UM SERVIDOR DE ARQUIVOS MULTIVERSÃO REPLICADOS

*Fernando Lopes Pedone Júnior**
Colégio Técnico Industrial - Universidade do Rio Grande
e-mail: ctiflp@cpd.furg.br

Antônio Marinho Pilla Barcellos
Cláudio Fernando Resin Geyer
Instituto de Informática - Universidade Federal do Rio Grande do Sul
e-mail: {marinho, geyer}@inf.ufrgs.br

ABSTRACT

Distributed configurations, where nodes may act independently, offer potential for fault tolerance through data replication. On the other hand, they imply an additional complexity to maintain information consistency.

This work proposes a replicated multiversion file server that aims to provide high data availability and operation atomicity. The file server explores the potential of distributed architectures and, at the same time, avoids that inconsistencies make system utilisation more difficult. In order to measure the complexity in system design, a prototype of the distributed file server was implemented.

Key-words: distributed file systems; fault tolerance.

1. INTRODUÇÃO

Arquiteturas distribuídas propiciam serviços com características não presentes em ambientes centralizados, como por exemplo alta disponibilidade. A idéia básica é que, na presença de falha em uma máquina, as demais podem assumir o serviço da faltosa, e o sistema continua a fornecer seus serviços. Em sistema de arquivos, este aumento de disponibilidade é introduzido com a replicação dos dados.

Por outro lado, características de falha independentes dos componentes de um sistema distribuído podem levar a situações indesejadas, como ocorre com um cliente que deixa um arquivo parcialmente alterado porque uma falha na sua máquina impediu que ele concluisse o seu processamento.

Neste trabalho apresenta-se uma proposta de servidor de arquivos que aborda os dois aspectos descritos através de mecanismos de replicação e atomicidade.

2. Tolerância a Falhas em Sistemas de Arquivos

Um sistema é considerado *tolerante a falhas* se na presença de falha em algum(ns) de seus componentes ele [CRI 91]:

(a) não deixa que o usuário perceba os defeitos resultantes de tais falhas, e/ou

(b) apresenta um comportamento de defeito bem definido.

A capacidade de um sistema de esconder de seus usuários as falhas nos seus componentes está normalmente associada ao seu nível de disponibilidade. Disponiblidade é um parâmetro que corresponde à medida do fornecimento adequado de um serviço (segundo especificado como comportamento normal) quando se considera a vida do sistema. Um sistema altamente disponível é aquele que durante a maior parte do tempo se comporta adequadamente [WEB 90]. Defeitos, resultantes de falhas nos componentes, normalmente levam o sistema a um comportamento inadequado. Aumento de disponibilidade é obtido fazendo-se com que, mesmo na presença de falhas, o sistema se comporte de maneira adequada. Dentro do contexto de um sistema de arquivos, cuja função básica é processar requisições de leitura e escrita envolvendo memória secundária, aumento de disponibilidade é obtido utilizando-se técnicas de replicação de dados. Desta forma, mesmo que uma das réplicas não esteja disponível, o sistema pode utilizar as demais para processar as requisições.

A simples replicação dos dados ameniza o problema de falha nos servidores ou em seus componentes, mas não resolve o problema de falha nos clientes. Por exemplo, um cliente que não consegue concluir as suas alterações em um arquivo, devido a uma falha na sua máquina, poderá deixar o arquivo em um estado intermediário (indefinido) entre o início e o término do processamento. Em um sistema tolerante a falhas essa situação (correspondente ao item **(b)** anteriormente apresentado) deve ser considerada e devidamente tratada, de maneira que ou todas as operações do cliente sejam aceitas e processadas ou então nenhuma. Este comportamento é característico de *ações atômicas*, que no caso de um sistema de arquivos podem ser delimitadas pelas operações de abertura e fechamento de arquivo.

Na abordagem acima descrita, os clientes passam a enxergar os arquivos como uma seqüência de estados consistentes altamente disponíveis, sendo cada estado gerado atomicamente a partir de um anterior. Esta abordagem é diferente da convencional Unix, onde processos concorrentes enxergam as alterações parciais uns dos outros.

