Um Repositório de Objetos para um Ambiente de Programação Distribuída

Marco Aurélio Medina de Oliveira

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Repositório de Objetos para um Ambiente de Programação Distribuída

Marco Aurélio Medina de Oliveira

Dezembro de 1997

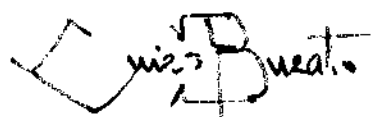
Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Prof. Dr. Alcides Calsavara
  (Departamento de Informática–PUC/PR)
- Prof. Dr. Célio C. Guimarães
  (Instituto de Computação—UNICAMP)
- Prof. Dr. Edmundo R. Madeira (suplente)
  (Instituto de Computação—UNICAMP)

Um Repositório de Objetos para um Ambiente de Programação Distribuída

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marco Aurélio Medina de Oliveira e aprovada pela Banca Examinadora.

Campinas, 22 de dezembro de 1997.



Prof. Dr. Luiz Eduardo Buzato
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Tese de Mestrado defendida e aprovada em 15 de dezembro de 1997 pela Banca Examinadora composta pelos Professores Doutores

Prof. Dr. Alcides Calsavara

Prof. Dr. Célio Cardoso Guimarães

Prof. Dr. Luiz Eduardo Buzato

Agradecimentos

A meus amigos, que fizeram a minha estada em Campinas agradável.

A meus irmãos, Lincoln e Flávio, pelo apoio e amizade.

A Cristina, por ter estado comigo nos bons e maus momentos.

A meus pais, Lauro e Aparecida, por terem me dado a vida, educação e, principalmente, amor.

A Deus, por nunca ter me desamparado nos momentos mais difíceis.

Resumo

O objetivo desta dissertação é a otimização do mecanismo de persistência do estado de objetos em um sistema de programação distribuída baseado em ações atômicas chamado Arjuna.

O mecanismo atual de persistência em Arjuna tem um desempenho insatisfatório em termos de velocidade. Isto deve-se a utilização do sistema de arquivos hospedeiro (Unix) como repositório de objetos de forma inadequada. A provisão de ações atômicas é obtida através da técnica de cópias *shadow* sobre o repositório de objetos.

A construção do repositório de objetos HiPer Store (*High Performance Store*) otimiza a persistência através da utilização de mecanismos de *log* e gerência de memória. Estes mecanismos já são utilizados em outros tipos de aplicações que suportam ações atômicas, como bancos de dados, e mostram-se eficientes também em Arjuna.

Abstract

The purpose of this dissertation is the optimization of the object state persistence mechanisms in a distributed programming system based on atomic actions called Arjuna.

At this moment, the persistence mechanism in Arjuna has not a satisfactory performance about speed. This is due the utilization of host file system (Unix) as object store in an inefficient way. Arjuna obtains atomic action support using the shadow copies technique over his object store.

The HiPer Store (High Performance Store) implementation optimizes the persistence using mechanisms like log and memory manager. This mechanisms are also used in other applications that support atomic actions, like databases, and has also a good performance in Arjuna.

Conteúdo

Agradecimentos v

Resumo vi

Abstract vii

1 Introdução 1
- 1.1 Contribuições . . . 2
- 1.2 Organização . . . 2

2 Fundamentos 4
- 2.1 O Modelo de Objetos e Ações . . . 4
  - 2.1.1 Objetos . . . 4
  - 2.1.2 Ações Atômicas . . . 5
  - 2.1.3 Persistência e o Modelo de Objetos e Ações . . . 5
- 2.2 *Log* . . . 6
- 2.3 Gerência de Memória . . . 9

3 Trabalhos Relacionados 13
- 3.1 Bancos de Dados . . . 13
  - 3.1.1 ENCORE-ObServer . . . 13
  - 3.1.2 O2 . . . 14
  - 3.1.3 POSTGRES . . . 15
  - 3.1.4 EXTRA . . . 15
  - 3.1.5 SHORE . . . 16
  - 3.1.6 ObServer2 . . . 18
- 3.2 Bancos de Dados Residentes em Memória . . . 19
  - 3.2.1 Características de BDRMs . . . 20
  - 3.2.2 Propostas de Bancos de Dados Residentes em Memória . . . 23
- 3.3 *Log Structured File System (LFS)* . . . 28

3.4 Análise dos Trabalhos Relacionados 29

4 Ambiente Alvo **31**
4.1 Arquitetura Geral 31
4.2 Arjuna: Repositório de Objetos 33
4.2.1 Chamadas ao Repositório 34
4.2.2 `ObjectStore` 38
4.2.3 `VolatileStore` 39
4.2.4 `FileSystemStore` 40
4.2.5 `MappedFile` 40
4.2.6 `SingleTypeMappedStore` 41
4.2.7 `MappedStore` 41
4.2.8 `ShadowingStore` 42
4.2.9 `ActionStore` 43
4.2.10 `FdCache` 43
4.2.11 `FragmentedStore` 44
4.2.12 `HashedStore` 46
4.3 Conclusões sobre os Repositórios de Arjuna 46

5 *HiPer Store* (*High Performance Store*) **48**
5.1 Arquitetura de *Software* 48
5.2 Algoritmo 54
5.3 Otimizações 60
5.4 Exemplo 61

6 Discussão e Conclusões **67**
6.1 Discussão 67
6.2 Conclusões 75

A Principais métodos de *HiPer Store* **76**
A.0.1 `MemoryStore` 76
A.0.2 `LogStore` 79
A.0.3 `DiskStore` 80

Referências Bibliográficas **81**

Lista de Figuras

3.1 Descrição esquemática da arquitetura proposta 26

4.1 A Arquitetura do Arjuna 32
4.2 Ativação de um Objeto Persistente 34
4.3 Máquina de estados do processamento de uma ação com um objeto em Arjuna 35

5.1 Funcionamento da *HiPer Store* 50
5.2 Hierarquia de Classes de *HiPer Store* integrada aos Repositórios Arjuna 51
5.3 Classes associadas a *HiPer Store* 51
5.4 Classe `DiskStore` 52
5.5 Classe `MemoryStore` 53
5.6 Classe `LogStore` 54
5.7 Manutenção de entradas em *Memory Store* 55
5.8 Compactação de *log* em *HiPer Store* 58
5.9 Execução de uma aplicação sobre HiPerStore 63
5.10 Máquina de estados do processamento de ações atômicas em Arjuna utilizando HiPer Store 65

Capítulo 1

Introdução

Uma análise de sistemas de software desenvolvidos recentemente, em particular bancos de dados, sistemas operacionais e ambientes de programação, fornece evidências de que a utilização do paradigma orientado a objetos tem promovido uma aproximação gradual entre esses sistemas, tanto na interface de programação, quanto na sua arquitetura de *software* [4]. Muitos desses sistemas orientados a objetos incluem sub-sistemas para:

- identificação e criação de classes e objetos;
- implementação de mecanismos de herança e hierarquias de classes;
- troca de mensagens (invocação de objetos);
- persistência (repositório de objetos);

Ambientes de programação orientados a objetos costumam implementar repositórios utilizando o sistema de arquivos do sistema operacional hospedeiro, enquanto sistemas de gerência de bancos de dados orientados a objetos utilizam repositórios projetados especialmente para otimizar a manipulação de objetos. Sistemas de arquivos, em geral, organizam arquivos como uma seqüência de blocos de disco. Já bancos de dados costumam organizar seus dados em páginas de dados. A integração de funções encontradas em sistemas de arquivos e em repositórios de objetos de bancos de dados pode beneficiar a manipulação de objetos persistentes realizada em ambientes de programação.

Mudanças na arquitetura/interface do repositório de objetos podem gerar mudanças em outros componentes do ambiente de programação, como por exemplo, nos componentes responsáveis pelo mapeamento entre objetos ativos (residentes em memória) e objetos passivos (residentes em memória estável).

Arjuna [28] é um ambiente de programação que tem seu repositório baseado no sistema de arquivos do Unix. Sua atual implementação não é adequada quando analisado o quesito de velocidade de processamento de ações atômicas [1].Um dos principais pontos passíveis de otimização refere-se a seus mecanismos de persistência e recuperação, mais precisamente a implementação de seu repositório de objetos.

1.1 Contribuições

Fizemos um estudo sobre a implementação do mecanismo de persistência combinado com o mecanismo de recuperação em vários tipos de sistemas, principalmente em bancos de dados. Também analisamos a atual implementação dos vários repositórios de Arjuna que, em última instância, são os responsáveis pela persistência em Arjuna. Este estudo possibilitou uma análise de vantagens e desvantagens da utilização das propostas e modelos de sistemas diversos como alternativa ao atual mecanismo de persistência e recuperação em Arjuna.

Optamos por uma alternativa que consideramos adequada a Arjuna e implementamos um protótipo para melhor visualização dos resultados. Em posse deste protótipo, realizamos uma série de testes que possibilitou a comprovação da melhoria de desempenho esperada com o novo repositório.

1.2 Organização

No Capítulo 2, analisamos alguns fundamentos necessários para a execução desta dissertação. Discutimos o modelo computacional adotado de objetos e ações na Seção 2.1. Nas Seções 2.2 e 2.3, revisamos os mecanismos de *log* e gerência de memória que serão utilizados no restante da dissertação. No Capítulo 3, analisamos os trabalhos relaciona-

[1] ações atômicas, ou simplesmente ações, também são conhecidas como transações

dos e como suas experiências foram relevantes para o desenvolvimento de nossas soluções. No Capítulo 4, vemos o ambiente alvo, analisando mais detalhadamente, na Seção 4.2, o mecanismo de persistência em Arjuna. Na Capítulo 5, apresentamos o repositório *HiPer Store* desenvolvido nesta dissertação. No Capítulo 6, temos a discussão dos resultados obtidos com o novo repositório e as considerações finais da dissertação.

Capítulo 2

Fundamentos

Este Capítulo apresenta um breve estudo a respeito de funcionalidades (objetos e ações) e mecanismos (*log* e gerência de memória) que serão utilizados durante o restante da dissertação.

2.1 O Modelo de Objetos e Ações

O modelo de objetos e ações integra os conceitos de objetos e ações atômicas. Este modelo foi adotado por Arjuna por prover encapsulamento advindo da orientação a objeto e facilidades para controle de concorrência através da utilização de ações atômicas. O fornecimento da abstração de objetos e ações facilita a programação de aplicações distribuídas tolerantes a falhas de *hardware* (queda de processador e falha de comunicação) [10, 22]. Ações permitem que programas distribuídos orientados a objetos sejam estruturados como coleções de objetos cujos métodos são atômicos.

2.1.1 Objetos

Um objeto é uma entidade que encapsula uma **estrutura de dados** e tem um **comportamento** que é determinado por suas **operações (métodos)**. Objetos com comportamento e estrutura de dados similares são agrupados em uma **classe**. Assim, a definição

de uma classe especifica os atributos e operações dos objetos da classe. Cada **instância** de uma classe (um objeto) tem algumas variáveis (**atributos**), determinados pela classe. As operações de um objeto têm acesso a estas variáveis e podem, conseqüentemente, modificar o estado do objeto.

2.1.2 Ações Atômicas

Podemos dizer que uma ação atômica é um processo que faz acesso a uma base de dados compartilhada de forma **atômica**, ou seja:

- cada ação acessa dados compartilhados sem interferir em outras ações, e
- se uma ação termina normalmente, então todos seus efeitos tornam-se permanentes, caso contrário, eles não tem efeito nenhum [2].

Ações podem ocorrer concorrentemente. Por isto, torna-se necessário controle de concorrência e recuperação para garantir atomicidade. As propriedades de ações são: equivalência serial, atomicidade e permanência de efeitos.

2.1.3 Persistência e o Modelo de Objetos e Ações

Para implementar ações atômicas, é necessária a existência de **objetos persistentes**. Objetos persistentes são aqueles que continuam a existir após o término da aplicação que os criou [1].

Objetos persistentes geralmente são implementados com o apoio de um repositório de objetos que reside em memória estável (disco). Os repositórios de objetos guardam o estado de um objeto em um determinado instante. Ações agem sobre os objetos modificando esse estado. Para garantir atomicidade, pode-se manter duas cópias do objeto, uma válida, com o estado anterior ao início da ação, e outra com o estado intermediário. A segunda cópia deve ser atualizada durante a finalização da ação. Após o término da ação, a segunda cópia torna-se a cópia válida. Caso a ação não se conclua, a cópia válida permanece sendo a anterior à transação. Esta técnica é conhecida como **cópia** *shadow* e é adotada em Arjuna.

Como repositórios de objetos para ambientes de programação distribuída geralmente são implementados utilizando o sistema de arquivos, podemos ter muitos acessos a disco por ação. Isto ocorre devido às atualizações necessárias no estado do objeto para garantir a atomicidade da ação. Assim, a atividade de escrita em um repositório de objetos é intensa. Além disso, a atualização se dá sobre o objeto inteiro, apesar de, na maioria das vezes, somente uma porção pequena do estado do objeto ter sido modificada, o que é potencialmente ineficiente.

2.2 *Log*

Uma das definições de *log* [1] é a representação do histórico de execução de alguma aplicação (por exemplo, banco de dados) armazenada em meio estável [2]. Em geral, a inserção de dados é feita anexando-os ao final do *log*. Em alguns casos, mesmo não guardando, necessariamente, informações de histórico, meios de armazenamento em que a inserção de dados é feita desta forma são chamados de *log* [23].

Considerando que banco de dados, assim como outros sistemas, utilizam ações atômicas, há a necessidade de procedimentos que restaurem sua base de dados a um estado consistente em caso de falhas durante a transação (por exemplo, falha de sistema). Esta tarefa é definida como recuperação. Um dos mecanismos mais utilizados para a recuperação é o *log*, existindo algumas variações na forma em que é utilizado.

Um *log* pode conter registros de *undo* e (ou) *redo*. Registros de *undo* são utilizados para anular efeitos de ações abortadas. Registros de *redo* são utilizados para refazer efeitos de ações terminadas que ainda não foram propagadas para a base de dados.

Os dados podem ser armazenados no *log* de três maneiras: valor, transição e operação:

- Um *log* de valor armazena os dados modificados pelas ações. Um *undo log* armazena *before-images* dos dados, ou seja, a imagem do dado anterior ao início da ação. Analogamente, um *redo log* armazena *after-images*. É a maneira mais utilizada.
- Um *log* de transição armazena a diferença entre a imagem antiga e a nova imagem. Pode ser feito através de um "ou exclusivo" entre os objetos. A compactação de dados pode ser feita facilmente num *log* desta natureza, através da combinação dos

[1] Para efeito de clareza, preferimos utilizar o termo *log* invés da tradução registros

dados e de suas diferenças. Já o acesso a um dado pode ser complicado, pois pode ser necessário atualizar o objeto para utilizá-lo.

- Um *log* de operação armazena o nome das operações, argumentos e, algumas vezes, seus resultados. Em caso de *undo log*, é necessário que haja, para toda operação, uma outra capaz de desfazer seus efeitos. É o menos utilizado.

O tamanho do registro depende da maneira como os dados são armazenados. O tamanho do registro é importante porque a inserção de um registro ao *log* adiciona um *overhead* fixo de acordo com o tamanho do registro. Um *log* de operação contém registros pequenos (poucos *bytes*). Um *log* de valor armazena grandes registros (de 30 *bytes* até vários milhares de *bytes*). O tamanho de um registro em um *log* de transição varia de acordo com o tamanho da modificação efetuada.

Além da informação contida no *log*, os metadados também contribuem para aumentar o tamanho do registro. Há metadados para identificação de registros lógicos e para gerenciamento físico do *log* como, por exemplo, informações que auxiliem sua compactação.

Por razões de eficiência, uma ação geralmente não modifica os dados diretamente em disco. Os dados são copiados para memória volátil, modificados e, ao final da ação essas modificações são propagadas para o meio estável. A finalização de uma ação pode ser decomposta em 3 partes:

- escrever as modificações feitas pela ação no *log*,
- escrever uma marca de final de ação no *log* e
- propagar as atualizações para a base de dados.

O momento em que a marca de final de ação é fisicamente escrita no *log* é considerado o encerramento de uma ação, pelo menos para fins de recuperação. Entretanto, consideraremos aqui a finalização como a conclusão dos 3 passos descritos anteriormente.

A informação necessária para desfazer ou refazer uma ação deve ser escrita no *log* antes da finalização da ação. Ou seja, o primeiro passo deve ser dado antes dos outros dois. Este procedimento é chamado *Write Ahead Logging* (WAL) e garante atomicidade e persistência da ação, pois a informação que garante a consistência da ação (refazendo ou anulando efeitos) estará no *log* caso necessária.

Podemos classificar 4 técnicas de recuperação, dependendo do momento em que os dados são propagados para a base de dados:

- *undo/no-redo*: Atualizações são propagadas antes do final da ação, não havendo necessidade portanto, de refazer efeitos de ações terminadas em caso de falha. É necessário manter *before-images* (valor anterior a ação) no *log* para possibilitar anular os efeitos de ações.
- *no-undo/redo*: Atualizações são propagadas após o término das ações. Assim, ações nunca precisam ser desfeitas, mas *after-images* devem ser mantidas no *log* até o final da ação (modificações propagadas para base de dados).
- *undo/redo*: Atualizações são propagadas em momentos arbitrários, a fim de otimizar a entrada-saída. É a técnica mais utilizada.
- no-undo/no-redo: Atualizações são propagadas ao final da ação. Implica em propagação atômica. Um exemplo é a técnica de cópias *shadow*. Quando o dado é reescrito, pode haver falhas, por isso são mantidas 2 cópias, uma válida e uma *shadow*. É pouco utilizada pois é menos eficiente que as outras técnicas que utilizam *log*.

Logs geralmente crescem rapidamente. Um meio de lidar com esse problema, é a separação do *log* em duas partes: uma parte temporária em memória volátil e outra permanente em disco. O *log* temporário contém informação *undo* que é retirada no final da ação. O *log* permanente em disco é compactado regularmente mantendo alguns *checkpoints*. Os registros no *log* só podem ser retirados quando não forem mais necessários. Uma entrada só pode ser removida do *log* se e somente se a ação a qual pertence tenha abortado ou o dado referenciado contenha o valor de uma ação mais recente. Caso a técnica de recuperação seja *no-undo*, registros que contenham informações já propagadas para a base de dados também podem ser removidos.

Quando lidamos com ações distribuídas, os mecanismos de recuperação tornam-se mais complexos. O protocolo *two phase commit* [2] é um dos algoritmos mais utilizados para lidar com este problema. O *two phase commit*, basicamente, divide o término da ação em duas fases: preparação e finalização. Quando combinamos este algoritmo com os mecanismos de *log* temos mais uma classe de registros que não podem ser removidos do *log*: pelo menos um *site* participante da ação (coordenador) não deve remover os registros do *log* referentes a ação até receber mensagens de finalização ou aborto da ação de todos os outros participantes. Isto é necessário porque um *site* que já tenha preparado sua ação

(primeira fase do *two phase commit*), pode cair e quando voltar estará em uma situação indefinida. Para resolver se deve finalizar ou abortar a ação, este *site* deverá procurar o coordenador da ação, que reterá esta informação enquanto algum dos participantes não mandar informações de aborto ou finalização da ação.

2.3 Gerência de Memória

Em bancos de dados, o acesso a dados é um fator crítico no desempenho do sistema. O acesso à memória principal é muito mais veloz que o acesso à memória secundária (disco). Por isto, é interessante que uma cópia da base de dados esteja na memória principal para evitar acessos a disco. A leitura de dados pode ser obtida de uma cópia em memória. A escrita precisa ser propagada para a cópia em disco e em memória. Neste caso, a cópia armazenada em meio estável (disco) é útil apenas para fins de recuperação [2].

Devido ao alto custo da memória principal até algum tempo atrás, manter toda a base de dados em memória era praticamente impossível. Por isto, optava-se por deixar apenas uma pequena parte da base de dados, que fosse bastante utilizada, em memória. A técnica de manutenção de parte da base de dados em memória é chamada *caching* ou *buffering*. Consiste em manter uma parte da memória volátil chamada *cache* para o armazenamento de dados. O *cache* consiste de uma coleção de *slots*, cada qual podendo armazenar o valor de um item de dados. A granularidade dos itens de dados armazenados no *cache* é aquela que pode ser escrita atomicamente em disco (página). Um *slot* contém o valor do dado e um *dirty bit*, que indica se o dado no *cache* foi modificado em relação a sua cópia em disco. Há também um diretório *cache*, que identifica o nome do item e o *slot* que o contém.

O tráfego de dados entre o disco e o *cache* é feito através das operações *Salva* e *Busca*. *Salva* copia o dado para disco se o *dirty bit* estiver marcado. *Busca* copia um dado do disco para um *slot* vazio, caso haja algum. Caso contrário, seleciona algum *slot*, *salva*-o e utiliza este *slot*. A seleção do *slot* a ser substituído é feita segundo a estratégia de realocação utilizada. No final desta seção, veremos com mais detalhes as técnicas de realocação.

Para ler um item de dados, lê-se o valor do *cache* se o item estiver ali, caso contrário, *busca*-se o objeto. Para escrever, escreve-se o objeto no *cache* (caso o item não se encontre no *cache*, é feita a criação de um *slot* para ele) e marca-se o *dirty bit* do *slot*. A propagação para o meio estável depende da estratégia de propagação utilizada [24].

Em determinados momentos, dados podem ter cópias diferentes em memória e em disco. Dizemos que a atualização da cópia em disco **sincroniza** as duas cópias. Quando fazemos uma alteração em um dado de forma que sua cópia em disco e em memória fiquem sincronizadas, dizemos que a escritas (ou entradas) são **síncronas**.

