UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Validação do Mecanismo de Tolerância a Falhas do SGBD InterBase Através de Injeção de Falhas

por

PAULO RICARDO RODEGHERI

Dissertação submetida à avaliação,
como requisito parcial para a obtenção de grau de Mestre
em Ciência da Computação

Profa. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, março de 2002

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Rodegheri, Paulo Ricardo

Validação do Mecanismo de Tolerância a Falhas do SGBD InterBase através de Injeção de Falhas / por Paulo Ricardo Rodegheri. - Porto Alegre: PPGC da UFRGS, 2002.

100p. : il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientadora: Weber, Taisy Silva.

1. Recuperação em banco de dados. 2. Tolerância a falhas. 3. Injeção de falhas. 4. Detecção de erros. I. Weber, Taisy Silva. II. Título

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora : Profa. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fensterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Inicialmente gostaria de agradecer a profª. Taisy, por ter me acolhido como seu orientando e por ter acreditado neste projeto. Agradeço pela confiança depositada em mim e pelo crédito dado à minha capacidade quando da seleção ao mestrado em 1999. Agradeço pelo exemplo de dedicação, sinceridade e permanente disponibilidade. Pelo incentivo e por estar sempre presente nos momentos de incerteza e angústia. Por ter realizado a tarefa de professora e orientadora de maneira exemplar. Pela forma alegre, carinhosa e dedicada com que trata todos os seus orientandos.

Gostaria também de agradecer à direção da URI - Campus de Erechim, nas pessoas do Prof. Júlio Brondani, da Profª. Helena Confortin e do Bel. Alcides Benincá, pela liberação de horários de trabalho e pelo incentivo. Sem esta contribuição, a conclusão deste mestrado não seria possível.

Igualmente lembro do Neilor, pela compreensão nos momentos difíceis, pelo apoio e pelo incentivo, pela ajuda na montagem dos horários e pelas ausências ao trabalho. Também ao Jacques, pelo incentivo, pelo apoio, pela acolhida e pela companhia no apartamento em Porto Alegre. Ao Fábio, pelo apoio e pela companhia nas longas viagens e pela constante troca de idéias. Ao Adário e a Niceli pelo incentivo e apoio.

Agradeço aos colegas de mestrado Ricardo Manfredini e Luís Cláudio, pelas permanentes trocas, pela camaradagem e pelos bons momentos. A troca de idéias em grupo foi fator fundamental na definição do rumo e conclusão deste trabalho.

Agradeço a UFRGS e ao PPGC pela oportunidade, à todo o corpo docente e aos funcionários pela presteza e dedicação.

Dedico esta dissertação à minha esposa Solange e aos meus filhos Giancarlo e Giovani, pela compreensão, apoio e dedicação. Por entenderem e aceitarem a distância, a ausência e o mau-humor. Dedico também a meu pai e minha mãe pelo exemplo humano e pela minha formação e educação.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas

ASIC	Aplication Specific Integrated Circuit
DBA	Database Administrator
FIDe	Fault Injection via Debugging
GPL	GNU Public License
GUI	Graphic User Interface
IRC	Internet Relay Chat
LGWR	Log Writes
MB	Megabytes
Mhz	Megahertz
OLTP	On Line Transaction Processing
PDA	Personal Digital Assistant
PL/SQL	Program Language / Structured Query Language
RAM	Random Access Memory
SQL	Structured Query Language
SGBD	Sistema Gerenciador de Banco de Dados
SWIFI	Software Implemented Fault Injection
TIP	Transaction Information Page
TPS	Transações Por Segundo

Resumo

O presente trabalho explora a aplicação de técnicas de injeção de falhas, que simulam falhas transientes de hardware, para validar o mecanismo de detecção e de recuperação de erros, medir os tempos de indisponibilidade do banco de dados após a ocorrência de uma falha que tenha provocado um *crash*. Adicionalmente, avalia e valida a ferramenta de injeção de falhas FIDe, utilizada nos experimentos, através de um conjunto significativo de testes de injeção de falhas no ambiente do SGBD.

A plataforma experimental consiste de um computador Intel Pentium 550 MHz com 128 MB RAM, do sistema operacional Linux Conectiva kernel versão 2.2.13. O sistema alvo das injeções de falhas é o SGBD centralizado InterBase versão 4.0. As aplicações para a carga de trabalho foram escritas em *scripts* SQL e executadas dentro de uma sessão chamada *isql*. Para a injeção de falhas foram utilizadas três técnicas distintas: 1) o comando *kill* do sistema operacional; 2) *reset* geral no equipamento; 3) a ferramenta de injeção de falhas FIDe, desenvolvida no grupo de injeção de falhas do PPGC da UFRGS.

Inicialmente são introduzidos e reforçados os conceitos básicos sobre o tema, que serão utilizados no decorrer do trabalho e são necessários para a compreensão deste estudo. Em seguida é apresentada a ferramenta de injeção de falhas Xception e são também analisados alguns experimentos que utilizam ferramentas de injeção de falhas em bancos de dados.

Concluída a revisão bibliográfica é apresentada a ferramenta de injeção de falhas – o FIDe, o modelo de falhas adotado, a forma de abordagem, a plataforma de hardware e software, a metodologia e as técnicas utilizadas, a forma de condução dos experimentos realizados e os resultados obtidos com cada uma das técnicas.

No total foram realizados 3625 testes de injeções de falhas. Com a primeira técnica foram realizadas 350 execuções, com a segunda técnica foram realizadas 75 execuções e com a terceira técnica 3200 execuções, em 80 testes diferentes. O modelo de falhas proposto para este trabalho refere-se a falhas de *crash* baseadas em corrupção de memória e registradores, parada de CPU, aborto de transações ou *reset* geral. Os experimentos foram divididos em três técnicas distintas, visando a maior cobertura possível de erros, e apresentam resultados bastante diferenciados. Os experimentos com o comando *kill* praticamente não afetaram o ambiente do banco de dados. Pequeno número de injeção de falhas com o FIDe afetaram significativamente a dependabilidade do SGBD e os experimentos com a técnica de *reset* geral foram os que mais comprometeram a dependabilidade do SGBD.

Palavras-chave: Recuperação em Bancos de Dados, Tolerância a Falhas, Injeção de Falhas, Detecção de erros.

TITLE: "VALIDATION OF FAULT TOLERANCE MECHANISMS OF INTERBASE DBMS THROUGH FAULT INJECTION"

Abstract

This work aims to explore the application of fault injection techniques, that simulates transient faults of hardware, in order to validate the detection and recovery mechanisms, and to measure the unavailable database times after a fault that has provoked a crash. In addition, it evaluates and validates the fault injection tool called FIDe, employed in the experiments, through a set of a significative fault injection tests at database management system environment.

The experimental platform consists of an Intel Pentium 550 MHz computer with 128 MB RAM, the Conectiva Linux operating system version kernel 2.2.13. The fault injection target system is the centralized DBMS InterBase version 4.0. The programs for workload were written in SQL scripts and were running inside a session called isql.

Initially, the basic concepts about the theme were introduced and reinforced. These concepts will be used at the work and they are necessary to comprise this study. In sequence, a fault injection tool Xception is presented and also some experiments that use fault injection tools in databases are analyzed.

After concluding the bibliographic review, the faults injection tool - FIDe, the adopted failure model, the approach form, the software and hardware platform, the techniques and methodology used, the form of conduction of experiments and the results obtained with each techniques are presented.

In total 3625 fault injection tests were executed. With the first technique 350 executions were performed, with the second technique 75 executions were done and with the third technique 3200 executions were performed. The latter technique was subdivided into 80 distinct tests. The failure model proposed for this work refers to crash faults based in memory and registers corruption, CPU halt, transaction abort or general reset. The experiments were split into three distinct techniques: 1) the kill command of operating system; 2) general reset in the equipment; 3) the fault injection tool called FIDe, developed at fault injection group at PPGC of UFRGS. These techniques aim the greater possible errors coverage and they show distinct results as follow: The experiment with kill command did not have significant affect on database environment. A small number of fault injection with FIDe affected significantly the database dependability and the experiments with the general reset technique seriously affected the DBMS dependability.

Keywords: Database Recovery, Fault Tolerance, Fault Injection, Errors Detection.

1 Introdução

Sistemas computacionais são constituídos de uma série de componentes de hardware, de software e outros, que eventualmente podem falhar. Estas falhas podem levar o sistema a apresentar um defeito, ou seja, o serviço fornecido pelo mesmo não está de acordo com o que foi especificado [LAP 92].

Defeitos em sistemas de controle, monitorados por computadores e que requerem alta disponibilidade, têm sérias conseqüências. Eles ameaçam vidas humanas em aeronaves, no controle de tráfego aéreo, no controle de estações de trem, no suporte médico, no controle de plantas industriais, no controle de plantas de energia nuclear, e em sistemas de defesa. Considere a recente queda da America Online que afetou seis milhões de usuários e a falha de software no lançamento do voô inaugural do Ariane 5 da Agência Espacial Européia [CAR 99].

A algum tempo ficou óbvio que a confiabilidade, a disponibilidade e a segurança destes sistemas não podem somente ser focados em projeto minucioso, em garantia de qualidade, em proteção, ou em outras técnicas que evitem falhas. Os sistemas de computação devem atender aos serviços esperados mesmo na presença de falhas - este é o propósito da tolerância a falhas. Mas antes que um sistema tolerante a falhas seja desenvolvido, é preciso que ele seja testado e validado. Estes sistemas realmente fazem recuperação de falhas? Podem esses sistemas recuperar-se de todos os tipos de falhas ou de apenas algumas? Sempre ou apenas sobre circunstâncias específicas? Eles tem alguns pontos específicos cujos erros levam a um *crash*?

As respostas a estas questões são importantes para construtores e usuários de sistemas tolerantes a falhas. Construtores desejam anunciar altos níveis de tolerância a falhas e usuários finais querem sistemas críticos confiáveis [CAR 99].

Muitas organizações utilizam sistemas gerenciadores de bancos de dados para monitorar e operacionalizar suas diversas atividades. Esta tecnologia é ainda imperfeita. Lapsos de comunicação, erros de sistema, erros de aplicação, queda de energia podem conduzir a transações errôneas gerando grandes prejuízos. Desta forma, quando um sistema falha, procedimentos de recuperação devem agir para restabelecer, validar e devolver o sistema a um estado normal, que existia antes da ocorrência da falha. Ações de prevenção de falhas em SGBDs devem ser rápidas, ou se falhas acontecerem, devem ser solucionadas rapidamente.

Estes esquemas de recuperação permitem a continuidade operacional após a ocorrência de uma falha. O processo de recuperação geralmente é lento e ocasiona a indisponibilidade do sistema aos usuários por intervalos de tempo demasiadamente longos. Apesar de comuns em SGBDs, poucos trabalhos na literatura enfocam a avaliação e validação de técnicas de recuperação nesses sistemas. Além disso, o aumento da tendência do uso de SGBDs em missões críticas e sistemas comerciais críticos levam a um maior interesse por avaliação da confiabilidade e disponibilidade de tais sistemas [COS 99].

SGBDs comerciais tradicionais (Oracle, Sybase, SQL Server, Oracle, Ingress) utilizam mecanismos de *log* e de *checkpointing* para obter um estado consistente após a detecção de uma falha. Estes mecanismos, geralmente, geram uma sobrecarga de processamento, são potencialmente lentos e requerem a intervenção do DBA [BOR 99].

A validação das propriedades de confiabilidade dos sistemas de computadores é intrinsecamente complexa e a crescente complexidade destes tende a dificultá-la. O uso de modelos analíticos nos sistema atuais é muito difícil porque os mecanismos envolvidos nas ativações de falhas bem como os processos de propagação dos erros são muito complexos e não completamente entendidos na maioria dos casos [COS 99]. Além disso, a verificação experimental através do monitoramento do sistema até que uma falha real ocorra, na maioria dos casos, é impraticável.

A avaliação experimental através da injeção de falhas tem-se mostrado uma forma atrativa para a validação específica dos mecanismos de tratamento de falhas, permitindo estimar medidas de tolerância a falhas, como cobertura de falhas e latência de erros [ARL 90].

O uso de técnicas que permitam evitar e/ou tolerar falhas por si só não é suficiente para garantir a confiança no serviço fornecido pelo sistema. Os métodos para a construção de sistemas não estão livres de falhas; portanto é necessário o uso de técnicas que permitam eliminar ao máximo as falhas residuais existentes no projeto e/ou implementação de sistemas. Estas técnicas, por sua vez, também são imperfeitas; não é possível ainda eliminar de todo a possibilidade de ocorrência de falhas. É importante se fazer uma previsão do efeito das mesmas sobre o sistema. Estes dois aspectos dizem respeito à validação de um sistema, o que incluiria então [LAP 92] :

- a eliminação de falhas, que envolve a verificação, o diagnóstico e a correção;
- a previsão de falhas, que visa obter, por avaliação, medidas que permitam caracterizar o comportamento do sistema em presença de falhas.

Um ponto crucial na validação de sistemas tolerantes a falhas diz respeito à validação dos seus mecanismos de tolerância a falhas. A importância da validação destes mecanismos se deve a duas razões principais [MAR 96] :

- a presença de falhas de projeto/implementação nesses mecanismos pode levar a deficiências de comportamento dos mesmos quando em presença das falhas para as quais eles foram projetados para tratar; essas deficiências podem levar o sistema a não mais fornecer o serviço correto;
- o efeito da eficiência dos mecanismos de tolerância a falhas sobre medidas tais como a confiabilidade do sistema.

1.1 Objetivos

O presente trabalho explora a aplicação de técnicas de injeção de falhas para validar os

mecanismo de detecção e de recuperação de falhas, e dos tempos de indisponibilidade do SGBD InterBase aos usuários após a ocorrência de uma falha que tenha provocado *crash*.

Objetiva, também, a validação da ferramenta de injeção de falhas utilizada - o FIDe - visando realimentar o processo de desenvolvimento e o aprimoramento desta ferramenta através dos resultados obtidos e das dificuldades encontradas na sua utilização na condução dos experimentos.

1.2 Motivação

A maioria dos SGBDs comercialmente disponíveis possuem suporte para recuperação de dados e tolerância a falhas, mesmo quando a plataforma de hardware base não tenha qualquer característica de tolerância a falhas. Entretanto, poucos trabalhos na literatura enfocam a avaliação/validação destas técnicas de tolerância a falhas [COS 99] [MAD 99] [MAN 01] [SAB 99].

O aumento da tendência do uso de SGBDs em missões críticas e sistemas comerciais críticos levam a um maior interesse por avaliação da confiabilidade e disponibilidade de tais sistemas [COS 99].

Além disto, o SGBD InterBase é utilizado amplamente em ambiente corporativo com plataforma cliente/servidor. A grande popularidade dos usuários do InterBase, tanto em nível nacional quanto mundial, deve-se ao fato de que grande número de aplicações escritas em Delphi utilizam o InterBase como gerenciador de banco de dados.

1.3 A Plataforma Experimental

A plataforma experimental consiste de um computador Intel Pentiun 550 MHz com 128 MB RAM, do sistema operacional Linux Conectiva 4.2 Kernel versão 2.2.13 e o SGBD centralizado InterBase versão 4.0. Para a injeção de falhas é utilizada a ferramenta FIDe, desenvolvida dentro de grupo de injeção de falhas do PPGC da UFRGS.

O sistema operacional GNU/LINUX foi escolhido como plataforma principalmente em função de suas características de software livre. A disponibilidade do código fonte, a vasta gama de listas de discussão nos mais variados tópicos, a documentação acessível, o ambiente de cooperação de canais de IRC e os próprios méritos do sistema: robustez, velocidade, e o fato de atender os requisitos POSIX. Além disso, o Linux tornou-se uma importante opção como sistema operacional para servidor de banco de dados corporativo. Optou-se, também pelo Linux, porque a ferramenta de injeção de falhas faz uso dos recursos do depurador deste sistema operacional.

Ultimamente, várias empresas perceberam a importância de lançar versões de seus produtos para o Linux, alguns em versão *freeware.* Alguns SGBDs disponíveis: InterBase, Oracle, Sybase, Informix, Zim, Adabas D, IBM DB2, PostgresSQL, Progress, Mysql [LIN 00].

1.4 O Sistema de Banco de Dados Alvo

O InterBase é um SGBD relacional que possui diversas características que o diferenciam consideravelmente de seus concorrentes. A grande maioria dos SGBDs comerciais tradicionais (Sybase, SQL Server, Oracle, Progress) utilizam mecanismos de *log* e de *checkpointing* para obter um estado consistente após a detecção de uma falha. Estes mecanismos geralmente geram uma sobrecarga de processamento e são potencialmente lentos. O InterBase não usa o conceito de *log* de transações e de *checkpointing*. Ao invés disto, mantém informações em TIPs (Transaction Information Pages) [BOR 2000].

No caso de uma falha de sistema, tão logo o servidor é posto *on-line*, o InterBase automaticamente busca nas TIPs por transações *uncommitted*. Qualquer registro encontrado em um estado *uncommitted* é desfeito e o sistema é imediatamente disponibilizado. Segundo o fornecedor, no InterBase restaurações automáticas após uma falha de *crash* levam tipicamente menos de um segundo, e não necessitam da intervenção do administrador como na maioria dos bancos de dados. Além disto, o processo de recuperação no InterBase é cooperativo e gradual, e não preemptivo e imediato como na maioria dos SGBDs, e apenas executa o algoritmo Undo [BOR 2000].

1.5 Metodologia Utilizada

A metodologia por mim utilizada neste trabalho consiste em injetar falhas, deliberadamente, no ambiente do SGBD, no momento que um processo realiza atualizações nos dados do banco de dados. As falhas são injetadas através de três técnicas distintas. A primeira técnica consiste em eliminar, através do comando *kill*, do sistema operacional, um processo pai que contém processos filhos, que realizam atualizações (inserções, alterações e deleções) em dados de tabelas do banco de dados. A segunda técnica consiste no *reset* geral do equipamento, no momento que um processo está sendo executado e realiza atualizações no banco de dados. A terceira técnica faz uso de uma ferramenta de injeção de falhas - o FIDe, para injetar falhas que simulem situações que podem ocorrer no ambiente do SGBD (alterar o conteúdo dos registradores, dos dados em memória, da área de código e as informações (estrutura *user*) do processo em depuração) e que podem afetar a consistência e integridade dos dados.

Optou-se, neste trabalho, por três enfoques distintos, com o objetivo de gerar a maior quantidade possível de ocorrências de falhas, reais através da primeira e segunda técnica e simuladas através da terceira técnica, que pudessem afetar o ambiente do banco de dados. Estes falhas possibilitam validar o comportamento e a eficiência dos mecanismos de detecção e recuperação de falhas e os tempos de indisponibilidade após falhas.

O InterBase possibilita a execução de escritas *bufferizadas*, também chamadas de escritas assíncronas, e de escritas diretamente em disco (*forced writes*). Se ocorrer uma falha de sistema antes da escrita de dados em disco, então informações podem ser

perdidas. Executando-se escritas forçadas garante-se integridade dos dados e segurança, mas perde-se desempenho, isto é, operações que envolvem modificações de dados serão mais lentas [BOR 2000]. Desta forma, os experimentos com cada uma das técnicas utilizadas foram realizados com escritas *bufferizadas* e com escritas forçadas em disco.

A verificação de que se houve ou não interferência na consistência e na integridade dos dados é feita através de nova tentativa de leitura e atualização dos dados das tabelas que foram alvo da injeção de falhas. Se a nova operação de leitura e/ou atualização foi concluída de forma normal, isto é, sem a indicação de erro, conclui-se que o mecanismo de recuperação obteve sucesso, ou recuperando efetivamente (desfazendo as transações que não obtiveram *commit*) ou mascarando os erros encontrados. Se a operação não foi concluída de forma normal, o mecanismo de detecção de erros exibe uma mensagem correspondente, identificando e informado o erro.

Outra forma de fazer-se a verificação da ocorrência de erros no banco de dados é através do arquivo **interbase.log,** que registra com detalhes as anormalidades ocorridas com o ambiente do banco de dados.

Os tempos de indisponibilidade foram medidos da seguinte maneira: fazia-se o processo de injeção de falhas (que podia ser concluído de forma normal ou anormal), abria-se uma sessão *isql* e fazia-se uma nova conexão com o banco de dados que, possivelmente, havia sido afetado anteriormente pela falha. Neste ponto cronometra-se manualmente os tempos de indisponibilidade (tempo de espera até que se pudesse utilizar novamente os dados das tabelas do banco de dados).

1.6 Resultados Alcançados

Os mecanismos de detecção e de recuperação de erros do SGBD mostraram-se bastante confiáveis quando foi utilizada a primeira técnica, que consiste em eliminar, através do comando *kill* um processo pai que contém um processo filho que estabelece atualizações (inserções, alterações e deleções) no banco de dados. Dos 350 testes realizados, em apenas um os mecanismos de recuperação e de detecção não foram eficientes, o que representa 0,29%, sendo portanto eficiente em 99,71 % dos casos.

A segunda técnica consiste no *reset* geral do equipamento, quando um processo realiza atualizações no banco de dados. No total foram executados 75 testes. Neste experimento os testes ocorreram em menor número porque consumiam quantidade de tempo muito maior e porque poderiam comprometer a integridade do ambiente de experimentação (equipamento, sistema operacional, banco de dados, procedimentos e aplicações). Dos 75 testes, 35 foram executados **sem** *forced writes* e os outros 40 **com** *forced writes*. **Sem** *forced writes*, o mecanismo de recuperação obteve sucesso em 17 vezes (48,57%). Nos outros 18 testes (51,43%) o mecanismo não obteve sucesso. Dos 40 testes realizados **com** *forced writes*, o mecanismo de recuperação obteve sucesso em 25 vezes (62,5%). Nos outros 15 testes (37,5%), o mecanismo não obteve sucesso. Resumindo, dos 75 testes que utilizaram *reset* geral, em 42 houve sucesso no processo de recuperação (56%). Nos demais 33 casos (44%), o mecanismo de recuperação não obteve sucesso. Em 100% dos casos, o mecanismo de detecção de erros obteve sucesso.

Utilizando-se a injeção de falhas através do FIDe foram realizados 3200 testes, **com** e **sem** o parâmetro *forced writes.* Do total de testes, em 58,75% a injeção de falhas não afetou o banco de dados, isto é, nenhuma anormalidade ocorreu ou a falha foi adequadamente mascarada. Em 36,25% a injeção de falhas afetou parcialmente, ou demorando para fazer a conexão com o banco de dados ou não concluindo esta conexão. Em 5,00% dos testes a injeção de falhas afetou seriamente a base de dados e as estruturas de dados, danificando permanentemente a base de dados.

O presente trabalho gerou, até o momento, os seguintes artigos: [ROD 01] [ROD 01A] e [ROD 00]. O artigo [ROD 00] também está disponível na página do Interbase-BR (http://www.warmboot.com.br/ib/artigos/crash.recovery.html) e como artigo na versão eletrônica da Revista do Linux (http://www.RevistaDoLinux.com.br/artigos).

1.7 Organização do Texto

Este trabalho aborda o uso da técnica de injeção de falhas visando validar o mecanismo de tolerância a falhas do SGBD centralizado InterBase, especificamente: detecção e recuperação de erros e indisponibilidade após a ocorrência de erros. São relatados os resultados de experimentos de injeção de falhas, através de duas técnicas manuais próprias e através de uma ferramenta de software - o FIDe (Fault Injection via Debugging).

O capítulo 2 enfoca e descreve as falhas e os processos de recuperação após falhas em bancos de dados, os tipos de armazenamento utilizados em bancos de dados, o movimento de dados entre blocos de disco e de memória *cache,* transações e suas características. Aborda também o uso de arquivos de *log,* os tipos de atualizações utilizadas em bancos de dados (imediata e adiada), e os métodos de otimização ou alternativos ao uso de *log* (*checkpointing* e *shadow paging*). O capítulo 3 descreve a técnica de injeção de falhas, as suas formas de aplicação e faz um relato e a análise de alguns experimentos de injeção de falhas em sistemas gerenciadores de bancos de dados. O capítulo 4 detalha a ferramenta de injeção de falhas – FIDe – sua forma de funcionamento e a arquitetura da ferramenta. O capítulo 5 relata o ambiente dos experimentos, os componentes da plataforma de trabalho, a metodologia utilizada e as técnicas utilizadas em cada um dos experimentos, a carga de trabalho e o modelo de falhas. O capítulo 6 aborda os experimentos realizados e os resultados obtidos, acompanhados de seus correspondentes gráficos e tabelas. No capítulo 7 são apresentadas as conclusões e as sugestões para trabalhos futuros.

2 Recuperação em Sistemas de Banco de Dados

Este capítulo apresenta uma visão genérica sobre falhas, os tipos de armazenamento utilizados por SGBDs, o conceito de transação, os principais mecanismos de recuperação e os algoritmos utilizados. Este estudo visava determinar se as estratégias de injeção de falhas seriam adequadas para os mecanismos de recuperação usados no SGBD. Uma vez que o conjunto de falhas que podem afetar um banco de dados centralizado é conhecido, os conceitos e a compreensão do processo de detecção e recuperação serviram como base para a definição do modelo de falhas proposto. O leitor familiarizado com os conceitos básicos da área não encontrará neste capítulo nenhuma nova contribuição ao tema. Neste caso, sugere-se a leitura direta do capítulo 3.

As principais referências bibliográficas deste capítulo são *Concurrency Control and Recovery in Database Systems* de P. Berstein, V. Hadzilacos e N. Goodman [BER 87], *Fundamentals of Database Systems* de R. Elmasri e S.B. Navathe [ELM 89], *Transaction Processing: Concepts and Techniques* de J. Gray e A. Reuter [GRA 93], *Sistema de Banco de Dados* de H.F. Korth e A. Silberschatz [KOR 95], *Fault-Tolerant Computer System Design* de D.K. Pradhan [PRA 96].

2.1 Falhas em Bancos de Dados

Existem vários tipos de falhas que podem ocorrer em um sistema, cada uma das quais necessita ser tratada de uma maneira diferente. O tipo mais simples de falha a tratar é aquela que não resulta na perda de informações no sistema. As falhas mais difíceis são aquelas que causam perda de informação. Para determinar como o sistema deve recuperar-se das falhas, é necessário identificar as falhas. Em seguida, é preciso considerar como estas falhas afetam o conteúdo do banco de dados. Pode-se então propor algoritmos para assegurar a consistência do banco de dados e a atomicidade da transação apesar das falhas. Estes algoritmos tem duas partes [KOR 95] :

- Ações tomadas durante o processamento normal da transação com o objetivo de assegurar que existam informações suficientes para permitir a recuperação das falhas.

- Ações tomadas em seguida as falhas para assegurar a consistência do banco de dados e a atomicidade da transação.

2.2 Tipos de Armazenamento

Existem vários tipos de armazenamento que são distinguidos por sua velocidade relativa, capacidade e resistência a falhas [KOR 95].

- **Armazenamento Volátil** : As informações residentes no armazenamento volátil não sobrevivem a quedas do sistema. Exemplos de tal armazenamento são a

memória principal e a *cache.* O acesso ao armazenamento volátil é muito rápido devido a velocidade de acesso da memória e porque é possível fazer acesso a qualquer item de dado diretamente.

- **Armazenamento Não Volátil** : As informações residentes neste tipo normalmente sobrevivem a quedas de sistema. Exemplos de tal armazenamento são discos e fitas magnéticas. O disco é utilizado para armazenamento *on-line,* enquanto as fitas são usadas para armazenamento histórico ou de reserva. Os discos são mais confiáveis do que a memória principal mas menos confiáveis do que as fitas magnéticas. Ambos, no entanto, estão sujeitos a falhas, que podem resultar na perda de informações. No estado atual da tecnologia, o armazenamento não volátil é mais lento do que o armazenamento volátil em diversas ordens de magnitude. Em sistemas de bancos de dados, os discos são usados para a maioria do armazenamento não volátil.