3. ARQUIVOS MULTIVERSÃO

Uma forma de fazer com que o usuário compreenda o processo de manipulação atômica dos dados é fazer com que os arquivos sejam realmente vistos como seqüências ordenadas de estados ou versões consistentes. *Arquivos multiversão* [MUL 86] são uma forma de operacionalizar tal abordagem. Arquivos multiversão utilizam a técnica de *shadow pages* para implementar atomicidade. Quando um processo abre um arquivo, ele recebe uma cópia idêntica à versão mais nova do arquivo solicitado. Todas as suas requisições de leitura e escrita utilizarão esta *cópia privada* (CP). Escritas alterarão somente a cópia privada e leituras retornarão dados da mesma. Isto garante que todas as escritas do processo serão percebidas de uma única vez (i.e., atomicamente) durante o fechamento do arquivo. Assim sendo, é simples o tratamento de uma falha em uma máquina executando o processo cliente ou o processo servidor que mantém a cópia privada. Em ambos os casos, basta a eliminação da cópia privada[1] . Se não houver falhas e o arquivo for normalmente fechado, então a cópia privada dará origem a uma nova versão do arquivo.

Considerando-se apenas processamento seqüencial, a semântica de arquivos multiversão parece ser a solução ideal para lidar com falhas em sistemas de arquivos. Entretanto, situações de processamento concorrente são possíveis, sendo uma maneira de tratá-las impedir operações de escritas concorrentes através do uso de *locks*. Desta forma, somente um processo consegue modificar o arquivo e os demais ficam bloqueados esperando pelo primeiro. Esta solução não foi adotada por dois motivos. Primeiro, considera-se um mecanismo automático de *lock* de arquivos demasiadamente restritivo para um sistema de arquivos, uma vez que *locks* muitas vezes limitam a concorrência em situações desnecessárias [KUN 81]. Além disso, o uso de *locks* implica adicionar ao sistema a lógica necessária a sua gerência e mais à detecção e resolução de *deadlocks*.

Sendo então possível a execução concorrente de escritores, optou-se por acomodar as alterações geradas por cada um deles segundo o *critério de seriação*. Freqüentemente utilizado em bancos de dados, o critério de seriação consiste em verificar se a execução concorrente de duas ou mais transações é equivalente a uma execução seqüencial envolvendo tais transações. Desta forma, quando um processo fecha um arquivo, se a versão da qual a sua cópia privada deriva ainda é a corrente, uma nova versão é imediatamente gerada. Caso contrário, verifica-se, segundo o critério de seriação, se as suas alterações podem ou não gerar uma nova versão. Se não for possível, então o processo deve refazer sua computação.

O teste de seriação é baseado nos conjuntos de leitura (RS) e escrita[2] (WS) dos processos concorrentes. Se n escritores executam concorrentemente, então para que o *n-ésimo* escritor seja aceito após os $n-1$ já terem sido, é necessário que

$$(WS_{p_1} \cup WS_{p_2} \cup ... \cup WS_{p_{n-1}}) \cap RS_{p_n} = \emptyset.$$

---

1 no caso de falha no servidor a cópia privada é eliminada no procedimento de *recovery*.

2 os conjuntos de leitura e escrita armazenam quais ítens do arquivo foram lidos ou escritos, respectivamente.

4. ARQUIVOS MULTIVERSÃO REPLICADOS

A junção de arquivos multiversão e esquemas de replicação resulta nos *arquivos multiversão replicados*. O que se pretende com a fusão destes dois mecanismos em um único é que este apresente as características de cada um daqueles individualmente: **atualizações atômicas** e **alta disponibilidade**.

A idéia básica de arquivos multiversão replicados, apresentada neste trabalho, é replicar as versões dos arquivos em diversos servidores. A gerência de réplicas em um ambiente distribuído é um problema conhecido e várias soluções já foram propostas [SIN 94]. Tais soluções podem ser divididas em dois grupos, conforme o uso ou não de votação [GOS 91]. *Votação balanceada* [GIF 79] foi escolhida por ser um método descentralizado e por fornecer a flexibilidade necessária em um sistema distribuído inerentemente heterogêneo.

