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

Condução de Experimentos de Injeção de Falhas em Banco de Dados Distribuídos

por

RICARDO AUGUSTO MANFREDINI

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

Profª. Drª. Taisy Silva Weber
Orientadora

Porto Alegre, julho de 2001.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Manfredini, Ricardo Augusto

Condução de Experimentos de Injeção de Falhas em Banco de Dados Distribuídos / por Ricardo Augusto Manfredini – Porto Alegre: PPGC da UFRGS, 2001.

70f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2001. Orientador: Weber, Taisy Silva.

1. Injeção de Falhas 2. Sistemas Distribuídos 3. Tolerância a Falhas I. Silva, Taisy. II. Título.

Agradecimentos

Várias pessoas de forma direta ou indireta contribuíram para a realização deste trabalho, porém algumas tiveram vital importância:

- À minha esposa e companheira Leonora, pela solidariedade, compromisso, cumplicidade e tolerância.
- À minha orientadora Taisy, que me acolheu como orientando e acreditou na conclusão deste trabalho, agradeço as problematizações, contrapontos e correções de rumos.
- Os colegas do grupo de pesquisa Luis Cláudio e Paulo Ricardo, pela troca constante de experiências, descobertas e incentivos.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

2PC	Two-Phase Commit
3PC	Three-Phase Commit
ACID	Atômica, Consistente, Isolada e Durável
BD	Banco de Dados
BDD	Bando de Dados Distribuído
BI	Before Image
ComFIRM	Comunication Fault Injection through OS Resources
COTS	Common Off-The-Shelf
DB	Database
DRDA	Distributed Relation Data Architecture
EM	Experiment Manager
FIDe	Fault Injection via Debugging
GCC	GNU C Compiler
GerPro-TPC	Gerador de Carga Progress especificação TPCc
GIR	Gerenciador de Injeções e Resultados
GLP	GNU Public Licence
GNU	Gnu not Unix
LAN	Local Area Network
MQTh	Maximum Qualified Throughput
OLTP	Online Transaction Processing
OMG	Object Management Group
PDA	Personal Digital Assistant
PFI	Protocol Fault Injection
SGBD	Sistema Gerenciador de Banco de Dados
SGBDD	Sistema Gerenciador de Banco de Dados Distribuído
SQL	Structured Query Language
TCP	Transmission Control Protocol
TPC	Transaction Processing Performance Council
TpmC	Transaction Processed per Minute
TC	Transaction Client

TPS	Transações Por Segundo
XML	Extensible Markup Language
WAN	Wide Area Network

Resumo

O presente trabalho realiza uma validação experimental, através da técnica de injeção de falhas por *software*, de sistemas de informações que utilizam gerenciadores de banco de dados distribuídos comerciais. Estes experimentos visam a obtenção de medidas da dependabilidade do SGBD utilizado, levantamento do custo de seus mecanismos de tolerância a falhas e a real aplicabilidade de SGBDs comerciais em sistemas de missão crítica. Procurou-se avaliar e validar as ferramentas de injeção de falhas utilizadas, no caso específico deste trabalho a ComFIRM e o FIDe.

Inicialmente são introduzidos e reforçados os conceitos básicos sobre o tema, que serão utilizados no decorrer do trabalho. Em seguida são apresentadas algumas ferramentas de injeção de falhas em sistemas distribuídos, bem como os modelos de falhas em banco de dados distribuídos. São analisados alguns estudos de aplicação de ferramentas de injeção de falhas em bancos de dados distribuídos.

Concluída a revisão bibliográfica é apresentado o modelo de *software* e *hardware* que foi implementado, destacando o gerador de cargas de trabalho GerPro-TPC e o gerenciador de injeções e resultados GIR. O GerPro-TPC segue as especificações TPC-c para a simulação de um ambiente transacional comercial padrão e o GIR realiza a integração das ferramentas de injeção de falhas utilizadas, bem como a elaboração do cenário de falhas a injetar e a coleta dos resultados das falhas injetadas.

Finalmente são descritos os experimentos realizados sobre o SGBD PROGRESS. São realizados 361 testes de injeções de falhas com aproximadamente 43.000 falhas injetadas em experimentos distintos. Utiliza-se dois modelos de falhas: um focado em falhas de comunicação e outro em falhas de *hardware*. Os erros resultantes das falhas injetadas foram classificados em erros ignorados/mascarados, erros leves, erros graves e erros catastróficos. Dos modelos de falhas utilizados as que mais comprometeram a dependabilidade do SGBD foram as falhas de *hardware*. As falhas de comunicação somente comprometeram a disponibilidade do sistema alvo.

Palavras-chaves: injeção de falhas, banco de dados distribuídos, tolerância a falhas em banco de dados distribuídos, recuperação de banco de dados.

TITLE: “CONDUCTION OF EXPERIMENTS OF FAULT INJECTION IN DISTRIBUTED DATABASE”

Abstract

The present work emphasizes the experimental validation of mission critical information systems under faults. The experiments here described apply software fault injection techniques to validate information systems supported by distributed COTS DBMS. These experiments aim to measure the dependability of the target DBMS, the cost of its recovery mechanisms and the real applicability of COTS DBMS in mission critical systems. The experiments also aim to evaluate and validate the fault injection tools used, in the specific case of this work ComFIRM and FIDe.

The initial chapter introduces the basic concepts on fault injection. The next chapters present some tools of fault injection applied to distributed systems, as well as the most common fault models in distributed databases. Some studies reporting the application of fault injection tools in distributed databases are also analyzed.

The final chapters describe the experiments realized on the PROGRESS DBMS, chosen as the target DBMS. 361 tests of fault injection were done with approximately 43.000 faults injected in different experiments. Two fault models are used: one focused in communication faults and another in hardware faults. The resulting errors due the injected faults were classified in unknown/masked errors, light errors, serious errors and catastrophic errors. The experiments show that, among the inject faults chosen according the fault models, the hardware faults is the type that most affects the dependability of the target DBMS. The experiments also show that the communication faults only affect the availability of the target system.

Keywords: fault injection, distributed database, fault tolerance in distributed database, database recovery.

1 Introdução

Atualmente os sistemas comerciais vêm crescendo em muito na sua complexidade, tanto em *software* como em *hardware*. Tais sistemas estão cada vez mais distribuídos em diversas plataformas de equipamentos, sistemas operacionais, interfaces, aplicativos, SGBDs, etc. Isto acarreta um alto grau de relacionamento entre seus componentes, sendo um campo fértil para a ocorrência de falhas, tornando muito difícil a avaliação de sua confiabilidade e disponibilidade.

A validação das propriedades de confiabilidade dos sistemas de computadores é intrinsecamente complexa e a crescente complexidade destes tendem 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 [COS99]. 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 específica validação dos mecanismos de tratamento de falhas, permitindo estimar meios de tolerância a falhas, como cobertura de falhas e latência de erros [ARL90]. Muitas técnicas já foram propostas para a injeção de falhas e normalmente subdividem-se em: técnica de injeção baseada em *hardware* que injeta fisicamente falhas no sistema alvo, técnica de simulação que simula um modelo do sistema alvo, e finalmente, a técnica que emula falhas de *hardware* e erros através de *software*.

1.1 Objetivos

Este trabalho tem como objetivos a aplicação das técnicas de injeção de falhas por *software* em SGBDD (sistemas gerenciadores de bancos de dados distribuídos), 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 objetiva a validação das ferramentas de injeção de falhas utilizadas[1], realimentando o processo de desenvolvimento e aprimoramento destas ferramentas através dos resultados obtidos e dificuldades encontradas pela sua utilização na condução dos experimentos.

1.2 Banco de Dados Alvo

A escolha do SGBD Progress deve-se a sua larga utilização como banco de dados embutido (*embedded)* em aplicações COTS (*common off-the-shelf* – pacote de *software*

[1] FIDe e ComFIRM desenvolvidas no PGCC da UFRGS

comercial). Segundo estudos divulgados pelo Gartner Group/Dataquest (importante instituto de pesquisa do Estados Unidos da América) de julho de 1999 [GAR99], o SGBD Progress é líder mundial pelo terceiro ano consecutivo na categoria de banco de dados embutido. Considerando-se como banco de dados embutidos aquele que está completamente contido dentro de outro software, onde freqüentemente o usuário desconhece a utilização deste SGBD.

O Progress utiliza técnicas de tolerância a falhas que estão presentes na quase totalidade dos SGBDs comerciais, com os mecanismos que as implementam podendo ser desativados, o que facilita o cálculo de seus custos bem como a avaliação da eficiência dos mesmos.

1.3 Organização do Texto

O texto está organizado de forma a introduzir o assunto da injeção de falhas no capítulo 2. Neste capítulo serão enfatizadas as diferentes técnicas de injeção de falhas, por *hardware*, por simulação e por *software*. São ainda apresentadas algumas ferramentas de injeção de falhas [KRI96][DAW96a, DAW96b][CAR95][FAB2000][GON2001] algumas clássicas como a FIESTA, OCHESTRA e XCEPTION, e ainda a ComFIRM e FIDe que foram utilizadas na condução dos experimentos desta dissertação.

No capítulo 3 é feita uma explanação sobre a ocorrência de falhas em SGBDD e como agem seus mecanismos de recuperação sob este modelo de falhas. E, finalizando este capítulo, são apresentados três experimentos de injeção de falhas em SGBD distribuídos comerciais. Estes experimentos focam a injeção de falhas em diversas vias, em *buffers* de memória, sistemas de comunicações e interfaces, servindo de base para esta dissertação.

No capítulo 4 são descritos os ambientes de *software* e *hardware* utilizados nos experimentos desenvolvidos, enfatizando-se o gerador de carga de trabalho PRO-TPCc (Progress modelo TPC-c) e o gerenciador de injeções e resultados GIR especialmente implementados para esta dissertação.

No capítulo 5 é calculado o custo computacional dos mecanismos de injeção de falhas presentes no SGBD Progress e descrevem-se os três experimentos realizados: um experimento manual, sem a utilização de ferramentas de injeção de falhas, um experimento simulando-se falhas de comunicação utilizando-se a ferramenta ComFIRM, e o experimento simulando falhas de *hardware* com a utilização da ferramenta FIDe.

No capítulo 6, este trabalho é finalizado resumindo-se o que foi realizado para a obtenção dos resultados por esta dissertação, os problemas enfrentados e as soluções encontradas. Fazem-se, também, apontamentos para a continuidade deste trabalho.

2 Introdução à Injeção de Falhas

Este capítulo apresenta uma visão global sobre injeção de falhas, apresentando suas principais definições. O leitor familiarizado com os conceitos básicos da área não encontrará neste capítulo nenhuma nova contribuição ao tema.

Considerando-se que sistemas computacionais são compostos por uma série de componentes de *hardware* e *software*, que podem falhar eventualmente, e estas falhas podem levar o sistema a apresentar algum defeito, ou seja, o serviço oferecido pelo mesmo não está de acordo com o que foi especificado [LAP92], estes defeitos, quando em sistemas utilizados para o controle de atividades críticas, podem acarretar grandes perdas econômicas ou de vidas humanas. As falhas que geram erros que podem levar a defeitos, normalmente são ocasionadas por mau funcionamento do *hardware* ou algum componente de *software* [LAP92].

Falhas de *hardware* que ocorrem durante a operação dos sistemas são classificadas principalmente pela sua duração, em:

Permanentes: causadas por falhas irreversíveis dos componentes, por exaustão do seu uso, falhas de projeto e fabricação ou ainda pelo seu mau uso. Estas falhas somente podem ser reparadas pela troca do componente ou seu conserto.

Transientes: são ativadas por condições ambientais como flutuação de voltagem, distúrbios eletromagnéticos ou radiação. Normalmente estas falhas não danificam os componentes envolvidos, mas podem levar o sistema a um estado errôneo ou inesperado. Estes tipos de falhas são muito difíceis de detectar e, segundo vários estudos [CAR98], ocorrem com muito mais freqüência que as permanentes.

Intermitentes: Ocorrem devido a uma instabilidade de *hardware* ou variação de seu estado. Podem ser consertadas pela troca do hardware ou ainda um reprojeto do mesmo.

Falhas de *hardware*, temporárias ou permanentes, que afetam um computador isolado podem se propagar pela rede em um sistema distribuído afetando vários outros nós desta rede. Nesses casos para evitar se deter no detalhe de um nodo, usa-se uma abstração de maior nível que classifica as falhas em sistemas distribuídos como falhas de comunicação. Nessa visão podemos adotar o modelo de Cristian [CRI86]: *crash*, omissão, temporização e bizantinas.

As falhas de *software* ocorrem, normalmente, por erros de especificação, de projeto ou de codificação. Muitos engenheiros de *software* afirmam que “um *software* é livre de erros somente até o próximo erro ser encontrado”. Muitas destas falhas podem ficar latentes e só ocorrerem durante a operação do sistema, especialmente quando sob pesadas ou não usuais cargas de trabalho ou sob determinados intervalos de tempo. Os *bugs* de natureza

não-determinísticas são os mais difíceis de se eliminar por verificação, validação ou teste, e, normalmente, os sistemas os contêm durante todo o seu ciclo de vida.

Tem-se o costume de atribuir a causa das falhas dos sistemas às falhas de *software* ou falhas de *hardware* permanentes, pela dificuldade de se identificar as falhas de *hardware* transientes e intermitentes [CAR98].

Considerando-se que as falhas podem ter as mais diversas origens, a injeção de falhas se constitui numa importante forma de avaliar e validar os mecanismos de tolerância a falhas em sistemas de computadores.

2.1 Técnicas de Injeção de Falhas

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 validá-los em presença de entradas particulares para as quais eles foram desenvolvidos para tratar: as falhas [MAR95].

Pode-se caracterizar a injeção de falhas por um domínio de entradas e um domínio de saída. O domínio de entrada corresponde a um conjunto de falhas a injetar ***F*** e um conjunto de ativações ***A*** que especifica as entradas destinadas a exercitar o sistema e, em conseqüência, ativar as falhas injetadas. O domínio de saída é caracterizado por um conjunto ***R*** de dados coletados e por um conjunto de medidas ***M*** derivadas da análise e tratamento dos conjuntos ***F, A*** e ***R***. Os conjuntos ***FARM***, figura 2.1, constituem os principais atributos da injeção de falhas.

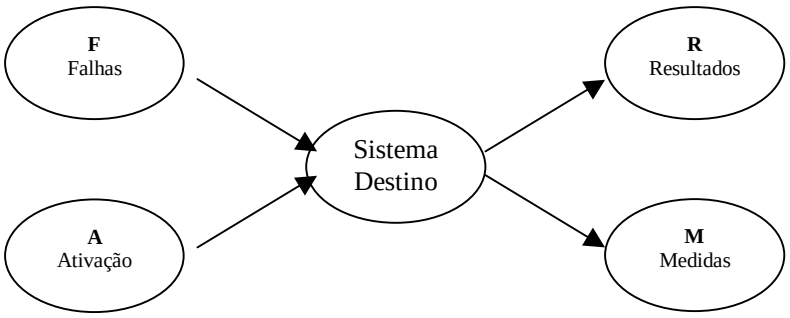


FIGURA 2.1 - Conjunto FARM

A injeção de falhas normalmente é realizada seguindo três principais técnicas: as baseadas em simulação, *hardware* ou *software*.

2.1.1 Injeção de Falha por Simulação

Nesta técnica, falhas são introduzidas em um modelo do sistema. Este modelo pode representar o comportamento ou a estrutura do sistema. As falhas podem ser aplicadas por

módulos especialmente construídos para este fim [CZE90], ou através de alterações do modelo original. As falhas são injetadas no modelo com a finalidade de analisar o processo de ativação das mesmas e a propagação dos erros, para avaliar o comportamento do modelo de tolerância a falhas.

As principais vantagens desta técnica são [SOT97]:

- pode ser aplicada durante o desenvolvimento do sistema, facilitando a detecção precoce das falhas;
- permite alto controle e observação da falha, pois permite que se selecione onde, quando e por qual período a falha será injetada.

Por outro lado, as desvantagens são:

- alto custo associado à simulação o que limita a porção do *hardware* e *software* que será modelado;
- dificuldade de transposição dos resultados da simulação para a prática, pois os sistemas reais normalmente não comportam-se nas falhas de forma idêntica a seus modelos.

2.1.2 Injeção de Falha por *Hardware*

Nesta técnica as falhas são injetadas fisicamente a um protótipo de *hardware*, *software* ou ambos. Elas são aplicadas por dispositivos especialmente construídos para este fim, que interagem com o sistema em teste.

Normalmente, neste enfoque, as falhas estão subdivididas em duas categorias:

- injeção de falha por hardware com contato, também conhecido como *pin-level*, que consiste em injetá-las fisicamente, geralmente sobre os pinos dos circuitos integrados [MAD93],
- injeção de falha por hardware sem contato, em que o injetor não mantém um contato físico direto sobre o sistema alvo, os componentes de hardware são submetidos a bombardeios de íons pesados [GUN89] ou ainda a aplicação de radiações eletromagnéticas.

Estas técnicas de injeção de falhas têm a vantagem de injetar falhas de hardware reais. Por outro lado, podem danificar os componentes sob teste, e como requerem o uso de componentes especiais de hardware, seu uso é focado a um determinado componente, possuindo uma alta integração com o mesmo. Também trazem consigo a dificuldade de monitoramento das falhas dentro de componentes digitais complexos como microprocessadores.

2.1.3 Injeção de Falhas por Software

Técnica onde as falhas são logicamente introduzidas, através de software específico, no sistema alvo, com o objetivo de emular as conseqüências de falhas físicas. Este procedimento apresenta algumas vantagens em relação às outras técnicas apresentadas:

- não necessita de equipamentos especiais de hardware;
- baixo custo;
- pode ser aplicada na fase de testes, pois não necessita que o hardware, em que irá atuar, esteja disponível;
- possibilita maiores facilidades no controle e observação do sistema durante os testes;
- não há risco de danificar os componentes sob teste,
- maior portabilidade, pois normalmente não precisam ser específicas a um determinado sistema alvo.

Esta técnica é adequada para a validação de mecanismos de tolerância à falha implementada por software, por possuir a capacidade de injetar falhas específicas que permitem ativar estes mecanismos, o que não pode ser garantido na injeção física de falhas.