Em [11], argumenta-se, com base no custo dólar do tempo de acesso a dados em disco e o custo da memória, que todo dado referenciado pelo menos a cada 5 minutos deveria estar em memória principal. Como o preço da memória vem caindo relativamente ao custo de acesso a disco por segundo, o tempo resultante tende a crescer. Portanto, a tendência é a manutenção de cada vez mais dados em memória [20].

Com a diminuição do custo de memória principal e a demanda por sistemas mais velozes, nos últimos anos a idéia de manter a base de dados residente em memória ganhou força e muito se pesquisou com esse intuito. Bancos de Dados Residentes em Memória (BDRM) armazenam toda, ou a maior parte da, base de dados em memória principal. A necessidade de entrada/saída com disco é (praticamente) eliminada. Em um banco de dados convencional (banco de dados residentes em disco), dados podem ser copiados para memória (*caching*) para otimizar acesso; em um BDRM os dados residem em memória e tem uma cópia reserva em disco. A principal diferença é que a cópia primária de um dado deve estar permanentemente em memória [20].

A maioria dos problemas associados a bancos de dados tradicionais também existem em um BDRM. Entretanto, as soluções adotadas para lidar com questões de bancos de dados como processamento de ações atômicas, estrutura de dados, otimização, recuperação e controle de concorrência tem um fraco desempenho ou não funcionam quando os dados são residentes em memória [9]. Na Seção 3.2, aprofundar-nos-emos neste tópico.

Considerando que a maioria dos sistemas existentes não permite ainda que toda a base de dados esteja em memória, boa parte dos BDRMs trabalham com a hipótese de que a maior parte da base de dados esteja residente em memória. Quando houver a falta de um dado, ou seja, necessita-se referenciar um dado que não está em memória, é necessário então buscá-lo em disco, como ocorre quando há *cache* de dados. Da mesma maneira, pode não haver espaço em memória para a alocação do novo item, necessitando haver realocação de dados.

O ideal é remover a página que não será referenciada pelo período de tempo mais longo no futuro. O que normalmente é feito é tentar inferir o futuro a partir do comportamento passado. A estratégia de realocação, também chamada de paginação quando lida com

páginas em memória, é a responsável por definir qual dado será escolhido para deixar a memória. Estratégias de realocação são utilizados tanto em *caching* como em BDRMs. Em [31], Ziviani descreve algumas estratégias bastante conhecidas:

- Menos Recentemente Utilizada: um dos algoritmos mais utilizados é o LRU (*least recently used*), que remove a página menos recentemente utilizada, partindo do princípio que o comportamento futuro deve seguir o passado recente. Neste caso, temos que registrar a seqüência de acesso a todos os dados. Pode ser implementado através de *time stamps* ou pelo uso de uma fila em que os dados referenciados passam para o final da fila dos elementos a serem retirados.

- Menos Freqüentemente Utilizada: O algoritmo LFU (*least frequently used*) remove a página menos freqüentemente utilizada. A justificativa é semelhante ao caso anterior, e o custo é registrar o número de acessos a todos os dados. Um inconveniente é que uma página recentemente trazida da memória secundária tem um baixo número de acessos registrado e, por isso, pode ser removida. Pode ser implementado com a utilização de um contador de referências em cada dado.

- Ordem de Chegada: O algoritmo FIFO (*first in, first out*) remove a página que está residente há mais tempo. Este algoritmo é o mais simples e o mais barato de se manter. A desvantagem é que ele ignora o fato de que a página mais antiga pode ser a mais referenciada. Pode ser implementado através de uma fila.

  Em [12], Gruenwald et al descreve algumas outras estratégias mais especializadas como:

- Conjunto de Trabalho: Um conjunto de trabalho (*working set*), é um conjunto de dados para os quais um processo acessa constantemente. A idéia é manter estes dados em memória e os dados que não façam parte do conjunto de trabalho ficam disponíveis para a realocação.

Outro fator a considerar quanto a gerência de memória, é a **carga** dos dados para a memória. Tanto sistemas com dados residentes em disco como em memória vão, de alguma forma, preencher a memória destinada a armazenamento de dados. Esta carga de dados ocorre na iniciação ou após falhas de sistema. Há duas principais estratégias para fazer-se a carga dos dados: sob demanda e prévia. A carga sob demanda simplesmente deixa o funcionamento do sistema buscar os dados conforme sua necessidade e assim preencher a memória incrementalmente. A carga prévia preenche a memória buscando previamente em disco os objetos para isto. Existem propostas mistas também. Em [12],

Gruenwald et al faz uma comparação de combinações de técnicas de carga e paginação em bancos de dados residentes em memória.

A combinação de objetos e ações atômicas fornece um poderoso mecanismo para a implementação de programas distribuídos. Uma aplicação que implemente pelo menos uma destas funcionalidades provavelmente utilizará mecanismos de *log* e de gerência de memória para alcançar um bom desempenho. A seguir, veremos como isto ocorre em algumas destas aplicações.

Capítulo 3

Trabalhos Relacionados

Existem muitos trabalhos que contribuíram para o desenvolvimento desta dissertação. Visto que repositórios estão presentes em diversos tipos de aplicações, foi necessário pesquisar várias áreas da computação. Por isto, dividimos este Capítulo em três Seções distintas, cada uma apresentando a maneira como repositórios são implementados nestas aplicações. Os trabalhos foram divididos em 3 tipos de aplicações: bancos de dados convencionais, bancos de dados residentes em memória e sistemas de arquivos. A seguir, fazemos uma análise comparativa das soluções existentes nestas aplicações, verificando as vantagens e desvantagens de cada abordagem e quais características seriam mais interessantes para o novo repositório de Arjuna.

3.1 Bancos de Dados

3.1.1 ENCORE-ObServer

ENCORE (*Extensible and Natural Common Object REsource*) [14] forma a camada superior (*front-end*) da gerência de objetos e ObServer (*Object Server*) a camada inferior (*back-end*) da gerência de objetos. São desenvolvidos para ambientes de computação cooperativos e implementados em C.

ObServer suporta controle de concorrência baseado em trancas (*locks*), recuperação baseada no *log*, e ações atômicas. Além disso, ObServer adapta-se ao modelo de ações atômicas através de trancas não restritivas (*soft locks*). ObServer também suporta agrupamento físico de objetos em função da classe ou de referências entre objetos.

3.1.2 O2

A arquitetura do O2 [14] é cliente-servidor e foi implementado em C. Em O2, não há remoção explícita de objetos persistentes. A gerência de objetos persistentes é feita utilizando *WISS* (*WIsconsin Storage System*) [7], um repositório de objetos com controle próprio de acesso a disco.

WISS pode ser utilizado como repositório de objetos (registros) para bancos de dados ou como componente de armazenamento de um sistema de arquivos. WISS não utiliza o sistema de arquivos *Unix* para evitar seus problemas de desempenho. WISS garante alocação física seqüencial de registros, gerencia seu próprio *buffer* sem o *overhead* da utilização do buffer *Unix* e suporta arquivos seqüenciais e índices implementados como árvores B+.

A arquitetura do sistema consiste de quatro níveis: camada física de entrada/saída, gerenciador de *buffer*, estruturas de armazenamento e métodos de acesso. A camada física (nível 0) gerencia os dispositivos físicos do disco e escalona as atividades de entrada/saída. Também aloca os registros em arquivos e gerencia espaço livre em cada volume (disco, tape, etc). O nível 1 executa a gerência de *buffer* fazendo chamadas ao nível 0 de leitura e escrita em disco. Mantém um conjunto de *buffers* de páginas e implementa sua estratégia de armazenamento de páginas na memória baseada em pistas (*hints*) de páginas mais usadas. A partir do nível 2, os acessos são a registros, diferente dos anteriores que eram a páginas. Além disso, os arquivos agora são identificados pelo nome enquanto que, anteriormente, eram identificados pelo identificador de arquivo. As estruturas utilizadas são arquivos seqüenciais, índices primários (árvore B+) e secundários e itens longos de dados (*long data itens*). Os métodos de acesso cuidam de procura seqüencial, procura por índice e busca por itens longos de dados. Sua interface permite a criação e destruição de arquivos, índices e itens longos de dados.

Apesar do WISS aparentar ser uma boa alternativa a utilização de arquivos *Unix*, ele não é, em princípio, totalmente auto suficiente como repositório de objetos, visto que mecanismos de recuperação e controle de concorrência praticamente inexistem.

3.1.3 POSTGRES

Por uma série de motivos, a orientação a objetos no POSTGRES [14, 29, 30] dá-se através da extensão de novos tipos de dados ao modelo relacional já existente.

Como método de acesso, POSTGRES suporta índices somente através de operadores (elementos que utilizam um ou mais operandos na linguagem de consulta). Como estrutura de índice *default*, POSTGRES utiliza árvores B+, mas permite ao usuário acoplar outros (hash, árvore R, grid-file, árvore K-D-B, etc) em caso de dados não convencionais

POSTGRES implementa um gerenciador de armazenamento sem reescrita (*no-overwrite*). Esta opção foi escolhida por razões de desempenho, visto que, diferente da técnica usada anteriormente (*write ahead log* (WAL)), não é necessária a utilização do *log* tradicional. O gerenciador sem reescrita só armazena informações sobre o estado de cada ação (terminado, abortado, corrente). Para um bom funcionamento dessa técnica é interessante a presença de memória principal estável, caso contrário será necessário escrever no disco todas as páginas da ação em sua decisão.

Há dois grupos de registros: os correntes e os históricos. Os registros históricos ficam armazenados em disco e seu índice tem como chave o tempo combinado com outro atributo. São utilizadas árvores R para suportar essa indexação. Quanto ao gerenciamento de *buffer*, POSTGRES usa um algoritmo LRU (*Least Recently Used*) modificado para manutenção de páginas na memória.

3.1.4 EXTRA

Extra [14] é implementado na linguagem de programação E, uma extensão de C++ com persistência de objetos. A gerência de objetos no EXTRA foi construída sobre o sistema de armazenamento EXODUS [5].

Exodus contém um gerenciador de armazenamento de objetos que provê leitura, escrita e atualização de objetos inteiros ou partes, sem importar o tamanho. Além disso, também é fornecido nesse nível gerência de *buffer*, controle de concorrência e mecanismos de recuperação para os objetos compartilhados.

O gerenciador de armazenamento de objetos fornece uma interface para criação e destruição de *arquivos de objetos* (coleção de objetos armazenados) ou seja, remoção de objetos persistentes. Também fornece abertura e fecho para procura nesses arquivos. Para o acesso a arquivos de objeto é utilizada uma estrutura de índice semelhante a árvore B+. Além disso, provê uma certa parametrização através de pistas visando desempenho como, por exemplo, definição de agrupamento dos objetos em relação a que objetos.

Objetos armazenados podem ser pequenos (poucos *bytes*) ou grandes (mais de uma página). O gerenciador de armazenamento de objetos provê versões de maneira bastante cara do ponto de vista de espaço. Simplesmente mantém várias cópias do objeto marcadas como versões antigas. O mecanismo de recuperação age diferentemente de acordo com o tamanho do objeto. Recuperação em objetos pequenos é feita utilizando *image logging*, enquanto que, em objetos grandes, é através de uma combinação de *shadow* e *logging*.

O controle de concorrência utiliza o protocolo (*two phase locking*), com opção para trancamento do objeto inteiro. Em caso de objetos grandes, é utilizado o protocolo *no-two-phase B+ tree locking* [5].

A gerência de *buffer* tem como granularidade blocos de *buffer* de comprimento variável ao invés de páginas. Gerência de espaço livre no *buffer* é feita através de técnicas de alocação dinâmica, enquanto que alocação do *buffer* em disco é dirigida pelos mecanismos de pistas do gerente de armazenamento de objetos.

3.1.5 SHORE

Conceitos Básicos SHORE (Scalable Heterogeneous Object REpository) [6] é um sistema de objetos persistentes desenvolvido na Universidade de Wisconsin. Parte da equipe que desenvolveu SHORE participou do projeto EXODUS. Por esse motivo, há certas semelhanças entre os projetos mas, o mais importante é o fato de que houve mudanças de projeto devido a experiência anterior. Entre as limitações do projeto EXODUS ditas como solucionadas ou otimizadas pelo projeto SHORE, estão a facilidade de acesso a objetos de tipos errados, dificuldade de compartilhamento entre dados de diferentes aplicações, falta de armazenamento de tipos de objeto, restrição à arquitetura cliente-servidor.

SHORE implementa comunicação entre servidores de mesmo nível. Em SHORE, todo processador que tenha acesso local a disco pode atuar como um servidor.

Objetos em SHORE podem ser diretórios, *links*, objetos individuais (*typed objects*), semelhante ao *Unix*. Entretanto, diferente do *Unix*, objetos podem ser acessados globalmente por identificadores únicos (IU).

SHORE pode ser considerado uma combinação de sistema de arquivos e bancos de dados, pois mantém características de ambos. Como características de sistema de arquivos, SHORE provê espaço de nomes de objetos flexível e mecanismos que permitem acesso a chamadas do sistema *Unix*. Em relação a bancos de dados orientados a objeto, podemos citar a linguagem para descrição de tipos de dados persistentes: SHORE Data Language (SDL), que pretende ser uma linguagem neutra com ligações a outras linguagens (atualmente só C++) para que elas possam trabalhar com o sistema.

Todos objetos são instâncias de **tipos de interface**, ou seja, tipos construídos com a interface do construtor de tipos. Podem ter métodos, atributos e relacionamentos. Além disso, SHORE suporta controle de concorrência por meio de trancas e recuperação via *logging*. Também há agrupamento de objetos. Mecanismos estes que são características importantes de bancos de dados. Logo, discutiremos alguns desses aspectos com mais detalhes.

Arquitetura Shore A comunicação funciona como um grupo de processos de comunicação chamados **servidores SHORE. Processos de aplicação** manipulam objetos, enquanto servidores lidam com páginas de volume de disco, cada qual gerenciada por um único servidor (não há replicação). Cada servidor atua como gerenciador do *cache*, recebe chamadas RPC, e é responsável pelo controle de concorrência e recuperação. O gerenciador do *cache* faz realocação de páginas utilizando o algoritmo LRU. Todos os objetos no *cache* estão em uma tabela de objetos.

O servidor SHORE é dividido em **interface de servidor** e **gerenciador de armazenamento.** A interface cuida da comunicação com aplicações, gerenciando o espaço de nomeação dos objetos (como no *Unix*). O gerenciador de armazenamento tem três camadas. A primeira lida com controle de ações atômicas e acesso a objetos e índices. A segunda implementa índices, ações atômicas, registros, controle de concorrência e recuperação. A terceira trata consistência do *cache*, comunicação, *log*, trancas e ações atômicas distribuídas.

O gerenciador de armazenamento trata dois tipos de identificadores: físico e lógico. O físico indica a posição em disco do objeto e o lógico é independente, deixando transparente

a reorganização física (*recluster*) dos objetos em disco. Para mapear identificadores físicos e lógicos remotos, cada volume contém um índice de árvore B+, juntamente com uma tabela *hash* na memória com as entradas mais recentes. A técnica utilizada reduz o número de acessos a índice a um por objeto, ao invés de um por página do objeto.

3.1.6 ObServer2

Em [16], Langworthy faz uma proposta interessante para obter alto desempenho no mecanismo de persistência em ObServer2. A proposta explora mecanismos de *cache* e *log*. Basicamente, as atualizações dos objetos são salvas no *log* e uma versão base fica armazenada em uma partição de banco de dados. As leituras, em geral, são feitas no *cache*.

O estado persistente do objeto é formado por sua versão base e seu histórico no *log*. A recuperação é feita assincronamente e sob demanda. Isto torna-a rápida, pois não há um momento em que toda recuperação é feita de uma só vez, ao invés disto, seu tempo fica diluído no funcionamento do sistema. Seu processo é parecido com atualização do *cache*. A versão desatualizada (em *cache* ou na partição de banco de dados) é atualizada aplicando-se as operações no *log*. As operações também devem ser aplicadas em caso de:

- requisição de espaço no *cache* para outros objetos;
- necessidade de espaço no *log*.

A aplicação de operações do *log* sobre a versão base na partição de banco de dados libera espaço no *log* (não há necessidade do histórico, visto que a versão base está atualizada) e no *cache* (objeto pouco requisitado tem menos prioridade de ocupar *cache*). A aplicação do histórico atualizando o *cache* também pode ter como objetivo diminuir o número de operações necessárias para atualizar o objeto quando requisitado. O processo de atualização poderá ser muito custoso caso sejam muitas as operações a serem feitas sobre o objeto para deixá-lo em estado consistente.

Para otimizar o desempenho do sistema, pode ser feita uma busca antecipada de objetos em disco, mantendo-os em memória. Os objetos são escolhidos baseado na semântica da aplicação ou no "relacionamento" entre os objetos. Esta busca deve ser feita em períodos ociosos.

Existem três tipos de registros no *log*: Registro de Intenções, registro de *Prepare* e registro de *Commit*. Um registro no *log* é endereçado pelo *Log Sequence Number* (LSN). As operações são armazenadas no registro de intenções. Além disso, um registro de intenções também contém um identificador, o *Transaction id* (Tid), e o LSN do registro de intenções anterior, formando assim uma lista de intenções.

Para cada ação em andamento, há uma entrada na *Transaction Table*, que indica a situação da ação (*running, prepared, commited*). Existe também outra estrutura importante, *Version Table*, que contém o mais recente LSN para cada objeto no *cache* desatualizado. Quando um objeto é lido, sua versão é verificada através de seu LSN e do LSN da *Version Table*. Se forem diferentes, o objeto está desatualizado e é necessária a aplicação das operações no *log* para atualizá-lo. Portanto, o LSN também funciona como um *time stamp* do objeto.

ObServer2 utiliza *two-phase-commit*. No final da ação, não é necessária atualização da versão base. Apenas a lista de intenções é gravada no *log* e, se não houver conflitos, é finalizada a fase preparação, através da gravação do registro de *Prepare* com o LSN do último registro de intenções e o Tid. Também são atualizadas *transaction table* e *version table*. Caso haja conflito (alguma outra ação invalida alguma operação desta) a ação é abortada. Em seguida, é salvo o registro de *Commit* com Tid e LSN do registro de *Prepare* e novamente atualizada a *Version Table*. Nesse momento, os clientes são notificados que não mais possuem a versão atualizada do objeto.

3.2 Bancos de Dados Residentes em Memória

Com o custo da memória principal diminuindo nos últimos anos [11] e a demanda por sistemas mais rápidos, passou-se a trabalhar com a possibilidade de toda a base de dados, ou pelo menos grande parte, ficar residente em memória, ao invés de residente em disco, como ocorre nos sistemas convencionais de bancos de dados [17].

Em [9], Dunham afirma que "A principal idéia de um banco de dados residente em memória (BDRM) não é sua arquitetura, mas a percepção de onde os dados residem. A maioria dos problemas associados a bancos de dados convencionais também está presente em BDRMs. Entretanto, as soluções normalmente adotadas para lidar com determinadas características destes sistemas como processamento de ações atômicas, estrutura de dados, otimização, recuperação e controle de concorrência, tem fraco desempenho ou mesmo são

incorretas quando os dados estão residentes em memória". Em um BDRM, a cópia de dados em memória é tratada como *primária*, enquanto a cópia em disco é *reserva*. Em bancos de dados convencionais, a cópia em disco é tratada como primária [17].

Muita pesquisa tem sido feita sobre BDRMs. Grande parte tem resultado em projetos, propostas de arquitetura, e mesmo protótipos. Aqui discutiremos as principais características de BDRMs que o distinguem dos bancos de dados convencionais e duas arquiteturas propostas de BDRMs que representam duas abordagens diferentes. Uma utiliza recuperação baseada em *checkpoint* [19] e outra faz uso de recuperação incremental [18]. Há várias outras propostas e implementações de BDRMs, mas acreditamos que estes dois trabalhos podem ilustrar o funcionamento geral de BDRMs.

3.2.1 Características de BDRMs

Diferenças entre BDRMs e bancos de dados convencionais: Em [20], Molina e Salem fazem algumas considerações interessantes sobre as diferenças entre a permanência de dados em disco e em memória principal:

- O tempo de acesso em memória principal é ordens de magnitude menor que em disco.
- Memória principal é normalmente volátil, ao contrário de disco. Porém, é possível se construir memória principal não volátil.
- Discos têm um custo fixo e alto por acesso que não depende da quantidade de dados buscados durante o acesso. Por isto, discos são orientados a blocos e memória principal não.
- Memória principal é normalmente diretamente acessível pelo(s) processador(es) e disco não. Isto torna os dados em memória mais vulneráveis a erros de *software*.

Molina e Salem também consideram a diferença entre um BDRM e um banco de dados convencional com *cache* muito grande. Um banco de dados convencional com estas características, apesar de poder até conter todos os dados em memória, continua otimizado para acesso a disco. Por exemplo, estruturas de acesso, como *B-Trees*, continuam sendo utilizadas, aplicações continuam a acessar dados através de um gerente de *buffer*, como

se os dados estivessem em disco. Em [17], Lehman et al demonstra que um BDRM pode ter um ganho de desempenho significativo sobre um banco de dados convencional mesmo com todos os dados no *cache*.