Na votação balanceada cada réplica possui um número de votos. Para ler um arquivo o usuário necessita reunir um quorum de leitura de q_r votos, e para escrever um quorum de escrita de q_w votos. Considerando N o número total de votos, duas condições devem ser observadas para que o algoritmo funcione adequadamente:

(1) $q_r + q_w > N$, que garante que uma leitura sempre utilizará a versão mais recente do dado; e

(2) $q_w > N/2$, que garante que escritas concorrentes serão sempre detectadas.

A gerência de versões é feita através do uso de números de versões associados a cada objeto.

4.1. Funcionamento Básico

Em arquivos multiversão replicados a gerência das réplicas ocorre na abertura e no fechamento dos arquivos. Quando um arquivo é aberto o servidor deve descobrir qual a sua versão corrente contactando os *membros de um quorum*[3] de leitura. Caso a versão corrente não esteja disponível localmente, ele deve solicitá-la a um servidor que a possua. Tendo a versão corrente do arquivo, o processamento é semelhante ao de arquivos multiversão, ou seja, cria-se uma cópia privada que será então utilizada pelo processo cliente.

Quando um processo cliente envia uma requisição para o fechamento de um arquivo, o *servidor principal* (aquele que armazena a cópia privada do cliente) distribui as alterações geradas na cópia privada a um grupo de servidores. Estes servidores, juntamente com o servidor principal, constituem os membros do quorum de escrita. Cada membro verifica individualmente se pode ou não aceitar as alterações recebidas e informa o resultado ao servidor principal. Esta troca de mensagens constitui a primeira fase do protocolo de atualização (a seguir descrito). Uma vez obtida a confirmação de todos os membros do quo-

[3] membros de um quorum são os elementos de um conjunto de servidores cuja soma do número de votos das réplicas é maior ou igual ao quorum.

rum, a atualização pode ser efetuada. Cada membro é avisado pelo servidor principal sobre o resultado da votação (segunda fase do protocolo).

O protocolo de geração de novas versões deve ser atômico: todos os participantes do quorum devem entrar em acordo sobre a geração ou não de uma nova versão. A não observação deste aspecto pode levar a inconsistências no sistema. Por exemplo, considere um quorum constituído de seis participantes. Se o servidor envia uma mensagem para criação de uma nova versão a dois membros e falha antes de enviar aos demais, menos da metade dos servidores terão sido atualizados, o que contraria a segunda condição do algoritmo de votação, i.e., $q_w > N/2$.

4.2. Uso Compartilhado

A alteração simultânea de um mesmo arquivo (por múltiplos escritores concorrentes) é possível em ambientes distribuídos. Na figura 1 isto ocorre com a cópia privada CP_1 no servidor S_2, com CP_2 em S_3 e com CP_3 em S_4. Fundamentalmente o problema é o mesmo que ocorre em arquivos multiversão centralizados, logo é possível empregar a mesma solução. A diferença principal é que neste caso as cópias privadas estão distribuídas. Portanto, para que uma nova versão possa ser gerada sem inconsistências o teste de seriação deve ser executado por cada membro do quorum de escrita. Isso permite que se duas ou mais cópias privadas forem submetidas simultaneamente pelo menos um membro do quorum perceberá e poderá verificar se suas execuções são ou não equivalentes a uma execução seqüencial.

