diminuir a probabilidade de indisponibilidade de recursos devido a falhas ou sobrecarga na infra-estrutura *Web*.

Na seção 2.2 discutiremos os principais componentes da *Web*. Na seção 2.3 apresentaremos os principais aspectos do protocolo HTTP. A seção 2.4 apresenta, de forma sucinta, o projeto do servidor *Web proxy* com suporte a recursos espelhados e, por fim, na seção 2.5 trataremos dos aspectos relacionados com a atualização dos recursos espelhados, que não são tratados em [Fonsêca 1998].

2.2. Componentes da *Web*

Os dois principais componentes da *Web* são o cliente, ou *Web browser*, e o servidor *Web*. O cliente é um programa que faz um pedido a um servidor remoto utilizando o protocolo HTTP, como mostra a Figura 1 abaixo. Ele nada mais é do que uma *interface* entre os documentos solicitados pelo usuário e o servidor. Existe uma grande variedade de *Web browsers*, muitos deles possuem diversas funcionalidades em comum, apesar de existirem algumas diferenças a nível de desempenho e suporte.

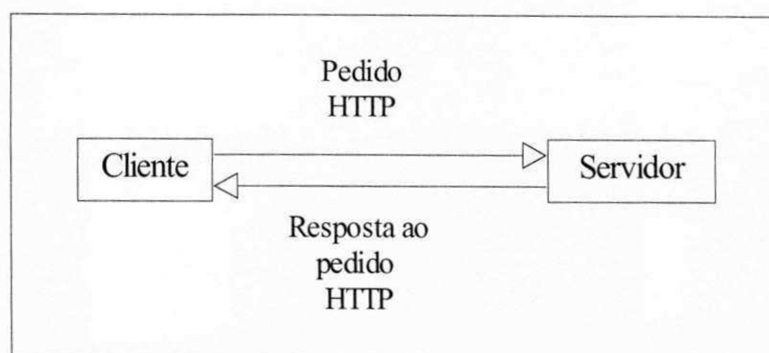


Figura 1 - Interação entre um Cliente e um Servidor *Web*

Por outro lado, os servidores são responsáveis por processar, fornecer e armazenar recursos (ex. um hipertexto, uma imagem, um formulário, etc.) que foram solicitados ou enviados pelo cliente. Além disso, eles podem registrar os endereços dos usuários que tiveram acesso aos recursos neles armazenados. A interação entre o cliente e o servidor ocorre da seguinte forma: inicialmente, o cliente solicita uma informação a um determinado servidor; ao receber o pedido, o servidor analisa sua base de dados para recuperar as informações solicitadas e as envia para o cliente; o cliente recebe as informações e as apresenta para o usuário.

2.3. Aspectos do Protocolo HTTP

O HTTP é um dos protocolos da camada de aplicação baseado em pedidos e respostas. Para cada pedido do cliente ao servidor é enviado uma mensagem de requisição, cujo formato é definido pelo protocolo HTTP. Esta mensagem inclui o método a ser aplicado ao recurso, o identificador do recurso, ou seja, sua URI (*Universal Resource Identifier)*, e a versão do protocolo em uso. A mensagem de

requisição pode conter ainda vários cabeçalhos que podem ser usados para passar parâmetros que modificam a execução de um método ou o formato das respostas do servidor. Alguns métodos manipulam recursos passados como parâmetro pelo cliente. Nesses casos, a mensagem de requisição pode conter também o recurso (a parte da mensagem que contém meta-informações sobre o recurso e o próprio recurso é denominada de entidade, na terminologia do HTTP).

Após receber e interpretar um pedido, o servidor envia mensagens de respostas para o cliente. Uma mensagem de resposta deve conter a versão do protocolo, seguido por um código numérico indicando o *status* da operação e sua correspondente explicação textual. As mensagens de respostas podem conter também cabeçalhos e entidades, que tanto podem representar um recurso solicitado quanto um erro de execução do método.

O código de *status* retornado em uma mensagem de resposta é um inteiro de três dígitos, onde o primeiro dígito representa a classe à qual a resposta pertence. As classes são as seguintes:

- 1xx: Informacional - requisição recebida, continue o processamento;
- 2xx: Sucesso - a ação foi recebida, entendida e aceita com sucesso;
- 3xx: Re-direção - outra ação deve ser realizada para que o pedido seja concluído;
- 4xx: Erro no cliente - a requisição contém erro de sintaxe;
- 5xx: Erro no servidor - o servidor falhou.

Além dos clientes e servidores, outros componentes podem estar envolvidos na comunicação entre o cliente e o servidor, tais como, o *proxy* e o *gateway*. Um *proxy* é um agente repassador de mensagens do cliente para o servidor. Ele é visto como um servidor para o cliente, e como um cliente para o servidor destino, podendo re-escrever,

se necessário, as mensagens que lhe são enviadas e repassá-las para outros servidores ou atendê-las. O *proxy* é freqüentemente utilizado como um agente auxiliar para aplicações que manipulam pedidos via protocolos que o cliente não implementa. Um *gateway* é um agente receptor de mensagens que atua como uma camada superior a de outros servidores e, se necessário, pode atuar como um tradutor de protocolos.

A Figura 2 mostra uma cadeia de comunicação que contém além do cliente e do servidor, um *proxy* e um *gateway*. Nesta cadeia o *proxy* recebe os pedidos enviados pelo cliente e os repassa para o *gateway*. Ao receber os pedidos, o *gateway* envia-os para o servidor destino para que sejam processados.

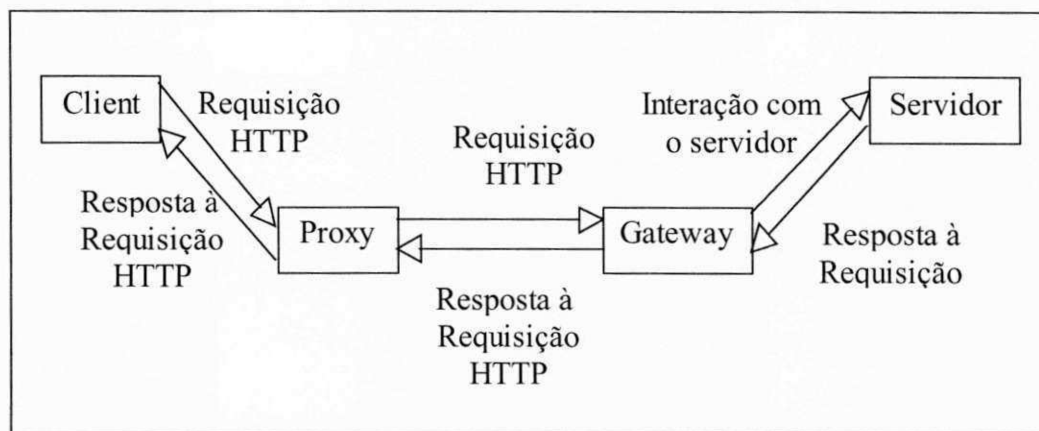


Figura 2 - Cadeia de Comunicação com Componentes Intermediários

Qualquer um dos componentes presentes na cadeia de comunicação mostrada na figura acima pode utilizar uma memória *cache* para agilizar o processamento das requisições. Assim, se a resposta de uma requisição esta armazenada na *cache* de algum componente intermediário, a cadeia de comunicação mostrada na figura acima é encurtada, tornando o processamento da requisição mais rápido. Por outro lado, nem todas as respostas podem ser armazenadas em *cache* para serem usadas no futuro (mensagens de erro são exemplos de respostas que não podem ser armazenadas na *cache*), além disso, requisições podem conter parâmetros que invalidam respostas armazenadas na *cache* (por exemplo, parâmetros utilizados para preservar a consistência da informação).

A versão do HTTP 1.1, especificada na RFC (*Request For Comments*) 2068,

define os seguintes métodos:

OPTIONS - Este método solicita informações sobre opções de comunicação disponíveis no pedido identificado pela URI. Ele permite que, antes de realizar qualquer operação sobre o recurso, o cliente adquira informações sobre opções e requisitos associados ao mesmo. Por exemplo, considere o seguinte pedido:

OPTIONS http://www.dsc.ufpb.br/~raquel.html HTTP/1.1

que requisitaria ao servidor *Web* do Departamento de Sistemas e Computação os métodos por ele implementados e aplicáveis ao recurso identificado pela URL.

GET - O método GET é usado quando se deseja obter um recurso de um servidor. Quando o método é executado com sucesso no servidor, a mensagem de resposta contém uma entidade representando o recurso solicitado. A forma como o método é processado pode variar dependendo dos parâmetros que são passados na mensagem de requisição. Por exemplo, considere o seguinte pedido:

GET http://www.dsc.ufpb.br/~raquel.html HTTP/1.1

que requisita ao servidor *Web* do Departamento de Sistemas e Computação o recurso identificado pela URL que contêm as informações de raquel.

HEAD - O método HEAD é similar ao método GET, mas neste método o servidor não deve enviar toda a informação do recurso na resposta. O recurso é composto pela informação e a meta-informação associada a ele. Este método retorna como resposta a meta-informação, e pode ser usado para testar a validade de uma ligação de hipertexto, acessibilidade e modificações recentes feitas sobre o recurso. Por exemplo, considere o seguinte pedido:

HEAD http://www.dsc.ufpb.br/labcom.html HTTP/1.1

que requisita ao servidor *Web* do Departamento de Sistemas e Computação as meta-informações associadas ao recurso identificado pela URL.

POST - O método POST é usado para solicitar que o servidor destino aceite a entidade enviada na requisição como uma nova subordinada do recurso identificado pela URI. Esse método foi projetado para que fosse possível realizar as seguintes tarefas: postagem de mensagens em listas de discussão, ***bulletin boards***, etc; provimento de informações submetidas a um processo que trata os dados associados a um formulário; adição de informações a um banco de dados; etc. Por exemplo, considere o seguinte pedido:

POST http://www.dsc.ufpb.br/cgi-bin/ferramenta.cgi HTTP/1.1

que requisita ao servidor *Web* do Departamento de Sistemas e Computação que processe a entidade enviada na requisição utilizando o recurso ferramenta.cgi.

PUT - O método PUT solicita que a entidade contida na requisição seja armazenada no servidor e identificada pela URI contida na requisição. Se o recurso identificado pela URI já existir, a entidade deve ser considerada como uma versão modificada do recurso existente. A diferença entre o método POST e o método PUT consiste na interpretação que é dada à URI contida na requisição enviada. No método POST, a URI identifica o recurso que irá processar a entidade contida na requisição. Por outro lado, a URI contida em uma requisição PUT identifica a própria entidade enviada ao servidor. Por exemplo, considere o seguinte pedido:

PUT http://www.dsc.ufpb.br/lsd.html HTTP/1.1

que requisita ao servidor *Web* do Departamento de Sistemas e Computação que armazene a entidade enviada na requisição e identifique-a pela URL contida na requisição.

DELETE - O método DELETE é usado para solicitar ao servidor a exclusão do recurso identificado pela URI contida na requisição. Por exemplo, considere o seguinte pedido:

DELETE http://www.dsc.ufpb.br/~raquel.html HTTP/1.1

que requisita ao servidor Web do Departamento de Sistemas e Computação que exclua o recurso identificado pela URL que contêm as informações de raquel.

TRACE - O método TRACE permite que um cliente envie uma requisição para um servidor e tenha esta mensagem, que pode ter sido re-escrita por intermediários, enviada de volta como resposta a sua requisição. Esse método é geralmente utilizado para realizar testes e diagnósticos do sistema. Por exemplo, considere o seguinte pedido:

TRACE http://www.dsc.ufpb.br/ HTTP/1.1

que verificaria quais os componentes intermediários existentes entre o cliente que enviou o pedido e o servidor *Web* do Departamento de Sistemas e Computação.

Como mencionado anteriormente, vários cabeçalhos podem estar contidos em mensagens de requisição/resposta HTTP. Algumas das informações contidas nos cabeçalhos das mensagens de requisição são:

- informações de autenticação; elas são enviadas quando o cliente deseja se autenticar com o servidor; essas informações de autenticação são enviadas na forma de credenciais, junto com a mensagem de requisição, para o servidor que mantém o recurso solicitado;

- endereço na Internet do administrador dos pedidos dos clientes; este campo determina quem é responsável pelo pedido solicitado, que pode ser contatado em casos de falhas;

- período máximo da última atualização; este campo é utilizado quando o cliente deseja obter o recurso desde que ele tenha sido modificado a partir da data especificada neste campo; se o recurso não foi modificado é enviado para o cliente uma resposta 304 (não modificado);

- o endereço no qual o identificador do recurso foi obtido; este cabeçalho permite que o servidor crie listas que contêm seus recursos de interesse, *cache* otimizadas, etc;

- informações sobre o *browser* que iniciou o pedido.

Por outro lado, as mensagens de respostas podem conter alguns cabeçalhos. Algumas informações contidas nesses cabeçalhos são:

- a localização exata do recurso identificado pela URI; a informação armazenada neste cabeçalho pode ser usada para finalidade de re-direção, ou seja, nas respostas 3xx este cabeçalho possui a URL do servidor que será usado na re-direção automática do recurso;

- informações sobre o *software* utilizado pelo servidor para tratar o pedido (ex. CERN/3.0, libwww/ 2.17);

- informações de autenticação; elas devem ser incluídas nas mensagens respostas 4xx (não autorizado); ela possui credenciais que indicam o esquema de autenticação utilizado e os parâmetros aplicados a URI.

Além desses cabeçalhos, as mensagens de requisições/respostas HTTP podem trazer ainda alguns cabeçalhos referentes à entidade. Algumas das informações contidas nesses cabeçalhos são:

- o conjunto de métodos suportados pelo recurso identificado pela URI; a finalidade deste campo é informar ao cliente quais são os métodos válidos associados ao recurso (ex. GET, HEAD);

- o mecanismo de codificação aplicado ao recurso; este campo é usado para permitir que um recurso seja compactado sem perder sua integridade (ex. x-gzip);

- o tamanho da entidade; uma aplicação deve utilizar este campo para indicar o tamanho da entidade a ser transferida;

- o tipo de entidade que será enviada para o cliente (ex. text/html);

- data de expiração; este campo indica a data/hora na qual a entidade deve ser considerada obsoleta; aplicações não devem armazenar esta entidade em *cache* se sua data estiver ultrapassada;

- a data e a hora da última modificação realizada sobre o recurso;

2.4. Projeto de um Servidor *Web proxy* com Suporte a Recursos Espelhados

Atualmente os recursos *Web* são identificados quase que exclusivamente através do esquema de identificação URL *(Uniform Resource Locator)* [Berners-Lee et al. 1994], cuja característica principal é utilizar a localização dos recursos como forma de nomeá-los. Por usar esse esquema de identificação, a confiabilidade da *Web* é reduzida, pois só é possível ter apenas uma cópia de um recurso *Web*; e na ocorrência de falhas os recursos podem se tornar indisponíveis.

Preocupado com isto, Fonsêca desenvolveu um trabalho que resolve esses problemas através do espelhamento de recursos *Web* [Fonsêca 1998]. Para isso, ele utilizou o esquema de identificação baseado em URNs (*Uniform Resource Names*). URNs têm como objetivo fornecer um identificador único, global, persistente e independente da localização. Uma URN é composta de URCs (*Uniform Resource Characteristics*), que contêm meta-informações, e URLs que definem a localização do recurso. Além disso, uma URN é sintaticamente composta de um NID (*Namespace Identifier*) e um NSS (*Namespace Specific String*). O NID é quem determina a interpretação sintática do NSS, e é quem o distingue com relação a outros esquemas de nomeação existentes. Por outro lado, o NSS é definido pelo NID, ou seja, o NSS é dependente e governado pelas regras do NID [Iannella et al. 1996]. Por exemplo, considere a seguinte URN hipotética:

urn:telefone:1234567890

que representa um número de telefone particular, cujo NID é a cadeia de caracteres *telefone*, e o NSS é a cadeia *1234567890*.

O NID proposto por Fonsêca é o WMR (*Web Mirroed Resources*), e o NSS é formado por três campos: o primeiro campo é um nome de domínio Internet que indica o domínio do gerente de um dado grupo de recursos; o segundo campo indica o grupo ao qual um determinado recurso pertence; e o terceiro campo é a identificação do recurso propriamente dito. Por exemplo, considere a seguinte URN:

urn:wmr:dsc.ufpb.br/curriculos/raquel.html

que poderia ser utilizada para identificar a página HTML que contém as informações sobre a aluna Raquel pertencente ao grupo currículos do domínio de gerência dsc.ufpb.br.

Além disso, o esquema de identificação, proposto por Fonsêca, oferece suporte para que um determinado grupo possua vários recursos espelhados. Por exemplo, considere as seguintes URNs:

urn:wmr:dsc.ufpb.br/currículos/raquel.html

urn:wmr:dsc.ufpb.br/currículos/tercio.html

que identificam unicamente os recursos raquel.html e tercio.html pertencentes ao grupo currículos.

Entretanto, para que fosse possível utilizar o esquema de URNs em seu trabalho, Fonsêca utilizou um serviço de resolução de URNs. Este serviço realiza o mapeamento de uma URN nas suas várias URLs através de um resolvedor. Este resolvedor foi dividido em duas partes, uma local e uma global. O resolvedor local extrai o nome de domínio de gerência e o nome do grupo da URN, e verifica se o mapeamento pode ser realizado utilizando as informações mantidas em sua *cache*; quando isso não é possível o resolvedor local faz uma consulta ao resolvedor global. O resolvedor global analisa sua base de dados para verificar quais são as URLs associadas com a URN, e retorna as respectivas URLs.

Como mostra a Figura 3, para realizar o mapeamento da URN nas suas várias URLs, Fonsêca utilizou as funcionalidades de um *proxy*. Quando um cliente faz uma requisição, o *proxy* intercepta-a e recorre a um resolvedor para descobrir a localização do recurso identificado pela URN. Uma vez adquirida as URLs associadas ao recurso em questão, o *proxy* recupera um dos recursos usando o protocolo HTTP e repassa-o para o cliente. Além disso, para mascarar problemas de indisponibilidade dos recursos, o *proxy* pode fazer múltiplas requisições aos diferentes servidores que mantêm cópias do recurso identificado pela URN.

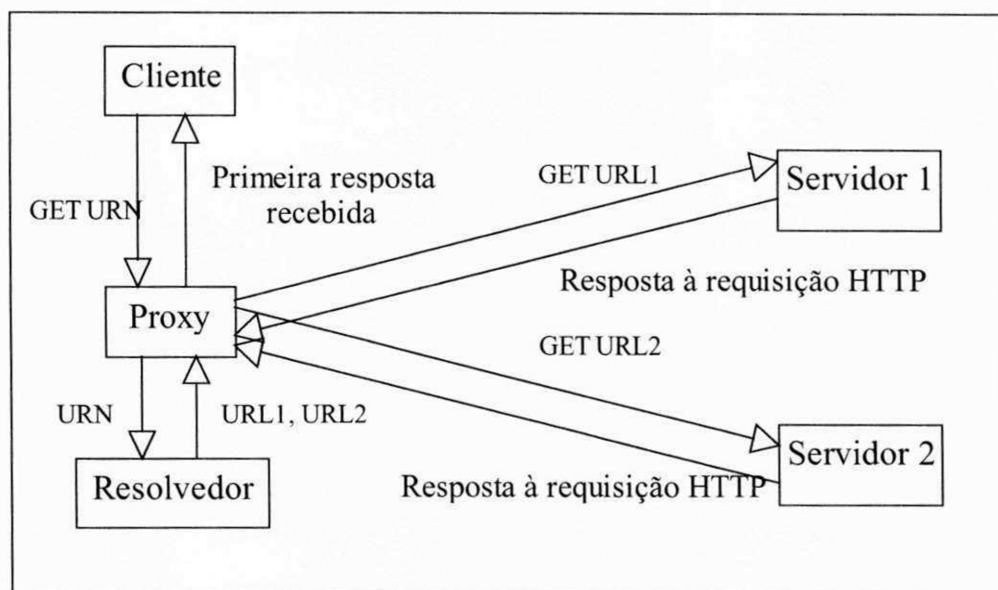


Figura 3 - Espelhamento de recursos com auxílio de um *proxy*

Fonsêca desenvolveu também em seu trabalho um esquema dinâmico de balanceamento de carga. Esse esquema tenta reduzir o tempo de resposta das requisições feitas pelos clientes sem sobrecarregar a rede.

Apesar de solucionado os problemas de acesso para leitura, Fonsêca não resolveu o problema de manter a consistência dos recursos espelhados, ou seja, o servidor *proxy* dá um tratamento especial apenas para as requisições que contêm os métodos GET ou HEAD. Outras requisições que contêm outros métodos definidos pelo protocolo HTTP não são reconhecidas.

2.5. Atualização dos Recursos Espelhados

O protocolo HTTP define alguns métodos que podem ser aplicados aos recursos armazenados no servidor para atualizá-los, tais como o método PUT e o método DELETE. O método DELETE deve ser aplicado em uma exclusão do recurso. O método PUT, por sua vez, deve ser aplicado quando se deseja criar um recurso, ou realizar uma atualização de um recurso já existente.

Quando esses métodos são aplicados sobre recursos espelhados, existe a possibilidade de serem introduzidas inconsistências. Se uma requisição de um método DELETE não for aplicada sobre todas as cópias do recurso, é possível que alguma cópia continue disponível. Inconsistências podem ser introduzidas também se uma requisição de um método PUT não atualizar todas as cópias do recurso espelhado.

