UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AFIDS - Arquitetura para Injeção de Falhas em Sistemas Distribuídos

por

IRINEU SOTOMA

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

Profa. Dra. Taisy Silva Weber
Orientadora

Porto Alegre, julho de 1997.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Sotoma, Irineu
AFIDS - Arquitetura para Injeção de Falhas em Sistemas Distribuídos / por Irineu Sotoma. - Porto Alegre: CPGCC da UFRGS, 1997.

80f.: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, BR-RS, 1997. Orientadora: Weber, T. S.

1. Sistemas Distribuídos. 2. Tolerância a Falhas. 3. Injeção de Falhas. 4. Orientação a Objetos. I. Weber, Taisy Silva. II. Título.

CONFIABILIDADE DE COMPUTADORES
CONFIABILIDADE : COMPUTADORES
TOLERANCIA : FALHAS
SISTEMAS DISTRIBUIDOS
INJEÇÃO : FALHAS
ORIENTAÇÃO : OBJETOS

CNPQ. 1.03.03.00-6

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Wrana Panizzi.
Pró-Reitor de Pós-Graduação: Prof. José Carlos Ferraz Hennemann.
Diretor do Instituto de Informática: Roberto Tom Price.
Coordenador do CPGCC: Prof. Flávio Rech Wagner.
Bibliotecária-Chefe do Instituto de Informática: Zita Prates de Oliveira.

Agradecimentos

Em primeiro lugar, gostaria de agradecer a minha orientadora, profa. dra. Taisy Silva Weber, pelos comentários, pela paciência e pela orientação que foram fundamentais para que eu pudesse concluir este trabalho de mestrado.

No início da pesquisa algumas pessoas foram fundamentais para o esclarecimento de temas como comunicação de grupo confiável (prof. dr. Raimundo Macedo - UFBa) e orientação a objetos (profa. dra. Maria Lúcia Lisboa - UFRGS).

Durante o desenvolvimento da arquitetura proposta, as questões sobre injeção de falhas foram discutidas com a profa. dra. Eliane Martins da Unicamp.

Na fase final da dissertação muitas dúvidas sobre as normas de apresentação da dissertação foram esclarecidas pela bibliotecária Ida Rossi, que possibilitou a conclusão do texto.

Quanto aos outros agradecimentos, eu seria injusto se fizesse uma lista de pessoas, já que, nesta caminhada, várias pessoas me ajudaram. A ajuda foi realizada de várias formas, não somente no aspecto técnico, profissional, mas também no aspecto pessoal. Então eu agradeço a todas àquelas pessoas - amigos, colegas de mestrado, professores, pesquisadores e funcionários, que me apoiaram de alguma forma durante o meu mestrado.

Finalmente, gostaria de agradecer à CAPES pelo apoio financeiro que possibilitou a realização deste trabalho e à UFRGS e à Unicruz pela disponibilidade de equipamentos que permitiu a elaboração do texto e do protótipo de AFIDS-ip.

Sumário

Lista de Figuras

Resumo

Sistemas distribuídos já são de amplo uso atualmente e seu crescimento tende a se acentuar devido à popularização da Internet. Cada vez mais computadores se interligam e trocam informações entre si. Nestes sistemas, requerimentos como confiabilidade, disponibilidade e desempenho são de fundamental importância para a satisfação do usuário. Estes requerimentos podem ser atendidos aproveitando-se da redundância já existente com as máquinas interligadas.

Mas para atingir os requisitos de confiabilidade e disponibilidade, protocolos tolerantes a falhas devem ser construídos. Tolerância a falhas visa continuar a fornecer o serviço de algum protocolo, aplicação ou sistema a despeito da ocorrência de falhas durante a sua execução. Tolerância a falhas pode ser implementada por hardware ou por software através de mascaramento ou recuperação de falhas.