Em contrapartida esta técnica normalmente embute em si algumas grandes desvantagens:

- o injetor de falhas pode causar um grande impacto sobre o sistema alvo, que pode alterar seus resultados devido ao *overhead* causado pela inserção do código do injetor de falhas,
- ocasiona dificuldade para implementar alguns tipos de falhas devido às limitações da maioria dos modelos.

Apesar das desvantagens, as vantagens acima citadas aliadas à flexibilidade da escolha do modelo de falhas, definiram ser essa a técnica a ser utilizada nessa dissertação.

2.2 Ferramentas de Injeção de Falhas

Nesta seção são apresentadas algumas ferramentas de injeção de falhas em sistemas distribuídos que implementam a técnica de injeção de falhas por *software*.

Entre as ferramentas destaca-se a ComFIRM [LEI2000] e FIDe [GON2001], em desenvolvimento no PGCC da UFRGS, que foram utilizadas na condução de alguns experimentos os quais serão mostrados nos capítulos 4 e 5.

A ferramenta FIESTA [KRI96] é descrita por tratar-se de uma ferramenta clássica de injeção de falhas. A ORCHESTRA [DAW96a] [DAW96b] serviu como motivadora para o desenvolvimento da ComFIRM e a XCEPTION [CAR95] pela sua utilização em trabalhos de validação de tolerância a falhas em SGBD.

2.2.1 FIESTA

FIESTA (**F**ault **I**njection for **E**mbedded **S**ystem **T**arget **A**pplications) [KRI96] ferramenta desenvolvida para o sistema operacional de tempo-real VxWorks. VxWorks é um sistema operacional comercial de tempo-real que roda sob a placa MC68040. FIESTA foi desenvolvida para avaliar a confiabilidade e as propriedades de correção de aplicações de controle em tempo-real rodando sob o VxWorks.

A plataforma (figura 2.2) *host* foi originalmente desenvolvida em *workstations* SUN rodando Solaris 2.5.1 e posteriormente migrada para máquinas Intel rodando LINUX pela facilidade e rapidez de desenvolvimento sob este sistema operacional. A plataforma *target* consiste em uma placa MC68040 rodando VxWorks.

O *front end* da ferramenta é a interface gráfica do sistema desenvolvida em Tcl/Tk (conjunto de objetos gráficos). Ele implementa o gerenciador de injeção de falhas (FI MGMT), onde são definidos os valores de todos os parâmetros relevantes para a injeção de falhas, e ainda, o usuário pode escolher quais funções de injeção de falhas serão utilizadas. Também implementa todo o ambiente de monitoramento das falhas injetadas e as respostas da plataforma *target*.

O depurador GDBM68K é uma versão modificada do gdb (distribuído por GNU Free Software Fundation) para rodar sob o MC68040. Como o ambiente alvo da FIESTA é para sistemas de tempo-real, o depurador foi projetado para interferir o mínimo possível na execução do sistema. Ele é capaz de realizar as seguintes operações:

- Definir ou excluir um breakpoint;
- Examinar e modificar registros na memória;
- Examinar e modificar estruturas de dados do kernel e rotinas de bibliotecas;
- Executar passo a passo instruções e continuar;
- Executar a aplicação;
- Capturar exceções e reportá-las para o usuário;
- Definir ou excluir *breakpoints* em interrupções ativadas e dispositivos.

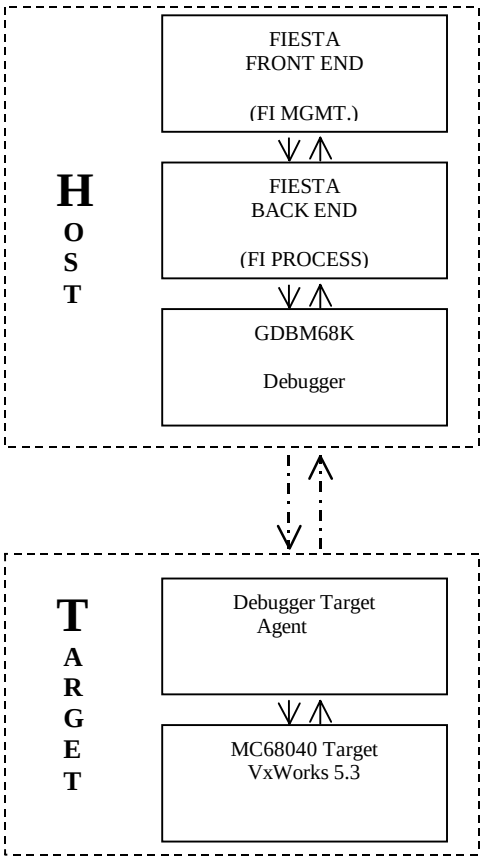


FIGURA 2.2 - Ambiente FIESTA

O *back end* implementa os processos de injeção de falhas (FI PROCESS) e foi completamente desenvolvido em C++. Os processos de ativação de falhas nada mais são que ativações de operações do GDBM68K. Este sistema originalmente suporta os seguintes modelos de falhas:

- Falhas Transientes de Endereçamento (modela erros de controle de fluxo);
- Falhas Transientes de Registradores;
- Mutação Randômica de código executável.

Como a FIESTA foi implementado em uma arquitetura aberta, novos modelos de falhas podem facilmente ser introduzidos.

A principal proposta da FIESTA é servir como arquitetura base para o desenvolvimento de ferramentas de injeção de falhas em quaisquer sistemas comerciais que suportem um bom depurador.

A ferramenta FIDe, que está descrita na subseção 2.2.5 e que é usada nesta dissertação, também baseia-se em recursos de depuração, mas ao contrário de FIESTA, não permite inserção de *breakpoints* e execução passo-a-passo.

2.2.2 ORCHESTRA

ORCHESTRA consiste num ambiente de injeção de falhas para teste de tolerância a falhas em sistemas de tempo-real distribuídos [DAW96a] [DAW96b]. Ele constitui um *framework* baseado em *scripts* para: sondagem e injeção de falhas, avaliação e validação de tolerância a falhas em protocolos distribuídos.

Foi originalmente desenvolvido para o sistema operacional de tempo real Mach e posteriormente foi portado para os sistemas operacionais SunOs e Solaris da Sun Microsystem.

ORCHESTRA visa primordialmente ser uma ferramenta para teste de aplicações distribuídas e protocolos de comunicações, portável para diferentes plataformas injetando falhas na pilha do protocolo. Um aspecto importante de sua arquitetura é a clara separação entre o mecanismo de injeção de falhas do protocolo alvo e o código dependente da plataforma. O módulo de injeção de falhas é dividido entre as partes que dependem e independem do protocolo.

A parte independente do protocolo consiste da parte que realmente injeta falhas e suas estruturas de dados. Também inclui a interface do usuário para geração dos *scripts* de injeção de falhas. Estes *scripts* podem ser especificados através da linguagem Tcl ou via diagramas de transição de estado. A parte dependente do protocolo consiste de código "grudado" na pilha do protocolo de comunicação. Este código implementa a interface entre o mecanismo de injeção de falhas e as camadas do protocolo, possuindo também, rotinas especiais de troca de mensagens do mecanismo.

A injeção de falhas é feita no nível das mensagens do protocolo. As mensagens que são enviadas e recebidas pelo protocolo alvo são interceptadas, manipuladas ou, ainda são acrescidas de mensagens aleatórias injetadas no sistema (figura 2.3).

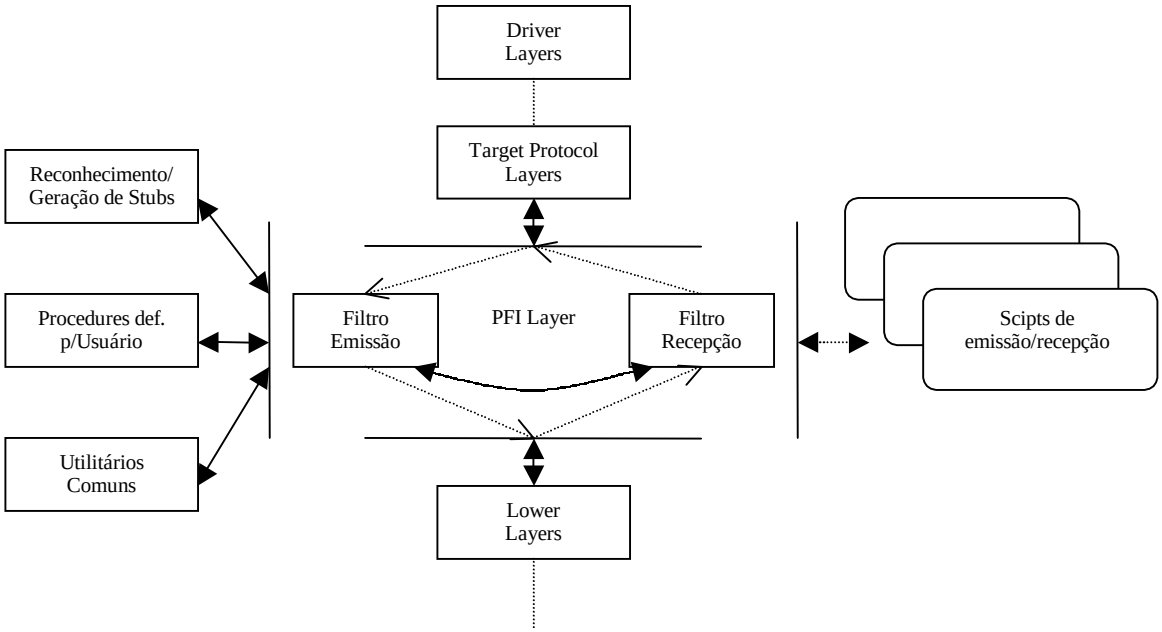


FIGURA 2.3 - Pilha do Protocolo com camada de PFI

A figura 2.3 demonstra que a camada de injeção de falhas (PFI Layer) é inserida entre duas camadas da pilha do protocolo de comunicação. A PFI pode interceptar mensagens que passam e pode manipulá-las. Por exemplo, a camada de PFI pode inserir mensagens específicas, atrasar uma mensagem por um pré-determinado intervalo de tempo, retransmitir uma mensagem, modificá-la no conteúdo ou, ainda, excluí-las. Basicamente a camada de PFI roda dois *scripts*, o **filtro de emissão** que é executado toda vez que uma mensagem é emitida pelo protocolo e o **filtro de recepção** que é ativado sempre que uma mensagem é recebida. Estes *scripts* realizam três tipos de operações sobre as mensagens que trafegam pela pilha do protocolo:

1. **Filtragem de Mensagens:** para interceptar e examinar uma mensagem.

2. **Manipulação de Mensagens:** para reter, retardar, reordenar, duplicar ou modificar uma mensagem.

3. **Injeção de Mensagens:** para sugerir que novas mensagens sejam introduzidas no sistema.

O pacote de **reconhecimento/geração de stubs** é invocado para determinar o tipo de mensagem sempre que ela é interceptada pelo PFI. Uma interface entre o ***script* de emissão/recepção** para o **reconhecimento/geração de stubs** existe para determinar quais tipos de pacotes serão tratados. E ainda, o **script de emissão/recepção** pode usar o pacote de **reconhecimento/geração de stubs** para gerar certos tipos de mensagens na camada de PFI.

Um aspecto importante considerado por ORCHESTRA é o de procurar não intervir muito sobre o sistema alvo, evitando ao máximo o “efeito Heisenberg” em que ferramentas de injeção de falhas podem ocasionar *overhead* do sistema, principalmente em sistemas de tempo-real. O mecanismo usado por ORCHESTRA é a alocação de recursos extras de CPU para as atividades de comunicação, já que o *overhead* seria sobre os sistemas de comunicação.

ORCHESTRA tem sido utilizado para testar mecanismos de tolerância a falhas em muitos sistemas comercias e de pesquisa, incluindo as seis principais implementações de TCP existentes, protocolos de comunicação de grupo e sistemas de áudio conferência em tempo-real.

2.2.3 Xception

Xception [CAR95] é um *software* de injeção e monitoração de falhas. Xception usa basicamente depuradores e monitores de desempenho existentes nos modernos processadores para injetar falhas nos sistemas alvo e monitorar o impacto das mesmas. Falhas são injetadas com uma mínima interferência com a 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, foi portada para outros processadores (SPARC, PowerPc 604, Pentium, etc) e outros sistemas operacionais (Solaris, AIX, Windows NT).

Xception está dividido em três módulos:

1. Um módulo injetor que está *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 grande 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 2.4.

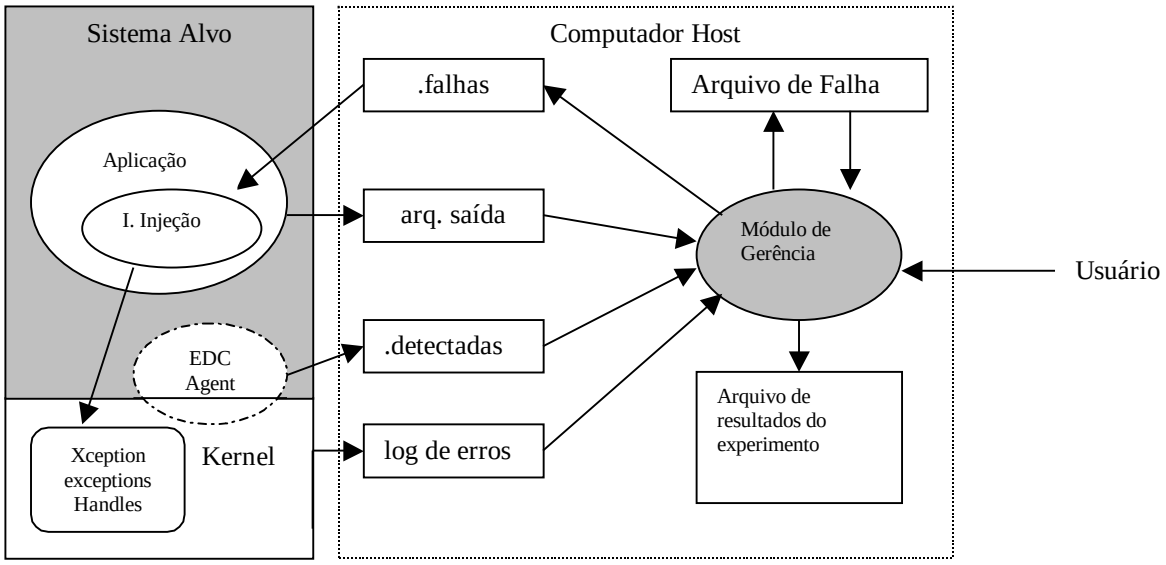


FIGURA 2.4 - Estrutura do Xception

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. Ainda, o *kernel* gera mensagens de erros 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 2.4, as quais, mais tarde, são salvas pelo Xception no arquivo de resultados do experimento.

As mensagens do status da injeção são enviadas para o *host* e gravadas no arquivo de log imediatamente antes da falha ser injetada, isto é, antes que processo de exceção retornar 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.

2.2.4 ComFIRM

A ferramenta ComFIRM [LEI2000] (*Communication Fault Injection through OS Resources Modification*), desenvolvida no PGCC da UFRGS, é um injetor de falhas de comunicação bastante abrangente e flexível, que se situa dentro do núcleo do sistema

operacional GNU/Linux, no ponto mais baixo do tratamento de mensagens pelo subsistema de rede, na camada independente de dispositivos como mostra a figura 2.5.

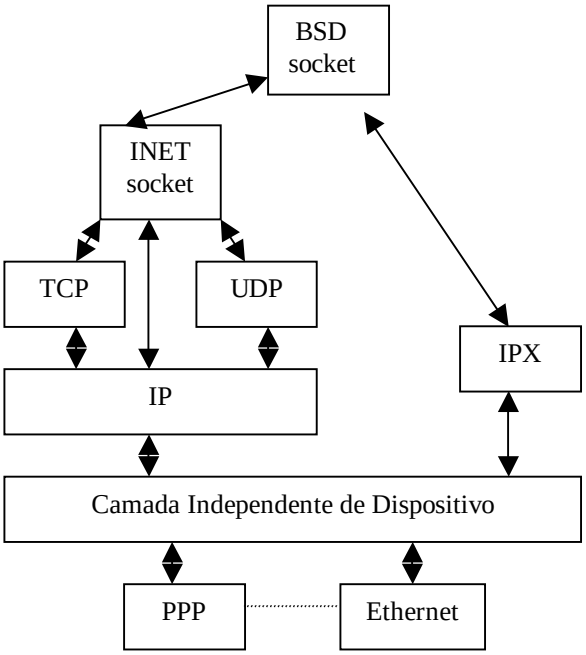


FIGURA 2.5 - Organização em camadas dos protocolos de comunicação suportados no GNU/Linux

Para sua utilização é necessário uma pequena modificação no núcleo do sistema operacional, onde se introduz as chamadas da ferramenta. Isto não constitui um problema, haja visto que todas a distribuições do Linux trazem seu código fonte. Basicamente a implementação da ConFIRM alterou as rotinas do *kernel* **dev_queue_xmit** que transmite e a **netif_rx** que recebe pacotes do subsistema de rede. As alterações nestas rotinas apenas verificam se a ferramenta está ativada e se existem regras de transmissão ou recepção, chamando as rotinas originais diretamente caso alguma condição não seja satisfeita. Caso as condições sejam satisfeitas, é ativada a ferramenta e as rotinas de injeção de falhas são chamadas.

A ferramenta foi projetada para permitir a seleção e manipulação de pacotes de rede, implementando vários recursos que são utilizados na definição de regras para a injeção de falhas bem como para seu controle e uso. Estes recursos são quatro arquivos virtuais que se localizam no diretório também virtual **/proc/net**, são eles:

1. **ComFIRM_Log:** arquivo somente de leitura, em que a ferramenta disponibiliza as informações sobre seu funcionamento;

2. **ComFIRM_Control:** arquivo somente de escrita, onde são gravados os comandos para inicializar, terminar e reinicializar a ferramenta;

3. **ComFIRM_TX_RULES:** arquivo que contém as regras de injeção de falhas na transmissão dos pacotes, e,

4. **ComFirm_RX_RULES:** arquivo que contém as regras de injeção de falhas na recepção dos pacotes.