Mesmo que o *proxy* envie as requisições para todos os servidores que armazenam os recursos, existe a possibilidade de ocorrerem falhas, e com isso permitir que inconsistências sejam geradas.

Considere o exemplo da Figura 4. Os servidores A, B, C e D armazenam o recurso espelhado X. Inicialmente, este recurso possui o mesmo estado em todos os servidores. Suponha que ocorreu um particionamento na rede e dois subgrupos foram gerados, um contendo os servidores A e B, e outro contendo os servidores C e D. Suponha também que o cliente envia uma mensagem de requisição, contendo o método PUT, para todos os servidores que armazenam o recurso X. Como ocorreu um particionamento na rede, apenas as cópias do recurso X que estão armazenadas nos servidores A e B serão atualizadas, tornando, com isso, as cópias do recurso inconsistentes, já que as cópias do recurso armazenadas nos servidores C e D não são atualizadas. Inconsistências podem ser geradas também se um dos servidores se tornar indisponível durante uma atualização ou exclusão do recurso X, uma vez que a operação de atualização ou remoção só será realizada sobre as cópias dos recursos que estão armazenadas nos servidores disponíveis.

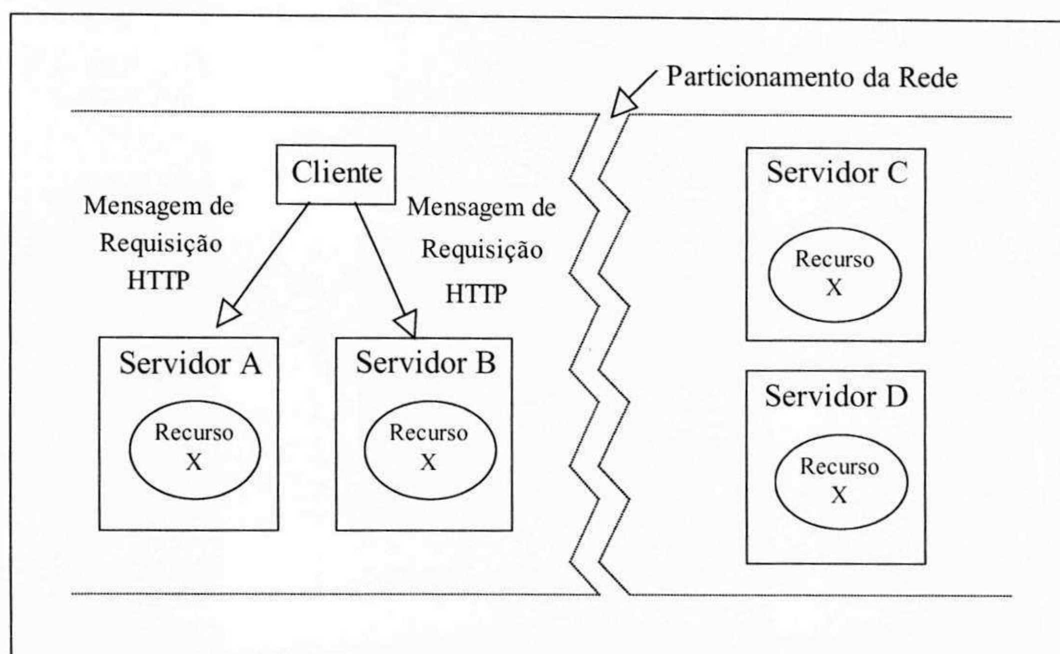


Figura 4 - Particionamento na Rede

Assim, além de executar os métodos sobre os recursos espelhados para atualizá-los, é necessário que o *proxy* e os servidores que armazenam os recursos colaborem efetivamente no processamento dos métodos para que inconsistências não sejam geradas.

Além disso, quando inclusões e exclusões de qualquer recurso espelhado precisam ser realizadas, é necessário que as informações de mapeamento do resolvedor de URNs sejam também atualizadas, preferencialmente, de forma transparente para o usuário.

Nosso trabalho visa exatamente resolver esses problemas de inconsistências através da utilização de um esquema de gerência de recursos espelhados que garante sua consistência independente das falhas que possam ocorrer durante a execução dos métodos PUT e DELETE. Em nosso esquema, utilizaremos alguns mecanismos de controle de consistência durante as atualizações dos recursos espelhados. Assim, através desses mecanismos, atualizações poderão ser realizadas tanto quando todos os servidores que mantêm cópias de um determinado recurso espelhado estão disponíveis quanto quando existe pelo menos um servidor disponível. A escolha de um destes mecanismos vai depender das necessidades do recurso espelhado.

Os mecanismos de controle de consistência utilizados para garantir a consistência dos recursos espelhados e o processamento das mensagens de requisição serão descritos com mais detalhes no capítulo 4. Antes, no capítulo 3, discutiremos alguns trabalhos existentes sobre soluções para o problema de consistência de cópias espelhadas.

Capítulo 3

Mantendo a Consistência da Informação Replicada

3.1. Introdução

A replicação consiste em criar cópias de uma determinada informação e distribuí-las em diversas localizações. A técnica de replicação é, geralmente, aplicada para aumentar a disponibilidade da informação na presença de falhas no sistema, e também como uma forma de reduzir o tráfego na rede de comunicação e diminuir o tempo de acesso às informações. Ao utilizar a técnica de replicação, é necessário pensar em como manter a consistência das informações replicadas, pois elas precisam estar sempre íntegras e consistentes. Assim, qualquer operação de atualização deve ser executada sobre todas as cópias de uma determinada informação replicada, para que elas possuam o mesmo estado durante todo tempo.

A consistência de informações replicadas pode ser afetada por diversos fatores, dentre os quais podemos citar: a possibilidade de dois usuários tentarem atualizar uma informação replicada simultaneamente; a indisponibilidade dos servidores que armazenam informações replicadas, o que pode gerar cópias desatualizadas; e a possibilidade do particionamento da rede isolar subconjuntos das informações replicadas criando a situação onde diferentes usuários podem alterá-las de diferentes formas.

O objetivo deste capítulo é apresentar algumas abordagens existentes que resolvem os problemas de inconsistência gerados por atualizações de informações replicadas mesmo na presença de particionamento de rede e indisponibilidade de servidores. Na seção 3.2 apresentaremos as classes de protocolos existentes para manter a consistência das informações replicadas, suas vantagens e desvantagens. Na seção 3.3 apresentaremos um breve discussão sobre as abordagens existentes.

3.2. Protocolos de Consistência

Como mencionado anteriormente, a consistência de informações replicadas pode ser afetada por diversos fatores, tais como: indisponibilidade do servidor, particionamento da rede, etc. Uma solução para manter a consistência de informações replicadas na presença de particionamento da rede seria recusar todas as operações de atualização e de leitura até que o particionamento deixasse de existir. Este método garantiria a consistência, mas eliminaria toda a disponibilidade, pois não seria permitido a execução de nenhuma operação de atualização e de leitura durante o intervalo de tempo que a rede estivesse particionada. Outra alternativa seria permitir apenas operações de leitura serem realizadas. Neste caso, como nenhuma atualização seria realizada, a consistência seria preservada e a disponibilidade, em relação ao método anterior, seria muito maior. Por outro lado, se certas operações de atualização fossem permitidas de maneira tal que a consistência não fosse comprometida, a disponibilidade seria ainda maior. O maior grau de disponibilidade seria conseguido se fosse permitido realizar operações de atualização em qualquer partição. Existem vários protocolos que podem ser utilizados para manter a consistência das informações replicadas; eles podem ser classificados nas seguintes classes: os protocolos pessimistas e os protocolos otimistas.

As estratégias pessimistas estão preocupadas com a consistência das réplicas mesmo que isso venha a comprometer sua disponibilidade. Elas previnem que inconsistências sejam geradas por limitar o acesso às informações replicadas. Por outro lado, as estratégias otimistas não limitam a disponibilidade; qualquer operação pode ser executada em qualquer partição que contenha uma réplica da informação. Elas assumem

que conflitos raramente acontecem e que se acontecerem poderão ser tratados posteriormente sem muito prejuízo para a aplicação.

Essas estratégias podem realizar as atualizações de forma síncrona ou assíncrona. As atualizações realizadas de forma síncrona utilizam apenas uma transação para alterar todas as réplicas. Por outro lado, atualizações realizadas de forma assíncrona permitem que cada réplica seja alterada por transações independentes: dessa forma, se uma operação sobre uma réplica falhar, as outras operações sobre as outras réplicas podem ser realizadas.

O problema de realizar atualizações síncronas é que, se pelo menos uma réplica falhar durante uma alteração, a transação será cancelada ou bloqueada, enquanto que, nas atualizações assíncronas existe a possibilidade de haver colisões (conflitos) quando partições são reintegradas. Colisões ocorrem quando cópias de uma determinada informação replicada são atualizadas em partições distintas. Colisões podem ocorrer também em uma mesma partição desde que as atualizações concorrentes não sejam serializadas, ou seja, após o tempo em que uma primeira atualização foi iniciada e processada em algumas cópias, uma segunda atualização é iniciada e processada em outras cópias antes que a primeira atualização tenha sido realizada sobre elas.

3.2.1. Estratégias Pessimistas

Existem vários métodos que utilizam a abordagem pessimista para evitar que inconsistências sejam geradas quando ocorre falhas nos nodos ou particionamento na rede durante atualizações das informações replicadas. No método *primary site* [Alsberg e Day 1976], um dos *sites*, que mantém uma cópia da informação replicada, é o primário enquanto que os outros *sites* são os *backups*. Qualquer operação a ser realizada sobre a informação replicada é primeiro enviada para o *site* primário e, periodicamente, ele envia seu estado para os *backups*. Se o primário falha, um dos *backups* torna-se o primário e inicia a execução das operações. Se ocorrer um particionamento, as operações são realizadas na partição que contém o *site* primário. Uma vantagem deste método é que não existe a possibilidade de haver colisões (conflitos), uma vez que todas

as operações são primeiro realizadas no *site* primário para depois serem realizadas sobre os *backups*. Sua desvantagem é que a disponibilidade é reduzida, uma vez que todas as operações só poderão ser realizadas em um *site*; além disso, se o *site* primário falhar, o sistema ficará indisponível até que seja escolhido o novo primário.

Outra método que utiliza a abordagem pessimista é o *weighted voting* [Gifford 1979]. Neste método, cada cópia da informação replicada possui um número de votos. Assim, o nodo que desejar executar uma operação de leitura ou atualização deve adquirir um quorum de leitura ou escrita, respectivamente. Além disso, é associado a cada cópia da informação um número de versão, que inicialmente é zero. Uma operação de leitura ou atualização é realizada da seguinte forma: inicialmente, o nodo que iniciou a operação envia um pedido para todos as cópias solicitando os seus votos; ao receber o pedido, cada cópia envia sua versão e o número de votos que ela possui; ao adquirir os votos de todas as cópias, o nodo verifica se o total de votos é igual ou maior do que o quorum necessário para realizar a operação; se o nodo adquiriu um quorum, a operação poderá ser realizada. Uma vantagem deste método é que ele oferece maior disponibilidade do que o método anterior. Por outro lado, ele aumenta a sobrecarga de comunicação, já que, é necessário adquirir um quorum para realizar as operações de leitura e atualização.

A seguir, apresentaremos, com mais detalhes, mais três métodos que utilizam a abordagem pessimista para manter a consistência de informações replicadas na presença de falhas dos nodos e particionamento da rede. A escolha destes métodos se deve ao fato de que, eles fornecem uma maior disponibilidade, em relação aos métodos acima citados. Além disso, eles não necessitam de um quorum para realizar operações de atualização sobre as informações replicadas tornando, com isso, a rede menos sobrecarregada.

- Partição Majoritária [Jajodia e Mutchler 1989]

Jajodia e Mutchler desenvolveram em seu trabalho um algoritmo de controle de consistência pessimista que visa gerenciar arquivos replicados na presença de

particionamento da rede. Na ocorrência de particionamento, as atualizações são realizadas na partição majoritária.

Para tal, atribui-se um nível de prioridade entre os *sites* que armazenam informações replicadas, ou seja, todos os *sites* são linearmente ordenados. Esta ordenação linear será utilizada pelo algoritmo de controle de consistência pessimista para decidir quem será a partição majoritária quando existem duas subpartições com o mesmo número de *sites*.

Além disso, são associados a cada cópia de uma determinada informação os seguintes atributos: número da versão (VN_i); número de *sites* que participaram da mais recente atualização (SC_i); *distinguished site* (DS_i), este atributo identifica o *site* de maior prioridade entre todos os outros *sites* que participaram da mais recente atualização; e um vetor de *sites* alterados (SV_i). O vetor de *sites* alterados é uma tupla ordenada onde cada elemento representa cada *site*. Assim, o vetor de *sites* alterados do *site* S_i tem valor 1 nas cópias que participaram da mais recente atualização e 0, caso contrário. Além disso, se ocorre um particionamento separando um *site* S_i de um *site* S_j, o elemento que representa o *site* S_j no vetor de *sites* alterados de S_i é mudado de 1 para 0.

O algoritmo de controle de consistência pessimista determina que um *site* S_i só pode iniciar um pedido de atualização se ele pertence à partição majoritária, para isso ele utiliza o seguinte algoritmo: O *site* S_i conta o número de 1's no vetor de *sites* alterados SV_i; se o número de 1's for menor que a metade do número *sites* que participaram da mais recente atualização (número de 1's > $SC_i/2$), ou se o número de 1's = $SC_i/2$ e se o *site* DS_i possui valor 1 no vetor de *sites* alterados, então S_i pertence à partição majoritária; caso contrário, S_i não pertence à partição majoritária.

Uma vez que S_i verificou que pertence à partição majoritária, ele envia uma mensagem de atualização para todos os *sites* pertencentes `a partição de S_i. Ao receber esta mensagem, cada *site* envia uma confirmação para S_i. Quando S_i recebe a confirmação de todos os *sites,* ele processa a atualização em sua réplica local e envia um *commit* para todos os *sites,* que ao receberem esta mensagem modificam os quatro

atributos associados à cópia sendo atualizada (VN_i, SC_i, DS_i e SV_i).

Para se recuperar de falhas durante pedidos de atualização, o algoritmo se comporta da seguinte forma: se ocorrerem falhas depois de S_i enviar a mensagem de atualização e antes de enviar o *commit*, ele deve enviar um *abort*. Qualquer *site* que recebe um pedido de *abort* pode realizá-lo se ele ainda não enviou uma confirmação da mensagem de atualização; se ocorrerem falhas depois que S_i envia o *commit* e antes de qualquer um dos *sites* recebê-lo, o *site* deve consultar S_i para que ele possa decidir se deve realizar ou não a atualização.

Considere o seguinte exemplo: assuma que os *sites* A, B, C, D e E armazenam uma cópia de um determinado arquivo *f* e estão linearmente ordenados da seguinte forma: A > B > C > D > E. Inicialmente, estes *sites* estão conectados e formam uma única partição. Suponha que o arquivo *f* foi atualizado nove vezes, assim o estado do banco de dados ficará da seguinte forma (o "-" no DS indica que nenhum dos *sites* têm maior prioridade em relações aos outros *sites).*

	A	*B*	*C*	*D*	*E*
VN:	*9*	*9*	*9*	*9*	*9*
SC:	*5*	*5*	*5*	*5*	5
DS:	-	-	-	-	-
SV:	*(1, 1, 1, 1, 1)*	*(1, 1, 1, 1, 1)*	*(1, 1, 1, 1, 1)*	*(1, 1, 1, 1, 1)*	*(1, 1, 1, 1, 1)*

Neste ponto ocorre um particionamento e os *sites* A, B e C são isolados dos *sites* D e E. Assim, o vetor de *sites* alterados ficará no seguinte estado:

	A	*B*	*C*	*D*	*E*
VN:	*9*	*9*	*9*	*9*	*9*
SC:	*5*	*5*	*5*	*5*	*5*
DS:	-	-	-	-	-
SV:	*(1, 1, 1, 0, 0)*	*(1, 1, 1, 0, 0)*	*(1, 1, 1, 0, 0)*	*(0, 0, 0, 1, 1)*	*(0, 0, 0, 1, 1)*

A partição contendo os *sites* A, B, e C formam uma partição majoritária, por isso eles podem continuar a alterar o arquivo. Suponha que eles alteram o arquivo *f* duas

vezes, e com isso as entradas VN e SC ficarão da seguinte forma:

	A	*B*	*C*	*D*	*E*
VN:	*11*	*11*	*11*	*9*	*9*
SC:	*3*	*3*	*3*	*5*	*5*
DS:	-	-	-	-	-
SV:	*(1, 1, 1, 0, 0)*	*(1, 1, 1, 0, 0)*	*(1, 1, 1, 0, 0)*	*(0, 0, 0, 1, 1)*	*(0, 0, 0, 1, 1)*

Suponha também que o *site* B torna-se isolado dos *sites* A e C. Depois disto, os *sites* A e C alteram o arquivo *f* duas vezes. Assim, os atributos do arquivo ficarão com o seguinte estado:

	A	*B*	*C*	*D*	*E*
VN:	*13*	*11*	*13*	*9*	*9*
SC:	*2*	*3*	*2*	*5*	*5*
DS:	-	-	-	-	-
SV:	*(1, 0, 1, 0, 0)*	*(0, 1, 0, 0, 0)*	*(1, 0, 1, 0, 0)*	*(0, 0, 0, 1, 1)*	*(0, 0, 0, 1, 1)*

Por fim, para que as subpartições sejam unidas é necessário que todas as cópias da nova partição sendo criada (esta nova partição é a união das subpartições) possuam a mesma versão, e o valor do SC_i e do DS_i tenham o mesmo valor. Para isso, os *sites* que desejam se unir à partição majoritária devem solicitar a versão do arquivo *f* e, juntamente com todos os *sites* pertencentes à partição majoritária, incrementá-la. Além disso, todos os atributos são atualizados de acordo com número de *sites* que participaram desta reconciliação. Considere o exemplo anterior, suponha que o *site* D une-se com os *sites* A e C. Assim, os atributos de A, C e D ficarão da seguinte forma:

	A	*B*	*C*	*D*	*E*
VN:	*14*	*11*	*14*	*14*	*9*
SC:	*3*	*3*	*3*	*3*	*5*
DS:	-	-	-	-	-
SV:	*(1, 0, 1, 1, 0)*	*(0, 1, 0, 0, 0)*	*(1, 0, 1, 1, 0)*	*(1, 0, 1, 1, 0)*	*(0, 0, 0, 1, 0)*

A vantagem deste método é que os *sites* podem determinar localmente se pertencem à partição majoritária, diminuindo, com isso, a sobrecarga de comunicação e

os riscos de *deadlock*[2]. Por outro lado, se ocorrer vários particionamentos e, durante este período, nenhuma atualização for realizada, os *sites* poderão ficar bloqueados até que sejam integrados, ou seja, se a última atualização foi realizada sobre todos os *sites* e, depois disto, ocorrer vários particionamentos, existe a possibilidade do número de 1's no vetor de *sites* alterados (SV) de cada *site* não ser maior do que a metade do número de *sites* que participaram da mais recente atualização, e com isso nenhum dos *sites* poderão realizar atualização até que sejam integrados.

- Votação Dinâmica [Davcev 1989]

Neste método, Davcev apresenta um esquema de votação dinâmica para controle de consistência e recuperação de arquivos replicados em sistemas distribuídos. Este esquema tolera falhas dos *sites* e particionamento da rede. A votação é realizada sempre que o arquivo replicado é aberto, quando ocorrem falhas nos *sites* ou na rede, ou quando ocorre uma reparação. A diferença deste método para o método anterior é que ele não necessita da maioria dos votos das cópias para executar a atualização. Neste esquema o número de votos requerido muda dinamicamente e depende unicamente do número de cópias que falharam durante a última atualização.

Neste método, um arquivo replicado consiste de uma coleção de cópias físicas, onde cada cópia física tem associado um número de versão e um vetor de partição. O número de versão de uma cópia física é um inteiro que representa o número de operações de atualização realizadas sobre ela. Por outro lado, o vetor de partição de uma cópia i é uma tupla, onde cada elemento representa uma cópia física. Ele é denominado como sendo um Z^i e contém as coordenadas Z_j^i que são os números de partição entre cópias físicas i e j para cada cópia física j de um arquivo replicado. O número de partição Z_j^i têm valor zero se não existe partições entre as cópias i e j, e o número de versão da cópia i, caso contrário. Além disso, cada coordenada Z_j^i tem associada um

[2]*Deadlock* é uma situação onde duas as mais operações estão em um estado de espera simultânea, ou seja, cada operação esta esperando pela outra para finalizar.

voto.