Recentemente, a injeção de falhas implementada por software tem sido um dos principais métodos utilizados para validar protocolos tolerantes a falhas em sistemas distribuídos, e muitas ferramentas têm sido construídas. Contudo, não há nenhuma biblioteca de classes orientada a objetos para auxiliar novos pesquisadores na construção da sua própria ferramenta de injeção de falhas. Este trabalho apresenta um proposta de uma arquitetura orientada a objetos escrita em C++ para sistemas operacionais UNIX usando *sockets*, de modo a alcançar aquele objetivo. Esta arquitetura é chamada de AFIDS (*Architecture for Fault Injection in Distributed Systems*).

AFIDS pretende fornecer uma estrutura básica que aborda as questões principais no processo de injeção de falhas implementada por software: a) a geração de parâmetros de falhas para o experimento, b) o controle da localização, tipo e tempo da injeção de falhas, c) a coleta de dados do experimento, d) a injeção efetiva da falha e e) a análise dos dados coletados de modo a obter medidas de dependabilidade sobre o protocolo tolerante a falhas. Ou seja, AFIDS pretende ser um *framework* para a construção de ferramentas de injeção de falhas. Segundo [BOO 96]: "Através do uso de *frameworks* maduros, o esforço de desenvolvimento torna-se mais fácil, porque os principais elementos funcionais podem ser reutilizados".

AFIDS leva em consideração várias questões de projeto que foram obtidas através da análise de oito ferramentas de injeção de falhas por software para sistemas distribuídos: FIAT [SEG 88], EFA [ECH 92, ECH 94], SFI [ROS 93], DOCTOR [HAN 93], PFI [DAW 94], CSFI [CAR 95a], SockPFI [DAW 95] e ORCHESTRA [DAW 96].

Para auxiliar a construção de AFIDS, é apresentada uma ferramenta de injeção de falhas que utiliza um objeto injetor de falhas por processo do protocolo sob teste. AFIDS e esta ferramenta são implementadas em C++ usando *sockets* em Linux. Para que AFIDS se torne estável e consistente é necessário que outras ferramentas sejam construídas baseadas nela. Isto é enfatizado porque, segundo [BOO 96]: "Um *framework* só começa a alcançar maturidade após a sua aplicação em pelo menos três ou mais aplicações distintas".

Palavras-chaves: Injeção de Falhas, Sistemas Distribuídos, Tolerância a Falhas, Orientação a Objetos.

TITLE: "AFIDS - ARCHITECTURE FOR FAULT INJECTION IN DISTRIBUTED SYSTEMS"

Abstract

Currently, distributed systems are already in wide use. Because of the Internet popularization their growth tend to arise. More and more computers interconnect and share information. In these systems, requirements such as reliability, availability and performance are fundamental in order to satisfy the users. These requirements can be reached taking advantage of the redundancy already associated with the computers interconnected.

However, to reach the reliability and availability requirements, fault tolerant protocols must be built. Fault tolerance aims to provide continuous service of some protocol, application or system in despite of fault occurrence during its execution. Fault tolerance can be implemented in hardware or software using fault masking or recovery.

Recently, the software-implemented fault injection has been one of the main methods used to validate fault tolerant protocols in distributed systems, and many tools has been built. However, there is no object-oriented class library to aid new researchers on the building of own fault injection tool. This work presents a proposal of an object-oriented architecture written in C++ for UNIX operating systems using *sockets*, in order to reach that purpose. This architecture is called AFIDS (Architecture for Fault Injection in Distributed Systems).

AFIDS intends to provide a basic structure that addresses the main issues of the process of software-implemented fault injection: a) the generation of fault parameters for the experiment, b) the control of the location, type and time of the fault injection, c) the data collection of the experiment, d) the effective injection of the faults, and e) the analysis of collected data in order to obtain dependability measures about the fault tolerant protocol sob test. According to [BOO 96]: "By using mature frameworks, the effort of the development team is made even easier, because now major functional elements can be reused.".

AFIDS regards various design issues that were obtained from the analysis of eight tools of software-implemented fault injection for distributed systems: FIAT [SEG 88], EFA [ECH 92, ECH 94], SFI [ROS 93], DOCTOR [HAN 93], PFI [DAW 94], CSFI [CAR 95a], SockPFI [DAW 95] e ORCHESTRA [DAW 96].