As regras para injeção de falhas são formadas pela concatenação de instruções que são interpretadas pela ferramenta. É necessário que estas instruções sejam codificadas em *bytes*, já que a ferramenta situa-se dentro do núcleo do sistema e a sua interpretação deve ser a mais rápida possível.

Existe um número considerável de instruções que, basicamente, estão agrupadas em dois grandes grupos: as instruções de testes, que verificam se é o momento oportuno para a injeção de falhas e as de ação, que realizam a injeção de falhas propriamente ditas.

A ferramenta permite o retardo, descarte, duplicação ou a alteração do conteúdo dos pacotes transmitidos ou recebidos pelo subsistema de rede.

2.2.5 FIDe

O FIDe (*Fault Injection via Debugging*) [GON2001], ferramenta em desenvolvimento no PGCC da UFRGS, é baseado na depuração de programas pela sua execução até uma chamada de sistema (*syscall)*. Leva em consideração que praticamente todo o acesso a recursos do sistema operacional e o acesso a *hardware*, no sistema operacional GNU/Linux, se dá via chamadas do sistema operacional.

O FIDe utiliza a chama de sistema *ptrace*, que permite executar um processo de três diferentes maneiras: passo a passo, usando *breakpoints* ou executá-lo até a próxima chamada de sistema. As primeiras duas maneiras necessitam a modificação do código fonte dos programas, que na maioria das vezes não estão disponíveis. Baseado nestas premissas, utiliza-se a terceira maneira que parece ser a melhor escolha.

Esta ferramenta trabalha com um arquivo de configuração que define as regras que serão usadas durante a execução da aplicação alvo, determinando o cenário de falhas. Um cenário de falhas pode conter muitas regras, as quais são definidas pela seguinte sintaxe:

```
main_rule
        rule_when
                rule [reg | memo | user | param]
                rule [reg | memo | user | param]
                ...
        rule_when
                rule [reg | memo | usr | param]
                ...
```

A regra **main_rule** define qual *syscall* a ferramenta deve interceptar. Quando esta regra é satisfeita, ou seja, quando ocorre a chamada de sistema definida pela regra, o FIDe avalia a regra **rule_when** e determina se já é possível, pela avaliação dos temporizadores ou contadores, a injeção de falhas. Muitas regras **rule_when** podem ser definidas dentro de uma **main_rule**.

A regra **rule_when** pode ser baseada em tempo, fluxo ou pela ocorrência de determinados valores nos registradores de memória. Este tipo de regra também determina se a injeção ocorrerá na chamada ou no retorno da *syscall*.

Se uma regra **rule_when** é satisfeita e a ferramenta está pronta para injetar falhas, existem quatro possibilidades de ações que poderão ser tomadas:

1. **rule_reg:** troca os valores dos registradores;
2. **rule_memo:** troca o conteúdo da memória;
3. **rule_paran:** troca o conteúdo da memória apontado pelos registradores mais um deslocamento, e,
4. **rule_user:** troca dados do *user struct* do processo, como as informações do próprio processo, do seu ambiente e da sua execução.

As regras de trocas podem ser por incremento, decremento, atribuição e os operadores matemáticos mais (+) e menos (-).

A figura 2.6 mostra a arquitetura do FIDe. Inicialmente cria-se o arquivo texto com as regras **main_rules,** que contém o conjunto de falhas a serem injetadas. Posteriormente ativa-se a ferramenta da seguinte forma: **fide –d –f <processo que se deseja injetar falhas>**, onde o parâmetro **–d** faz a ferramenta gravar um arquivo com o *log* de suas ações e o **–f** força a ferramenta a também gravar no seu *log* todas a aberturas de arquivos realizadas pelo processo que se deseja injetar falhas.

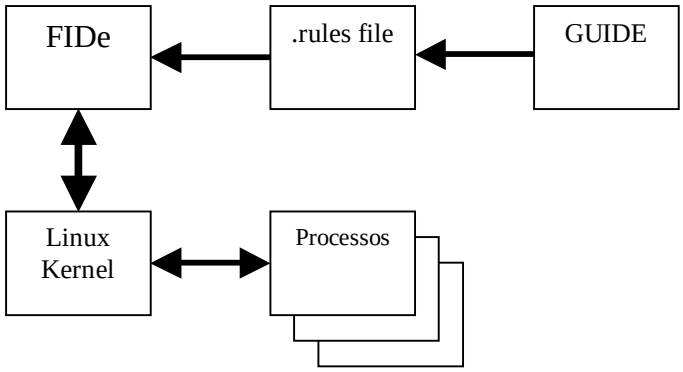


FIGURA 2.6 – Arquitetura do FIDe

O módulo do FIDe, chamado GUIDE, no período em que foram realizados os estudos desta ferramenta ainda não estava disponível. Este módulo será um *front end* para a geração de regras e cenários que constituirão o arquivo de regras da ferramenta.

O FIDe está sendo desenvolvido especialmente para esta dissertação, que por sua vez visa também realimentar o desenvolvimento do FIDe. Sendo uma ferramenta ainda em desenvolvimento algumas dificuldades tiveram que ser contornadas para o seu uso, como serão mostradas no capítulo 4.

3 Recuperação e Injeção de Falhas em Banco de Dados Distribuídos

Neste capítulo, na seção 3.1 é feito um resumo da recuperação de falhas em BDD. Na seção 3.2, são descritos três experimentos de injeção de falhas em BDD [GOS97] [COS98] [SAB98], os quais serviram como embasamento desta dissertação.

3.1 Recuperação em Banco de Dados Distribuídos

Recuperação de falhas em Banco de Dados (BD) consiste basicamente em restaurar o banco de dados para um estado consistente após a ocorrência de falhas. Quando uma falha ocorre, uma ou mais transações podem estar ativas e ainda não terem sofrido *commit*. Considerando-se que as transações em BD devem ser atômicas, o efeito dessas transações parciais deve ser removido (*undo*). Ou ainda, uma transação pode ter sofrido *commit* e uma falha ocorre antes de seus resultados terem sido gravados em disco, segundo o princípio que uma transação deva ser durável, esta deve ser refeita (*redo*).

3.1.1 Estado do Banco de Dados

O estado corrente de um banco de dados é a combinação de dados de três lugares diferentes (figura 3.1). Os locais de um BD são:

1. Em disco, a parte do BD residente em memória permanente contém todos os dados que foram gravados. Eles podem estar incompletos e inconsistentes visto que alguns dados que sofreram alterações podem ainda não terem sido gravados;

2. Em memória compartilhada, que contém todos os dados que foram recentemente alterados e ainda não gravados em disco (*db flush*);

3. No *log* de transações, o *log* de *undo/redo* contém todos os registros que foram recentemente alterados no BD. O *log* de transações está sempre incompleto porque velhos registros de *log* são descartados e seu espaço é reutilizado.

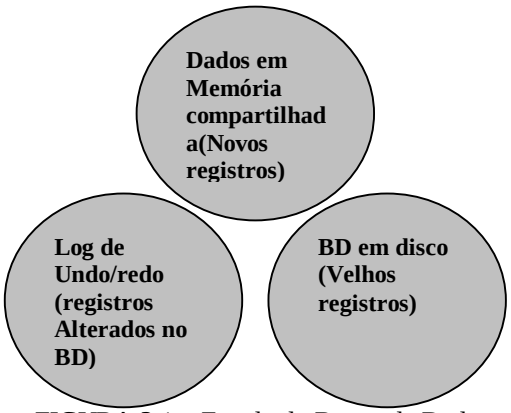


FIGURA 3.1 – Estado do Banco de Dados

O banco de dados somente estará em um estado consistente se considerarmos todos os três lugares possíveis de residência dos dados. O BD em disco somente estará num estado consistente após ele ter sofrido *shutdown*, onde todos os dados residentes em memória serão gravados em disco e suas entradas removidas do *log* de transações.

3.1.2 Ocorrência de Falhas em BD

Nos sistemas de bancos de dados distribuídos as falhas podem ocorrer basicamente nas seguintes áreas: falhas no nodo, falhas de comunicações entre os nodos e falhas de interfaces.

3.1.2.1 Falha de Nodo

Este tipo de falha ocorre quando um ou mais nodos que compõem o BD distribuído falha. Pode ocorrer quando o SGBD tenta gravar em memória permanente (*db flush*) os *buffers* que contém as transações sofridas pelo banco de dados ou ainda quando são gravados os *logs* das transações. Também pode ocorrer quando um ou mais nodos pararem de funcionar (*system down)*.

Quando este tipo de falha ocorre, mesmo sabendo que a maioria das implementações de SGBDs assumem um modelo *fail-stop*, o problema raramente é uma **falha total** (onde todos os nodos falham), normalmente é uma falha parcial e o seu tratamento depende muito do modelo de distribuição adotado.

Nos modelos de distribuição de dados e o de replicação assíncrona, a recuperação dar-se-á principalmente pela utilização dos princípios de atomicidade do *commited* presentes no protocolo *Two-Phase Commit Protocol* (2PC).

3.1.2.2 Falha de Comunicação

Ocorre quando uma falha de comunicação impede a troca de mensagens entre os diversos nodos que compõem o BD distribuído. Sua causa pode ser a corrupção de mensagens devido a ruídos eletromagnéticos, mau funcionamento de um *link*, rompimento físico do *link* ou, ainda, grandes atrasos de troca de mensagens, o que provoca *time-out* nos protocolos de comunicação.

Estas falhas normalmente são tratadas pelo protocolo de comunicação, pelas técnicas de detecção e correção de erro, retransmissão de mensagens ou por re-roteamento. Quando o protocolo de comunicação não pode recuperar estas falhas e fica impossibilitada a comunicação entre os nodos, o SGBD deverá usar a mesma estratégia de recuperação para as falhas de nodos.

Nos modelos de distribuição de dados e o de replicação síncrona, a recuperação dar-se-á, principalmente pela utilização do protocolo *Two-Phase Commit Protocol* (2PC). Já nos modelos que usam replicação assíncrona, como exceção ao modelo ponto-a-ponto, como as transações de atualização ocorrem em somente um nodo, a sua replicação para outros nodos poderá ser postergada até que a comunicação entre os nodos for restabelecida.

3.1.2.3 Falha de Interface

Nos SGBDs atuais é crescente a possibilidade de integrar código da aplicação de banco de dados com *software* de uso geral, como planilhas de cálculos, objetos CORBA[2] ou DCOM[3], processadores de texto, interfaces clientes, etc.

Esta integração normalmente utiliza uma das principais características do projeto orientado a objeto, que é o conceito do encapsulamento, com ênfase nos tipos abstratos de dados e isolamento da informação. Um objeto consiste de uma estrutura de dados e operações que agem sobre estes dados. A estrutura de dados interna somente pode ser acessada de fora do objeto por rotinas bem definidas, chamadas de **interfaces.**

Mas esta facilidade pode introduzir um novo tipo de erro nos SGBDs, que são as falhas entre o relacionamento destes objetos, sob os quais o SGBD não possui controle. Alguns objetos podem acessar diretamente a área de *buffer* do banco de dados tornando-a muito vulnerável e podendo acarretar **falhas de nodos**, ou ainda algum objeto externo a aplicação que está atualizando um nodo pode estar residindo num nodo diferente onde podem ocorrer **falhas de comunicação.**

Como falhas de interfaces podem ocasionar outras falhas, elas podem ser uma ponte para a propagação de falhas em todo o sistema distribuído [MIL94]. As falhas mais comuns neste modelo são: travamento do objeto, aborto, atraso de resposta, resposta incorreta ou omissão de mensagem.

3.1.3 O Algoritmo de *Two-Phase Commit* do SGBD Progress

A figura 3.2 exemplifica o algoritmo de 2PC presente no SGBD PROGRESS, que implementa este protocolo usando arquivos de *log* [BER87]. Para assegurar que suficientes informações estão presentes no *log* (para realizar operações de *undoing*), as seguintes regras devem ser usadas para gravar informações no *log:*

1. Quando o coordenador inicia o 2PC, ele identifica a transação gravando-a no seu BI (*log)* e instrui os outros BD que participam da transação para fazerem o mesmo;

2. Se um participante vota YES ou NO, ele grava no seu *log* esta informação antes de enviá-la para o coordenador;

[2] CORBA é um *framework* padrão para desenvolvimento de objetos distribuídos suportado pela OMG
[3] DCOM arquitetura proprietária da Microsoft, utilizando ActiveX, para objetos distribuídos

3. Antes de o coordenador enviar uma mensagem de *commit* para os nodos participantes, ele grava um registro de *commited* no seu *log*;

4. Se o coordenador envia uma mensagem de *abbort*, ele grava um registro de *abbort* no seu *log;*

5. Após receber uma mensagem de *commit* ou *abbort*, um nodo participante grava este registro no seu *log.*

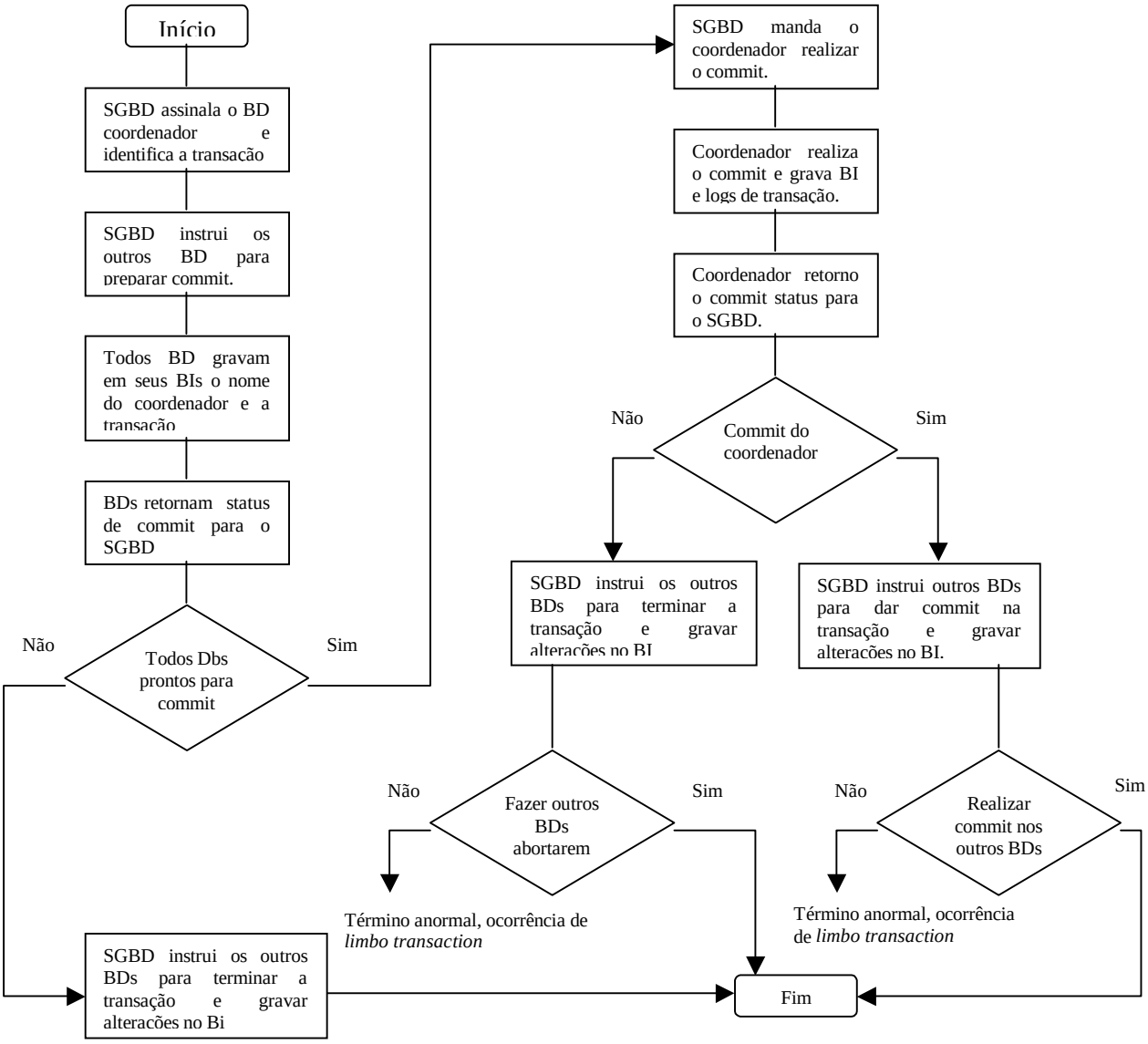


FIGURA 3.2 - Algoritmo do protocolo *two-phase commited* do SGBD Progress

Com estas regras bem implementadas, um nodo **S** pode realizar uma recuperação, em caso de falha, se ele for o coordenador desta transação pela presença do registro de

início do 2PC no seu *log*. Se este registro não estiver presente, o nodo somente poderá ser um participante da recuperação.

Num processo de recuperação, o coordenador verifica se contém em seu *log* um registro de *commit* ou *abort* da transação. Se este registro existir significa que o coordenador tomou uma decisão antes da falha e ele deverá implementar esta decisão. Caso contrário, ele decidirá pelo *abort* inserindo este registro no seu *log*. Para isto funcionar perfeitamente, é fundamental que os registros de *commit* sejam gravados no *log* antes do envio da mensagem de *commit*.

Um nodo participante, no processo de recuperação, também verifica se contém em seu *log* um registro de *commit* ou *abort* da transação. Se este registro existir significa que o participante tomou uma decisão antes da falha e ele deverá implementar este decisão. Se em seu *log* contém um registro de *yes* significa que o nodo falhou antes de votar ou votou *no*, neste caso deverá inserir um registro de *abort* em seu *log*. Se o *log* contiver registro *yes*, mas não contem um registro de *commit* ou *abort*, o nodo não pode unilateralmente decidir o que fazer (*commit* ou *abort*) e deverá usar um **Protocolo de Terminação** (normalmente *time-out*). Os métodos de gravar registros no *log* e de recuperação descritos acima asseguram a atomicidade da transação em banco de dados distribuídos [JAL94].