Estas definições podem ser melhor ilustradas através do seguinte exemplo. Assuma que o arquivo *f* consiste de três cópias físicas localizadas nos *sites* A, B e C. Inicialmente, o número de versão x_i de todas as cópias é 1; os números de partição em todos os vetores de partição Z^i são 0; a cópia no *site* A tem voto 2, e todos as outras cópias têm voto 1. Assim, os vetores ficarão da seguinte forma:

$A\ (x = 1)$	$B\ (x = 1)$	$C\ (x = 1)$
$Z^A = (0{:}2,\ 0{:}1,\ 0{:}1)$	$Z^B = (0{:}2,\ 0{:}1,\ 0{:}1)$	$Z^C = (0{:}2,\ 0{:}1,\ 0{:}1)$

Assuma que ocorre oito operações de atualização; em seguida, ocorre um particionamento, e com isso o *site* C separa-se dos *sites* A e B. Assim, os vetores ficarão da seguinte forma:

$A\ (x = 9)$	$B\ (x = 9)$	$C\ (x = 9)$
$Z^A = (0{:}2,\ 0{:}1,\ 9{:}1)$	$Z^B = (0{:}2,\ 0{:}1,\ 9{:}1)$	$Z^C = (9{:}2,\ 9{:}1,\ 0{:}1)$

Nesta abordagem, as operações de atualização são realizadas sobre todas as cópias físicas armazenadas nos *sites* pertencentes à partição majoritária. Além disso, uma operação de atualização é realizada por um coordenador, que nada mais é do que o *site* que iniciou sua execução. Uma operação de atualização comporta-se da seguinte forma:

1) o coordenador envia uma mensagem *prepare* para cada *site* pertencente à partição majoritária;

2) cada *site,* quando estiver pronto para realizar a operação, retorna uma mensagem *pronto* para o *site* coordenador;

3) ao receber as mensagens *pronto* de todos os *sites* pertencentes à partição majoritária, o coordenador envia uma mensagem *complete* para todos os *sites.*

Além disso, Davcev definiu as operações *major* e *merge*. A função *major* determina se um determinado *site* pertence à partição majoritária. Esta função retornará

"*false*", se o *site* não pertence à partição majoritária, "*read ok*", indicando que apenas operações de leitura poderão ser realizadas, e "*read/write ok*" indicando que o *site* pertence à partição majoritária. Seu algoritmo comporta-se da seguinte forma: ele verifica quais são todas as coordenadas zero do vetor de partição e soma seus respectivos votos WZ; em seguida, verifica quais são as maiores coordenadas e soma seus respectivos votos WMZ (no exemplo mencionado anteriormente, a maior coordenada seria $Z^A{}_3 = 9$ e seu voto é 1); por fim, o valor retornado seria:

se WZ>WMV então retorne "read/write ok"

se WZ == WMV então retorne "read ok"

caso contrário retorne "false"

A função *merge* determina se partições podem ser unidas para que seja criado uma única partição. Ela retorna "*true*", se partições podem ser unidas para gerar uma partição majoritária única ou se a partição pode ser expandida, ou "*false*", caso contrário. Seu algoritmo comporta-se da seguinte forma:

1) para cada partição *i* e cada *site j* dentro da partição, determine WZ_j e WMV_j;

2) se $WZ_j > WMV_j$ continue no passo 3), caso contrário, se as partições possuem o mesmo estado, os *sites* destas partições podem ser consolidados por setar suas correspondentes entradas nos vetores de versão para zero. WZ e WMV são então recalculados para determinar se a nova partição é majoritária. Caso ela não seja uma partição majoritária, a função *merge* retorna "*false*";

3) Vamos supor que o *site* k, com versão x_k, está sendo integrado à partição majoritária. Se $x_k = x_j$ então os dados armazenados nas duas partições não divergem. Se $x_j > x_k$ então o arquivo armazenado no *site j* da partição *i* deve ser copiado no *site k*.

4) altere o número de versão do *site k* e o vetores de partição de cada *site j* da partição *i* e do *site k* da seguinte forma:

$$x_k = x_j, \qquad Z_k^j = Z_j^k = 0$$

Nesta situação, o algoritmo retorna "*true*".

Continuando com o exemplo anterior. Assuma que duas operações de atualização são realizadas e, em seguida, o *site* B desliga-se do *site* A. Assim os vetores de partição e o número de versão ficarão da seguinte forma:

$A\ (x = 11) \qquad B\ (x = 11) \qquad C\ (x = 9)$

$Z^A = (0{:}2,\ 11{:}1,\ 9{:}1) \quad Z^B = (11{:}2,\ 0{:}1,\ 9{:}1) \quad Z^C = (9{:}2,\ 9{:}1,\ 0{:}1)$

Suponha agora que o *site* A pode se comunicar com o *site* C. Estes *sites* são unidos para formar uma nova partição majoritária. Assim, os vetores de partição e o número de versão terão o seguinte estado:

$A\ (x = 11) \qquad B\ (x = 11) \qquad C\ (x = 11)$

$Z^A = (0{:}2,\ 11{:}1,\ 0{:}1) \quad Z^B = (11{:}2,\ 0{:}1,\ 9{:}1) \quad Z^C = (0{:}2,\ 9{:}1,\ 0{:}1)$

A vantagem deste método é que não é necessário a intervenção do usuário para realizar a recuperação de falhas dos nodos e particionamento da rede. Além disso, ele oferece uma maior disponibilidade do que os métodos anteriormente descritos. Por outro lado, sua desvantagem é que, durante as operações de recuperação, as cópias das informações ficam indisponíveis, já que, é necessário bloqueá-las para que a recuperação seja realizada.

- Protocolo de Validação em Duas Fases (*Two-Phase Commit Protocol*) [Jalote 1994]

A finalidade do protocolo de validação é garantir que, ou todos os nodos realizam uma operação de atualização, ou nenhum deles realiza a operação. O protocolo mais comumente utilizado para garantir esta funcionalidade é o protocolo 2PC (*Two-Phase Commit).* Considere que o nodo onde a transação é iniciada é chamado de coordenador, enquanto que, os outros nodos são chamados de participantes. Antes de iniciar o protocolo 2PC, o coordenador envia a operação de atualização para todos os participantes; os participantes executam a operação e enviam uma resposta para o

coordenador. Quando o coordenador recebe a resposta de todos os participantes, o protocolo é iniciado.

O 2PC é realizado em duas fases bem evidentes. Na primeira fase, o coordenador envia pedidos de *commit* para todos os participantes. Cada participante envia uma resposta indicando se querem ou não realizar o *commit*, ou seja, uma resposta SIM ou NÃO, respectivamente. Na segunda fase, o coordenador recebe todos as respostas dos participantes; se todos os participantes enviaram um SIM, o coordenador envia um COMMIT para todos os participantes; caso contrário, ele envia um ABORT. Cada participante espera por uma mensagem do coordenador. Ao receber a mensagem, o participante realiza ou um COMMIT ou um ABORT e envia a confirmação para o coordenador. Quando o coordenador recebe a resposta de todos os participantes, a operação é finalizada.

Além disso, para evitar que os nodos esperem indefinidamente por respostas, em caso de falhas, tanto o coordenador quanto os participantes só esperam por mensagens em um tempo finito. Assim, se o coordenador não receber uma confirmação do pedido de *commit*, ele pode decidir realizar o ABORT. Se os participantes não recebem os pedidos de *commit,* eles podem decidir abortar. Por outro lado, se eles não recebem o COMMIT ou ABORT durante um tempo finito, eles devem consultar outros participantes para decidir o que fazer.

Para garantir que as transações[3] concorrentes sejam realizadas sobre todos os nodos na mesma ordem, é utilizado o protocolo de controle de concorrência 2PL estrito distribuído (*strict Two-Phase Locking*). Este protocolo é executado em duas fases. Na primeira fase, todos os bloqueios são adquiridos em todos os nodos; na segunda fase, todos os bloqueios são liberados; porém, essa liberação só é realizada no fim da

[3]Uma transação é uma coleção de operações executadas sobre uma ou mais informações que faz com que elas passem de um estado de consistência para outro.

transação global[4], ou seja, quando todos as subtransações da transação global são realizadas sobre todos os nodos.

A vantagem desta abordagem é que ao mesmo tempo que a consistência é preservada, os conflitos são evitados, pois operações concorrentes são realizadas na mesma ordem sobre todos os nodos. Por outro lado, aumenta a sobrecarga de comunicação, pois em um sistema com n nodos, o número total de mensagens necessárias para realizar uma atualização seria $6n$.

Uma forma de reduzir o número total de mensagens seria enviar a operação de atualização na primeira fase do protocolo 2PC. Assim, o número total de mensagens necessárias para realizar um atualização seria $4n$ [Date 1985].

A desvantagem deste método é que, além de aumentar a sobrecarga na rede, ele reduz a disponibilidade, uma vez que, todos os nodos ficam bloqueados enquanto uma transação está em execução.

3.2.2. Estratégias Otimistas

Como mencionado anteriormente, estratégias otimistas realizam operações de atualização sobre as informações replicadas mesmo na presença de particionamento da rede ou falhas de alguns servidores que mantêm cópias da informação replicada. Nesta estratégia, as inconsistências são eliminadas quando as partições são integradas.

Existem vários métodos que utilizam a abordagem otimista para resolver os problemas de inconsistência das informações replicadas. O método *optimistic protocol* utiliza grafos de precedência para detectar os conflitos entre transações quando as partições são integradas [Davidson 1984]. Nestes grafos, os nodos representam as transações e os arcos indicam o conflito entre duas transações. Quando duas transações estão em conflito, uma ou ambas poderão ser refeitas, com isso, a consistência é

[4]transação global é uma transação formada de várias transações, onde cada transação é executada em um determinado nodo.

preservada. A desvantagem deste método é que refazer transações em conflito reduz a disponibilidade das informações replicadas. Por outro lado, se conflitos raramente acontecem, ele oferece alta disponibilidade durante particionamentos na rede.

Outro método que utiliza a abordagem otimista é a *log transformations* [Blaustein et. al 1983], ele é similar ao método anterior. Durante particionamentos, os *logs* mantêm quais transações foram executadas e em qual ordem. Assim, após a reconciliação, os *logs* são analisados e as transações em conflito são refeitas. A desvantagem deste método é que ele requer a intervenção do usuário quando partições são reparadas. Por outro lado, as transações sendo refeitas podem ser realizadas enquanto transações normais estão executando.

Nesta seção apresentaremos dois métodos que utilizam a abordagem otimista para manter a consistência das informações replicadas. Estes métodos foram escolhidos por mais se adequarem às necessidades de nosso projeto.

- Vetores de Versão [Parker et. al. 1983]

Esta abordagem utiliza vetores de versão para detectar inconsistências entre cópias de uma determinada informação replicada pertencentes a partições distintas. Um vetor de versão é caracterizado como sendo uma seqüência de *n* pares, onde *n* é o número de *sites* que armazenam cópias da informação replicada. Cada par possui o nome do *site* e a mais recente versão da informação. Para cada cópia de uma determinada informação replicada é associado um vetor de versão.

Esta abordagem considera também que os conflitos podem ser classificados como sendo conflitos de nome ou conflitos de versão. Os conflitos de nome ocorrem quando dois ou mais conjuntos de informações (ex. um arquivo) são criados em partições distintas com o mesmo nome. Por outro lado, os conflitos de versão ocorrem quando dois conjuntos de informações existentes são alterados em transações distintas.

Os conflitos causados por operações de atualização sobre as cópias de informações replicadas em partições distintas são detectados através da comparação dos

vetores de versões. Um vetor v domina um vetor v' se eles são vetores de versão de uma mesma informação replicada e $v_i \geq v_i'$ para todo i = 1,..., n. Assim, dois vetores estão em conflito se não existe uma relação de dominação entre os dois. Os vetores de versão são usados da seguinte forma:

1) Sempre que um *site* S inicia uma operação de atualização, o componente referente a este *site* no vetor de versão é incrementado;

2) Exclusões e renomeações de arquivos são tratados como alterações. A exclusão de um arquivo consiste em atribuir valor zero para sua versão no vetor de versão. Assim, quando todas as versões de um arquivo têm o valor zero, ele pode ser excluído do sistema;

3) Quando conflitos de versão são reconciliados dentro de uma partição, a versão do arquivo será a maior das versões armazenadas nos vetores de versão dos *sites* que participaram da reconciliação. Além disso, o *site* que iniciou a reconciliação incrementa sua versão.

4) Quando cópias de uma determinada informação são armazenadas em novos *sites,* são incluídos informações dos novos *sites* em todos os vetores de versão de todos os *sites* que também armazenam esta cópia.

Considere o seguinte exemplo (Figura 5): assuma que inicialmente os *sites* A, B e C armazenam a mesma versão de um determinado arquivo *f*. Suponha que ocorre um particionamento e com isso duas subpartições são geradas, uma contendo os *sites* A e B, e outra contendo o *site* C. Além disso, suponha que A atualiza *f* duas vezes. Assim, os vetores de versão de A e B são iguais e possuem o seguinte estado: <A:2, B:0, C:0>. Logo em seguida, ocorre um particionamento, e com isso B separa-se de A e liga-se a C. Desde que C não realizou nenhuma atualização sobre *f*, não existe conflito entre B e C (o vetor de versão de B, < A:2, B:0, C:0>, domina o vetor de versão de C, < A:0, B:0, C:0>), e com isso os vetores de B e C terão o seguinte estado: <A:2, B:0, C:0>. Durante esta nova partição, A atualiza *f* uma vez e C duas vezes tornado os vetores A, B e C da

seguinte forma, respectivamente: <A:3, B:0, C:0>, <A:2, B:0, C:1> e <A:2, B:0, C:1>. Quando essas partições são integradas, é detectado um conflito entre os vetores de versão, uma vez que nenhum dos vetores domina os outros. Além disso, vamos supor que a integração é iniciada no *site* B, então a reconciliação é realizada como mostra o passo 3), mencionado anteriormente. Assim, o vetor final, depois da reconciliação realizada no *site* B, seria <A:3, B:1, C:1>.

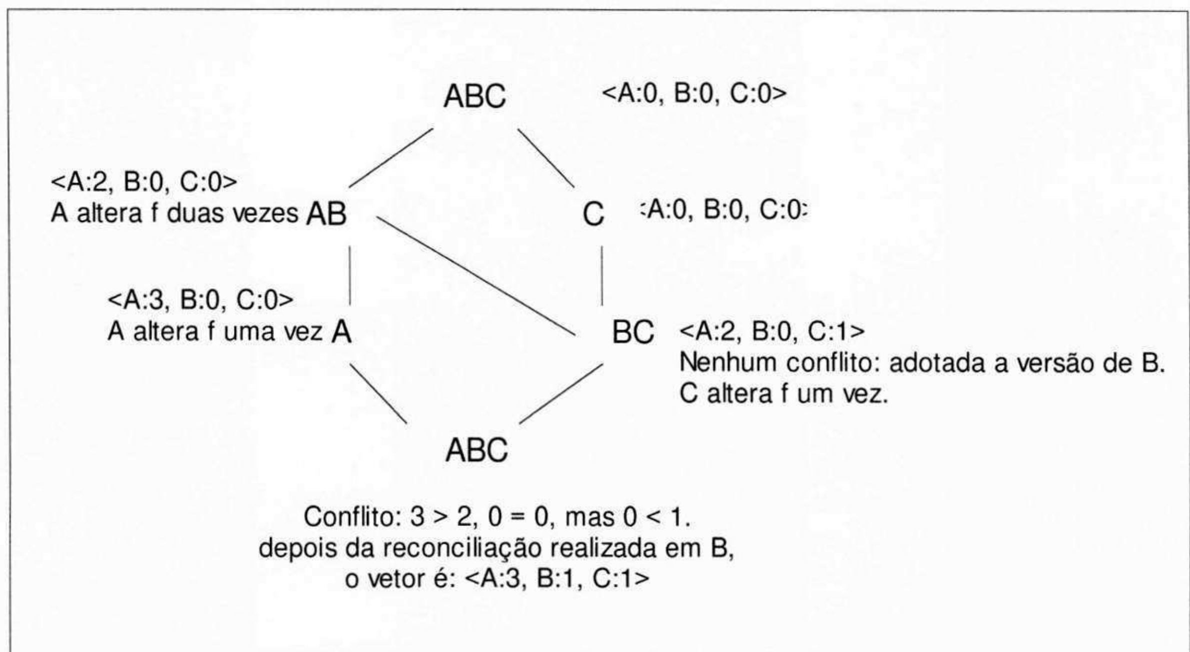


Figura 5 - Conflito detectado por vetores de versão incompatíveis

Apesar deste método oferecer alta disponibilidade, existe a possibilidade dos usuários terem acesso a arquivos com versões não atualizadas, uma vez que este método não oferece nenhum mecanismo que proíbe a leitura de uma cópia de uma determinada informação quando as outras cópias estão sendo atualizadas em outras partições.

Além disso é necessário considerar o tamanho dos vetores de versão, pois sistemas que possuem uma grande quantidade de *sites* e, consequentemente, cópias, terão vetores de versão muito amplos tornando difícil sua implementação.

- Alterações Autônomas [Ceri et al.]

Nesta abordagem as operações de atualização sobre as cópias de informações replicadas são realizadas de forma independente, ou seja, o resultado da operação sobre

uma determinada cópia não interfere na execução da mesma operação sobre outra cópia. Todas as cópias estão disponíveis para leitura e atualização, mesmo na presença de falhas, e as cópias de uma mesma informação constituem um grupo.

Para manter a consistência entre as cópias pertencentes a um mesmo grupo Ceri, Houtsma e Keller utilizaram o protocolo ROWA (*Read One Write All*). Este protocolo permite que operações de leitura sejam realizadas sobre qualquer cópia de uma determinada informação, e requer que as atualizações sejam realizadas em toda as cópias. Utilizar este protocolo não afeta a disponibilidade, uma vez que se ocorrer um particionamento durante uma operação de atualização, a atualização será novamente realizada em uma das partições.

Esta abordagem ainda define que cada *site* mantém um *log* das ações que têm sido aplicadas sobre a informação replicada. Este *log* será utilizado para reconciliar partições. Quando uma reconciliação é executada, os *sites* pertencentes às partições sendo integradas comparam seus *logs*. As ações contidas nos *logs* são ordenadas e as operações já realizadas são removidas. Então, as ações são realizadas sobre todas as cópias pertencentes às partições sendo reconciliadas, produzindo, com isso, um novo estado para as cópias. Depois disto, os *logs* de todas as cópias são atualizados para incluir todas as ações que produziram o novo estado das cópias.

Além do *log*, cada *site* mantém um vetor de recepção. Este vetor armazena o tempo em que uma determinada operação de atualização foi executada. Ele é uma tupla, onde cada elemento representa o tempo em que uma atualização foi realizada sobre a cópia de uma determinada informação replicada armazenada em um dado *site*, ou seja, a i-ésima entrada do vetor (S_i: t_i) armazena o tempo t_i gasto para realizar uma atualização sobre a cópia armazenada no *site* S_i. Assim, quando uma atualização é executada em uma partição, todos os *sites* pertencentes à partição alteram a cópia da informação, o *log* e o vetor de recepção com o tempo de execução da operação. Este tempo é exatamente o mesmo em todos os *sites*, e é determinado pelo o *site* que iniciou a atualização. O vetor de recepção será utilizado também para reconciliar partições.

O algoritmo de reconciliação comporta-se da seguinte forma: para cada partição, escolha um *site* como representante; escolha um dos representantes para ser o coordenador e outro para ser o participante; envie o vetor de recepção do participante para o coordenador; se o vetor do coordenador e do participante são iguais, calcule o novo estado para o vetor de recepção e finalize a reconciliação; caso contrário, compare cada elemento dos vetores de recepção e realize as operações de atualização que ainda não foram realizadas em cada partição a partir das informações contidas no *log*.

Considere o seguinte exemplo: Inicialmente, três *sites* A, B, C armazenam uma cópia de uma conta bancária com valor 1000, todos os *logs* estão vazios, e todos os vetores de recepção são 0. Suponha que eles executam um depósito de 500 na conta X no tempo t_0. Os valores dos itens de dados são: X_A, X_B, X_C = 1500; os vetores de recepção são $VR_A = VR_B = VR_C = [t_0, t_0, t_0]$; e os *log* são H_A, H_B, H_C = < t_0, X, depósito, retirada, 500>, onde a operação ***retirada*** é a operação inversa do ***depósito***. Suponha que ocorre um particionamento e dois grupos são gerados {A} e {B, C}. No *site* A é executado no tempo t_1 um depósito de 1000 na conta X, com isso o valor do item é X_A=2500, o vetor de recepção $VR_A = [t_1, t_0, t_0]$ e o *log* incluirá a operação <t_1, X, depósito, retirada, 1000>. No tempo t_2, um depósito de 100 é realizado nas contas de B e C, assim os valores dos itens serão $X_B = X_C$ = 1600, os vetores de recepção são $VR_B = VR_C = [t_0, t_2, t_2]$ e os *logs* de cada *site* incluirão a operação <t_2, X, depósito, retirada, 100>. Suponha que os *sites* B e C separam-se, e no tempo t_3 os *sites* A e B se reconciliam, assim os valores dos itens de dados serão $X_A = X_B$ = 2600, os vetores de recepção são $VR_A = VR_B = [t_3, t_3, t_2]$ e os *logs* de cada *site* possuem o seguinte estado $H_A = H_B$ = <t_0, X, depósito, retirada, 500>, <t_1, X, depósito, retirada, 1000>, <t_2, X, depósito, retirada, 100>. A reconciliação final entre os *sites* {A, B} e {C} no tempo t_4 produzirá os vetores de versão $VR_A = VR_B = VR_c = [t_4, t_4, t_4]$ e nenhuma mudança será realizada nos bancos de dados e nos *logs* dos *sites* A e B. Entretanto, a alteração que foi realizada no tempo t_1 no *site* A será realizada no *site* C, o banco de dados e o *log* do respectivo *site* serão iguais aos dos *sites* A e B.