In order to aid the AFIDS building, a fault injection tool that uses one injector object in each process of protocol under test is shown. AFIDS and this tool are implemented in C++ using *sockets* on Linux operating system. AFIDS will become stable and consistent after the building of others tools based on it. This is emphasized because, according to [BOO 96]: "A framework does not even begin to reach maturity until it has been applied in at least three or more distinct applications".

Keywords: Fault Injection, Distributed Systems, Fault Tolerance, Object Orientation.

1 Introdução

Em sistemas distribuídos, um dos pontos críticos é tolerância a falhas. Esta característica visa continuar a fornecer um serviço a despeito de quedas de energia, falhas de processador, falhas da rede de comunicação, falhas de software (software com alguma falha de projeto ou implementação) ou até mesmo falhas provocadas por usuários (uso incorreto de alguma aplicação). De modo a implantar realmente tolerância a falhas em um sistema, é necessária a elaboração de protocolos tolerantes a falhas, ou seja que suportem a ocorrência de possíveis falhas durante a sua execução através de mascaramento ou recuperação.

Uma das formas de validar o projeto/implementação de um protocolo tolerante a falhas é através de ferramentas de injeção de falhas. Estas ferramentas injetam falhas durante a execução do protocolo e fornecem dados sobre o comportamento do protocolo na presença destas falhas injetadas. A análise destes dados indica se o projeto/implementação do protocolo atende à especificação, ou seja, se o protocolo realmente mascara ou recupera as falhas a que se propôs.

1.1 Motivação

A injeção de falhas implementada por software tem sido cada vez mais usada para validar protocolos tolerantes a falhas em sistemas distribuídos. Muitas ferramentas de injeção de falhas implementada por software foram construídas para várias plataformas de hardware e software. Entretanto, a maioria destas ferramentas são específicas a um determinado hardware ou software. EFA [ECH 92] executa sobre o sistema operacional distribuído *A/ROSE*. SFI [ROS 93] e DOCTOR [HAN 93] executam sobre o sistema distribuído HARTS. Algumas delas precisam de um sistema operacional adicional para possibilitar a injeção de falhas, por exemplo, PFI [DAW 94] inicialmente usou o sistema operacional *x-kernel* [HUT 91] e DOCTOR [HAN 93] correntemente usa o *x-kernel*.

Mais ainda, não há nenhuma biblioteca orientada a objetos disponível para auxiliar novos desenvolvedores de software a construir a sua própria ferramenta de injeção de falhas implementada por software para sistemas distribuídos. Este trabalho mostra uma proposta de uma arquitetura orientada a objetos chamada AFIDS (Arquitetura para a Injeção de Falhas em Sistemas Distribuídos - *Architecture for Fault Injection in Distributed Systems*), que está sendo escrito em C++ para sistemas operacionais UNIX usando *sockets*. A ênfase é sobre a injeção de falhas de comunicação em sistemas distribuídos.

No grupo de tolerância a falhas da UFRGS alguns trabalhos dentro da área de área de tolerância a falhas em sistemas distribuídos têm sido desenvolvidos. Alguns exemplos dos trabalhos desenvolvidos pelo grupo, e que podem ser validados através de injeção de falhas envolvem replicação de dados, recuperação de processos e comunicação de grupo confiável. ADC [BAR 95] e FIX [TEI 95] enfocaram, respectivamente, difusão confiável e recuperação de processos, e precisaram usar injeção de falhas para validar as suas implementações. Mas as soluções adotadas envolvendo injeção de falhas foram específicas a cada um dos sistemas.