Aspectos particulares de BDRMs: Existem alguns processos afetados pela manutenção de cópias primárias de objetos em memória [20]. Destacamos a seguir alguns aspectos que podem ser de maior interesse:

- Controle de Concorrência: A utilização de trancas *locks* de granularidade pequena (campos, registros...) visa diminuir a contenção de dados. Visto que acesso a dados em memória tende a ser mais rápido que em disco, o ganho na utilização de trancas com granularidade pequena diminui. É recomendada a utilização de trancas com maior granularidade, como relações.

  Em sistemas convencionais, trancas são implementadas através de tabelas *hash*. Considerando que em BDRMs os objetos estão em memória, pode ser mais eficiente armazená-los junto aos objetos.

- *Commit Processing*: Para lidar com falhas de *hardware*, é necessário haver uma cópia reserva e um *log* sobre as atividades das ações. O *log* necessita estar em um meio estável, como disco, e os registros retratando a ação devem estar no *log* antes dela finalizar.

  Processamento do *log* é a única entrada em disco efetuada durante a ação. Por isto, torna-se muito importante a maneira como é feito, pois afeta decisivamente o tempo de resposta do sistema. Além disso, o *log* pode tornar-se um gargalo no sistema, visto que é utilizado por todas as ações.

  Para lidar com esses problemas, algumas soluções já foram sugeridas. A utilização de memória principal estável para uma pequena porção do *log*, chamada de *log tail*, em que são feitas as atualizações melhora o tempo de resposta. Para evitar o gargalo, pode ser feito pré finalização (*pre-committing*) e finalização de grupo de ações. Lehman et al [17] cita a utilização de objetos replicados e disco *append-only* ao invés do mecanismo de *log tail*.

- Métodos de Acesso: Métodos de acesso projetados para armazenamento orientado a blocos, como B-Tree, não são adequados a BDRMs. Para lidar com dados residentes em memória, tem sido proposto o uso de estruturas como *hash* e árvores. *T-Tree*

é uma estrutura desenvolvida especialmente para BDRMs [17]. Essencialmente, *T-Tree* é uma árvore binária com nó multi-item e que usa balanceamento semelhante ao da árvore AVL [15].

- Recuperação: Há duas maneiras principais de fazer recuperação em BDRMs: incremental e *checkpoint*.

  Visto que o *log* pode crescer muito devido às operações serem feitas sobre as cópias primárias dos objetos e não serem repassadas às cópias reservas, a recuperação pode ser muito longa. Além disso, como a memória tende a conter (quase) toda base de dados, a troca de páginas entre memória principal e disco tende a ser rara ou até inexistente. Portanto, paginação não pode ser o mecanismo primário para propagação de atualizações no *log* para as cópias reservas.

  A utilização de *checkpoints* (*dump*) visa diminuir o tempo de recuperação. *Checkpoint* torna a base de dados em disco mais atualizada, através da sincronização de toda (ou quase) base de dados. A escrita do *checkpoint* é grande. Por isto, recomenda-se que seja feita seqüencialmente e utilizando E/S com blocos bem grandes. A sincronização da base de dados elimina a necessidade da manutenção dos registros do *log* existentes, diminuindo portanto, o tempo de recuperação.

  Em uma aproximação direta de *checkpoint*, pode haver necessidade de controle sobre atividades relacionadas ao processamento de ações atômicas, como, por exemplo, concorrência. Isto pode deteriorar o desempenho do sistema. *Fuzzy Checkpoint* é uma variação deste mecanismo que não necessita de controle sobre o processamento de ações atômicas e o *checkpoint*. Em [19], Lin e Dunham apresentam um estudo de algumas variedades de *fuzzy checkpoint*.

  Outro problema da utilização de *checkpoint* ocorre após a falha. Restaurar o *checkpoint* implica na leitura de toda a base de dados, o que pode levar muito tempo. Esta situação ocorre também durante a iniciação do sistema. Em [12], Gruenwald et al faz um estudo comparativo entre técnicas de carga da base de dados e paginação em BDRMs. Uma outra opção para a carga da base de dados para memória é fazê-la sob demanda, até que todos os dados estejam em memória.

  Recuperação incremental [18] atualiza cópias reservas aos poucos, em paralelo com o processamento de ações atômicas, mantendo assim, a base de dados em disco razoavelmente atualizada, diminuindo portanto, o tempo de recuperação. Segundo Levy e Silberschatz [18], a utilização de *checkpoint* apresenta problemas, dos quais destacamos alguns:

- *checkpoint* interfere com o processamento de ações atômicas. O *checkpoint* simples pode até paralisá-lo. Mesmo com *fuzzy checkpoint*, ocorre alguma contenção de memória, pois ambos os processos (*checkpoint* e o processamento de ações atômicas) acessam a mesma memória.
- *checkpoints* em disco precisam ter pelo menos duas cópias da base de dados em disco para tratar falhas durante o *checkpoint*.

3.2.2 Propostas de Bancos de Dados Residentes em Memória

Recuperação através de *checkpoint*: Em [13], Hangmann descreve um projeto de BDRM que apresentaremos para ilustrar a utilização de *checkpoint* nestes sistemas. Em bancos de dados convecionais, recuperação se dá através do *log* em conjunto com a base de dados em disco. Mas nestes sistemas, a base de dados está praticamente atualizada, com exceção das ações que estivessem ativas antes da queda, e o *log* é tipicamente pequeno, contendo informações justamente sobre estas ações. Já em BDRMs, o *log* tende a crescer indefinidamente, o que torna a recuperação extremamente demorada.

Checkpoint fornece um ponto de partida para recuperação a partir do *log*. Infelizmente, *checkpoint* precisa de sincronia com o processamento de ações, podendo até paralisá-lo. Uma alternativa é a utilização de *fuzzy checkpoint* que atualiza periodicamente a base de dados sem sincronia. O *checkpoint* é inconsistente, visto que pode conter informações de ações incompletas. Porém, o estado consistente da banco de dados pode ser restaurado através do *log*, já que o *log* contem entradas destas ações.

Fuzzy Checkpoint é feito através de uma longa escrita seqüencial em disco, o que a torna mais eficiente. Muito pouca comunicação é necessária entre a gerência de ações e o *checkpoint*, tornando desnecessário sincronia entre os processos. Somente registros de início e fim de *checkpoint* precisam ser inseridos no *log*.

Como o *log* vai crescendo, é necessário efetuar periodicamente sua compactação. Utilizando memória principal estável, pode-se protelar a escrita ao *log* físico, mas, em algum momento, ela ocorrerá. Neste momento, somente os registros de *redo* são escritos em disco. Estes registros contêm só a parte modificada das páginas de dados.

Após o *fuzzy checkpoint*, a maioria do *log*, com exceção dos registros referentes a ações ativas, torna-se inútil. Assim, de tempos em tempos é feita a compactação do *log*, elimi-

nando os registros sobre ações terminadas antes do *checkpoint*. O *checkpoint* é feito em um *buffer* circular em disco, com espaço para pelo menos 2 cópias para lidar com falhas durante sua execução.

Não deve haver praticamente espera para fazer acesso ao disco, visto que o acesso é limitado às atividades do *log* e *checkpoint*. Para diminuir ainda mais o tempo de acesso, as duas atividades podem ser feitas em discos separados.

Para re-iniciar o sistema após queda, os seguintes passos são necessários:

- Processar o *log* e encontrar os parâmetros de re-início;
- trazer o *fuzzy checkpoint* para memória;
- Processar o *log* para descobrir o destino de ações ativas;
- Processar o *log* e atualizar a base de dados;
- iniciar o sistema.

O modelo considerado aqui pressupõe que a base de dados esteja completamente em memória. Para estender esta proposta relaxando esta premissa, é necessário armazenar, junto às páginas em memória, o *status* limpo/sujo, conforme estejam ou não consistentes com o *checkpoint*. Em caso de falta de página (página procurada está somente em disco), apenas páginas limpas podem ser utilizadas na troca entre páginas em disco e páginas em memória.

Recuperação Incremental: Este modelo determina que atividades relacionadas a recuperação devem ser executadas incrementalmente, sem interferir no processamento de ações atômicas [18]. A recuperação de dados também é feita incrementalmente e sob demanda. O modelo apresentado utiliza páginas de dados como organização da base de dados, um gerente de *buffer* que controla a leitura e escrita efetuadas por ações. Também utiliza memória principal estável para uma porção do *log* (*log tail*). As entradas do *log* contêm as cópias anterior e posterior das porções afetadas das páginas modificadas pela ação, juntamente com a posição da porção modificada na página. O controle de concorrência é feito através de trancas com granularidade de página.

Cada página pode ter três estados:

- corrente: cópia do objeto em memória;
- reserva: cópia do objeto em disco;
- final: cópia que reflete a última ação terminada que modificou o dado.

Os termos *primary database* (PDB) e *backup database* (BDB) denotam respectivamente as páginas de dados em memória principal e as páginas de dados em disco.

Temos ainda que, uma página x pode ser:

- *dirty*, se e somente se reserva[x] != corrente[x], senão x é *clean*;
- *stale*, se e somente se reserva[x] != final[x], senão x é *fresh*;
- *up-to-date*, se e somente se corrente[x] = final[x]

A arquitetura utilizada considera o uso somente de registro *redo*, ou seja, apenas atualizações de ações finalizadas com sucesso vão para BDB, e decompõe o processamento de ações da recuperação. A Figura 3.1 ilustra o funcionamento do modelo.

Duas técnicas integradas implementam o modelo:

- *Log Driven Backup*(LDB): utilização de registros do *log* para atualizar BDB;
- Marca *Fresh/Stale*: manutenção em memória estável de *status fresh/stale* das páginas em memória.

Log Driven Backup: LDB evita que o número de operações que diferenciam reserva[x] e final[x] seja muito grande. Registros do *log* são produzidos por ações ativas e acrescidas no *log tail*. Até a ação terminar, os registros ficam "na espera". Se a ação abortar, os registros são utilizados para desfazer os efeitos no PDB. O acumulador passa adiante só os registros de ações finalizadas com sucesso, ou seja, registros de *redo*. A seguir, o *logger* passa registros do *log* para o disco do *log* liberando memória principal estável.

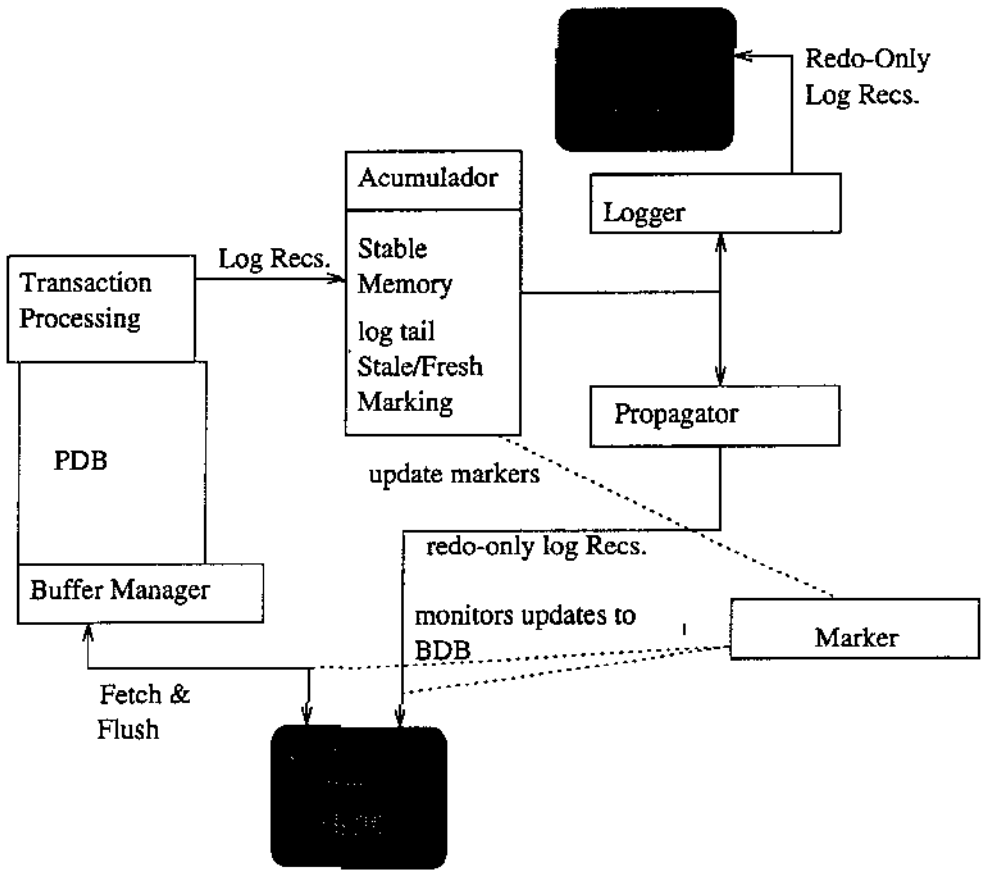


Figura 3.1: Descrição esquemática da arquitetura proposta

Paralelamente, o propagador atualiza o BDB através dos registros do *log*, por isto esta técnica é chamada de *Log Driven Backup*. Perceba que os registros do *log* utilizados pelo propagador já podem estar no disco do *log*. Por isto, os registros são agrupados de acordo com as páginas que modificam, diminuindo o número de acessos a disco. O propagador e o *logger* podem ser gerenciados por processadores diferentes já que os dois processos são independentes.

Além do propagador, o gerente de *buffer* troca páginas com o BDB conforme a demanda. Devido a utilizar lógica somente *redo*, estas páginas devem ser sempre finais. A fim de manter correção, o propagador atualiza somente as páginas que estão no PDB (*Safe Fetch Rule*). A *Single Propagation Rule* diz que "quando todos os registros correspondentes a uma página foram aplicados ao BDB pelo propagador, o gerente de *buffer* não precisa regravar esta página no BDB". Esta regra é proposta para ganho de eficiência, e pode ser implementada através do *Log Sequence Number*. Não entraremos em detalhes aqui dos detalhes desta possível implementação.

Marca *Stale/Fresh*: O objetivo desta técnica é permitir recuperação rápida após a queda do sistema. O processamento de ações volta a funcionar imediatamente após o sistema voltar a ativa.

O procedimento para acesso a uma página x por uma ação T é o seguinte:

```
Se x e' stale
   pegue a copia reserva de x;
   Recupere todos os registros de log relevantes a x;
   Aplique estes registros a copia de x, tornado-a up-to-date;
T acessa x.
```

Para implementar este procedimento, a técnica de Marca *Stale/Fresh* é utilizada, indicando quais páginas são *stale* e quais são *fresh*. A indicação desta marca precisa ser estável, pois é utilizada no durante a recuperação, após a falha. Como é atualizada constantemente, sua manutenção deve ser realizada em memória principal estável, evitando a perda de desempenho que acarretaria tais acessos em disco.

3.3 *Log Structured File System (LFS)*

Trabalhos recentes indicam que, com a proliferação de memória *cache* nos sistemas, o gargalo dos sistemas de arquivos tem passado para a escrita, ao invés da leitura [27]. LFS [23] é um sistema de arquivos que executa todas suas escritas contiguamente. Essa política o torna claramente otimizado para escrita, diferente dos sistemas de arquivos tradicionalmente otimizados para leitura.

LFS trata o disco como um conjunto de seguimentos de *log append only*. Ou seja, não há reescrita de blocos. Com isso, o processo de escrita é otimizado, pois é diminuído o tempo de procura de blocos livres para a escrita. Geralmente, é possível acumular-se grandes quantidades de dados no *cache* efetuando-se escritas de forma rápida e assíncrona. Este processo, entretanto, cria fragmentação muito grande no disco, pois os blocos re-escritos ficam *mortos* no disco e esse espaço será, em breve, necessário. Para solucionar esse problema, é necessária a ação do *cleaner*, um coletor de lixo do sistema. Nesse período, todo o processamento do sistema é interrompido até o *cleaner* encerrar sua atividade. Esse *overhead* causado pelo *cleaner* causa uma degradação bastante grande no sistema [26], mas algumas heurísticas vem sendo estudadas e implementadas para diminuir o impacto da atuação do *cleaner* no sistema [3].

No caso de ações atômicas, podemos visualizar uma série de vantagens na utilização do LFS. Como descrito anteriormente, durante o processamento de uma ação atômica é intensa a atividade em disco, pois as mudanças nos estados dos objetos geram gravações em disco. Com a substituição do sistema de arquivos tradicional por um sistema de arquivos otimizado para escrita, poderíamos melhorar o desempenho do repositório. Testes realizados anteriormente demonstram que o desempenho do sistema de arquivos LFS para suporte a ações é superior aos sistemas de arquivos tradicionais. Entretanto, o *overhead* do *cleaner* ainda degrada significativamente seu desempenho [26].

Atualmente, existem implementações públicas do LFS para o sistema operacional *Sprite* [23] e para *Unix BSD* [26]. O sistema de arquivos LFS-BSD foi implementado posteriormente, e procura corrigir alguns problemas identificados na versão original escrita para *Sprite*. Seu código está disponível e é de fácil acesso [21, 25].

3.4 Análise dos Trabalhos Relacionados

As alternativas descritas neste Capítulo, se utilizadas para otimizar o mecanismo de persistência de Arjuna, teriam uma série de vantagens e desvantagens.

A utilização de um repositório de objetos nos moldes de um sistema de banco de dados convencional [7] otimizaria escrita e leitura em disco dos objetos, mas acarretaria uma série de modificações no ambiente de programação distribuída. Definir exatamente o que seria afetado por estas mudanças e como elas deveriam ser efetuadas tornar-se-ia parte importante desse trabalho. Questões como atomicidade das ações, controle de concorrência, mais especificamente granularidade de trancas, provavelmente seriam afetadas, visto que os objetos passariam a ser acessados por atributos, não necessitando o trancamento deles como um todo. Tráfego do objeto na rede também deveria ser alterado pelo mesmo motivo. A simples indicação dessas modificações é um trabalho bastante complexo, sendo sua implementação de um grau de dificuldade bastante alto. A partir desta constatação, percebemos que este caminho poderia não ser uma escolha apropriada, considerando-se o trabalho necessário a sua implementação e o tempo previsto para o mestrado.

Uma outra alternativa seria a substituição do sistema de arquivos existente por um mais adequado ao suporte de ações atômicas, como o LFS [23, 26]. Esta solução praticamente eliminaria a fase de adaptação do ambiente a seu repositório, visto que a interface atual do Arjuna com seu repositório permaneceria a mesma e otimizaria o processo de escrita em disco dos objetos. Entretanto, podemos ter que adaptar o LFS para os sistemas operacionais em que usaremos Arjuna (Linux ou SunOS). Esse tipo de adaptação se dá próximo ao núcleo do sistema, sendo de grande complexidade. Além disso, existem questões quanto a confiabilidade do LFS, que não é amplamente utilizado e testado, problemas quanto ao momento adequado de se acionar o *cleaner*, desempenho geral dependente da quantidade de espaço livre em disco (quanto mais espaço, menos acionado é o *cleaner*), etc.

Outra opção é utilizar o LFS como repositório de objetos mas em nível de usuário. Dessa maneira, a interface de Arjuna com o repositório teria de ser modificada, pois não seriam feitas mais chamadas ao sistema de arquivos e sim ao repositório derivado de LFS que existiria como um processo em nível de usuário. Também seria necessário verificar o funcionamento da interface com o dispositivo de disco utilizada pelo LFS, pois as escritas passariam ao lado do sistema de arquivos. De qualquer maneira, esse trabalho mostra-se menos complexo que lidar com programação de baixo nível próximo ao núcleo do sistema. Porém, Essa solução continua a apresentar os outros problemas descritos na anterior.

A solução adotada por *ObServer2* visa principalmente minimizar escritas dos estados dos objetos ao fim de cada ação e a leitura de objetos em disco. Estes são os pontos mais críticos levantados nos repositórios de Arjuna, visto que seu acesso a objetos armazenados no repositório é lento pois sua implementação utiliza a técnica de cópias *shadow* sobre o sistema de arquivos *Unix*. A utilização de um *log* utilizando as técnicas de propagação de dados *undo/redo*, *undo/no-redo* ou *no-undo/redo*, por si só minimiza o número de escritas síncronas em disco. A combinação de *log* com *cache* otimiza ainda mais o desempenho do sistema.

Os bancos de dados residentes em memória caminham na mesma direção com maior ênfase. A noção de ter cópias primárias dos objetos em memória ao invés de em disco altera os pontos críticos do sistema, tornando o acesso ao repositório uma tarefa mais leve que em bancos de dados convencionais. A utilização de uma alternativa neste sentido, adaptada ao modelo Arjuna, deve surtir um efeito positivo nas duas modalidades de acesso aos objetos. Também devemos considerar que BDRMs geralmente utilizam como meio de propagação de dados *logs no-undo/redo*. Além disto, sua implementação afetaria basicamente os mecanismos de persistência e recuperação de Arjuna, estendendo-se às ativações/desativações dos objetos por outros módulos, sem afetar com muita intensidade a arquitetura geral de Arjuna.

Parece-nos que a utilização de um repositório com características de bancos de dados residentes em memória descrita aqui é a mais viável e deve ter um efeito bastante satisfatório. Em resumo, suas vantagens estão na otimização e diminuição de escritas em disco através do uso de *log*, diminuição de acessos a disco através do uso de memória principal como meio de armazenamento para o repositório e menor complexidade de implementação. Como desvantagem em relação a algumas outras opções, poderíamos citar alteração no ambiente alvo, porém estas alterações são de escopo bem delimitado, não sendo motivo assim de desqualificação desta alternativa.