Finalizando, as operações de atualização são realizadas utilizando o protocolo

2PC (*Two Phase Commit*). Este protocolo é utilizado para garantir que transações atômicas[5] realizam ou um *commit* ou um *abort*. Durante a primeira fase, o *site* que iniciou a transação (coordenador), envia sua intenção para todos os outros *sites* que armazenam uma cópia da informação replicada (subordinados). Uma vez recebido o pedido, os subordinados respondem com sua decisão, ou seja, se podem realizar o *commit* ou o *abort*. Se todos os subordinados podem realizar o *commit*, o coordenador realiza o *commit* e envia uma mensagem para os subordinados indicando que eles devem fazer o mesmo.

Utilizar o protocolo 2PC para realizar as operações de atualização não compromete a disponibilidade da informação replicada, uma vez que se a transação abortar, porque ocorreu algum particionamento ou alguns *sites* se tornaram indisponíveis, ela será refeita com o novo grupo de *sites* disponíveis.

A vantagem deste método é que ele oferece alta disponibilidade na presença de falha dos *sites* e particionamento da rede, por isso ele é totalmente aplicável em sistemas que não requerem perda de disponibilidade devido a esses tipos de falhas. Por outro lado, se ocorre alguma falha em alguns dos *sites* ou particionamentos na rede durante a execução de uma transação, é necessário que ela seja refeita sobre o novo conjunto de *sites,* além disso, inconsistências podem ser geradas durante a execução de atualizações concorrentes uma vez que não é utilizado o protocolo 2PL para garantir a isolação das atualizações.

3.3. Discussão sobre as Abordagens

Na seção anterior descrevemos vários trabalhos existentes sobre soluções para o problema de consistência de informações replicadas. Cada um deles possui desvantagens e, dependendo do sistema em que são utilizados, podem ser ineficientes.

---

[5]Uma transação é considerada atômica se ela ou é realizada por completo ou não é realizada [Coulouris et al. 1996].

O método *Partição Majoritária*, por exemplo, realiza atualizações apenas na partição majoritária, reduzindo com isso, a disponibilidade das informações replicadas. Por outro lado, o método *Vetores de Versão* oferece alta disponibilidade, mas é inapropriado em sistemas que necessitam disponibilizar uma grande quantidade de cópias de uma determinada informação replicada.

O método *Log Transformations* requer a intervenção do usuário para reparar partições. Já o método *Votação Dinâmica* as reparações são realizadas sem a intervenção do usuário; por outro lado, ele reduz a disponibilidade durante reparações, pois é necessário bloquear todas as cópias para reparar as partições.

O método *Weighted Voting* aumenta a sobrecarga de comunicação, pois necessita de um quorum para realizar as operação de leitura e atualização. Os métodos *Optimistic Protocol* e *Alterações Autônomas* necessitam refazer as transações quando ocorrem colisões e falhas nos nodos ou particionamentos, respectivamente. Finalizando, o método de validação em duas fases reduz a disponibilidade dos nodos enquanto uma atualização está em execução.

Além disso, pudemos perceber também que as abordagens otimistas permitem que inconsistências sejam geradas para oferecer alta disponibilidade. Por outro lado, as abordagens pessimistas reduzem a disponibilidade para evitar inconsistências.

Diante destas considerações, optamos por utilizar em nosso projeto as duas abordagens, otimistas e pessimistas; e, além disso, escolhemos algumas características de alguns dos métodos, aqui descritos, para desenvolver um esquema de gerência de recursos *Web* espelhados que garanta a consistência dos recursos *Web* espelhados e minimize alguns dos problemas acima descritos.

Capítulo 4

Projeto de um Suporte para Atualização de Recursos Espelhados na *Web*

4.1. Introdução

Como mencionado na seção 2.5, as atualizações dos recursos espelhados podem ser realizadas através dos métodos PUT e DELETE; para isto, é necessário verificar como o processamento desses métodos modifica o estado do servidor que os executa, para que as cópias dos recursos se mantenham consistentes; além disso, é necessário garantir que todos os servidores que armazenam os recursos espelhados processem todas as requisições contendo esses métodos, e que elas sejam processadas na mesma ordem.

Neste capítulo apresentaremos nossa proposta que visa exatamente fornecer um esquema para a gerência de recursos *Web* espelhados, permitindo a manutenção de semânticas de atualizações otimista e pessimista [Davidson et al. 1985], dependendo das necessidades do recurso espelhado. Definiremos também um serviço de autenticação de gerentes e de recursos, que permite que os gerentes de um determinado recurso realizem alterações sobre os servidores que o armazenam.

O capítulo está estruturado da seguinte forma. Na seção 4.2 apresentaremos os

mecanismos de suporte para a gerência dos recursos *Web* espelhados, discutiremos as duas semânticas de consistência disponíveis para os recursos espelhados e o impacto que cada uma delas pode causar na implementação dos métodos PUT e DELETE. Na seção 4.3 apresentaremos o esquema de autenticação de gerentes utilizado durante as operações de atualização dos recursos *Web* espelhados.

4.2. Gerência de Recursos *Web* Espelhados

Os pedidos de atualização sobre um determinado recurso *Web* espelhado devem ser realizados de maneira que sua consistência seja preservada. Para isso, desenvolvemos um esquema de gerência de recursos *Web* espelhados, onde as atualizações sobre os recursos são realizadas de forma concorrente e sem conflitos. Além disso, este esquema pode utilizar mecanismos de controle de consistência otimista e pessimista durante as atualizações sobre os recursos espelhados, dependendo do recurso.

Para resolvermos os problemas de inconsistências gerados por atualizações de recursos espelhados é necessário que tanto o *proxy*, definido por Fonsêca em seu trabalho [Fonsêca 1998], quanto os servidores que armazenam os recursos espelhados, colaborem entre si no processamento da atualização. Assim, utilizamos as facilidades de um *gateway* para realizar qualquer funcionalidade adicional requerida dos servidores (Figura 6). O *gateway* atua como uma camada superior a dos servidores que mantêm os recursos espelhados, e é responsável por processar os pedidos de atualização de forma que a consistência dos recursos seja preservada.

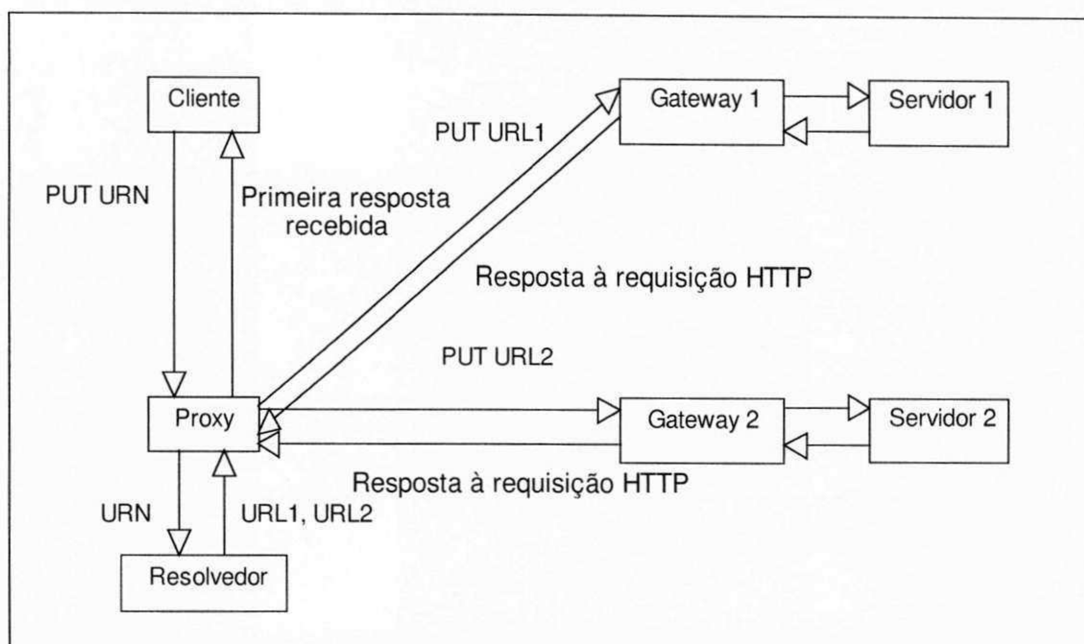


Figura 6 - Arquitetura para Atualizações de Recursos Web Espelhados

Como mostra a figura 6, cada pedido de atualização solicitado por um gerente de um determinado recurso espelhado é interceptado pelo *proxy*, que recorre a um resolvedor para obter a localização das cópias do recurso. De posse das localizações das cópias do recurso, o *proxy* envia a mensagem de requisição, contendo o método PUT ou DELETE, para todos os servidores que mantêm cópias do recurso espelhado. Ao receber as mensagens de requisição, o *gateway* processa-as e envia uma resposta para o *proxy* que solicitou o pedido de atualização.

4.2.1. Modelo dos Recursos *Web* Espelhados

Em nosso projeto, definimos alguns atributos que dão suporte aos recursos *Web* espelhados. Cada recurso espelhado possui os seguintes atributos: o mais recente estado (versão); o histórico das transações executadas (*log*); a semântica de consistência (pessimista ou otimista), um *flag* que indica o estado da mais recente atualização (realizado, cancelado, esperando e desistente) e o nome dos gerentes, que são os administradores do recurso. Além destes atributos, definimos um nível de prioridade entre os gerentes de um mesmo recurso, uma vez que cada recurso *Web* espelhado pode ter mais de um gerente. O *log* contém o nome de quem realizou a mais recente atualização (gerente do recurso) e a localização da cópia do recurso antes de ter sido

realizada a mais recente atualização. Por outro lado, quando o *flag* possui o estado realizado ou cancelado indica que a requisição foi realizada ou cancelada, respectivamente. O estado esperando indica que o *gateway* está esperando por um pedido de *commit* ou *abort*. O estado desistente indica que o *gateway* esperou pelo pedido de *commit* ou *abort* em um tempo finito e está consultando outros servidores que mantêm cópias do referente recurso para adquirir seu estado. Assim, tanto o estado contido no *flag* quanto as informações armazenadas no *log* serão utilizadas pelo *gateway* caso ocorra alguma falha durante as operações de atualizações sobre recursos com semântica de consistência pessimista.

Esses atributos serão utilizados para garantir a consistência dos recursos espelhados durante as operações de atualização mesmo na presença de falhas na máquina servidora ou particionamento da rede.

Além disso, a escolha do tipo de semântica de consistência associada ao recurso depende única e exclusivamente do gerente, ou seja, o gerente é quem decide que tipo de semântica de consistência o recurso espelhado deve ter, de acordo com a importância das informações nele contidas, no momento de sua criação. Recursos que possuem semântica de consistência otimista são atualizados independente do número de cópias disponíveis. Por outro lado, atualizações sobre os recursos com semântica de consistência pessimista só serão realizadas se todas as suas cópias estiverem disponíveis.

4.2.2. Características dos Pedidos de Atualização

Pedidos de atualização solicitados por gerentes de recursos espelhados são realizados através de mensagens de requisição enviadas do cliente para o servidor. As mensagens de requisição dos métodos DELETE possuem cabeçalhos que contêm as informações para autenticação do gerente (nome e senha).

As mensagens de requisição do método PUT contêm, além da entidade que representa os dados do recurso a ser armazenado, a versão do recurso e alguns cabeçalhos com as mesmas informações mencionadas anteriormente. Quando o recurso

não existe, e está sendo criado pela requisição PUT, o cabeçalho da mensagem de requisição deve conter também os endereços dos servidores que armazenarão as cópias do recurso e sua semântica de consistência. Assim, antes de realizar a atualização, o *proxy* deve incluir na base de dados do resolvedor os endereços dos servidores que armazenarão as cópias do recurso; para isso, o resolvedor deve permitir a atualização dinâmica de sua base de dados [Vixie et al. 1997, Eastlake 1997].

Ao receber uma determinada requisição de um método PUT ou DELETE, o servidor pode realizar ou não a requisição; sua decisão vai depender das seguintes condições:

- Requisição PUT

- Quando a requisição PUT está sendo utilizada para criar um recurso já existente, é retornado para gerente uma mensagem de erro indicando a não realização do pedido; caso contrário, o servidor executa a requisição e retorna a mensagem 201 (Criado);

- Quando a requisição PUT está sendo utilizada para atualizar um recurso não existente, é retornado para o gerente a mensagem 404 (Não encontrado); caso contrário, o servidor executa a requisição e retorna a mensagem 200 (Ok) indicando que a atualização foi realizada com sucesso; por fim, se o recurso não pode ser atualizado por quaisquer outros motivos, é enviado para o gerente uma mensagem de erro indicando o motivo da sua não realização.

- Requisição DELETE

- Quando a requisição DELETE está sendo realizada sobre um recurso existente, o servidor executa a requisição e retorna para o gerente a mensagem 200 (Ok); caso contrário, o servidor retorna a mensagem 404 (Não encontrado).

4.2.3. Atualização dos Recursos *Web* Espelhados

Os pedidos de atualização dos recursos *Web* espelhados podem ser solicitados

por qualquer gerente de um determinado recurso espelhado. O gerente do recurso deve informar os atributos necessários para que os métodos PUT ou DELETE sejam realizados, tais como: informações de autenticação; a URN; e, se o recurso espelhado está sendo criado, a localização do servidores que armazenarão as cópias do recurso e a sua semântica de consistência. Uma vez informados os atributos do recurso, o pedido de atualização é enviado para o *proxy* para que ele seja processado. Se o recurso não está sendo criado, ao receber o pedido, o *proxy* consulta o resolvedor para obter a localização das cópias do recurso e consulta um dos servidores que mantém uma cópia do recurso para obter a semântica de consistência do recurso e sua versão. Por fim, o pedido de atualização é enviado na forma de uma mensagem de requisição, como especificado na seção anterior.

- *Atualização de Recursos com Semântica de Consistência Otimista*

As atualizações sobre os recursos com semântica de consistência otimista são realizadas em transações independentes, ou seja, o processamento de uma requisição sobre uma determinada cópia do recurso espelhado não interfere no seu processamento sobre outras cópias do recurso. Assim, nesta abordagem, o processamento da requisição será realizado independente do número de cópias disponíveis.

Para garantir essa funcionalidade, o processamento dos comandos PUT e DELETE é realizado da seguinte forma: após a resolução da URN, o *proxy* envia as requisições para todos os servidores que armazenam uma cópia do recurso e espera por uma resposta durante um período de tempo finito; se algum servidor não estiver disponível, o *proxy* atualiza uma lista de requisições pendentes incluindo a requisição não realizada, que é processada de tempos em tempos em segundo plano.

As requisições do gerente são recebidas pelos *gateways;* cada *gateway* verifica inicialmente a autenticidade e as permissões do gerente que requisitou a operação; se o gerente tem permissões para executar o método, o *gateway* processa a requisição, atualiza a versão e envia uma resposta para o *proxy* indicando que a requisição foi processada; caso contrário, o *gateway* envia para o *proxy* uma mensagem de erro; se o

proxy recebe alguma mensagem de erro ou não recebe a resposta em um tempo finito, ele atualiza a lista de requisições pendentes incluindo a requisição não realizada; por outro lado, se o *proxy* recebe pelo menos uma mensagem indicando que a requisição foi processada, ele envia uma mensagem para o gerente informando que o processamento do método foi realizado.

Em nosso projeto, a lista de pendências é processada pelo *proxy* que, de tempos em tempos, analisa as requisições contidas na lista de pendências e as processa, em segundo plano, sobre a(s) cópia(s) do recurso espelhado não atualizada(s).

As operações realizadas nas bases de dados dos servidores *Web* que mantêm recursos espelhados poderão ser executadas de forma concorrente, desta forma, o acesso a um arquivo em uma mesma base de dados pode causar interferências. Essas interferências podem ser facilmente evitadas através de algum mecanismo de controle de concorrência.

Em nosso projeto cada servidor que mantêm recursos espelhados gerencia localmente os acessos concorrentes aos seus recursos através do mecanismo de controle de concorrência 2PL (*two phase lock*) [Bernstein et al. 1987]. Este método de bloqueio possui duas fases bem evidentes. Na primeira fase, todos os pedidos de bloqueio são adquiridos; enquanto que na segunda fase, eles são liberados. Assim, ao receber uma requisição, o *gateway* bloqueia o recurso identificado na requisição, realiza o seu processamento e libera o bloqueio.

Mesmo com a utilização do mecanismo de controle de concorrência descrito acima, as atualizações concorrentes sobre um recurso pode ainda gerar colisões, uma vez que, os *gateways* não esperam pelo fim da transação global para liberar os bloqueios. Por outro lado, colisões não são geradas quando cada recurso espelhado possui apenas um gerente, pois as atualizações sobre um recurso espelhado só serão realizadas por um gerente. Além disso, no caso de particionamentos, as atualizações são realizadas apenas em uma partição, ou seja, na partição que possui o *proxy* que iniciou a atualização.

O tratamento de colisões será descrito na seção posterior, antes apresentaremos o mecanismo utilizado em nosso projeto para manter a consistência de recursos com semântica de consistência pessimista.

• *Atualização de Recursos com Semântica de Consistência Pessimista*

As atualizações sobre os recursos com semântica de consistência pessimista são realizadas de forma atômica, ou seja, ou todas as cópias do recurso espelhado são atualizadas ou nenhuma delas é atualizada. Para garantir essa funcionalidade, utilizamos o protocolo 2PC (*Two Phase Commit*) [Coulouris et. al. 1996]. Este protocolo é executado em duas fases. Na primeira fase, o nodo que iniciou uma operação envia um pedido de *commit* para todos os outros nodos. Cada nodo envia uma resposta se quer ou não realizar o *commit*. Na segunda fase, o nodo que iniciou a operação coleta as respostas de todos os nodos, se todos os nodos querem realizar o *commit*, o nodo envia um COMMIT para todos os nodos, caso contrário, ele envia um ABORT; após adquirir a decisão, cada nodo realiza o COMMIT ou o ABORT e envia a confirmação para o nodo que iniciou a operação.

Mesmo com a utilização deste protocolo e o mecanismo de controle de concorrência 2PL, descrito anteriormente, inconsistências podem ser geradas durante as operações de atualização. Para evitar que inconsistências sejam geradas, é necessário garantir que os bloqueios só sejam liberados quando todos os *gateways* finalizem a atualização. Assim, para garantir essa funcionalidade, utilizamos o mecanismo de controle de concorrência 2PL estrito distribuído, descrito na seção 3.2.1. A característica deste mecanismo é que os bloqueios só são liberados, na segunda fase, quando se tem certeza que todos nodos realizaram uma determinada operação.

Por outro lado, se ocorrerem falhas durante a execução de uma operação de atualização, existe a possibilidade dos recursos que estavam sendo atualizados ficarem bloqueados indefinidamente. Para evitar esse problema, tanto o *proxy* quanto o *gateway* esperam por respostas durante um tempo finito. Assim, se o *proxy* não receber uma resposta do *gateway* indicando que ele quer ou não realizar o *commit* em um tempo

finito, o *proxy* pode decidir abortar a operação de atualização. Por outro lado, se o *gateway* não receber o *commit* ou o *abort* em um tempo finito, ele deve consultar outros servidores para decidir o que fazer.

Assim, o processamento dos comandos PUT e DELETE é realizado da seguinte forma: após a resolução da URN, o *proxy* envia as requisições para todos os servidores que armazenam uma cópia do recurso e espera por uma resposta durante um período de tempo finito (Figura 7). Após a verificação da autenticidade e das permissões do gerente, cada *gateway* adquire os bloqueios necessários para processar a requisição, processa a requisição, e atualiza o *flag* do recurso atribuindo-lhe o estado esperando e um *log* com as operações que devem ser realizadas sobre o recurso; em seguida o *gateway* envia uma resposta para o gerente informando que está pronto para realizar o *commit*; quando o *proxy* recebe essa mensagem de todos os *gateways*, ele envia uma nova requisição para todos os *gateway* solicitando que o *commit* seja realizado; só então o *gateway* realiza o *commit*, atualiza a versão do recurso e o *flag* atribuindo-lhe o estado realizado, libera os bloqueios adquiridos na primeira fase, remove a requisição do seu *log* e envia uma confirmação para o *proxy*.

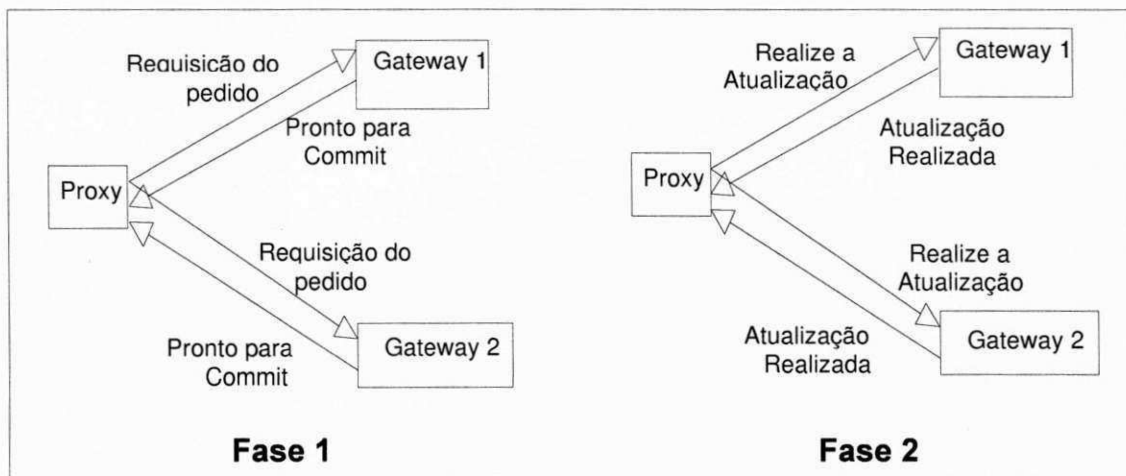


Figura 7 - Atualização Atômica Finalizada

Se pelo menos um *gateway* não responder dentro de um tempo finito ou enviar uma mensagem indicando que não deseja realizar o *commit*, o *proxy* envia uma mensagem que causa o cancelamento da operação para todos os *gateways* que estão prontos para realizar o *commit* (Figura 8). Finalizando, ao receber mensagens de

cancelamento, o *gateway* atualiza o *flag* do recurso atribuindo-lhe o estado cancelado, libera os bloqueios adquiridos na primeira fase e remove a requisição do seu *log*.