1.2 Objetivos

AFIDS pretende fornecer uma estrutura básica que aborda as questões principais no processo de injeção de falhas implementada por software: a geração de parâmetros de

falhas para o experimento, o controle da localização, tipo e tempo da injeção de falhas, a coleta de dados do experimento, a injeção efetiva da falha e a análise dos dados coletados de modo a obter medidas de dependabilidade sobre o protocolo tolerante a falhas. A arquitetura se concentra na injeção de falhas de comunicação, mas é possível estendê-la para fornecer mecanismos para injetar falhas de memória e processador. A arquitetura suporta as seguintes falhas aplicadas sobre mensagens: omissão de recebimento, omissão de envio, valor, queda de canal de comunicação, queda de processo, tempo e bizantina. Estas falhas são as mesmas suportadas por PFI [DAW 94].

A área de orientação a objetos define um *framework* como um conjunto de classes abstratas reutilizáveis de uma aplicação ou subsistema e a forma que os objetos destas classes colaboram entre si [JOH 96]. AFIDS pretende ser um *framework* escrito em C++ para ser usado em qualquer sistema operacional UNIX. Para mostrar a utilização de AFIDS, é apresentado o projeto e a implementação de uma ferramenta, chamada AFIDS-ip (ferramenta baseada em AFIDS com 1 objeto injetor para cada processo do protocolo sob teste). Esta ferramenta é implementada no sistema operacional *Linux* com a utilização do compilador gcc da *Free Software Foundation*. Inicialmente, esta ferramenta avaliará o protocolo de comunicação de grupo (Newtop) implementado por Macedo [EZH 95].

Futuramente, AFIDS-ip será utilizado para validar outros protocolos de tolerância a falhas. E para refinar ainda mais AFIDS, uma outra ferramenta será implementada, chamada AFIDS-in (ferramenta baseada em AFIDS com 1 objeto injetor para cada nodo do sistema). AFIDS-in diminuirá o *overhead* da ferramenta sobre o sistema a ser testado, visto que utilizará apenas um objeto injetor em cada nodo, ao invés de um objeto injetor por processo.

1.3 Organização do Texto

Esta dissertação inicia com uma introdução à injeção de falhas (nomenclatura, objetivos e classificação) e à orientação a objetos (nomenclatura e apresentação do modelo de BOOCH [BOO 94]). O capítulo 3 mostra uma análise das ferramentas de injeção de falhas existentes para sistemas distribuídos e discute algumas diretrizes que devem ser seguidas no projeto de novas ferramentas de injeção de falhas que envolvem falhas de comunicação em sistemas distribuídos. O capítulo 4 apresenta AFIDS (motivação, objetivos, sistema destino e componentes). O capítulo 5 ilustra a utilização de AFIDS através de AFIDS-ip (componentes, cenário de execução, detecção de terminação da execução do protocolo tolerante a falhas e implementação das falhas de comunicação). O capítulo 6 apresenta um exemplo de utilização da ferramenta. O capítulo 7 apresenta as conclusões e perspectivas futuras. O anexo 1 descreve os formatos das estruturas de dados que armazenam os dados relacionados aos arquivos utilizados por AFIDS. O anexo 2 apresenta em C++ os protocolos das classes de AFIDS. O anexo 3 especifica a ferramenta AFIDS-ip usando o método de Booch [BOO 94]. Finalmente o anexo 4 apresenta o formato dos pacotes das mensagens trocadas entre os componentes de AFIDS-ip.

2 Conceitos Básicos

Neste capítulo são apresentados sucintamente os conceitos básicos da injeção de falhas na seção 2.1 e da orientação a objetos segundo Booch [BOO94] na seção 2.2. A leitura do conteúdo das seções 2.1 e 2.2 é necessária apenas aos leitores não familiarizados com os assuntos abordados.

2.1 Injeção de Falhas

Esta seção introduz a injeção de falhas. A injeção de falhas é um conjunto de técnicas utilizadas para acelerar a ocorrência de falhas, erros e defeitos em um sistema, que vem sendo utilizada para validar mecanismos de tolerância a falhas através da introdução controlada das falhas no sistema [MAR 93].

2.1.1 Definições básicas relacionadas com injeção de falhas

Cobertura: É um fator importante usado para quantificar a eficiência dos mecanismos de tolerância a falhas. Geralmente é definido em termos probabilísticos como [MAR 93a]:

c = Pr{serviço correto continue a ser fornecido / ativação de uma falha}

Dormência: Intervalo de tempo entre a ocorrência de uma falha e sua ativação com um erro, ou seja, dormência de falha [ARL 90].