A utilização de mecanismos de *log* e gerência de memória para prover funcionalidades como objetos e, principalmente, ações atômicas aparece em quase todas as aplicações pesquisadas. Em geral, sua utilização contribui para o bom desempenho destas aplicações. A seguir, veremos o funcionamento do ambiente de computação distribuída Arjuna, particularmente, sua implementação das funcionalidades de objetos e ações.

Capítulo 4

Ambiente Alvo

O projeto do repositório de objetos visa a melhoria do desempenho de Arjuna [28]. Arjuna é um ambiente de programação distribuída implementado em C++ e construído em função do modelo de objetos e ações. Arjuna foi um bom laboratório para este projeto, pois dispõe de várias implementações para seu repositório de objetos, o que nos permitiu a análise de vários tipos de abordagens, e seu código fonte está disponível. Também podemos citar o fato de Arjuna ser um sistema representativo de um conjunto de ambientes de programação distribuída como Camelot [10] e Encina [8].

A seguir, veremos o funcionamento geral de Arjuna, aprofundando-nos mais especificamente nos mecanismos de persistência e recuperação.

4.1 Arquitetura Geral

As principais características de Arjuna são modularidade, flexibilidade e integração de serviços. Para alcançar modularidade Arjuna foi dividido em vários sub-sistemas com interfaces bem definidas. Os outros dois objetivos foram alcançados através da implementação de uma biblioteca de classes em C++, permitindo ao usuário modificar classes para adaptar a funcionalidade do sistema às suas necessidades. Isto propicia flexibilidade e uma maneira integrada e simples de usar serviços diferentes. A Figura 4.1 apresenta a arquitetura do Arjuna [22].

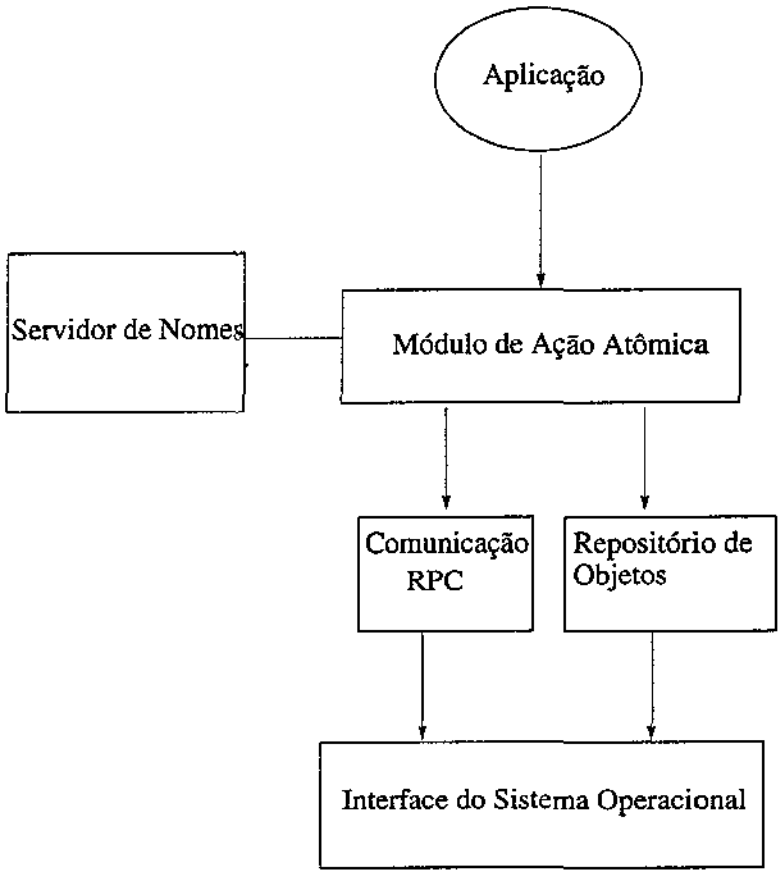


Figura 4.1: A Arquitetura do Arjuna

As referências a objetos são feitas através de nomes. O servidor de nomes provê os serviços de resolução de nomes e localização de objetos. O serviço de resolução de nomes mapeia o nome de um objeto para um identificador único (IU). O serviço de localização utiliza o IU de um objeto para obter o nó em que ele se encontra.

Acessos a objetos remotos são feitos via chamada de procedimento remoto (RPC). O modelo cliente-servidor é adotado para o acesso a um objeto remoto. Um servidor gerencia o estado de um objeto. Quando um cliente deseja executar operações em determinado objeto, ele efetua sua localização através do servidor de nomes e envia a requisição ao servidor. O servidor, então, importa o estado do objeto, executa as operações e exporta os resultados para o cliente. O armazenamento e a recuperação dos estados dos objetos são feitos através de uma *store daemon*. Quando os objetos estiverem ao alcance local do servidor, ele os acessa sem intermediação do *store daemon* por questões de eficiência.

O módulo de ação atômica provê suporte para ações atômicas através de operações de início, término e aborto de ações. Também fornece a interface de programação do Arjuna.

4.2 Arjuna: Repositório de Objetos

Para implementar ações, o módulo de ação atômica trabalha em conjunto com o módulo repositório de objetos. Este módulo será discutido mais detalhadamente, pois está fortemente ligado ao projeto.

O módulo repositório de objetos fornece um meio de armazenamento estável para os estados dos objetos. Estes objetos são identificados pelo IU e seu estado é armazenado na forma *passiva*. Quando um servidor lê um objeto do repositório, ele ativa a representação passiva do objeto que armazena o estado terminado do objeto até então, e importa esse estado para efetuar as operações sobre ele. Quando o servidor escreve um objeto, o processo é inverso. O ciclo de um objeto persistente é mostrado na figura 4.2.

Para que a manipulação de objetos se dê atomicamente, o módulo de ação atômica utiliza o protocolo de término de duas fases (*two phase commit*) [2]. Adicionalmente, o repositório trabalha com a técnica de cópias *shadow*, ou seja, o repositório gerencia duas cópias do estado do objeto: uma armazena o estado do objeto anterior ao início da ação e outra, considerada *shadow*, armazenará o estado do objeto posterior a execução da ação. Em caso de falha, o estado válido permanece como o anterior ao início da ação. Em caso de

sucesso, o estado válido passa a ser a cópia *shadow*, onde terá sido escrito o estado do objeto. Ou seja, o estado do objeto como um todo é ali gravado, apesar de, em geral, apenas alguns atributos (talvez somente um) terem sido modificados. Além disso, o objeto como um todo é trancado (*locked*), sendo que, na maioria das vezes, poucos atributos são modificados durante a ação.

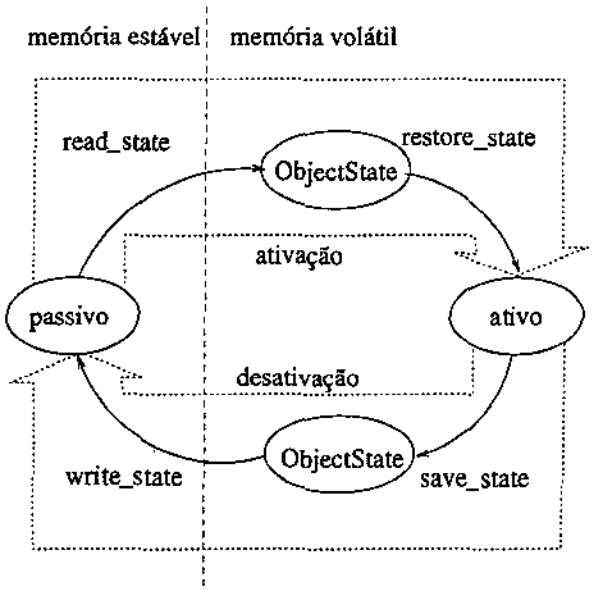


Figura 4.2: Ativação de um Objeto Persistente

4.2.1 Chamadas ao Repositório

Para melhor otimizar o mecanismo de persistência de objetos em Arjuna foi necessária a compreensão não só dos repositórios onde efetivamente os objetos são armazenados, mas também a seqüência de ativação e desativação dos estados dos objetos e outros objetos necessários para efetuar ações atômicas no sistema.

Para melhor visualizarmos a seqüência de ativação e desativação de um objeto durante uma ação atômica "A", vamos analisar o formato típico de uma aplicação simples em Arjuna, que contenha ações e objetos persistentes, observando como o controle de concorrência (*two phase lock*) e de ações (*two phase commit*) interagem com o repositório de objetos.

```
A.Begin()
if setlock(mode)
   A.End()
else
   A.Abort()
```

A figura 4.3 mostra a máquina de estados que representa o funcionamento de uma ação atômica com um único objeto.

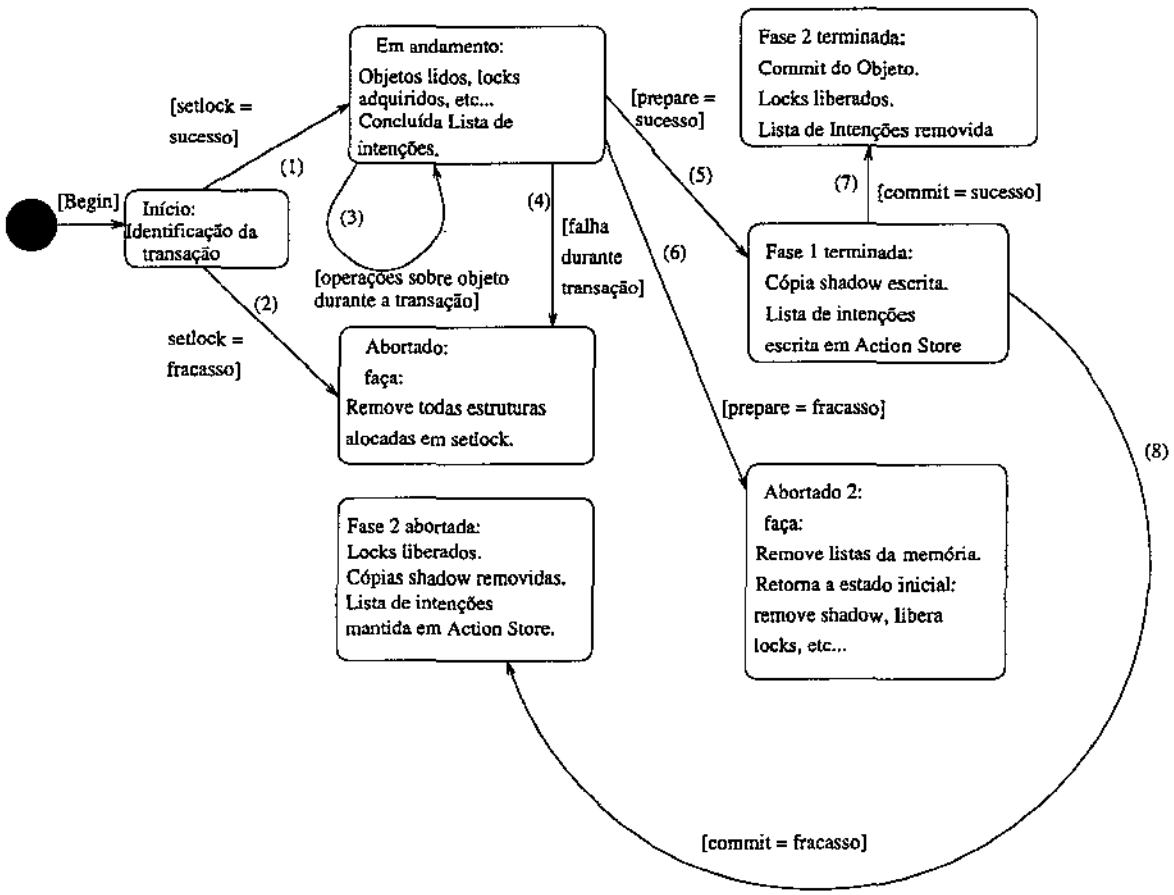


Figura 4.3: Máquina de estados do processamento de uma ação com um objeto em Arjuna

O método `Begin` inicia a ação, determinando sua identificação e seu tipo (Figura 4.3, Início). O método `Abort` é chamado em caso de fracasso do método `setlock` ou da ação em si (Figura 4.3, transições 2 e 4). Os métodos que, em mais alto nível, vão ativar e

desativar os objetos são respectivamente `setlock` e `End` ou `Abort`. Portanto, esses métodos serão analisados mais detalhadamente, principalmente sua relação com os repositórios.

setlock Durante a primeira ação, o objeto repositório (o *default* é *Fragmented Store*) é instanciado conforme os métodos da classe `ObjectStore`. Nas próximas ações, a mesma instância é utilizada. O estado do objeto é lido do repositório corrente. Após, o objeto é inserido na lista de intenções (`pendingList`). Depois disto, é instanciado um objeto de *lock* e também inserido na lista de intenções. Então, o objeto de *lock* é escrito na *Lock Store*. Esse repositório fica em um segmento de memória compartilhada, e é onde efetivamente as trancas são adquiridas (*two phase lock, primeira fase*). Por último, o registro de ativação referente à ação é instanciado e também é inserido na lista de intenções (Figura 4.3). Isto é feito para todos os objetos a serem ativados.

End Este método dá início à primeira fase do protocolo *two phase commit*. Primeiramente, o estado do objeto ativado é retirado da lista de intenções e passado à forma passiva. Então é escrito no repositório como cópia *shadow*. É então inserido em outra lista: lista de preparação (`preparedList`). O objeto de *lock* é o próximo da lista e é também inserido na lista de preparação. Finalmente, é retirado o registro de ativação que sofre o mesmo processo. O processo repete-se para todos objetos a serem salvos depois da ação. Em seguida, é instanciado o objeto repositório *Action Store* e a lista de intenções é gravada neste repositório. A lista de intenções contém um objeto com informações sobre a transação (objetos, trancas, etc). Nesse momento é encerrada a fase de preparação do *two phase commit* (Figura 4.3, Fase 1 terminada).

Terminada a primeira fase do protocolo com sucesso (Figura 4.3, transição 5), inicia-se a segunda. Primeiramente, o estado do objeto é consolidado, tornando a cópia *shadow* válida. Em seguida, as trancas adquiridas para o objeto são liberadas (*two phase lock*, segunda fase). Estes objetos são encontrados através da lista de preparação. Após esse processo, a lista de intenções é removida do *Action Store* (Figura 4.3, Fase 2 terminada). Assim, termina a ação.

Em caso de alguma falha durante a primeira fase, é iniciado o método `phase2Abort` (Figura 4.3, transição 6). Este método remove da memória todas as listas (preparação, intenção, etc) e recupera o estado do sistema anterior ao início da ação. Para isto, as trancas adquiridas são liberadas e as cópias *shadows* são removidas. Entretanto, a lista de intenções é mantida no repositório *Action Store* com fins de recuperação (Figura 4.3,

Abortado 2).

Se houver falhas durante a segunda fase do *two phase commit* (Figura 4.3, transição 8), as trancas são liberadas, as cópias *shadow* são removidas, mas a lista de intenções é mantida em *Action Store* para fins de recuperação (Figura 4.3, Fase 2 abortada).

Abort O método `Abort` remove as estruturas alocadas (listas, objetos, trancas, etc) pelo método `setlock`, inclusive o objeto repositório `FragmentedStore` (Figura 4.3, Abortado).

Caso a ação conclua com sucesso, as estruturas alocadas são desalocadas, inclusive o objeto repositório `ActionStore`. O único a permanecer em memória é o objeto repositório `FragmentedStore`.

Ações encadeadas comportam-se como ações de leitura (que não modificam o objeto) em sua interação com o repositório. As cópias *shadows* são escritas em disco somente na preparação da ação **alto nível** (que engloba as ações encadeadas). Somente a lista de intenções referente à ação alto nível é escrita em *Action Store* para posterioermente ser removida.

A seguir, descrevemos as classes que implementam todas as variações de repositórios em Arjuna.

4.2.2 `ObjectStore`

Classe básica para todos repositórios. Implementa métodos para a criação e remoção de objetos repositório, além de organizar a hierarquia de diretórios da aplicação. A hierarquia de classes de uma aplicação determina a hierarquia de diretórios do repositório associado. O caminho (diretório) de cada repositório é constituído do diretório raiz dos repositórios adicionado da parte final do diretório onde se encontra a aplicação.

Lida basicamente com duas estruturas: `SetUpInfo` e `StoreList`. `SetUpInfo` é uma lista com *flags* de controle e identificado por _tn (`typename`), que indica o tipo do repositório. Nessa estrutura, há ainda uma lista de instâncias (`StoreList`) de repositórios derivados da classe *ObjectStore*.

Há métodos especiais para criação e remoção de instâncias `ObjectStore`, os quais criam e removem a partir da lista `SetUpInfo`. A criação e remoção de uma instância de um objeto

repositório é feita através dessa classe, ao invés dos métodos `new` e `delete` do C++. A idéia é criar apenas uma instância de cada repositório por processo para otimizar o sistema.

4.2.3 `VolatileStore`

Esta classe utiliza como repositório de objetos um segmento de memória (1 MB) compartilhada. Assim, a persistência dos objetos é restrita ao tempo da memória volátil. Para armazenar o repositório em memória estável existe o método **pack** pertencente a classe. Para recuperar da memória estável para o segmento de memória compartilhado existe o método **unpack**.

O segmento de memória é acessado via métodos da classe **sharedseg**. Basicamente, os objetos são recuperados através do método `sharedseg->get` e salvos com `sharedseg->put`. Depois de recuperados, os estados tornam-se ativos através da método `storedata->unpack`. Para salvar um objeto no repositório é necessário torná-lo passivo utilizando `storedata->pack`. A classe `storedata`, que contém estes métodos, armazena temporariamente os estados dos objetos ativos.

Quando é efetuada a leitura de um objeto através do método `read_state`, são recuperados os estados de **todos** os objetos do repositório e colocados numa lista de `storedata`. Após, é feita uma busca seqüencial nessa lista, encontrado o objeto e colocado em uma instância da classe `ObjectState`. Todo repositório *Volatile Store* é trancado no início desse processo e nesse ponto liberada.

Após as operações serem efetuadas, o objeto pode ser salvo no repositório através do método `write_state`. Primeiramente, o repositório é trancado, lidos **todos** os objetos e colocados na lista `storedata`. Uma busca seqüencial em busca do objeto modificado e, caso encontrado, é removido e em seu lugar inserido o novo estado do objeto. Caso o objeto não seja encontrado, o estado do objeto é inserido no primeiro ponto em que haja espaço suficiente na lista, ou seja, um ponto na lista que se inserido o objeto não seja necessário atualizar-se o deslocamento dos objetos seguintes na lista. Nesse momento, **todo** o repositório é salvo e liberado.

Comentários: Nesse repositório percebemos vários pontos passíveis de otimização. Na leitura ou gravação de um objeto, todo repositório é trancado sem real necessidade, e sim pela maneira implementada, que é funcional e simples, porém ineficiente. Além disso,

todos os objetos são recuperados, colocados numa lista intermediária (`storedata`), feita uma busca seqüencial pelo estado procurado e, só então, passado para uma instância de *objectstate*. Há uma repetição de trabalho desnecessária. Os estados poderiam ser lidos já buscando-se o objeto desejado, sem que para isso seja necessário salvá-los em uma lista. Algum tipo de ordenação ou indexação também otimizaria a busca. Durante a escrita, todos os objetos são recolocados no repositório, mas somente o estado do objeto modificado deveria ser salvo.

Essas modificações aumentariam a complexidade da implementação da *Volatile Store*, mas acreditamos que esse custo é pequeno se comparado ao possível ganho de desempenho.

4.2.4 `FileSystemStore`

Nesta classe há métodos relativos à localização dos objetos na hierarquia de diretórios. Seus métodos podem retornar todos *Uids* dos objetos de um tipo (ou classe), e todos os tipos. O arquivo do objeto é identificado pelo *Uid* e *type*. Cria hierarquia de diretórios de acordo com as classes utilizando o método `locate_store` da classe `ObjectStore`. Também há métodos `lock` e `unlock` diretamente no arquivo que contém o objeto.

4.2.5 `MappedFile`

Esta classe implementa funções básica para o repositório *Mapped Store*. *Mapped Store* agrupa os estados de todos os objetos em um arquivo mapeado em memória. As classes que implementam o repositório *Mapped Store* são: `MappedFile`, `SingleTypeMappedStore`, `MappedStore`.

`MappedFile` também fornece funções básicas para a classe `SingleTypeMappedStore`. Fornece as interfaces e implementações para abertura, fechamento, mapeamento e sincronização dos arquivos que contém os objetos. Na ocasião da abertura do arquivo, torna-o múltiplo do tamanho de página utilizado. A sincronização do repositório se faz apenas sobre as páginas que diferirem entre o arquivo mapeado e sua cópia em disco.

O arquivo a ser mapeado em memória é utilizado como repositório de objetos, de onde pode-se recuperar ou salvar os estados deles. Essa classe fornece métodos para acesso aos objetos, sendo a localização dos objetos dentro do arquivo dada pela posição inicial +

deslocamento do objeto no repositório.

4.2.6 SingleTypeMappedStore

Utiliza a estrutura MappedStoreData de forma análoga a StoreData em VolatileStore. Há um número máximo de repositórios (7) e um número máximo de estados por repositório (2048). Cada repositório é uma instância de MappedFile. O repositório corrente é selecionado antes de qualquer acesso a objetos através de uma função *hash*.

Para a leitura de um objeto, o repositório é trancado (usa-se semáforos), e depois todo o repositório mapeado é copiado para uma lista MappedStoreData. Então, é feita uma busca seqüencial nessa lista para encontrar o objeto e transformado em instância de ObjectState. Depois o repositório é liberado. Ao término da ação são necessários dois métodos: write_uncommitted e commit_state.