- **Armazenamento Estável** : As informações residentes de forma estável nunca são perdidas (nunca deve ser entendido com restrições, uma vez que, teoricamente, isto não pode ser garantido). Para implementar-se uma aproximação de tal armazenamento, normalmente necessita-se da duplicação de informações em diversos meios não voláteis de armazenamento (normalmente discos), com modos independentes de falhas, e atualizar estas informações de maneira controlada.

O sistema de banco de dados reside na memória não volátil (normalmente um disco). O banco de dados é dividido em unidades de armazenamento de tamanho fixo chamadas *blocos*, que são as unidades tanto de alocação de armazenamento quanto de transações de dados. As transações transferem informações do disco para a memória principal e depois as devolvem para o disco. As operações de entrada e saída são feitas em unidades de blocos. Os blocos residentes no disco são chamados de blocos físicos, enquanto os blocos residentes temporariamente na memória principal são chamados blocos de *buffer* (ver figura 2.1).

Um bloco pode conter diversos itens de dados. O conjunto exato de itens que um bloco contém é determinado pela forma da organização física dos dados que estiver sendo usada.

Os movimentos de blocos entre o disco e a memória principal são iniciados pelas duas operações seguintes :

- *Input*(X) : que transfere para a memória o bloco físico no qual reside um item de dado X.

- *Output*(X) : que transfere para o disco o bloco de *buffer* no qual reside X e substitui o bloco físico apropriado.

As transações interagem com o sistema de banco de dados transferindo dados de variáveis de programa para o banco de dados e, do banco de dados para as variáveis de programa. Esta transferência de dados é realizada usando duas operações :

- *read*(X,xi), que atribui o valor do item de dado **X** para a variável local **xi.** Esta operação é executada da seguinte forma :

  1) Se o bloco no qual X reside não está na memória principal, então emite *input*(X).
  2) Atribui para xi, o valor de X do bloco de *buffer*.

- *write*(X,xi), que atribui o valor da variável local **xi** para o item de dado **X** no bloco de *buffer***.** Esta operação é executada da seguinte forma :

  1) Se o bloco no qual X reside não está na memória principal, então emite *input*(X).

  2) Atribui o valor de **xi** para **X** no bloco de *buffer* para **X.**

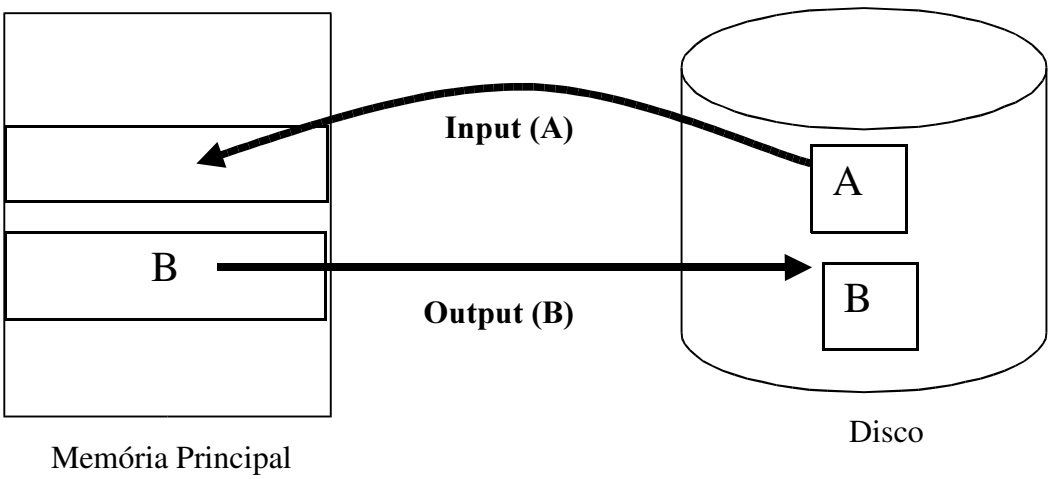

FIGURA 2.1 - Operações de Armazenamento de Blocos [KOR 95].

Ambas as operações podem requerer a transferência de um bloco do disco para a memória principal. Entretanto, elas não requerem a transferência imediata de um bloco da memória principal para o disco. Um bloco de *buffer* é gravado para o disco porque o gerenciador de *buffer* precisa de espaço na memória para outros blocos ou porque o sistema de banco de dados deseja refletir a mudança de X no disco. Diz-se que o sistema de banco de dados "força a saída" do bloco de *buffer* se ele emite um *output*(X).

Quando uma transação precisa fazer o acesso a um item de dado X pela primeira vez, ela deve executar read(X,xi). Todas as atualizações de X são então executadas em **xi.** Depois que a transação fizer o acesso a X pela ultima vez, ela precisa executar *write*(X,xi) a fim de refletir a mudança de X no próprio banco de dados.

A operação *output*(X) não precisa ter efeito imediatamente depois que *write* (x,XI) é executado, uma vez que o bloco no qual X reside pode conter outros itens de dados aos

quais ainda se está fazendo acesso. Assim, a saída propriamente dita ocorre mais tarde. Deve-se notar que, se o sistema cair depois que uma operação *write*(X,xi) for executada, mas antes que *output*(X) tenha sido executado, o novo valor de X nunca será escrito no disco e, portanto, estará perdido [KOR 95].

As operações primitivas sobre o disco são executadas pelo sistema operacional e estas primitivas são invocadas por *syscalls* a partir do SGBD.

2.3 Modelo de transação

Uma transação é uma unidade de programa que faz o acesso e possivelmente atualiza vários itens de dados. Cada um destes itens é lido uma única vez pela transação e é gravado no máximo uma vez se ela atualizar aquele item de dado. Requer-se que as transações não violem qualquer restrição de consistência do banco de dados. Isto é, se era consistente quando uma transação iniciou, o banco de dados precisa estar consistente quando a transação terminar com sucesso. Entretanto, durante a execução de uma transação, pode ser necessário, temporariamente, permitir a inconsistência. Essa inconsistência temporária, ainda que necessária, pode levar a dificuldades se ocorrer uma falha [KOR 95].

Existem diversas propriedades que transações atômicas devem possuir. Estas propriedades devem ser forçadas pelos métodos de controle de concorrência e recuperação dos SGBDs [ELM 89]:

- Atomicidade: uma transação é uma unidade atômica de processamento, isto é, ou é executada em sua totalidade ou não deve ser executada no todo.

- Consistência: a execução correta de uma transação deve levar o banco de dados de um estado consistente a outro estado consistente.

- Durabilidade (ou persistência): uma vez que a transação alterou o banco de dados e as alterações obtiveram *commit*, estas alterações nunca devem ser perdidas devido a defeitos subsequentes.

- Isolamento: uma transação não deve tornar suas alterações visíveis para outras transações até que tenha obtido *commit.*

Uma transação pode ser finalizada de duas maneiras: com sucesso (*commited*) ou cancelada (*aborted*). Quando uma transação é finalizada com sucesso, todas as mudanças feitas por ela tornam-se estáveis no banco de dados. Quando uma transação é cancelada, todas as mudanças feitas por ela devem ser desfeitas (*rollback*) [ELM 89].

Uma transação precisa estar em um dos seguintes estados (figura 2.2):

- Ativo : É o estado inicial.

- Parcialmente Concluído : Depois que a última instrução foi executada.

- Falho : Depois da descoberta de que uma execução normal não pode mais prosseguir.

- Abortado : Depois que a transação foi desfeita e o banco de dados foi restaurado ao seu estado anterior ao início da transação.

- Concluído : Depois da transação ser completada "com sucesso".

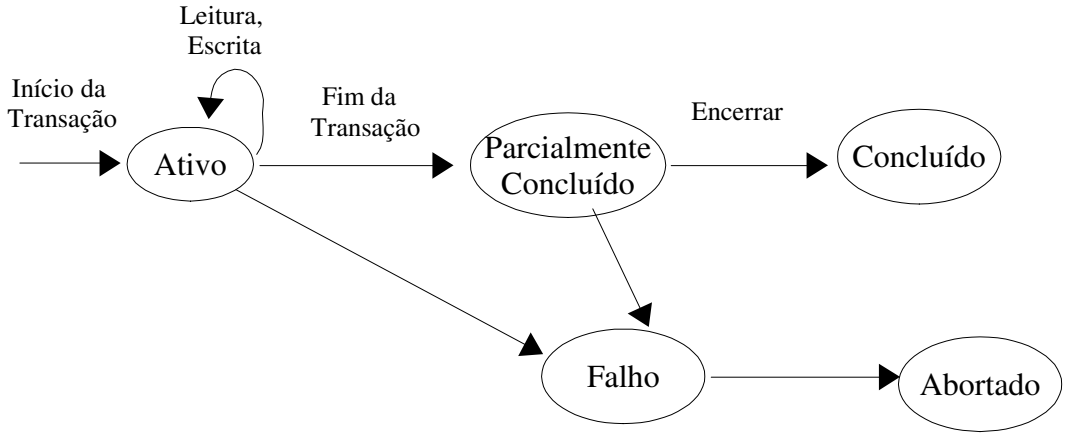

FIGURA 2.2 – Transição de Estados para Execução de Transações [ELM 89].

A execução de uma transação é correta se, tomando um banco de dados consistente, T deixa o banco de dados no próximo estado consistente (ver figura 2.3). Suponha que o estado inicial consistente do banco de dados seja S1. A transação T altera o estado consistente de S1 para S2. Durante esta transação o banco de dados pode ir a um estado inconsistente. Este estado inconsistente, entretanto, não é visível para as outras transações concorrentes com T. Se o sistema ou a transação T falhar então o banco de dados não estará apto a obter S2, consequentemente o estado visível será um estado inconsistente. Deste estado pode-se ou avançar para o estado S2 ou voltar para o estado S1.

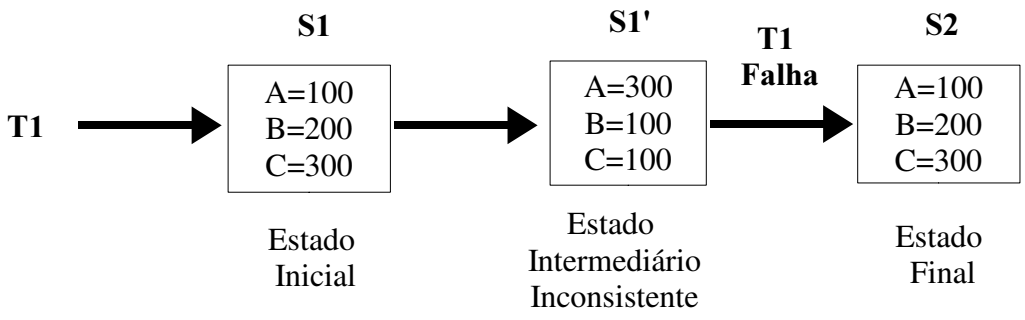

FIGURA 2.3 - Estados de uma transação [KUM 99].

Deve-se então desenvolver mecanismos que levem de um estado inconsistente (S1`) a um estado consistente – S1 ou S2 (figura 2.4).

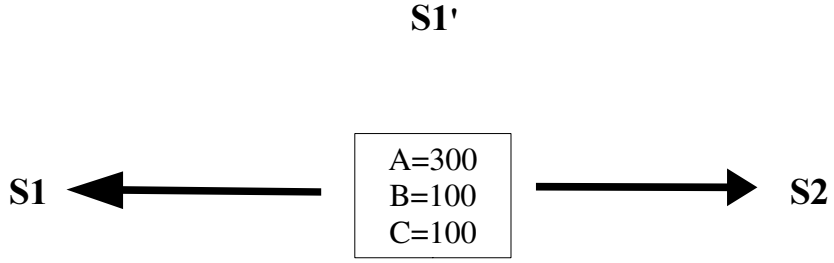

FIGURA 2.4 - O Processo de Recuperação [KUM 99].

Estes algoritmos são chamados algoritmos de recuperação de banco de dados. Antes de desenvolver algoritmos de recuperação deve-se atentar para o processo de recuperação. Considere a seguinte situação (figura 2.5):

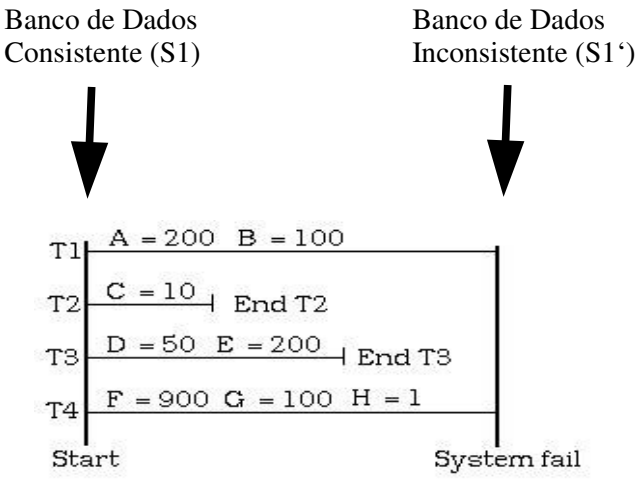

FIGURA 2.5 - Falhas na execução de transações concorrentes [KUM 99].

Se deseja-se voltar ao estado consistente *S1* perde-se as atualizações de T2 e T3. Se deseja-se obter um estado consistente *S2* então deve-se executar um *roll-back* em T1 e T4 e reexecutar estas transações novamente. Uma boa idéia, neste caso, é executar ambas.

Deve-se executar *rollback* nas transações T1 e T4 para restaurar o último valor consistente de A,B,F,G, e H. Um *roll-back* requererá o último valor consistente de A,B,F,G, e H. Estes valores serão encontrados em um local chamado *log* de transações.

2.4 Recuperação em Bancos de Dados

Sistemas de banco de dados empregam ações atômicas, conhecidas como transações, para manter consistência e integridade em atividades concorrentes. Uma vez que

transações são atividades atômicas, na eventualidade de uma transação ser abortada, suas ações devem ser desfeitas para restaurar a consistência ao sistema. O problema é a velocidade das recuperação, que normalmente é muito lenta devido as operações de entrada/saída de disco, necessárias para desfazer ações já feitas em memória. Devido a propriedade tudo-ou-nada das ações atômicas, uma importante quantidade de trabalho deve ser abandonada necessariamente quando um erro interno é encontrado [PRA 96].

As técnicas de *shadowing* e *logging* são duas típicas implementações de mecanismos orientados a recuperação em sistemas de bancos de dados [PRA 96].

2.5 Recuperação baseada em Log

A estrutura mais largamente usada para registrar modificações no banco de dados é o *log* . Cada registro de *log* descreve uma única gravação no banco de dados e tem os seguintes campos [KOR 95]

- Nome da Transação : O nome único da transação que executou a operação *write.*
- Nome do item de dado : O nome do item de dado gravado.
- Valor Antigo : O valor do item de dado antes da gravação (*Before Image*).
- Novo Valor : O valor que o item de dado terá após a gravação (*After Image*).

Outros registros especiais de *log* existem para gravar eventos significativos durante o processamento de transações, tais como o início de uma transação ou a conclusão ou aborto da mesma. Exemplo dos vários tipos de registros de *log*:

- <Ti **start**> = A transação Ti iniciou

- <Ti, Xj, V1, V2> = A transação Ti executou uma gravação num item de dado Xj. Xj tem o valor V1 antes da gravação e terá o valor V2 depois.

- <Ti **commit**> = A transação Ti foi concluída com sucesso.

Toda vez que uma transação executa uma gravação, é essencial que o registro de *log* para essa gravação seja criado antes que o banco de dados seja modificado. Uma vez que um registro de *log* exista, pode-se gravar a modificação do banco de dados se isto for desejável. Tem-se também a possibilidade de desfazer uma modificação que já tenha sido gravada no banco de dados. Isto pode ser feito com o valor antigo (*before image*) de *log*.

A fim de que os registros de *log* sejam úteis para recuperar falhas de disco e de sistema, o *log* precisa residir num dispositivo de armazenamento não volátil. Preliminarmente, assume-se que todo registro de *log* é gravado em meio não volátil tão logo seja criado. Mais adiante, ver-se-á quando é seguro relaxar este requisito para minimizar a sobrecarga imposta pela gravação do *log*. Existem duas técnicas para usar o *log* com o objetivo de assegurar a atomicidade das transações apesar das falhas (modificação imediata e adiada do banco de dados). É importante observar que o *log* físico contém

um registro histórico completo de todas as atividades do banco de dados [KOR 95].

Cada operação de inserção, deleção ou atualização para um objeto transacional gera uma inserção no *log*. Isto sinifica que o *log* pode facilmente se transformar em um problema de desempenho, e também significa que o *log* pode ficar muito grande.

É interessante observar que sendo um componente do sistema, o *log* também está sujeito ao mesmo modelo de falhas do resto do sistema e pode ser um dos fatores de redução de disponibilidade.

Existem duas regras que são aplicadas no processo de recuperação [BER87]:

- **REGRA UNDO** : Se o endereço de X no banco de dados não volátil atualmente contém o último valor *committed* de X, então tal valor deve ser salvo no banco de dados não volátil antes de ser sobrescrito no banco de dados não volátil por um valor *uncommitted* (figura 2.6).

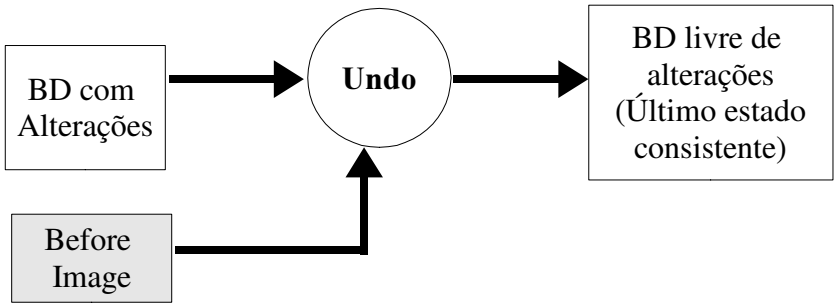

FIGURA 2.6 - O processo Undo [HUN 99].

REGRA REDO : Antes que uma transação possa ser *committed*, o valor por ele escrito para cada item de dado deve estar no armazenamento não volátil (no banco de dados ou no *log*) (figura 2.7).

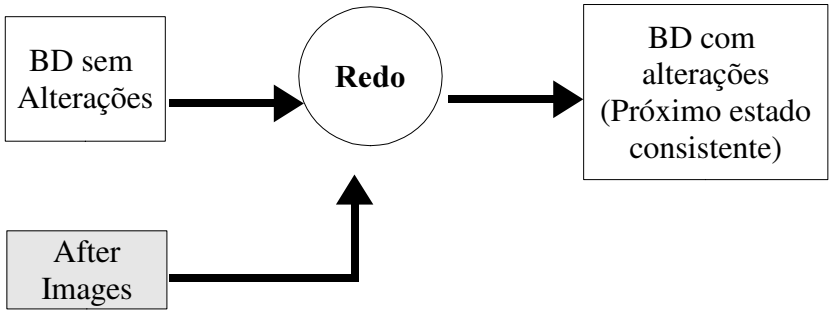

FIGURA 2.7 - O processo Redo [HUN 99].

2.6 Técnicas de Atualização

Uma transação pode estabelecer suas atualizações para o banco de dados através de duas

maneiras distintas [KOR 95]:

- Atualização adiada (Deferred update) : Esta modalidade de atualização grava todas as modificações de itens de dados no *log*, mas adia a execução de todas as operações *write* de uma transação até que a transação esteja parcialmente compromissada (após última instrução ser executada). Após isto grava um registro <T*i commit*> no *log* e atualiza o banco de dados. Caso ocorra uma falha, o mecanismo de recuperação requer apenas Redo. Com isto, pode-se simplificar a estrutura do *log*, pois a técnica *Redo* exige apenas o novo valor (*After Image)* do item de dado. *Redo* precisa ser idempotente*,* isto é, executá-lo diversas vezes será equivalente a executá-lo apenas uma vez.

- Atualização Imediata (Update-in-Place) : A técnica de atualização imediata permite a gravação de modificações no banco de dados enquanto a transação ainda está no estado ativo. As modificações gravadas por transações ativas são chamadas *uncommitted.* Logo que uma transação atualiza itens de dados, ela grava os registros *log* correspondentes, atualiza o banco de dados não volátil e grava um registro <*Ti* ***commits***> no *log*. Isto significa que se uma transação T modifica dois itens de dados ele então escreverá duas vezes no arquivo de *log*. Isto é muito custoso em termos de tempo devido ao I/O de disco. Caso ocorra uma falha, para a recuperação, ambas as técnicas *Undo* e *Redo* podem ser necessárias. *Undo* utiliza o valor antigo *(Before Image)* dos registros de *log* para restaurar os itens de dados modificados. As operações *Undo e Redo* devem ser idempotentes a fim de garantir o comportamento correto. A operação *Undo* deverá ser executada antes da operação *Redo.*

2.7 Checkpointing

Quando ocorre uma falha no sistema é necessário consultar o *log* para determinar quais transações precisam ser refeitas e quais precisam ser desfeitas. Em princípio, o *log* inteiro precisa ser varrido. Existem duas dificuldades principais nesta abordagem [KOR 95] :

- O processo de busca consome tempo.

- A maioria das transações que, de acordo com o algoritmo, precisam ser refeitas já gravaram suas atualizações no banco de dados. Embora refazê-las não cause nenhum mal, isto, todavia, atrasará a recuperação.

Para reduzir este tipo de sobrecarga, usa-se a técnica de *checkpointing.* Durante a execução, o sistema mantém o *log* usando uma das duas técnicas de atualização descritas anteriormente (adiada ou imediata). Adicionalmente, o sistema estabelece periodicamente pontos de verificação que requerem a seguinte sequência de ações :

1. Gravação no dispositivo não volátil de todos os registros de *log* atualmente residentes em memória principal.

2. Gravação no disco de todos os blocos de *buffer* modificados.

3. Gravação no dispositivo não volátil de um registro de *log* **<checkpoint>**.

A presença de um registro **<checkpoint>** no *log* permite ao sistema refinar seu procedimento de recuperação.

2.8 Shadow Paging

Uma alternativa para técnicas de recuperação baseadas em *log* é a paginação com imagem (*shadow paging* – o termo em português aparece na edição traduzida [KOR 95]). Sob certas circunstâncias, a paginação com imagem pode exigir menos acesso a disco do que os métodos baseados em *log.* Existem, no entanto, algumas desvantagens no bloco de imagens.

Conforme ilustra a figura 2.8, a idéia chave por trás desta técnica é manter duas tabelas de páginas durante a existência de uma transação, a tabela de páginas corrente (A) e a tabela de páginas imagem (A`). Quando a transação se inicia, as tabelas de páginas são idênticas (A e A`). A tabela de páginas imagem (A`) nunca é alterada durante a transação. A tabela de páginas corrente (A) pode ser alterada quando uma transação executa uma operação **write.** Todas as operações **input** e **output** usam a tabela de páginas correntes (A) para localizar uma página do banco de dados no disco.

Para o processo de recuperação de um defeito durante a execução de uma transação, é sufuciente apenas liberar as páginas modificadas do banco de dados e descartar a tabela de páginas corrente. O estado do banco de dados antes da execução da transação fica disponível através da tabela de páginas imagem, e o estado é recuperado pela redefinição de que a tabela de páginas imagem venha a ser a tabela de páginas corrente novamente. O banco de dados é então retornado para o estado anterior da transação que foi executada quando ocorreu o *crash*, e todas as páginas modificadas são descartadas. Concluindo a transação corresponde a descartar a tabela de páginas imagem anterior [ELM 89].

A vantagem da paginação com imagem é que ela desfaz os efeitos da transação executada de forma muito simples. Além disto, não há a necessidade de refazer (*redo*) qualquer transação. Em um ambiente multiusuário com transações concorrentes, *logs* e *checkpoints* devem ser incorporados à técnica de *shadow paging* [ELM 89].

Pode-se citar, ainda, como vantagem sobre as técnicas baseadas em *log,* que a sobrecarga da saída de registros de *log* é eliminada, e a recuperação de quedas é significativamente mais rápida (uma vez que nenhuma operação Undo e Redo é necessária). Entretanto, existem desvantagens com relação a fragmentação de dados e *garbage collection*. Além destas desvantagens, paginação com imagem é mais difícil de adaptar-se a sistemas que permitem diversas transações concorrentemente. Em tais sistemas, algum armazenamento em *log* é requerido normalmente, mesmo que seja utilizada paginação com imagem. O SGBD System R, por exemplo, usa uma

combinação de paginação com imagem e um esquema de *log* [KOR 95].

O banco de dados InterBase, utilizado como sistema alvo neste experimento, faz uso da técnica de paginação com imagem, combinada com um esquema de *journal*, que tem semelhanças com o esquema de *log.* Entretanto, o esquema de *log* é utilizado pelo InterBase de forma diferenciada.

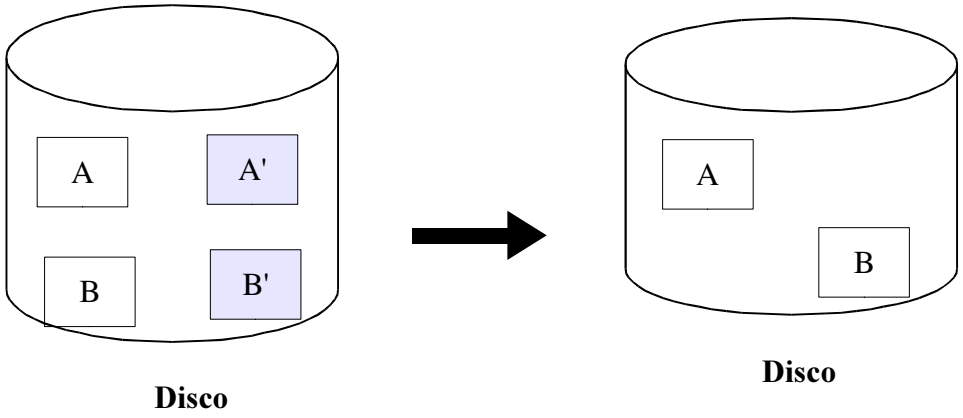

FIGURA 2.8 - A técnica *shadow paging* [HUN 99].

2.9 Técnicas de Recuperação em Uso

Segundo Lomet [LOM 98], as técnicas de recuperação em bancos de dados, seja de falhas de sistema ou de transações abortadas, baseadas em *log* evoluíram, principalmente através da liberação rápida e antecipada de bloqueios, do aumento do controle de concorrência e do uso de operações lógicas ao invés de operações físicas (que evitam a necessidade de copiar fisicamente dados para o *log*). Esta evolução, embora não represente um grande avanço tem encontrado espaço em sistemas de bancos de dados no mercado atual. Entretanto, enquanto for possível obter recuperação sem o uso de técnicas baseadas em *log* (ex. : através da técnica de *shadow paging*), alguns bancos de dados comerciais não usarão as técnicas baseadas em *log*.

O banco de dados InterBase, utilizado como sistema alvo neste trabalho, utiliza a técnica de *shadow paging* para a implementação de seu processo de recuperação.

2.10 Conclusão

O estudo e a compreensão do conjunto de conceitos e técnicas abordados neste capítulo é essencial para o entendimento do processo de recuperação seja ele baseado em mecanismos de *log* ou através de *shadow paging*. Além disto, o entendimento dos tipos de falhas, que podem afetar um SGBD centralizado, auxiliou na determinação das estratégias de injeção de falhas. Uma vez que o conjunto de falhas que podem afetar um banco de dados centralizado é conhecido, os conceitos e a compreensão do processo serviram como base para a definição do modelo de falhas proposto.