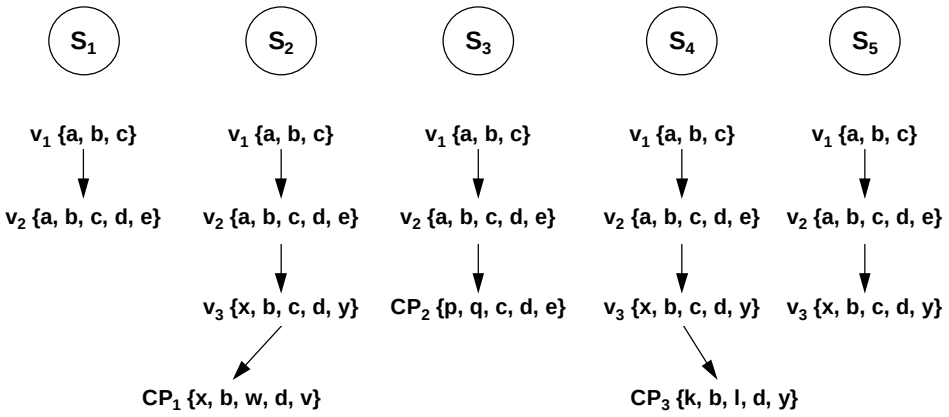

Fig. 1. Cinco réplicas de um arquivo. Cada réplica possui um voto e os quoruns de leitura e escrita são de três votos. Todos os servidores possuem as versões v_1 e v_2. A versão v_3 foi gerada em S_2, S_4 e S_5. Ainda estão em uso as cópias privadas CP_1, CP_2 (iniciada antes da geração da versão v_3) e CP_3. Repare que todos os três alteram as suas cópias privadas.

4.3. Visão Parcial dos Arquivos

No exemplo da figura 1, quando a cópia privada CP_1 for submetida para a geração de uma nova versão, o servidor S_2 deve escolher quaisquer dois outros servidores que estejam em funcionamento para participarem do quorum de escrita. Se S_1 e S_5 forem selecionados, então S_2 deve enviar-lhes as alterações para

a composição da nova versão. Como CP_1 derivou da versão v_3, o servidor S_5, juntamente com as alterações de CP_1, pode gerar a nova versão v_4. Entretanto, o mesmo não ocorre com S_1, pois este não possui a versão v_3 (não participou do quorum de escrita de v_3) e portanto não pode gerar v_4. Situação semelhante ocorrerá com CP_2, pois, se esta cópia privada passar no teste de seriação distribuído, nem mesmo o servidor que a contém (S_3) poderá gerar a versão v_5, visto que ele não possui as versões v_3 e v_4.

A solução para estes problemas deve ser geral (resolver ambos os problemas acima), de baixo custo (o algoritmo para manipulação de quoruns já introduz um custo elevado na geração de novas versões) e não deve causar perda de consistência dos dados. Partindo de uma observação simples, descrita abaixo, será mostrado que o mecanismo de visão parcial satisfaz estes requisitos.

Quando o servidor S_2 distribui os dados de CP_1, o que ele realmente precisa é que um quorum de servidores armazene estas informações para que, quando necessário, a versão v_4 esteja disponível. Não importa se S_1 armazenará toda a versão v_4 ou apenas as modificações em relação à v_3. O que realmente é preciso é que estas informações possuam a mesma disponibilidade da versão v_3, para que v_4 possa ser construída quando solicitada. O resultado disso é que, mesmo não armazenando as versões anteriores, o servidor guarda as alterações da nova versão e passa assim a possuir uma ***visão parcial*** do arquivo.

Como pode ser facilmente constatado, a solução de visão parcial é geral. Pode igualmente ser considerada de baixo custo porque não acrescenta nenhuma mensagem ao algoritmo de votação e nem aumenta o tamanho das mensagens. Finalmente, não ocasiona perda da consistência já que os quoruns continuam sendo obedecidos.

Ainda resta mostrar que o algoritmo de visão parcial não compromete o uso dos arquivos, ou seja, ainda é possível que a versão corrente (completa) seja obtida a partir de um quorum de leitura, possivelmente reunindo-se vários fragmentos do arquivo. Considere que S_i é o conjunto de alterações geradas na criação de uma versão, com $1 \leq i \leq n$ e n sendo a versão corrente. S_i foi gerado utilizando-se um quorum de escrita, logo a existência de um quorum de leitura garante o acesso ao mesmo. Estando então disponíveis os segmentos S_1, ..., S_n a versão corrente completa pode ser formada, já que esta é composta de todas as escritas geradas no arquivo.