Basicamente o procedimento desses métodos é o seguinte: tranca o repositório, passa todos estados para lista (descrito anteriormente), busca posição para atualização, atualiza-se (escreve como *shadow* ou passa para estado original) o estado, escreve todos estados em repositório e depois sincroniza-o com o arquivo em disco. Em geral, esses métodos utilizam métodos da classe MappedFile para os acessos mais básicos ao repositório.

4.2.7 MappedStore

Esta classe lida com outra MappedStoreData. Nesse caso, MappedStoreData é uma instância de SingleTypeMappedStore + um identificador obtido através do parâmetro type. Ou seja, enquanto SingleTypeMappedStore manipula objetos de um único tipo, a classe MappedStore trabalha com objetos de vários tipos utilizando instâncias da classe SingleTypeMappedStore.

Esta classe provê basicamente os mesmos métodos de SingleTypeMappedStore. A diferença é que primeiramente é feita uma busca seqüencial em MappedStoreData pelo tipo, e então, utiliza-se o método da instância SingleTypeMappedStore encontrada para acessar o objeto. Por exemplo, o método allobjuid faz busca seqüencial pelo tipo na lista MappedStoreData e depois se utiliza do método SingleTypeMappedStore->allobjuid para terminar a tarefa.

Comentários: Percebemos que esses repositórios sofrem praticamente dos mesmos problemas de desempenho de `VolatileStore`, acrescidos de outros devido a trabalharem com memória estável. A cada acesso a um objeto todo o repositório é trancado. Todo o repositório é passado a lista. Depois todos os objetos voltam ao repositório (mapeado em memória) duas vezes e as páginas alteradas são baixadas em disco duas vezes `write_uncommitted` e `commit_state` ao fim da ação. Além disso, seus métodos de busca são bastante rudimentares. Em `MappedStore`, por exemplo, a busca de um objeto é feita através de duas buscas seqüenciais: uma pelo tipo e outra pelo IU do objeto e `SingleTypeMappedStore`. Existem outros métodos mais eficientes (árvores, hash,...) que poderiam ser utilizados.

Como vantagens, percebemos que durante a leitura é evitado o acesso a disco, pois o repositório está mapeado em memória. Temos aí *cache* para leitura. Além disso, quando muitos objetos são alterados durante a ação, as páginas alteradas desses objetos são escritas de uma vez durante o término da ação. Nos outros repositórios, cada objeto é escrito independentemente e seqüencialmente, o que implica em mais acessos a discos e mais chamadas ao sistema.

4.2.8 ShadowingStore

Neste repositório, os objetos são mapeados um a um em arquivos. Há caracteres especiais no final dos nomes dos arquivos que indicam se o arquivo representa uma cópia diferente da original: "#": *hidden* e "!":*shadow*.

Para encontrar os objetos que serão manipulados, o arquivo com seu estado é buscado na hierarquia de diretórios. Caso o interesse seja a leitura do objeto, este é encontrado, trancado (`locked`) e aberto. Com o conteúdo do arquivo, é instanciado um objeto `ObjectState`, que representa o estado do objeto. Depois o arquivo é liberado e fechado.

No início da primeira fase do *two phase commit*, o objeto é encontrado, trancado e aberto. Após, é efetuada a escrita como cópia *shadow* e baixado em disco. O arquivo é então fechado e a tranca liberada. Para a segunda fase, o arquivo é novamente encontrado e então renomeado sem o caracter especial *shadow*.

Comentários: Esta classe herda métodos relativos a hierarquia de diretórios da classe `FileSystemStore`. Trabalha com granularidade de objetos na manipulação destes (trancas, leituras, escritas, etc), diferente dos anteriores que trabalhavam com o repositório todo em geral. Porém, apesar dessas otimizações, esse repositório ainda faz muito acesso a disco por objeto e por transação.

4.2.9 `ActionStore`

Esta classe não trabalha com cópias *shadows*. Herda de `ShadowingStore` operações com arquivo original. Seu objetivo é o armazenamento de informações para término de ação atômica. Métodos com *uncommit* retornam Falso.

4.2.10 `FdCache`

Esta classe provê métodos e estruturas para a utilização de *cache* pelos repositórios *Fragmented Store* e *Hashed Store*. Segundo [22], esse esquema de *cache* resultou em um desempenho 30 por cento superior na escrita, sem afetar a leitura, em relação ao repositório implementado pela classe `ShadowingStore`.

Provê *cache* para *file descriptor* (fd), blocos de cabeçalho para os arquivos recentemente abertos, e cópias do objeto. A estrutura onde ficam armazenados esses dados é uma fila duplamente ligada, sendo cada elemento composto do fd e do cabeçalho (estrutura `statedictionary`). Há apontadores para o início e o final da fila. Há um número máximo de entradas limitado pelo número de arquivos que um processo pode manter abertos. Este número determina o tamanho máximo do *cache*.

A procura é seqüencial e o elemento encontrado passa para início da fila. Para que um elemento entre no *cache*, verifica-se antes se já há uma entrada ativa mais antiga do elemento na fila. Se isto ocorrer, o elemento passa para o início da fila. Caso contrário, o novo elemento é inserido no início da fila.

Entradas podem ficar desativadas por ocasião de compactação de *cache*. Nesse caso, um número máximo equivalente às entradas atuais divididas por dois permanecem ativas. As outras entradas são desativadas a partir do final da fila. Utiliza algoritmo LRU (*Least Recently Used*) para ordenação das entradas. As entradas desativadas ficam com fd = -1

e os arquivos associados são fechados. A compactação ocorre por problemas de excesso de arquivos abertos. Também há eliminação de entrada em *cache* para o caso repetição de fechamento do arquivo associado, removendo-se a estrutura.

4.2.11 `FragmentedStore`

Neste repositório, as cópias *shadow* e original permanecem no mesmo arquivo, sendo a posição de cada uma determinada no cabeçalho do arquivo. O deslocamento entre as cópias é de 16Mb, o que determina tamanho máximo de um objeto em 16Mb.

Em suas buscas por objetos, procura-se primeiro em *cache*. Operações relativas a *cache* são implementadas na classe `FdCache`. Se a informação desejada estiver em *cache* não é necessário acessar o arquivo. Se for atualização de dados pertencentes ao cabeçalho, apenas o cabeçalho será escrito no arquivo.

No início dos métodos que manipulam os objetos, o arquivo é trancado e aberto (`openandlock`). Para isso, o estado do objeto é primeiramente buscado em *cache*. Se o estado não for encontrado, o arquivo é aberto através dos métodos herdados de `ShadowingStore` (Seção 4.2.8). Caso o fd no *cache* contenha fd -1, ou seja, arquivo fechado, o arquivo é reaberto e recolocada essa entrada no início do *cache*.

Independente de estar anteriormente em *cache*, coloca-se a nova entrada associada no início do *cache* e o arquivo é trancado. Nesse momento, as funções para manutenção de *cache* que se façam necessárias, como compactação, por exemplo, são ativadas. O cabeçalho e a cópia do estado são lidos. Há um tamanho máximo para que a cópia do objeto fique em *cache*.

Na leitura de um objeto, o arquivo com seu estado é primeiramente aberto como descrito acima. É verificado se o objeto em *cache* é legal (tamanho menor que o máximo). Se for, o estado do objeto é lido do *cache*. Caso contrário, lê-se o estado do objeto diretamente do arquivo. Depois, instancia-se `ObjectState` e a tranca do arquivo é liberada.

Neste repositório, há dois modos de funcionamento: com protocolo *commit* completo e relaxado. O completo é o que vem sendo descrito normalmente: escreve cópia *shadow* e passa-a para válida. O modo relaxado escreve apenas cabeçalho de cópia *shadow*, com cópia vazia, e escreve cópia válida na segunda fase do protocolo *two phase commit*.

Vejamos o modo relaxado: Durante a primeira fase do protocolo *two phase commit*, o objeto provavelmente terá uma entrada associada em *cache* e o arquivo deve estar aberto (exceção em caso de compactação). É posta uma tranca no arquivo e é recolocada a entrada em *cache* no início da fila. Então, apenas o cabeçalho é escrito e sincronizado em disco. O arquivo também é liberada da tranca, mas é mantida a entrada em *cache*.

Na segunda fase do protocolo, o arquivo é novamente trancado e é recolocada entrada em *cache*, como já descrito. São escritos o cabeçalho e a cópia válida e sincronizados em disco. A entrada em *cache* é eliminada e o arquivo fechado.

Vejamos agora o modo completo: Durante a escrita, o objeto provavelmente terá uma entrada associada em *cache* e o arquivo deve estar aberto (exceção em caso de compactação). É posta uma tranca no arquivo e recolocada a entrada em *cache* no início da fila. O cabeçalho e a cópia *shadow* são colocados em disco juntos, além de recolocar a cópia do estado do objeto no *cache*. O arquivo é liberada da tranca, mas é mantida entrada em *cache*.

Na segunda fase do protocolo, o arquivo é trancado como no modo relaxado. É recolocada a entrada em *cache* no início da fila, como já descrito. São trocados os valores entre *shadow* e original no cabeçalho. Escreve cabeçalho e baixa-o em disco. Depois o arquivo é truncado com tamanho do cabeçalho + cópia original. Vale citar que só nesse momento, o arquivo passa a ter um tamanho compatível com o objeto, pois anteriormente (`write_uncommitted`), pode ser de aproximadamente 16Mb, caso a cópia *shadow* seja a segunda do arquivo. Quando o arquivo é liberado, sua entrada em *cache* é eliminada e o arquivo é fechado.

Comentários: Otimiza a `shadowingStore`, visto que, durante a escrita não necessita abrir o arquivo referente ao objeto, pois o *file descriptor* do arquivo aberto está em *cache*. O `commit_state` é beneficiado pois, diferente dos outros repositórios em que havia a escrita do estado do objeto em disco, agora só atualiza o cabeçalho e trunca o arquivo (assincronamente), além de não fazer nova abertura. Analisando-se a leitura do objeto em *cache*, podemos verificar que, em geral, isto ocorre somente durante e pela ação atômica que o abriu, pois o *cache* só será utilizado quando um objeto estiver sendo acessado, e não estiver trancado. Em geral, esta situação ocorre na segunda fase do protocolo *two phase commit*, pois a atualização do cabeçalho ocorre sobre o mesmo arquivo sendo trabalhado.

4.2.12 HashedStore

Herda praticamente todos os métodos de `fragmentedStore`, com exceção daqueles que trabalham na árvore de diretórios. A procura de um arquivo na hierarquia de diretórios utiliza o método `hash` da classe `uid`.

4.3 Conclusões sobre os Repositórios de Arjuna

Após analisarmos em detalhe os repositórios existentes em Arjuna, pudemos verificar que é possível melhorar o desempenho do sistema através do projeto e implementação de um repositório de objetos mais eficiente, isto é, adequado às características de processamento impostas pelo uso de ações atômicas. Basicamente, podemos classificar em dois grupos as otimizações possíveis: de leitura e de escrita.

Quanto à leitura, temos um problema de desempenho devido a ausência do uso de *cache* de maneira eficiente. O repositório *Fragmented Store* (Seção 4.2.11) implementado pelas classes `FragmentedStore` e `fdcache` armazena informações em *cache* apenas para objetos que estejam sendo utilizados por alguma transação. Portanto, o *cache* só será utilizado quando um objeto estiver sendo acessado e não estiver trancado. Em geral, esta situação ocorre no final da transação, pois a escrita do cabeçalho ocorre sobre o mesmo arquivo sendo acessado. Neste caso, a abertura do objeto não se faz necessária, visto que o objeto permanece aberto durante toda a ação atômica. Mas não há um *cache* para objetos que não estejam em uso.

No caso do repositório *Mapped Store* (Seção 4.2.5), há cache para leitura dos objetos, porém seu acesso não é otimizado. Pelo contrário, não há ordenação alguma, nem por chave, nem por freqüência de acesso. Além disso, várias vezes todos os objetos do repositório são movimentados do repositório mapeado para a lista `storedata` e vice-versa, o que degrada o desempenho do sistema.

Quanto à escrita, temos problemas em relação ao seu tamanho e freqüência. Geralmente todo o objeto é escrito no final de cada transação. No repositório *Mapped Store*, as páginas alteradas em todo o repositório são sincronizadas de uma só vez ao final da transação, porém, existe a movimentação citada acima entre os objetos no repositório e a lista `storedata`.

No repositório *Fragmented Store*, são necessárias as escritas da cópia *shadow*, da marca de *commit*, da lista de intenções e da remoção da lista de intenções. Portanto, são necessários quatro acessos a disco para a finalização de uma ação atômica com um objeto. Porém, quando temos ações atômicas encadeadas temos um custo menor, pois a lista de intenções fica restrita apenas à ação atômica alto nível.

A análise do funcionamento de Arjuna, particularmente os mecanismos utilizados para prover persistência e recuperação, mostra-nos que Arjuna, ao contrário das aplicações vistas no Capítulo 3, não utiliza, ou utiliza de forma ineficiente, os mecanismos de *log* e gerência de memória. Para obter ações atômicas, Arjuna utiliza cópias *shadows*, implementando uma técnica *no-undo/no-redo*, reconhecidamente menos eficiente que outras técnicas que utilizam *log* (Seção 2.2).

Verificando esses fatos, concluímos que a adição destes mecanismos (*log* e gerência de memória) como meio para obter-se ações atômicas melhorará o desempenho geral do ambiente Arjuna. A adição destes mecanismos se dá através da construção de um novo repositório que substituirá os antigos repositórios de Arjuna. A seguir, descrevemos este novo repositório.

Capítulo 5

HiPer Store (High Performance Store)

O estudo comparativo dos mecanismos de persistência adotados por Arjuna (Capítulo 4) com outros mecanismos de persistência encontrados na literatura (Capítulo 3) indica que Arjuna poderia ter seu desempenho melhorado caso fosse munido de um repositório de objetos que combinasse mecanismos de *log* e gerência de memória. Esta conclusão nos levou a construção de *HiPer Store* (*High Performance Store*), um novo repositório de objetos para Arjuna.

Este Capítulo apresenta o funcionamento do repositório implementado, descrevendo sua arquitetura de *software*, algoritmo e principais otimizações conseguidas. A seguir, ilustramos o funcionamento do novo repositório através do acompanhamento da execução de uma aplicação.

5.1 Arquitetura de *Software*

O repositório *HiPer Store* utiliza mecanismos diferentes dos atuais repositórios de Arjuna para prover persistência e recuperação. Mecanismos de *log* e de gerência de memória são combinados para melhorar o desempenho de escritas e leituras nos repositórios de objetos de Arjuna. Em *HiPer Store*, consideramos que praticamente todos os objetos devem permanecer em memória [11]. Entretanto, com o advento de dados cada vez maiores, como dados hipermídia, talvez a memória disponível nunca venha a comportar realmente todos os dados [17]. Consideramos então, a necessidade de haver uma combinação de

aproximações em que temos parte dos objetos temporariamente armazenados somente em disco e outra parte em memória e disco.

Memory Store foi construído a fim de gerenciar cópias de objetos em memória e *Disk Store* para lidar com cópias de objetos em disco. Cópias de objetos que se encontram em *Memory Store* são consideradas primárias, enquanto as cópias encontradas em *Disk Store* são chamadas de reservas. A terminologia adotada aqui é similar à utilizada pelos projetistas de bancos de dados residentes em memória (Seção 3.2).

Apesar de eficiente, percebemos que essa aproximação não possui mecanismos para tolerância a falhas de *hardware* (queda de sistema, por exemplo). Neste caso, não há necessariamente cópia atualizada do estado do objeto em memória estável. Para resolver esse problema de forma eficiente foi construído *Log Store*. *Log Store* gerencia um *log* de transição *redo/no-undo* (Seção 2.2) em meio estável e mapeado em memória. Optamos pela utilização de um *log* porque sabemos que a utilização de técnicas *no-undo/no-redo*, como cópias *shadows*, para implementar mecanismos de recuperação é menos eficiente que outras técnicas que utilizam *log* (Seção 2.2). Isto acontece porque as técnicas que utilizam *log* necessitam de menos escritas síncronas em disco que cópias *shadow*.

O *log* é gerenciado através de *Log Sequence Number* (lsn) (Seção 3.2) armazenados em uma tabela *hash*, chamada de *lsnTable*. Esta tabela é encarregada de indicar todos os objetos desatualizados e seus respectivos LSNs. *lsnTable* é armazenada em memória principal estável e é utilizada tanto para gerência de *log* como para recuperação.

Como vimos, *HiPer Store* divide-se em 3 componentes principais que, em conjunto, implementam o novo repositório de Arjuna:

- *Disk Store*,
- *Memory Store*,
- *Log Store*.

A Figura 5.1 ilustra a funcionalidade de cada parte de *HiPer Store*.

A implementação dos diferentes componentes de *HiPer Store* foi feita de forma integrada a Arjuna. As classes `MemoryStore`, `DiskStore`, `LogStore` e associadas formam o repositório *HiPer Store*. A Figura 5.2 situa o repositório na hierarquia de classes Arjuna. As Figuras

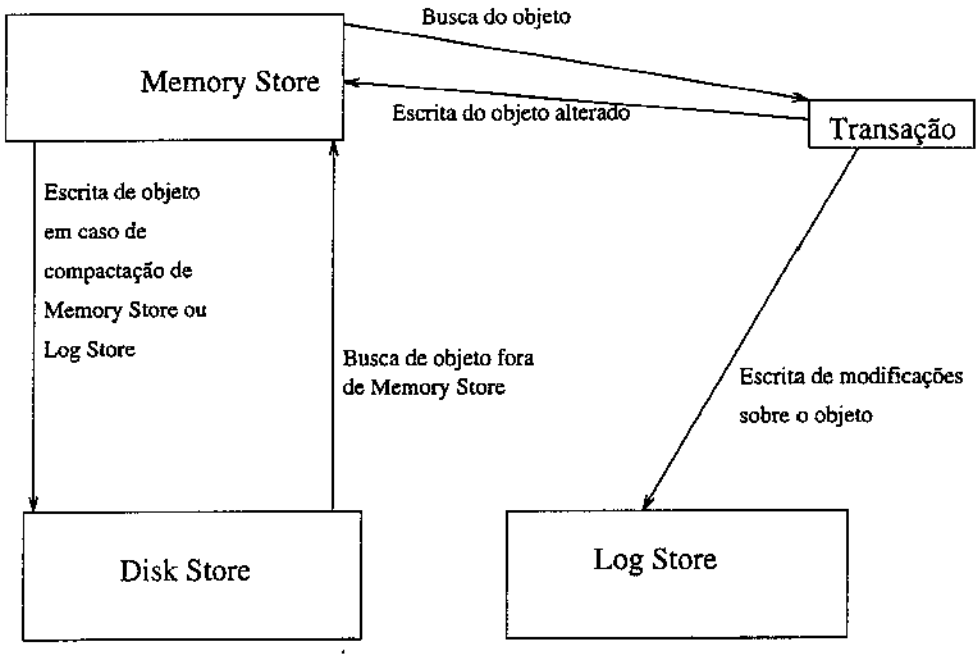


Figura 5.1: Funcionamento da *HiPer Store*

5.3, 5.4, 5.5 e 5.6 apresentam o modelo das classes que constroem o repositório *HiPer Store*.

Instâncias da classe `LogRecord` são as entradas do *log* gerenciado por *Log Store*. Na classe `LogRecord`, o atributo `typeRecord` indica o tipo do registro: Normal (armazena as modificações do objeto), de *Prepare* e de *Commit*. `objUid` é o identificador do objeto associado ao registro. `lsn` (*log sequence number*) indica o endereço em *log* desse objeto. O atributo `active` informa se o registro ainda é válido ou se seu espaço pode ser reutilizado. O registro fica desativado por ocasião da compactação de *Log Store* ou da retirada de elementos de *Memory Store*, que também implica em desativação de registros de *log*. O registro tem ainda o lsn do registro anterior referente ao objeto, além da informação propriamente dita do registro. Caso o registro seja Normal, conterá as modificações no objeto no seguinte formato: posição dos *bytes* modificados e seus conteúdos, diferença entre os estados e seu tamanho em caso de ampliação do estado e tamanho da cópia modificada, em caso de diminuição do estado.