3 Injeção de Falhas

Este capítulo apresenta uma visão genérica sobre injeção de falhas e suas formas de aplicação. Relata e analisa alguns experimentos com ferramentas de injeção de falhas em SGBDs [CAR 98] [CAR 99] [COS 98] [COS 99] [MAD 99] [SAB 98] [SAB 99], que serviram como base para o desenvolvimento deste trabalho.

Um amplo estudo na área de erros de software em dois SGBDs comerciais (IMS e BD2) com ampla distribuição é relatado por M.P. Sulllivam [SUL 92].

Atualmente, provavelmente, o Centro de Informática e Sistemas da Universidade de Coimbra, que desenvolveu o Xception, seja o grupo com maior contribuição na área de injeção de falhas por software em Sistemas Gerenciadores de Bancos de Dados. Este grupo utiliza a ferramenta em diversos experimentos de injeção de falhas em SGBDs, com um significativo número de trabalhos e artigos publicados [CAR 98] [CAR 99] [COS 98] [COS 99] [MAD 99].

Sistemas computacionais são constituídos de uma série de componentes de hardware, de software e outros que podem falhar eventualmente [MAR 96]. Estas falhas podem levar o sistema a apresentar um defeito[1], ou seja, o serviço fornecido pelo mesmo não está de acordo com o que foi especificado [LAP 92].

Para alguns sistemas, a ocorrência de defeitos pode implicar em custos muito elevados em termos econômicos ou até mesmo em termos de vidas humanas. Como as falhas não podem ser de todo evitadas, pode-se tentar contrabalançar o seu efeito através de redundância. Esta redundância pode ser aplicada para se tolerar falhas que ocorram em componentes de hardware, de software e até no sistema. Assim, por exemplo, uma forma de compensar o efeito de falhas de hardware consiste em usar circuitos extras, como processadores duplicados ou memórias adicionais. A redundância é portanto a base das técnicas de tolerância a falhas, as quais visam garantir que o sistema continue a fornecer o serviço conforme a especificação mesmo em presença de falhas em alguns de seus componentes [LAP 92].

O uso de técnicas que permitam evitar e/ou tolerar falhas por si só não é suficiente para garantir a confiança no serviço fornecido pelo sistema. Os métodos para a construção de sistemas não estão livres de falhas; portanto é necessário o uso de técnicas que permitam eliminar ao máximo as falhas residuais existentes no projeto e/ou implementação de sistemas. Estas técnicas, por sua vez, também são imperfeitas; não é possível ainda eliminar de todo a possibilidade de ocorrência de falhas. É importante se fazer uma previsão do efeito das mesmas sobre o sistema. Estes dois aspectos dizem respeito à validação de um sistema, o que incluiria então [LAP 92] :

- a eliminação de falhas, que envolve a verificação, o diagnóstico e a correção;

1 Uma falha (“fault”) é a causa direta de um erro. Um erro é a parte do estado do sistema que pode levar a um defeito (“failure”). Não existe ainda um consenso quanto a terminologia a ser usada em português, os termos usados aqui estão baseados em [LEI 87].

- a previsão de falhas, que visa obter, por avaliação, medidas que permitam caracterizar o comportamento do sistema em presença de falhas.

Um ponto crucial na validação de sistemas tolerantes a falhas diz respeito a validação dos seus mecanismos de tolerância a falhas. A importância da validação destes mecanismos se deve a duas razões principais [MAR 96] :

- a presença de falhas de projeto/implementação nesses mecanismos pode levar á deficiências de comportamento dos mesmos quando em presença de falhas para as quais eles foram projetados para tratar; essas deficiências podem levar o sistema a não mais fornecer o serviço correto;

- o efeito da eficiência dos mecanismos de tolerância a falhas sobre medidas tais como a confiabilidade do sistema.

Não é facil caracterizar o comportamento de um sistema tolerante a falhas. Esta dificuldade advém das seguintes causas [MAD 99] :

- Os componentes que formam um sistema tolerante a falhas são muito complexos;

- As técnicas de tolerância a falhas são complexas;

- Há grandes incertezas relativamente à natureza das falhas, sua ativação e propagação de erros.

O uso de técnicas formais de verificação (como provas de programas) é altamente desejável para garantir a corretude do sistema, em particular, de seus mecanismos de tolerância a falhas. O uso de métodos analíticos (modelos de Markov, etc.) também são necessários para a obtenção de medidas de eficiência desses mecanismos. No entanto, devido a complexidade dos sistemas tolerantes a falhas, sua aplicação é limitada a partes do sistema e a consideração de um número reduzido de falhas [MAR 96].

Estas técnicas devem então ser completadas com a validação experimental. A injeção de falhas pode ser aplicada de diversas formas, em diferentes fases do ciclo de desenvolvimento de um sistema. O teste por injeção de falhas permite que sejam cobertos os dois aspectos da validação mencionados por Arlat [ARL 89] onde se pode observar que :

- na eliminação de falhas, o objetivo é reduzir ao máximo as falhas de projeto/implementação existentes nos mecanismos de tolerância a falhas;

- na previsão de falhas, o objetivo é avaliar a eficiência dos mecanismos de tolerância a falhas. Neste caso os testes servem para se obter estimativas de parâmetros tais como o fator de cobertura e a latência de detecção de erros.

A injeção de falhas, nome genérico dado a um conjunto de técnicas utilizadas para acelerar a ocorrência de falhas, erros e defeitos em um sistema, vem sendo largamente utilizada na validação dos mecanismos de tolerância a falhas de um sistema, pois

permite valida-los em presença de entradas particulares para as quais eles foram desenvolvidos para tratar : as falhas.

Na eliminação de falhas, a injeção de falhas tem por objetivo verificar se os mecanismos de tolerância a falhas se comportam de acordo com a sua especificação. Na previsão de falhas o objetivo é obter medidas da eficiência desses mecanismos de tolerância a falhas [MAR 96].

3.1 Formas de Aplicação

As falhas que afetam a confiabilidade de um sistema podem ser divididas em : falhas de hardware (“imitam” a conseqüência de falhas de hardware sobre o sistema) e falhas de software (falhas de projeto/implementação do software, designadas também como falhas de concepção).

A maioria dos estudos utilizando injeção de falhas são relativos à introdução de falhas de hardware. Estas falhas de hardware podem ser aplicadas de diferentes formas, dependendo do nível de abstração utilizado para representar o sistema (modelo ou protótipo) e do nível de aplicação das falhas (físico ou lógico). As formas de injeção mais empregadas são a simulação de falhas, a injeção física de falhas e a injeção de falhas por software [MAR 96].

Na **simulação de falhas**, falhas lógicas são introduzidas em um modelo do sistema. Este modelo pode representar o comportamento ou a estrutura (em termos de portas e/ou transistores) do sistema. As falhas são injetadas no modelo com o intuito de analisar o processo de ativação de falhas e de propagação de erros, para avaliar o comportamento dos mecanismos de tolerância a falhas. As falhas podem ser aplicadas por módulos especialmente construídos para este fim, ou através de mutações do modelo original.

A **injeção física de falhas** consiste em aplicar as falhas físicas a um protótipo do hardware (e/ou do software) do sistema. As falhas são injetadas geralmente sobre os pinos de circuitos integrados, mas pode-se também submeter os componentes ao bombardeamento por íons pesados ou ainda a aplicação de radiações eletromagnéticas. Neste caso os testes visam principalmente estudar o comportamento dos mecanismos de tolerância a falhas implementados por hardware, mas pode-se também testar software tolerante a falhas [MAR 89]. As falhas são aplicadas por um dispositivo especialmente construído para este fim, que interage com o sistema em teste.

Uma técnica que ganhou rapidamente grande aceitação é a injeção de falhas implementada por software (SWIFI = Software Implemented Fault Injection). Diferentemente de simulação, esta técnica requer que um sistema alvo seja construído, ao menos como um protótipo. Basicamente consiste da injeção de falhas no protótipo usando um software específico.

A primeira vista, o escopo de injetores SWIFI pode parecer completamente simples: o software tem acesso a registradores do processador, área de memória, ASIC (Aplication Specific Integrated Circuit) endereçáveis, e nada além disto em um sistema [CAR 99].

Na verdade, esta aparente simplicidade é enganosa. Como as ferramentas SWIFI obtém o controle total das funções do processador e estão aptas a manipular as informações que elas processa, podem emular um conjunto muito maior de tipos de falhas de hardware que a princípio se evidencia. Na prática, muitas ferramentas SWIFI podem injetar falhas que imitam os efeitos de falhas reais de hardware - falhas no controle de endereçamento, na unidade de aritmética, na memória e em uma quantidade de outras unidades funcionais. Além disto, estas ferramentas, por trabalharem no nível de software, podem também injetar falhas que imitem falhas reais de software [CAR 99].

As principais vantagens, com relação as outras técnicas, são a menor complexidade e esforço de desenvolvimento necessários, uma vez que não são necessários nem hardware dedicado e nem modelos muito detalhados. Além disto, ferramentas SWIFI podem aumentar portabilidade, podem ser facilmente expandidas para incluir novas classes de falhas e são imunes ás interferências físicas e elétricas, comuns em ferramentas de injeção de falhas físicas. Claro, algumas falhas não podem ser emuladas por ferramentas SWIFI, tais como falhas discretas nos tempos de acesso do *BUS*. Entretanto, os efeitos destas falhas são os inevitáveis erros de acesso á memória ou a carga de dados corrompidos para registradores - todos efeitos que podem ser emulados por ferramentas SWIFI [CAR 99].

Ferramentas SWIFI são uma parte de código de software muito incomum. Tais ferramentas esperam que algum evento ocorra (*trigger*), como um intervalo de tempo passar, uma mensagem ser enviada, ou uma célula específica de memória ser acessada; então interrompe o que o processador estiver executando, corrompe o sistema de alguma forma, e retoma a tarefa interrompida. Para fazer isto, ferramentas SWIFI podem, ou usar mecanismos de depuração disponíveis, ou modificar bibliotecas de comunicação para interceptar mensagens, ou até mesmo executar parcialmente em modo de supervisão no baixo nível do processador, ou no nível de manipulação de exceções. A ferramenta Xception, descrita na seção 3.2 usa esta técnica [CAR 99].

A técnica de injeção de falhas implementada por software tem as seguintes vantagens, em relação as outras duas apresentadas:

- não necessita de equipamento especial de hardware, como é o caso da injeção física de falhas;
- pode ser aplicada mais cedo (na fase de testes) do que a injeção física, pois não é necessário que o hardware onde vai residir o sistema alvo esteja disponível;
- oferece maior facilidade no controle e observação do sistema durante os testes, como a simulação, sem apresentar o tempo elevado de processamento desta última.
- apresentam baixa complexidade e custos de desenvolvimento reduzidos;
- oferecem maior facilidade de adaptação a diferentes sistemas alvos;
- não provocam interferências físicas com o sistema alvo.

Esta técnica é adequada para a validação de mecanismos de tolerância a falhas implementados por software, por ter a capacidade de injetar condições de erro específicas que permitam ativar esses mecanismos, o que não pode ser garantido na injeção física de falhas.

Segundo Madeira [MAD 99], a técnica de injeção de falhas implementada por software oferece algumas desvantagens típicas :

- não injeta falhas em unidades periféricas;
- não injeta falhas nas linhas de controle (BUS);
- a representatividade dos modelos de falhas é menos evidente;
- o injetor de falhas tem um grande impacto no comportamento do sistema alvo;
- as possibilidades de disparar falhas são reduzidas.

Na injeção de falhas por software o injetor pode situar-se na aplicação, entre a aplicação e o sistema operacional ou no próprio sistema operacional. A inserção do injetor de falhas diretamente na aplicação é baseada na alteração do código fonte da aplicação e pode interferir em características de temporização e mesmo mascarar, inadvertidamente, algum erro presente. A colocação do injetor de falhas no sistema operacional, pela alteração de seu código fonte ou por intermédio de algum mecanismo fornecido pelo mesmo, é bastante poderosa, dado o baixo nível de abstração em que se encontra. Como inconveniente, em alguns casos, há o fato de que as falhas injetadas agem sobre o ambiente e todas as aplicações podem sofrer influência do injetor. No caso de teste de uma aplicação específica, torna-se difícil limitar o escopo.

Quando o injetor de falhas está situado entre a aplicação e o sistema operacional, pode ser codificado como uma biblioteca, alterando chamadas específicas de uma determinada aplicação, como um filtro ou redirecionador, interceptando requisições ao sistema operacional, ou como um depurador. O injetor codificado como um depurador, ou baseado em técnicas de depuração, pode beneficiar-se de recursos oferecidos por um determinado processador ou por recursos fornecidos pelo sistema operacional [ROS 96]. A ferramenta de injeção de falhas, FIDe, utilizada nesta dissertação enquadra-se nesta categoria.

3.2 Injeção de Falhas em SGBDs com a ferramenta Xception

Xception [CAR 98] é um *software* de injeção e monitoração de falhas. Xception usa basicamente depuradores e monitores de desempenho existentes nos processadores modernos para injetar falhas nos sistemas alvo e monitorar o impacto das mesmas. Falhas são injetadas com uma mínima interferência na aplicação alvo. Esta aplicação não sofre modificações. Não são inseridos *traps* de *software* e nem a aplicação precisa

ser executada em modo *trace*.

Xception fornece um conjunto de falhas que podem ser ativadas, incluindo falhas espaciais e temporais. Pode injetar falhas em qualquer processo em execução no sistema alvo, inclusive no sistema operacional. O conjunto de falhas pode ser definido pelo usuário. Inicialmente foi implementado sobre uma máquina paralela utilizando processador PowerPC 601 (IBM/Motorola) e sistema operacional PARIX (versão paralela do Unix). Atualmente, está portada para outros processadores (SPARC, PowerPc 604, Pentium, etc.) e outros sistemas operacionais (Solaris, AIX, Windows NT).

Xception está dividido em três módulos (figura 3.1):

1) Um módulo injetor que esta *linkado* com o sistema alvo;
2) Uma biblioteca com funções para serem chamadas pelo usuário da aplicação para iniciar a injeção de falhas;
3) O módulo principal que roda no sistema *host* e implementa a interface com usuário para definição de falhas, injeção automática de falhas e coleta de resultados.

A injeção de falhas pode ser iniciada por qualquer processo que chame a função *StartXception()* presente na biblioteca de funções da ferramenta. A maior parte do código do Xception roda na parte *host* e a porção que roda no sistema alvo é mínima, não necessitando de modificações. A estrutura do Xception é mostrada na figura 3.1.

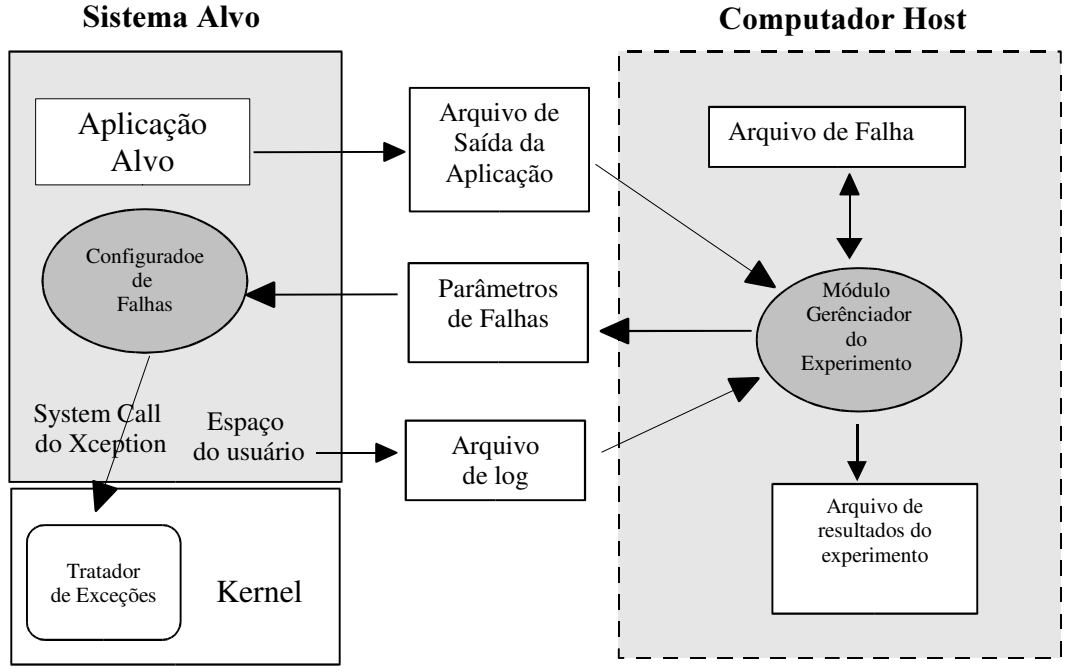

FIGURA 3.1 - Estrutura do Xception [CAR 98].

Xception prevê um completo conjunto de falhas que podem ser ativadas, incluindo as espaciais e temporais, ou ainda, ativadas pela manipulação de dados na memória.

Após a injeção de cada falha, Xception coleta os resultados e adiciona no arquivo de saída, que pode ser posteriormente analisado. Erros no nível de aplicação, como "acesso ilegal" ou ainda "instrução ilegal", podem ser detectados pelo sistema embutido no mecanismo de detecção de erro. Adicionalmente, o *kernel* gera mensagens para o *host* identificando a condição de erro. Essas mensagens, e também as do estado da injeção enviada pelo módulo Xception no nível do *kernel*, são gravadas no arquivo de *log* de erros, como mostrado na figura 3.1, as quais, mais tarde, são salvas pelo Xception no arquivo de resultados do experimento.

As mensagens do estado da injeção são enviadas para o *host* e gravadas no arquivo de *log* imediatamente antes da falha ser injetada, isto é, antes que o processo de exceção retorne o controle para a aplicação com dados errados. Esta mensagem contém informações de tempo para permitir o cálculo da latência na detecção do erro pelo *host.* A principal contribuição desta ferramenta é a possibilidade de injeção de falhas em qualquer processo em execução no sistema alvo, incluindo o sistema operacional, o que possibilita a injeção de falhas em aplicações em que o código fonte não esteja disponível.

Diversas publicações relatam o uso da ferramenta Xception [COS 98] [COS 99] [MAD 99] objetivando avaliar a integridade de dados e a disponibilidade em SGBDs. Nestes experimentos foi utilizado como sistema alvo o SGBD Oracle 7.3 versão Enterprise Edition rodando com Windows NT 4.0. O servidor foi um Intel PentiumPRO 200 Mhz com 128 MB de RAM. A arquitetura do sistema é uma típica configuração cliente/servidor com TCP-IP. O cliente roda em um Intel Pentium 150 Mhz com Windows workstation 4.0. As aplicações foram escritas em PL/SQL.

O modelo de falhas inclui falhas transientes de hardware e falhas de software. Falhas são injetadas apenas em *threads* ou processos do banco de dados. Apesar da facilidade de injetar falhas na camada de comunicação e nos componentes do cliente, os autores optaram por concentrar as falhas mais no servidor, uma vez que elas representam o maior ponto de enfraquecimento de dependabilidade dos sistemas.

O interesse maior dos autores foi obter uma amostra do impacto de falhas em termos de integridade de dados e disponibilidade. A integridade de dados foi enfocada em 3 níveis:

1) em nível de aplicação: usando o conjunto de regras semânticas e testes de consistência (especificados pelos *benchmarks* TPC);

2) em nível de integridade referencial: comparando as regras do banco de dados (dicionário de dados) com os dados armazenados;

3) em nível de arquivo: arquivos do banco de dados e arquivos de *log* são verificados através de suas integridades e de suas estruturas de dados internas.

A disponibilidade foi avaliada através de :

1) tempo médio do banco de dados para recuperar-se, com a visão do usuário final;

2) que tipo de recuperação foi necessária (parcial ou total);

3) esforço de recuperação, automático ou assistido pelo administrador de banco de dados (manual).

A recuperação no Oracle é baseado no mecanismo de *redo log.* O processo *log writes* escreve o *redo log* de *buffer* para o *redo log* de disco. Quando o processo do usuário é finalizado, o processo *log writes* salva as entradas *redo* que estão no *buffer* nos arquivos *redo* em disco (outros eventos podem também fazer com que entradas *redo* sejam escritas no disco). Os arquivos *redo log* em disco (no mínimo 2) trabalham de forma circular e o processo arquivo copia os arquivos *redo log* de disco para um dispositivo de armazenamento terciário quanto estes estiverem cheios. O Oracle executa *checkpoint* periodicamente (vários eventos podem disparar *checkpoints*) que representam estados consistentes do banco de dados, dos quais é possível iniciar a recuperação.

A recuperação consiste em uma fase de avanço, em que as entradas *redo* gravadas no *log* são usadas para regenerar os dados (e outras estruturas tais como segmentos *rollback*), seguido por uma fase de retrocesso, em que transações não recuperáveis são desfeitas para trazer o banco de dados a um estado consistente.

O conjunto de falhas aplicadas no experimento pretende, a princípio, ser representativo para falhas transientes de hardware que afetam sistemas de computação. As falhas são disparadas baseadas em tempo. Uma escolha aleatória de intervalos de tempo definida entre o início do processo do SGBD e um tempo de duração definida para a carga de trabalho foi usada para disparar a injeção de falhas. Desta forma, falhas foram distribuídas através do tempo de execução afetando potencialmente o código estático de execução do Oracle ou qualquer módulo de código carregável (*DLL*). No total foram injetadas 250 falhas [CAR 98].

A nível de resultados foram analisados 4 itens :

1. Impacto das falhas numa perspectiva do sistema operacional;
2. Mecanismos de detecção de erros;
3. Tempo de Recuperação;
4. Tempo médio para reparo.

Com relação ao impacto de falhas numa perspectiva do sistema operacional, foram usadas três avaliações, todas baseadas no código de retorno do processo servidor Oracle. As transações foram classificadas como **corretas** quando o processo finaliza a execução com código de saída normal, isto é, quando erros não foram detectados ou quando erros foram mascarados pela ação do mecanismo intrínseco de recuperação. Foram classificadas como **abortadas** quando o processo Oracle foi abortado devido a uma condição de erro interno. Foram classificadas como **suspensas** quando o processo Oracle foi congelado e concluído pelo injetor de falhas. Das 250 operações de injeção de falhas, em 217 (86,8%), retornaram como corretas, em 20 (8%) retornaram como abortadas, e em 13 (5.2%) foram suspensas.

Na avaliação dos mecanismos de detecção de erros, as exceções do sistema operacional

detectaram cerca de 37% dos erros, mostrando-se como o mais efetivo mecanismo de detecção de erros. O mecanismo *watchdog* do usuário/administrador detectou 30% dos erros. O teste de consistência TPC-HA não detectou uma única corrupção em dados do usuário, que é definitivamente uma boa marca de robustez do sistema alvo.

Na avaliação dos tempos de recuperação, em 80% dos casos foi menor que 1 minuto. Apenas poucas ações de recuperação necessitaram mais que um minuto, e o tempo de reparo mais alto foi de cerca de 4 minutos.

Com relação aos tempos médios para reparo, a média foi de aproximadamente 30 segundos para disponibilizar o banco de dados ao usuário, após ele ter sido posto *on line*, após a detecção de erro/defeito.

Embora tenham sido rodados em ambiente cliente/servidor com Windows NT 4.0. e utilizaram como sistema alvo o SGBD Oracle 7.3 versão Enterprise Edition, os experimentos relatados com a ferramenta Xception [COS 98] [COS 99] [MAD 99], foram os que mais contribuíram para esta dissertação, porque, igualmente, objetivavam a avaliação da integridade de dados e a disponibilidade, e se detiveram nas falhas transientes de *hardware* em um banco de dados centralizado. Também serviram para guiar a forma de condução dos experimentos (metodologia, carga de trabalho, modelo de falhas).

3.3 Experimentos de Injeção de Falhas no SGBDD ClustRa

Sabaratnam e Torbjornsem [SAB 98] [SAB 99], da Universidade de Ciência e Tecnologia da Noruega publicaram alguns artigos relatando resultados obtidos com o uso da técnica de injeção de falhas, tendo como sistema alvo o SGBD Distribuído ClustRa. Dos dois experimentos descritos abaixo, um enfoca a avaliação dos mecanismos de mascaramento de falhas, a propagação de erros para as réplicas e os tempos necessários para o processo de recuperação [SAB 98]. O outro enfoca o custo de verificações extras de consistência através de *checksums,* introduzidas na *cache,* com o o objetivo de preservar a integridade do banco de dados[SAB 99].

3.3.1 Mascaramento e Impacto de Falhas no SGBDD ClustRa

Sabaratnam e Torbjornsem [SAB 98] utilizaram a técnica de injeção de falhas, visando avaliar o impacto das falhas e os mecanismos de mascaramento de falhas pelas réplicas e erros propagados para as réplicas. Neste experimento o sistema alvo foi o SGBD distribuído ClustRa, através de replicação.

A injeção de falhas foi aplicada sobre a estrutura de dados presente na área de *buffer* de dados. A injeção de falhas também pode ser aplicada a outras estruturas do SGBD como *buffers* de *log* , *locks*, mensagens, etc. Segundo os autores, os *buffers* de dados foram escolhidos pelo seu papel crucial na manutenção da integridade do SGBD.

Foram criados diferentes cenários de injeção de falhas no SGBD ClustRa - versão 1.1. Os impactos destas falhas foram analisados, bem como a degradação do desempenho e o tempo gasto pelo sistema no processo de recuperação.

3.3.2 Plataforma para os Experimentos

ClustRa provê alta disponibilidade, alto desempenho e tolerância a falhas para aplicações de tempo real. Constitui-se de um SGBD replicado que roda sobre estações de trabalho UNIX. Um nodo ClustRa é uma estação de trabalho que executa um processo ClustRa. Os nodos são agrupados em unidades de desastres (*disaster unit*) diferentes, tendo nodos de falhas independentes, conectados através de linhas de comunicação velozes (grande largura de banda) duplicadas. A duplicação é utilizada nos processos, dados e comunicação, para minimizar a indisponibilidade de serviços e perda de dados, conforme mostra a figura 3.2. Os dados são divididos em fragmentos e cada fragmento é replicado em cópias primárias e secundárias (*hot standby*). São colocadas em nodos diferentes que pertencem a unidades de desastres distintas, tal que a união de réplicas diferentes em cada unidade de desastre fazem o banco de dados completo.

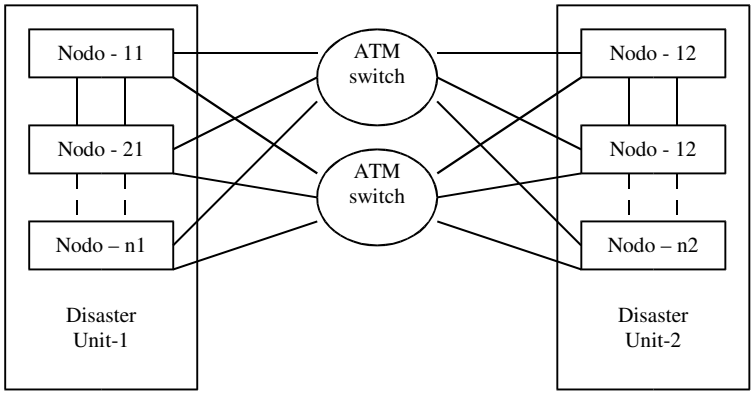

FIGURA 3.2 - Arquitetura do ClustRa.