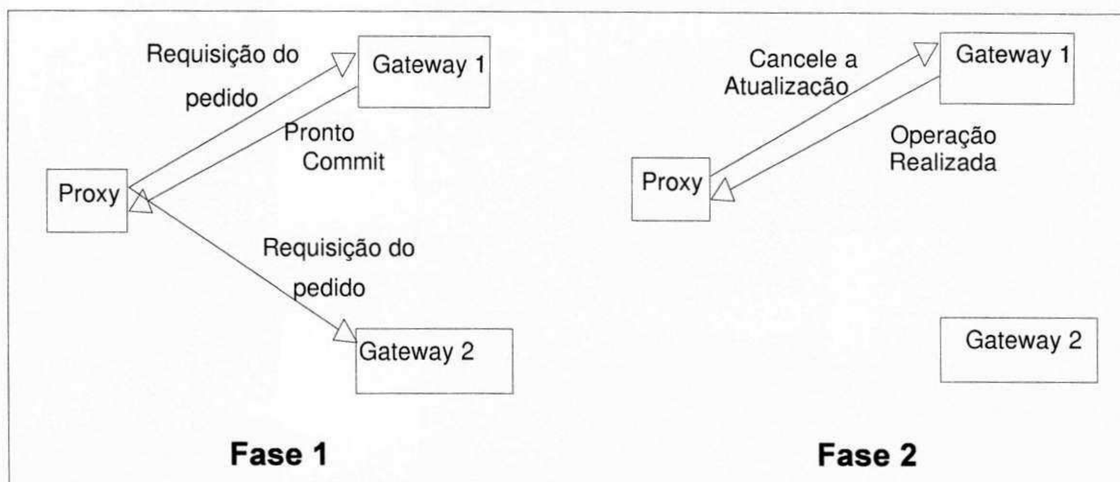


Figura 8 - Atualização Atômica Cancelada

Considerando que em casos de particionamentos e falhas tanto do *proxy* quanto do *gateway*, eles são recuperados em um tempo finito; se o *proxy* receber uma confirmação de pelo menos um *gateway* indicando que ele realizou o *commit,* o *proxy* pode finalizar a atualização, pois se algum *gateway* não recebeu o *commit* em um tempo finito, ele atualiza o *flag* do recurso sendo atualizado atribuindo-lhe o estado desistente e consulta outros servidores para decidir o que fazer, ou seja, para decidir se realiza o *commit* ou o *abort*. As falhas tanto do *proxy* quanto do *gateway* são tratadas quando esses componentes se recuperam e processam seus *logs*. O processamento dos *logs* será descrito com mais detalhes na próxima seção. Além disso, trataremos também dos problemas gerados durante as atualizações dos recursos na presença de particionamentos da rede.

4.2.3.1. ATUALIZAÇÕES NA PRESENÇA DE FALHAS

4.2.3.1.1. ATUALIZAÇÕES OTIMISTAS NA PRESENÇA DE FALHAS

Mesmo com a utilização do mecanismo de controle de consistência 2PL, descrito anteriormente, as atualizações sobre os recursos com semântica de consistência otimista podem gerar colisões. Como descrito na seção 3.2, uma colisão ocorre quando o mesmo recurso, que é fisicamente espelhado em dois ou mais servidores, é alterado

durante um período de latência assíncrono, ou seja, após o tempo em que a primeira alteração aconteceu, uma segunda alteração ocorre e é processada em um servidor antes que a propagação da primeira alteração tenha sido concluída. A Figura 9 abaixo descreve como este processo ocorre.

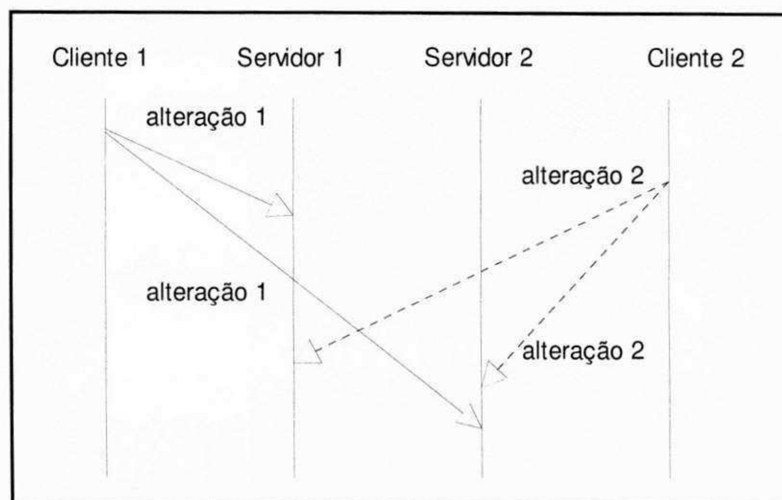


Figura 9 - Conflito de Transações em Servidores Espelhados

Como mostra a figura acima, ocorreu um conflito entre as alterações 1 e 2, pois a *alteração 1* foi iniciada antes da *alteração 2* e deveria ser processada nos dois servidores nesta mesma ordem. Entretanto, a *alteração 2* é processada no *servidor2* antes da *alteração 1*.

Uma colisão pode acontecer também se a atualização de um recurso espelhado com semântica de consistência otimista for realizada na presença de particionamentos da rede, pois os recursos que possuem mais de um gerente podem ser atualizados em partições distintas e, com isso, é possível que, quando as partições são integradas e os *proxys* de cada partição executem sua lista de pendências, as cópias dos recursos sejam atualizadas em uma ordem distinta.

Para evitar que colisões ocorram é necessário garantir que atualizações concorrentes ocorram na mesma sequência sobre todas as cópias do recurso espelhado. Assim, em nosso projeto nós permitimos que as colisões ocorram, mas evitamos que, após o término das atualizações concorrentes e a integração das partições, as cópias fiquem inconsistentes.

As colisões são detectadas pelo *gateway* da seguinte forma: ao receber a requisição, o *gateway* verifica se a versão do recurso contida na requisição é compatível com a versão do recurso mantida pelo servidor; se a versão estiver compatível, a operação é realizada; caso contrário é detectado uma colisão. Os problemas de colisões são resolvidos da seguinte forma: se o conjunto de cópias de um recurso espelhado possui mais de um gerente, é definido um certo grau de prioridade entre eles; desta forma, quando o *gateway* detectar alguma colisão, a atualização válida será a que foi realizada pelo gerente de maior prioridade, sendo as outras atualizações descartadas.

Portanto, antes de realizar uma operação de atualização sobre um determinado recurso espelhado, o *gateway* deve verificar se a versão do recurso contida na requisição é equivalente a versão do recurso mantida no servidor; se for detectado uma incompatibilidade entre as versões, o *gateway* verifica quem realizou a mais recente atualização; se a atualização foi realizada por um gerente de maior prioridade, o *gateway* cancela a atualização e envia para o *proxy* uma mensagem indicando que não foi possível realizar a atualização, pois a versão adquirida esta obsoleta; caso contrário, o *gateway* realiza a atualização descartando, com isso, a última atualização realizada e envia para o *proxy* uma mensagem indicando que a atualização foi realizada com sucesso.

Essas considerações acima descritas podem ser melhor ilustradas pelo seguinte exemplo: Considere que o recurso espelhado X está espelhado em três servidores, denominados servidor 1, servidor 2 e servidor 3; além disso, considere que ele possui dois gerentes, denominado gerente 1 e gerente 2, e que o gerente 1 tem maior prioridade que o gerente 2. Suponha que o gerente 1 inicia uma atualização 1 utilizando o *proxy* 1, enquanto que, o gerente 2 inicia uma atualização 2 utilizando o *proxy* 2 (Figura 10). Além disso, suponha também que os gerentes estão com a mesma versão do recurso, e que os servidores recebem os pedidos na seguinte sequência:

Servidor 1 e 2: recebe a atualização 1 e depois a atualização 2;

Servidor 3: recebe a atualização 2 e depois a atualização 1.

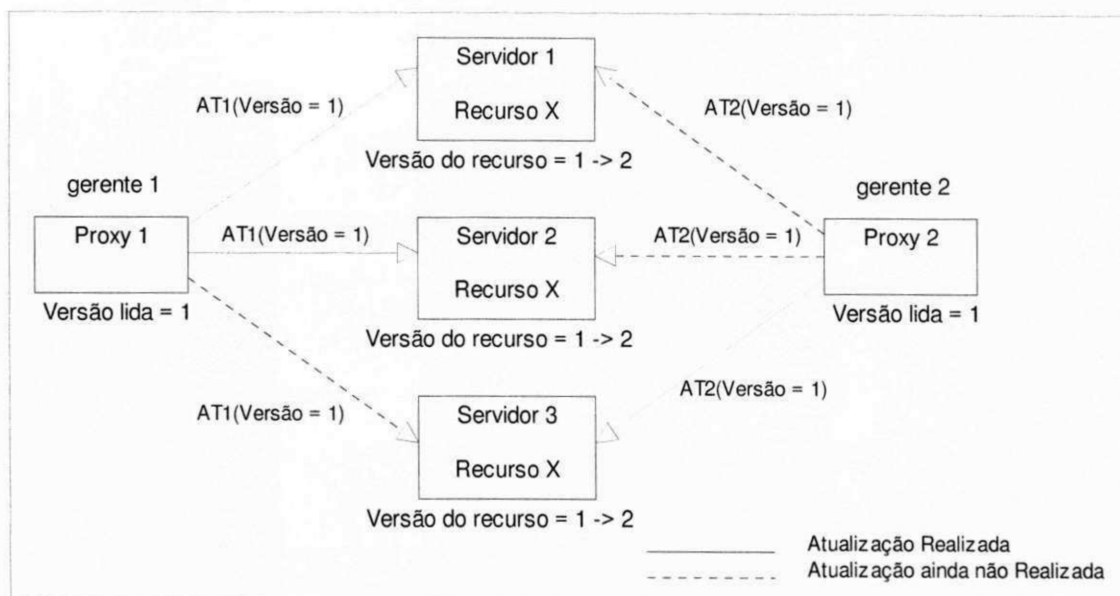


Figura 10 - Exemplo de Colisão em Atualizações Concorrentes

Como mostra a figura 10, o *gateway* do servidor 1 e do servidor 2 incrementa a versão do recurso X ao receber a atualização 1, e o *gateway* do servidor 3 incrementa a versão do recurso X ao receber a atualização 2. Assim, ao receber o pedido de atualização 2, o *gateway* do servidor 1 detecta uma colisão, pois a versão contida na requisição não é igual a versão mantida no servidor; em seguida, o *gateway* verifica a prioridade entre os gerentes 1 e 2, como o gerente 1 possui maior prioridade que o gerente 2, a atualização válida será a atualização 1, enquanto que a atualização 2 é cancelada. O mesmo processo ocorre com o servidor 2. No servidor 3, o *gateway* recebe os pedidos na ordem inversa, mas o resultado será o mesmo, pois a atualização 2 é descartada, já que, a atualização válida é a atualização 1.

No caso de particionamentos na rede, as colisões são detectadas quando as partições são integradas. Considere o exemplo descrito anteriormente. Suponha que ocorre um particionamento e, com isso, o servidor 3 desliga-se dos servidores 1 e 2. Suponha também que a atualização 1, é realizada apenas nos servidores 1 e 2, e a atualização 2 é realizada apenas no servidor 3. Assim a lista de pendências dos *proxys* 1 e 2 ficarão da seguinte forma:

lista de pendências do proxy 1: realize uma atualização no recurso X do servidor 3;

lista de pendências do proxy 2: realize uma atualização no recurso X dos servidores 1 e 2.

Assim, quando as partições são integradas, as requisições pendentes são realizadas e, consequentemente, os conflitos são detectados e resolvidos como descrito anteriormente.

Além dos problemas de colisões e particionamentos acima descritos, existe a possibilidade de ocorrerem falhas tanto no *proxy* quanto no *gateway* durante o processamento das operações de atualização otimista. Se o *proxy* falha antes de concluir o envio das operações de atualização para todos os servidores que mantêm cópias do recurso, algumas cópias podem não ser atualizadas.

As falhas do *proxy* são tratadas quando ele se recupera e processa seu *log*. Assim, é necessário que ao se recuperar de uma falha, o *proxy* analise seu *log* e verifique quais cópias do recurso espelhado não foram atualizadas; uma vez obtidas essas informações, ele envia as operações de atualização para as cópias não atualizadas.

Se o *gateway* falha antes de enviar uma mensagem indicando que a atualização foi realizada, o *proxy*, depois de esperar pela resposta do *gateway* em um tempo finito, inclui a atualização na lista de pendências. Como as atualizações que estavam sendo realizadas pelo *gateway* vão ser processadas pelo *proxy*, o *gateway* não necessita tratar sua falhas. Por outro lado, quando o *proxy* tenta realizar a operação novamente, existe a possibilidade dela já ter sido realizada, pois *gateway* pode ter realizado a operação e falhado antes de enviar a resposta para o *proxy*. Assim, para evitar que a operação seja executada novamente, o *gateway* deve analisar as versões e os gerentes, como descritos anteriormente, mas se última atualização foi realizada pelo gerente da atualização corrente, o *gateway* conclui que a cópia do recurso já foi atualizada e, com isso, envia uma resposta para o *proxy* indicando que está cópia já foi atualizada.

A vantagem desta abordagem é que as partições são reparadas sem a intervenção dos gerentes, pois os *proxys* que realizaram atualizações durante os particionamentos se encarregam de atualizar os recursos espelhados não atualizados através das informações

contidas nas listas de pendências. Além disso, ela oferece alta disponibilidade, pois permite que as atualizações sejam realizadas em qualquer partição. Por outro lado, usuários podem ter acesso a recursos ainda não atualizados, pois os bloqueios realizados sobre cada cópia podem ser liberados antes do fim da transação global.

4.2.3.1.2. ATUALIZAÇÕES PESSIMISTAS NA PRESENÇA DE FALHAS

Como descrito anteriormente, para evitar que os recursos fiquem bloqueados indefinidamente quando ocorrem particionamentos e falhas no *proxy* ou no *gateway* durante atualizações dos recursos espelhados, tanto o *proxy* quanto o *gateway* esperam por respostas durante um tempo finito. Assim, se o *proxy* não receber uma confirmação dos *gateways* indicando que eles realizaram o *commit*, existe a possibilidade deles terem realizado o *commit* ou não. Neste caso, o *proxy* pode enviar o pedido novamente, pois se os *gateways* realizaram o *commit*, eles, ao adquirir a requisição, constatarão que a requisição já foi processada, e com isso enviarão para o *proxy* uma resposta indicando que esta requisição realizou o *commit.* Por outro lado, se pelo menos um *gateway* enviou a confirmação para o *proxy*, ele pode finalizar a atualização, pois os *gateways* que não enviaram a confirmação, se não realizaram o *commit*, consultarão outros servidores que mantêm cópias do recurso e, consequentemente, realizarão o *commit*. Se ocorrerem particionamentos durante o envio dos pedidos de *abort*, o *proxy* finaliza a atualização, pois os *gateways* que não receberam o *abort* irão também consultar outros servidores e, consequentemente, realizarão o *abort*.

Ao consultar um dos servidores, o *gateway* adquire o estado contido no *flag* do recurso. Assim, se o estado contido no *flag* é realizado, o *gateway* realiza o *commit*; se o estado é cancelado, o *gateway* realiza o *abort*; se o estado é esperando, o *gateway* consulta outros servidores que mantêm cópias do recurso; e se todos as cópias do recurso possuem o estado desistente; o *gateway* decide abortar, pois todos os *gateways* estão consultando uns aos outros. Porém, se apenas um *gateway* realizar o *commit* e, em seguida, ele receber uma nova requisição sobre o mesmo recurso; o *proxy* que iniciou está nova requisição realizará o *abort*, pois as cópias do recurso estão bloqueadas já que a mais recente atualização não foi concluída. Considere que este *gateway* não recebe o

abort. Assim, o *flag* deste recurso terá o estado esperando, e quando os outros *gateways* consultarem esta cópia para adquirir o estado da requisição que realizou o *commit*, eles adquirirão um estado inconsistente gerando, com isso, uma inconsistência. Desta forma, ao consultar os outros servidores para decidir o que fazer, o *gateway* deve adquirir, além do estado contido no *flag*, a versão do recurso e o nome do gerente que realizou a mais recente atualização.

Assim, se as versão são iguais, o *gateway* analisa o estado do *flag*, pois isto significa que o *gateway* consultado está processando a requisição corrente. Por outro lado, se as versões são iguais, o estado do flag é realizado e o *gateway* está processando uma requisição do método PUT sobre um recurso já existente, ele não deve realizar o *commit*, pois, provavelmente, o *gateway* deste servidor consultado não recebeu está requisição, já que, se este *commit* fosse desta requisição a versão adquirida seria maior. Além disso, se for uma requisição do método DELETE, o *gateway* realiza o *abort*, pois isto significa que o recurso não foi excluído.

Se a versão adquirida é maior do que a versão da cópia do recurso sendo atualizado, o *gateway* verifica que tipo de atualização está sendo realizada; se for uma requisição do método PUT sobre um recurso já existente, o *gateway* realiza o *commit*, pois isto significa que o servidor consultado realizou o *commit* da requisição. Se for uma requisição do método PUT de um recurso não existente, o *gateway* verifica o valor da versão adquirida e o nome do gerente adquirido; se a versão é igual a um e os gerentes são compatíveis, o *gateway* realiza o *commit*, pois ao seu incluído, a versão do recurso é um; caso contrário, ele realiza o *abort*.

Se a versão adquirida é menor, o *gateway* verifica também que tipo de atualização está sendo realizada; se for uma requisição do método PUT sobre um recurso já existente; o *gateway* realiza o *abort*, pois isto significa que o servidor consultado não realizou o *commit* de uma outra requisição do método PUT e está consultado outros servidores que mantêm cópias do recurso. Se for uma requisição do método DELETE, o *gateway* verifica o valor da versão; se for igual a zero, significa que o *gateway* do servidor consultado está tentando concluir a inclusão do recurso, então o

gateway espera durante um tempo finito para realizar a consulta novamente; se a versão for maior que zero, o *gateway* realiza o *abort*; por outro lado, se ao consultar o *gateway* o recurso não existe, o *gateway* realiza o *commit*.

Além dos problemas gerados pelos particionamentos durante o processamento das requisições PUT e DELETE sobre os recursos espelhados, existe a possibilidade de ocorrerem falhas tanto no *proxy* quanto no *gateway*. Se o *proxy* falhar antes de enviar o *commit* ou o *abort*, o *gateway*, provavelmente, realizará o *abort* quando o tempo esgotar. Por outro lado, se o *proxy* falhar durante o envio dos pedidos de *commit*, existe a possibilidade de algumas cópias receberem o *commit* e outras não.

Assim, ao se recuperar da falha, o *proxy* analisa seu *log*, se o *proxy* falhou durante o envio do *commit*, ele realiza o mesmo processamento que foi realizado quando o tempo esgota na espera da confirmação de um *commit*, como descrito anteriormente.

Para se recuperar de falhas, o *gateway* analisa seu *log* para verificar em que instante a atualização do recurso foi interrompida; se o *gateway* falhou antes de receber o *commit* ou o *abort*, o *gateway* consulta um dos servidores que mantêm cópias do recurso espelhado para obter o ser estado, fazendo as mesmas considerações descritas anteriormente; se o servidor estiver indisponível, o *gateway* consulta outro servidor e assim por diante.

4.3. Autenticação de Gerentes de Recursos

Para garantir que os métodos PUT e DELETE só sejam executados por gerentes dos respectivos recursos, utilizamos um método baseado em senhas e criptossistema de chave pública.

Cada gerente, ao solicitar um método PUT ou DELETE, deve incluir na mensagem de requisição seu nome e senha, que foram previamente cadastrados em todos os servidores que mantêm cópias dos recursos espelhados. Antes da mensagem de requisição ser enviada, a senha do gerente é criptografada com a chave pública do

servidor que receberá o pedido. Ao receber a mensagem de requisição, o servidor decriptografa a senha do gerente com sua chave privada e a compara com a senha armazenada em sua base de dados. A execução do método só é realizada se o servidor puder validar o nome e a senha do gerente.

Além disso, utilizamos um esquema de distribuição de chaves, onde as chaves públicas de cada servidor *Web* que mantém os recursos espelhados são armazenadas nas bases de dados do resolvedor. Assim, a distribuição de chaves será gerenciada pelo resolvedor. Isto permite que o resolvedor seja usado como um mecanismo de distribuição de chaves [Kaufman e Eastlake 1997].