A classe `TableEntry` contém o identificador do objeto e o `lsn` do último registro em *log* dele. Além disso, há um apontador para outro registro para tratamento de colisões na tabela *hash*. Instâncias desta classe são as entradas para a classe `TableLSN`. A instância

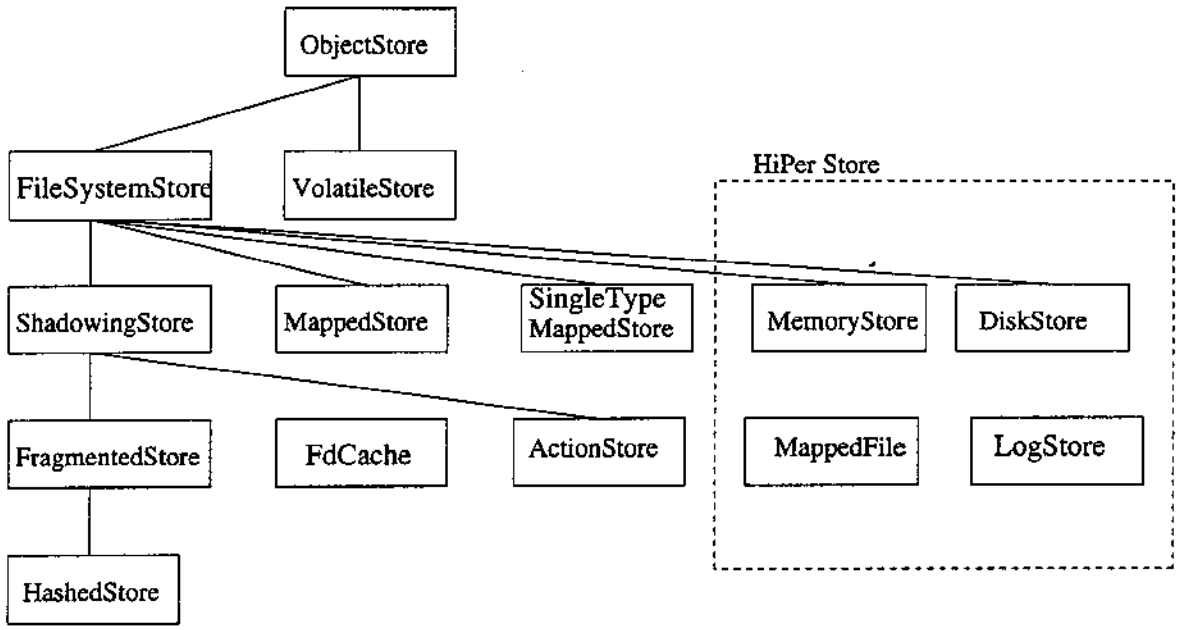


Figura 5.2: Hierarquia de Classes de *HiPer Store* integrada aos Repositórios Arjuna

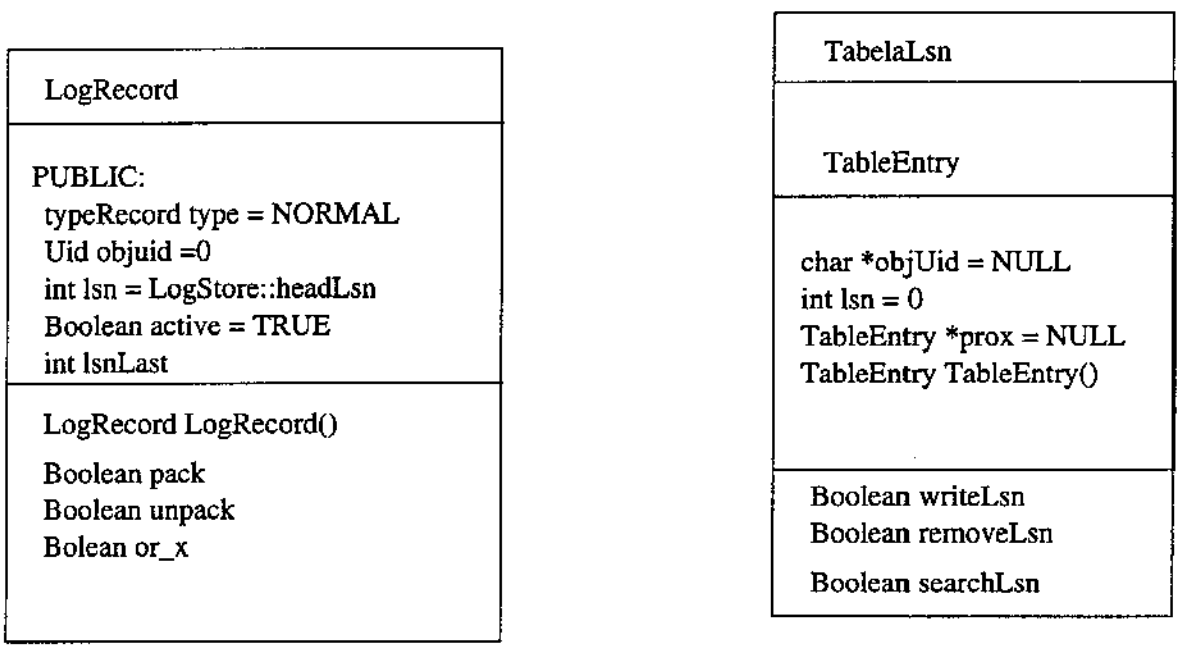


Figura 5.3: Classes associadas a *HiPer Store*

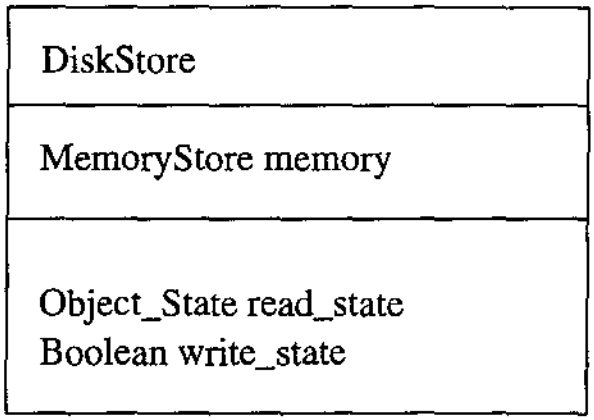


Figura 5.4: Classe `DiskStore`

de `TableLsn` é construída como um vetor de `TableEntry` e gerencia o lsn dos objetos que contém registros em *log*. Deve ficar em memória primária estável, não em disco, pois tem acesso freqüente e isto causaria uma perda de desempenho inaceitável. Para solucionar isto, consideramos que a instância de `TableLSN` deve ficar em memória com bateria (Seção 3.2).

A classe `DiskStore` implementa o repositório *Disk Store*. Esta classe contém os métodos para manipulação das cópias reservas do objeto. É filha da classe `FSStore` (Seção 4.2.4). Seus métodos são semelhantes aos encontrados na classe `ShadowingStore` (Seção 4.2.8), mas não implementam cópias *shadow*.

A classe `LogStore` é o componente central do repositório *Log Store*. É responsável pela manipulação de registros em *log*. Contém uma instância da classe `MappedFile` (Seção 4.2.5) e através dela cria e mapeia para memória o arquivo que tem a função de *log*, além de efetuar a entrada/saída no *log*. É criada por uma instância de `MemoryStore` a que também tem acesso, afim de ativar métodos de retirada de objetos de *Memory Store* durante a compactação do *log*.

Instâncias da classe `memoryEntry` são as cópias primárias dos objetos. A classe `MemoryStore` é responsável pelo repositório *Memory Store*. É aqui que são gerenciadas as cópias primárias dos objetos. Por isto, esta classe contém as interfaces com os outros módulos de Arjuna para a manipulação de objetos. Além disso, aciona instâncias de `DiskStore` e `LogStore` para lidar com as cópias reservas dos objetos e o *log*.

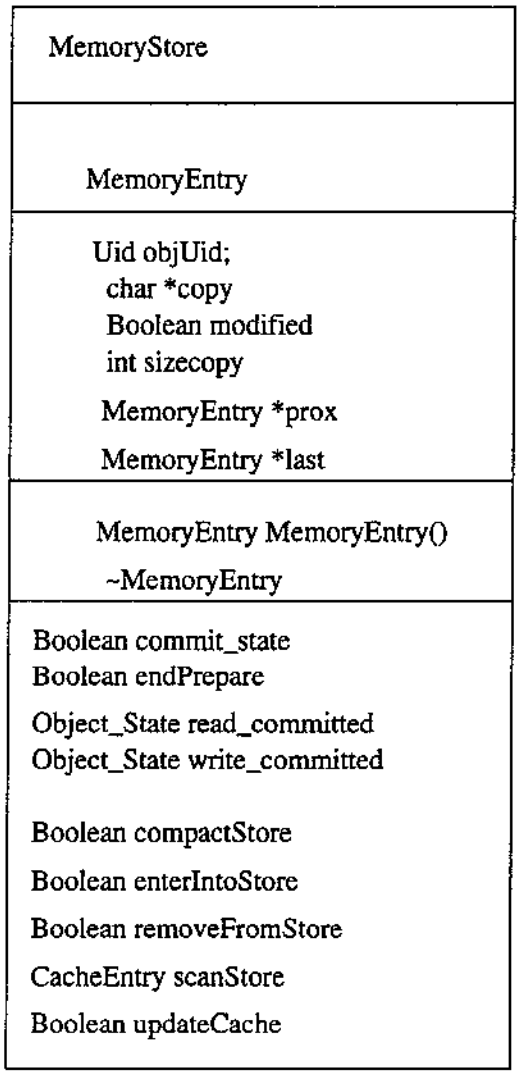


Figura 5.5: Classe `MemoryStore`

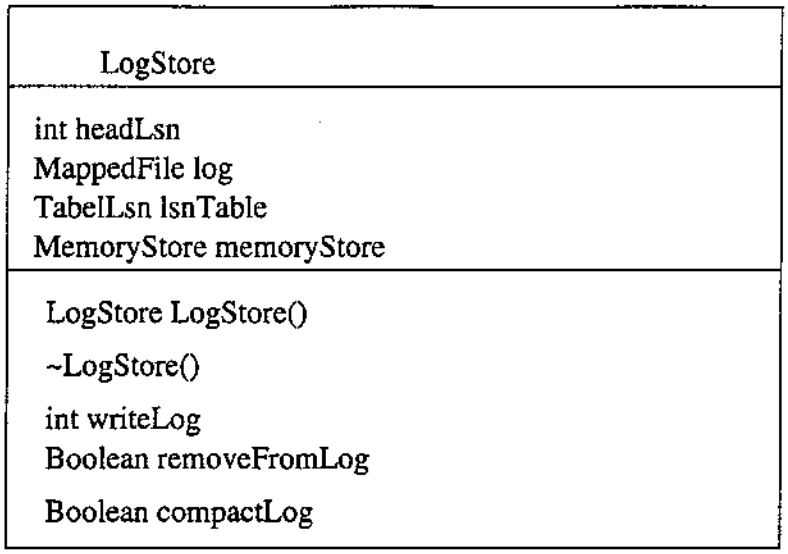


Figura 5.6: Classe `LogStore`

5.2 Algoritmo

Descreveremos agora, o funcionamento do algoritmo implementado em *HiPer Store*, visualizando como seus componentes interagem para proverem persistência em Arjuna. Na seção anterior, vimos que *Memory Store* é responsável pela manipulação das cópias primárias dos objetos, *Disk Store* pela manipulação das cópias reservas e *Log Store* pela gerência do *log*.

As cópias gerenciadas por *Memory Store* ficam em memória compartilhada e são acessadas através de uma árvore AVL [15] indexada pelo identificador único do objeto. Quando estados de objetos são modificados por ações, apenas sua cópia primária é modificada. Ao final da primeira ação, a cópia reserva é considerada desatualizada. A manutenção de entradas em *Memory Store* é feita através do algoritmo LRU: cada objeto acessado é passado para o final da fila de objetos a serem retirados da memória. *Memory Store* cresce até o limite possível, determinado pelo tamanho da memória ou pelo número de segmentos de memória compartilhada disponível no sistema. Quando a memória é preenchida e precisa-se de determinado objeto que não tem cópia primária, há a necessidade de escrever objetos em disco para liberar espaço em memória para a criação da cópia primária do objeto. Os objetos escolhidos para serem escritos em disco são os primeiros da fila, sendo que o menor número deles sairá da memória para a entrada do novo objeto. Ou seja, o

espaço é liberado sob demanda. A Figura 5.7 mostra manutenção de entradas em *Memory Store*.

Como podemos ter vários processos modificando o repositório, torna-se necessário o controle de concorrência sobre a fila LRU e a árvore de acesso AVL. O controle sobre a fila LRU é feito sobre o elemento modificado e seus vizinhos, mantendo assim a consistência da fila. Quanto à árvore AVL, o controle de concorrência é feito sobre toda a árvore, e assim sobre todo o repositório, através de trancas de leitura e escrita.

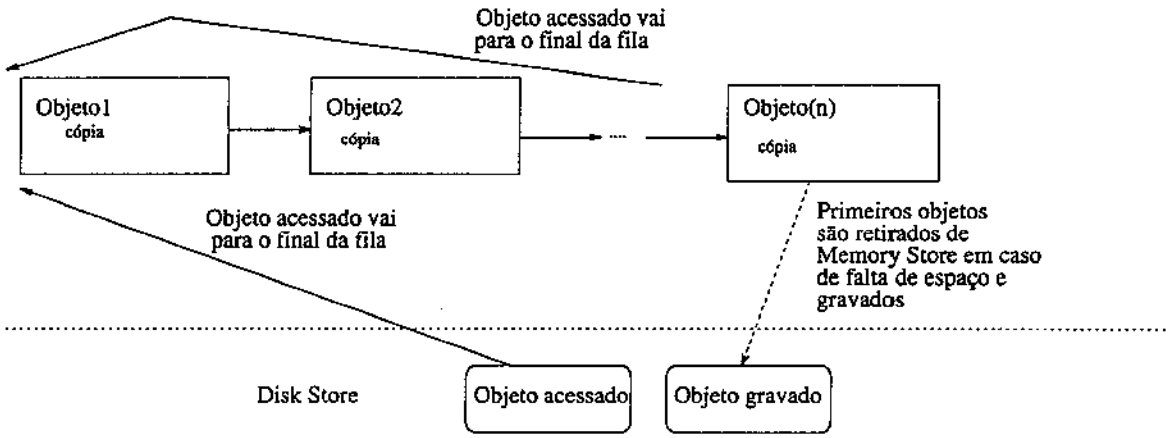


Figura 5.7: Manutenção de entradas em *Memory Store*

Log Store armazena as modificações (diferenças) existentes entre a cópia primária e a reserva do objeto. Chamaremos estas modificações de histórico do objeto. O histórico é gravado por *Log Store* através de registros de modificações, de *Prepare* e de *Commit* de ações. O registro contendo as modificações é obtido através da comparação entre os estados linearizados da cópia primária e do novo estado do objeto modificado pela ação. A recuperação por *log* deve seguir a mesma lógica, tendo como objeto base a cópia reserva.

Suponhamos uma aplicação qualquer, com mais de um cliente fazendo acesso aos mesmos objetos. Inicialmente, não há objetos persistentes. A primeira cópia de um objeto desta natureza terá como cópia reserva um objeto vazio. Este objeto base é escrito em *Disk Store* durante a criação do objeto. Em *Log Store* é armazenado o histórico de atualizações do objeto e, ao final da ação, será inserido em *Memory Store* a cópia do objeto com as alterações ocorridas durante a ação atômica e que, a partir deste momento, será considerada a cópia primária. Havendo necessidade de manipulação de um objeto, verificar-se-á a presença de sua cópia primária, para só depois, caso não encontrada, fazer acesso a sua cópia reserva. A cópia reserva pode estar atualizada ou não. Caso esteja atualizada

(referência a objeto não existente em *lsnTable*) a cópia reserva é utilizada pela ação. Caso contrário, necessariamente houve alguma falha. O mecanismo de recuperação atualizará a cópia reserva através de seu histórico em *log* para que possa ser utilizada pela ação.

Durante a ação, cada objeto lido de *Disk Store* é armazenado em *Memory Store*. O protocolo *two phase commit* continua sendo usado, mas sua implementação agora não utiliza mais o mecanismo de cópias *shadows*. Agora, na fase de preparação do *two phase commit*, são gravados em *Log Store* registros com as modificações do objeto, juntamente com o registro de *Prepare*. Na segunda fase, são efetuadas a inserção do registro de *Commit* e a remoção do registro de *Prepare* em *Log Store*. Em seguida, a cópia primária do objeto é atualizada.

O *log* é construído como um arquivo Unix que é mapeado em memória através da chamada de sistema `mmap`. O acesso ao *log* utiliza métodos herdados da classe `MappedFile` (Seção 4.2.5). Todas as modificações dos objetos são escritas primeiramente no *log* virtual e só são atualizadas em disco no final da primeira fase do *two phase commit*, juntamente com o registro de *Prepare*, que anteriormente iria para o repositório *Action Store* (Seção 4.2.9). *Action Store* é utilizado por Arjuna para guardar informações referentes ao protocolo *two phase commit*; informações que juntamente com a cópia *Shadow* armazenada permitem a recuperação do estado do objeto consistente em caso de falha. *Log Store* mantém todas as informações referentes a recuperação no mesmo repositório. Na segunda fase do *two phase commit*, será colocado no *log* o registro de *Commit* da ação e, depois, retirado o registro de *Prepare* que agora não é mais necessário. Após estes passos, o *log* é novamente sincronizado. As sincronizações de *log* baixam em disco apenas as páginas que diferirem entre o *log* físico e seu mapeamento em memória. Só permanecerão em disco registros para *redo*. Caso a ação aborte antes do final da primeira fase, nenhum registro permanecerá no *log*. Se a ação abortar entre o final da primeira fase e da segunda, os registros de atualização referentes a ação são retirados do *log* virtual e este é sincronizado.

Há momentos em que os meios de armazenamento gerenciados por *Log Store* ou *Memory Store* podem se exaurir. Para evitar que isto ocorra em *Log Store*, o *log* é comprimido cada vez que atinge metade de sua capacidade. No Capítulo 2.2, descrevemos sob que circunstâncias registros podem ser retirados do *log*. Na compactação, alguns objetos são escolhidos e todas suas entradas referentes a ações já finalizadas são eliminadas do *log*. Isso implica na sincronização das cópias primária e reserva destes objetos. Notemos que sendo as cópias primárias atualizadas, não há necessidade de *undo* em nenhum momento. Este processo também é utilizado em ObServer2 (Seção 3.1.6). A seqüência de atualização da cópia reserva é a seguinte:

- escrita da cópia primária sobre a reserva,
- retirada de registros de *log* associados e
- caso não haja mais registros do objeto no *log* (nenhuma ação em andamento manipula o objeto), retirada da entrada associada ao objeto da *lsnTable*. O fato de um objeto estar sem entradas em *lsnTable* é considerado como marca de atualização do objeto.

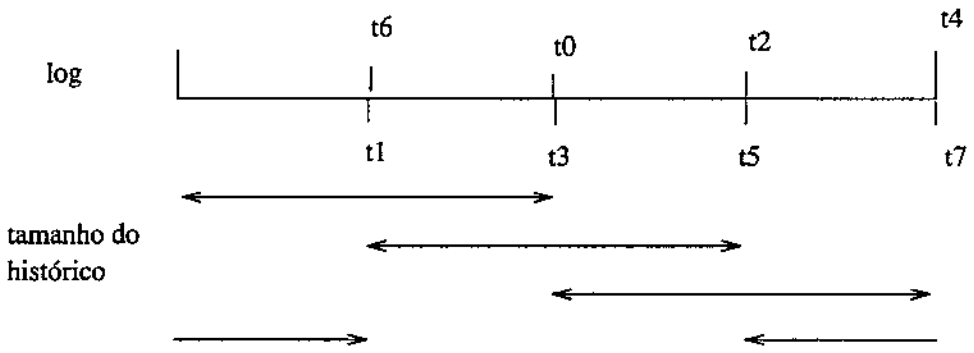
Notamos que, entre o primeiro e o terceiro passo, a cópia do objeto está atualizada mas não está marcada como tal. Entretanto, isto não afeta a correção da recuperação já descrita, pois a reaplicação do histórico sobre a cópia atualizada não altera seu valor.

Perceba que este processo além de liberar espaço em *Log Store*, também torna o histórico da base de dados menor. Na realidade, a compactação de *log* em *HiPer Store* exerce função semelhante às técnicas *log driven backup* e *checkpoint* em bancos de dados residentes em memória (Seção 3.2), tornando o tempo de recuperação menor. A Figura 5.8 mostra a variação do tamanho do histórico no *log*.

Note que o tamanho máximo do histórico é teoricamente constante, movendo-se pelo *log*. Esta técnica é conhecida como janela móvel (*moving window*) do *log* [19]. Em *HiPer Store*, o tamanho da janela móvel varia entre entre 0 (estado inicial) e metade do *log*, sendo que, depois de algum tempo, a variação deveria estar sempre entre a metade e um quarto do *log*.

O último passo na atualização da cópia reserva tem efeitos bastante positivos. De fato, muitas vezes, objetos pouco acessados mantém entradas na *lsnTable*, o que diminui a velocidade de acesso aos elementos da tabela. A compactação do *log* acaba por remover entradas pouco utilizadas, liberando espaço em memória e otimizando o desempenho das buscas efetuadas na *lsnTable*.

Entretanto, é necessário haver um controle de concorrência entre a compactação do *log* e a inserção de registros referentes a ações em andamento para manter a consistência entre as cópias primárias, as cópias reservas e seus históricos em *log*. Isto pode adiar a compactação até o preenchimento total do *log*. Neste caso, as ações que estiverem em andamento liberam as trancas referentes ao *log* para que espaço seja liberado. Depois refazem os passos necessários para escrever, de forma consistente, seus registros em *log*. Note que esta situação dificilmente ocorrerá, sendo a tendência normal haver sempre



t0: histórico chega a metade do log
t1: log sofre compressão até metade do tamanho atual

t2: histórico chega a metade do log
t3: log sofre compressão até metade do tamanho atual

t4: histórico chega a metade do log
t5: log sofre compressão até metade do tamanho atual

t6: histórico chega a metade do log
t7: log sofre compressão até metade do tamanho atual

Figura 5.8: Compactação de *log* em *HiPer Store*

espaço em *log* visto que a compactação se inicia com bastante antecedência (metade do *log* ainda está livre).

Há uma situação em que poderíamos ter *deadlock*. Alguma ação pode escrever o registro de atualização e demorar a escrever o registro de *Commit*, de forma que o *log* dê a volta neste intervalo. Nesta situação, a ação não pode ser concluída por falta de espaço em *log* e a compactação não pode prosseguir porque o último registro do *log* pertence a uma ação não concluída. Note que esta situação pode ocorrer se a ação for interrompida depois de ter escrito o registro de atualização e antes do registro de *Commit*. Neste caso, detectamos a situação de *deadlock* e abortamos a ação, removendo também o registro de atualização.