Foram realizados experimentos, utilizando enfoques distintos, descritos a seguir [SAB 98]. Com o primeiro enfoque foram injetadas falhas em uma área qualquer do *buffer* de dados. A posição inicial para a injeção de falhas, que possa corromper os *buffers* de dados, foi distribuída uniformemente na área de *buffer.* O número de dados corrompidos foi baseado em estudos feitos por Sullivan e Chillarege [SUL 91], mas foram ajustados para o estudo de caso. A distribuição do número de bytes que foram corrompidos foi: 60% de 1 a 4 bytes, 35% de 5 a 1024 bytes e 5% de 1 a 9 KB.

Este tipo de corrupção geral de dados não traz qualquer informação específica a respeito da:

1) Eficiência da validação das estruturas de dados, ou;

2) Relação casual entre a fonte de erros e a gravidade do impacto do erro, isto é, que componente particular no *buffer* de dados causa o pior impacto.

Se estas respostas fossem conhecidas, o componente mais fraco poderia ser reforçado com a descoberta da melhor técnica para limitar o impacto do erro.

Com o segundo enfoque [SAB 98], foram injetados erros específicos sobre componentes particulares na área de *buffer* de dados.

Em ambos os enfoques, a metodologia consiste em inicializar o SGBD, gerar uma certa carga de trabalho, injetar falhas, parar o SGBD e analisar os *logs.* Os experimentos são administrados pelo gerenciador do experimento (EM - *experiment manager*). Os processos ClustRa são inicializados em quatro nodos, dois nodos são sobressalentes e dois são ativos. O gerador de carga de trabalho é inicializado 240 segundos após ter sido inicializado o SGBD. Espera-se 30 segundos. Após esse período o injetor de falhas é ativado e injeta um erro específico nos *buffers* de dados. A carga de trabalho é mantida durante quase 300 segundos quando é consultado o conteúdo do banco de dados. O SGBD é parado após 300 segundos. Na fase de análise, o conteúdo do banco de dados é comparado com o conteúdo do banco de dados mantido pela transação cliente (TC - *transaction client*), para verificar qualquer discrepância. Além disso, o cliente confere as respostas dadas pelo SGBD para todas as transações. O SGBD registra em *log* todas as recuperações realizadas bem como os tempos para estas recuperações.

3.3.3 Resultados Reportados

Nestes experimentos, com o primeiro enfoque, (falhas injetadas em uma área qualquer do *buffer* de dados), foram realizadas 725 operações. Com o segundo enfoque (falhas de tipos específicos injetados sobre componentes particulares na área de *buffer* de dados) foram realizadas 400 operações. No total foram realizadas 1125 execuções de injeção de falhas. O tempo utilizado para cada execução levou em torno de quinze minutos, totalizando 306 horas.

Como resultado dos experimentos, dois aspectos de tolerância a falhas foram avaliados:

1) Efetividade de descoberta de erros;

2) Eficiência de Recuperação.

A efetividade da detecção de erros foi medida pela cobertura de erros e defeitos fatais.

Quanto à **cobertura de erros,** com o primeiro enfoque, foram detectados 61% das 725 falhas injetadas. Foram mascaradas 97,5% das falhas descobertas. Com o segundo enfoque, foram detectados 62% das falhas injetadas, subdivididos nos vários tipos de erros. Ainda, foram mascarados 98% das falhas.

Falhas fatais foram definidas como: falhas em dobro (onde dois nodos falham quase que simultaneamente) e corrupção de dados. A primeira afeta a disponibilidade do

SGBD e a segunda afeta a integridade de seus dados. Com o primeiro enfoque, 1,5% das falhas injetadas ocasionaram falhas em dobro e 1% ocasionaram corrupção de dados. Com o segundo enfoque, 2% das falhas injetadas ocasionaram falhas em dobro e não foi verificada nenhuma corrupção de dados.

A eficiência de recuperação é caracterizada pelo período de tempo em que o sistema está reduzido ao nível de tolerância a falhas e a degradação de seu desempenho devido a atividade de tolerância a falhas.

O período de tempo em que o sistema está em recuperação, é o tempo em que o sistema roda sem replicação após o travamento de um nodo. Ele é calculado pelo tempo entre o travamento do nodo e sua recuperação com sucesso. Este período é muito importante, pois se o nodo falhar antes da sua recuperação terminar com sucesso, uma falha em dobro ocorrerá. Quanto mais longo for este tempo, maiores são as chances de uma falha em dobro ocorrer.

Degradação do desempenho reflete a sobrecarga provocada pelos mecanismos de tolerância a falhas. Degradação de desempenho é medida do ponto de vista do cliente. Cada execução mede o número das transações que tiveram sucesso por segundo (TPS). A média de TPS por falha é calculada para cada tipo de erro. Esta média é comparada com o TPS médio das 50 transações executadas livres de erros. A atividade de reparação tem geralmente um impacto negativo em relação ao desempenho, pois após uma falha de nodo, o nodo ativo deve auxiliar o nodo falho a restabelecer o nível de tolerância a falha. Além de servir a TCs (Transações Clientes) habituais, o nodo ativo tem que enviar a imagem do banco de dados, se necessário, e as mudanças que aconteceram no banco de dados depois que o nodo falhou, para o nodo recuperado. Esta atividade extra reduz o processamento de TC e dependendo do tipo de falha injetada as perdas de desempenho variaram de 12% a 21%.

3.4 Custo para Garantia de Segurança no SGBDD ClustRa

Em outro artigo dos mesmos autores [SAB 99] é relatado um estudo que avalia o custo de verificações extras de consistência através de *checksums*, introduzidas na *cache* de dados, com o objetivo de preservar a integridade do banco de dados, tendo como conseqüência pequena perda de desempenho. Com o auxílio da técnica de injeção de falhas, avalia também a melhora na cobertura de erros, as técnicas de tolerância a falhas e a ocorrência de duplicação de defeitos, gerando como resultado colateral, períodos de indisponibilidade.

A plataforma de teste deste experimento foi o SGBD distribuído ClustRA, descrito anteriormente. Um estudo elaborado pelos autores do SGBDD ClustRA aponta para a necessidade de se introduzir alguma forma de verificação de consistência a fim de preservar a corrupção da imagem do banco de dados. Verificações de consistência podem ser introduzidas no software em diferentes níveis, para obter diferentes níveis de segurança. Esta segurança não é obtida gratuitamente. A maioria das verificações de consistência aumentam o impacto adverso em tempo de resposta. Isto pode não ser aceitável para algumas aplicações críticas. Entretanto, é necessária para quantificar o custo de diferentes níveis de segurança a fim de determinar as questões de custo de

desempenho.

Os autores sugerem que uma forma de detecção de erros é incorporar verificações de consistências, asserções e exceções apropriadas, baseadas em regras semânticas, que pertencem ao SGBD. Uma outra forma (que foi utilizada neste trabalho), é anexar *checksums* a objetos específicos do SGBD, tais como *buffer* de dados, registros de *log*, bloqueio de transações, mensagens, e execução de instruções. Estas duas formas são complementares e ambas implementam e reforçam a robustez do SGBD, mas como mencionado anteriormente, o custo será muito alto e portanto, a escolha entre um tipo de verificação de consistência faz-se necessária. O *buffer* de dados é a estrutura de dados chave em um SGBD. Parte ou toda a imagem do banco de dados é armazenada nela para processamento. A corrupção definitiva, quando ocorre na imagem do banco de dados, causada por hardware ou software malicioso, pode levar a impactos catastróficos de integridade,

Checksums são geralmente usados para detectar erros e não para mascará-los. Um *checksum* anexado a um objeto é calculado sempre que o objeto é atualizado e é armazenado com o objeto ou em outra situação de tolerância a falhas. Quando o objeto é acessado, seu *checksum* é recalculado e comparado com o *checksum* armazenado. A probabilidade que um *checksum* de 32 bits não detecte um erro é tão baixa quanto $2.3 * 10^{-10}$. Entretanto, os autores assumem que a maioria das coberturas que não são detectadas pelos *checksums* do sistema operacional ou pela verificação de tipos em tempo de execução serão detectadas pelo mecanismo de *checksum* proposto no trabalho.

A sobrecarga de *checksum* é esperado, gerando menor impacto no desempenho quando a carga no sistema for baixa que quando ela for alta. A fim de avaliar a sobrecarga dos *checksums* com diferentes cargas, diferentes cargas de trabalho foram geradas pela variação do:

1) número de transações clientes (TC) utilizando o SGBD concorrentemente, e

2) número de registros de dados que uma transação atualiza. Quando o número de registros acessados por uma transação é baixo, a sobrecarga para envio e recepção de mensagens predomina no tempo de resposta, sendo maior que o custo para cálculos de *checksums*.

Quarenta diferentes cargas de trabalho foram geradas pela combinação do número de transações clientes (2,4,6 e 8) e o número de registros acessados por cada transação (1,2,4,6 e 8). Vinte testes foram conduzidos em duas implementações do SGBD - uma com *checksums* e outra sem *checksums.* Cada teste consistia de dez execuções, totalizando 400 execuções. Cada execução levou em torno de 10 minutos, totalizando 66 horas.

Uma execução do teste consistia em inicializar o SGBD e o gerador de cargas de trabalho, parar o SGBD e analizar os *logs.* Estes testes foram monitorados por um gerenciador de experimento (EM). Processos ClustRA foram inicializados em quatro nodos. A carga de trabalho era inicializada 240 segundos após o SGBD ser inicializado. A carga de trabalho era mantida por aproximadamente 300 segundos e então o SGBD era parado. Após isto, a informação encontrada no *log* do SGBD era analisada e as

métricas de desempenho eram calculadas.

Neste trabalho, os resultados obtidos pelos autores mostram que a sobrecarga de processamento de *checksum* em um SGBD foi punido com uma carga de trabalho alta, causando uma redução no *throughput* de 5%. A cobertura de detecção de erros aumentou de 62% para 92%. Os experimentos de injeção de falhas mostram que a corrupção na imagem do banco de dados caiu de 13% para 0%. Isto indica que as aplicações que requerem alta segurança, mas podem dispor de 5% de perda de desempenho, podem adotar mecanismos extras de *checksum*.

3.5 Conclusão

Embora estes experimentos utilizem como sistema alvo um SGBD distribuído, no qual o modelo de falhas difere consideravelmente de um modelo de falhas de um SGBD centralizado e as técnicas e objetivos sejam bastante diferentes, contribuíram de forma significativa para essa dissertação, uma vez que proporcionaram uma visão clara do modo de abordar este tipo de experimento (metodologia, medidas de avaliação, carga de trabalho, resultados). Além disto, existem algumas semelhanças entre esses experimentos e os conduzidos nessa dissertação: impacto das falhas na integridade dos dados, cobertura na detecção de erros e tempos de indisponibilidade.

4 O Injetor de Falhas FIDe

Este capítulo apresenta o FIDe (*Fault Injection via Debugging)* [GON 2002], uma ferramenta de injeção de erros, desenvolvida no PPGC da UFRGS, que foi usada nesta dissertação. A ferramenta é baseada nos mecanismos de depuração oferecidos pelo sistema operacional Linux.

O FIDe possui um princípio de funcionamento bastante simples: executar a aplicação até que seja encontrada uma das chamadas de sistema passíveis de injeção, um contador atinja o valor definido ou um determinado tempo tenha passado. Neste ponto, as alterações em valores de registradores ou conteúdo de memória são efetuados, segundo um roteiro de injeção (seção 4.2).

A arquitetura do FIDe, seus componentes básicos e a sua interface com o sistema operacional, com o processo em teste e com o operador são apresentadas na seção 4.3.

O FIDe proporciona grande flexibilidade, através de suas simples e poderosas, regras de injeção. Por meio destas regras são construídos os cenários utilizados nos testes e validações. Exemplos de criação de regras aparecem nessa dissertação no capítulo 6, seção 6.3.

4.1 Funcionamento da Ferramenta

A função *ptrace* oferece três alternativas para depuração de código objeto: a) execução do processo instrução por instrução (passo-a-passo); b) execução até um ponto de parada (*breakpoint*) e c) execução até encontrar uma chamada de sistema (entrada ou saída da mesma). Executar a aplicação até encontrar uma chamada de sistema foi a alternativa escolhida para o desenvolvimento do FIDe. Todo o acesso a recursos do sistema e acessos a hardware são executados via chamadas de sistema. Assim, pode-se alterar e adulterar o comportamento e resposta de funções como leitura e gravação de arquivos, criação de processos, alocação de memória, envio de sinais, operações em *sockets* e outras. O FIDe executa a aplicação até que encontre a entrada de uma chamada de sistema. Neste ponto a aplicação pára e devolve o controle ao FIDe (figura 4.1).

Verifica-se o valor de **orig_eax**, que identifica a chamada de sistema na qual se está entrando. Caso este valor coincida com uma das chamadas em observação, o FIDe verifica a cadeia de regras referente a esta chamada. Caso seja o momento da injeção, isto é, as regras tenham sido satisfeitas, são executadas as ações de injeção. As ações de injeção definem como devem ser manipulados valores de registradores, memória ou parâmetros passados á chamada de sistema. Neste ponto, a chamada de sistema ainda não foi executada [GON 2002].

O próximo passo, independente de haver injetado erros ou não na entrada da chamada de sistema, é ordenar que a aplicação seja executada até a saída da chamada em questão. Este passo executa a chamada de sistema com parâmetros alterados ou não e pára ao fim

da mesma (figura 4.1). Como no passo anterior, é verificado o valor de **orig_eax** e comparado com as chamadas de sistema presentes nas regras de injeção. São também executados os procedimentos de injeção referente a saída desta chamada de sistema, caso a verificação tenha resultado positivo e existam regras para isso. Aqui, a chamada já foi executada e são alterados os resultados da mesma.

Este ciclo se repete até que seja finalizado o processo. Um processo pode ser finalizado normalmente, por ter executado todo seu processamento, ou por alguma condição anormal. No caso de finalização por condição anormal, erros, violação de segmento ou outro motivo qualquer, um sinal é enviado ao injetor. A análise deste sinal pode revelar a causa.

Duas chamadas consecutivas à função **ptrace** vão executar toda uma chamada de sistema. Isto é, a primeira chamada a função **ptrace** entra em uma *syscall* e pára. Uma segunda execução da função **ptrace** executa a chamada de sistema, parando o término desta [GON 2002].

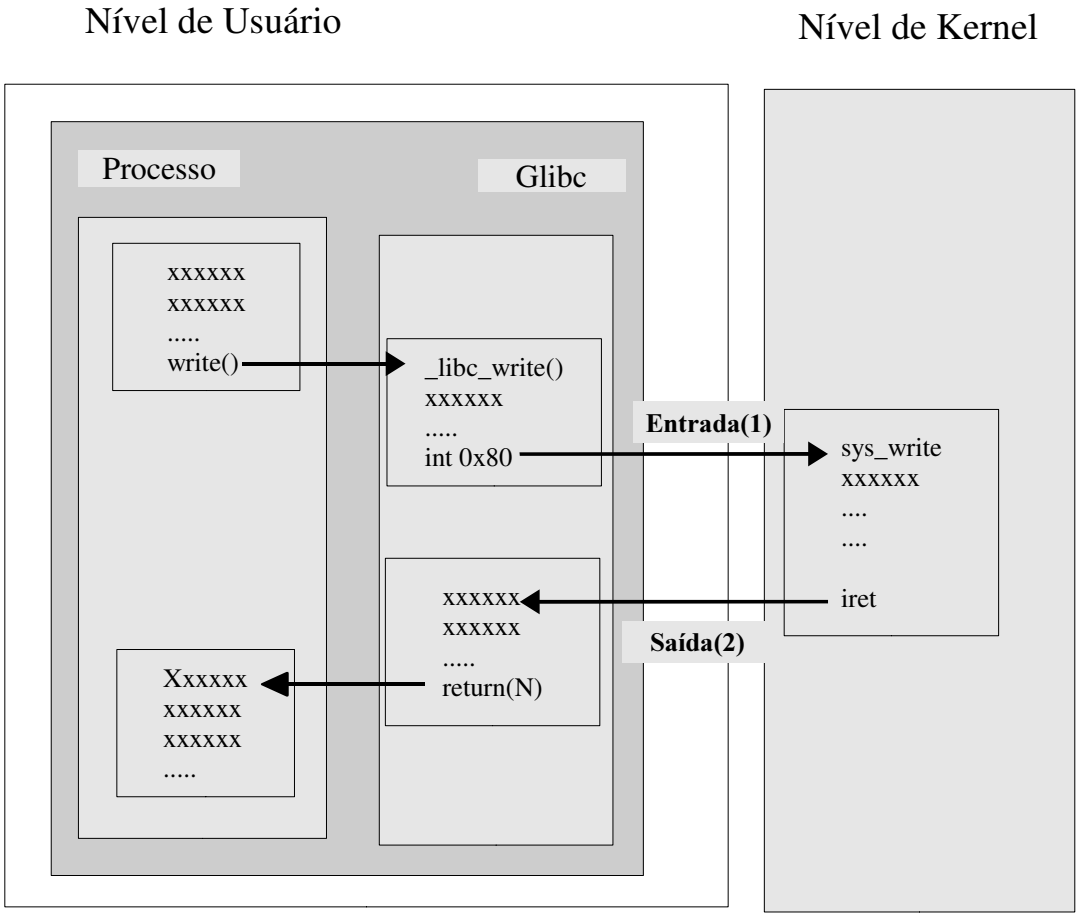

FIGURA 4.1 – Interação do Processo, Glibc e Chamada de Sistema [GON 2002].

Se o processo sob injeção cria novos processo, via chamadas de sistema **fork** ou **clone**, o FIDe também passa a monitorar e injetar falhas nestes.

4.2 Arquitetura da Ferramenta

A ferramenta é composta basicamente por um injetor de falhas que executa e manipula uma aplicação sob teste, controlado por um roteiro de injeção. Este roteiro explicita onde, quando e como devem acontecer as injeções de falhas (figura 4.2).

A comunicação entre o injetor e o processo alvo, aplicação em teste, se dá via chamada de sistema **ptrace**. Esta chamada é a interface entre o injetor e as rotinas de depuração presentes no *kernel* do Linux. O injetor pode controlar o processo alvo, dentro das possibilidades oferecidas pelo *kernel*.

Após a leitura e validação do cenário de falhas, o FIDe inicializa em memória o processo alvo, já pronto para ser depurado. Para tanto, o FIDe cria uma cópia de si próprio (**fork**()) e no processo filho ativa a *flag* de depuração (**ptrace**()). A próxima operação é executar o processo alvo (**execve**()) [GON 2002].

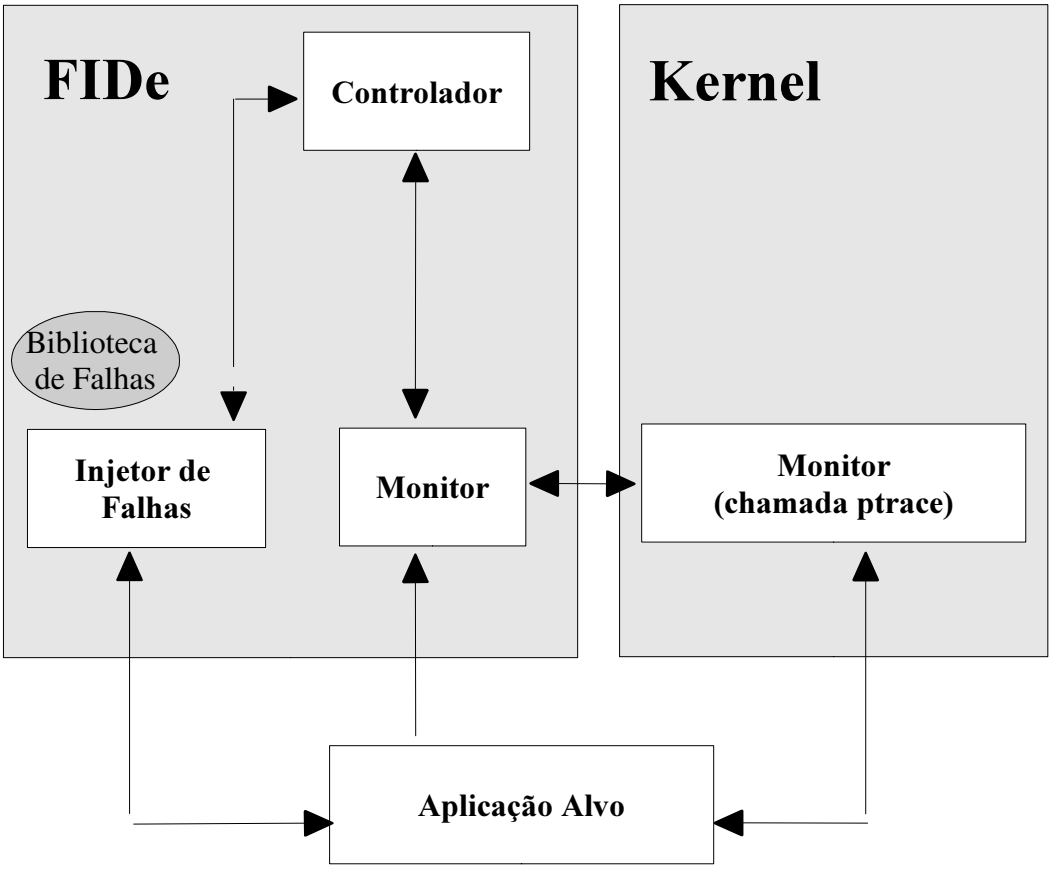

FIGURA 4.2 – Arquitetura do FIDe [GON 2002].

O FIDe envia comandos, identificados como controle na figura 4.2, e dados para a

chamada **ptrace** e recebe dados da mesma. Como exemplos de comandos, podemos citar: execute o processo *x* até a próxima chamada de sistema, envie o sinal SIGHUP para o processo *y,* leia o valor do registrador **eax** no processo *z,* entre outros. Os dados enviados e recebidos podem ser valores de registradores, conteúdo de memória ou dados da estrutura **user** do processo sob injeção.

Como as chamadas de sistema possuem tempos diferentes de execução e uma mesma *syscall* pode variar seu tempo de execução em função dos parâmetros recebidos, o sistema operacional fornece um mecanismo de sincronização. O injetor executa uma chamada **ptrace** e espera, via função **wait**(), que o processo sinalize o fim da execução. O sinal enviado neste caso é SIGTRAP.

A exceção a esta regra são as chamadas de sistema como **wait** e **rt_sigsuspend**, que mantém o processo suspenso até que um determinado evento ocorra. Neste caso, o injetor ficaria esperando até que este evento ocorresse [GON 2002].

4.3 Regras de Injeção

O arquivo de cenário define o comportamento do injetor durante a execução da aplicação em teste. Este arquivo é composto basicamente por regras, aninhadas ou não, que definem quando deve acontecer uma intervenção e o quê deve ser feito. O efeito da injeção de falhas é tão próximo de uma falha real quanto o for o cenário definido.

A definição das regras de injeção se dá por meio de uma linguagem mnemônica. Esta linguagem permite acesso a todas as possibilidades do FIDe e é composta por comandos de um caractere e valores numéricos (em decimal). Para facilitar a verificação das regras criadas, enquanto ainda não existe uma GUI para a criação das mesmas, foi criado um *script*, chamado fide_rules.py, na linguagem Python, que realiza a leitura e tradução das regras para linguagem natural.

Uma regra é composta por três partes: *regra principal, regras de decisão* e *regras de ação.*

A regra principal define **qual** a chamada de sistema deve ser alvo de injeção. Um roteiro de injeção pode conter várias regras principais, mas não permite o aninhamento destas. A regra principal é definida pelo tipo **main_rule.**

As regras de decisão definem condições de **quando** e **onde** devem ser injetadas as falhas. Uma regra principal pode conter várias regras de decisão, aninhadas ou não. Podem ser baseadas em tempo, demanda ou ativadas por um gatilho. Podem, ainda, indicar que a injeção deverá acontecer na entrada da chamada de sistema, na saída da chamada de sistema ou em ambos os pontos. Estas regras são identificadas pelo tipo **rule_when.**

As regras de ação, por sua vez, definem **o quê** deve ser feito durante o processo de injeção. Uma regra de decisão, aninhada ou não, pode conter várias regras de ação. Estas regras podem agir sobre o conteúdo de registradores (**rule_reg**), conteúdo de memória

(**rule_memo e rule_param**) e sobre as estruturas de controle do processo utilizadas pelo sistema operacional - em especial a estrutura **user (rule_user)**.

Todas as regras são analisadas e executadas na ordem em que foram declaradas no arquivo de cenário. Abaixo, os seis tipos de regra e suas categorias:

REGRA PRINCIPAL

- **main_rule:** define qual a *syscall* a ser observada. Um arquivo de cenário pode conter várias regras deste tipo, isto é, pode definir a atuação do injetor sobre várias *syscalls* diferentes.

REGRAS DE DECISÃO

- **rule_when:** define em que ponto da *syscall* acontecerá a injeção da falha e quando acontecerá a injeção. A injeção pode acontecer na entrada da *syscall*, na saída ou em ambas. A condição de ativação da injeção pode ser baseada em tempo, demanda ou em função de um *trigger* (valor de um registrador).

REGRAS DE AÇÃO

- **rule_reg:** age sobre registradores, permitindo a execução de operações lógicas e aritméticas sobre seus valores ou mesmo a definição de um valor arbitrário para um registrador.

- **rule_memo:** age sobre o conteúdo de posições de memória, permitindo o mesmo conjunto de operações de **rule_reg.**

- **rule_param:** permite que se modifique, assim como nas regras anteriores, o conteúdo de uma posição de memória apontada por um registrador, acrescida ou não de um deslocamento.

- **rule_user:** age sobre os dados da estrutura *user* do processo sofrendo injeção de falhas. Esta estrutura possui dados sobre o processo, sua execução, além de outras informações.

O formato de uma regra é:

main_rule
rule_when
...
rule_reg | rule_memo | rule_param | rule_user]
...

Uma vez que a regra **main_rule** seja satisfeita, isto é, a *syscall* em execução combine

com uma das *syscalls* definidas nas regras, verifica-se suas respectivas regras **rule_when.** Se estas regras forem satisfeitas começa a injeção de falhas definidas pelas regras **rule_reg, rule_memo, rule_param** e **rule_user** subseqüentes.

4.3.1 main_rule

Uma regra do tipo **main_rule** é definida pelo mnemônico 1, seguido do número da *syscall* desejada. Os comandos e seus parâmetros são separados por espaços em branco e cada linha abriga apenas uma regra. Um exemplo desta regra seria [GON 2002]:

1 5

A regra acima diz respeito à *syscall* de número 5 (Open).

TABELA 4.1 – Lista de Operações.

Token	***Significado***
"="	atribuição de valor
+	soma
-	subtração
*	multiplicação
&	Operação lógica "E"
|	Operação lógica "OU"
>	Deslocamento para direita
<	Deslocamento para esquerda

4.3.2 rule_when

As regras do tipo **rule_when** são um pouco mais complexas e seguem o modelo abaixo:

W onde quando op {reg_id]

- *onde:* indica em que ponto da *syscall* deve acontecer a injeção da falha: I (entrada), E (saída) ou B (ambos);

- *quando:* define se a injeção será baseada em tempo, em fluxo ou ativada por

um gatilho (valor de um registrador). As possibilidades são:

- **tempo**: A (depois de *op* segundos), D (durante *op* segundos), E (a cada *op* segundos) e O (uma vez, depois de *op* segundos);
- **demanda**: F (depois de *op* chamadas a esta *syscall*), N (nas próximas *op* chamadas a esta *syscall*), H (a cada *op* chamadas a esta *syscall*) e R (uma vez, depois de *op* chamadas a esta *syscall*);
- **gatilho**: T, acionado quando o valor do registrador reg_id for igual a *op*. Os possíveis valores de *reg-id* são apresentados na tabela 4.2

- *op*: é composto de uma operação e um valor. As operações aqui listadas são válidas para todos os tipos de regras e agem sobre o valor de registradores e conteúdos de memória. Nos casos onde se faz necessária a existência de um segundo operando, o valor do item *reg_id* é utilizado para este fim, como um inteiro de 32 bits. O conjunto de operações é apresentado na tabela 4.1.