3.2 Injeção de Falhas em Banco de Dados Distribuídos

Cada vez mais utiliza-se SGBD comerciais (COTS – *common off-the-shelf*) para aplicações de missão crítica, por exemplo, controle de transmissão de energia e telecomunicações, que devem estar disponíveis 24x7x52 horas no ano. Portanto, eles devem estar providos de mecanismos eficientes de tolerância a falhas, tanto em nível da integridade dos dados bem como sua disponibilidade.

Quase a totalidade dos SGBD comerciais disponíveis oferecem suporte para recuperação de dados e tolerância a falhas, mesmo que as plataformas de *hardware* em que eles rodam não sejam tolerantes a falhas.

Apesar desta massiva utilização, poucos experimentos foram realizados para avaliar e validar as técnicas de tolerância a falhas presentes em SGBD comerciais.

Nesta seção serão descritos três experimentos [GOS97][COS98][SAB98] de injeção de falhas por software com utilização de SGBD.

3.2.1 Um experimento de Injeção de Falhas de Interfaces

Nos estudos realizados por Sudipto Ghosh [GOS97], foi identificado o modelo de falhas no nível dos componentes de interfaces, onde poderia ser possível injetar falhas. Estes estudos utilizaram a ferramenta TAMER (*Testing, Analysis and Measurement Environment for Robusteness*) [MIL94].

Os testes foram realizados num sistema distribuído com uma arquitetura *client-server*, figura 3.3, com a parte servidor contendo aproximadamente 80.000 linhas escritas

em C com SQL embutido rodando em plataforma UNIX. A parte cliente roda em plataforma Win/Microsoft. O servidor comunica-se com os clientes através de um subsistema de comunicações. O servidor está conectado a um banco de dados Oracle, onde todos os dados são armazenados.

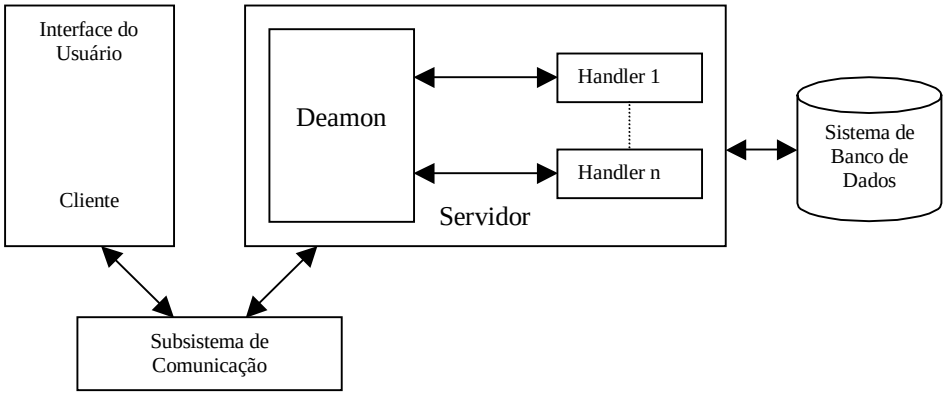


FIGURA 3.3 - Arquitetura *client-server*

Neste estudo de caso, primeiro identificam-se e selecionam-se as interfaces entre os componentes, são elas:

1. A interface cliente-servidor foi identificada como o mecanismo de comunicação entre o cliente e o servidor, e as primitivas utilizadas para esta comunicação são: *Log on*, *log off*, envio e recepção de mensagem;

2. A interface Servidor-Deamon-Handler é aquela que dependendo do serviço solicitado pelo cliente, o *deamon* do servidor chama um *handler* em particular que conterá alguma estrutura de dados utilizada pelo servidor.

Para cada interface identificada, foi definida uma falha a ser injetada. Foram examinados os parâmetros envolvidos na interface, e dependendo do seu tipo, foi selecionada a falha. Por exemplo, se o parâmetro é uma string de caracteres, foi testada com um *null* ou vazio caracter. A tabela 3.1 mostra as interfaces e tipo de algumas falhas que foram injetadas.

TABELA 3.1 - Erros Injetados por tipo de interface

Interface	**Primitiva**	**Parâmetro**	**Falha Injetada**
Cliente-Servidor	Lon On	Login	Null String vazia
		Password	Null Sting vazia
Servidor-Handler	Handler-1	Return code Return Code Request string Response string Redundant parameters	Código de erro (travamento) Falha de procura no BD String vazia Null Sting vazia
	Handler 2	Return code Return code Request string Response string Redundant parameters	Código de erro (travamento) Falha de procura no BD String vazia Null Sting vazia

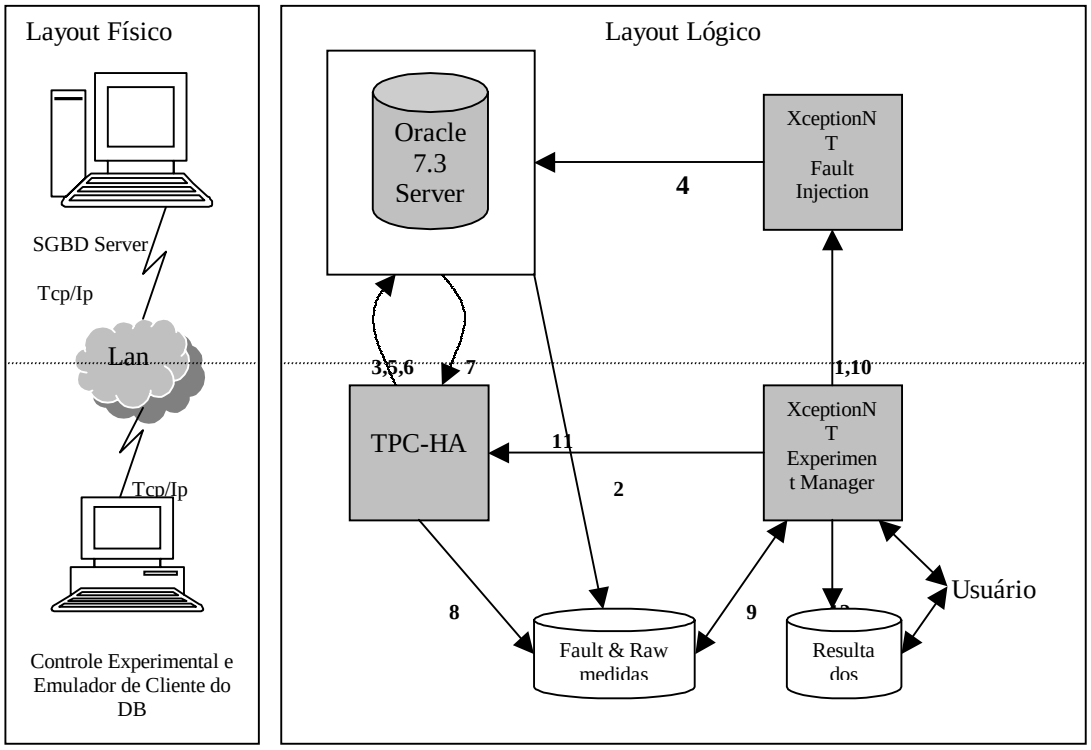
A principal contribuição deste estudo de caso foi a de propor um método de analisar o sistema como um todo, pois injetando-se falhas nas interfaces entre os componentes pode-se observar todo o comportamento do sistema quando da ocorrência de uma falha.

3.2.2 Delphos

O projeto Delphos [COS98, COS99] tem por objetivo primordial avaliar técnicas de tolerância a falhas em SGBDs comerciais, através de injeção de falhas. Este utilizou uma arquitetura cliente servidor de banco de dados, apesar do SGBD escolhido permitir distribuição.

Também são objetivos do projeto Delphos o estabelecimento de uma metodologia para a medição da confiabilidade de COTS SGBD e visualização do impacto de falhas em banco de dados. Além disto, este projeto propõe-se a providenciar uma base para o projeto de metodologias de programação defensivas que efetivamente utilizem as técnicas de tolerância a falhas presentes nos SGBDs, como transações e *checkpoints.*

A grande maioria dos mecanismos para se garantir confiabilidade presente nos COTS SGBDs, tratam de falhas permanentes e assumem um modelo de *fail-stop* para o sistema. Tem-se mostrado um exíguo interesse para falhas transientes de *hardware*.



1. Parâmetros de falhas
2. Inicio da carga de trabalho
3. Arquivos DB
4. Injeção de Falhas
5. Shutdown DB instância
6. Startup DB instâncias
7. "Probes" & medidas
8. Medidas de dados brutos
9. Dados brutos e falhas
10. Leitura do impacto da falha
11. Atividades de log e trace
12. Resultados processados

FIGURA 3.4 - Layout Testado

Delphos avalia o comportamento de COTS SGBDs na presença de falhas transientes. Estas falhas são injetadas utilizando-se a ferramenta Xception [CAR95]. O sistema alvo (figura 3.4) consiste de um SGBD Oracle 7.3 rodando sob o sistema operacional WindowsNT 4.0 sob plataforma de *hardware* Intel P6, e os clientes na rede rodam a ferramenta de TPC.

No estudo realizado em Delphos foram injetadas falhas nas *threads* do servidor de banco de dados. Não foram injetadas falhas na parte cliente nem na parte de comunicação.

A ferramenta de injeção de falhas (XceptionNT, modificação do Xception original para rodar sobre o sistema operacional WINNT), utiliza-se das características de *debugging* oferecidas pelos processadores Pentium para que o processo de injeção de falhas seja o mínimo intrusivo possível.

Foi utilizada a ferramenta TPC-A para avaliar o desempenho das transações OLTP (*On-Line Transaction Processing)* sob SGBD, nas quais foram injetadas falhas, pois ela constitui-se na prática um padrão de medição de desempenho.

O estudo realizado interessou-se principalmente no impacto das falhas sob a integridade dos dados e de sua disponibilidade.

A integridade dos dados é vista sob três níveis:

1. Nível da aplicação: uso de um conjunto de regras semânticas e testes de consistências especificados para o TPC *benchmark*;

2. Nível de integridade referencial: comparação das regras do banco de dados relacional (dicionário de dados) com os dados de fato armazenados;

3. Nível de arquivo: arquivos do banco de dados e *logs* são verificados da integridade de suas estruturas de dados internas.

A disponibilidade do banco de dados é vista como:

1. Tempo médio de recuperação do banco de dados visto pelo usuário final;

2. O tipo de recuperação sofrida pelo banco de dados (total ou parcial);

3. O esforço de recuperação, automática ou pelo administrador do banco de dados (manual).

Os resultados da injeção de falhas provêm da obtenção de dados em diferentes níveis. Desde o nível de máquina (*hardware*), do sistema operacional, do SGBD para o nível da aplicação. O ambiente de avaliação oferece, para o nível de máquina, leituras precisas da localização da falha injetada. No nível de sistema operacional, exceções, paradas, condições de travamento e código de retorno dos processos do banco de dados também são registrados no *log*. No nível do ambiente da aplicação TPC-A, as atividades são registradas no *log* para medir-se o tempo de suas execuções e as transações perdidas.

O principal objetivo deste estudo foi o de realizar uma avaliação experimental da confiabilidade e disponibilidade presentes em COTS SGBD. Combinado num *framework* uma ferramenta de injeção de falhas (Xception) que se propõem a ser a mínima intrusiva possível sob os sistemas alvos, e uma ferramenta de avaliação de desempenho (*benchmark*) padrão a TPC-A.

3.2.3 Um experimento com o Banco de Dados Distribuídos ClustRa

Em experiências realizadas por Maitrayi Sabaratnam [SAB98] sob o SGBD distribuído através de replicação ClustRa, foram avaliados os impactos das falhas nesses sistemas. Foram ainda avaliados os mecanismos de mascaramento de falhas pelas réplicas e ainda erros propagados para as réplicas.

Este experimento introduz um método de injeção de falhas genérico que poderia ser adotado por qualquer SGBD da fase de projeto/protótipo até a fase de desenvolvimento e teste, para avaliar a efetividade dos mecanismos de tolerância a falhas e sua influência

sobre o desempenho. Estas avaliações, quando realizadas no início do projeto, permitem aos desenvolvedores tomarem decisões sobre o projeto, bem como descobrir e corrigir falhas do mesmo em suas fases iniciais.

A implementação deste método exige pequenas alterações nos SGBDs, já que explora a interface de cliente existente nestes gerenciadores. A injeção de falhas neste estudo se propõe a ser a mínima intrusiva possível sobre o sistema alvo. Neste experimento o método da injeção de falhas é aplicado sobre a estrutura de dados presente na área de *buffer* de dados. Este método também pode ser aplicado a outras estruturas do SGBD como *log buffers*, *locks*, mensagens, etc. Os *buffers* de dados foram escolhidos pelo seu papel crucial na manutenção da integridade do SGBD.

São criados diferentes cenários de injeção de erros no SGBD ClustRa – versão 1.1. Os impactos destes erros são analisados bem como a degradação do desempenho e o tempo gasto pelo sistema no processo de recuperação de falha.

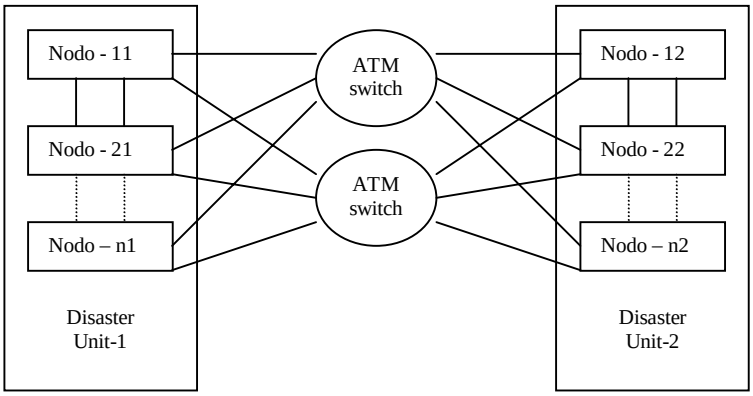


FIGURA 3.5 – Arquitetura do ClustRa

ClustRa provê alta disponibilidade, alto desempenho e tolerância a falhas para aplicações de tempo real. Constitui-se de um SGBD replicado rodando 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) e duplicadas. A duplicação é utilizada nos processos, dados e comunicação, para minimizar a indisponibilidade de serviços e perda de dados, conforme mostra a figura 3.5. 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.

Realizaram-se duas experiências descritas a seguir: na primeira foram injetadas falhas em uma área qualquer do *buffer* de dados. A posição inicial para a injeção de falhas, que venham a corromper os *buffers* de dados, é distribuída uniformemente na área de *buffer*. O número de dados corrompidos foi baseado em estudos feitos por Sullivan e

Chillarege [SUL91], mas foram ajustados para o estudo de caso. A distribuição do número de bytes que são corrompidos é: 60% 1-4 bytes, 35% 5-1024 bytes e 5% 1-9 KB.

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

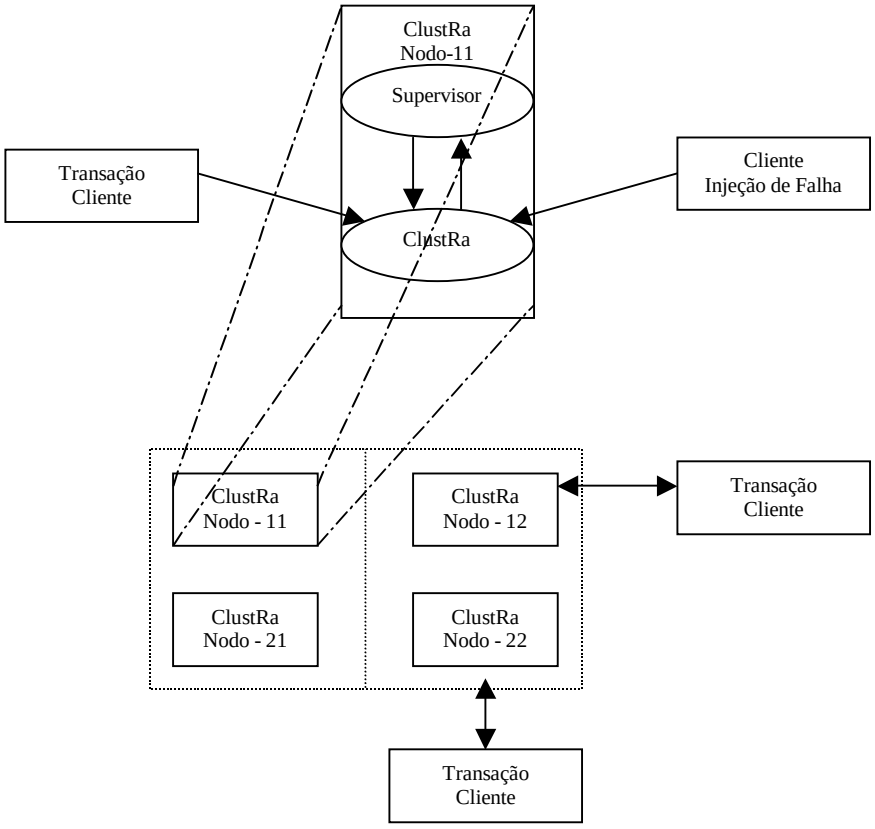
1. Eficiência da validação das estruturas de dados, ou;

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 são conhecidas, o componente mais fraco pode ser reforçado com a descoberta da melhor técnica para limitar o impacto do erro.

No segundo experimento, foram injetados erros específicos sobre componentes particulares na área de *buffer* de dados. Os erros possíveis de serem injetados são: *Blockid*, corrupção do identificador de bloco; *NoOfReccords*, corrupção do número de registros; *NoOfBytesFree*, corrupção do número de bytes livres no bloco; *HighWater*, corrupção do nível do buffer; *NextPointer*, corrupção dos ponteiros de blocos; *KeyDescriptor*, corrupção da chave do descritor do bloco; *AdmDescriptor*, corrupção do descritor do administrador do bloco; *NextLevelPoint*, corrupção do apontador para o próximo nível da b-tree que contém o *buffer*; *UserData*, corrupção da área de dados do *buffer*.

O experimento de injeção de falhas consiste em inicializar o SGBD, gerar uma certa carga de trabalho, injetar falhas, para o SGBD e analisar os *logs*. O experimento é administrado pelo gerenciador do experimento (EM – *experiment manager*). Os processos ClustRa são inicializados em quatro nodos, como ilustrado na figura 3.6, 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 dos tipos de erros acima citados 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 entregues pelo SGBD contra seu banco de dados interno para todas as transações. O SGBD registra em *log* todas as recuperações realizadas bem como o tempo para esta recuperação.