No caso de *Memory Store*, não é necessária a compactação. Apenas libera-se espaço suficiente para a nova entrada e os objetos retirados de *Memory Store* são escritos em *Disk Store*. Ao atualizar-se uma cópia reserva, ocorre a retirada dos registros em *log* que refletiam o histórico do objeto. Portanto, a retirada de objetos em *Memory Store* implica na retirada de registros de *Log Store*. mas a recíproca não é verdadeira.

Ao final da aplicação, o mapeamento do *log* é removido da memória. Cada aplicação faz seu próprio mapeamento do *log*, evitando a permanência de várias cópias do *log* em memória. As cópias em memória são mantidas, pois são consideradas primárias.

Boa parte do controle de concorrência é feita utilizando semáforos. Os semáforos são criados a partir dos identificadores dos objetos, porém, não são removidos da memória juntamente com a cópia primária do objeto. Assim, acaba crescendo rapidamente o número de semáforos utilizados, exaurindo rapidamente este recurso do sistema. Por isto, estabelecemos um número máximo de semáforos a serem utilizados e, quando passamos deste número, destruímos o semáforo após seu uso. Infelizmente, este semáforo pode estar sendo utilizado por outro processo, ou mesmo pelo próprio na compactação do *log*. Por isto, é necessário haver controle de concorrência sobre a utilização de semáforos e sua destruição. O controle é feito através de um semáforo específico para acesso a outros semáforos.

Existem alguns outros pontos de concorrência além dos descritos até aqui: a *lsnTable* precisa de controle de concorrência sobre suas entradas. Este controle é feito implementado-se uma política de trancas de leitura e escrita sobre cada fila da tabela *hash*.

5.3 Otimizações

Destacamos os principais aspectos que modificam o repositório HiPer Store em relação aos repositórios de Arjuna. Procuramos especificar se as modificações são conseqüência do uso do mecanismo de *log* ou de gerência de memória:

- A escrita do registro de modificações em *log* melhora a eficiência em relação a escrita do estado do objeto em disco como era feito anteriormente em Arjuna. A escrita é menor, pois não é necessário escrever o objeto como um todo, mas apenas suas alterações, além da escrita ser seqüencial em *log*.

- Em ações em que mais de um objeto é salvo, o processo apresenta outras vantagens. A escrita referente a todos objetos é fundida em uma única entrada, enquanto que anteriormente, os objetos eram salvos um a um. Isto ocorre pela centralização das entradas em um único arquivo, o *log*.

- Devemos considerar a eliminação de entradas devido a gravação no mesmo repositório de informações que eram salvas anteriormente em dois repositórios (*Fragmented Store* e *Action Store*). Ao final da primeira fase do protocolo, a sincronização é feita sobre as páginas alteradas do *log* com os registros incluídos dos objetos alterados e do registro *Prepare* associado. Portanto, se compararmos os repositórios comumente usados em Arjuna (*Fragmented Store* e *Action Store*) temos a diminuição de sincronizações em disco na fase de preparação de n+1 para 1, onde n é o número de objetos alterados.

- Os registros de *Commit* são sincronizados ao mesmo tempo, juntamente com a remoção do registro de *Prepare*, ao final da segunda fase do *two phase commit* em *log*, enquanto anteriormente era necessário transformar a cópia *shadow* em *committed* para cada objeto que fosse modificado pela ação, e remover a lista de intenções de *Action Store*. Portanto, temos a diminuição de sincronizações na segunda fase do processamento da ação de n+1 para 1, onde n é o número de objetos modificados pela ação.

- Geralmente o objeto é lido de *Memory Store*, evitando-se operações em disco.

- A atualização da cópia primária é feita em *Memory Store*, evitando-se operações em disco.

Vemos que o repositório *HiPer Store* otimiza os repositórios Arjuna nos seguintes pontos: leitura e atualização da cópia primária (*Memory Store*), escrita de atualizações do objeto (*Log Store*), freqüência de sincronizações em disco, graças ao *log* virtual (*Log Store*).

Acreditamos que as modificações que terão efeitos mais positivos em relação ao desempenho de Arjuna serão aquelas feitas devido a utilização do *log*, visto que a diminuição maior de operações em disco, principalmente escritas, é principalmente devido a utilização do mecanismo de *log*. Realmente, é esperado que a troca de uma técnica *no-undo/no-redo* (cópias *shadow*) por uma *no-undo/redo* que utiliza *log* otimize bastante o desempenho do sistema.

A utilização de cópias primárias em memória, como ocorre em bancos de dados residentes em memória, também otimiza o sistema. Sua contribuição é muito importante, visto que a tendência de queda nos custos de memória principal possibilitam a utilização desta técnica. Como bancos de dados residentes em memória também utilizam *logs*, acreditamos que esta abordagem é realmente a mais adequada para a implementação de um repositório em Arjuna.

5.4 Exemplo

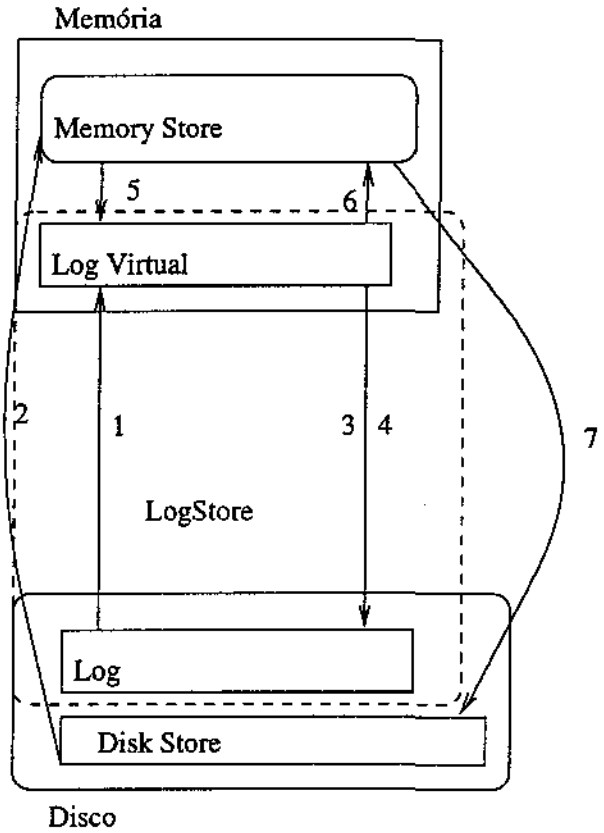
Para ilustrar simplificadamente o comportamento de *HiPer Store* utilizaremos como aplicação uma pilha persistente e atômica. Cada modificação sobre a pilha (empilha, desempilha) é feita atomicamente. A Figura 5.9 ilustra o funcionamento genérico de *HiPer Store*. Sobre a figura, comentaremos o que ocorre durante a execução desta aplicação específica. Na iniciação de Arjuna é criado o arquivo de *log* e mapeado em memória (5.9, passo 1). Inicialmente, temos a criação da pilha através de uma ação atômica. É escrita uma cópia vazia em *Disk Store* e gravado os dados da pilha no *log* como seu histórico. Ao final da ação, é criada a cópia primária da pilha em *Memory Store* (5.9,passo 2). A primeira ação atômica é empilha. O código deste método é transcrito a seguir para analisarmos melhor o que ocorre.

Aplicação:

```
main()
{
(1)   stack = new Stack();
(2)   stack.push(1);
}

void Stack::Push(enum stackOutcome& outcome, int  value)
{
(3)    AtomicAction A;
(4)    A.Begin();
(5)    outcome = unknown;
(6)    if (setlock(new Lock(WRITE), 0) == GRANTED)
(7)    {
(8)       values[top] = value;
(9)       top ++;
(10)      outcome = done;
(11)   }
(12)
(13)   if (outcome == done)
(14)   {
(15)     if (A.End() != COMMITTED)
(16)       outcome = not_done;
(17)   }
(18)   else
(19)   {
(20)     A.Abort();
(21)     outcome = not_done;
       }
}
```

A primeira chamada ao método indica o valor 1 a ser empilhado (linha 2). Dentro do método *setlock* (linha 6) o objeto é buscado em *Memory Store* e encontrado. Depois disso, as trancas são adquiridos e a pilha recebe o elemento 1. No método *End* (linha 15), o objeto com estado anterior é buscado em *Memory Store* e é construído um registro indicando as modificações entre os dois estados do objeto. A modificação representa apenas a presença do valor 1 na pilha. A modificação é indicada através de um *ou exclusivo byte a byte*



Passos:

1. Mapeamento do log físico para memória na inicialização
2. Acesso a um objeto em Disk Store e passagem para Memory Store
3. Sincronização de log no final da primeira fase do two phase commit
4. Sincronização de log no final da segunda fase do two phase commit
5. Remoção de registros em log associados a objeto salvo em Disk Store
6. Em compactação de log, indicação de objetos que serão salvos em Disk Store
7. Gravação de objeto em Disk Store

Figura 5.9: Execução de uma aplicação sobre HiPerStore

entre os dois estados, indicando a posição e o conteúdo das modificações no estado do objeto. Em caso de modificação do tamanho do objeto, é construído um *buffer* contendo a ampliação (em caso de aumento do estado) o tamanho, e a posição de corte (em caso de diminuição do estado). Por exemplo:

obj1: 0000010100100
obj2: 0000010111100001
ou exclusivo: 0000000001100

Resultará em um registro de modificação no *log* composto da seguinte forma: posições 9 e 10, modificações 1 e 1, tamanho 3 e *buffer* 001. O registro de modificações é escrito no *log* virtual, e em seguida os objetos de *Prepare* são transformados em registros de *Prepare* e também inseridos em *Log Store*. Ao final desta fase, o método *endPhase* sincroniza as páginas alteradas do *log* em disco (5.9, passo 3). A partir daí, inicia-se a segunda fase do *two phase commit*. O registro de *Commit* é escrito em *Log Store* e o registro de *Prepare* removido. Então, o método *endPhase* sincroniza novamente as páginas alteradas em disco (5.9, passo 4).

Quando *Log Store* necessitar de mais de espaço e os registros referentes ao objeto pilha forem escolhidos para liberar esse espaço (5.9, passo 6), a cópia em *Memory Store* será gravada em *Disk Store* (5.9, passo 7) e seus registros em *Log Store* serão liberados (5.9, passo 5).

Quando o *Memory Store* necessitar de espaço para novos objetos, o objeto pilha pode ser eleito para ser retirado para a liberação desse espaço. Isso implica na escrita do objeto residente em *Memory Store* no repositório *Disk Store* (5.9, passo 7) e a remoção dos registros em *Log Store* relacionados a pilha e que pertençam a ações já finalizadas (5.9, passo 5). Em seguida, o objeto pilha é removido de *Memory Store* e a nova entrada pode residir em *Memory Store* (5.9, passo 2).

A Figura 5.10 mostra a máquina de estados referente ao processamento de ações atômicas em Arjuna, quando utilizamos o repositório *HiPer Store*. Percebemos que a lógica do processamento de ações atômicas é pouco modificada em relação a máquina de estados descrita anteriormente (Seção 4.2.1), mantendo-se inclusive, quase inalterada a interface com o repositório.

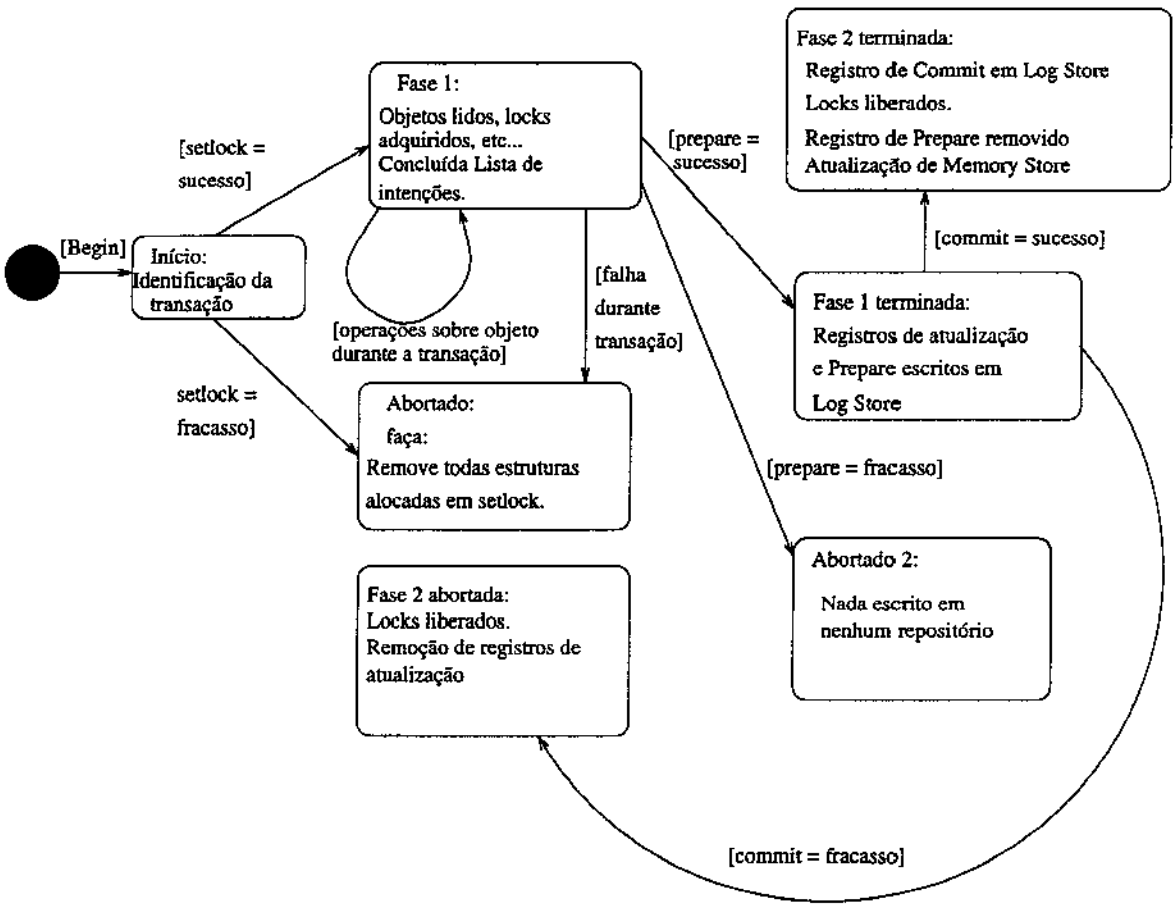


Figura 5.10: Máquina de estados do processamento de ações atômicas em Arjuna utilizando HiPer Store

A descrição do funcionamento de *HiPer Store* demonstra a consistência do repositório implementado e mostra os principais pontos de otimização. *HiPer Store* modifica o Ambiente Arjuna utilizando mecanismos de *log* e de gerência de memória para prover objetos e ações. Com isto, o repositório deixa de ser prejudicado pelas entradas síncronas necessárias anteriormente, melhorando assim, seu desempenho.

No próximo capítulo, veremos a comparação entre o repositório *HiPer Store* e o repositório *Fragmented Store* utilizado atualmente em Arjuna, mostrando em que grau *HiPer Store* otimiza o ambiente Arjuna. Em seguida, apresentamos as considerações finais da dissertação.

Capítulo 6

Discussão e Conclusões

Neste capítulo, compararemos o desempenho de Arjuna utilizando seu repositório mais eficiente e utilizando o repositório desenvolvido *HiPer Store*. Discutiremos a respeito destes resultados e, em seguida, apresentaremos as considerações finais da dissertação.

6.1 Discussão

O repositório *HiPer Store* foi implementado em sua totalidade no que tange persistência em Arjuna. Entretanto, o mecanismo de recuperação não foi totalmente aprofundado devido a seu escopo não pertencer propriamente aos objetivos deste trabalho. Porém, podemos dizer que o caminho para sua implementação está preparado. Todas as informações necessárias para se efetuar a recuperação são tratadas.

Para concluir-se a recuperação seria necessário implementar o processo de atualização do objeto através de seu histórico. Situação que só ocorre após falhas. A situação mais complicada seria o tratamento da situação em que a ação atômica completa a primeira fase do *two phase commit* e o nó cai. Neste caso, para abortar ou finalizar a ação, seria necessário consultar o coordenador da ação (ver Capítulo 2.2) que necessariamente teria esta informação.

Quando temos a transição ativada pelo fracasso do *commit* (Figura 5.10), mantemos em *log* o registro de *Prepare*, com a lista de intenções, da mesma forma que é mostrado na Seção 4.2.1. Porém, como o *log* é circular isto poderá trazer problemas. Para contornar este problema, deixamos como sugestão implementar um novo *log* que acomodaria somente registros de *Prepare* nesta situação. Ou seja, quando o *commit* da ação falhasse, este registro poderia ser removido do *log* gerenciado por *Log Store* e ficar em um outro *log*. Os outros registros de *Prepare* de ações atômicas que são concluídas com sucesso não necessitariam deste transporte, visto que são removidos ao final da ação. Em nosso caso, não prevemos esta situação, visto que seu tratamento é complexo e não afeta a qualidade dos testes em termos de desempenho do repositório, já que é uma situação atípica. Podemos afirmar, portanto, que o desempenho apresentado por *HiPer Store* não se modificará significativamente quando funcionar em conjunto com o mecanismo de recuperação.

Os testes foram feitos nos computadores `pinheiros`, `hans` e `piteco` que tem as respectivas configurações:

	System Model	Main Memory	Virtual Memory	CPUs	OS Name	OS Version
`pinheiros`	SPARC 10	256 MB	1.0 GB	4	SunOS	5.5
`hans`	SPARC 1+	32 MB	517 MB	1	SunOS	5.5
`piteco`	AMD-KS-PR100	16 MB	36 MB	1	Linux	2.0.29

As máquinas `piteco` e `hans` têm disco local, enquanto a máquina `pinheiros` utiliza o *Network File System* (NFS). Os testes na máquina `piteco` foram feitos de forma isolada, ou seja, modo mono-usuário. Nas máquinas `hans` e `pinheiros`, os testes forma feitos em modo multi-usuário, mas com pouca carga extra. Os testes na máquina `piteco` utilizam uma versão anterior a última de *HiPer Store*. Isto porque houve problemas na utilização de bibliotecas *threads* no sistema operacional *Linux*, necessárias à compactação de *log*. Não utilizamos processamento paralelo na máquina `pinheiros`.

Utilizamos como aplicação para testar o desempenho do repositório filas atômicas. São quatro filas, sendo que na primeira são adicionados 10 elementos, na segunda 20, na terceira 30 e na quarta 40 elementos. Em seguida, todos os elementos são removidos. Cada alteração nas filas (adição ou remoção de elementos) é uma ação atômica. Fazemos ainda algumas variações dentro desta aplicação visando demonstrar o comportamento de *HiPer Store* em diversas situações.

- ações atômicas encadeadas: As ações encadeadas não escrevem o registro de *Prepare* (não o removendo depois) nem o de *Commit*. Somente as ações alto nível executam este processo (seção 4.2.1).

- Pré-existência dos objetos: Quando se trata da criação de um objeto, *HiPer Store* cria uma cópia reserva vazia em disco. Este acesso a disco não ocorre quando o objeto já é existente. No repositório *Fragmented Store* este acesso nunca ocorre.

- Compactação de *log*: Mesmo sendo em *background*, a compactação de *log* pode causar perda de desempenho no sistema. Vale lembrar que depois da compactação de *log* temos a remoção das entradas em *lsnTable* dos objetos que não tem mais registros em *log*, liberando espaço em memória e melhorando o acesso à tabela.

- Espaço em memória ocupado: Dependendo do espaço ocupado, temos variações no desempenho. Isto ocorre devido a *swap* e tempo de acesso aos elementos em memória de *HiPer Store*, tais como objetos e entradas na *lsnTable*. Caso a memória esteja cheia, objetos são escritos em disco sincronamente para liberar espaço em memória para novos objetos, causando perda de desempenho.

  Tanto na máquina `hans` como `pinheiros` temos 1000 segmentos de memória compartilhada disponíveis. Na máquina `piteco`, temos 130 segmentos disponíveis. Para mostrar o comportamento de *HiPer Store* diante da ocupação de memória, verificamos nas máquina `hans` e `pinheiros` seu desempenho em 4 situações distintas, variando conforme o número de segmentos ocupados:

  - memória *pouco* ocupada: entre cerca de 70 e 400 segmentos de memória compartilhada utilizados;
  - memória *parcial*mente ocupada: entre cerca de 350 e 700 segmentos de memória compartilhada utilizados;
  - memória *bastante* ocupada: entre cerca de 600 e 1000 segmentos de memória compartilhada utilizados;
  - memória em cheia: inicialmente cerca de 1000 segmentos utilizados.

  Quando trabalhamos com objetos pré-existentes, a taxa de ocupação de memória é dada pelo limite inferior descrito acima. A variação na ocupação de memória deve-se a fatores como a inclusão de novos objetos, compactação de *log* e memória.

Resultados: Os resultados a seguir são uma média seguida do desvio padrão de aproximadamente 20 execuções cada nas situações descritas acima. Executamos as aplicações

(inserção e remoção de elementos nas 4 filas atômicas) utilizando os repositórios *Fragmented Store* (F.S.), que tem o melhor desempenho dos repositórios Arjuna, e o repositório HiPer Store (H.S.). Após a apresentação dos dados, procuramos comentar o que levou a estes resultados.