- *reg_id*: identifica qual o registrador a ser utilizado nas regras disparadas por gatilho. Nas outras regras, quando necessário, contém o valor a ser utilizado como segundo operando. Os registradores são identificados na tabela 4.2.

Como exemplo de regras do tipo **rule_when**, a regra abaixo [GON 2002] define que as falhas serão injetadas na saída da *syscall*, a cada cinco chamadas.

W E H 5

As regras que controlam a forma de injeção podem agir sobre o conteúdo de registradores (**rule_reg**), o conteúdo de posições de memória (**rule_memo**), o conteúdo de posições de memórias apontadas por registradores (**rule_param**) – geralmente parâmetros da *syscall* – e sobre dados da estrutura *user* do processo sob injeção (**rule_user**).

4.3.3 rule_reg

As regras **rule_reg** são expressas da seguinte forma:

R *reg_id op valor*

Os possíveis valores de *reg_id* e *op* podem ser encontrados na tabela 4.2 e 4.1 respectivamente. *Valor* é um inteiro de 32 bits. No exemplo abaixo, define-se que caso **main_rule** e **rule_when** sejam satisfeitas, o novo valor do registrador *ecx* será resultado de uma operação lógica “OU” entre seu valor antigo e o número 7.

R 1 | 7

TABELA 4.2 – Lista de Registradores [BEC 98].

Valor	*Registrador*
0	EBX
1	ECX
2	EDX
3	ESI
4	EDI
5	EBP
6	EAX
7	DS
8	ES
9	FS
10	GS
11	ORIG_EAX
12	EIP
13	CS
14	EFL (eflags)
15	UESP (esp)
16	SS

4.3.4 rule_memo

As regras **rule_memo** são expressas segundo o modelo abaixo:

M *end op valor*

O endereço de memória a ser alterado pela regra, identificado por *end,* é um valor de 32 bits. Os outros itens seguem o modelo utilizado nas regras anteriores. Caso a regra definida faça referência a um endereço de memória fora do escopo do processo sob injeção de falhas, uma violação de segmentação será gerada, juntamente com um *core dump*, e o processo será abortado. Esta exceção não foi tratada, justamente para aumentar as possibilidades de geração de falha.

Abaixo, um exemplo de regra [GON 2002] que define o conteúdo da posição de memória de endereço 115000, e subseqüentes já que valor possui 32 bits, para 0.

M 115000 = 0

4.3.5 rule_user

A regra **rule_user** difere da regra anterior no fato de que não é apontado um endereço de memória e, sim, um elemento da estrutura user do processo. A regra é definida conforme o modelo abaixo:

U *pos op valor*

Desta forma, no exemplo abaixo [GON 2002], para alterar o primeiro parâmetro da estrutura *user*, o valor do registrador **ebx,** tem o seu valor multiplicado por 2:

U 0 * 2

4.3.6 rule_param

As regras **rule_param** diferem das regras anteriores em dois aspectos: o endereço de memória é apontado por um registrador e pela possibilidade de deslocar-se, para frente e para trás, a partir deste endereço. A regra é definida como no modelo abaixo:

P *reg_base [offset] [ajuste] op valor*

O parâmetro **reg_base** indica qual registrador apontará o endereço base. Os possíveis valores deste argumento, identificação dos registradores, são indicados na tabela 4.2. Os parâmetros *offset* e *ajuste* possuem o mesmo formato e são compostos por um identificador, um indicador de direção e um valor ou identificação do registrador a ser utilizado.

O identificador de *offset* pode ser: 0 (*offset*) ou R (registrador). O indicador de direção informa ao injetor se ele deve avançar (+) ou retroceder (-) a partir do endereço base. Desta forma [GON 2002], avançar 4 bytes seria representado como “0 + 4” e retroceder **ecx** bytes seria “R – 1”.

O parâmetro *ajuste* permite que se aponte para uma posição exata, quando do uso de registradores no *offset*.

Para armazenar o valor 10, lembrando-se que se trata de um valor de 32 bits, na posição apontada pelo registrador **edx**:

P 2 = 10

Para armazenar o valor 10, cinco bytes à frente da posição apontada pelo registrador

edx:

P 2 0 + 5 = 10

Para armazenar o valor 10, **ecx** bytes à frente da posição apontada pelo registrador **edx.**

P 2 R + 1 = 10

Para armazenar o valor 10, na posição apontada por [**edx+ecx**+5].

P 2 R + 1 0 + 5 = 10

4.4 Criação de Regras

A validade de um experimento está fortemente vinculada às regras de injeção e aos cenários utilizados. Os comandos para a criação de regras de injeção do FIDe, são flexíveis e poderosos o suficiente para que se crie um cenário de erros com alto grau de detalhamento. Por trabalhar com manipulação de parâmetros passados para as chamadas de sistema, sejam estes registradores ou dados em memória, o FIDe permite que se realize experimentos sem conhecer previamente a aplicação sob teste [GON 2002].

Exemplos de criação de regras aparecem nas publicações sobre o FIDe [GON 2002] [ROD 01A] e nessa dissertação no capítulo 6 seção 6.3.

4.5 Injeção de Falhas com o FIDe no SGBDD Progress

Em recente estudo enfocando injeção de falhas em SGBDs, desenvolvido no grupo de tolerância a falhas do PPGC/UFRGS, Manfredini [MAN 01] usou as ferramentas de injeção de falhas FIDe e ComFIRM [LEI 00]. Este trabalho teve como objetivo a aplicação das técnicas de injeção de falhas por software em sistemas gerenciadores de bancos de dados distribuídos (SGBDD), visando a avaliação de sua disponibilidade e a eficiência de seus mecanismos de tolerância a falhas, bem como a obtenção do custo computacional destes mecanismos.

Também visava a validação das ferramentas de injeção de falhas utilizadas (FIDe e ComFIRM), realimentando o processo de desenvolvimento e o aprimoramento destas ferramentas através dos resultados obtidos e dificuldades encontradas pela sua utilização na condução dos experimentos.

Os experimentos de Manfredini correram em paralelo com os experimentos relatados nessa dissertação. Apesar dos experimentos possuírem objetivos semelhantes, o SGBD alvo é totalmente diverso, usando mecanismos de recuperação e modelo de falhas diferente do encontrado no InterBase. Devido a isso, foi possível usar o FIDe no Progress em um estágio de desenvolvimento do FIDe anterior ao necessário para aplicá-

lo no InterBase. Os experimentos de Manfredini validaram o FIDe em sua fase inicial de desenvolvimento. Os experimentos com o InterBase permitiram concluir essa validação

Os servidores do banco de dados foram dois computadores de arquitetura Intel Pentium III de 600 MHZ com 256 MB de RAM. Como cliente foram utilizadas 2 estações de trabalho de arquitetura Intel com processadores Pentium III de 450 MHZ com 128 MB de RAM. A interligação dos servidores e das estações de trabalho é feita através de uma rede local padrão Ethernet de 100 Mbit com cada equipamento num segmento específico da rede suportado através de um *switch*.

Os seguintes componentes de software completaram a plataforma: sistema operacional GNU/Linux, versão de *kernel* 2.2.16, para os servidores do Banco de Dados Distribuído (BDD), WIN98/2000 para os clientes, o SGBD Progress, o sniffer Dsniff, as ferramentas de injeção FIDe e ComFIRM, o gerador de carga de trabalho GerPro-TPC e o gerenciador de injeções e resultados GIR.

Para a implementação do BDD, do Gerador de Carga e do Gerenciador de Injeção e Resultados, foi utilizada a versão do Progress 8.3c.

O Progress utiliza técnicas de tolerância a falhas como: arquivos de *log* (*logs* de *before image* para os processos de *undo* e *after image* para os processos de *redo*), *checkpoint* para atualizar nos arquivos físicos do BD a porção do BD residente em *buffers* de memória e o protocolo *two phase commited* para garantir atomicidade do *commitment*. Alguns destes mecanismos são opcionais, e podem ser ativados e desativados, o que facilita o cálculo do custo computacional destes.

O GerPro-TPC (**Ger**ador de Carga **Pro**gress especificação **TPC**c), desenvolvido por Manfredini para o experimento, segue as especificações e recomendações TPC Benchmark C (TPC-C). TPC-C é uma carga de trabalho OLTP. Ele mistura transações de leitura e atualizações da base de dados visando simular atividades encontradas em aplicações complexas de OLTP, tendo como principais características:

- Execução simultânea de múltiplos tipos de transações;
- Execução de transações *On-line* e *batch*;
- Múltiplas seções ativas;
- Controle sobre o tempo de execução das transações;
- Significativo volume de leitura/gravação de dados em disco;
- Integridade das transações (propriedade ACID);
- Não uniformidade na distribuição de acessos aos dados através de suas chaves primárias e secundárias;
- Banco de dados constituído de muitas tabelas de variados tamanhos, atributos e relacionamentos;
- Concorrência no acesso e atualização das tabelas.

As métricas sugeridas pelo TPC-C são medidas pelo número de ordens processadas por

minuto. Múltiplas transações são usadas para simular uma atividade comercial usual, sendo que cada transação é submetida a um comparador de tempo de resposta. O desempenho é medido pelo número de transações por minuto (tpmC).

O modelo de dados proposto pelo TPC e adotado pelo GerPro-TPC, simula uma empresa atacadista com regiões de vendas distribuídas geograficamente associadas a depósitos regionais. Cada depósito regional supre dez regiões. Cada região atende 10.000 clientes. Todos os depósitos mantêm um estoque de 100.000 itens vendidos pela empresa. Os clientes podem fazer novos pedidos ou consultar a posição de um pedido. Os pedidos são compostos em média por 10 itens. Um porcento de todos os itens de pedidos não são atendidos pelo depósito a que a região pertence, mas outros depósitos atende-os.

Inicialmente, antes de qualquer transação TPC ou injeção de falhas o banco de dados é populado com 2.595.055 registros. A geração destes registros segue regras rígidas descritas na especificação do *benchmark* TPC-C.

A ferramenta ComFIRM foi utilizada para simulação de comunicação entres os servidores e clientes que compõem o BDD. Já o FIDe foi utilizado para simulação de falhas transientes e permanentes de hardware, como falhas de memória e leitura/gravação de dados em disco.

Embora o SGBD Progress seja um banco de dados distribuído e, por conseqüência, o conjunto de falhas seja diferente do conjunto de falhas de um banco de dados centralizado, este trabalho contribuiu significativamente porque utilizou o FIDe como ferramenta de injeção de falhas e porque houve, no decorrer do desenvolvimento de ambos os trabalhos, trocas de informações e discussões quanto as dificuldades encontradas e a melhor forma de solucioná-las. Os conhecimentos obtidos através dos experimentos conduzidos por Manfredini, com o FIDe, possibilitaram a melhor compreensão do impacto das falhas no ambiente do SGBD e facilitaram a escolha do conjunto de falhas a injetar e a montagem dos cenários de falhas que foram utilizadas nesta dissertação.

5 O Ambiente dos Experimentos

Neste capítulo são apresentados, de forma detalhada, os componentes da plataforma de hardware e software utilizada, a carga de trabalho, o modelo de falhas e a metodologia de testes adotada.

5.1 Plataforma de Trabalho

Nos experimentos foram utilizados os seguintes recursos:

- um microcomputador com processador Intel Pentium 550 MHz, 128 MB de memória Ram, disco winchester de 4.0 Gigabytes;
- o sistema operacional Linux Conectiva 4.2 Kernel versão 2.2.13;
- o SGBD centralizado InterBase versão 4.0;
- a ferramenta de injeção de falhas FIDe;
- um conjunto de *scripts*, escritos em SQL, para gerar a carga de trabalho (ver figura 5.1).

5.2 O Sistema Operacional GNU/Linux

O sistema operacional GNU/Linux foi escolhido como plataforma operacional do servidor do BD e das ferramentas de injeção de falhas utilizadas, principalmente por suas características de *software* livre. Originalmente desenvolvido para a arquitetura i386, atualmente está disponível para uma ampla gama de arquiteturas de PDA a *mainframes* passando por supercomputadores. Desde sua primeira versão está sob licença GPL, o que significa que qualquer pessoa tem o direito de usar, copiar e modificar o sistema livre de pagamento de *royalty*. Essas características, entretanto, não foram usadas nessa dissertação.

Outro motivo determinante na escolha do GNU/Linux foi devido a necessidade do FIDe, que usa os recursos do sistema operacional. A ferramenta é baseada nos mecanismos de depuração oferecidos pelo Linux.

5.3 O SGBD InterBase

O InterBase foi originalmente concebido e criado por um grupo de ex-empregados da Digital Equipment Corporation (DEC) com o desejo de produzir um SGBD relacional inovador que oferecesse maiores benefícios substanciais que outros bancos de dados existentes. O produto nasceu em 1985 como Groton Database Systems e depois foi renomeado para InterBase. Em 1991 foi comprado pela Ashton Tate. A Borland adquiriu o InterBase em 1992 como parte da posse da Ashton Tate [BOR 2000].

O SGBD InterBase é utilizado amplamente em ambiente corporativo com plataforma cliente/servidor. Um grande número de empresas e a maioria do desenvolvedores de aplicações em Delphi utilizam o InterBase como gerenciador de banco de dados [INT 01]. Foi este o motivo principal da escolha do InterBase como SGBD alvo desta dissertação.

5.3.1 Características do SGBD InterBase

O InterBase é um SGBD relacional que possui diversas características que o diferenciam consideravelmente de seus concorrentes. Tradicionais SGBDs comerciais (Sybase, SQL Server, Oracle, Progress) utilizam mecanismos de *log* e de *checkpointing* para obter um estado consistente após a detecção de uma falha. Estes mecanismos geralmente geram uma sobrecarga de processamento e são potencialmente lentos [BOR 2000].

O InterBase não usa *logs* de transação e *checkpointing* para manter informações sobre transações que ainda não foram escritas mas obtiveram *commit*. Ao invés disto, mantém informações em TIP (Transaction Information Pages). Esta informação significa o estado de qualquer transação: ativa, concluída, desfeita, preparada (para *commit* de duas fases). No caso de uma falha de sistema, tão logo o servidor for posto *on-line*, o InterBase automaticamente busca nas TIPs por transações que não obtiveram *commit* (*uncommitted*). Qualquer registro encontrado em estado *uncommitted* é desfeito. Este processo geralmente leva menos de um segundo para a maioria das bases de dados [BOR 2000].
Cada registro do TIP tem suas versões anteriores ligadas à transação. Após um *crash*, transações falham, deixando seu estado registrado no TIP como ativo. Quando outra transação vier a ler um registro criado por uma destas transações que falharam (mortas), ela ignora a versão primária e lê a primeira versão anterior *committed.* Quando outra transação tenta atualizar um registro criado por uma das transações mortas, ela testa o estado daquela transação através do identificador de bloqueio que toda transação possui. Se existe bloqueio, então a transação está morta. A transação atual configura o estado da transação morta de 0 para 3 (de ativo para preparada), e descarta o dado por ela criado [BOR 2000].

O InterBase utiliza um arquivo único, monolítico, com extensão **gdb,** como base de dados, onde são dispostas as tabelas, os índices, o esquema, e todas as demais informações relativas a base de dados. O utilitário de manutenção de bases de dados é o **gfix**. Pode-se executar uma variedade de tarefas de manutenção de bases de dados, incluindo *garbage collection*, reparação e recuperação de transações "limbo". Para escritas *bufferizadas* utiliza-se o comando **gfix -w async teste.gdb** e para escritas forçadas usa-se o comando **gfix -w sync teste.gdb**, sendo que **teste.gdb** é o nome da base de dados das tabelas que são o alvo das injeções de falhas.

5.3.2 Detecção e Recuperação de Erros no InterBase

O InterBase utiliza uma arquitetura multi-generacional. Isto significa que múltiplas versões de registros de dados são armazenados diretamente nas páginas do banco de dados. Quando um registro é modificado (alterado ou deletado), o InterBase mantém uma cópia do estado anterior do registro e cria uma nova versão do registro modificado (*shadow paging*).

Os defeitos gerados por erros de I/O do sistema operacional (detectados pelo valor de retorno do código de estado) são através de corrupção de dados (detectados através de verificações em todo o código) ou por erros de sintaxe (também detectados por verificações no código, mas em diferentes partes do mesmo). Erros que indicam a possibilidade de corrupção de dados tem um manipulador de erros que marca um *bit* do bloco de conexão, mostrando que esta conexão é suspeita. Qualquer requisição para esta conexão terá este *bit* identificado com a mensagem "*can't recover after bug check*".

Recuperação no InterBase é realizada quando há uma tentativa de conexão com uma base de dados e nesta conexão há a indicação de que ela seja suspeita. Desta forma, recuperação significa desfazer as transações que falharam (Undo). Se, posteriormente, alguma nova transação tentar alterar ou deletar um registro que foi inserido, alterado ou deletado por uma transação que falhou, esta nova transação analisa a transação anterior e compreende que ela falhou. Neste ponto, a nova transação desfaz o trabalho realizado pela transação anterior, restaura a versão de dados anterior, se for conveniente, e realiza as mudanças propostas por ela.

No caso de uma falha de sistema, tão logo o servidor seja posto *on-line* através de uma conexão e houver a indicação que esta conexão seja suspeita, o InterBase automaticamente busca nas TIPs por transações *uncommitted.* Qualquer registro encontrado em um estado *uncommitted* é *rolled back* e o sistema é imediatamente disponibilizado. Segundo o fornecedor, no InterBase restaurações automáticas após uma falha de *crash* levam tipicamente menos de um segundo, e não necessitam da intervenção do administrador como na maioria dos bancos de dados. Este requisito foi o fator chave na seleção do InterBase para o tanque de guerra Abrams M1 das forças armadas Norte Americanas. Quando o canhão do tanque dispara, ele cria uma enorme onda eletromagnética que causa o *crash* do computador do tanque. Tão logo o computador é reinicializado, o InterBase por sí mesmo restaura-se e é imediatamente e instantaneamente disponibilizado. Este tipo de capacidade de restauração, que não é oferecida por nenhum outro SGBD relacional, garante que o banco de dados esteja sempre disponível, mesmo em situações drásticas. Portanto, recuperação no InterBase é cooperativo e gradual e não preemptivo e imediato. Além disto, recuperação no InterBase executa apenas Undo [BOR 2000].

Por *default*, o InterBase executa escritas *bufferizadas*, também chamadas de escritas assíncronas. Diferentemente de escritas forçadas (*forced writes*), quando executa escritas *bufferizadas*, ele não escreve fisicamente dados no disco até que um evento pré-definido ocorra. Este evento ocorre quando uma certa quantidade de informação é coletada para uma escrita, um evento associado tenha ocorrido, ou um certo intervalo de tempo tenha passado. Se escritas forçadas não estão disponíveis, igualmente o InterBase

executa uma escrita interna. Os dados podem não ser fisicamente escritos em disco, porque o sistema operacional utiliza *buffers* para escrita em disco. Se ocorrer uma falha de sistema antes da escrita de dados em disco, então informações podem ser perdidas. Executando-se escritas forçadas garante-se integridade dos dados e segurança, mas perde-se desempenho, isto é, operações que envolvem modificações de dados serão mais lentas. [BOR 2000].

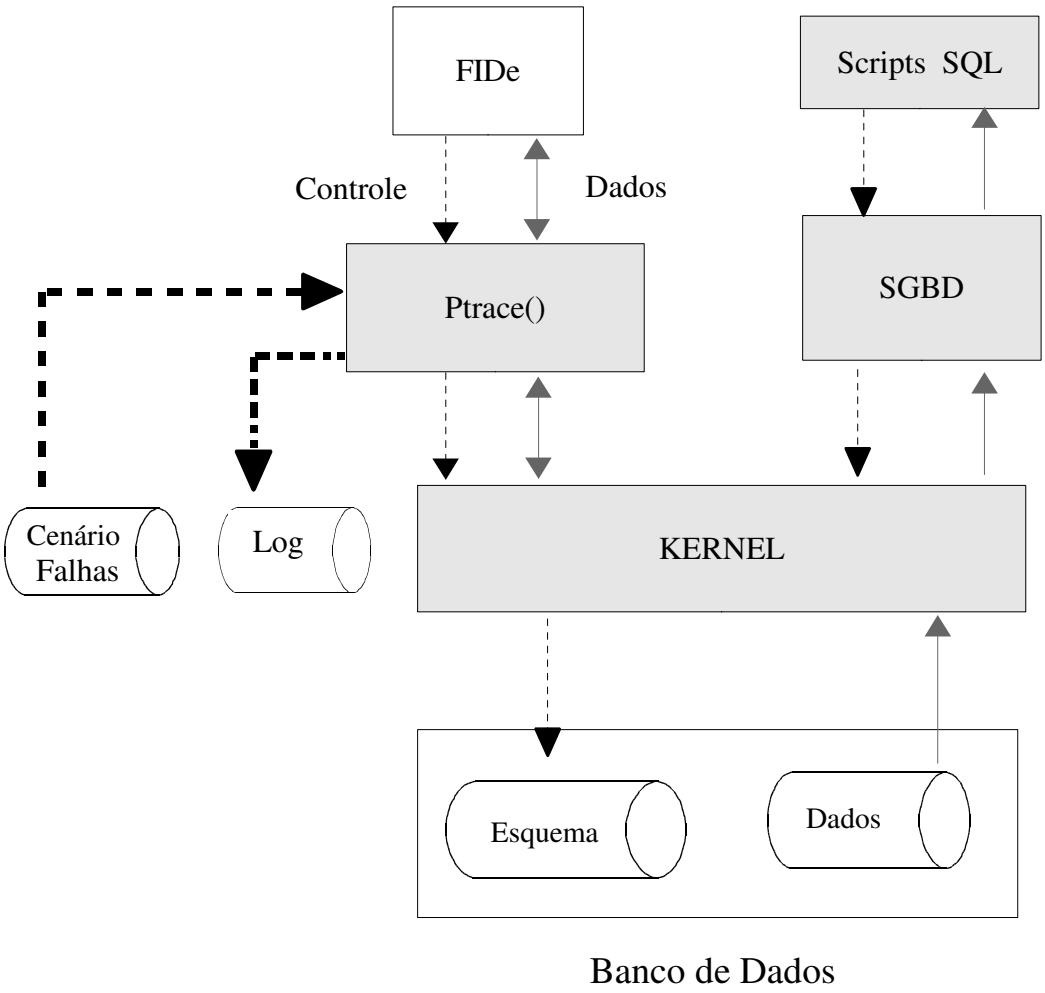

FIGURA 5.1 – Componentes da Plataforma.

5.4 Metodologia Utilizada

Alguns eventos podem gerar problemas nas estruturas do banco de dados (término anormal de uma aplicação do banco de dados: isto não afeta a integridade do banco de dados, e erros de escrita no sistema operacional ou hardware: estes erros geralmente criam problemas com a integridade do banco de dados). A partir destas considerações, optou-se, neste trabalho, por três enfoques distintos, com o objetivo de gerar a maior quantidade possível de ocorrências de falhas, reais através da primeira e segunda técnica

e simuladas através da terceira técnica, que pudessem afetar o ambiente do banco de dados. Estas falhas possibilitam validar o comportamento e a eficiência dos mecanismos de detecção e recuperação de falhas e os tempos de indisponibilidade após falhas.

O InterBase possibilita a execução de escritas *bufferizadas*, também chamadas de escritas assíncronas, ou de escritas diretamente em disco (*forced writes*). Se ocorrer uma falha de sistema antes da escrita de dados em disco, então informações podem ser perdidas. Executando-se escritas forçadas garante-se integridade dos dados e segurança, mas perde-se desempenho, isto é, operações que envolvem modificações de dados serão mais lentas [BOR 2000]. Desta forma, os experimentos com cada um dos enfoques e com cada uma das técnicas utilizadas foram realizados com escritas *bufferizadas* e com escritas forçadas em disco.

A metodologia por mim utilizada neste trabalho consiste em injetar falhas, deliberadamente, no ambiente do SGBD, no momento que um processo realiza atualizações nos dados de tabelas do banco de dados. Foram utilizados dois enfoques:

- experimentos de injeção de falhas sem ferramenta;
- experimentos de injeção de falhas com ferramenta.

5.4.1 Experimentos sem Ferramenta

Com o enfoque de injeção de falhas sem ferramenta, foram utilizadas duas técnicas distintas. A primeira técnica consiste em eliminar, manualmente, através do comando *kill* do sistema operacional, um processo pai que contém um processo filho, que realiza atualizações (inserções, alterações e deleções) em dados de tabelas do banco de dados. A segunda técnica consiste no *reset* geral do equipamento (manual) , no momento que um processo está sendo executado e realiza atualizações no banco de dados.

O comando *kill* (ou duplo **ctrl-c**) é utilizado para finalizar um processo que está rodando em *background*. Normalmente o comando *kill,* que é controlado pelo sistema operacional e corresponde a uma *syscall* (número 37), termina o processo com segurança, dando ao processo a chance de gravar arquivos de *backup*, liberar recursos do sistema e concluir sub-processos. Com a opção -9 termina um processo imediatamente. Nesta situação pode deixar sub-processos rodando e causar problemas com recursos do sistema. A opção -9 deve apenas ser utilizada como ultimo recurso [FOC 01].

De modo oposto, o *reset* geral do sistema é um evento externo, não controlado, assíncrono ao sistema operacional e pode "surpreender" o próprio sistema operacional num estado inconsistente, tendo dessa forma maior chance de emular falhas como queda de energia ou falhas de componentes de hardware vitais, que levam a *crash*.

Nos experimentos sem ferramenta de injeção de falhas foi utilizada a seguinte sequência de passos:

1) em modo terminal abria-se uma sessão *isql* (que permite execução de comandos ou *scripts* SQL);
2) executava-se um comando de conexão com o banco de dados (que permite acesso aos dados da tabela que compõe o banco de dados);
3) executava-se um *script* SQL, que realizava operações de seleção, inserção, alteração, ou deleção de registros nas tabelas do banco de dados;
4) em seguida, em diferentes tempos após o início da execução do *script*, executava-se ou o comando *kill* (primeira técnica) ou o *reset* geral do sistema (segunda técnica);
5) após isto, com o objetivo de verificar se o procedimento anterior (*kill* ou *reset*) havia afetado a consistência e/ou a integridade dos dados, fazia-se uma nova conexão com o banco de dados, apurava-se o tempo de indisponibilidade e executava-se um *script* SQL de leitura e contagem ou exibição em tela, ou alteração, de todos os dados da tabela alvo da injeção de falhas e que seria possivelmente afetada. Se a nova operação de leitura e/ou atualização ocorria de forma normal, isto é, sem a indicação de erro, conclui-se que o mecanismo de recuperação obteve sucesso, ou recuperando efetivamente (desfazendo as transações que não obtiveram *commit*) ou mascarando os erros encontrados. Se a operação não foi concluída de forma normal, o mecanismo de detecção de erros informa que houve a identificação do erro. Adicionalmente, fazia-se a leitura do arquivo **interbase.log**. Este arquivo registra, de forma detalhada, todos os erros que ocorrem com o banco de dados, indicando também as páginas do banco de dados que são órfãs (páginas que sofreram atualizações e não obtiveram *commit* - não devem utilizar espaço em disco e devem retornar como espaço livre).