As informações de autenticação são enviadas em um cabeçalho denominado *Autorização*, definido pelo protocolo HTTP. O cabeçalho *Autorização* consiste de credenciais contendo as informações de autenticação do gerente. O formato deste cabeçalho, em nosso esquema de autenticação, é definido da seguinte forma:

Autorização = "Autorização" ":" **credenciais**

credenciais = "Autent-Rec-Espelhado" **pedido-autenticação**

pedido-autenticação = **<nome-gerente** ":"**domínio** ":"**senha-criptografada>**

senha-criptografada = $\{senha\}_{PKs}$

onde: **nome-gerente** - cadeia de caracteres;

domínio - define o espaço de proteção do esquema. Através deste campo o *gateway* identifica qual esquema de autenticação utilizar. Em nosso esquema, o domínio é a cadeia de caracteres *RecursosEspelhados*;

$\{senha\}_{PKs}$ - senha criptografada com a chave pública do servidor;

Quando o servidor que mantém um recurso espelhado não aceita as credenciais enviadas na mensagem de requisição, ele envia para o *proxy* uma mensagem 401 (não autorizado) e um cabeçalho WWW-*Authenticate* que contém o tipo de esquema de autenticação aplicado ao recurso. Este cabeçalho é definido da seguinte forma:

WWW-*Authenticate* = "WWW-Autenticação" ":" "Autent-Rec-Espelhado"
recusa
recusa = "domínio" "=" **domínio**;

Através desse domínio o *proxy* identifica qual foi o esquema de autenticação utilizado. Por exemplo, considere a seguinte urn: *urn:wmr:dsc.ufpb.br/alunos/raquel.html*. Assuma que o nome do gerente é a cadeia de caracteres *raquel* e sua senha é a cadeia de caracteres *raquel*. Assim, para atualizar o recurso *raquel.html*, o *proxy* enviaria uma mensagem de requisição com o seguinte cabeçalho:

Autorização: Autent-Rec-Espelhado raquel:RecursosEspelhados:$\{raquel\}_{PKs}$

Se o gerente não for autentico, o *gateway* envia a seguinte resposta:

HTTP/1.1 401 Não autorizado
WWW-Autenticação: Autent-Rec-Espelhado domínio="RecursosEspelhados"

4.4. Projeto de uma Ferramenta de Atualização

O projeto de uma possível ferramenta de atualização de recursos *Web* espelhados deve permitir que três tipos de operações sejam realizadas sobre os recursos espelhados, que são: operações de inclusão, exclusão e alteração. As operações de inclusão e alteração seriam realizadas pelo método PUT, definido pelo protocolo HTTP. Por outro lado, as operações de exclusão seriam realizadas pelo método DELETE.

A interação da ferramenta com o *proxy* se daria da seguinte forma: inicialmente, a ferramenta obteria todas as informações necessárias para que a atualização fosse realizada, tais como, as informações de autenticação, a URN, a versão do recurso e sua semântica de consistência; em seguida, a ferramenta montaria o pedido de atualização no formato definido pelo protocolo HTTP; e, finalizando, o pedido seria enviado para o *proxy* para que a atualização fosse realizada, como descrito anteriormente.

Capítulo 5

Aspectos da Implementação

5.1. Introdução

Como mencionado no capítulo anterior, as operações de atualização sobre os recursos Web espelhados são realizadas pelos métodos PUT e DELETE. Para que essas operações sejam executadas sem que a consistência das cópias dos recursos seja comprometida, inserimos ao sistema as funcionalidades de um *gateway*. Assim, os *gateways* juntamente com o *proxy*, definido por Fonsêca em seu trabalho [Fonsêca 1998], realizam as operações de atualização sobre os recursos espelhados.

Por outro lado, para que as operações de atualização fossem iniciadas pelos *proxy* seria necessário que incluíssemos ao *proxy* os módulos que implementam as atualizações sobre os recursos Web espelhados com semântica de consistência pessimista e otimista. Além disso, seria necessário também que incluíssemos ao protocolo HTTP os cabeçalhos definidos na seção 4.2.2, para que os métodos PUT e DELETE fossem executados. Para isso, o protótipo de nosso implementação teria que ser desenvolvido sobre implementações do *proxy* e do servidor *Web* Apache [Laurie e Laurie 1997], uma vez que a implementação do espelhamento de recursos, desenvolvido por Fonsêca em seu trabalho, foi implementado sobre esta infra-estrutura. Tendo em vista que, o protótipo do trabalho desenvolvido por Fonsêca estava sendo realizado em paralelo com o nosso e sobre a mesma infra-estrutura, optamos por não desenvolver o protótipo de nossa implementação sobre implementações do *proxy* e do servidor *Web*

Apache. Além disso, no protótipo de nossa implementação não implementamos o criptossistema de chaves públicas.

Nosso trabalho foi implementado da seguinte forma: construímos uma ferramenta de atualização, onde as funcionalidades, anteriormente descritas, inseridas no *proxy* para realizar as operações de atualização são realizadas por ela; e inserimos também no sistema um *daemon* para realizar as funcionalidades do *gateway*. Além disso, utilizamos o protocolo TCP/IP, como protocolo de comunicação, nas operações de atualização sobre os recursos *Web* espelhados. Assim, optar por esta implementação não comprometeu as funcionalidades, anteriormente descritas, do nosso esquema de gerência de recursos *Web* espelhados.

O objetivo deste capítulo é descrever a implementação de nossa ferramenta para gerência de recursos *Web* espelhados e do *daemon* de atualização. Ele foi estruturado da seguinte forma: na seção 5.2 apresentaremos a implementação do nosso esquema de gerência de recursos *Web* espelhados, onde utilizamos uma ferramenta de atualização e um *daemon* para realizar as funcionalidades requeridas pelo *proxy* e pelo *gateway*, respectivamente. Na seção 5.3 apresentaremos a implementação dos módulos de recuperação da ferramenta e do *daemon* utilizados para recuperar a consistência dos recursos espelhados caso a ferramenta ou o *daemon* sejam impossibilitados de concluir a atualização dos recursos.

5.2. Implementação do Esquema de Gerência de Recursos *Web* Espelhados

No capítulo anterior descrevemos nossa proposta para gerência dos recursos *Web* espelhados, onde as operações de atualização sobre os recursos espelhados são realizadas parte no *proxy* e parte no *gateway*, como mostra a Figura 6 apresentada no capítulo anterior. Nossa intenção agora é descrever como a ferramenta de atualização e o *daemon* foram implementados para que realizassem as operações de atualização sobre os recursos *Web* espelhados com os mesmos resultados que seriam alcançados pelo *proxy* e pelo *gateway*. A Figura 11 mostra a arquitetura utilizada para implementarmos

nosso esquema de gerência de recursos *Web* espelhados.

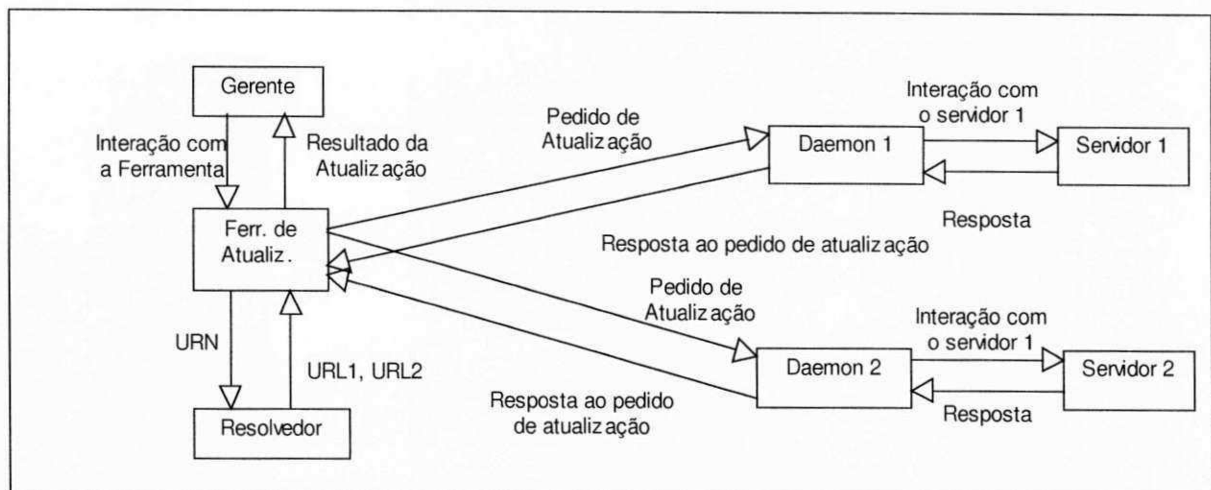


Figura 11 - Arquitetura Implementada para Atualizações de Recursos *Web* Espelhados

Como mostra a figura acima, as operações de atualização são enviadas pela ferramenta de atualização e recebidas pelo *daemon* para que sejam executadas. As operações realizadas pelo *proxy* e pelo *gateway* para manter a consistência do recursos espelhados são realizadas pela ferramenta de atualização e pelo *daemon* da mesma forma como descrito na seção 4.2.3, respectivamente.

5.2.1. Ferramenta de Atualização

Basicamente a implementação da ferramenta de atualização foi divida em 3 módulos: *interface* com o usuário, atualização pessimista e atualização otimista. O módulo de *interface* com o usuário foi implementado utilizando um formulário HTML e *interface* CGI (*Common Gateway Interface*) [Gundavaram 1996]. Este módulo apresenta para um determinado gerente de um recurso espelhado um formulário no qual ele deve informar os dados necessários para que a operação seja realizada.

Como descrito na seção 2.5, operações de inclusão e alteração sobre os recursos espelhados podem ser realizados pelo método PUT, enquanto que operações de exclusão de recursos podem ser realizados pelo método DELETE. Assim, a implementação desses métodos consiste, basicamente, na implementação das operações de inclusão e de alteração, e exclusão dos recursos espelhados, respectivamente. Por esta razão, nossa ferramenta executa os métodos PUT e DELETE através de operações

de inclusão, alteração e exclusão do recurso. Além desses operações, nossa ferramenta executa as operações de inclusão e exclusão do grupo.

Para que operações de alteração sejam executadas sobre os recursos espelhados é necessário que o gerente do recurso espelhado preencha o formulário com as seguintes informações: nome e senha do gerente do recurso espelhado; a URN referente ao recurso, além do arquivo que contém a informação para atualização. Esta URN é composta de uma URC e uma ou mais URLs. Em nosso projeto, a URC é mantida pelo servidor que armazena uma cópia do recurso identificado pela URN e contém as seguintes informações: versão do recurso, semântica de consistência, nome do recurso e do grupo ao qual pertence, o número da porta de comunicação, localização dos servidores que mantêm as cópias do recurso, o nível de prioridade entre os vários gerentes do recurso, caso o grupo de recursos possua vários gerentes, e um *flag* que indica o estado da mais recente atualização realizada sobre o recurso. Além disso, como descrito na seção 2.4, esta URN possui o nome de domínio de gerência do recurso, o nome do grupo ao qual o recurso espelhado pertence e o nome do recurso propriamente dito. Por exemplo, considere a seguinte URN:

urn:wmr:dsc.ufpb.br/alunos/raquel.html

que poderia ser utilizada para identificar a página HTML que contém as informações sobre a aluna Raquel, cujas cópias estão armazenadas em todos os servidores que espelham o grupo alunos do domínio de gerência dsc.ufpb.br.

Para ter acesso às outras informações sobre o recurso é necessário que a ferramenta recorra a um resolvedor para descobrir a localização dos servidores que mantêm as cópias do recurso identificado pela URN. Em seguida, a ferramenta deve consultar um dos servidores que mantém uma cópia do recurso espelhado para obter as seguintes informações contidas na URC: versão do recurso e sua semântica de consistência. Caso o servidor escolhido não esteja disponível, a ferramenta contata outro servidor e assim por diante.

Para realizar operações de exclusão do recurso o gerente fornece apenas a URN

referente ao recurso e informações de autenticação do gerente (nome e senha). A semântica, a versão e a localização dos servidores que mantêm o recurso espelhado são obtidas como descrito anteriormente. Por outro lado, para realizar operações de inclusão do recurso, o gerente fornece a URN referente ao recurso, informações de autenticação do gerente, semântica de consistência e o recurso propriamente dito. Para obter a localização dos servidores que espelham o recurso pertencente ao grupo, a ferramenta recorre a um resolvedor.

Além disso, para que essas operações sejam realizadas sobre as bases de dados dos servidores que mantêm recursos espelhados, é necessário que a localização dos servidores esteja armazenada na base de dados do resolvedor.

Para realizar operações de inclusão do grupo, é necessário que o gerente informe a URN, as informações de autenticação do gerente e a localização dos servidores que manterão cópias dos recursos pertencentes ao grupo. Por outro lado, para realizar as operações de exclusão do grupo, o gerente fornece apenas a URN e as informações de autenticação. Neste caso, antes da ferramenta executar o pedido de exclusão do grupo, ela deve atualizar a base de dados do resolvedor para excluir a localização dos servidores que mantêm cópias dos recursos pertencentes ao grupo. Porém, nos casos de inclusão do grupo, a ferramenta atualiza apenas a base de dados do resolvedor incluindo a localização dos servidores que manterão cópias dos recursos pertencentes ao grupo.

É importante esclarecer que as operações de exclusão do grupo são realizadas de forma otimista; caso o grupo possua recursos com semântica de consistência pessimista, o gerente deve excluí-los e, em seguida, realizar esta operação.

5.2.1.1. BASES DE DADOS DO RESOLVEDOR

Como definido por Fonsêca em seu trabalho [Fonsêca], a localização dos servidores que mantêm cópias dos recursos espelhados são armazenadas nos registros

TXT do DNS[6]. Na Figura 12 há uma parte da base de dados de um servidor DNS.

pesquisas.dsc.ufpb.br	**IN**	**TXT**	**"www.dsc.ufpb.br, www.di.ufpe.br"**
proj-lsd. lsd.dsc.ufpb.br	**IN**	**TXT**	**"www.dsc.ufpb.br, www.unicamp.br"**

Figura 12 - Extrato da base de dados do DNS

De acordo com a Figura 12, o servidor *Web* do Departamento de Sistemas e Computação e o servidor *Web* do Departamento de Informática mantêm cópias dos recursos espelhados pertencentes ao grupo *pesquisas.* Assim, podemos perceber que, recursos espelhados pertencentes a um mesmo grupo são mantidos pelos mesmos servidores.

Para obter a localização das cópias de um dado recurso é necessário informar para o resolvedor o nome do grupo ao qual o recurso espelhado pertence e o seu domínio de gerência; o resolvedor concatena essas informações, realiza a consulta e retorna a localização das cópias do recurso espelhado.

5.2.1.1.1. ATUALIZAÇÃO DAS BASE DE DADOS DO RESOLVEDOR

Como mencionado anteriormente, ao realizar uma operação de inclusão ou exclusão de um determinado grupo, é necessário que a ferramenta inclua ou exclua a localização dos servidores que manterão ou mantêm as cópias dos recursos pertencentes ao grupo, respectivamente. Para isso, nós inserimos um *daemon,* denominado *daemonRes*, que é responsável por receber os pedidos de inclusão/exclusão do grupo solicitados pela ferramenta e executá-los sobre a base de dados do resolvedor. Um pedido possui as seguintes informações: o grupo do recurso, o domínio de gerência e informações de autenticação, caso a operação seja uma exclusão do grupo. Quando a operação é uma inclusão, além das informações acima descritas, o pedido possui a localização dos servidores que manterão as cópias dos recursos pertencentes ao grupo.

Ao receber um pedido, o *daemon* verifica se o gerente tem as permissões

[6]DNS é o bem conhecido servidor de nomes da Internet [AL97].

necessárias para realizar a operação; se eles não tem as permissões necessárias, o *daemon* retorna uma resposta NOTOK; caso contrário, a operação é executada e é retornado para a ferramenta uma mensagem indicando que a operação foi realizada com sucesso.

Ao receber uma resposta do *daemon,* a ferramenta verifica se a operação foi realizada; caso ela tenha sido realizada, a ferramenta inicia ou finaliza a operação de exclusão ou inclusão do grupo, respectivamente; caso contrário, a operação é cancelada e é retornado para o gerente uma mensagem indicando o motivo de sua não realização.

5.2.1.2. MÓDULO DE ATUALIZAÇÃO PESSIMISTA

Este módulo é responsável pela execução dos pedidos de atualização sobre os recursos com semântica de consistência pessimista. Para implementarmos este módulo, utilizamos o conceito de transação atômica, ou seja, ou todos os servidores realizam a operação de atualização sobre os recursos espelhados ou nenhum deles realiza a operação. Inicialmente, este módulo solicita pedidos de conexão para todos os servidores que mantêm um determinado recurso espelhado; se pelo menos um pedido não foi satisfeito, a atualização é cancelada. Caso todos os servidores aceitem o pedido de conexão, o módulo de atualização envia o recurso atualizado e todos os atributos necessários para que o servidor possa realizar a atualização, tais como: sua versão, informações de autenticação, o nome do recurso e do grupo, o domínio de gerência e o tipo de atualização. Se a atualização for uma inclusão do recurso, o módulo envia, além das informações descritas anteriormente, a semântica de consistência do recurso e a localização dos servidores que mantêm o recurso espelhado. Porém, este módulo não envia pedidos com operações de exclusão do grupo. Na Figura 13, apresentamos um esboço do algoritmo que implementa o módulo de atualização pessimista.

Módulo de Atualização Pessimista

Início

1 ***Enquanto*** existirem cópias do recurso espelhado ***faça***

2 Abra uma conexão com o servidor que mantém esta cópia

3 ***Se*** a conexão for realizada

4	Armazene o descritor de *socket* em um conjunto de descritores
5	***Caso contrário***
6	Finalize a atualização, e inclua o pedido e o endereço dos servidores em um arquivo para que seja realizado em segundo plano
7	***Se*** foi possível abrir uma conexão com todos os servidores que mantêm uma cópia do recurso
8	***Enquanto*** existir cópias do recurso espelhado ***faça***
9	Selecione um dos descritores de *socket* contidos no conjunto de descritores que esteja pronto para enviar um pedido ou receber uma resposta
10	***Para*** todos os descritores de *socket* ***faça***
11	***Se*** o descritor estiver pronto para escrita
12	Envie o pedido
13	***Se*** todos os pedidos foram enviados
14	Inicie o *timeout*
15	***Se*** o descritor estiver pronto para leitura
16	Receba a resposta
17	***Se*** o *timeout* expirou ou um dos servidores não permitiu a atualização
18	Envie um *abort* para todos os servidores que confirmaram o pedido de atualização
19	***Caso contrário***
20	Envie a confirmação para todos os servidores
21	Se a atualização não foi realizada
22	Inclua o pedido e o endereço dos servidores em um arquivo para que seja executado em segundo plano.
Fim	

Figura 13 - Algoritmo do Módulo de Atualização Pessimista

Como mostra a figura acima, no passo 8 é iniciado a atualização atômica propriamente dita. Nossa ferramenta faz o controle da atualização atômica a nível de mensagens, ou seja, ela envia os pedidos de atualização para todos os servidores que anteriormente foram conectados e espera por uma confirmação; se pelo menos um dos servidores não confirmar ou não autorizar o pedido de atualização em um período de tempo finito, então a ferramenta envia um pedido de *abort* para todos os servidores que

confirmaram o pedido de atualização, como mostra a Figura 8. Caso todos os servidores confirmem o pedido de atualização, a ferramenta envia um pedido de *commit* para todos os servidores (Figura 7). Considerando que as partições são integradas em um tempo finito, se ocorrerem particionamentos durante o envio do pedido de *commit* ou *abort*, e isso resultar no não recebimento de sua confirmação pela ferramenta durante um período de tempo finito, a ferramenta envia uma resposta para o gerente indicando que atualização pode ter realizado o *commit* ou não. Assim, o gerente pode decidir enviar o pedido de atualização novamente.