Latência: Intervalo de tempo entre um erro e sua primeira percepção pelos mecanismos de tolerância a falhas, ou seja, latência da detecção do erro [ARL 90].

Sistema destino: É o sistema onde as falhas serão injetadas [ARL 90].

2.1.2 Conjuntos FARM

O processo de injeção de falhas pode ser explicado através dos conjuntos FARM [ARL 90], ilustrado na figura 2.1.

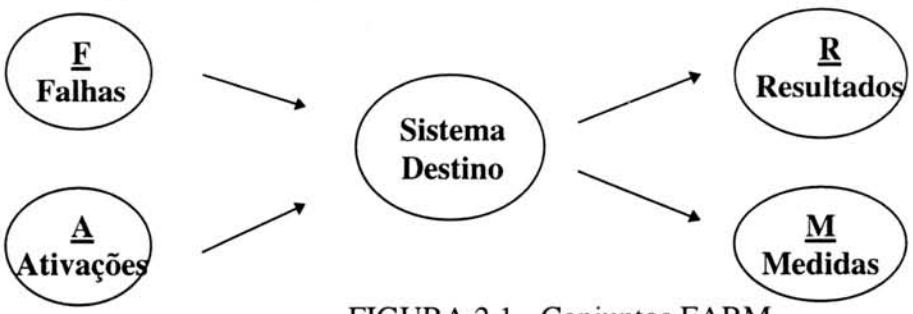


FIGURA 2.1 - Conjuntos FARM

Os conjuntos de entrada no sistema são:

- Conjunto **F:** representa o conjunto de falhas que irão ser injetadas no sistema.
- Conjunto **A:** representa o conjunto de ações que irão ativar as falhas do conjunto F.

Os conjuntos de saída no sistema são:

- Conjunto **R:** representa o conjunto de resultados do processo de injeção de falhas.
- Conjunto **M:** representa o conjunto de medidas de dependabilidade que podem ser obtidas através do uso do conjunto R.

2.1.3 Objetivos da injeção de falhas

A injeção de falhas é composta por duas metas principais complementares [ARL 90]: validação e auxílio ao projeto.

Na validação, a função da injeção de falhas está relacionada ao conceito de cobertura e pode ser vista como um meio de testar os métodos e mecanismos de tolerância a falhas através da introdução de falhas. Na prática, as duas questões principais são:

- *a validação dos procedimentos de verificação* (por exemplo, conjuntos de teste) usados para revelar falhas durante todas as fases do processo de desenvolvimento;
- *a validação dos mecanismos de tolerância a falhas* (por exemplo, detecção, recuperação, etc) visando alcançar a dependabilidade do sistema na fase operacional.

A injeção de falhas envolve ainda dois aspectos da validação ([KAR 94], [ARL 90]):

- previsão de falhas: Serve para avaliar a eficiência de mecanismos de manipulação de falhas para estimar medidas tais como cobertura e latência da detecção de erros.
- eliminação de falhas: É o processo de identificação e eliminação de falhas de projeto em mecanismos de tolerância a falhas.

O auxílio ao projeto ocorre à medida que a injeção de falhas é aplicada no projeto, ajudando a melhorar os mecanismos de teste e o próprio protocolo tolerante a falhas [ARL 90].