O conteúdo abaixo é um fragmento do arquivo **interbase**.**log**:

athenas Mon Jul 31 17:27:13 2001
Database: /home/paulo/interbase/i686_V4.0G/bin/teste.gdb
internal gds software consistency check (cannot find record back version (291))

athenas Mon Jul 31 17:37:06 2001
Database: /home/paulo/interbase/i686_V4.0G/bin/teste.gdb
internal gds software consistency check (cannot find record back version (291))

athenas Mon Jul 31 17:45:29 2001
Database: /home/paulo/interbase/i686_V4.0G/bin/teste.gdb
internal gds software consistency check (cannot find record back version (291))

athenas Mon Jul 31 17:54:35 2001
Page 608 is an orphan

athenas Mon Jul 31 17:54:35 2001
Page 609 is an orphan

5.4.2 Experimentos com Ferramenta

Com o enfoque de injeção de falhas com ferramenta, a técnica fez uso da ferramenta de injeção de falhas - o FIDe, para injetar falhas que simulem falhas transientes de hardware e que gerem situações que podem ocorrer no ambiente do SGBD (alterar o conteúdo dos registradores, dos dados em memória, da área de código e as informações (estrutura *user*) do processo em depuração) e que podem afetar a consistência e integridade dos dados.

Nos experimentos com a ferramenta de injeção de falhas foi utilizada a seguinte sequência de passos:

1) Monitorados pelo depurador *strace* executavam-se operações de seleção, inclusão, alteração e deleção sobre dados do banco de dados, sem o injetor de falhas, e verificava-se quais *syscalls* eram utilizadas (através de arquivo gerado pelo depurador *strace)*;
2) a partir de informações obtidas pelo arquivo gerado pelo depurador, montava-se o cenário de falhas que se desejava injetar e seus respectivos parâmetros (capítulo 4.4);
3) executava-se o FIDe, que utiliza o cenário de falhas previamente montado, passando-se os parâmetros necessários (qual arquivo serviria como cenário de falhas, em qual aplicação seria feita a injeção de falhas, qual o arquivo alvo da injeção de falhas - ex.: ./fide -d fide.rules ./isql teste.gdb). Este comando, abria uma sessão *isql* (que permitia a execução de comandos ou *scripts* SQL) e executava um comando de conexão com o banco de dados teste.gdb (que permitia acesso aos dados da tabela que compõe o banco de dados);
4) a partir deste ponto, podia-se verificar se a injeção de falhas já havia ocorrido, se a mesma afetara ou não a conexão com o banco de dados, a integridade dos dados e as estruturas do banco de dados;
5) se nenhum erro de conexão ou de consistência com o banco de dados havia ocorrido, executava-se um *script* SQL, que realizava operações de seleção, inserção, alteração, ou deleção de registros nas tabelas do banco de dados;
6) saia-se da sessão *isql* (com *quit*, fazendo-se roll back das transações realizadas ou com *exit*, estabelecendo *commit* das operações realizadas);
7) após isto, com o objetivo de verificar a consistência e a integridade dos dados, fazia-se uma nova conexão com o banco de dados, sem a ferramenta de injeção de falhas, apurava-se o tempo de indisponibilidade (caso houvesse) e executava-se um *script* SQL de leitura e contagem de todos os dados da tabela alvo da injeção de falhas e que poderia ter sido afetada. Se a nova operação de leitura e/ou atualização era encerrada de forma normal, isto é, sem a indicação de erro, concluia-se que o mecanismo de recuperação obtivera sucesso, ou recuperando efetivamente (desfazendo as transações que não obtiveram *commit*) ou mascarando os erros encontrados. Se a operação não era concluída de forma normal, o mecanismo de detecção de erros informava o erro detectado ou havia o travamento do processo. Adicionalmente, fazia-se a leitura do arquivo **interbase.log**. Este arquivo registra, de forma detalhada, todos os erros que ocorrem com o banco de dados.

5.4.3 Medidas Adotadas

Como medidas práticas de confiabilidade poderiam ser usadas a taxa de defeitos (número de defeitos em um dado período de tempo), Mean Time To Failure (MTTF - tempo esperado até a primeira ocorrência de defeito), Mean Time To Repair (MTTR - Tempo Médio para Reparo do sistema) e Mean Time Between Failure (MTBF - tempo médio entre as falhas do sistema). Em função da metodologia adotada (injeção deliberada de falhas), optou-se por descartar a taxa de defeitos, MTTF e MTBF. As medidas de confiabilidade adotadas neste trabalho se resumem a taxa de cobertura de detecção de erros, taxa de cobertura de recuperação de erros e MTTR.

A disponibilidade e confiabilidade do SGBD pode ser avaliada pelo impacto das falhas injetadas sobre a base de dados, segundo os tipos definidos no modelo de falhas. A confiabilidade pode ser avaliada no nível de tabelas afetadas, pela verificação da integridade da estrutura interna dos arquivos de dados. Já a disponibilidade pode ser medida pelo tempo médio de recuperação do BD diante da falha, o tipo de recuperação sofrida pelo BD (total ou parcial) e pelo esforço de recuperação, automático ou pelo administrador do BD (manual).

Quando foi utilizado o comando *kill* (ou duplo **ctrl-c**) e a injeção de falhas através do FIDe, os tempos de indisponibilidade, foram medidos de seguinte maneira: após a injeção de falhas, fazia-se uma nova conexão com o banco de dados (comando *connect* de uma sessão *isql)* que poderia ter sido afetado pela injeção de falha e, neste ponto, cronometra-se manualmente o tempo de indisponibilidade (tempo de espera até que se pudesse utilizar novamente os dados das tabelas).

Quando foi utilizado a técnica com *reset* geral, os tempos de indisponibilidade foram medidos da seguinte maneira: após o *reset* geral, havia uma indisponibilidade longa, até que o sistema operacional fosse reconfigurado e recarregado. Após isto, fazia-se uma nova conexão com o banco de dados (comando *connect* de uma sessão *isql)* que havia sido afetado pelo *reset* e, neste ponto, cronometra-se manualmente o tempo de indisponibilidade (tempo de espera até que se pudesse utilizar novamente os dados das tabelas).

5.5 A Carga de Trabalho

A carga de trabalho foi gerada através da execução de *scripts* de inserção de registros, escritos em SQL *(Structured Query Language)*, dentro de uma sessão *isql*. Os testes sem a ferramenta de injeção de falhas foram realizados em um banco de dados que continha tabelas com diferentes quantidades de registros lógicos (60.000, 180.000, 240.000, 300.000 ou 450.000). Cada um destes registros lógicos ocupava 78 *bytes,* e eram compostos de atributos de tipo *smallint, varchar* ou *numeric* de pequeno tamanho. Estas tabelas eram simples e não possuíam relacionamentos (ausência de chaves estrangeiras). A menor tabela tinha 4,6 MB e a maior tabela 22,3 MB. Foram realizadas transações de inserção de 60.000 registros (*insert*), alteração de 240.000, 300.000 e 450.000 registros (*update*) e exclusão de 300.000 e 450.000 registros (*delete*). Também foram realizadas operações de seleção de partes ou de todos os registros de uma tabela.

Os testes com a ferramenta FIDe foram realizados em um banco de dados que continha tabelas com pequeno número de registros lógicos (de 10 a 80). Cada um destes registros lógicos ocupava 78 *bytes,* e eram compostos de atributos de tipo *smallint, varchar* ou *numeric* de pequeno tamanho. Estas tabelas eram simples e não possuíam relacionamentos (ausência de chaves estrangeiras). O banco de dados como um todo tinha em torno de 220 KB. Foram realizadas transações de inserção de 30 registros, alteração de 30 a 80 registros e exclusão de 30 registros. Também foram realizadas operações de seleção de partes ou de todos os registros de uma tabela.

5.6 Modelo de Falhas

Há 3 tipos de falhas que são mais importantes em sistemas de bancos de dados centralizados, conhecidas como **falhas de transação, falhas de sistema e falhas de mídia.** Uma falha de transação ocorre quando uma transação aborta. Uma falha de sistema refere-se a perda ou corrupção do conteúdo de armazenamento volátil (memória principal - isto pode ocorrer quando falta energia ou quando o sistema operacional falha). Embora uma falha de sistema operacional possa não corromper toda a memória principal, é normalmente muito difícil determinar quais partes foram realmente corrompidas por uma falha. Deste modo, geralmente assume-se o pior e reinicializa-se toda a memória principal. Uma falha de mídia ocorre quando qualquer parte do armazenamento não volátil é destruído (por ex., se alguns setores do disco são danificados). As técnicas usadas para suportar falhas de mídia são conceitualmente similares àquelas usadas para suportar falhas de sistema [BER 87].

Existem diversas razões possíveis para que uma transação falhe no decorrer de sua execução. Algumas destas razões são [ELM 89]:

- Queda de sistema (*crash*): Um erro de hardware ou software ocorre no sistema de computação durante a execução da transação. Se o hardware falhar, o conteúdo da memória interna pode ser perdida.

- Um erro de transação ou sistema: Algumas operações na transação podem causar uma falha, tais como *overflow* de inteiro ou divisão por zero. Defeitos em transação podem também ocorrer devido a valores de parâmetros errados ou devido a um erro de lógica na programação. Também, o usuário pode interromper a tansação durante sua execução (ex. Control C em ambiente Unix ou VAX/VMS).

- Erros locais ou condições de exceção detectadas pela transação: Durante a execução de transações, podem ocorrer certas condições que necessitem o cancelamento dea transação. Por exemplo, dados necessários para a transação podem não ser encontrados. Isto pode ser feito por um *Abort* programado na própria transação.

- Forçado pelo controle de concorrência: o método de controle de concorrência pode decidir abortar a transação, sendo reinicializado mais tarde, devido a

violação na serialização ou devido a diversas transações estarem em estado de *deadlock*.

- Defeitos em disco: alguns blocos de disco podem perder seus dados devido a mal funcionamento de leituras e escritas ou devido a quebra da cabeça de leitura/escrita do disco.

Nas operações diárias, um banco de dados é, algumas vezes, exposto a eventos que podem gerar determinados problemas nas estruturas do banco de dados. Estes eventos incluem [BOR 99]:

- **Término anormal de uma aplicação do banco de dados:** Isto não afeta a integridade do banco de dados. Quando uma aplicação é cancelada, dados que obtiveram *commit* são preservados e atualizações que não obtiveram *commit* são desfeitas. Se o InterBase já escolheu e determinou uma página para as atualizações que não obtiveram *commit* esta página pode ser considerada uma página orfã. Páginas orfãs não utilizam espaço de disco e devem ser retornadas como espaço livre.

- **Erros de escrita no sistema operacional ou hardware:** Estes erros geralmente criam problemas com a integridade do banco de dados. Erros de escrita podem gerar quebra ou perda das estruturas de dados, tais como uma página do banco de dados ou índices. Esta corrupção das estruturas de dados podem frustrar a recuperação de dados que obtiveram *commit*.

Um banco de dados somente estará em um estado consistente se considerarmos os possíveis lugares de residência de dados (disco, memórias) ou de valores que controlam e monitoram as operações sobre estes dados (registradores de CPU). Portanto, qualquer falha num destes locais pode levar o banco de dados a um estado inconsistente.

Numa falha de *crash* o sistema falha através de uma parada. Uma vez que houve uma parada, o sistema permanece neste estado [SCH 93]. O modelo de falhas proposto para este trabalho refere-se a falhas de *crash* baseadas em corrupção ou perda de memória ou de registradores de CPU, parada de CPU ou *reset* geral. Pode-se classificar, também, a corrupção de dados em memória e erros em operações de leitura e escrita em disco como falhas de sistema.

6 Experimentos e Resultados

Neste capítulo são descritos com detalhes cada um dos enfoques de injeção de falhas com suas respectivas técnicas, o conjunto de testes realizados e os resultados obtidos. Através destes testes, foi possível observar o comportamento do banco de dados InterBase na ocorrência de falhas, validar o mecanismo de detecção de falhas, validar o mecanismo de recuperação e apurar os tempos necessários para disponibilizar o banco de dados aos usuários após a ocorrência de uma falha que tenha provocado um *crash*

6.1 A Primeira Técnica de Experimentação

A primeira técnica consiste em eliminar, de forma manual, através do comando *kill,* um processo pai que contém um processo filho que estabelece atualizações (inserções, alterações e deleções) no banco de dados **com** e **sem** o parâmetro *forced writes*. Deve-se observar ainda, que o momento em que o comando *kill* foi executado, variou aleatoriamente em cada um dos testes.

Como descrito no capítulo anterior, um banco de dados está sujeito a eventos que podem comprometer suas estruturas. Esta técnica foi utilizada visando validar os mecanismos de detecção e recuperação do SGBD InterBase, simulando um tipo de erro relativamente simples e que teoricamente não deveria afetar a integridade dos dados e das estruturas do banco de dados

Foi definido o número de 50 execuções para cada tipo de teste porque é um número considerável e, como não havia variação nos resultados (exceção feita ao teste número 4) pressupõe-se que, se houvessem mais execuções, o resultado seria o mesmo.

No total foram 350 execuções, divididas em 6 testes distintos, descritos a seguir e acompanhados de seus resultados e cujo resumo é mostrado na tabela 6.1.

- Teste 01:
  - Descrição : execução de um processo que lê 60.000 registros de uma tabela e os insere em outra tabela que contém 240.000 registros **sem** *forced writes.*
  - Número de execuções: 50
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 100% dos casos, de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado.

- Teste 02:
  - Descrição: execução de um processo que lê 60.000 registros de uma tabela e os insere em outra tabela que contém 240.000 registros **com** *forced writes.*
  - Número de execuções: 50
  - Resultado: O InterBase obteve sucesso no processo de recuperação em 100% dos casos, de maneira rápida e eficiente, isto é, o banco de dados foi

prontamente disponibilizado.

- Teste 03:
  - Descrição: execução de um processo que altera um campo tipo *varchar* de 50 bytes em 300.000 registros de uma tabela, **sem** *forced writes.*
  - Número de execuções: 50
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 100% dos casos, de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado.

- Teste 04:
  - Descrição: execução de um processo que altera um campo tipo *varchar* de 50 bytes em 300.000 registros de uma tabela, **com** *forced writes.*
  - Número de execuções: 100
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 99 execuções (99 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. O mecanismo de detecção, numa única execução, não foi eficiente, uma vez que nenhuma mensagem de erro foi exibida. A tabela quando acessada, após alguns segundos travou.

- Teste 05:
  - Descrição: execução de um processo que exclui todos os registros de uma tabela que continha 300.000 registros **sem** *forced writes.*
  - Número de execuções: 50
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 100% dos casos, de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado.

- Teste 06:
  - Descrição: execução de um processo que exclui todos os registros de uma tabela que continha 300.000 registros **com** *forced writes.*
  - Número de execuções: 50
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 100% dos casos, de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado.

Os dados da tabela 6.1 foram obtidos através dos testes 01 a 06 descritos acima, com a técnica de eliminação de um processo em execução através do comando *kill*. Visando facilitar a visualização dos resultados, as figuras 6.1 e 6.2 mostram os mesmos dados desta tabela.

TABELA 6.1 – Eficiência da recuperação matando processo pai (Kill).

Operação Realizada	***# de Execuções***	***# de Recuperação com Sucesso***	***% Sucesso***	***# de Recuper. com Erro***	***% Erro***
Inserção de 60.000 registros - **sem FW**	50	50	100,00%	0	0,00%
Inserção de 60.000 registros - **com FW**	50	50	100,00%	0	0,00%
Alteração de 240.000 registros - **sem FW**	50	50	100,00%	0	0,00%
Alteração de 300.000 registros - **com FW**	100	99	99,00%	1*	1,00%
Deleção de 300.000 registros - **sem FW**	50	50	100,00%	0	0,00%
Deleção de 300.000 registros - **com FW**	50	50	100,00%	0	0,00%

(*) = Erro não detectado

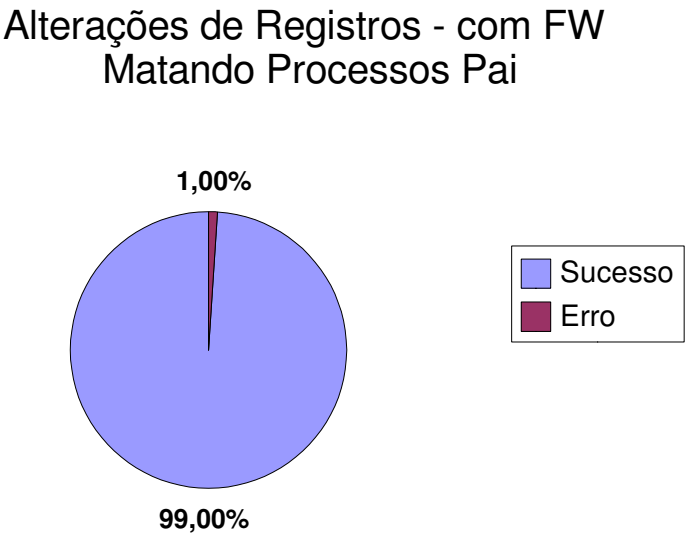

FIGURA 6.1 - Gráfico da eficiência da recuperação - Matando Processo Pai - Alteração com FW.

Como mostram os dados da figura 6.1, quando foi utilizada a técnica de matar o processo pai que continha um processo filho que alterava um campo tipo *varchar* de 300.000 registros de uma tabela, **com** *forced writes,* de um total de 100 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em 99% dos casos. Em uma ocorrência (1%) o mecanismo não obteve sucesso (este erro não foi identificado pelo mecanismo de detecção. Só foi possível detectá-lo quando houve nova tentativa de acesso a tabela e houve o travamento do processo, sem identificação ou mensagem do que havia ocorrido).

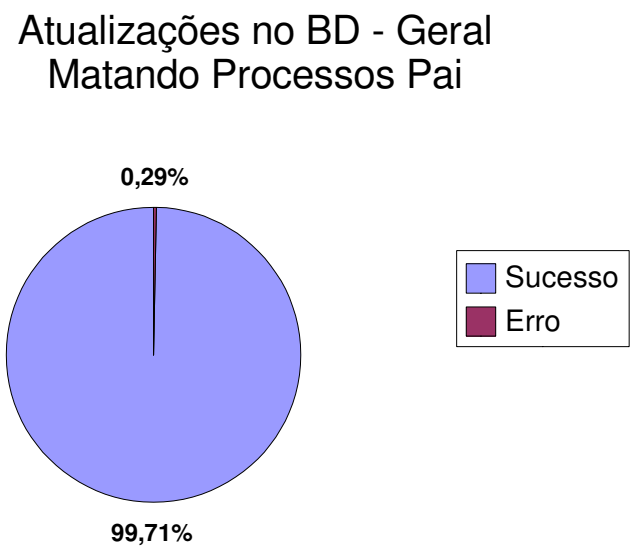

FIGURA 6.2 - Gráfico da eficiência da recuperação - Matando Processo Pai - Média Geral - **com** e **sem** *forced writes.*

Como mostram os dados da figura 6.2, de um total de 350 experimentos em que foi utilizada a técnica de matar o processo pai que continha um processo filho que realizava atualizações no banco de dados (inserções, alterações e deleções), **com** e **sem** *forced writes,* o mecanismo de recuperação do InterBase mostrou-se eficiente em 349 dos casos (99,71%) e em apenas uma ocorrência o mecanismo não obteve sucesso (0,29%).

Conclusão: Os testes realizados com esta técnica ficaram em conformidade com o previsto no modelo de falhas (capítulo 5.6 - Término anormal de uma aplicação do banco de dados: Isto não afeta a integridade do banco de dados. Quando uma aplicação é cancelada, dados que obtiveram *commit* são preservados e atualizações que não obtiveram *commit* são desfeitas). Nos 350 experimentos em que foi utilizado o comando *kill*, mesmo com a opção -9, como mostram os dados da tabela 6.1, em apenas 1 caso (0,29%) houve problema com as tabelas do banco de dados alvo. Neste caso, os mecanismos de detecção e de recuperação de erros não foram eficientes, e só se soube do problema poque em nova tentativa de acesso ao banco de dados, houve o travamento da operação. A cobertura de detecção em função das falhas detectadas e falhas injetadas foi de 99,71% e a cobertura de recuperação em função das falhas injetadas também foi de 99,71%. Neste único caso, não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente.

6.2 A Segunda Técnica de Experimentação

A segunda técnica consiste no *reset* geral do equipamento, de forma manual, no momento em que um processo estava sendo executado e estabelecia atualizações (inserções, alterações e deleções) no banco de dados, **com** e **sem** *forced writes*. Deve-se observar ainda, que o momento em que o *reset* geral foi realizado, variou aleatoriamente em cada um dos experimentos.

Como descrito no capítulo 5.6, um banco de dados está sujeito a eventos que podem comprometer suas estruturas. Esta técnica foi utilizada visando validar os mecanismos de detecção e recuperação do SGBD InterBase, simulando um tipo de erro considerado grave e que poderia afetar a integridade dos dados e das estruturas do banco de dados

No total foram executados 75 experimentos, divididos em 7 diferentes testes, descritos a seguir e acompanhados de seus resultados e cujo resumo é mostrado na tabela 6.2. Os experimentos realizados utilizando a segunda técnica foram em menor número porque o experimento consumia uma quantidade de tempo muito maior e porque poderia comprometer a integridade do ambiente de experimentação (equipamento, sistema operacional, banco de dados, dados e aplicações).

- Teste 01:
  - Descrição: execução de um processo que lê 60.000 registros de uma tabela e os insere em outra tabela que contém 240.000 registros, **sem** *forced writes.*
  - Número de execuções: 12
  - Resultados: O InterBase obteve sucesso no processo de recuperação em 9 execuções (75 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. Nas 3 execuções em que ocorreram erros (25 % dos casos), a tabela ficou danificada e inacessível. Nas tentativas de acessar o banco de dados, após o *crash*, foi exibida a seguinte mensagem de erro: **database file appear corrupt ( )**
    wrong page type
    page "xxxxx" is of wrong type (expected 7, found 5)

- Teste 02:
  - Descrição: execução de um processo que lê 60.000 registros de uma tabela e os insere em outra tabela que contém 300.000 registros, **com** *forced writes.*
  - Número de execuções: 10
  - Resultados:O InterBase obteve sucesso no processo de recuperação nas 10 execuções (100 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado.

- Teste 03:
  - Descrição: execução de um processo que lê 150.000 registros de uma tabela e os insere em outra tabela que contém 300.000 registros, **com** *forced writes.*
  - Número de execuções: 10
  - Resultados:O InterBase obteve sucesso no processo de recuperação nas 10 execuções (100 % dos casos), de maneira rápida e eficiente, isto é, o banco de

dados foi prontamente disponibilizado.

- Teste 04:
  - Descrição: execução de um processo que altera um campo tipo *varchar* de 50 bytes em 300.000 registros de uma tabela, **sem** *forced write.*
  - Número de execuções: 13
  - Resultados: O InterBase obteve sucesso no processo de recuperação em sete execuções (53,8 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. Nas seis execuções em que ocorreram erros (46,2 % dos casos), a tabela ficou danificada e inacessível. Nas tentativas de acessar o banco de dado, após o *crash*, foram exibidas as seguintes mensagens de erros :

    - **Internal gds software consistency check (cannot find record back version (291)) (** 3 vezes)
    - **Internal gds software consistency check (wrong record length (183))**
    - **Internal gds software consistency check (applied diferences will not fit in record (177))**
    - **database file appear corrupt ( )**
      wrong page type
      page "xxxxx" is of wrong type (expected 5, found 4)

- Teste 05:
  - Descrição: execução de um processo que altera um campo tipo *varchar* de 50 bytes em 300.000 registros de uma tabela, **com** *forced write.*
  - Número de execuções: 10
  - Resultados: O InterBase obteve sucesso no processo de recuperação em três execuções (30 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. Nas sete execuções em que ocorreram erros (70 % dos casos), a tabela ficou danificada e inacessível. Nas tentativas de acessar o banco de dado, após o *crash*, foram exibidas as seguintes mensagens de erros :

    - **Internal gds software consistency check (cannot find record back version (291)) (** 2 vezes)
    - **Internal gds software consistency check (wrong record length (183))**
    - **Internal gds software consistency check (applied diferences will not fit in record (177))**
    - **Internal gds software consistency check (cannot find record fragment (248))** (3 vezes)

- Teste 06:
  - Descrição: execução de um processo que deleta 450.000 registros de uma tabela, **sem** *forced writes.*
  - Número de execuções: 10
  - Resultados: O InterBase obteve sucesso no processo de recuperação em

apenas uma execução (10 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. Nas nove execuções em que ocorreram erros (90 % dos casos), a tabela ficou danificada e inacessível. Nas tentativas de acessar o banco de dado, após o *crash*, foram exibidas as seguintes mensagens de erros :

- **Internal gds software consistency check (cannot find record back version (291))**
- **Internal gds software consistency check (cannot find record fragment (248))** (2 vezes)
- **Internal gds software consistency check (decompression overran buffer (179))** (6 vezes)

- Teste 07:
  - Descrição: execução de um processo que deleta 350.000 ou 450.000 registros de uma tabela, **com** *forced writes.* Em alguns experimentos iniciou-se a deleção a partir do primeiro registro e em outros a partir do 100.001º registro.
  - Número de execuções: 10
  - Resultados: O InterBase obteve sucesso no processo de recuperação em apenas duas execuções (20 % dos casos), de maneira rápida e eficiente, isto é, o banco de dados foi prontamente disponibilizado. Nas oito execuções em que ocorreram erros (80 % dos casos), a tabela ficou danificada e inacessível. Nas tentativas de acessar o banco de dado, após o *crash*, foram exibidas as seguintes mensagens de erros :

    - **Internal gds software consistency check (cannot find record back version (291)) (** 4 vezes)
    - **Internal gds software consistency check (wrong record length (183))**
    - **Internal gds software consistency check (decompression overran buffer (179))** (3 vezes)

Os dados da tabela 6.2 foram obtidos através dos testes 01 a 07 descrito acima, com a técnica de *reset* geral. Visando facilitar a visualização dos resultados, as figuras 6.3 a 6.12 mostram os mesmos dados desta tabela.

TABELA 6.2 – Eficiência da recuperação com *reset* geral no equipamento.

Operação Realizada	***# de Execuções***	***# de Recuper. com sucesso***	***% de Sucesso***	***# de Recuper. com Erro***	***% de Erro***
Lê 60.000 regs.de uma tabela e os insere em outra tabela que contém 240.000 registros - **sem FW**	12	9	75,00%	3	25,00%
Lê 60.000 regs. de uma tabela e os insere em outra tabela que contém 300.000 registros - **com FW**	10	10	100,00%	0	0,00%
Lê 150.000 regs. de uma tabela e os insere em outra tabela que contém 300.000 registros - **com FW**	10	10	100,00%	0	0,00%
Alteração de um campo tipo *varchar* em 300.000 regs. de uma tabela – **sem FW**	13	7	53,80%	6	46,20%
Alteração de um campo tipo *varchar* em 300.000 regs. de uma tabela -**com FW**	10	3	30,00 %	7	70,00 %
Deleção de todos os registros de uma tabela (450.000) - **sem FW**	10	1	10,00 %	9	90,00 %
Deleção de 350.000 ou 450.000 regs. de uma tabela - **com FW**	10	2	20,00 %	8	80,00 %

➔ Em todos os casos houve a detecção do erro (uma mensagem foi exibida ao usuário quando de nova tentativa de acesso).