4.4. Planos de Trabalho

Uma analogia com planos de trabalho facilita a compreensão da proposta de visões parciais. Utilizando-se a abordagem de planos de trabalho, cada versão (ou melhor o conjunto de escritas gerados em uma versão) constitui um plano. Os planos são sobrepostos segundo a ordem de criação, ou seja, o plano i está sobre o plano j se $v_i > v_j$. Segmentos da versão que não foram escritos funcionam como janelas para os planos de baixo, e quando um servidor não dispõe da versão v_i, o plano i é uma grande janela para os planos abaixo dele.

Desta forma, a visão que um servidor tem de um arquivo é obtida olhando-se para o plano de cima. Quando uma versão que não está presente é obtida o que ocorre é que ela encobre as informações presentes nos planos de baixo

(atualiza estas informações). A figura 2 apresenta o mesmo exemplo da figura 1 considerando-se planos de trabalho.

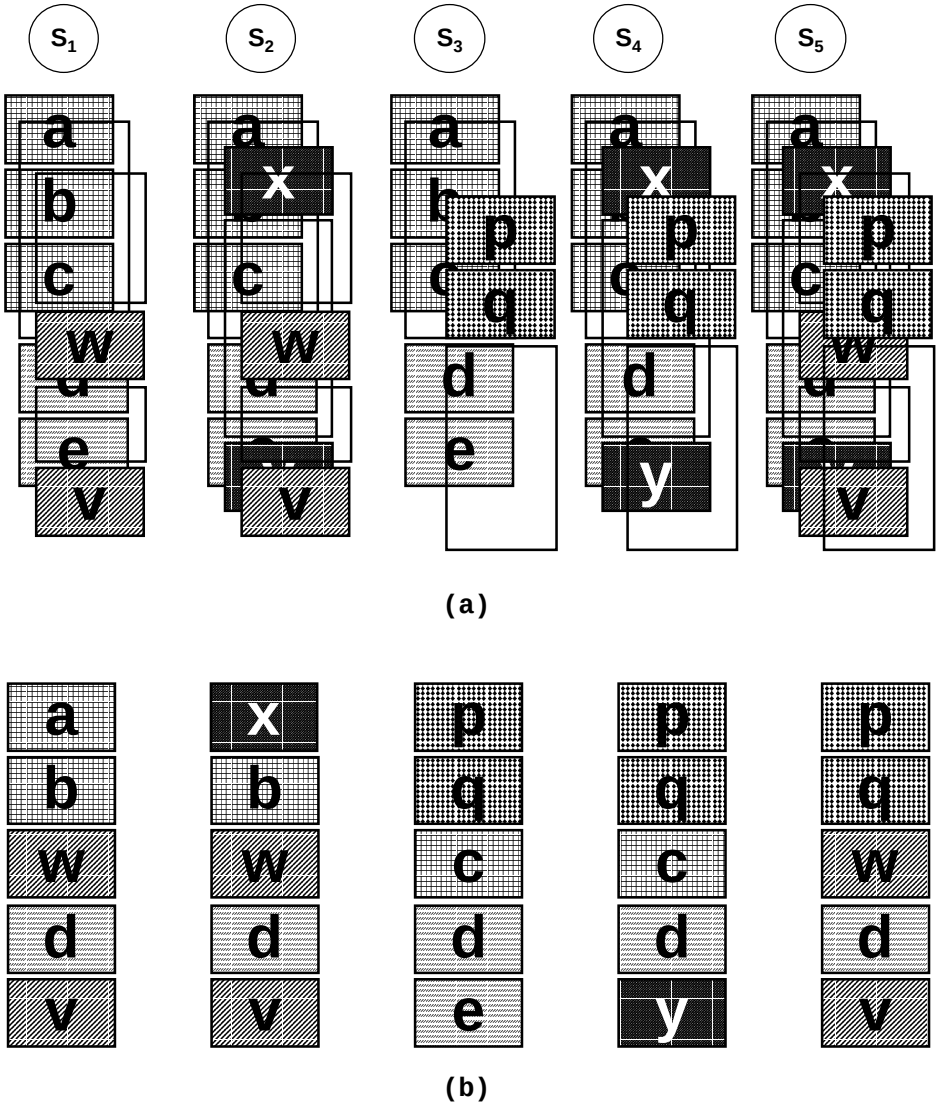

Fig. 2.A abordagem de planos de trabalho. (a) Planos de trabalho representando a figura 1 após terem sido submetidas e aceitas as cópias privadas CP_1 e CP_2. CP_1 utilizou os servidores S_1, S_2 e S_5. CP_2 utilizou S_3, S_4 e S_5; **(b)** visão que cada servidor possui do arquivo.