(o nodo-11 está ampliado).
Inicialmente, Nodo-11 e Nodo-12 estão ativos, e Nodo-21 e Nodo-22 são sobressalentes

FIGURA 3.6 – A configuração do experimento

Foram executadas 725 operações para a primeira experiência, 50 execuções para cada tipo de erro específico (*Blockid*, *NoOfReccords*, *NoOfBytesFree*, *HighWater*, *NextPointer*, *KeyDescriptor*, *AdmDescriptor*, *NextLevelPoint* e *UserData)* da segunda experiência e 50 execuções livre de erros, totalizando 1125 execuções. O tempo utilizado para cada execução leva 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 e

2. Eficiência de Recuperação.

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

Quanto à **cobertura de erros,** na primeira experiência, foram detectados 61% das 725 falhas injetadas. Foram mascaradas 97,5% das falhas descobertas. Na segunda experiência, foram detectados 62% das falhas injetadss, subdivididos nos vários tipos de erros. E, 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. Na primeira experiência 1,5% das falhas injetadas ocasionaram falhas em dobro e 1% corrupção de dados. Na segunda experiência, 2% das falhas injetadas ocasionaram falhas em dobro e não foi verificada nenhuma corrupção de dados.

Analisando-se estas falhas é difícil tirar qualquer conclusão, mas estas falhas fatais podem indicar uma propagação de erro ou alguma outra falha de projeto no mecanismo de recuperação.

A eficiência de recuperação é caracterizada pelo período de tempo em que o sistema está reduzido no 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 um 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. Os resultados obtidos nas duas experiências indicam uma falha de projeto no mecanismo de recuperação de erros do SGBD.

Degradação do desempenho reflete o *overhead* provocado pelos mecanismos de tolerância a falhas. Degradação de desempenho é medida pelo ponto de vista do cliente. Cada execução mede o número de 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 variam de 12% a 21%.

3.2.4 Conclusão

Os três experimentos acima citados limitaram-se a injetar falhas cada qual numa área específica. O projeto Delphos concentrou seus estudos nas falhas de *hardware*, o experimento de Sudipto Ghosh nas falhas de interfaces e o experimento utilizando o SGBD ClustRa as falhas de comunicação. Todos eles tiveram como principal objetivo a avaliação da dependabilidade dos SGBDs utilizados. Destes experimentos os que mais contribuíram

para o desenvolvimento desta dissertação foram os que se detiveram nas falhas de *hardware* e comunicação.

4 Ambiente do Experimento

Neste capítulo é descrito o ambiente utilizado para a condução do experimento. Procurou-se utilizar somente características comuns na grande maioria das plataformas comerciais da atualidade, tanto em nível de *hardware* como de *software.*

4.1 Plataforma de Hardware

Como servidores do banco de dados foram utilizados dois computadores de arquitetura Intel Pentium III de 600 MHZ com 256 MB de RAM (figura 4.1). Como estações clientes foram utilizadas 2 estações de trabalho de também de arquitetura Intel com processadores Pentium III de 450 MHZ com 128 MB de RAM. A interligação dos servidores e das estações de trabalho através de uma rede local padrão Ethernet de 100 Mbit com cada equipamento num segmento específico da rede suportado através de um Switch.

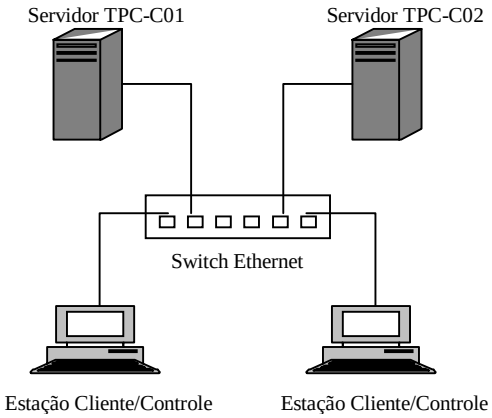


FIGURA 4.1 - *Layout* físico

4.2 Plataforma de *Software*

Nesta seção são descritos os componentes de software utilizados no ambiente do experimento como mostra a figura 4.2.

Utilizaram-se os seguintes componentes de *software*: sistema operacional GNU/Linux para os servidores do 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, estes dois implementados especificamente para esta dissertação.

4.2.1 O Sistema Operacional GNU/Linux

O sistema operacional GNU/Linux foi escolhido como plataforma operacional dos servidores do BDD 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 GLP, o que significa que qualquer pessoa tem o direito de usar, copiar e modificar o sistema livre de pagamento de *royalty*.

Podemos citar como suas principais características:

- *Multi-tasking* – Linux suporta *multi-tasking* preemptiva, todos os processos rodam independentemente entre si;
- Multiusuário;
- Independência de Arquitetura;
- Execução sob Demanda – Somente a parte que está sendo executada de um programa está na memória;
- Paginação – Quando a memória física do sistema é esgotada, o Linux procura por páginas de memória de 4 Kbyte que podem ser liberadas, ou seja, aquelas que já foram gravadas em disco. Neste caso, ele descarta estas páginas da memória liberando-a, todas as outras páginas são gravadas em disco. Se alguma página é requisitada ela é carregada novamente na memória. Diferente do *swapping* utilizado em antigas variações de UNIX, onde toda a memória utilizada por um processo era gravada no disco rígido, trazendo uma perda significativa de desempenho;
- *Cache* de Disco Rígido Dinâmica – O Linux dinamicamente ajusta o tamanho da memória *cache* utilizada pelo disco rígido de acordo com a disponibilidade de memória física;
- Bibliotecas Compartilhas;
- Suporte ao padrão POSIX 1003.1 e em parte o *System V* e o BSD;
- Memória Protegida – O que previne que um processo acesse a memória do *kernel* do sistema ou utilizada por outro processo;
- Suporte a vários formatos de arquivos executáveis e diferentes tipos de sistemas de arquivos.

Foi utilizado no experimento a versão de *kernel* 2.2.16 que era a última versão estável quando do início dos experimentos. A escolha do Linux para ser o sistema operacional dos servidores do SGBD foi motivada pelas ferramentas de injeção de falhas utilizadas, já que estas, obrigatoriamente, necessitam deste sistema operacional.

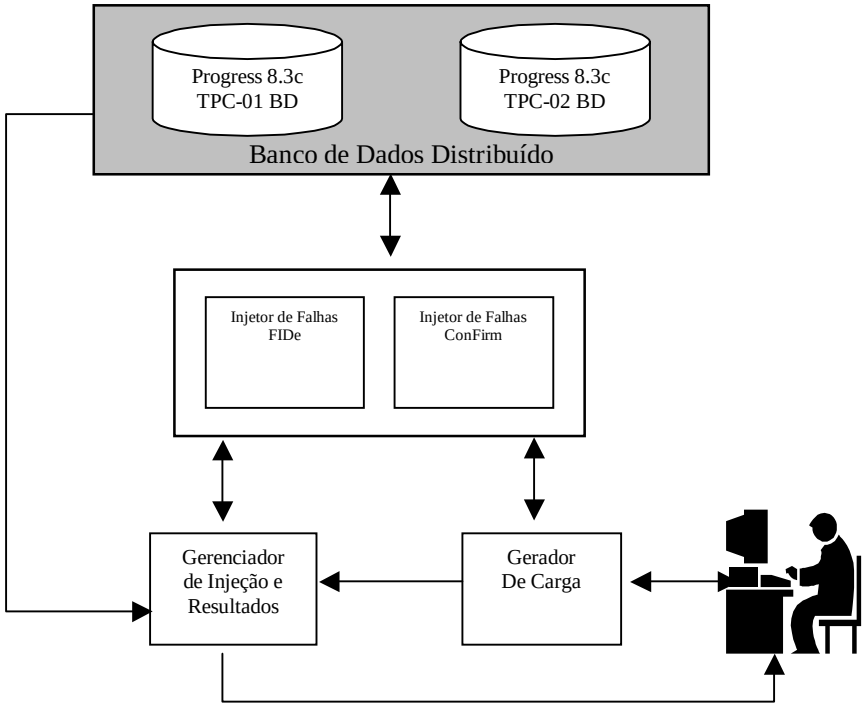


FIGURA 4.2 - *Layout* Lógico

4.2.2 O SGBD PROGRESS

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

A escolha do SGBD Progress deve-se a sua larga utilização como banco de dados embutido (*embedded)* em aplicações COTS. Na arquitetura utilizada os dados são distribuídos entre vários BD que compõem o modelo de dados. Na arquitetura de distribuição usada para a condução do experimento, uma tabela somente poderá estar presente num nodo que compõem o BDD.

O Progress utiliza técnicas de tolerância a falhas que estão presentes na quase totalidade dos SGBDs comerciais como: arquivos de *log* (*logs* de *before image* para os processos de *undo* e *after image* para os processos de *redo*) [BER87], *checkpoint* para atualizar nos arquivos físicos do BD a porção do BD residente em *buffers* de memória e o protocolo *twophase commited* para garantir atomicidade do *commitment*. Alguns destes mecanismos são opcionais, como mostra a figura 4.3, e podem ser ativados e desativados, o que facilita o cálculo do custo computacional destes.

Outra característica, que definiu a utilização do SGBD Progress, foi a existência das tabelas VST (*Virtual System Tables* - Tabela 4.1) [PRO2000]. São tabelas mantidas pelo SGBD, que podem ser acessadas pela aplicação, e que contém informações do funcionamento interno do banco de dados coletadas pelo monitor de transações como o estado do banco e seu desempenho, tais como: número de transações que sofreram

commited, transações que sofreram *undo/redo*, registros lidos, registros alterados, registros excluídos, tempo total que o servidor está ativo, entre outras.

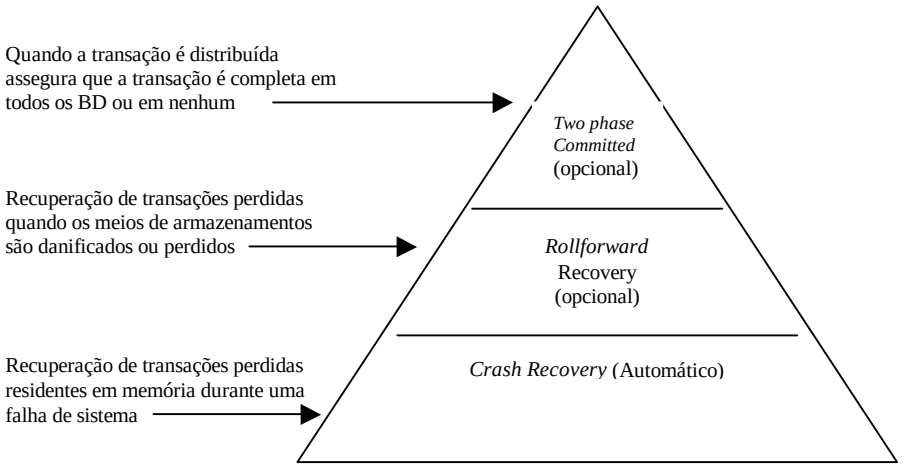


FIGURA 4.3 - Mecanismos de *Recovery* do Progress

As informações sobre o estado do BD, contidas nas VSTs, permitem obter com clareza o momento e a quantidade de vezes em que o BD entrou no processo de recuperação de falhas, e ainda, se estas falhas foram mascaradas ou simplesmente ignoradas pelo BD.

TABELA 4.1 - Exemplo de Tabela VTS _ActOther

Coluna	**Descrição**
_Other-Commit	Número de transações que sofreram *commited.*
_Other-FlushMblk	Número de vezes que o *master block do* BD foi gravado em disco.
_Other-Misc	Outras informações.
_Other-Trans	Número da transação.
_Other-Undo	Número de transações que sofreram *undo*.
_Other-UpTime	Número em segundos que o BD está ativo.
_Other-Wait	Número de vezes que um processo ficou esperando um recurso computacional.

4.2.3 Ferramentas de Injeção de Falhas

Foram utilizadas no experimento duas ferramentas de injeção de falhas para que se atenda os modelos de falhas que este trabalho propõe. Estas ferramentas foram a ComFIRM e o FIDe, já explicadas no capítulo 3.

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

4.2.4 Gerador de Carga e Gerenciador de Injeção e Resultados

O módulo Gerador de Carga (GerPro-TPC) é o responsável pela geração e execuções controladas das transações TPCc, que visam simular um ambiente de produção de um sistema de banco de dados mais realista possível, que possibilite a injeção e a ativação de falhas para posterior comparação do comportamento do SGDB sob funcionamento normal e sob injeção de falhas.

O módulo Gerenciador de Injeção e Resultados (GIR) é o responsável pela interface com o usuário para a geração dos cenários de falhas, ativação das ferramentas de injeção de falhas e leitura e armazenamento dos *logs* das ferramentas e do comportamento do SGBD.

Estes módulos foram implementados utilizando a linguagem 4gl nativa do SGBD Progress. Maiores detalhes destes módulos são descritos nas seções 4.3 e 4.4 respectivamente.

4.2.5 O Sniffer Dsniff

A ferramenta Dsniff [DSN2000] possui uma série de funções para manipulação, interceptação e *dump* de pacotes que trafegam numa LAN/WAN. Esta ferramenta foi utilizada nos experimentos manuais descritos na seção 5.2.

4.3 O Gerador de Carga GerPro-TPC

O GerPro-TPC (**Ger**ador de Carga **Pro**gress especificação **TPC**c), desenvolvido para esta dissertação, segue as especificações e recomendações TPC Benchmark C (TPC-C) [TPC99]. 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).

4.3.1 O Modelo de Dados

O modelo de dados proposto pelo TPC (figura 4.4) 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, os quais são atendidos por outros depósitos.

Inicialmente, antes de qualquer transação TPC ou injeção de falhas o banco de dados é populado com 2.595.055 registros, distribuídos de acordo com a tabela 4.2. A geração destes registros segue regras rígidas descritas na especificação do *benchmark* TPC-C. Detalhes destas especificações podem ser encontras nas normas e recomendações TPC-C [TPC99].

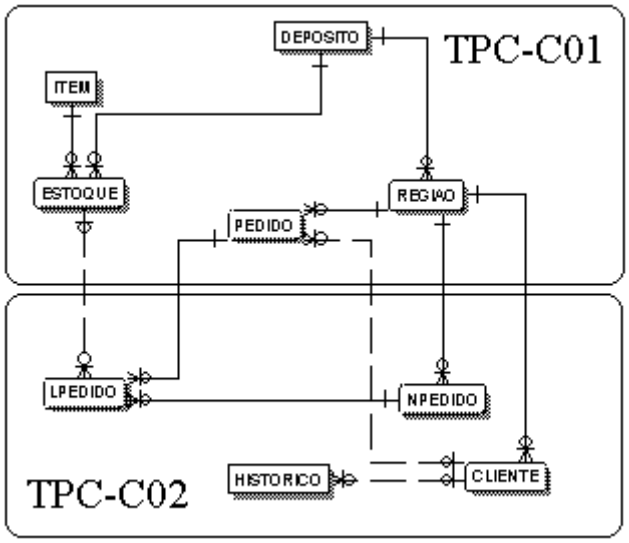


FIGURA 4.4 – Modelo de dados TPC-C

Como o modelo de dados não especifica como deve ser a distribuição dos mesmos no BDD, deixando-a livre para a implementação. A distribuição adotada para estes experimentos, uma tabela está presente em somente um BD que compõe o BDD. As tabelas foram distribuídas da seguinte forma:

- Tabelas do BD TPC-C01: Depósito, Estoque, Item, Pedido e Região.
- Tabelas do BD TPC-C02: Cliente, Histórico, Lpedido e Npedido.

Esta distribuição foi utilizada para que sempre que for executada alguma transação sobre o BDD, esta seja obrigatoriamente distribuída, permitindo tanto a injeção de falhas de comunicação como as que simulam falhas transientes de *hardware.*

TABELA 4.2 – População Inicial do BDD

Tabela	**Registros**	**Observações**
Depósitos	5	
Regiões	50	10 por depósito
Itens	100.000	
Estoque	500.000	100.000 por depósito
Clientes	150.000	3.000 por região
Históricos de Clientes (Histórico)	150.000	1 por cliente
Pedidos	150.000	3000 por região
Itens de Pedidos (Lpedido)	1.500.000	10 por pedido
Novos Pedidos (Npedido)	45.000	Últimos 900 pedidos por região
Total de Registros	2.595.055	

4.3.2 Transações TPC-C sobre o BDD

Foram implementadas todas as transações da especificação TPC-C. As transações foram escritas utilizando-se a linguagem de programação Progress-4gl nativa do SGBD Progress. Inicialmente utilizou-se a linguagem C com SQL/89 embutido e o compilador GCC para o programa que insere os 2.595.955 registros iniciais do BDD. Por questões de desempenho a linguagem C foi abandonada após observar-se que as chamadas SQL eram mais lentas que a execução na linguagem nativa do SGBD. Como o escopo deste trabalho não era o de criar um gerador de cargas portável, mas sim conduzir o experimento de injeção de falhas, foi feito a opção pela linguagem nativa.

Para as transações de carga especificaram-se poucos parâmetros de entrada a serem fornecidos pelo operador do GerPRO-TPCc para permitir sua execução sem qualquer intervenção. Somente é especificado o número de transações a serem executadas. Todas as ações de criar, alterar, excluir e transações que sofreram propositadamente *undo* devem ser mostradas no momento que ocorrem segundo a especificação.

As transações criadas segundo a especificação foram:

1. Transação de Novos Pedidos (figura 4.5) – Consiste na inclusão de um número informado de novos pedidos. Deve ser uma transação única para cada novo pedido, apesar de haver em média doze inclusões e 2 atualizações da base de dados. Internamente, nesta transação e em todas as outras, os dados alfanuméricos são gerados randomicamente. Como mostra a figura, no rodapé da tela são informados o número de transações efetivamente criadas e o tempo total de execução em segundo.