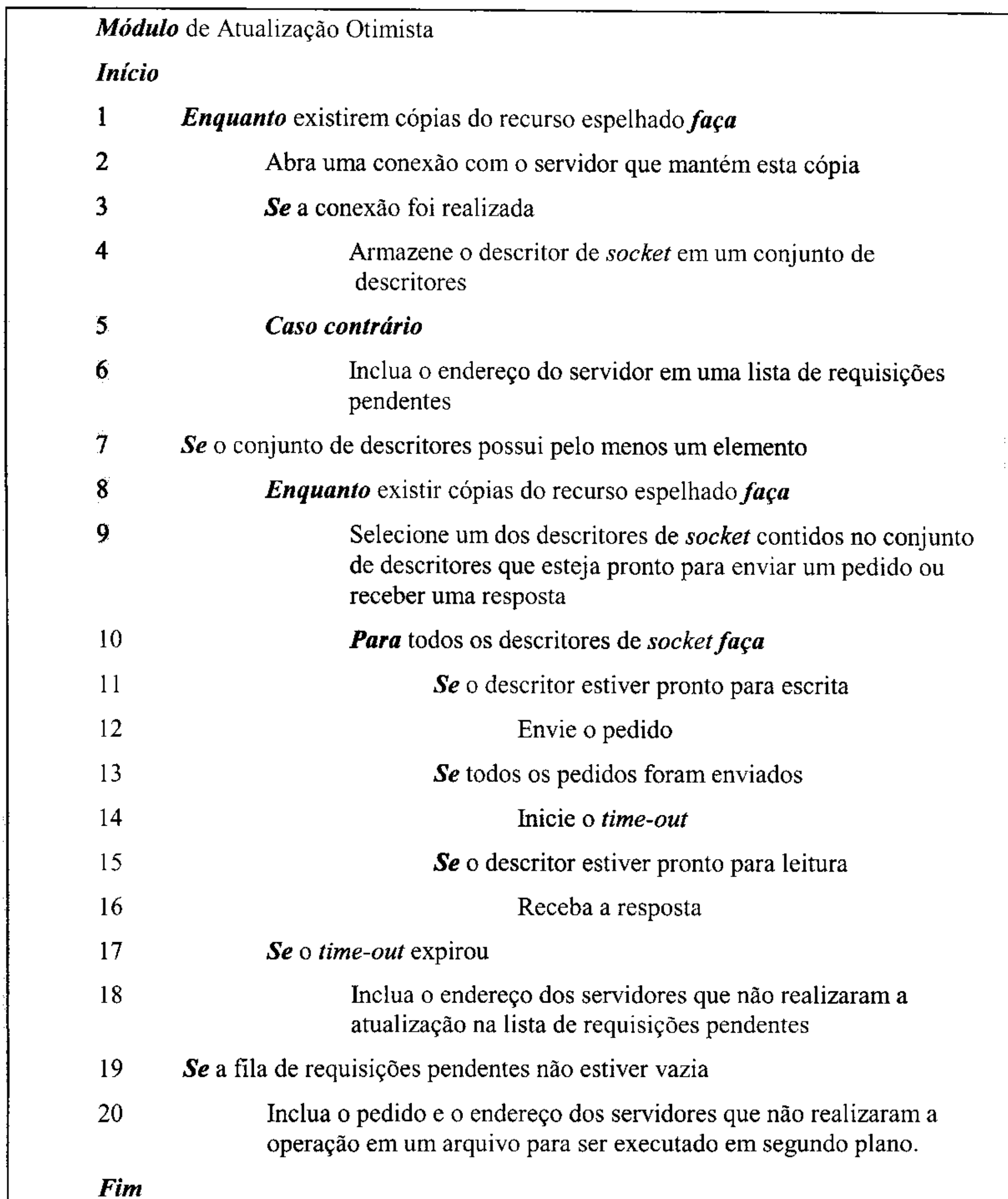
Quando um pedido de atualização é abortado, a ferramenta inclui o pedido em uma lista de requisições pendentes para ser executado, mais tarde, em segundo plano. A execução em segundo plano será discutida posteriormente.

Caso a ferramenta falhe durante operações de atualização, é necessário verificar em que situação a ferramenta falhou, e com isso a ferramenta pode decidir realizar a atualização novamente ou não. Para isso, implementamos um módulo de recuperação da ferramenta que realizará o tratamento de falhas. Este módulo será discutido com mais detalhes na seção 5.3.1. Antes, na seção 5.2.1.3 apresentaremos o módulo de atualização pessimista, na seção 5.2.1.4 discutiremos como será realizado a execução em segundo plano e na seção 5.2.2 descreveremos o processamento das operações de atualização realizado pelo *daemon*.

5.2.1.3. MÓDULO DE ATUALIZAÇÃO OTIMISTA

Ao contrário do módulo de atualização pessimista, este módulo envia os pedidos de atualização para os servidores que mantêm os recursos espelhados em transações independentes, ou seja, a não realização da operação sobre uma determinada cópia de um recurso, não interfere em sua realização sobre outra cópia do recurso. Este módulo envia, além dos pedidos de inclusão, exclusão e alteração do recurso, os pedidos de exclusão do grupo. Quando a operação é uma exclusão do grupo, o módulo envia as informações de autenticação, o tipo de atualização, o nome do grupo e o domínio de gerência. Na Figura 14, apresentamos um esboço do algoritmo que implementa o

módulo de atualização otimista.



Módulo de Atualização Otimista

Início

1 ***Enquanto*** existirem cópias do recurso espelhado ***faça***

2 Abra uma conexão com o servidor que mantém esta cópia

3 ***Se*** a conexão foi realizada

4 Armazene o descritor de *socket* em um conjunto de descritores

5 ***Caso contrário***

6 Inclua o endereço do servidor em uma lista de requisições pendentes

7 ***Se*** o conjunto de descritores possui pelo menos um elemento

8 ***Enquanto*** existir cópias do recurso espelhado ***faça***

9 Selecione um dos descritores de *socket* contidos no conjunto de descritores que esteja pronto para enviar um pedido ou receber uma resposta

10 ***Para*** todos os descritores de *socket* ***faça***

11 ***Se*** o descritor estiver pronto para escrita

12 Envie o pedido

13 ***Se*** todos os pedidos foram enviados

14 Inicie o ***time-out***

15 ***Se*** o descritor estiver pronto para leitura

16 Receba a resposta

17 ***Se*** o ***time-out*** expirou

18 Inclua o endereço dos servidores que não realizaram a atualização na lista de requisições pendentes

19 ***Se*** a fila de requisições pendentes não estiver vazia

20 Inclua o pedido e o endereço dos servidores que não realizaram a operação em um arquivo para ser executado em segundo plano.

Fim

Figura 14 - Algoritmo do Módulo de Atualização Otimista

Semelhantemente ao módulo anterior, inicialmente, este módulo solicita pedidos de conexão para todos os servidores que mantêm um determinado recurso espelhado e envia as mesmas informações mencionadas anteriormente. O que diferencia do módulo anterior é que se um ou mais pedidos não forem realizados, eles são incluídos em uma

fila de requisições pendentes para que sejam executados em segundo plano. No passo 8 da Figura 14 é iniciado o envio do pedido de atualização propriamente dito, este pedido é enviado para todos os servidores que anteriormente foram conectados e espera por uma confirmação; o servidor que não confirmar a finalização da atualização, é incluído na fila de requisições pendentes para que seja executado mais tarde em segundo plano. Uma vez enviado o pedido, no passo 19 é verificado se a fila possui algum servidor que não realizou a atualização; se houver, as informações referentes ao pedido de atualização são armazenados em um arquivo, que será utilizado, mais tarde, para realizar a operação em segundo plano. A execução em segundo plano será discutida na próxima seção.

5.2.1.4. Execução em Segundo Plano

Como mencionado anteriormente, para que seja possível realizar uma operação de atualização em segundo plano é necessário que todos as informações referentes ao pedido de atualização sejam armazenadas em um arquivo. Este arquivo é identificado por um número e armazenado em um diretório denominado *./fila*. Assim, os arquivos contidos neste diretório são ordenados de acordo com o número que lhe são atribuídos.

Para realizar a execução em segundo plano, nós inserimos ao sistema um *daemon*, denominado *daemon-fila*, que tem com função analisar, de tempos em tempos, se existe algum arquivo dentro do diretório *./fila*. Assim, para cada arquivo contido neste diretório, o *daemon-fila* obtém as informações nele contidas e realiza a operação de atualização utilizando um dos módulos anteriormente descritos. A escolha de qual módulo utilizar vai depender do tipo de semântica de consistência do recurso a ser atualizado. Por fim, ao término de cada operação realizada, o seu respectivo arquivo é eliminado do diretório *./fila*.

5.2.2 *Daemon*

O *daemon* é responsável por receber os pedidos de atualização e processá-los. Além disso, ele permite que se obtenha informações sobre um determinado recurso espelhado, tais como, sua versão e semântica de consistência. A implementação do

daemon foi dividida em 2 módulos: processamento de atualização otimista e processamento de atualização pessimista. O *daemon* processa pedidos de atualização de forma concorrente, ou seja, para cada pedido de conexão, o *daemon* cria um novo processo que será usado para executar o pedido. Assim, o *daemon* poderá executar vários pedidos de atualização ao mesmo tempo (Figura 15).

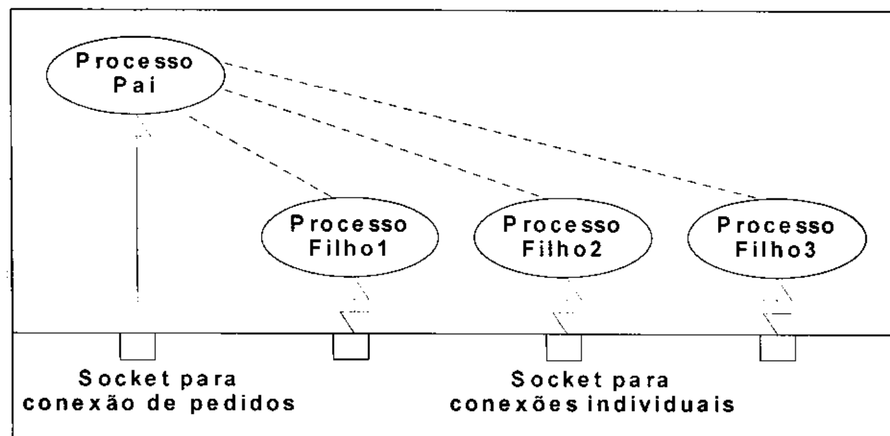


Figura 15 - Estrutura de Processos do *Daemon*

Uma vez criado o processo filho, o *daemon* está pronto para receber o pedido. O *daemon* pode receber três tipos de pedidos: os pedidos de atualização; pedidos de *consulta_estado_do_recurso*, solicitados por rotinas de recuperação do *daemon*; e pedidos de *consulta_dados*, solicitados pela ferramenta. Para cada pedido de atualização, o *daemon* verifica as permissões do gerente do recurso; se o gerente tem as permissões necessárias para executar o pedido, ele é processado de acordo com o tipo de semântica de consistência do recurso identificado no pedido; caso contrário, é retornado para o gerente uma resposta de ***não autorizado***. Por outro lado, ao receber pedidos de ***consulta_estado_do_recurso***, o ***daemon*** envia o versão do recurso, o estado contido no seu *flag* do recurso e o nome do gerente que realizou a mais recente atualização sobre o recurso, caso ele exista. Ao receber pedidos de *consulta_dados*, o *daemon* envia a versão e a semântica do recurso identificado no pedido, caso ele exista. A seguir, apresentaremos as bases de dados associadas aos recursos espelhados e a implementação dos módulos de processamento de atualização pessimista e otimista.

5.2.2.1. BASE DE DADOS DOS RECURSOS ESPELHADOS

Em nosso esquema, os dados contidos na URC e as informações de autenticação enviadas nos pedidos de atualização são gerenciadas pelo *daemon* e armazenados em memória estável. Cada servidor que mantém recursos espelhados possui em sua base de dados um arquivo de senhas (*passwd*) que é utilizado pelo *daemon* sempre que um pedido de atualização é solicitado. Ele armazena o nome do gerente e sua respectiva senha. Além desse arquivo, o servidor mantém outro arquivo que é identificado pelo nome do recurso e possui a extensão *urc*. Este arquivo é associado a cada recurso espelhado, onde são armazenados os dados contidos na URC.

São associados também a cada recurso com semântica de consistência otimista, um arquivo que possui a extensão *ger* e é identificado pelo nome do recurso, onde é armazenado o nome do gerente que realizou uma atualização. Por outro lado, são associados a cada recurso com semântica de consistência pessimista, além do arquivo descrito anteriormente, um arquivo de *log*, onde são registradas todas as operações realizadas pelo o *daemon* durante uma operação de atualização pessimista sobre um recurso. Este arquivo é identificado pelo nome do recurso e possui a extensão *log*. Ele armazena o tipo de atualização e os vários estados da operação, tais como, recebeu pedido, enviou resposta, recebeu confirma/desfaça e enviou resposta.

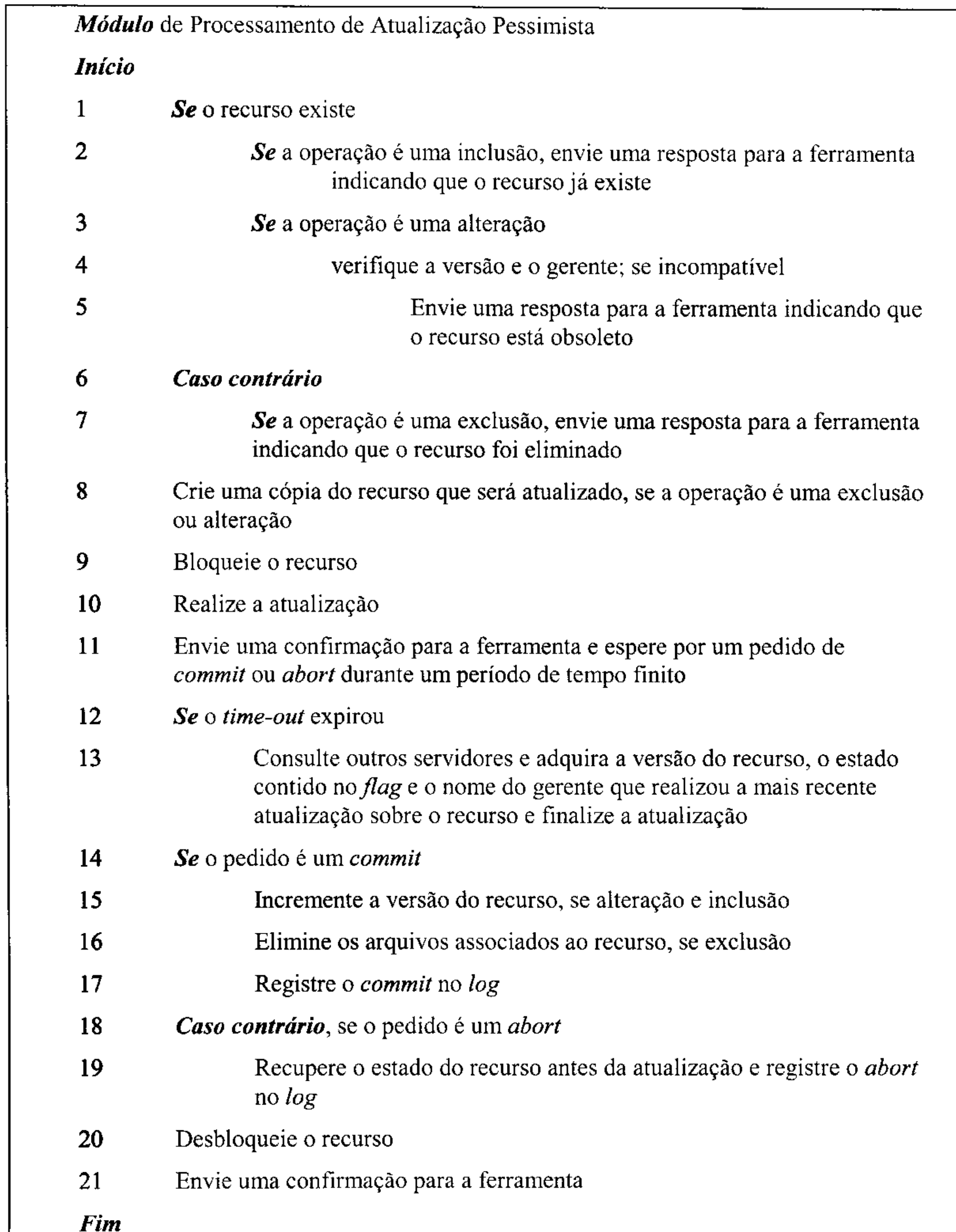
Além disso, o *daemon* gerencia um arquivo denominado *daemon.log*, onde são registradas todas as atualizações realizadas pelo *daemon* a partir de uma certa data pré-definida. Ele armazena a data, a hora e o nome do grupo e do recurso espelhado.

Por outro lado, a ferramenta cria durante cada nova operação de atualização um arquivo denominado *ferramenta.log*, onde são armazenados o tipo de atualização que estava sendo realizada e os vários estados da operação.

5.2.2.2. MÓDULO DE PROCESSAMENTO DE ATUALIZAÇÃO PESSIMISTA

Este módulo é responsável pelo processamento propriamente dito do pedido de atualização. Ele é solicitado quando o recurso a ser atualizado possui semântica de

consistência pessimista. Na Figura 16, apresentamos um esboço do algoritmo que implementa o módulo de processamento de atualização pessimista.



Módulo de Processamento de Atualização Pessimista

Início

1 ***Se*** o recurso existe

2 ***Se*** a operação é uma inclusão, envie uma resposta para a ferramenta indicando que o recurso já existe

3 ***Se*** a operação é uma alteração

4 verifique a versão e o gerente; se incompatível

5 Envie uma resposta para a ferramenta indicando que o recurso está obsoleto

6 ***Caso contrário***

7 ***Se*** a operação é uma exclusão, envie uma resposta para a ferramenta indicando que o recurso foi eliminado

8 Crie uma cópia do recurso que será atualizado, se a operação é uma exclusão ou alteração

9 Bloqueie o recurso

10 Realize a atualização

11 Envie uma confirmação para a ferramenta e espere por um pedido de *commit* ou *abort* durante um período de tempo finito

12 ***Se*** o *time-out* expirou

13 Consulte outros servidores e adquira a versão do recurso, o estado contido no *flag* e o nome do gerente que realizou a mais recente atualização sobre o recurso e finalize a atualização

14 ***Se*** o pedido é um *commit*

15 Incremente a versão do recurso, se alteração e inclusão

16 Elimine os arquivos associados ao recurso, se exclusão

17 Registre o *commit* no *log*

18 ***Caso contrário***, se o pedido é um *abort*

19 Recupere o estado do recurso antes da atualização e registre o *abort* no *log*

20 Desbloqueie o recurso

21 Envie uma confirmação para a ferramenta

Fim

Figura 16 - Algoritmo do Módulo de Processamento de Atualização Pessimista

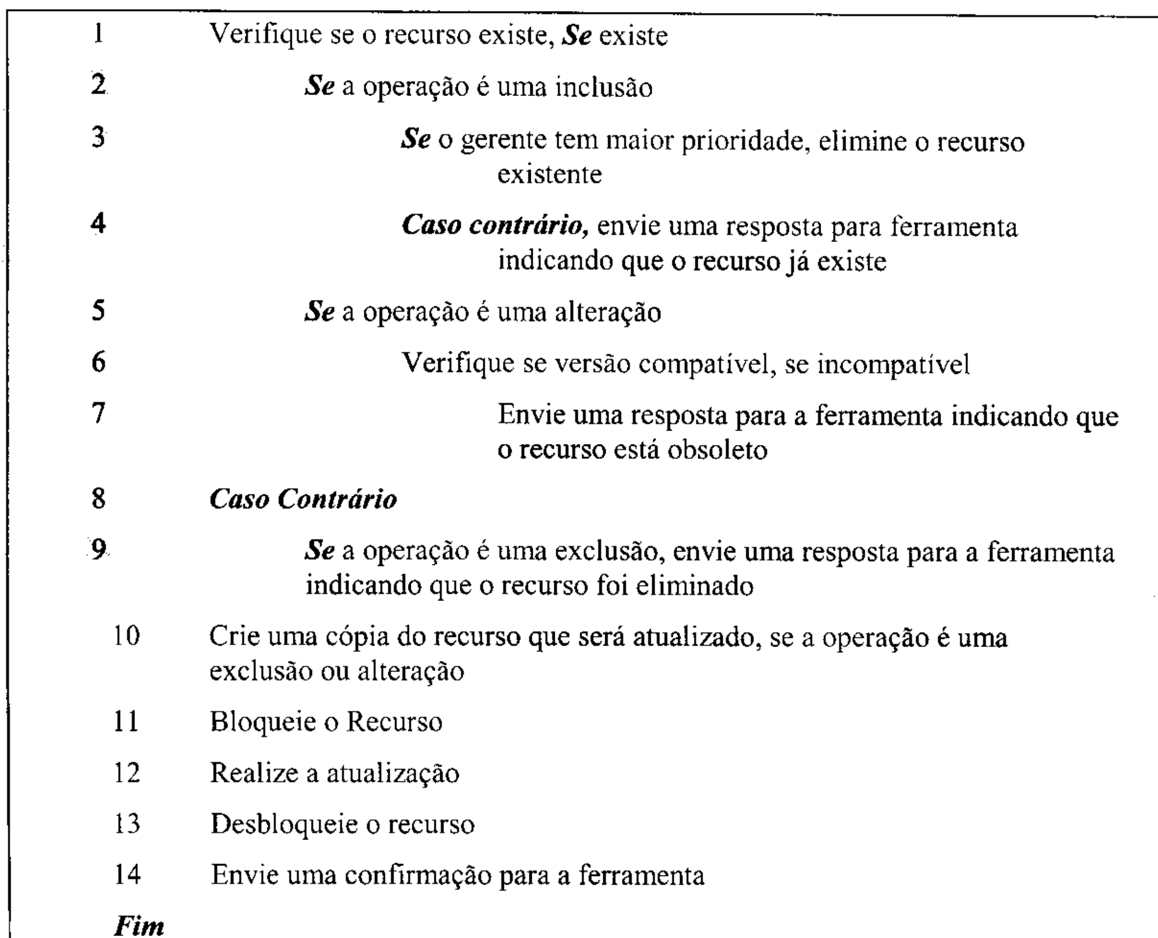
Inicialmente, é verificado se o recurso a ser atualizado existe; se o recurso existe, a operação pode ser iniciada, desde que a atualização seja uma alteração ou exclusão. No passo 8, o módulo cria uma cópia do recurso, que será utilizada pela operação de alteração e exclusão, caso seja solicitado um pedido de *abort* durante a atualização. No passo 4 da Figura 16, é feita a comparação entre a versão contida no pedido e a versão do recurso armazenado no servidor; caso elas sejam incompatíveis, o processamento do pedido de atualização não é realizado. Além disso, no passo 11 o *daemon* espera por um pedido da ferramenta; se a ferramenta não enviar o pedido em um tempo finito, ele consulta outros servidores para decidir o que fazer, de acordo como descrito na seção 4.2.3.1.2.