4.5. Composição da Versão Corrente

Um servidor não pode criar uma cópia privada para um processo baseando-se em uma visão parcial do arquivo. É necessário antes que ele possua a versão corrente completa. Este processo de atualização envolve o envio de mensagens pelo servidor aos participantes de um quorum de leitura, informando-os qual a sua visão do arquivo - quais versões estão presentes. Cada participante responde dizendo de que forma pode contribuir (quais versões a mais ele possui). Com estas informações o servidor principal pode determinar a melhor maneira de construir a versão corrente dentre os participantes do quorum de leitura.

Parâmetros estáticos e dinâmicos podem ser utilizados na determinação dos locais de onde serão trazidas as versões necessárias. Em uma rede de porte considerável, composta de muitos computadores agrupados em subredes, o servidor principal pode considerar a proximidade física na escolha de um local. Outra possibilidade, baseada em informações dinâmicas, é fazer com que cada participante informe, juntamente com as versões locais, um valor indicativo da atual carga de trabalho (p. ex. número de requisições processadas nos últimos instantes, ou número de cópias privadas em uso). Além disso, no processo de construção de uma visão completa é útil tentar trazer versões subseqüentes de um mesmo servidor. Primeiro porque quanto menor o número de servidores envolvidos menor o número de mensagens. Segundo, porque desta forma somente a versão mais nova de um objeto é enviada, caso ele tenha sido modificado em mais de uma versão (na determinação da versão corrente completa só interessam os dados mais recentes[4]).

O processamento envolvido na atualização da versão corrente pode atrasar consideravelmente a resposta a uma requisição de abertura de arquivo caso o servidor necessite de muitas versões (p. ex. de todo o arquivo). Uma forma de amenizar este processo é fazer com que cada servidor, em determinados intervalos de tempo e quando a carga de trabalho estiver baixa, atualize a sua visão dos arquivos.

5. IMPLEMENTAÇÃO

Com o propósito de avaliar a complexidade envolvida na implementação do servidor de arquivos descrito, está sendo implementado um protótipo do sistema. A plataforma de hardware está baseada em um conjunto de estações de trabalho Sun SPARC interligadas por uma rede Ethernet. Cada estação utiliza o sistema SunOS 4.1.1 como sistema operacional nativo e o sistema de rede heterogêneo HetNOS [BAR 93] como sistema de comunicação entre processos. Os servidores de arquivo, bem como o HetNOS, executam a nível de aplicação Unix. A figura 3 ilustra esta estruturação.

5.1. HetNOS

O sistema operacional de rede heterogêneo HetNOS *(Heterogeneous Network Operating System)* é composto por um conjunto de camadas dispostas sobre sistemas operacionais nativos cujo objetivo é propiciar uma plataforma homogênea de desenvolvimento de aplicações distribuídas. O conjunto de máquinas integradas pelo HetNOS é visto pelo usuário como uma máquina virtual distribuída, tanto a nível de ferramentas como chamadas de sistema.

A comunicação entre processos no HetNOS é de alto nível, pois está baseada em um espaço global de nomes simbólicos independente de localidade, simplificando a implementação do protótipo. O paradigma de comunicação é a troca de mensagens, tendo sido utilizado apenas o conjunto de primitivas síncronas para o desenvolvimento do projeto.

---

[4] esta abordagem considera que os processos estão sempre interessados na versão corrente do arquivo, e nunca em uma versão anterior. Alguns sistemas, entretanto, não funcionam dessa forma (p.ex., vide [GIF 88]).