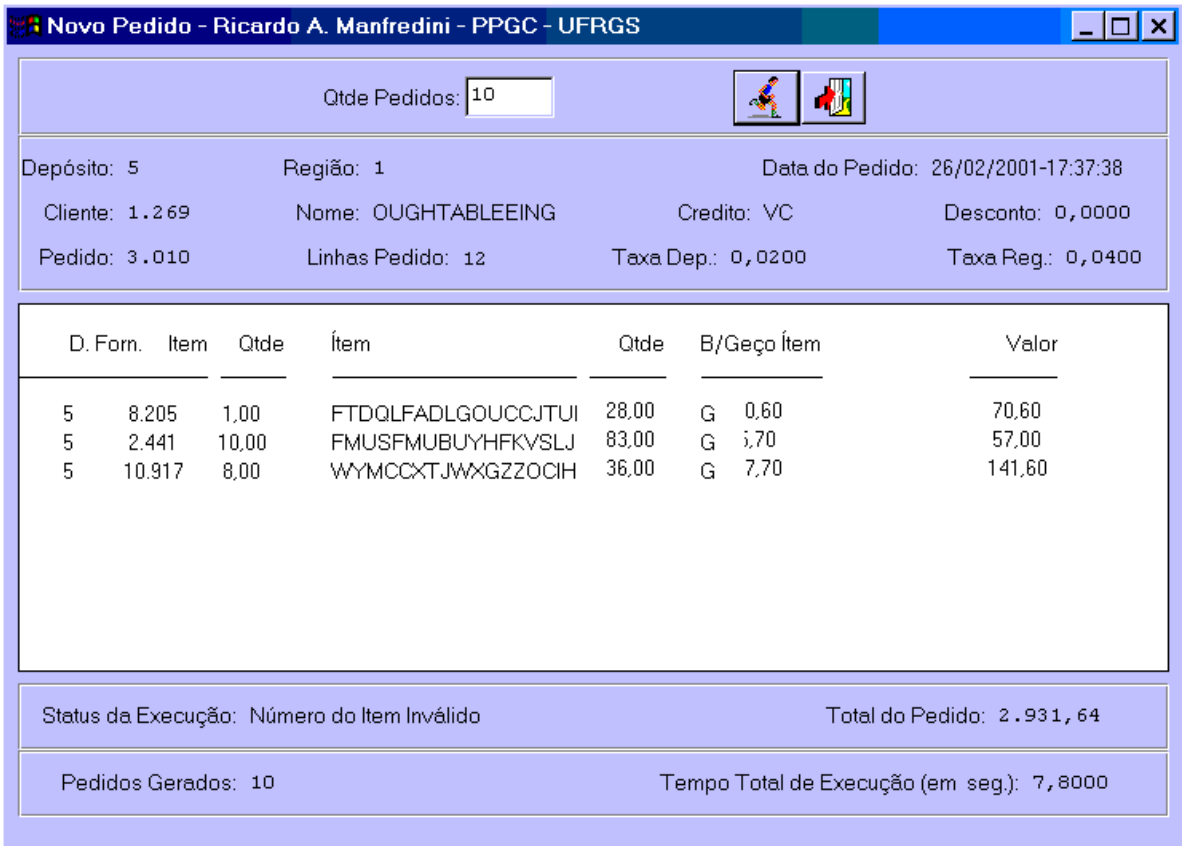


FIGURA 4.5 - Transação de Novos Pedidos

2. Transação de Pagamento (figura 4.6) - Transação que envolve uma inclusão e três alterações na base de dados.

3. Transação de Consulta de Pedidos e de Entrega – Somente consultas na base de dados e não relevantes ao experimento.

4. Transação de Entrega de Produtos – Segundo a especificação esta é uma transação *batch*, onde cada transação é composta de dez entrega de pedidos ainda não entregues. Não é informado o total de entregas realizadas.

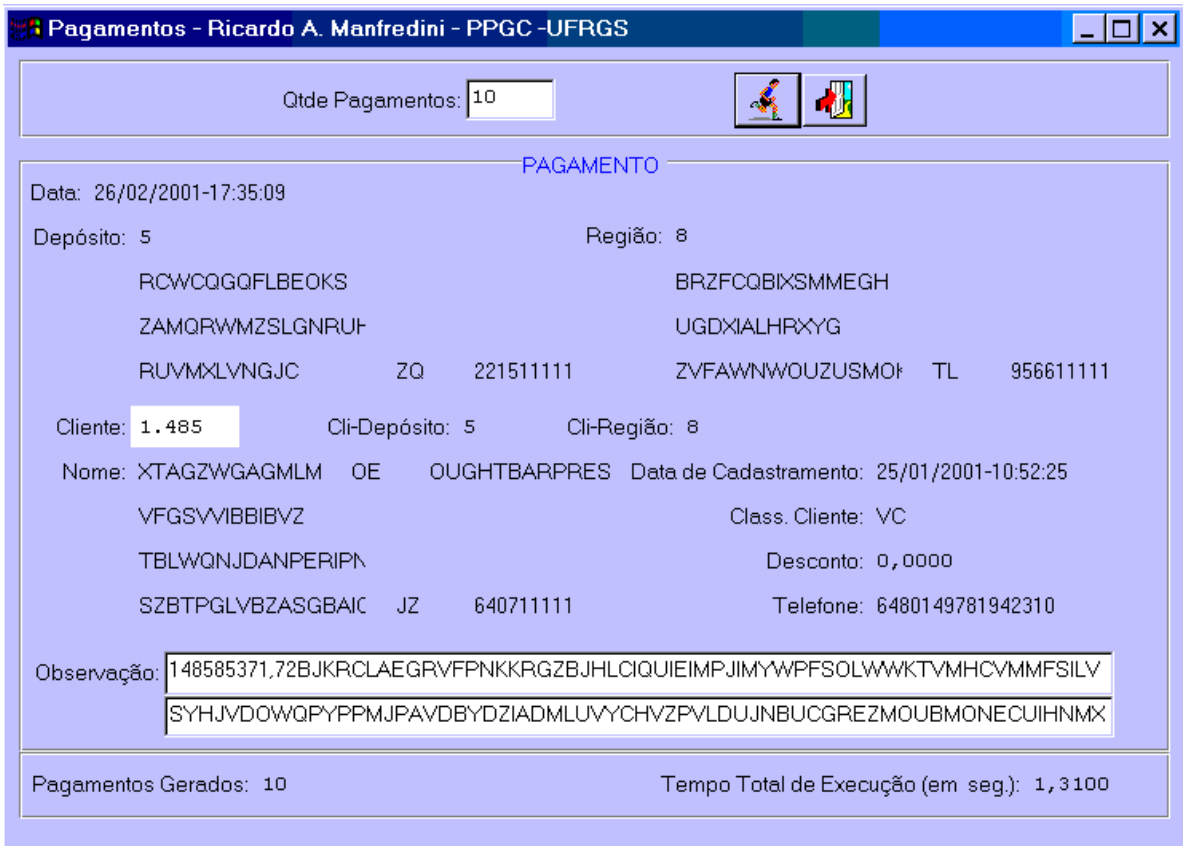


FIGURA 4.6 – Transação de Pagamentos

4.4 O Gerenciador de Injeções e Resultados GIR

A função do GIR é permitir o armazenamento histórico das transações realizadas pelo GerPro-TPC e dos cenários de falhas utilizados, ativar as ferramentas de injeção, coletar os *logs* das ferramentas e verificar comportamento do SGBD sob falhas.

Para sua implementação foi realizada uma pequena alteração no modelo de dados mostrado na figura 4.4. Foram criadas duas novas tabelas (ACTSUM01 e 02) uma para cada BD que compõe o BDD. Estas tabelas contêm o histórico totalizado de toda a atividade do SGBD, com dados extraídos das VST de cada um dos BD. No início de cada uma das transações acima explicadas é criado um registro nestas tabelas e no seu término são totalizadas as atividades do SGBD.

Como mostra a figura 4.7, a linha do *browse* que aparece em negrito representa uma transação de *Pagamento* que na parta inferior da janela é totalizada as atividades de leituras, gravações e exclusões nos BDD e nos seus *logs*, o número de *undos* sofridos pelo BDD, o número de *checkpoints* realizados no intevalo de tempo da transação bem como seu tempo total de execução.

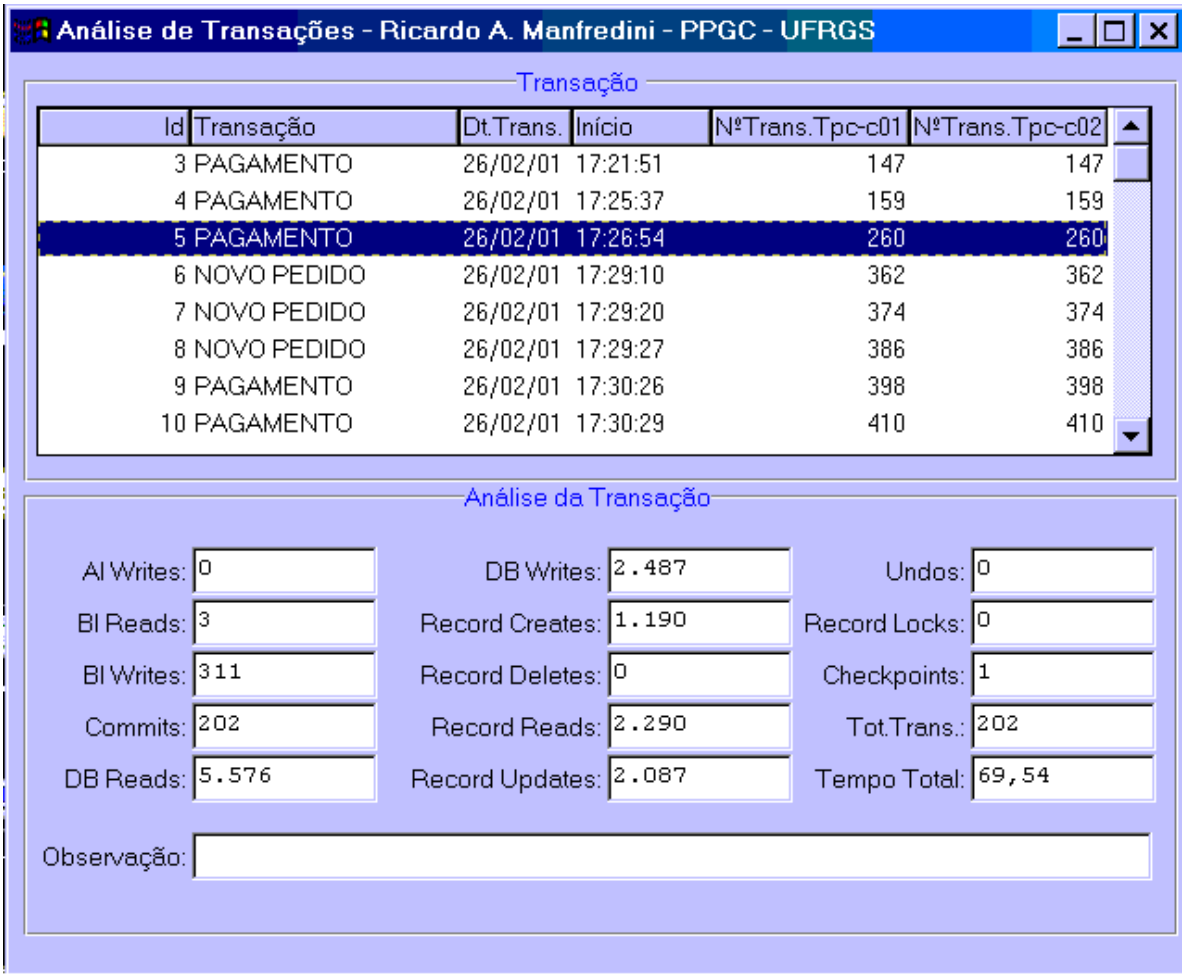


FIGURA 4.7 – Analisador de Transações

A tabela 4.3 mostra de forma mais detalhada os dados relevantes que foram extraídos das VST e inseridos nas tabelas ACTSUM a cada transação TPC-C. Estes dados, posteriormente no experimento, foram utilizados para verificar o resultado das experiências de injeção de falhas e o comportamento do SGBD.

TABELA 4.3 – Tabela ACTSUM

Coluna	**Descrição**
AIWrite	Número de gravações no arquivo de log de *redo* do SGBD
DBWrites	Número de gravações físicas no banco
Undos	Número de processos de *undo* sofridos
BIReads	Número de leituras ao log de *undo*
RecordCreates	Número de registros criados pela transação TPC-C
RecordLocks	Número de registros que permaneceram bloqueados após a transação
BIWrite	Número de gravações no log de *undo*
RecordDeletes	Número de registros excluídos pela transação
Checkpoints	Número de *checkpoints* realizados pelo SGBD durante a transação
Commits	Número de transações confirmadas (*commitment)*
RecordReads	Registros lidos pela transação
TotTrans.	Número total de transações executadas
DBReads	Número de leituras físicas no banco de dados
RecordUpdates	Número de registros atualizados pela transação

4.5 Alterações Efetuadas nas Ferramentas de Injeção

Com a utilização sistemática das ferramentas de injeção de falhas e a necessidade de integrá-las no GIR, foram necessárias pequenas modificações nas mesmas para facilitar a condução do experimento.

Na utilização da ComFIRM, sempre que inicializávamos os servidores e os clientes do BD era necessária a execução do Dsniff para podermos identificar qual a porta TCP utilizada por estes processos. Isto se fez necessário porque o SGBD Progress utiliza um intervalo de portas entre 1020 e 2020 que são escolhidas de forma aleatória. Sendo utilizadas portas diferentes no servidor e no cliente a cada processo.

A ComFIRM possui uma interface gráfica, a ComFIRM´s Face, para a sua utilização e ativação, porém esta interface não totaliza as falhas injetadas e o arquivo de *log* somente registra o último experimento realizado. Para solucionar estas dificuldades, o GIR passou a realizar a interface com a ferramenta, possibilitando:

1. Sua ativação, gravando os controles específicos de ativação, encerramento, reinicialização, remoção de regras de transmissão e recepção no arquivo virtual /proc/net/ComFIRM_Control;

2. Coletar dados da execução da ferramenta do arquivo de *log* /proc/net/ComFIRm_Log;

3. A partir dos cenários de falhas, gravar as regras de injeção da ferramenta diretamente nos arquivos /proc/net/ComFIRM_TX_Rules e /proc/net/ComFIRM_RX_Rules. Estas regras são armazenadas para fins históricos.

Para que o GIR pudesse realizar a interface com a ComFIRM, foi feita a tradução do ComFIRM´s Face escrita em Perl para Progress 4gl.

Já na utilização do FIDe, o próprio código da ferramenta foi alterado para facilitar sua utilização no caso específico dos experimentos relatados no capítulo 5. Estas alterações foram:

1. Os arquivos de *log* (fide.report) e regras (fide.rule) passaram a ter nomes variáveis, definidos pelo Gerenciador de Injeções de Resultados o que possibilitou a execução concorrente de várias instâncias da ferramenta no mesmo nodo da rede;

2. Foi adicionada uma opção para a ferramenta identificar quais, quando e com que freqüência as *syscalls* são chamadas pelo processo que está sendo controlado pela ferramenta. Isto facilitou a definição de qual *syscall* utilizar nos cenários de falhas;

3. As falhas injetadas passaram a ser totalizadas no momento de sua injeção e não mais somente no término da execução da ferramenta, pois se alguma falha injetada provocasse o término anormal do processo sob injeção e da ferramenta esta totalização não seria realizada, perdendo-se a informação;

4. O arquivo de *log* passou a ser formatado no padrão XML para facilitar a sua integração com o GerPro-TPC bem com outros BD.

4.6 Conclusão

Este capítulo descreveu os ambientes de *hardware* e de *software* que foram implementados para o desenvolvimento dos experimentos que são descritos no capítulo 5.

Para o ambiente de *hardware* procurou-se utilizar somente equipamentos padrões, de larga utilização em ambientes corporativos e acadêmicos, que se assemelham muito com a realidade destas instituições.

O ambiente de *software* foi o que maior dificuldades trouxe para a sua construção. Isto se deve principalmente pela necessidade de integração de ferramentas distintas, principalmente a ComFIRM e o FIDe, num ambiente único. Os módulos GerPro-TPC e GIR foram integralmente desenvolvidos em Progress 4gl, contendo aproximadamente 5.000 linhas de código fonte.

5 Os Experimentos de Injeção de Falhas em BDD

Neste capítulo é descrito como foram conduzidos os experimentos de injeção de falhas, o modelo de falhas utilizado e os resultados obtidos.

5.1 Modelos de Falhas

Os sistemas distribuídos possuem um grande número de componentes acarretando dificuldades de elaboração de um modelo de falhas que se aplique a todos os seus componentes. Por esta razão, vários tipos de modelos de falhas são necessários, dependendo do tipo de componente que se deseja estudar [PRA96]. O estudo dos modelos de falhas ajudam a aprimorar o processo de desenvolvimento de software e prevenir a ocorrência destas falhas [GOS97].

Para a definição dos modelos de falhas foram usados resultados obtidos por outros pesquisadores que avaliaram os tipos de falhas mais comuns em banco de dados distribuídos: falhas de *buffers* [COS99], falhas de interfaces [GOS97] e falhas de comunicação [SAB99].

Os modelos de falhas adotados tratam, principalmente, de falhas em sistemas distribuídos como: perda de mensagens, falhas de temporização, omissão de transmissão ou recepção, falhas *bizantinas* (arbitrárias) ou ainda computação incorreta [CRI86].

Em virtude de estarmos tratando de SGBD, também fazem parte dos modelos a corrupção de dados na memória (*buffers*) e corrupção de dados em disco. Um BD somente estará em um estado consistente se considerarmos os três lugares possíveis de residência dos dados (disco, memória e arquivos de *log*). Portanto, qualquer falha num destes três locais pode levar o BD a um estado inconsistente.

Os modelos de falhas definem o conjunto de testes que são criados pelo módulo Gerenciador de Injeções e Resultados e que posteriormente serão os parâmetros de entrada das ferramentas de injeção de falhas. Nos experimentos iniciais, o conjunto de falhas definido pelos modelos de falhas é injetado diretamente sobre o SGBD, sem o auxílio das ferramentas injeção de falhas que compõem o modelo do experimento. Essa injeção de falhas inicial, conduzida manualmente, compreende o cancelamento dos processos servidores e clientes do BD, e ainda geração de tráfego artificial na rede e interceptação de pacotes de mensagens através da utilização da ferramenta Dsniff [DSN2000].