5.2.2.3. Módulo de Processamento de Atualização Otimista

Este módulo é solicitado se o recurso a ser atualizado possui semântica de consistência otimista. Na Figura 17, apresentamos um esboço do algoritmo que implementa o módulo de processamento de atualização otimista. Inicialmente, é verificado se o recurso a ser atualizado existe; se o recurso existe, operação pode ser iniciada, desde que a atualização seja uma alteração ou exclusão. No passo 3, é verificado se o gerente que realizou a inclusão tem maior prioridade do que o gerente da inclusão corrente; caso o gerente da inclusão corrente tenha maior prioridade, a inclusão é iniciada. No passo 6, é realizado uma comparação entre a versão do recurso contida no pedido e a versão do recurso propriamente dita; se as versões estiverem incompatíveis significa que ocorreu uma colisão, então o módulo verifica a prioridade entre o gerente da atualização corrente e o gerente que realizou a mais recente atualização; a atualização válida será a que foi realizada pelo gerente de maior prioridade. Através desse critério é possível realizar atualizações de forma concorrente sem que a consistência das cópias dos recursos espelhados seja comprometida. Além disso, no passo 12 é realizado a atualização propriamente dita; se for uma inclusão ou atualização do recurso, além de atualizá-lo, sua versão é incrementada.; se for uma exclusão, o recurso é eliminado.

Módulo de Processamento de Atualização Otimista

Início



1 Verifique se o recurso existe, ***Se*** existe
2 ***Se*** a operação é uma inclusão
3 ***Se*** o gerente tem maior prioridade, elimine o recurso existente
4 ***Caso contrário,*** envie uma resposta para ferramenta indicando que o recurso já existe
5 ***Se*** a operação é uma alteração
6 Verifique se versão compatível, se incompatível
7 Envie uma resposta para a ferramenta indicando que o recurso está obsoleto
8 ***Caso Contrário***
9 ***Se*** a operação é uma exclusão, envie uma resposta para a ferramenta indicando que o recurso foi eliminado
10 Crie uma cópia do recurso que será atualizado, se a operação é uma exclusão ou alteração
11 Bloqueie o Recurso
12 Realize a atualização
13 Desbloqueie o recurso
14 Envie uma confirmação para a ferramenta
Fim

Figura 17 - Algoritmo do Módulo de Processamento de Atualização Otimista

5.3. Módulos de Recuperação

Apesar de resolvidos os problemas de inconsistência gerados por particionamento da rede e indisponibilidade dos servidores que mantêm os recursos espelhados, é necessário garantir a consistência dos recursos caso a ferramenta seja impossibilitada de concluir a sua atualização. Além disso, é necessário que depois de uma falha do *daemon*, ele recupere a consistência dos recursos que estavam sendo atualizados antes da falha. Assim, implementamos dois módulos que recuperam a consistência dos recursos espelhados caso ocorram falhas na ferramenta ou no *daemon* durante uma operação de atualização, são eles: módulo de recuperação da ferramenta e módulo de recuperação do *daemon*.

5.3.1. Módulo de Recuperação da Ferramenta

Durante toda a execução dos módulos de atualização otimista e pessimista, um arquivo de *log* é atualizado, para que, em caso de falhas, ele seja usado para recuperar a consistência dos recursos que estavam sendo atualizados. Este arquivo de *log* é atualizado de acordo com o tipo de atualização que estava sendo realizada, ou seja, se a operação de atualização estava sendo realizada pelo módulo de atualização otimista, por exemplo, é incluído no arquivo de *log* o tipo de atualização e todos os estados da operação de atualização executada pela ferramenta.

Inicialmente, este módulo analisa o arquivo de *log* e verifica que tipo de atualização estava sendo realizada e em que ponto a atualização foi interrompida, para que decisões apropriadas sejam tomadas. Dois módulos foram definidos para recuperação da ferramenta, são eles: módulo que recupera operações de atualização otimista e pessimista.

O módulo que recupera operações de atualização otimista comporta-se da seguinte forma: se a operação for interrompida antes que o pedido tenha sido enviado para todos os servidores que mantêm o recurso espelhado ou antes que pelo menos um dos servidores tenham confirmado a atualização, o módulo conclui que existe a possibilidade do pedido não ter sido processado por todos os servidores, com isso, ele envia uma resposta para o administrador da ferramenta indicando que o pedido de atualização deve ser realizado novamente; se a operação de atualização foi interrompida depois que todos os servidores tenham processado o pedido, mais havia elementos na fila de requisições pendentes e as informações referentes a esta operação ainda não haviam sido armazenadas em memória estável para que fosse realizada em segundo plano, o módulo registra as informações da operação no arquivo e o *daemon-fila* se encarregará de realizar a operação como descrito anteriormente.

Por outro lado, o módulo que recupera operações de atualização pessimista possui as seguintes características: se a operação for interrompida antes que a ferramenta envie os pedidos de *commit* ou de *abort* para todos os servidores, este

módulo inclui o pedido em uma lista de requisições pendentes para ser realizado novamente em segundo plano; se a operação for interrompida depois que pedido de *abort* tenha sido enviado, o módulo conclui que o pedido foi realizado pelo *daemon* mesmo que ele não o tenha recebido; finalizando, se a operação for interrompida depois de iniciar o envio dos pedidos de *commit*, o módulo conclui que se pelo menos um dos servidores recebeu o *commit*, ele foi realizado sobre todas as cópias do recurso. Por outro lado, se nenhum deles recebeu o *commit*, a atualização foi abortada. Assim, este módulo envia o pedido novamente em segundo plano, pois se o *commit* foi realizado, o *daemon* enviará uma resposta indicando que o recurso realizou o *commit*.

Uma característica importante do módulo de recuperação pessimista é que ele deve esperar um certo tempo pré-definido para iniciar o envio da operação de atualização para todos os servidores, pois é necessário garantir que os servidores realizaram o *abort* ou o *commit* da respectiva operação. Além disso, os pedidos que estão sendo enviados novamente são descartados pelo *daemon*, caso ele já os tenha executado.

5.3.2. Módulo de Recuperação do Daemon

Este módulo é ativado na inicialização do *daemon*, ou seja, antes do *daemon* iniciar a receber pedidos, ele ativa o módulo de recuperação para recuperar a consistência dos recursos que estavam sendo atualizados no momento da falha.

Inicialmente, este módulo analisa o arquivo *daemon.log* e obtém a data da última atualização registrada. A partir desta informação, este módulo refaz todas as atualizações realizadas um dia anterior da data obtida em diante. Para cada atualização registrada, o módulo de recuperação do *daemon* verifica sobre qual recurso este módulo estava atualizando, e a partir desta informação, ele analisa o respectivo arquivo de *log* do recurso. Para cada atualização não concluída, este módulo cria um novo processo que será utilizado para consultar outros servidores. Depois disto, o *daemon* está pronto para receber pedidos. É importante esclarecer que o *daemon* não espera pela finalização das consultas para iniciar a receber pedidos de atualização, ou seja, após ativada todas as

consultas necessárias para que todos os recursos não atualizados sejam atualizados, o *daemon* pode iniciar a receber pedidos independente destas consultas terem sido finalizadas ou não.

As operações de atualização realizadas sobre recursos com semântica de consistência otimista não são refeitas, pois, provavelmente, a ferramenta incluiu estas operações na sua lista de pendências ,e com isso as operações serão realizadas. Por outro lado, as operações de atualização realizadas sobre recursos com semântica de consistência pessimista são refeitas, desde que o arquivo de *log* referente ao recurso não possua uma mensagem indicando a finalização da operação.

Assim, se a operação foi interrompida antes do *daemon* receber o pedido de *commit* ou *abort*, o módulo de recuperação do *daemon* cria um novo processo que será usado para consultar outros servidores que mantêm cópias do recurso sendo atualizado para obter seu estado. Por outro lado, se a operação foi interrompida nas demais situações a consistência do recurso sendo atualizado não é comprometida, pois provavelmente ela realizou ou um *commit* ou um *abort*.

Capítulo 6

Conclusões

O enorme crescimento da *Web* têm despertado nos últimos anos o interesse das organizações que passam a utilizá-la para distribuir informações e serviços por todo mundo e a um custo extremamente baixo. Entretanto, a *Web* possui alguns problemas que causam ainda insatisfação por parte de seus usuários; um deles é o problema da quebra de uma ligação em um hipertexto. Preocupado com isso, Fonsêca desenvolveu um projeto [Fonsêca 1998], onde ele resolve este problema parcialmente através do espelhamento de recursos *Web* utilizando as funcionalidades de um servidor *Web proxy*.

Neste trabalho apresentamos um esquema de gerência de recursos *Web* espelhados, impulsionado pela consideração que ao utilizar o mecanismo de espelhamento de recursos *Web* é necessário pensar em como manter a consistência dos recursos espelhados, principalmente quando ocorrer falhas nas máquinas servidores e particionamento na rede durante operações de atualização sobre os recursos espelhados. Nosso esquema estende o trabalho desenvolvido por Fonsêca, que resolve apenas os problemas de acesso para leitura. Assim, para resolvermos os problemas de inconsistência gerados pelos acessos para escrita, incrementamos as funcionalidades do servidor *Web proxy* para garantir a consistência dos recursos espelhados durante as atualizações; e utilizamos um componente que pode se introduzido na atual infra-estrutura *Web*, cujo objetivo é efetuar os pedidos de atualização no servidor.

No capítulo 3, descrevemos alguns métodos que utilizam a abordagem

pessimista e otimista para manter a consistência de informações replicadas na presença de falhas dos nodos e particionamento da rede. Cada método define vários atributos, que são associados às cópias das informações replicadas, necessários para que sua consistência seja preservada. Por outro lado, alguns deles reduzem a disponibilidade, o *Log transformations* necessita da intervenção do usuário para recuperar partições e o *Weighted voting*, além de reduzir a disponibilidade, aumenta a sobrecarga de comunicação. Em nosso trabalho utilizamos algumas considerações apresentadas em alguns destes métodos para desenvolver nosso esquema de gerência que mantêm a consistência das bases de dados dos servidores que mantêm recursos espelhados e minimiza alguns dos problemas acima descritos, tais como, disponibilidade reduzida, aumento na sobrecarga de comunicação e intervenção do usuário para reparar partições.

Em nosso esquema de gerência, os recursos espelhados podem ser atualizados através dos métodos PUT e DELETE, definido pelo protocolo HTTP, e é possível associar aos recursos, semânticas de consistência otimista e pessimista, dependendo de suas necessidades. Alem disso, nosso esquema é capaz de tolerar falhas em servidores e canais de comunicação de maneira transparente.

Nós definimos também em nosso trabalho, um esquema de autenticação de gerentes de recursos que utiliza senhas de acesso e criptografia de chave pública para garantir a autenticidade do gerente do recurso durante atualizações do recurso espelhado.

6.1. Contribuições

No trabalho desenvolvido por Fonsêca [Fonsêca 1998], para possibilitar o espelhamento de recursos, as facilidades de um *proxy* e um esquema de identificação baseado em URNs (*Uniform Resource Names*) são utilizados. Porém, Fonsêca só resolve os problemas de acesso para leitura, ou seja, o servidor *proxy* dá um tratamento especial apenas para as requisições que contêm os métodos GET e HEAD, as requisições que contêm outros métodos, tais como, PUT e DELETE, não são tratadas pelo *proxy*. Nosso trabalho vai ao encontro da solução deste problema, uma vez que,

fornece uma ferramenta de gerência das bases de dados de servidores que mantêm recursos espelhados que garante a consistência dos recursos durante a execução destas requisições.

Uma vantagem do nosso esquema é que ele oferece flexibilidade para os gerentes dos recursos, por permitir que eles escolham o tipo de semântica de consistência do recurso espelhado. Além disso, ele permite que as operações de atualização, não realizadas pela ferramenta, sejam executadas em segundo plano de forma transparente para o usuário.

Diante da importância da *Web* e da necessidade de usufruir de seus recursos livres do problema de quebra de ligações, encontramos a relevância deste trabalho de dissertação. Ele visa exatamente fornecer um esquema de gerência de recursos *Web* espelhados, ou seja, o problema de quebra de ligações foi resolvido através do espelhamento dos recursos *Web*, mas era necessário também garantir que os recursos *Web* espelhados estivessem sempre íntegros e consistentes.

6.2. Direções para Trabalhos Futuros

Uma requisição HTTP pode conter qualquer um dos métodos discutidos na seção 2.3. Os métodos GET, HEAD, PUT e DELETE possuem a propriedade de idempotência, ou seja, desprezando-se os efeitos colaterais relacionados com erros e expiração da validade dos recursos armazenados em *cache*, o efeito do processamento da requisição não muda, mesma que ela seja processada mais de uma vez [Fielding et al. 1997].

No capítulo 4, foi apresentado nosso esquema de gerência de recursos *Web* espelhados. Como foi possível observar durante a leitura daquele capítulo, foram implementados apenas os métodos PUT e DELETE pelo *proxy* e pelo *gateway* para realizar as atualizações sobre os recursos *Web* espelhados. A razão para a não implementação do método POST é que existe a possibilidade dele, em algumas situações, ter um comportamento não idempotente, e com isso dificulta a garantia de

consistência da informação mantida pelo servidor, mesmo no caso de recursos com semântica de consistência otimista.

Uma possível direção para trabalhos futuros seria investigar as possibilidades da implementação do método POST pelo *proxy* e pelo *gateway* para realizar operações de atualização sobre os recursos *Web* espelhados. Outras perspectivas para trabalhos futuros seriam: avaliar o desempenho do esquema de gerência dos recursos *Web* espelhados sobre uma grande quantidade de cópias dos recursos armazenadas em servidores distantes geograficamente; incluir operações de compressão de dados na transmissão para que a quantidade de dados transmitidos fosse reduzida, uma vez que, arquivos muito extensos levariam muito tempo para serem atualizados; permitir a atualização de diretórios e de arquivos que requerem políticas diferentes de atualização. Assim, bases de dados poderiam ser atualizadas utilizando operações de substituição e os arquivos de *log* poderiam ser atualizados utilizando operações de inclusão da nova informação no fim do arquivo.

As operações de atualização sobre diretórios permitirão que vários arquivos sejam atualizados através de uma única operação de atualização, ou seja, vários arquivos pertencentes ao mesmo diretório seriam atualizados através de uma única transação, ao invés de uma transação para cada arquivo.

6.3. Considerações Finais

A técnica de replicação vem sendo amplamente utilizada para tornar as informações altamente disponíveis, mesmo na presença de falhas nas máquinas servidoras que mantêm as cópias das informações replicadas ou particionamento da rede. Um dos problemas em manter informações replicadas é que inconsistências podem ser geradas se ocorrerem falhas durante as operações de atualização. Existem vários métodos que visam resolver os problemas de inconsistência, cada um deles possui vantagens e desvantagens. Entretanto, a escolha de qual método utilizar em um determinado sistema dependerá das suas necessidades.

Uma das dificuldades de nosso projeto foi encontrar um método que melhor se adequasse às suas necessidades. Diante da análise de cada um deles, nós utilizamos algumas características de alguns dos métodos existentes, tais como, uso de *logs*, versões, protocolos de consistência e concorrência para garantir a consistência da informação replicada.

Referências Bibliográficas

[Albitz e Liu 1997] Albitz, P. Liu, C., *DNS and BIND*, O'Reilly & Associates, Inc., 2ª edição, USA, 1997.

[Alsberg e Day 1976] Alsberg, P. P. *A Principle for Resilient Sharing of Distributed Databases*. 2nd International Conference on Software +Engineering, pp. 562-570, 1976.

[Bernstein et al. 1984] Bernstein, T. Masinter, L. McCahill, M., *Uniform Resource Locators*, RFC1738, CERN, Xerox Corporation, University of Minnesota, dezembro 1994.

[Bernstein et al. 1986] Bernstein, T. Fielding, R. T. Nielsen, H. F., *Hypertext Transfer Protocol - HTTP 1.0*, RFC 1945, MIT/LCS, UC Irvine, maio 1996.

[Bernstein et al. 1987] Bernstein, P. A. Hadzilacos, V. Goodman, N., *Consistency Control and Recovery in Databases Systems,* Addison-Wesley, 1987.

[Berners-Lee et al. 1994] Berners-Lee, T. Cailiau, R. Luotonen, A. Nielsen, H. F. Secret, A., *The World-Wide Web*, Communications of the ACM, 37(8), pp. 76-82, agosto de 1994.

[Blaustein et. al. 1983] Blaustein et. al. *Maintaining Replicated Databases Even in the Presence of Network Partitions*, In Proceedings of the 11th IEEE International Conference on Distributed Computing Systems, pp. 353 - 360, IEEE, New York, setembro de 1983.

[Ceri et al.] Ceri, S. Houtsma, M. A. W. Keller, A. M. Samarati, P. *Achieving Incremental Consistency among Autonomous Replicated Database,*. URL: http://www-db.stanford.edu

/pub/Keller.

[Coulouris et al. 1996] Coulouris, G. Dollimore, J. Kindberg, T., *Distributed Systems: Concepts and Design*, Second Edition, Queen Mary and Westfield College University of London, Addison-Wesley, 1996.

[Date 1985] Date, C. J., *An Introduction to Database Systems*. Addison-Wesley Publishing Company, Vol. II, julho 1985.

[Davidson 1984] Davidson, S. B. *Optimism and Consistency in Partitioned Distributed Database Systems*, ACM Transactions on Database Systems, 9 (3): 456-481, setembro 1984

[Davidson et al. 1985] Davidson, S. B. Garcia-Molina, H. Skeen, D. *Consistency in Partitioned Networks*, ACM Computing Surveys, vol. 17, No. 3, pp. 341 - 370, setembro 1985.

[Davcev 1989] Davcev, D. *Dynamic Voting Scheme in Distributed Systems*. IEEE Transactions on Software Engineering, 15 (1): 93 - 97,

janeiro 1989.

[Eastlake e Kaufman 1997] Eastlake, D. Kaufman, C., *Domain Name System Security Extensions*, RFC 2065, CyberCash, Inc., janeiro 1997.

[Eastlake 1997] Eastlake, D., *Secure Domain Name System Dynamic Update,* RFC 2137, CyberCash, Inc., abril 1997.

[Fielding et al. 1997] Fielding, R. T. Gettys, J. Mogul, J. C. Nielsen, H. F. Berners-Lee, T., *Hypertext Transfer Protocol - HTTP 1.1*, RFC 2068, DEC - Digital Equipament Corporation, MIT/LCS, janeiro 1997.

[Fonsêca 1998] Fonsêca, T. E. A., *Projeto e Implementação de um Servidor Web Proxy com Suporte a Recursos Espelhados*, Dissertação de Mestrado da Coordenação de Pós-graduação em Informática, UFPB, CG, abril de 1998.

[Gifford 1979] Gifford, D. K. *Weighted Voting for Replicated DataI.* 7th ACM Symposium on Operation Systems, pp. 150 - 162, dezembro 1979.

[Gundavaram 1996] Gundavaram, S., *CGI Programmind on the World Wide Web*, O'Reilly & Associates, Inc., 1ª edição, março 1996.

[Iannella et al.] Iannella, R. Sue, H. Leong, D., *BURNS: Basic URN Service Resolution for the Internet*, Research Data Network Cooperative Research Centre - RDN CRC, University of Queensland - Australia, URL: http://eee.dstc.edu.au/RDU/

reports/Apwebp6, 1996.

[Ingham et al. 1996] Ingham, D. Little, M. Caughey, S., *Fixing the "Broken-Link" Problem: The W3Objects Approach*,5th International World Wide Web Conference, Paris, France, maio 1996. URL: http://arjuna.ac.uk:80/w3objects/papers/www5/Overview.html.

[Jalote 1994] Jalote, P., *Fault Tolerance in Distributed Systems,* PTR Prentice Hall, 1994.

[Jajodia e Mutchler 1989] Jajodia, S. Mutchler, D. *A.., Pessimistic Consistency Control Algorithm for Replicated Files which Achieves High Availability*, IEEE Transaction on Software Engineering, vol. 15, No. 1, pp. 39- 46, janeiro 1989.

Laurie e Laurie 1997 Laurie, B. Laurie, P., *Apache - The Definitive Guide*, O'Reilly & Associates, Inc., 1ª edição, USA, 1997.

[Moats 1997] Moats, R., *URN Syntax*, RFC 2141, AT & T, maio 1997.

[Parker et al. 1983] Parker et al., *Detection of Mutual Inconsistecy in Distributed Systems*. IEEE Transaction on Software Engineering, vol. SE-9, No. 3, pp. 240 - 247, maio 1983.

[Vixie et al. 1997] Vixie, P. Thomson, S. Rekhter, Y. Bound, J., *Dynamic Updates in the Domain Name System (DNS UPDATE)*, RFC 2136, abril 1997.