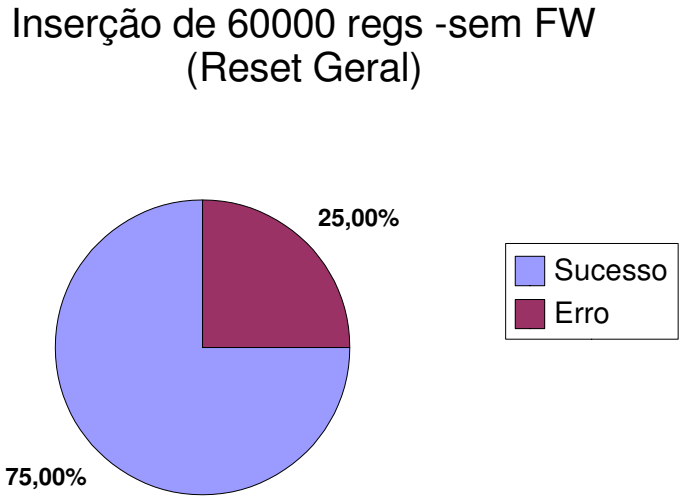

FIGURA 6.3 - Gráfico eficiência da recuperação - *Reset* Geral - inserção sem FW**.**

Como mostram os dados da figura 6.3, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo lia 60.000 registros de uma tabela e os inseria em outra tabela que já continha 240.000 registros, **sem** *forced writes.* De um total de 12 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em nove (75%) e não obteve sucesso nos outros três casos (25%).

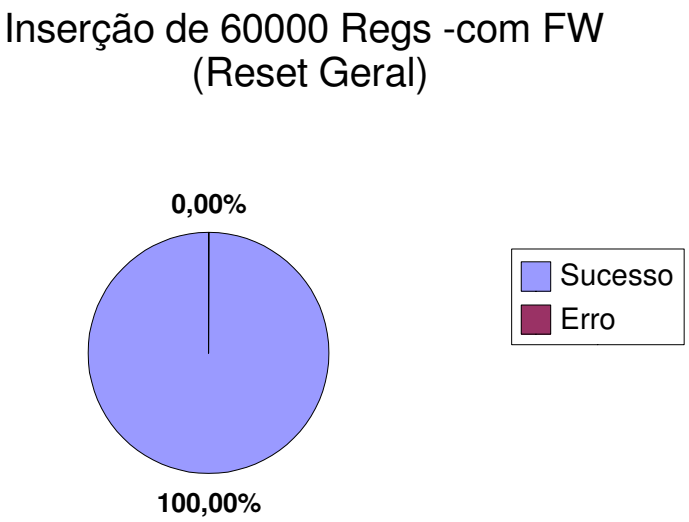

FIGURA 6.4 - Gráfico eficiência da recuperação - *Reset* Geral - inserção com FW**.**

Como mostram os dados da figura 6.4, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo lia 60.000 registros de uma tabela e os inseria em outra tabela que já continha 300.000 registros, **com** *forced writes.* De um total de 10 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em 100% dos casos.

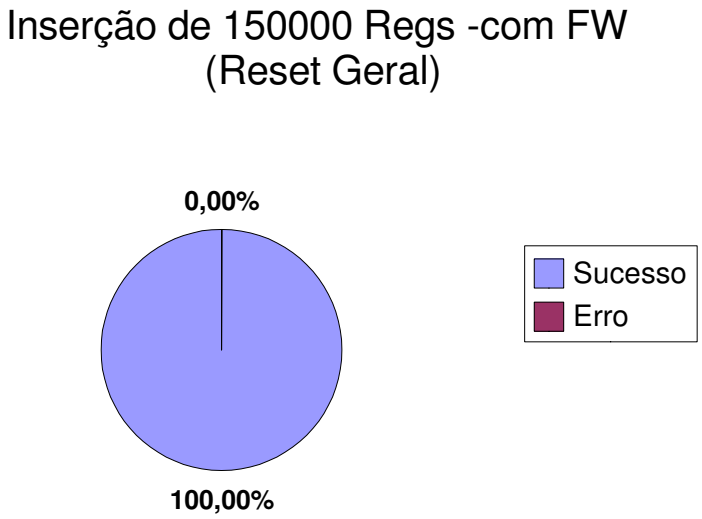

FIGURA 6.5 - Gráfico eficiência da recuperação - *Reset* Geral - inserção com FW.

Como mostram os dados da figura 6.5, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo lia 150.000 registros de uma tabela e os inseria em outra tabela que já continha 300.000 registros, **com** *forced writes.* De um total de 10 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em 100% dos casos.

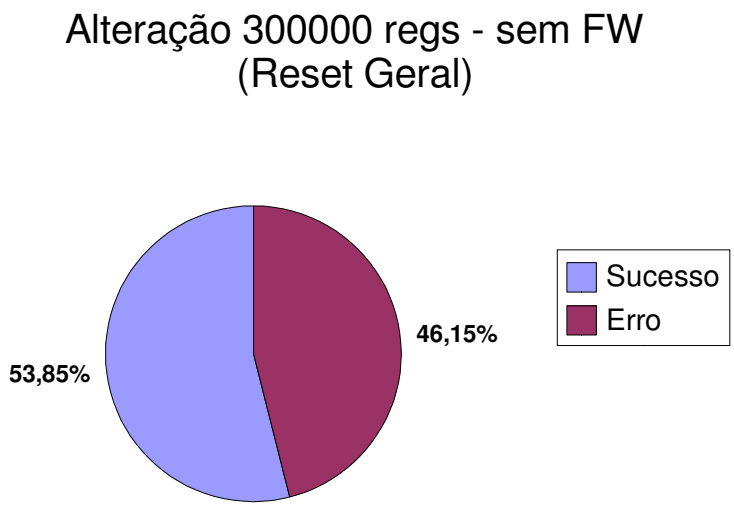

FIGURA 6.6 - Gráfico eficiência da recuperação - *Reset* geral - alteração sem FW.

Como mostram os dados da figura 6.6, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo alterava um campo tipo *varchar* de uma tabela com 300.000 registros, **sem** *forced writes.* De um total de 13 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em sete (53,80%) e não obteve sucesso nos outros seis casos (46,20%).

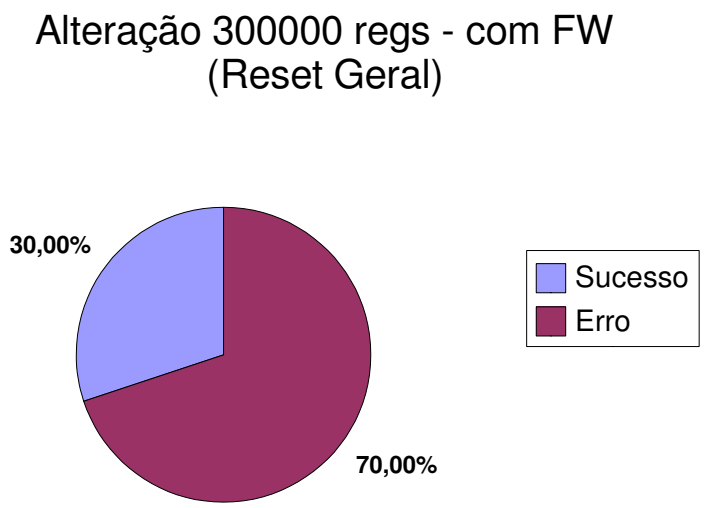

FIGURA 6.7 - Gráfico eficiência da recuperação - *Reset* geral - alteração com FW.

Como mostram os dados da figura 6.7, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo alterava um campo tipo *varchar* de uma tabela com 300.000 registros, **com** *forced writes,* de um total de 10 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em apenas três (30%) e não obteve sucesso nos outros sete casos (70%). Cabe ressaltar aqui, a ineficiência do mecanismo de recuperação, uma vez que foi utulizado o parâmetro *forced writes* e a expectativa deveria, por lógica, ser completamente diferente.

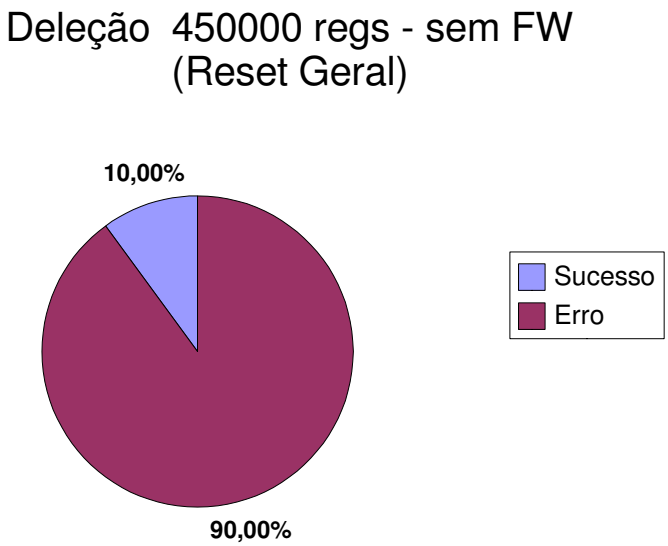

FIGURA 6.8 - Gráfico eficiência da recuperação - *Reset* geral - deleção sem FW.

Como mostram os dados da figura 6.8, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo deletava os registros de uma tabela com 450.000 registros, **sem** *forced writes.* De um total de 10 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em apenas um (10%) e não obteve sucesso nos outros nove casos (90%).

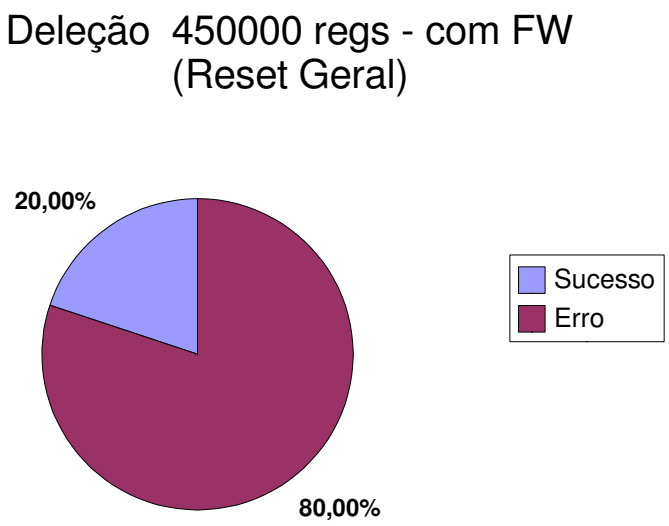

FIGURA 6.9 - Gráfico eficiência da recuperação - *Reset* geral - deleção com FW.

Como mostram os dados da figura 6.9, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo deletava os registros de uma tabela com 350.000 ou 450.000 registros, **com** *forced writes,* de um total de 10 experimentos, o mecanismo de recuperação do InterBase mostrou-se eficiente em apenas dois (20%) e não obteve sucesso nos outros oito casos (80%).

O mecanismo de recuperação do InterBase apresentou os piores resultados de eficiência, quando foi utilizada a técnica de *reset* geral com operações de deleção, mesmo com o parâmetro *forced writes* ativado.

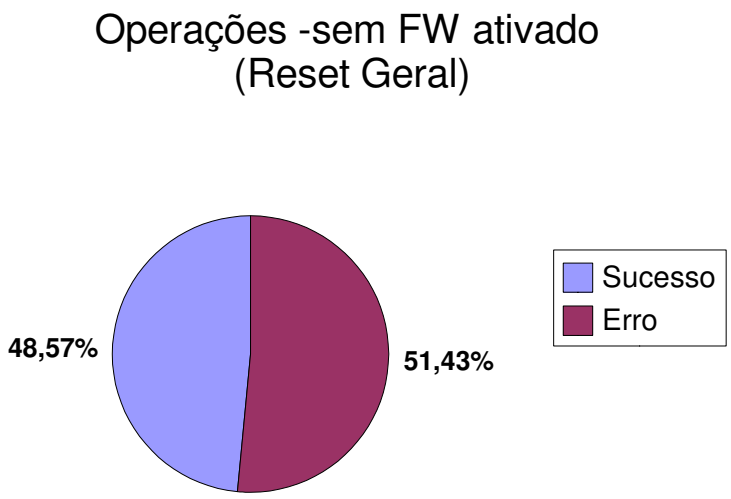

FIGURA 6.10 - Gráfico eficiência da recuperação - *Reset* geral - Média sem FW.

Como mostram os dados da figura 6.10, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo estabelecia atualizações (inserções, alterações e deleções) no banco de dados, **sem** *forced writes,* de um total de 35 execuções, o mecanismo de recuperação do InterBase mostrou-se eficiente em 17 casos e não obteve sucesso nos outros 18 casos. A cobertura de recuperação em função das falhas injetadas foi de 48,57% e a cobertura de detecção em função das falhas injetadas foi de 100%. Não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente.

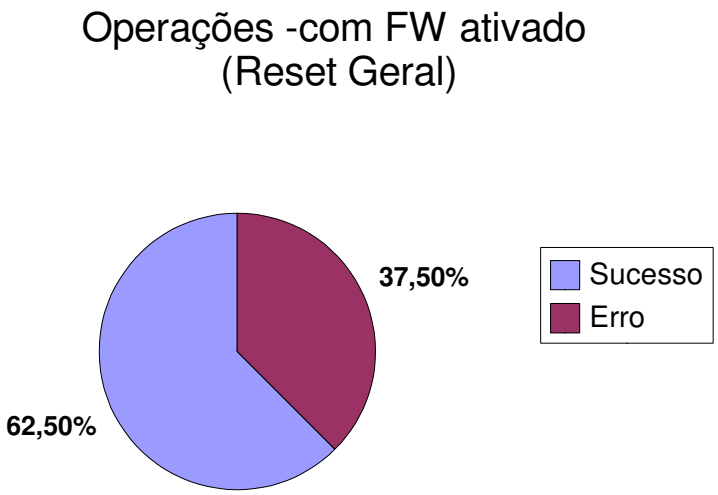

FIGURA 6.11 - Gráfico eficiência da recuperação - R*eset* geral - Média com FW.

Como mostram os dados da figura 6.11, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo estabelecia atualizações (inserções, alterações e deleções) no banco de dados, **com** *forced writes,* de um total de 40 execuções, o mecanismo de recuperação do InterBase mostrou-se eficiente em 25 casos e não obteve sucesso nos outros 15 casos. A cobertura de recuperação em função das falhas injetadas foi de 62,50% e a cobertura de detecção em função das falhas injetadas foi de 100%. Não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente.

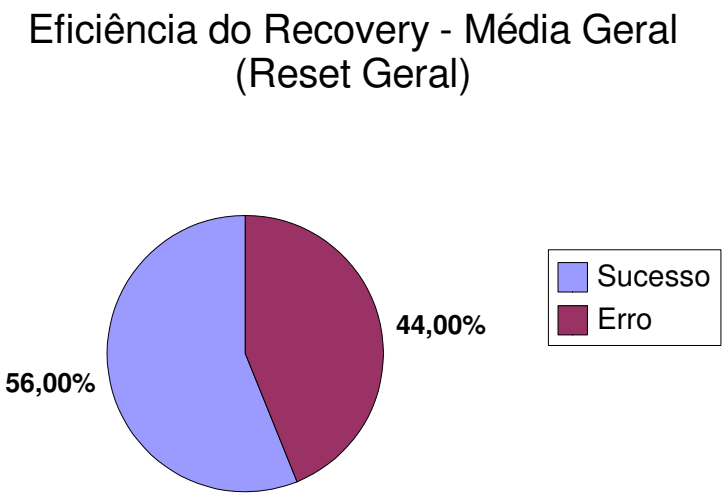

FIGURA 6.12 - Gráfico eficiência da recuperação - *Reset* geral - Média Geral.

Como mostram os dados da figura 6.12, quando foi utilizada a técnica de *reset* geral no equipamento no momento que um processo estabelecia atualizações (inserções, alterações e deleções) no banco de dados, **com** e **sem** *forced writes,* de um total de 75 execuções, o mecanismo de recuperação do InterBase mostrou-se eficiente em 42 casos e não obteve sucesso nos outros 33 casos. A cobertura de recuperação em função das falhas injetadas foi de 56% e a cobertura de detecção em função das falhas injetadas foi de 100%. Não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente.

Quando esta técnica foi utilizada, o mecanismo mostrou-se muito eficiente com operaçõe de inserção. Dos 32 experimentos, **com** e **sem** *forced writes,* em 29 dos casos (90,62%) o mecanismo obteve sucesso e em apenas três casos (9,37%) o mecanismo não obteve sucesso. Quando foram utilizadas operações de alteração e deleção, de um total de 43 experimentos, **com** e **sem** o parâmetro *forced writes,* em apenas 13 casos (30,23%) o mecanismo obteve sucesso, nos demais 30 casos (69,77%) o mecanismo foi ineficiente.

6.2.1 Conclusão

Conforme descrito no modelo de falhas (capítulo 5.6) - Erros de escrita no sistema

operacional ou hardware: Estes erros geralmente criam problemas com a integridade do banco de dados. Como mostram os dados da tabela 6.2, nos 75 experimentos com a segunda técnica, que consistia em *reset* geral do equipamento, a cobertura de recuperação foi de 56%. O mecanismo de detecção de erros mostrou-se muito eficiente, uma vez que todos os erros foram detectados e uma mensagem correspondente foi mostrada ao usuário quando da tentativa de acesso ao banco de dados (100% de cobertura de detecção). Os tempos de indisponibilidade, quando houve recuperação, foram extremamente baixos (em torno de um segundo), mesmo quando o conjunto de operações realizadas e de dados atualizados era grande. Quando não houve recuperação, os tempos de indisponibilidade foram os necessários para a restauração da cópia da base de dados.

6.3 Experimentos com uso da ferramenta

Os experimentos descritos nesta seção foram conduzidos utilizando-se a ferramenta de injeção de falhas - FIDe, descrita anteriormente no capítulo 4, para simular falhas transientes de hardware, no momento que um processo realiza operações de seleção, inclusão, alteração ou deleção de dados em tabelas de um banco de dados, **com** e **sem** o parâmetro *forced writes*. O conjunto de *syscalls* selecionadas para a injeção de falhas foi obtido através da análise dos dados do arquivo gerado pelo depurador *strace*.

Como descrito no capítulo 5.6, um banco de dados está sujeito a eventos que podem comprometer suas estruturas. Esta técnica foi utilizada visando validar os mecanismos de detecção e recuperação do SGBD InterBase, simulando tipos de erros de hardware que podem ocorrer durante o processamento de transações e que podem afetar seriamente a integridade dos dados e das estruturas do banco de dados

6.3.1 Seleção de Syscalls e Modelo de Falhas

Como a ferramenta permite a montagem de um grande número de possíveis cenários de injeção de falhas para cada uma das *syscalls* existentes [BEC 98], optou-se pela seleção de algumas *syscalls* que são mais freqüentemente utilizadas (*get_pid, get_uid, readlink, old_mmap, new_stat, mprotect*) ou que seriam logicamente utilizadas na execução das operações de seleção, inclusão, alteração e deleção (*open, close, read, write, exit*). As *syscalls* utilizadas nos testes estão descritas na tabela 6.3. Outras *syscalls* utilizadas pela aplicação e testadas no injetor não afetaram o funcionamento da aplicação ou o banco de dados.

Foi definido o número de 20 execuções para cada tipo de teste porque é um número considerável e porque, praticamente, não havia variação nos resultados. Portanto, pressupõe-se que, se houvessem mais execuções, o resultado seria o mesmo.

No total foram 3200 execuções, divididas em 80 testes distintos, cada teste sendo repetido 20 vezes **com** e 20 vezes **sem** o parâmetro *forced writes.* As operações realizadas nos testes realizavam contagem, consulta, inserção, alteração ou deleção de

registros de uma ou mais tabelas. As operações eram realizadas em tabelas com pequeno número de registros lógicos (de 1 a 70 registros).
Os experimentos com o FIDe foram realizados com carga de trabalho diferente dos experimentos anteriores (*kill* e *reset*), porque objetivavam a injeção da falha, a observação da ocorrência ou não do erro e a observação do comportamento dos mecanismos de tolerância a falhas, independente de quando este erro ocorresse. Desta forma, optou-se por pequena carga de trabalho para agilizar a execução do conjunto de testes em função dos objetivos. Entretanto, para que se fizesse uma análise comparativa entre as técnicas, o ideal seria a realização de experimentos mais homogêneos, com idêntica carga de trabalho.

As falhas injetadas, definidas através dos cenários, foram com os registradores, **ecx** (1), **edx** (2) e **eax** (6). As falhas injetadas simulavam uma quantidade menor ou maior de dados a serem gravados conforme o que tinha sido requisitado, através da alteração do valor do registrador **edx** (ex.: R 2 - 1, R 2 + 2, R 1 - 2, R 1 + 2), alteravam (somavam ou subtraiam) o conteúdo da posição apontada pelo registrador **ecx** (P 1 - 2, P 1 + 2), ou alteravam a informação de retorno da quantidade de bytes gravados (registrador **eax ->** R 6 - 1). A freqüência (a cada "x" chamadas da *syscall*) e o momento da injeção da falha (na chamada, retorno ou ambas) variaram nos diversos testes. A tabela 6.4 mostra o conjunto de *syscalls* utilizadas, que afetaram de alguma forma o ambiente do banco de dados. A tabela 6.5 mostra as *syscalls* utilizadas, o número total de execuções para cada *syscall* e as conseqüências no banco de dados.

Embora a injeção de falhas tenha se restringido à mudança de valores de registradores de CPU, as mesmas, indiretamente acabaram afetando as operações realizadas em memória. A idéia básica reside no fato que cada chamada de sistema utiliza os registradores para o recebimento de parâmetros ou devolução de resultados. Desta forma, modificar o valor de um registrador pode significar escrever em outro arquivo, diferente do arquivo que havia sido requisitado, gravar mais ou menos dados que o conteúdo devido, armazenar informações em área diferente da especificada, alocar mais memória do que o desejado, alterar o código de retorno de uma operação realizada, alterar a data do sistema, alterar o endereço inicial de um conjunto de dados a serem lidos, etc. Estas alterações irão comprometer os resultados dependendo do registrador modificado e da chamada de sistema utilizada.

Exemplos de cenários de falhas utilizados:

1 4 => *Siscall* 4 (*write*)
W I H 2 => Na chamada da *syscall*, a cada duas chamadas
R 2 = 0 => Altera o valor do registrador **edx** para 0

1 55 => *Syscall* 55 (fcntl)
W B H 1 => Na chamada e no retorno da *syscall*, a cada chamada
P 1 - 2 => Subtrai 2 do conteúdo da posição apontada pelo registrador **ecx**

1 90 => *Syscall* 90 (Mmap)
W B H 1 => Na chamada e no retorno da *syscall*, a cada chamada
R 6 - 1 => Atribui o valor -1 ao registrador **eax**

TABELA 6.3 – Syscalls selecionadas para a injeção de erros.

Syscall	***Nome***	***Syscall***	***Nome***
1	Exit	45	Brk
2	Fork	54	Ioctl
3	Read	55	Fcntl
4	Write	67	Sigaction
5	Open	85	Readlink
6	Close	90	Mmap
10	Unlink	91	Munmap
13	Tyme	106	Newstat
19	Lseek	107	Newlstat
20	Getpid	108	Newfstat
23	Setuid	117	Ipc
24	Getuid	122	Newuname
27	Alarm	125	Mprotect
29	Pause	141	Getdents
37	kill	143	Flock

TABELA 6.4 – Cenários de falhas injetadas que afetaram o BD.

Syscall	*Quando*	*Operação*	*Ação*
3	Chamada	A cada chamada	Diminui 5 do valor de edx
3	Chamada/Retorno	A cada chamada	Diminui 1 do valor de ebx
4	Chamada	A cada 4 chamadas	Diminui 1 do valor de edx
4	Chamada	Cada duas chamadas	Altera o valor de ecx para -1
4	Chamada	A cada 4 chamadas	Soma 2 ao conteúdo da posição apontada por ecx
4	Retorno	A cada 4 chamadas	Diminui 2 do conteúdo da posição apontada por edx
4	Chamada/Retorno	A cada chamada	Altera o valor de eax para -1
6	Chamada	A cada chamada	Diminui 2 do conteúdo da posição apontada por ecx
19	Chamada	A cada chamada	Altera o valor de ecx para 0
19	Chamada	A cada 4 chamadas	Diminui 1 do valor de edx
19	Chamada	A cada chamada	Diminui 2 do conteúdo da posição apontada por ecx
19	Chamada	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
20	Chamada	A cada chamada	Diminui 5 do conteúdo da posição apontada por ecx
24	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
37	Retorno	A cada duas chamadas	Diminui 2 do conteúdo da posição apontada por ecx
45	Chamada/Retorno	A cada chamada	Diminui 2 do conteúdo da posição apontada por ecx
45	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
54	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
55	Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
90	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
90	Chamada/Retorno	A cada chamada	Altera o valor de eax para -1

Syscall	*Quando*	*Operação*	*Ação*
91	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
117	Chamada/Retorno	A cada chamada	Diminui 2 do conteúdo da posição apontada por ecx
117	Retorno	A cada duas chamadas	Diminui 2 do conteúdo da posição apontada por edx
122	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
125	Chamada/Retorno	A cada chamada	Diminui 2 do conteúdo da posição apontada por ecx
141	Retorno	A cada duas chamadas	Soma 2 ao valor de ecx
141	Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por edx
143	Chamada/Retorno	A cada chamada	Soma 2 ao valor de ecx
143	Chamada/Retorno	A cada chamada	Soma 2 ao conteúdo da posição apontada por ecx
143	Chamada/Retorno	A cada chamada	Altera o valor de eax para -1

TABELA 6.5 – Syscalls executadas e conseqüências no BD.

Syscall	*Execuções*	*Resultado*
1 -Exit	120	Não Afetou
2 - Fork	80	Não Afetou
3 - Read	120	Danificou dados e estruturas e afetou conexão
4 - Write	400	Danificou dados e estruturas e afetou conexão
5 - Open	80	Não afetou
6 - Close	80	Afetou conexão
10 - Unlink	80	Não afetou
13 - Tyme	80	Não afetou
19 - Lseek	160	Danificou dados e estruturas e afetou conexão
20 - Getpid	80	Afetou conexão
23 - Setuid	80	Não afetou
24 - Getuid	80	Afetou conexão
27 - Alarm	80	Não afetou

Syscall	*Execuções*	*Resultado*
29 - Pause	80	Não afetou
37 - Kill	80	Afetou conexão
45 - Brk	80	Afetou conexão
54 - Ioctl	120	Afetou conexão
55 - Fcntl	120	Afetou conexão
67 - Sigaction	80	Não afetou
85 - Readlink	80	Não afetou
90 - Mmap	120	Afetou conexão
91 - Munmap	120	Afetou conexão
106 - Newstat	80	Não afetou
107 - Newlstat	80	Não afetou
108 - Newfstat	80	Não afetou
117 - Ipc	120	Afetou conexão
122 - Newuname	120	Afetou conexão
125 - Mprotect	80	Afetou conexão
141 - Getdents	120	Afetou conexão
143 - Flock	120	Afetou conexão

6.3.2 Resultados Obtidos

Com a ferramenta de injeção de falhas FIDe foram realizados 3200 execuções, com 80 testes distintos. Cada teste foi executado 20 vezes **com** e 20 vezes **sem** o parâmetro *forced writes*. Foram montados cenários para injeção de falhas para 30 *syscalls* distintas, com no mínimo dois parâmetros diferentes para cada cenário (mudança de valor do registrador **edx** e mudança do conteúdo apontado pelo registrador **ecx**). As *syscalls* mais utilizadas nos testes foram as de número 4 (*write*), 19 (*lseek*) e 143 (*flock*).