Numa segunda fase, para a implementação dos conjuntos de testes, foram utilizadas as ferramentas ConFIRM [OLI00] e FIDe [GON2001], respectivamente para falhas de comunicação e corrupção de *buffers* de memória e leitura/gravação em disco.

5.2 Experimento com Injeção Manual

Inicialmente foram realizados alguns testes, a fim de visualizarmos o comportamento do BDD sob negação de serviços como queda de *link* de comunicação, indisponibilidade de um servidor ou grande tráfego de dados na rede de comunicação.

Como primeiro experimento executou-se um programa que popula a base de dados, que gera os 2.595.055 (tabela 5.2) registros do BDD, como uma transação única. Este processo na plataforma de *hardware* descrita na seção 4.1 dura em média 3 horas e 50 minutos. Quando este processo estava quase no seu final, já tinha gerado 2.590.000 registros, o processo servidor do banco de dados foi cancelado manualmente. Após inicializamos novamente os processos servidores do BDD, este procedimento demorou aproximadamente 2 horas e 15 minutos. Todos os 2.590.000 registros foram excluídos. Isto nos assegurou que a propriedade ACID fosse mantida pelo SGBD Progress, apesar do longo intervalo de tempo que ele ficou indisponível. Este experimento já nos deu uma pequena amostra da disponibilidade deste SGDB.

Num segundo momento utilizou-se o *sniffer* Dsniff para a geração de tráfego na rede e para a identificação e receptação dos pacotes utilizados na comunicação entre os servidores e os clientes do BDD. Esta técnica somente ocasionou a diminuição do desempenho, na execução das transações TPC-C.

O desempenho foi medido pelas métricas sugeridas pela especificação TPC-C. A métrica usada para reportar o máximo desempenho qualificado (MQTh – *Maximum Qualified Throughput)* é o número de pedidos por minuto. O MQTh é o número total de transações de *Novo Pedido* completas dividido pelo tempo total de execução. O nome da métrica utilizada para expressar o MQTh é tpmC (Transaction Processed per Minute do modelo TPC-C).

5.3 Metodologia de Injeção de Falhas Utilizada

No caso específico deste trabalho, os experimentos de injeção de falhas com as ferramentas anteriormente citadas, seguem os seguinte passos:

1. São inicializados os servidores dos BDs em cada nodo participante. Aguarda-se aproximadamente 60 segundos para que todas as *threads* disparadas pelos processos servidores sejam inicializadas;

2. A seguir são inicializados um ou mais GerPro-TPC, dependendo do experimento a realizar, sendo que utilizou-se sempre somente um gerador por estação cliente;

3. A cada intervalo de aproximadamente um segundo, o módulo Gerenciador de Injeções e Resultados, conforme a figura 4.2, coleta e armazena os dados das VSTs nas tabelas ACTSUM, mostrada na tabela 4.3, de cada um dos BDs;

4. Após terminar a inicialização dos módulos GerPro-TPC e GIR injetam-se falhas de forma aleatória, temporizada e sob demanda[4], dependendo do cenário de falhas que se deseja injetar;

5. Depois de finalizada a injeção de falhas, todos os BDs são paralisados e realizada a análise seus *logs*;

6. Ativa-se novamente os BDs e são executados os doze testes de consistência definidos pela especificação TPC-C para verificar se as falhas injetadas não levaram o banco de dados a um estado não consistente. Assegura-se que sempre antes de iniciarmos a injeção de falhas o banco de dados encontra-se num estado consistente.

Estes seis passos obrigatoriamente devem ser repetidos a cada experimento. Eventualmente dependendo do impacto das falhas sobre o BDD o passo seis pode ser estendido pela necessidade de executarmos as ferramentas de recuperação do SGBD Progress.

5.4 Custo dos Mecanismos de Tolerância a Falhas

Esta seção mostra o custo calculado em tpmC dos mecanismos de tolerância a falhas presentes no SGBD Progress.

Todas as medidas mostradas nesta seção e na seção 5.5 foram tomadas considerando-se 100 transações *Novo Pedido*. Somente foram considerados os grupos de 100 transações que no intervalo de tempo de suas execuções ocorreram dois chekpoints no banco de dados e foram executados propositadamente no mínimo dois *undos*. Foram considerados somente dez experimentos num universo de trinta que apresentaram as condições de *chekpoints* e *undos*. Foram descartados os dez experimento de maior desempenho, sempre medido em tpmC, e também foram desconsiderados os dez que obtiveram menor desempenho.

O critério de desconsiderar os testes de maior e menor desempenho deve-se a uma característica do sistema operacional Linux conhecida como *round robin* [BEC98]. Neste método de *scheduling*, cada processo tem uma fatia de tempo (*timeslice*), isto significa que o primeiro processo na fila tem um tempo pré-definido para ser executado no processador. Quando este tempo acaba, o próximo processo na fila de "prontos para executar", o processo identificado como de maior prioridade entrará em execução. Este *perde-ganha* processador ocasiona uma variação de tempos de execução dos experimentos.

Cada transação envolvia em média: 322 registros gravados no *log* de *redo*, 2717 páginas gravadas nos BDs, 2 *undo* intencionais feitos pela aplicação, 8 registros lidos do *log* de *undo*, 200 *commiteds*, 7431páginas lidas dos BDs, 1234 registros criados, 2732 registros lidos e 2189 registros atualizados.

[4] Essas formas são definidas pelas ferramentas utilizadas

A tabela 5.1 sumariza execução das transações selecionadas mostrando o tempo gasto em segundos para a execução de cada grupo de 100 transações bem como o seu desempenho em tpmC. As colunas dois e três mostram as execuções com os mecanismos de tolerância a falhas presentes no SGBD utilizado desativados. Nas colunas quatro e cinco o tempo como estes mecanismos ativados.

TABELA 5.1 – Custo dos Mecanismos de Tolerância a Falhas

Experimento	**Execução de 100 Transações com Mecanismos de Tolerância a Falhas Desativados no SGBD em Segundos**		**Execução de 100 Transações com Mecanismos de Tolerância a Falhas Ativados no SGBD em Segundos**	
	Tempo em segundos	**TpmC**	**Tempo em segundos**	**tpmC**
1	45,40	132,16	58,84	101,97
2	45,73	131,20	58,90	101,87
3	45,92	130,66	59,07	111,57
4	46,18	129,93	59,16	101,42
5	46,32	129,53	59,33	101,13
6	46,65	126,62	60,17	99,72
7	49,34	121,60	61,09	98,21
8	51,91	115,58	61,62	97,37
9	55,38	108,32	61,69	97,26
10	57,21	104,88	62,32	96,28
Média	49,03	123,25	60,22	99,68

Como é mostrado pela tabela 5.1 os mecanismos de tolerância a falhas presentes no SGBD provocam uma redução de desempenho médio de 23,64%, com uma menor redução de 8,93% e maior de 28,60%.

5.5 Custo das Ferramentas de Injeção de Falhas

Sabe-se que ferramentas de injeção de falhas devem afetar o mínimo possível os sistemas alvos. Nas medidas mostradas nesta seção procurou-se verificar a degradação de desempenho do BDD com as ferramentas de injeção de falhas ativas, mas sem a geração de falhas.

As colunas três e quatro da tabela 5.2 demonstram que a ferramenta ComFIRM acarreta no sistema alvo uma diminuição de 2,66% no seu desempenho e o FIDe reduz o desempenho em 5,44%. Já o uso concomitante das duas ferramentas ocasiona em média uma redução de 8,10%. Como era de se esperar, a ComFIRM, é menos onerosa que o FIDe, visto que ele localiza-se no *kernel* do sistema operacional ao contrário do FIDe que executa os processos em modo *trace*.

TABELA 5.2 – Custo ComFIRM e FIDe

Experimento	Execução de 100 Transações com a ComFIRM ativada sem Injeção		Execução de 100 Transações com o FIDe ativado sem Injeção		Execução de 100 Transações com a ComFIRM e o FIDe ativados sem Injeção	
	Tempo em segundos	tpmC	Tempo em segundos	tpmC	Tempo em segundos	tpmC
1	59,13	101,47	60,20	99,68	62,42	96,12
2	61,14	98,13	61,12	98,17	62,47	96,04
3	61,45	97,64	61,96	98,83	63,16	95,00
4	61,80	97,09	62,38	96,18	64,65	92,25
5	61,82	97,05	63,00	95,24	65,04	92,25
6	61,84	97,02	64,17	93,50	65,22	92,00
7	61,97	96,82	65,09	92,18	65,70	91,32
8	62,00	96,77	65,72	91,30	65,94	91,00
9	63,06	65,15	65,81	91,17	66,90	89,67
10	63,94	93,84	65,84	91,13	69,79	85,97
Média		97,10		94,54		92,21

Apesar de ter sido calculado os custos das duas ferramentas ativadas simultaneamente, nos experimentos descritos na seção 5.6, foram realizados com somente uma ferramenta injetando falha por teste. Isto se deve pela dificuldade de observação dos resultados destas injeções simultâneas, onde seria impossível determinar se o defeito ocasionado foi pelas falhas injetadas pela ComFIRM ou pelo FIDe.

5.6 Execução dos Experimentos

O primeiro passo para um experimento de injeção de falhas é a definição do conjunto de falhas que será aplicado, sempre respeitando o modelo de falhas anteriormente definido. Para isto foi implementado o módulo para dar suporte e documentar todo o processo de injeção de falhas, o GIR. Isto se fez necessário porque as ferramentas de injeção utilizadas somente mantém *logs* dos resultados do último experimento realizado, não mantendo histórico dos cenários de falhas que foi utilizada em cada experimento. Este módulo também analisa os *logs* de cada experimento armazenando o tipo de falha injetada, a quantidade de injeções, as atividades do BDD e os cenários de cada experimento. Também gera os arquivos com os parâmetros de entrada das ferramentas utilizadas. Este módulo é executado concomitantemente com o GerPro-TPC. Desta forma, mesmo durante a execução do experimento, já é possível visualizar os seus resultados parciais.

Como é descrito na seção 5.3, a cada experimento é dado *shutdown* nos BDs e são verificadas as suas integridades através de ferramentas específicas do SGBD. Considerou-se que BD está num estado íntegro na inexistência de: índices e páginas do BD danificadas e transações pendentes. Quando for detectada alguma corrupção do BD, estas informações devem ser incluídas manualmente no GIR. Esta interrupção do BD é necessária para evitar a contaminação entre os efeitos de sucessivos experimentos.

5.6.1 Conjunto de Falhas

Na injeção de falhas de comunicações utilizaram-se os seguintes cenários:

1. A cada 1, 10, 100 pacotes transmitidos um pacote é descartado;
2. A cada 1, 10, 100 pacotes transmitidos um pacote é duplicado;
3. Perda de 20% de todos pacotes transmitidos;
4. Atraso de transmissão de todos os pacotes em 1, 10, 50, 100 e 500 ms;
5. Perda de todos os pacotes transmitidos durante 10, 20, 50, 100, 500 segundos;
6. Duplicação de 10, 20, 50, 100% dos pacotes transmitidos;
7. Alteração de 1 a n *bytes*, após n segundo de 1 a n pacotes transmitidos, sendo n um número randomicamente gerado entre 1 e 100 e, finalmente,
8. Os cenários de 1 a 7 são repetidos para pacotes recebidos.

A transmissão e recepção dos pacotes são vistas no lado servidor do BDD. Todos os experimentos são gerados a partir do módulo GIR.

Os cenários de injeção utilizados na injeção de *hardware* são mostrados na tabela 5.3. A tabela 5.4 mostra todas as *syscalls* chamadas pelo SGBD, bem como sua freqüência média em 100 transações. A tabela 5.3 mostra somente os cenários que provocaram algum distúrbio ou reação do SGBD, pois foram criados cenários para a quase totalidade das *syscalls* chamadas, bem como cenários compostos por várias chamadas e não somente uma chamada como demonstra a tabela.

TABELA 5.3 – Cenário de falhas injetadas em memória e disco

Syscall	Quando	Operação	Ação
3	Chamada	Na 127º chamada	Slip um bit do registrador ebx
3	Chamada	Na 127º chamada	Alterou o registrador edx para 1
4	Chamada	A cada 127 segundos	Alterou o registrador edx para 1
4	Chamada	A cada 127 chamadas	Alterou o registrador edx para 1 e slip um bi no registrador esi
5	Retorno	A cada 1 chamada	Zera a posição de memória 4120400H
5	Chamada/Retorno	1 única vez	Altera todos os registradores
6	Chamada/Retorno	A cada 50 chamadas	Desloca o registrador edx 1 bit para a direita
19	Chamada	A cada 217 chamadas	Slip um bit na posição de memória 4013400H
19	Chamada	As próximas 2000 chamadas	Slip o primeiro bit do registrado edx
36	Chamada/Retorno	As próximas 10 chamadas	Zerou todos os registradores
99	Chamada/Retorno	A cada 1 chamada	Slip um bit do registrador eax
99	Retorno	A cada 1 chamada	Slip um bit do registrador ebx

Nas colunas **Freqüência Média** da tabela 5.4 alguns itens estão marcados com *****. Isto indica que a freqüência destas chamadas se mantém constante, independente do número de transações executadas.

TABELA 5.4 – *Syscalls* chamadas pelo SGBD Progress

Syscall	Nome	Freqüência Média	Syscall	Nome	Freqüência Média
1	Exit	1*	55	Fcntl	40*
3	Read	261400	76	Getrlimit	2*
4	Write	185200	78	Gettimeofday	3600
5	Open	9800	85	Readlink	6*
6	Close	11000	90	Mmap	26*
10	Unlink	4*	91	Nummap	10*
13	Time	32600	99	Statfs	16*
19	Seek	259600	102	Socketcall	4*
20	Getpid	14*	106	Stat	94*
23	Umount	2*	107	Lstat	8*
24	Getuid	8*	108	Fstat	16*
27	Alarm	10	117	Ipc	44*
29	Pause	15	119	Sigreturn	5
33	Access	86*	125	Mprotect	10*
36	Sync	2	136	Personality	2*
37	Kill	4*	168	Poll	4800
45	Brk	17400	174	Sigaction	44*
47	Getgid	4*	183	Getcwd	4*
54	Ioctl	7800			

5.6.2 A Utilização das Ferramentas de Injeção

Como é descrito no capítulo 4 na seção 4.5, para utilização e integração da ComFIRM e do FIDe ao ambiente desenvolvido, foram realizadas algumas modificações nas mesmas. No caso do FIDe foi alterado o fonte da ferramenta, na ComFIRM foi desenvolvida uma nova interface para a ferramenta.

Estas alterações foram reportadas aos desenvolvedores das ferramentas para sua possível incorporação ao código das mesmas.

5.6.3 Resultados dos Experimentos

Foram realizados 361 testes de injeção de falhas de comunicação e de *hardware*, sendo injetadas 43.081 falhas. Os erros gerados por estas falhas foram classificados em quatro tipos. São eles: **erros ignorados/mascarados**, **erros leves**, **erros graves** e **erros catastróficos**.

Erros ignorados/mascarados: foram aqueles para os quais, depois de encerrado o experimento, não foram encontrados registros nos *logs* dos BDs e nem um incremento no número de *undos* foi registrado pelo Gerenciador de Injeções e Resultados e, ainda, a integridade referencial dos dados foi mantida.

Erros leves: foram aqueles para os quais também não foram encontrados registros de sua existência, mas foi identificada uma diminuição do desempenho na execução das transações.

Erros Graves: foram classificados como erros graves aqueles que, de alguma maneira, comprometeram a disponibilidade do BDD como a derrubada ou travamento de clientes ou servidores. Na ocorrência desta categoria de erros não se fez necessária a utilização de ações corretivas sobre o BDD e nem ficou comprometida a sua integridade referencial e nem foram violadas as regras de negócio estabelecidas pelo modelo TPC-C.

Erros Catastróficos: foram aqueles que além de comprometerem a disponibilidade também comprometeram a integridade do BDD. Algumas vezes estes erros ocorreram sem deixar registros nos *logs* e nem no Gerenciador de Injeções e Resultados, ficavam latentes. Sua ocorrência só era identificada pela execução do utilitário *proutil* do SGBD Progress, que verifica a integridade da base de dados, como mostra a figura 5.1.

```
               Sat Mar 17 12:00:46 2001
12:00:46 proutil -C idxcheck session begin for root on /dev/pts/2. (451)
12:00:46 BI File Threshold size (-bithold): 0. (6550)
12:07:28 SYSTEM ERROR: read wrong dbkey at offset 85245952 in file /home/rica/Mestrado/DB/tpc-c01.db
  found 2664104, expected 2663968, retrying. (1152)
12:07:38 Corrupt block detected when reading from database. (4229)
12:07:38 SYSTEM ERROR: wrong dbkey in block. Found 2664104, should be 2663968 (1124)
12:07:38 ** Save file named core for analysis by Progress Software Corporation. (439)
12:07:38 proutil -C idxcheck session end. (334)
```

FIGURA 5.1 – *Log* do SGBD Identificando Um Erro Catastrófico

A figura 5.2 mostra o resultado total dos 361 testes realizados. Nota-se que a grande maioria dos experimentos não provocaram qualquer modificação no comportamento sob o sistema alvo, eles representaram 85,60% ou 309 execuções. Nestes experimentos que foram classificados como erros ignorados/mascarados tomaram-se todas as precauções para se certificar que as falhas injetadas não ficaram latentes.

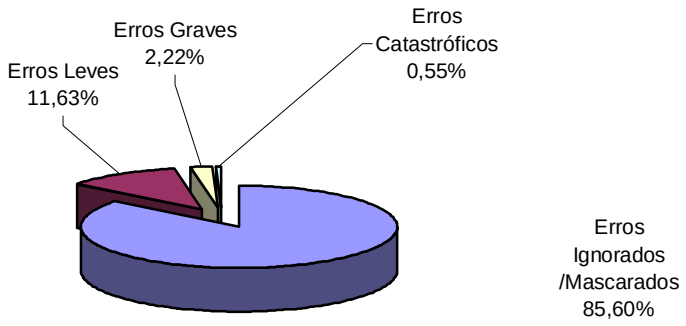


FIGURA 5.2 - Impacto da Injeção de Falhas

Os experimentos que ocasionaram erros leves foram 11,63% ou 42 execuções, nesta classificação encontram-se quase a totalidade das injeções de falhas de comunicação, que ocasionaram somente uma diminuição no desempenho.