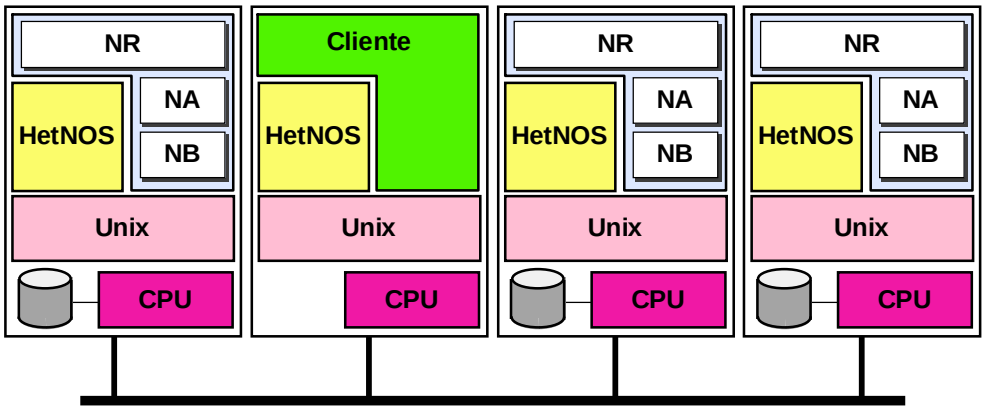

Fig. 3. Estrutura do servidor de arquivos multiversão replicados. Exemplo de configuração de hardware e software envolvendo três servidores e uma aplicação. O sistema de arquivos é constituído dos módulos Nível de Replicação (NR), Nível de Arquivo (NA) e Nível de Bloco (NB).

5.2. Nível de Bloco

O Nível de Bloco desempenha a tradicional função de gerência de disco, tratando suas peculiaridades e idiossincrasias de maneira a prover o nível superior com um espaço unidimensional de blocos de tamanho fixo a partir de um espaço tridimensional (cilindro, superfície e setor) característico de tais meios de armazenamento.

Por se tratar de um protótipo executando em modo usuário, o Nível de Bloco faz na verdade uma simulação do dispositivo de disco utilizando arquivos Unix. Para aproximar-se do comportamento real de um disco, garante-se, através da chamada *fsync()*, que as escritas sejam síncronas.

5.3. Nível de Arquivo

O Nível de Arquivo implementa os arquivos multiversão centralizados e todas as chamadas relacionadas a eles. Existem operações para criação de cópias privadas, leitura e escrita de dados, geração de novas versões e verificação de seriação.

Um arquivo é constituído de um conjunto de blocos de tamanho fixo, cada um possuindo um número de versão associado (indicando a versão na qual o bloco foi gerado). Cada arquivo possui um *mapa de blocos*, que armazena os ponteiros de localização dos blocos em disco bem como seus números de versão. Esta estrutura facilita o teste de seriação e reduz o espaço em disco necessário para armazenar o arquivo já que apenas a versão mais nova de um bloco é mantida.

Cada cópia privada é constituída de um conjunto de blocos, cada bloco possuindo *flags* de leitura e de escrita *(dirty bit)* para indicar o tipo de acesso realizado sobre o mesmo. O teste de seriação utiliza essas informações

em conjunto com o mapa de blocos da versão corrente do arquivo para verificar se uma nova versão pode ser gerada.

Dois mecanismos são usados para aumentar o desempenho da gerência de cópias privadas. O primeiro consiste em aproveitar a grande quantidade de memória (possivelmente virtual) usualmente presente nas estações de trabalho, possibilitando ao Nível de Arquivo manter os blocos e tabelas relacionadas com cópias privadas a maior parte do tempo em memória. Estas informações são forçadas para disco somente no fechamento do arquivo. O outro mecanismo utiliza a técnica de *copy-on-write* [MUL 86] na criação de cópias privadas, reduzindo a alocação de blocos somente aos realmente necessários (atualizações).

O Nível de Arquivo implementa ainda a abstração de *armazenamento estável* [LAM 81] utilizado pelo Nível de Replicação para o armazenamento de tabelas e registros de *log*, sendo este último usado no protocolo de encerramento atômico.