- Objetos não existentes e ações atômicas encadeadas: As alterações nas filas são divididas em dois grupos: adição e remoção de elementos. A adição de todos os elementos de todas as filas fica dentro de uma ação alto nível, sendo cada adição de elemento uma ação encadeada. A aplicação é análoga para remoção de elementos.

	`pinheiros`			
Fragmented Store	2,635s e 0,213s			
HiPer Store	pouco	parcial	bastante	cheia
	1,763s e 0,243s	2,435s e 0,377s	2,782s e 0,285s	2,785 e 0,328s

	`hans`			
Fragmented Store	3,629s e 0,068s			
HiPer Store	pouco	parcial	bastante	cheia
	4.731s e 0,183s	5,106s e 0,377s	5,614s e 0,314s	5,921s e 0,402s

	`piteco`
Fragmented Store	1,331s
HiPer Store	1,206s

- Objetos pré-existentes e ações atômicas encadeadas:

	`pinheiros`			
Fragmented Store	2,671s e 0,125s			
HiPer Store	pouco	parcial	bastante	cheia
	1,178s e 0,051s	1,220s e 0,995s	1,509s e 0,883s	1,95s e 0,025s

	`hans`			
Fragmented Store	3,567s e 0,074s			
HiPer Store	pouco	parcial	bastante	cheia
	3,990s e 0,199s	4,492s e 0,109s	4.656s e 0,022s	4,661s e 0,125s

	`piteco`
Fragmented Store	1,208s
HiPer Store	1,071s

- Objetos não existentes e ações atômicas simples:

	pinheiros			
Fragmented Store	62,355s e 2,010			
HiPer Store	pouco	parcial	bastante	cheia
	23,957s e 2.303s	25,617s e 2,826s	31,185s e 3,598s	30,477s e 2,914s

	hans			
Fragmented Store	35,963s e 0,442s			
HiPer Store	pouco	parcial	bastante	cheia
	38,836s e 3,980s	49,142s e 3,789s	59,470s e 4,746s	57,988s e 9,786s

	piteco
Fragmented Store	27,931s
HiPer Store	16,242s

- objetos pré-existentes e ações atômicas simples:

	pinheiros			
Fragmented Store	60,592s e 1,036s			
HiPer Store	pouco	parcial	bastante	cheia
	20,895s e 0,986s	23,304s e 2,573s	30,405s e 3,859s	31,085s e 4,522s

	hans			
Fragmented Store	35,608s e 0,555s			
HiPer Store	pouco	parcial	bastante	cheia
	29,867s e 0,123s	38,885s e 3,261s	42,671s e 5,337s	51,316s e 4,062s

	piteco
Fragmented Store	29.712s
HiPer Store	15.688s

A primeira coisa digna de nota é o fato do desempenho ser tão distinto nas diferentes máquinas. Isto se deve ao fato do repositório *Fragmented Store* fazer muito mais uso do disco que o repositório *HiPer Store*. Além disto, podemos dizer que a relação é quase inversa em termos de processamento e uso de memória. Escritas na máquina `pinheiros` tem um alto custo em relação às outras máquinas. Isto ocorre porque as escritas em disco não são locais, passando pelo NFS para chegarem a seu destino. A latência imposta pelo uso de NFS degrada o desempenho das escritas síncronas feitas na máquina `pinheiros`. Podemos notar que tanto a máquina `hans` como a `piteco` têm melhor desempenho em relação às escritas em disco que a máquina `pinheiros`, apesar de terem um processador bem mais lento. Isto faz com que aplicações em que o desempenho depende fundamentalmente de escritas em disco tenham um desempenho fraco na máquina `pinheiros` quando comparado às outras duas. No entanto, a máquina `pinheiros` tem melhor desempenho quando são exigidos poder de processamento e memória em detrimento de escritas síncronas em disco.

Para comprovarmos este fato, verifiquemos o que acontece quando temos ações simples. Neste caso, temos muitas escritas síncronas em disco utilizando o repositório *Fragmented Store*. Perceba que o desempenho na `hans` e na `piteco` é substancialmente melhor que na `pinheiros`. Já quando temos ações encadeadas, o desempenho na `pinheiros` é melhor que na `hans`, pois temos poucas entradas síncronas em disco e muito mais processamento em si. Como o repositório *HiPer Store* minimiza o número de acessos a disco em relação à *Fragmented Store*, fazendo, para isto, mais uso de processamento, aplicações que utilizam *HiPer Store* sempre têm desempenho inferior na máquina `hans`. Sobre a máquina `piteco` podemos considerar que seu desempenho real deverá ser levemente inferior ao mostrado quando utilizamos *HiPer Store*. Isto porque o protótipo testado na máquina `piteco` não dispunha de compactação de *log*. Apesar disto, podemos afirmar que o desempenho de *HiPer Store* continuará sendo superior, pois a influência das compactações de *log* prejudica discretamente o desempenho geral da aplicação, como veremos mais adiante. Portanto, consideramos como pré requisito para o bom desempenho de *HiPer Store* a presença de muita memória e de um processador poderoso em detrimento da velocidade de acesso a disco. Isto é até intuitivo, visto que *HiPer Store* considera como cópias primárias objetos em memória.

Percebemos que a pré-existência de objetos favorece o desempenho comparativo de *Hiper Store* em relação a *Fragmented Store*. Isto se deve ao fato de, não existindo o objeto, *HiPer Store* escrever cópias reservas vazias em disco durante sua criação, fato que não ocorre em *Fragmented Store*. Percebemos também que o desempenho em ações encadeadas é melhor tanto para *Fragmented Store* como para *HiPer Store*. Isto acontece porque ações

encadeadas fazem as atualizações em disco que garantem a atomicidade da ação somente para a ação alto nível.

Podemos notar que, muitas vezes, o desvio padrão encontrados nas execuções de *HiPer Store* é bastante grande. Isto se deve a alguns fatores: quando analisamos objetos não existentes, verificamos que há um aumento de tempo a medida que o número de objetos aumenta em memória. Verificamos que com o aumento de ocupação, passamos a ter *overhead* gerado por *swap*. e perda de desempenho durante o acesso às estruturas existentes em *HiPer Store*, principalmente a tabela *hash lsnTable*. O outro fator é a compactação de *log* que muitas vezes ocorre durante as execuções, principalmente quando temos a aplicação de ações simples, que são mais demoradas. Na aplicação com ações encadeadas, normalmente, a compactação acaba sendo feita depois da finalização da aplicação, em período ocioso. Como vimos no Capítulo anterior, a compactação muitas vezes libera entradas da *lsnTable*, visto que elimina as entradas que não tenham registros em *log*. Com isto, memória é liberada diminuindo *swap* e o tempo de acesso à *lsnTable*. Realmente, existem casos de melhoras de quase 50% após a compactação do *log*. Por exemplo, houve um caso na máquina **hans** que o tempo caiu de 74,960s para 46,860s. Neste caso, tínhamos memória cheia e inserção de objetos, gerando várias escritas em disco. Com a compactação, liberou-se cerca de 120 segmentos de memória que, provavelmente, não estavam sendo utilizados na *lsnTable*, apenas gerando *overhead* em seu acesso. A eliminação destas entradas melhorou substancialmente o desempenho do sistema.

A duração das aplicações em que é efetuada a compactação, em geral, demora cerca de 4s a mais do que as outras. Este tempo pode variar bastante, dependendo do estado do *log*. O que acontece é que, em geral, uma compactação muitas vezes libera registros do *log* que não são considerados na redução do tamanho geral por não estarem na cauda do *log*. Este trabalho feito em uma compactação é aproveitado pela próxima, que não terá que desativar o registro, nem remover a entrada associada da *lsnTable*, caso seja este o caso. Também devemos considerar que a ativação da compactação de *log* não é freqüente, variando de acordo com o tamanho do *log* e com a atividade no repositório. Utilizamos aqui um *log* de aproximadamente 1 MB.

Verificamos que os desempenhos de *HiPer Store* e *Fragmented Store* são diferentes quando analisamos ações simples e encadeadas. O ganho de desempenho de *HiPer Store* é maior em ações simples. Isto se explica pelo fato de ações simples, em *Fragmented Store*, gravarem a lista de intenções em *Action Store* ao final de cada ação e removê-la a seguir. Em *HiPer Store*, estas entradas em disco são eliminadas. A lista de intenções é gravada como registro *Prepare*, juntamente com o registro de atualização, ao final da primeira

fase do *two phase commit*. A remoção da lista de intenções, no caso o registro *Prepare*, é sincronizada em disco ao final da segunda fase do *two phase commit*, juntamente com o registro de *Commit* da ação.

Se analisarmos os resultados da máquina `pinheiros`, podemos dizer que no pior caso comparativo para *HiPer Store*, temos um desempenho parecido, sendo que *HiPer Store* leva 95% do tempo de *Fragmented Store* (ações encadeadas, objetos inexistentes e memória cheia). Já na situação mais favorável, *HiPer Store* leva 34% do tempo de *Fragmented Store* para executar a aplicação (ações simples, memória pouco ocupada e objetos pré-existentes). Na máquina `hans`, na situação mais favorável à *HiPer Store*, temos um pequeno ganho, com *HiPer Store* levando cerca de 84% do tempo de *Fragmented Store*. Já no pior caso comparativo, *Fragmented Store* leva 61% do tempo de *HiPer Store*. Na máquina `piteco`, no pior caso comparativo para *HiPer Store*, temos um desempenho parecido, com *HiPer Store* levando 90% do tempo de *Fragmented Store*. Já na situação mais favorável, *HiPer Store* leva cerca de 53% do tempo de *Fragmented Store* para executar a aplicação.

Estes dados reforçam a tese de necessitarmos de um cenário adequado para termos em *HiPer Store* uma opção mais eficiente em relação ao repositório *Fragmented Store*. Necessariamente, devemos ter espaço disponível em memória, como nas máquina `pinheiros` (256MB) e `piteco` (rodada em modo mono-usuário), além de poder de processamento. Isto em detrimento de velocidade de entrada/saída. A utilização de NFS torna a utilização de *HiPer Store* mais indicada ainda. Adicionalmente, percebemos que *HiPer Store* terá melhor desempenho em aplicações que necessitem fazer acesso a objetos existentes, ao invés de criação de objetos.

Temos que considerar que o ganho de desempenho apenas entre os repositórios é maior que o descrito, pois os tempos colhidos incluem tarefas que nada tem haver com os mecanismos de persistência e recuperação do sistema, tais como controle de concorrência e as operações pertencentes à ação em si. Isto torna a diferença percentual menor do que seria se medíssemos somente o tempo gasto pelos mecanismos de persistência e recuperação de Arjuna. Infelizmente, as modificações que foram efetuadas estão diluídas no processamento de ações, sendo praticamente impossível isolá-las para fazer uma avaliação mais precisa.

De qualquer maneira, estes resultados comprovam que houve um ganho de desempenho muito significativo para *HiPer Store*, variando bastante dependendo da situação aplicação. Isto demonstra que a utilização de mecanismos de *log* e gerência de memória são adequados

para prover funcionalidades de objetos e, principalmente, ações atômicas de forma eficiente. Existem ainda algumas otimizações que poderão melhorar o desempenho de *HiPer Store*. A seguir, veremos o que pode ser feito e as considerações finais desta dissertação.

6.2 Conclusões

Este trabalho integra um novo repositório de objetos a um ambiente de programação distribuída baseado em objetos e ações, otimizando os mecanismos de persistência e recuperação. Este repositório utiliza vários mecanismos presentes em sistemas de outras naturezas, principalmente bancos de dados, como *log* para garantir atomicidade de ações e gerência de memória para a manutenção de cópias primárias de objetos. O resultado é um aumento no desempenho de aplicações desenvolvidas com as ferramentas fornecidas pelo ambiente comparado ao repositório atual, que utiliza como repositório de objetos os arquivos do sistema de arquivos do Unix integrado com mecanismo de cópias *shadow* para garantia de atomicidade da ação.

Várias otimizações existentes em outros sistema também poderão surtir um bom efeito em Arjuna. A utilização de mecanismos de *pre fetch* e *cache* distribuído utilizados em *ObServer2* (Seção 3.1.6), assim como a manutenção de trancas no próprio objeto como em alguns bancos de dados residentes em memória (Seção 3.2), ao invés de manter uma tabela *hash* com este propósito, devem melhorar ainda mais o desempenho de Arjuna. Ainda em bancos de dados residentes em memória, também podemos explorar a idéia de acesso a dados diretamente na cópia primária, ao invés de fazer uma cópia com a qual a ação trabalha.

Em *HiPer Store*, a única entrada síncrona em disco durante o processamento de ações atômicas é a escrita do *log* ao final das fases do *two phase commit*. Para otimizar este ponto da arquitetura, existem pelo menos 2 alternativas: a utilização de memória principal não volátil para parte do *log* (*log tail*), ou a replicação ativa em memória de *log*, garantindo sua estabilidade sem a necessidade de acesso síncrono a disco. A utilização do mecanismo de *log tail* já é utilizada extensivamente em BDRM. Em ambas as técnicas, o *log* vai para disco em *background*, sem perda de desempenho para o sistema.

Além disto, seria interessante a avaliação do efeito da variação do tamanho do *log* no desempenho de Arjuna. Também é necessário concluir o mecanismo de recuperação para que o repositório possa ser realmente utilizado de forma confiável e segura.

Apêndice A

Principais métodos de HiPer Store

Descrevemos o funcionamento em pseudo-código dos principais métodos da classe *HiPerStore*, agrupados por suas classes.

A.0.1 MemoryStore

commit_state: Atualiza *Log Store* com o registro de *commit*, a lsnTable com novos lsn e *Memory Store* com a cópia modificada.

```
Boolean commit_state(Object_state obj)
     logSystem.writeLog(COMMIT)
     lsnTable.writeLsn(objUid, COMMIT)
     updateEntry(obj)
fim
```

endPhase: Sincronizam páginas alteradas de *log* em disco ao final das fases.

```
Boolean endPhase()
     logSystem.log.sync()
fim
```

read_committed: Procura cópia primária primeiramente. Se falha, tenta a reserva.

```
Object_state *read_committed(char *objUid)
     Se objeto = scanStore(objUid,FIRST)
          retorna objeto
     objeto = backup.read_state(objUid)
     retorna objeto
fim
```

remove_committed: Retira registros de *log* (prepare) no final da segunda fase do *two phase commit*.

write_uncommitted: Escreve em log os registros de atualizações e atualiza tabela de lsn com novo lsn.

```
Boolean  write_uncommitted(Object_state *obj)
     lsn = logSystem.log.writeLog(obj,NORMAL)
     lsnTable.writeLsn(lsn,objUid)
fim
```

compactStore: Libera espaço para entrada de tamanho *size* em *Memory Store* e, conseqüentemente, espaço em *log*.

```
Boolean compactStore(int size)
     Retira objetos partindo do final da fila de Memory Store ate'
     liberar espaco ''size'' para nova entrada
     Para cada objeto,
          backup.write_state(obj)
          logSystem.removeFromLog(objUid)
          RemoveFromStore(objeto)
fim
```

removeFromStore: Remove entrada de *Memory Store*, mantendo-o consistente.

enterIntoStore

```
Boolean enterIntoStore(cacheEntry)
     Se falta espaco em Memory Store
          compactStore(tamanho de cacheEntry)
     Insere entrada no inicio de Memory Store (LRU)
fim
```

scanStore: Retorna entrada associada a identificador do objeto.

updateEntry

```
Boolean  updateEntry(Object_state obj)
     velha_imagem = scanStore(objUid)
     RemoveFromStore(velha_imagem)
     enterIntoStore(obj)
fim
```

Observação: Todos os acessos ao repositório *Memory Store* são feitos através de uma árvore AVL.

A.0.2 LogStore

writeLog: Escreve em *log* registros de atualizações, PREPARE e COMMIT e retorna LSN.

```
int  writeLog(Object_state *obj,typeRecord tipo)
     Cria registro de log a partir do objeto e tipo;
     Se falta espaco em log
          compactLog()
      Se registro e' NORMAL
          velha_imagem = memoryStore.read_committed(objUid)
          Se velha_imagem = 0
              velha_imagem = estado vazio
          memoryStore.backup.write_state(velha_imagem)
          Cria registro com diferencas entre nova e velha imagem
          do objeto
     Tranca headLsn
     Copia headLsn, incrementa tamanho do registro e libera headLsn
     Insere registro em log virtual
     Atualiza variaveis de controle de log
fim
```

compactLog: Compacta *log* para pelo menos metade do tamanho e escreve em *Disk Store* os objetos afetados. Função chamada em *background* utilizando *multi threads*. É ativado sempre que tamanho do *log* chega a metade de sua capacidade. Somente um *thread* pode fazer a compactação por vez.

```
Boolean compactLog()
     Tranca tailLsn
     Enquanto (tamanho de log > metade atual)
        Pega ultimo registro de log
        image = memoryStore.scanStore(registro.Uid)
        tranca objeto em Memory Store
        Se objeto foi modificado
           memoryStore.backup.write_state(image)
        removeFromLog(registro)
     libera log
fim
```

removeFromLog: Remove todos os registros do *log* de ações já terminadas associadas a um determinado objeto.

```
Boolean removeFromLog(logRecord removed)
   lsnTable.readLsn(objUid)
   Busca em log lsn do ultimo registro COMMIT do objeto
   Libera objeto em Memory Store
   ''Navega'' em log, a partir do lsn do ultimo COMMIT, tornando registros
   desse objeto inativos e decrementa tamanho atual do log sempre que
   retira registro indicado por tailLsn
fim
```

A.0.3 DiskStore

genPathName, read_state, write_state: Análogo aos métodos correspondentes da classe ShadowingStore, mas não lida com cópias *shadow*.

Referências Bibliográficas

[1] Malcolm Atkinson. The object-oriented database system manifesto. Technical report, Altair Tech. Rep., 1989.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[3] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in Log Structured File Systems. In *USENIX Technical Conference*, January 1995.

[4] Luiz Eduardo Buzato. *Management of Object-Oriented Action-Based Distributed Programs.* PhD thesis, Dept. of Computing Science, University of Newcastle upon Tyne, October 1994.

[5] Michael J. Carey et al. The architecture of the Exodus extensible dbms. In *International Workshop on Object-Oriented Database System*, pages 231–256. A. P. Buchmann, 1991.

[6] Michael J. Carey et al. Shoring up persistent aplications. In *ACM SIGMOD Conference*, pages 383–394, 1994.

[7] H-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the Wisconsin Storage System. *Software-Practice and Experience*, 15(10):943–962, October 1985.

[8] Transarc Corporation. Encina. Product Overview, 1991.

[9] Margaret H. Eich. Foreword main memory databases: Current and future issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):507–508, December 1992.

[10] Jeffrey L. Eppinger and Alfred Z. Spector. A Camelot perspective. *Unix Review*, 1989.

[11] Jim Gray and Franco Putzoli. The 5-minute rule for trading memory for disc accesses and the 10-byte rule for trading memory for cpu time. In *ACM SIGMOD conference*, pages 395–398, May 1987.

[12] Le Gruenwald, Jing Huang, Yuwei Chen, and Woochun Jun. Evaluation of reloading and paging in main memory database systems. *Journal of the Brazilian Computer Society*, 2(3):24–35, April 1996.

[13] Robert B. Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE transaction on computers*, C-35(9):839–843, September 1986.

[14] Won Kim. *Introduction to Object-Oriented Databases*, chapter 15, pages 199–210. MIT Press, 1991.

[15] D. Knuth. *The art of computer programming*, volume 1. Addison-Wesley, 1973.

[16] David E. Langworthy. *ObServer2: Extensible High Performance Support for Persistence.* Phd thesis proposal, Brown University, September 1993.

[17] Tobin J. Lehman, Eugene J.Shekita, and Luis-Felipe Cabrera. An evaluation of Starbust's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, December 1992.

[18] Eliezer Levy and Avi Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, December 1992.

[19] Jun-Lin Lin and Margaret H. Dunham. Dynamic segmented fuzzy checkpointing for main memory databases. *submitted to International Conference on Data Engineering*, 1997.

[20] Hector Garcia Molina and Kenneth Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, December 1992.

[21] O'Reilly and Associates. Código lfs. CD 4.4Lite-2.

[22] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. *The Arjuna System Programmer's Guide.* Department of Computing Science, The University, Newcastle upon Tyne, 1995.

[23] Mendel Rosemblum and John K. Ousterhout. The LFS Storage Manager. *Summer USENIX Tecnical Conference*, 1990.

[24] Michel Ruffin. A survey of logging uses. Technical Report 36, Broadcast technical report, October 1994.

[25] Margo Seltzer. Atualizações do lfs. ftp://virtual.harvard.edu/pub/margo/usenix.195/lfs.tar.gz.

[26] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a Log Structured File System for Unix. *USENIX*, January 1993.

[27] Margo Irene Seltzer. *File System Performance and Transaction Support.* PhD thesis, University of California at Berkeley, 1992.

[28] Santosh K. Shrivastava. Lessons learned from building and using the Arjuna distributed programming system. *Apresentado no 6o SCTF - Simpósio de Computadores Tolerantes a Falhas*, 1995.

[29] Michael Stonebreaker. Object management in Postgres using procedures. In *International Workshop on Object-Oriented Database System*, pages 53–64, September 1986.

[30] Michael Stonebreaker, Lawrence A. Rowe, and Michael Hirohama. The implementation of Postgres. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[31] Nivio Ziviani. *Projeto de algoritmos: com implementações em Pascal e C.* Pioneira Informática, 1993.