Os erros graves foram ocasionados por 2,22% ou 8 experimentos de injeção de falhas. Nesta categoria estão os experimentos que de alguma forma ocasionavam a interrupção dos serviços do BDD e exigiram a intervenção do administrador do BD. Todos os experimentos classificados nesta categoria foram identificados pelo SGBD e deixaram registros nos *logs* como mostra a figura 5.5. Estes experimentos somente comprometeram a disponibilidade do BDD não ocasionando quebras de integridade referencial nem das regras de negócios estabelecidas pelo modelo.

Os erros catastróficos foram provocados por 0,55% ou dois teste. Estes testes ocasionaram a corrupção dos dados e/ou índices do BDD bem como a quebra das regras de negócios. Nestes casos não foi possível tornar o BDD consistente novamente, foi necessário restaurar o *backup* e, a partir dos *logs* de *redo*, também restaurar todas as transações que já tinham sofrido *commited* desde o último processo de *backup*. Este processo pode demorar de alguns minutos a várias horas, dependendo do tamanho do BDD e dos *logs* de *redo*, no caso específico destes testes demorou no primeiro 38 minutos e no segundo 17 minutos.

A figura 5.3 totaliza os 261 testes de injeção de falhas de *hardware*, nestes testes que ocorreram a totalidade dos erros classificados como catastróficos e uma pequena elevação dos erros considerados graves, de 2,22% considerando-se o total dos 361 testes para 2,30% nos testes específicos de injeção de falhas de *hardware*.

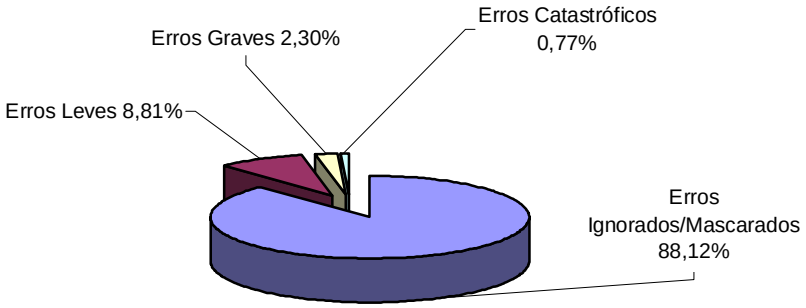


FIGURA 5.3 - Impacto da Injeção de Falhas de Hardware

Já figura 5.4 mostra a totalização dos 100 testes de injeção de falhas de comunicação. Nestes testes não ocorreu nenhum erro classificado como catastrófico. Nota-se também uma diminuição dos erros graves e um grande acréscimo nos erros classificados como leves, ou seja, aqueles nos quais notou-se uma diminuição de desempenho do sistema.

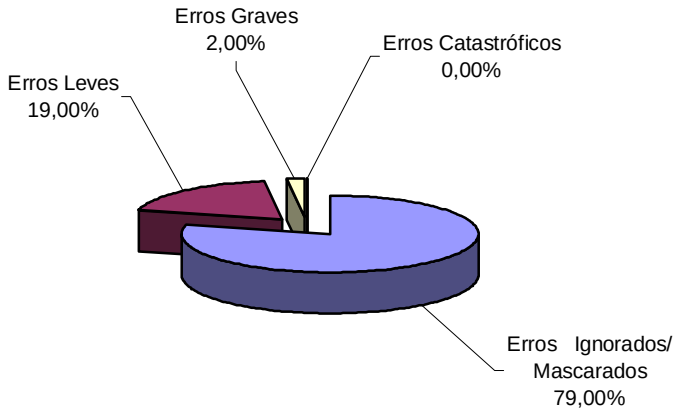


FIGURA 5.4 - Impacto da Injeção de Falhas de Comunicação

```
00:50:34 Usr     4: HANGUP signal received. (562)
00:50:34 Usr     4: Begin transaction backout. (2252)
00:50:34 Usr     4: Transaction backout completed. (2253)
00:50:34 Usr     4: Logout by root on /dev/pts/2. (453)
00:52:43 SRV     1: Login usernum 25, userid root, on DarkSide 2. (742)
00:54:01 SRV     1: Logout usernum 25, userid root, on DarkSide 2. (739)
00:57:06 SRV     1: Login usernum 25, userid root, on DarkSide 3. (742)
00:59:00 SHUT    4: Server shutdown started by root on /dev/pts/2. (542)
00:59:00 BROKER  0: Begin normal shutdown (2248)
00:59:00 SRV     2: Logout usernum 24, userid rica, on benfare-not03. (739)
00:59:00 SRV     1: Logout usernum 25, userid root, on DarkSide 3. (739)
00:59:00 SRV     2: Stopped. (2520)
00:59:00 SRV     1: Stopped. (2520)
00:59:00 Usr     3: Logout by root on /dev/pts/2. (453)
00:59:02 BROKER  0: Multi-user session end. (334)
```

FIGURA 5.5 – *Log* do SGBD Identificando Um Erro Grave

Todos os experimentos que ocasionaram erros graves e catastróficos foram repetidos mais de uma vez, com o mesmo cenário de falhas, e verificou-se que os erros se repetiam da mesma forma. Estas segundas execuções não foram computadas nos 361 experimentos utilizados para classificar os erros gerados.

5.7 Conclusões dos Experimentos

Neste capítulo foi demonstrado todo o processo conduzido para a execução do experimento a que este trabalho se propunha.

Foi definido o modelo de falhas onde o experimento iria tratar de falhas de comunicação e de *hardware*. Inicialmente, foram realizados alguns experimentos manuais, sem a utilização das ferramentas ComFIRM e FIDe, para a verificação do comportamento do BDD sob a negação de alguns serviços, simulando falhas de nodos e quedas de *links* de comunicação.

Após, realizou-se o cálculo do custo médio dos mecanismos de tolerância a falhas embutidos no SGBD Progress, onde constatou-se que estes diminuem o desempenho, medido em tpmC, do BDD em 23,64%. Calculou-se, ainda, qual seria o impacto das ferramentas de injeção de falhas utilizadas sob o sistema alvo, chegando-se a uma redução média de 8,10% no desempenho, com o uso concomitante das duas ferramentas.

Definidos os cenários de falhas que implementariam o modelo, a metodologia para a condução do experimento e as alterações necessárias nas ferramentas de injeção de falhas, passou-se aos experimentos propriamente ditos. Foram considerados para os cálculos estatísticos 361 execuções que nos levaram a concluir que as falhas de comunicação somente afetam a disponibilidade do BDD que podem ser contornadas através de replicações ou redundância da base de dados. Já as falhas e *hardware* transientes ou permanentes além de afetarem a disponibilidade, muitas vezes afetavam a confiabilidade dos dados armazenados do BDD, onde 0.77% das falhas desta categoria injetadas comprometeram o BDD.

6 Conclusão

A técnica de injeção de falhas em sistemas distribuídos, apresentada neste trabalho, constitui-se numa importante ferramenta para a sua validação. A sua aplicação nas fases iniciais do desenvolvimento de sistemas distribuídos possibilita a detecção e correção de erros de projeto e implementação. E ainda, medidas de confiabilidade destes sistemas podem ser obtidas.

Das diversas ferramentas aqui apresentadas, apesar dos resultados favoráveis apresentados por seus experimentos, a grande maioria delas são focadas em sistemas alvos específicos e seus modelos de falhas.

Com a crescente utilização de ferramentas COTS para a implementação de sistemas distribuídos, aumenta o número de componentes destes sistemas e por conseqüência aumenta o seu universo de possíveis falhas. Outro agravante é que normalmente estas ferramentas COTS possuem código proprietário o que inviabiliza a injeção de falha diretamente sobre seus componentes.

Os experimentos relatados nesta dissertação mostraram que a injeção de falhas por software utilizando ferramentas que localizam-se entre o sistema operacional e a aplicação como a ComFIRM, ou ainda, com aquelas que permitem a execução controlada da aplicação como o FIDe, constitui-se um método viável para a validação de banco de dados distribuídos.

Apesar de ter sido utilizado um SGBD proprietário como o Progress, a metodologia adotada, para a condução dos experimentos relatados no capítulo 5, não exigiu a disponibilidade do código fonte do SGBD, o que permite a reprodução destes em outros gerenciadores de bancos de dados.

Os dados resultantes dos experimentos realizados no desenvolvimento desta dissertação, bem como os experimentos realizados por outros pesquisadores mostrados no capítulo 3, permitem concluir que a utilização de SGBDs em sistemas de missão crítica devem ser cuidadosamente avaliadas, já que estes mostram-se muito sensíveis a falhas de *hardware*, que podem ocasionar corrupção de dados comprometendo a sua dependabilidade, e falhas de comunicação que afetam a sua disponibilidade.

As principais dificuldades enfrentadas foram a necessidade de integração de duas ferramentas distintas de injeção de falhas em um ambiente único. Bem como a falta de documentação dos mecanismos de comunicação e gerenciamento de memória/arquivos de dados/*logs* do SGBD utilizado.

Como trabalhos futuros pode-se sugerir a repetição dos experimentos em outro SGBD, mas para isto será necessário portar o gerador de carga GerPRO-TPCc e o gerenciado de injeções e resultados GIR para o SGBD a utilizar. Outra perspectiva para a ampliação destes estudos seria a partir dos resultados das injeções a elaboração de uma base

de conhecimento que pudesse auxiliar e conduzir a elaboração de novos cenários de injeção de falhas.

Bibliografia

[ARL90] ARLAT, Jean et al. Fault Injection for Dependability Validation: A methodology and Some Applications. **IEEE Transactions on Software Engineering**, New York, v.16, n. 2, p. 166-182, Feb. 1990.

[BEC98] BECK, M. et al. **Linux Kernel Internals.** 2nd ed. Harlow: Addison-Wesley, 1998.

[BER87] BERNSTEIN, P.A.; HADZILACOS, V.; GOODMAN, N. **Concurrency Control and Recovery in Database Systems.** Redding, Massachusetts: Addison-Wesley, 1987.

[BER97] BERNSTEIN, P.A. **Principles of Transaction Processing**. Portland: Morgan Kaufmann, 1997.

[CAR95] CARREIRA, João; MADEIRA, Henrique; SILVA, João Gabriel. Xception: Software-Fault Infection and Monitoring in Processor Functional Units. In: WORKING CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APLICATIONS, 1995. **Proceedings…** Urbana-Champaign: IEEE Computer Society Press, 1995.

[CAR98] CARREIRA, João; SILVA, João Gabriel. **Why do Some (weird) People Inject Faults?** Coimbra: Dependable Systems Group, Dep. Of Computer Engineering, University of Coimbra, 1998. Disponível em: <http://dsg.dei.uc.pt>. Acesso em: 13 jan. 2000.

[CAR99] CARREIRA, João; MADEIRA, Henrique; SILVA, João Gabriel. **Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers.** Coimbra: Dependable Systems Group, Dep. Of Computer Engineering, University of Coimbra, 1999. Disponível em: <http://dsg.dei.uc.pt>. Acesso em: 13 jan. 2000.

[COS98] COSTA, Diamantino; MADEIRA, H.; SILVA, J.G. Delphos: Dependability Evaluation of COTS DBMS. In: FTCS, 28., 1998. **Proceedings…** Munich:IEEE Computer Society Press, 1998.

[COS99] COSTA, Diamantino; MADEIRA, H. **Experimental Assessment of COTS DBMS Robustness Under Transient Faults.** Coimbra: Dependable Systems Group, Dep. Of Computer Engineering, University of Coimbra, 1998. Disponível em: <http://dsg.dei.uc.pt> Acesso em: 13 jan. 2000.

[CRI86] CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock Synchronization in the Presence of Omissions and Performance Faults, and Processor Joins. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEM, 16., 1986. **Proceedings…** Viena: Computer Society Press, 1986.

[CZE90] CZECK, E.W.; SIEWIREK, D.P. Effects of Transient Gate-Level Faults on Program Behaviour. In: FTCS, 20., 1990. **Proceedings…** Newcastle:IEEE Computer Society Press,1990.

[DAW96a] DAWSON, Scott et al. **ORCHESTRA:** a Fault Injection Environment for Distributed Systems. [S.l.] Electrical Engineering and Computer Science Department, University of Michigan, 1996. (Technical Report CSE-TR-318-96).

[DAW96b] DAWSON, Scott et al. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT, 1996. **Proceedings…** Sendai: IEEE Computer Society Press. 1996. p. 404-414.

[DSN2000] SONG, D. **DSNIFF -** A Collection of Tools for Network Auditing and Penetration Testing**.** Disponível em: <http://www.monkey.org/~dugsong/dsniff>. Acesso em: 05 jun. 2000.

[GAR99] GARTNER GROUP. **Overview of 1999 DBMS Market Size, Vendor Shares and Forecast**. Disponível em: <http://www.gartner.com>. Acesso em: 23 mar. 2000.

[GON2001] GONÇALVES, Luis Cláudio et al. Testing Fault Tolerance Mechanisms in DBMS Through Fault Injection. In: IEEE LATIN-AMERICAN TESTE WORKSHOP, 2., 2001. **Proceedings…** Cancun: IEEE Computer Society Press, 2001. p. 278-284.

[GOS97] GOSH, Sudipto; HORGAN, J.R. Software Fault Injection Testing on a Distributed System – A Case Study. Software Engineering Research Center, Purdue University, **Proceedings…** Quality Week Europe. Disponível em: <http://www.cs.purdue.edu>. Acesso em: 17 fev. 2000.

[GUE97] GUERRAOUI, R.; SCHIPPER, A. Software-Based for Fault Tolerance**. Computer**, Los Alamitos, v. 30, n. 4, p. 68-74, Apr. 1997.

[GUN89] GUNNEFLO, U.; KARLSSON, J.; TORIN, J. Evaluation of error detection schemes Using Fault Injection by Heavy-Ion Radiation. In: FTCS, 19., 1989. **Proceedings…** Chicago: IEEE Computer Society Press, 1989. p. 340-347.

[JAL94] JALOTE, P. **Fault Tolerance in Distributed Systems.** Englewood Cliffs: Prentice Hall, 1994.

[KRI96] KRISHNAMURTHY, N.; JHAVERI, V.; ABRAHAM, J.A. **A Design Methodology for Software Fault Injection in Embedded Systems.** Austin: The Computer Engineering Research Center, University of Texas at Austin, 1996. Technical Report.

[LAP92] LAPRI, J.C. **Dependability:** Basic Concepts and Terminology, in Dependable Computing and Fault Tolerance. New York: Springer Verlag, 1992.

[LEI2000] LEITE, Fábio Olivé. **ComFIRM – Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional, 2000.** Dissertação (Mestrado) - PPGC da UFRGS, Porto Alegre.

[MAD93] MADEIRA, Henrique et al. Pin-level Fault Injections for dependability Validation: Some Research Results at the University of Coimbra. In: IEEE INT. WORKSHOPING ON FAULT AND ERROR INJECTION FOR DEPENDABILITY VALIDATION OF COMPUTER SYSTEMS, 1993. **Proceedings…** Gothenburg:IEEE Computer Society Press, 1993.

[MAN2001] MANFREDINI, Ricardo A. Avaliação de Disponibilidade e Confiabilidade de Bancos de Dados Distribuídos Através de Injeção de Falhas Por Software. In: SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO TOLERANTE A FALHAS, 9., 2001. **Proceedings...** Florianópolis: SBC, 2001.

[MAR95] MARTINS, Eliane. **ATIFS:** um Ambiente de Testes baseado em Injeção de Falhas por Software. Campinas: UNICAMP, 1995. (Relatório Técnico – 24). Disponível em: <http://www.ic.unicamp>. Acesso em: 17 fev. 2000.

[MIL94] DeMILLO, Richard A.; LI, T.; MATHUR, Aditya P. **Architecture of TAMER:** a Tool for Dependability Analysis of Distributed Fault-Tolerant Systems. Purdue: Software Engineering Research Center, Purdue University, 1994. Disponível em: <http://www.cs.purdue.edu>. Acesso em: 17 fev. 2000.

[ÖZS99] ÖZSU, M.T.; VALDURIEZ, P. **Principles of Distributed Database System**. Englewood Cliffs:Prentice Hall, 1999.

[PRO2000] PROGRESS SOFTWARE CORPORATION. **Progress System Administration Reference.** Boston, 2000.

[PRA96] PRADHAN, Dhiraj. Fault-Tolerant Computer System Design. Upper Siddle River:Prentice Hall, 1996.

[SAB98] SABARATNAM, Maitrayi; TORBJØRNSEM, Ø. Evaluating the Effectiveness of Fault Tolerance in Replicated Database Management Systems. In: ANN, INT'L SYMP. ON FAULT TOLERANT COMPUTING, 29., 1998. **Proceedings…** Madison:IEEE Computer Society Press, 1998.

[SAB99] SABARATNAM, Maitrayi; TORBJØRNSEM, Ø. Cost of Ensuring Safety in Distributed Management Systems. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 1999. **Proceedings...** Hong Kong:IEEE Computer Society Press, 1999.

[SOT97] SOOMA, Irineu. **AFIDS – Arquitetura para Injeção de Falhas em Sistemas Distribuídos, 1997.** Dissertação (Mestrado) - CPGCC da UFRGS, Porto Alegre.

[SUL91] SULLIVAN, M.; CHILLAREGE, R. Software defects and their Impact on System Availability – A study of Field Failures in Operation Systems. In: FTCS, 21., 1991. **Proceedings...** Montreal: IEEE Computer Society Press, 1991. p. 2-9.

[TPC2000] TRANSACTION PROCESSING PERFORMANCE COUNCIL. **TPC BENCHMARK C. Standard Specification Revision 3.5.** Disponível em: <http://www.tpc.org>. Acesso em: 11 out. 2000.

[VOA97] VOAS, Jeffrey. Software Fault Injection: Growing 'Safer' Systems. In: IEEE AEROSPACE CONFERENCE, 1997. **Proceedings...** Snowmass: IEEE Computer Society Press, 1997.