Do total de 80 testes distintos alguns não afetaram o ambiente do banco de dados, outros afetaram parcialmente (ou demorando para fazer a conexão, ou não completando a conexão com o banco de dados, ou travando o processo de conexão) e outros afetaram a consistência dos dados e as estruturas de dados e danificaram permanentemente a base de dados. Os testes com as *syscalls* 3 (*read*), 4 (*write*) e 19 (*lseek*) foram as que corromperam seriamente a base de dados.

Em cada um dos testes, **com** e **sem** o parâmetro *forced writes* não houve diferenciação nos resultados, isto é, houve uniformidade nos resultados (os 20 testes geraram o mesmo resultado). Dos 80 testes realizados, em 47 (58,75 %) a injeção de falhas não afetou o banco de dados, isto é, nenhuma anormalidade ocorreu ou a falha foi adequadamente

mascarada. Em 29 testes (36,25 %) a injeção de falhas afetou parcialmente, ou demorando para fazer a conexão com o banco de dados, ou não concluindo esta conexão, ou travando o processo de conexão com o banco de dados. Em 2 testes (2,5%) não ocorreu a detecção do erro e houve o travamento do processo em execução. Em 4 testes (5,00 %) a injeção de falhas afetou seriamente a base de dados e as estruturas de dados, danificando permanentemente a base de dados. Visando oferecer melhor visualização, estes resultados são mostrados na figura 6.13.

Quando a injeção de falhas corrompeu a base de dados ou as estruturas do banco de dados, as seguintes mensagens foram exibidas (ou no momento da injeção, ou em posterior tentativa de conexão com a banco de dados):

- **I/O Error for file "/home/athenas/...../teste.gdb"**
  Error while trying to read from file
  Bad address

- **Teste.gdb is not a valid database**

- **Database file appears corrupt**
  wrong page type

- **Error while trying to access file**

Em alguns casos, quando a injeção de falhas não conseguiu concluir a conexão com o banco de dados, sem contudo, ter afetado a base de dados, as seguintes mensagens foram exibidas (ou no momento da injeção, ou na tentativa de nova conexão com a banco de dados):

- **Lock conflict on no wait transaction**

- **Fatal lock manager erro: semop failed (acquire)**

- **Operating system directive open failed -No such file or directory**

- **Operating system directive flock failed -Invalid argument**

- **internal gds software consistency check (page in use during flush (210))**

- **Fatal lock manager error: invalid lock id (1716), errno: 0**

- **ISC_post_event: semctl failed with errno = 22**

- **I/O Error for file "/home/athenas/...../teste.gdb"**
  Error while trying to write to file
  Bad address

O mecanismo de detecção de erros apenas duas vezes (2,5 % dos casos) não identificou

o erro e houve o travamento da operação. Nos demais casos (97,5 %) o mecanismo de detecção identificou o erro e exibiu uma mensagem correspondente ao usuário (no momento da injeção da falha ou em posterior tentativa de conexão/acesso ao banco de dados). Em um teste (1,25% dos casos), com a *syscal* 143, quando atribuiu-se o valor -1 ao registrador **eax**, as execuções tiveram resultados diferentes: algumas vezes ocorria a conexão com o banco de dados, algumas vezes ocorria o travamento da operação, outras vezes não ocorria a conexão e a sessão *isql* era interrompida.

Os tempos de indisponibilidade variaram desde 2 a 3 segundos, quando o erro foi detectado e um aviso correspondente foi exibido, ou a conexão ou a sessão *isql* foi encerrada, até um longo tempo, em que foi necessária a intervenção do usuário para que se matasse o processo em execução. Quando houve corrupção de dados ou de estruturas, o tempo de indisponibilidade foi o necessário para restaurar a cópia do banco de dados (alguns segundos para um pequeno banco de dados).

A ferramenta mostrou-se extremamente flexível, permitindo que se fizessem alterações nos cenários e parâmetros de injeção de falhas de forma bastante simples e clara. A ferramenta gera um arquivo chamado **fide**.**report** que registra as *syscalls* chamadas e as falhas injetadas. O conteúdo abaixo é um fragmento do arquivo fide.report :

```
Beggining the execution of ./isql...
   pid linked list (first_pid)->0804e1c8
   main_rule(0804e280)->(0804e1f0)
  --[ add_pid: activate pid[1204] in 0 seconds
incrementing number of processes!!!!!
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [90]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [90]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [125]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [125]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [125]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [125]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [106]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [106]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [5]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [5]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [90]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [90]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [6]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [6]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [24]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [37]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [37]  sig: 5  nprocs: 1
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [37]  sig: 5  nprocs: 1
---[1204] The syscall 37 matches with one of the rules
 - Not time to inject faults yet
 - looking for the next rule_when
 - not found!
[-0000057f-Trace/breakpoint trap-]   pid: 1204 st:1407 syscall: [37]  sig: 5  nprocs: 1
---[1204] The syscall 37 matches with one of the rules
```

- found a rule_param rule...
[Memory [0x0] value changed from 0xffffffff Error chaging param at [0x0]...

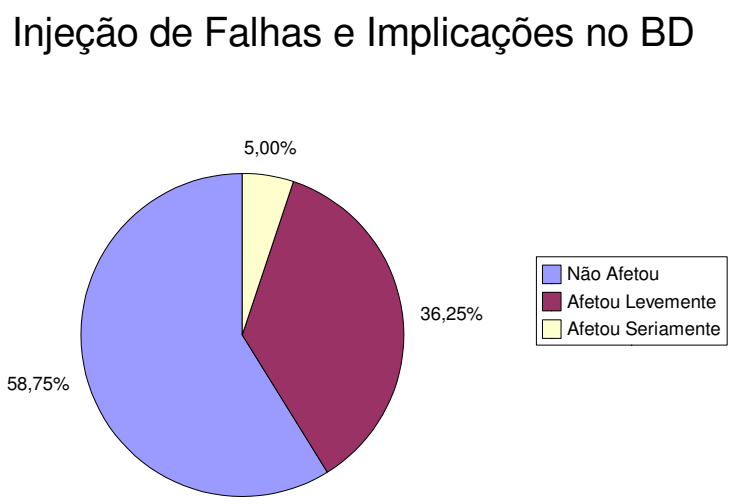

FIGURA 6.13 - Gráfico da implicação das falhas injetadas no banco de dados.

Como mostram os dados da figura 6.13, quando foi utilizada a técnica de injeção de falhas através da ferramenta FIDe, **com** e **sem** o parâmetro *forced writes,* de um total de 80 experimentos, em 47 casos (58,75%) as falhas não afetaram o banco de dados, em 27 casos (36,25%) afetaram parcialmente (ou retardando a conexão ou não concluindo a conexão com o banco de dados), e em 4 casos (5,00%) as falhas afetaram seriamente os dados e as estruturas da base de dados. A cobertura de recuperação em função das falhas injetadas foi de 95% e a cobertura de detecção em função das falhas injetadas foi de 97,5%. Não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente. Quando houve o travamento do processo em execução, os tempos de indisponibilidade foram longos. Estes processos só foram interrompidos pelo usuário, através do comando *kill*.

6.3.3 Conclusão

Conforme previsto no modelo de falhas (capítulo 5.6), os erros de escrita no sistema operacional ou hardware geralmente criam problemas com a integridade do banco de

dados. Como mostram os dados da figura 6.13, quando utilizou-se a injeção de falhas através da ferramenta, em 58,75% dos testes as falhas injetadas não afetaram o banco de dados ou se afetaram, foram eficientemente mascaradas. Em 36,25% dos testes as falhas injetadas afetaram parcialmente, ou retardando a conexão com o banco de dados ou não concluído esta conexão. Em 5,00% dos testes as falhas injetadas afetaram seriamente os dados e as estruturas do banco de dados, corrompendo-o definitivamente.

Em apenas duas vezes (2,5% dos casos) o mecanismo de detecção não identificou o erro e houve o travamento da operação. Nos demais casos (97,5 %) o mecanismo de detecção identificou o erro e exibiu uma mensagem correspondente ao usuário (no momento da injeção da falha ou quando de nova tentativa de acesso ao banco de dados).

Os tempos de indisponibilidade variaram muito quando a falha retardava ou impedia a conexão com o banco de dados. Quando houve o processo de recuperação, os tempos de indisponibilidade foram extremamente baixos (em torno de um segundo), mesmo quando o conjunto de operações realizadas e de dados atualizados era grande. Quando houve corrupção de dados ou de estruturas, o tempo de indisponibilidade foi o necessário para restaurar a cópia do banco de dados (alguns segundos para um pequeno banco de dados). Apenas houve longo tempo de indisponibilidade, quando o mecanismo não identificou o erro e houve o travação do processo em execução. Neste caso, foi necessária a intervenção do usuário, matando o processo que estava travado.

7 Conclusão e Trabalhos Futuros

Os objetivos deste trabalho, descritos anteriormente, são a validação dos mecanismos de detecção de falhas, do mecanismo de recuperação de falhas, e a apuração dos tempos de indisponibilidade do banco de dados após a ocorrência de falhas do tipo *crash.* Também avalia a aplicabilidade da técnica de injeção de falhas em SGBDs e testa numa aplicação real a ferramenta de injeção de falhas FIDe. A seguir são relatadas as conclusões obtidas, relativas a cada um destes quesitos.

A partir dos resultados obtidos pelos experimentos pode-se constatar que os mecanismos de detecção e recuperação de falhas do SGBD InterBase mostraram comportamentos bastante diferenciados com relação a cada uma das técnicas utilizadas.

Quando foi utilizada a primeira técnica, que consistia em eliminar, manualmente, através do comando *kill* do sistema operacional, um processo pai que continha processos filhos que estabeleciam atualizações (inserções, alterações e deleções) no banco de dados os mecanismos mostraram-se bastante confiáveis, ficando em conformidade com o definido no modelo de falhas (capítulo 5.6). Com esta técnica a cobertura de detecção de falhas foi de 99,71% e a cobertura de recuperação também foi de 99,71%. No único caso em que não houve a detecção do erro, não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados, pois o mesmo estava corrompido definitivamente).

Quando utilizou-se a segunda técnica, que consistia em *reset* geral no equipamento no momento em que um processo estabelecia atualizações (inserções, alterações e deleções) no banco de dados, o mecanismo de detecção mostrou-se muito eficiente (detectou o erro em 100% dos casos). Embora esteja previsto no modelo de falhas que erros de escrita no sistema operacional ou hardware geralmente criam problemas com a integridade do banco de dados, o mecanismo de recuperação mostrou-se muito ineficiente, principalmente nas operações de alteração e deleção. Deve-se, aqui, fazer uma observação importante: o *reset* geral no equipamento caracteriza-se como um procedimento extremo e era executado quando um conjunto grande de transações estabelecia atualizações no banco de dados.

Usando a segunda técnica, nos experimentos de inserção de dados houve uma clara diferença entre o uso ou não do parâmetro *forced writes*. Nos experimentos de inserção de registros **com** *forced writes,* o mecanismo de recuperação mostrou-se extremamente eficiente (100%). Nos experimentos de inserção **sem** *forced writes,* o mecanismo de recuperação obteve sucesso em 75% dos casos. Fazendo-se uma média geral dos experimentos de inserção, **com** e **sem** *forced writes* obtém-se uma média geral de sucesso do mecanismo de recuperação entre 86,36 e 90,62%.

Os experimentos de alteração de dados, com a segunda técnica, apresentaram resultados surpreendentes, uma vez que esperava-se melhor desempenho do mecanismo de recuperação com o uso do parâmetro *forced writes,* visto que este é utilizado com o propósito de garantir segurança e integridade. Nos experimentos de alteração **sem**

forced writes, o mecanismo de recuperação mostrou-se eficiente em 53,8% dos casos. Dos experimentos de alteração **com** *forced writes*, o mecanismo de recuperação obteve sucesso em apenas 30% do casos. Totalizando os experimentos de alteração, **com** e **sem** o parâmetro *forced writes* obtém-se uma média geral de sucesso do mecanismo de recuperação de 43,47% (apenas regular).

Nos experimentos de deleção de dados, utilizando-se a segunda técnica, **com** *forced writes,* o mecanismo de recuperação mostrou-se eficiente em apenas 20% dos casos Dos experimentos de deleção **sem** *forced writes*, o mecanismo de recuperação obteve sucesso em apenas 10% dos casos. Fazendo-se uma média geral dos experimentos de deleção, **com** e **sem** o parâmetro *forced writes* obtém-se uma média geral do mecanismo de recuperação de 15% (extremamente baixo).

Do total de experimentos com a técnica de *reset* geral, **sem** *forced writes* o mecanismo de recuperação obteve sucesso em 48,57% dos casos, e **com** *forced writes* o mecanismo de recuperação obteve sucesso em 62,5% dos casos. Totalizando-se os experimentos com *reset* geral, **com** e **sem** o parâmetro *forced writes,* obtém-se uma média geral de cobertura de recuperação de 56% e uma cobertura de detecção de erros de 100%. Não houve tempo de indisponibilidade para o processo de recuperação e a indisponibilidade foi de alguns poucos segundos (tempo para a restauração da cópia do banco de dados), quando o mesmo foi corrompido definitivamente.

Pode-se concluir que, com a técnica de *reset* geral, o mecanismo de recuperação comportou-se satisfatoriamente apenas com as operações de inserção. Nas operações de alteração e deleção o mecanismo mostrou-se ineficiente, mesmo quando utilizado o parâmetro *forced writes*.

A injeção de falhas através do FIDe mostrou que algumas falhas não afetaram o ambiente do banco de dados, outras afetaram parcialmente (demorando para fazer a conexão, não completando a conexão com o banco de dados, ou travando o processo em execução) e outras afetaram a consistência e as estruturas de dados e danificaram permanentemente a base de dados. Embora o cenário de falhas tenha sido bastante semelhante para quase todas as *syscalls*, as de número 3 (*read*), 4 (*write*) e 19 (*lseek*) foram as que corromperam a base de dados de forma definitiva.

Em conformidade com o previsto no modelo de falhas (erros de escrita no sistema operacional ou hardware geralmente criam problemas com a integridade do banco de dados) a cobertura de recuperação foi de 95% e a cobertura de detecção de falhas foi de 97,5%.

Com a injeção de falhas através do FIDe, os tempos de indisponibilidade variaram em torno de 2 a 5 segundos, quando o erro foi detectado e um aviso correspondente foi exibido ao usuário, ou quando a conexão com banco de dados ou a sessão *isql* foi encerrada, até um longo tempo, em que foi necessária a intervenção do usuário para que se matasse o processo em execução. Quando houve corrupção de dados ou de estruturas de dados, o tempo de indisponibilidade foi o necessário para restaurar a cópia do banco de dados (alguns segundos para um pequeno banco de dados).

Fazendo-se uma análise geral, baseada nos diversos experimentos realizados neste

trabalho, pode-se concluir que:

1) O mecanismo de recuperação do InterBase mostrou-se eficiente, exceto com as operações de alteração e deleção com a técnica de *reset* geral, que apresentaram sérios problemas e baixa eficiência.

2) O mecanismo de detecção do InterBase mostrou-se extremamente eficiente, pois em mais de 99,7% dos casos o erro foi detectado, identificado e uma mensagem correspondente foi exibida ao usuário e o acesso não foi permitido. Este tipo de comportamento é interessante e desejável, uma vez que um banco de dados que esteja em um estado inconsistente e permita acesso aos dados dele pode gerar informações inconsistentes.

3) Pode-se concluir também que, com relação a disponibilidade, o InterBase mostrou-se muito eficiente, visto que o tempo necessário para disponibilizar o banco de dados ao usuário foi extremamente baixo (aproximadamente um segundo), mesmo quando o conjunto de operações e o tamanho das tabelas atualizadas eram grandes. Em apenas 3 testes houve longa indisponibilidade (travamento do processo em execução) e foi necessária a intervenção do usuário. Esta eficiência no desempenho do processo de recuperação é graças ao mecanismo utilizado pelo InterBase, que utiliza TIPs ao invés de arquivos de *log.*

4) A técnica de injeção de falhas, (com e sem a ferramenta), mostrou-se adequada para validar os mecanismos de tolerância a falhas no banco de dados alvo deste trabalho.

Embora não esteja definido como objetivo deste trabalho, deve-se ressaltar a robustez do sistema operacional Linux. A versão utilizada nos experimentos mostrou-se altamente confiável. Em nenhum momento houve o travamento do sistema ou a interrupção dos serviços. Nos 75 experimentos em que foi utilizado o *reset* geral do equipamento, que é claramente uma agressão a todo o ambiente operacional, em apenas um, o que representa 1,33% foi necessário a intervenção do operador e o uso manual do comando *fsck*. Em todas as outras tentativas (98,67%) o próprio sistema operacional recuperou-se automaticamente.

A partir dos estudos até aqui realizados, com a avaliação e a validação da ferramenta de injeção de falhas FIDe, que possui fortes características de flexibilidade e parametrização, juntamente com as observações e os resultados obtidos, propomos um trabalho comparativo entre diversos sistemas gerenciadores de bancos de dados (ORACLE, POSTGRES, MYSQL, SQL-SERVER, IBM-DB2, etc.). Este trabalho, com um cenário de falhas definido, testaria uma variada gama de produtos comerciais com relação a diversos requisitos (disponibilidade, tempos médios de recuperação, eficiência do mecanismo de detecção de falhas, eficiência do mecanismo de recuperação de falhas, número de transações concluídas e número de transações desfeitas em função de um determinado tempo de recuperação). Pode-se ainda propor trabalhos similares, mas que enfocassem ou simulassem outros tipos de falhas (hardware, comunicação em ambiente distribuído, etc.).

Outra alternativa interessante para trabalhos futuros seria o mapeamento do conjunto de falhas que um determinado banco de dados consegue detectar e se o mecanismo de recuperação do mesmo, com que freqüência ou razão, obtém sucesso neste procedimento.

Ainda, utilizando-se o FIDe, poderia-se fazer experimentos em diversos ambientes, além de bancos de dados, verificando o nível de interferência e degradação de desempenho, em processos que rodem com o injetor de falhas em modo concorrente.

Bibliografia

[ARL 89] ARLAT, J. et al. Fault-injection for dependability validation of fault-tolerant computing systems. In: FAULT TOLERANCE COMPUTING SYMPOSIUM, FTCS, 19., 1989, Chicago. **Proceedings...** [S.l.:s.n.], 1989. p. 348-355.

[ARL 90] ARLAT, J. et al. Fault-injection for dependability validation: a methodology and some applications. **IEEE Transactions on Software Engineering**, New York, v. 16, n. 2, Feb. 1990.

[BAR 99] BARCELOS, P.P.A.; LEITE, O.; WEBER, T. S. Implementação de um Injetor de Falhas de Comunicação. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, SCTF, 8., Campinas. **Anais...** p. 225-239.

[BEC 98] BECK, M. et al. **Linux Kernel Internals.** 2nd ed. New York: Addison-Wesley, 1998.

[BER 87] BERNSTEIN, P.; HADZILACOS, V.; GOODMAN, N. **Concurrency Control and Recovery in Database Systems.** Reading, Massachusetts: Addison-Wesley, 1987.

[BOR 96] BORLAND INTERNATIONAL Inc. **Borland InterBase -Language Reference.** Scotts Valley, Califórnia, 1999.

[BOR 99] BORLAND INTERNATIONAL Inc. **Borland InterBase Workgroup Server - Installing and Running on Linux.** Scotts Valley, Califórnia, 1999.

[BOR 2000] A COMPARACION of INTERBASE vs. SQL SERVER. Disponível em: <http://www.borland.com/interbase/papers>. Acesso em: jul. 2000.

[CLA 95] CLARK, J. A.; PRADHAN, D. K. Fault Injection - A Method for Validating Computer-System Dependability. **IEEE Micro**, New York, v. 28, n. 6, p.45-56, June 1995.

[CAR 98] CARREIRA, J. et al. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. I**EEE Transactions on Software Engeneering**, New York, v. 24, n. 2, p.125-136, Feb. 1998.

[CAR 99] CARREIRA, J.; COSTA, D.; SILVA, J.G. Fault Injection Spot-checks. **IEEE Spectrum**, New York, v. 36, n. 8, Aug. 1999.

[COS 98] COSTA, D.; MADEIRA, H.; SILVA, J.G. Delphos : Dependability Evaluation of COTS DBMS. In: FAULT TOLERANCE COMPUTING SYMPOSIUM, FTCS, 28., 1998, Munich. **Fast Abstracts...** [S.l.:s.n.], 1998. p. 86.

[COS 99] COSTA, D.; MADEIRA, H. **Experimental Assessment of COTS DBMS Robustness under Transiente Faults.** Disponível em: <http://dsg.dei.uc.pt>. Acesso em: set. 2000.

[ELM 89] ELMASRI, R.; NAVATHE, S.B. **Fundamentals of Database Systems.**

Califórnia: The Benjamin/Cummings, 1989.

[FOC 2001] FOCUS on Unix - Process Management. Disponível em: <http://unix.about.com/library/weekly/aa062501d.htm>. Acesso em: out. 2001.

[FRA 2000] FRANKLIN M.J. **Concurrency Control & Recovery**. Disponível em: <http://www.dcs.stand.ac.uk/~vangelis/work/summaries.html>. Acesso em: jan. 2000.

[GON 2000] GONÇALVES, L.C.R. **Injeção de Falhas via Depuradores.** 2000. Trabalho Individual (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[GON 2002] GONÇALVES, L.C.R. **Injeção de Falhas via Depuradores.** 2001 90p. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

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

[HSU 97] HSUEH, M.; TSAI, T.K.; IYER. R.K. Fault-Injection Techniques and Tools. **Computer**, New York, v.30, n.4, p.75-82, Apr. 1997.

[HUN 99] HUNTER, A. **Security and Recovery.** Disponível em: <http://www.osiris.sunderland.ac.uk/ahu/rds/lec5.htm>. Acesso em: jun. 1999.

[INT 2001] INTERBASE-BR. **Quem usa o Interbase?** Disponível em: <http://www.warmboot.com.br/ib/artigos/quem_usa_ib.html>. Acesso em: nov. 2001.

[JAL 94] JALOTE, P. **Fault Tolerance in Distributed Systems.** New Jersey: Prentice Hall, 1994.

[KOR 95] KORTH, H. F.; SILBERSCHATZ, A. **Sistema de Banco de Dados.** 2.ed. São Paulo: Makron Books, 1995.

[KUM 99] KUMAR, V. **Database Recovery.** Disponível em: <http://www.cstp.umck.edu/~kumar/cs470/network/UGDB_Recovery_Intro.html>. Acesso em: jun. 1999.

[LAP 92] LAPRIE, J.C. Dependability: Basic Concepts and Terminology. In: **Dependable Computing and Fault Tolerance**. Vienna: Springer-Verlag, 1992.

[LEE 99] LEE, S.; PYLKANEN, K.; SOMASUNDARAM, R. **Database Recovery.** Disponível em:

<http://members.bellatlantic.net/~sihlee/papers>. Acesso em: jun. 1999.

[LEI 87] LEITE, J.C.B.; LOQUES FILHO, O.G. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, SCTF, 2., 1987. **Mini-curso:** Introdução à Tolerância a Falhas. Campinas: [s.n.], 1987.

[LEI 2000] LEITE, F.O. **ConFIRM – Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional.** 2000. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[LIN 2000] LINUXCARE, Product Comparisons -Databases. Disponível em: <http://www.linuxcare.com/products>. Acesso em: set. 2000.

[LOM 98] LOMET, D. B. Recovery Mechanisms in Database Systems. In: **Advanced Recovery Tecniches in Practice**, New Jersey: Prentice Hall, 1998.

[MAD 99] MADEIRA, H. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, SCTF, 8., 1999, **Mini-curso:** Avaliação de Técnicas de Tolerância a Falhas em Sistemas de Gestão de Bases de Dados, Campinas: [s.n.], 1999.

[MAN 2001] MANFREDINI, R.A. **Condução de Experimentos de Injeção de Falhas em Banco de Dados Distribuídos.** 2001. 70p. Dissertação (Mestrado em Ciência da Computação) - Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[MAR 89] MARTINS, E. et al. Testing Multipeer Protocols in the Presence of Faults. In: **Workshop on Protocol Test Systems**, 2., 1989, Berlim. **Proceedings...** [S.l.:s.n.], 1989. p. 77-91.

[MAR 96] MARTINS, E. **ATIFS - Um ambiente de Testes baseado em Injeção de Falhas por Software**. 1995. 23p. Relatório Técnico - DCC 95-24 - Departamento de Ciência da Computação - Universidade Estadual de Campinas, Campinas.

[MOH 92] MOHAN, C. et al. ARIES : A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks using Write-Ahead Logging. **ACM Transactions on Database Systems**, New York, v.17,n.1, p. 94-162, Mar. 1992.

[PRA 96] PRADHAN, P.K. **Fault-Tolerant Computer System Design**. New Jersey: Prentice Hall, 1996.

[ROD 2001] RODEGHERI, P. R.; WEBER, T. S. Disponibilidade, Detecção e Recuperação de Erros no SGBD InterBase 4.0 Utilizando Software Livre. In: WORKSHOP SOBRE SOFTWARE LIVRE - WSL, 2., 2001.

Anais... Porto Alegre: SBC, 2001. p.1-4.

[ROD 2001A] RODEGHERI, P. R. et al. Testing Fault Tolerance Mechanisms in DBMS Through Fault Injection, In: IEEE LATIN AMERICAN TEST WORKSHOP. 2., 2001, Cancún. **Digest of papers.** [Amissville: IEEE Computer Society], 2001. p.278-284.

[ROD 2000] RODEGHERI, P. R.; WEBER**,** T. S**.** Avaliação do Mecanismo de Recovery do SGBD InterBase em Ambiente Linux. In: FÓRUM DE TECNOLOGIA, 3., 2000, Santo Ângelo. **A Tecnologia na Evolução da Sociedade**: anais. Santo Ângelo: URI, 2000. 1 CD.

[ROS 96] ROSEMBERG, J.B. **How Debuggers Work** : Algorithms, Data Structures, and Architecture. New York: John Wiley & Sons, 1996.

[SAB 98] SABARATNAM, Maitrayi; TORBJØRNSEM, Ø. **Evaluating the Effectiveness of Fault Tolerance in Replicated Database Management Systems.** Oslo: Norwegian University of Science & Technology, 1998. Disponível em: <http://www.idi.ntnu.no>. Acesso em: ago. 2000.

[SAB 99] SABARATNAM, M.; TORBJØRNSEM, Ø. Cost of Ensuring Safety in Distributed Database Management Systems. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 1999, Hong Kong. **Proceedings...** [Amissville: IEEE Computer Society], 1999.

[SCH 93] SCHNEIDER, F.B. What Good are Models and What Models are Good? In: **Distributed Systems.** 2nd ed. New York: Addisson Wesley, 1993. (ACM Press Frontier Series).

[SOT 97] SOTOMA, I.; WEBER, T. S. AFIDS - Arquitetura para Injeção de Falhas em Sistemas Distribuídos. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 15., 1997. **Anais...** São Carlos: UFScar, 1997. p.294-309.

[SUL 91] SULLIVAN, M.; CHILLAREGE, R. Software defects and their Impact on System Availability - A study of Field Failures in Operation Systems. In: FAULT TOLERANCE COMPUTING SYMPOSIUM, FTCS, 21., 1991, Montreal, **Proceedings...** [Amissville: IEEE Computer Society], 1991. p. 2-9.

[SUL 92] SULLIVAN, M.P. **System Support for Software Fault Tolerance in Highly Available in Database Management Systems.** 1992. 162p. Ph. Thesis. University of California at Berkeley, California.