5.4. Nível de Replicação

Toda a interação entre cliente e servidor (envio de requisições e respostas) e entre os próprios servidores (gerência de quoruns de leitura e escrita) é feita no Nível de Replicação.

O Nível de Replicação se preocupa em reunir as informações necessárias para compor a versão corrente do arquivo (seção 4.5), além de distribuir as informações para o teste de seriação (seção 4.2) e para a geração atômica de novas versões (seção 4.1). O protocolo de atualização atômica utilizado é o *two-phase-commit* [GRA 93].

6. CONCLUSÃO

Este trabalho apresentou um sistema de arquivos tolerante a falhas baseado no uso de operações atômicas e replicação de dados.

Espera-se com esses mecanismos, em primeiro lugar, facilitar o tratamento de falhas durante a utilização dos arquivos e aumentar o nível de disponibilidade. Em segundo lugar, o mecanismo de arquivo multiversão simplifica o projeto da *cache* dos clientes, uma vez que uma cópia privada pode ser considerada um dado imutável (não há problemas com a manutenção da consistência). Finalmente, espera-se o aumento de desempenho porque cópias privadas podem ser enviadas de uma só vez aos clientes. Como todo o processamento subseqüente pode ser realizado de forma local, diminui-se a carga do servidor e o tráfego de mensagens na rede.

Em um primeiro momento, planeja-se a adaptação do modelo para que o mesmo dê origem a um sistema de arquivos implementado sobre o sistema Mach [RAS 89]. Após então este servidor será integrado a um *microkernel* desenvolvido exclusivamente para uma máquina distribuída conectada através de *transputer links*. O desenvolvimento do hardware e do software para tal máquina faz parte de um consórcio que integra diversos centros de pesquisa da região sul do Brasil.

BIBLIOGRAFIA

[ACC 86] ACCETA, M. et alli. Mach: a new kernel foundation for unix development. In: Summer'1986 Usenix Conference, **Proceedings**, p.93-112, July 1986.

[BAR 93] BARCELLOS, A.M.P. Projeto do Sistema Operacional de Rede Heterogêneo HetNOS. In.: Seminário Integrado de Software e Hardware, 20 (XX SEMISH). **Anais**. Florianópolis, SC. set. 1993. Rio de Janeiro, SBC, 1993. p.718-732.

[CRI 91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v.34, n.2, pp.56-70, Feb. 1991.

[GIF 79] GIFFORD, D.K. Weighted Voting for Replicated Data. In: Symposium on Operating Systems Principles, 7, **Proceedings**, Pacific Grove, p.150-162, Dec. 1979.

[GOS 91] GOSCINSKI, A. **Distributed Operating Systems: The Logical Design.** Addison-Wesley, 1991, 913 p.

[GRA 93] GRAY, J.; REUTER, A. **Transaction Processing: Concepts and Techniques.** Morgan Kaufmann, San Mateo, 1993, 1070 p.

[KUN 81] KUNG, H.T.; ROBINSON, J.T. On Optimistic Methods for Concurrency Control. **ACM Transactions on Database Systems**, v.6, n.6, pp.213-226, June 1981.

[LAM 81] LAMPSON, B.W. Atomic Transactions. In: Distributed Systems - Architecture and Implementation. Spring-Verlag, West Berlin, 1981. pp.246-265.

[SIN 94] SINGHAL, M.; SHIVARATRI, N. **Advanced Concepts in Operating Systems: Distributed, Database, and Multiprocessor Operating Systems.** McGraw-Hill, 1994, 522p.

[MUL 86] MULLENDER, S.J.; TANENBAUM, A.S. The Design of a Capability-Based Distributed Operating System. **The Computer Journal**, v.29, n.4, p.289-299, Apr. 1986.

[WEB 90] WEBER, T.W.; JANSH-PÔRTO, I.E.S.; WEBER, R.F. **Fundamentos de Tolerância a Falhas.** IX Jai. X Congresso SBC, 22 a 27 julho 90, Vitória. 75p.