2.1.4 Classificação das técnicas de Injeção de Falhas

As técnicas de injeção de falhas podem ser classificadas em:

a) Injeção de Falhas Aplicadas a Modelos de Simulação

Em sistemas simulados, injeção de falhas é usada em vários níveis de abstração, como circuito, porta, RT (*register-transfer*), ou processos (alto nível) [KAR 94, CLA 95]. Uma vantagem desta técnica é poder ser aplicada durante o desenvolvimento do sistema, o que facilita a detecção precoce de falhas de projeto. Outra vantagem é permitir alta controlabilidade e observalidade, através da seleção livre tanto do tempo e localização para a injeção de falhas, quanto dos pontos onde serão observadas as respostas do sistema. A principal desvantagem é o alto custo associado a simulação, o que coloca limitações práticas sobre a quantidade de hardware e software que pode ser modelada. Outra desvantagem está associada à transposição dos resultados da avaliação para a prática. Sistemas reais geralmente não se comportam sob falhas de forma idêntica a seus modelos. Uma ferramenta que implementa este tipo de injeção é FOCUS [CHO 92]. Muitos ambientes de simulação são validados por injetores específicos integrados ao código. No ADC [BAR 95], o ambiente de avaliação de protocolos de difusão confiável é um ambiente simulado e o injetor de falhas faz parte do ambiente.

b) Injeção de Falhas em Sistemas Físicos

Devido à complexidade envolvida, não é possível validar um sistema apenas pela injeção de falhas baseada em simulação. Por isso deve-se fornecer meios de injetar falhas em sistemas físicos. As técnicas de injeção em sistemas físicos podem ser subdivididas em [KAR 94]:

- a nível de pinos: falhas são injetadas fisicamente através de pinças de injeção (forçagem) ou posicionando o circuito sobre soquetes conectados a um injetor de falhas (inserção) [ARL 90, MAR 93a].
- através de distúrbios externos: falhas são injetadas através da exposição do circuito a íons pesados ou de distúrbios na voltagem de fontes de alimentação.
- *built-in*: mecanismos de injeção de falhas são incorporados a circuitos integrados. A injeção de falhas integra-se a técnicas de *scan chain* usadas em teste de circuitos.
- por software: falhas injetadas em um sistema modificam o estado do hardware/software do sistema sob controle do software, levando o sistema a agir como se falhas de hardware estivessem presentes [KAN95]. Basicamente consiste em interromper a execução da aplicação de alguma forma (geralmente inserindo um *trap* de software ou executando em modo *trace*) e executar o código do injetor de falhas. Esse código emula falhas de hardware pela inserção de erros em diferentes partes do sistema como registradores, memória, ou código da aplicação [CAR 95a]. Exemplos dessa classe de ferramentas são FIAT [SEG 88], EFA [ECH 92], SFI [ROS 93], DOCTOR [HAN 93], FINE [KAO 93], PFI [DAW 94], FERRARI [KAN 95], Xception [CAR 95], ProFI [LOV 93].

As técnicas de injeção de falhas a nível de pinos e através de distúrbios externos tem a vantagem de injetar falhas de hardware reais. Mas podem danificar o componente sob teste [KAN 95, LOV 93], requerem o uso de hardware especial e as ferramentas de injeção de falhas são dedicadas a determinado sistema, e há dificuldade em controlar e observar as falhas dentro do processador [CAR 95a].

A injeção de falhas implementada por software tem problemas em modelar alguns tipos de falhas, como aquelas afetando a seção de controle do processador. A execução do injetor de falhas afeta as características de temporização do sistema, prejudicando o teste de funções de tempo críticas [KAR 94]. ORCHESTRA [DAW 96] aproveita características do *Real-Time Mach* para minimizar o *overhead* da injeção de falhas no teste de protocolos tolerantes a falhas em sistemas de tempo real.

A injeção de falhas implementada por software tem como vantagens [CAR 95a]: baixo custo (não requer hardware dedicado), baixa complexidade e esforço de desenvolvimento, portabilidade aumentada, fácil expansibilidade para novos tipos de falhas, nenhum problema com interferências externas e nenhum risco de danificar o sistema destino (sistema sob teste).

A arquitetura proposta AFIDS (Arquitetura para Injeção de Falhas em Sistemas Distribuídos) fornece uma estrutura orientada a objetos para desenvolver ferramentas de injeção de falhas por software em sistemas distribuídos. AFIDS pretende explorar todas as vantagens da injeção de falhas implementada por software, principalmente a portabilidade (através de uma arquitetura bem estruturada) e diminuição do esforço de desenvolvimento (através da reutilização de classes genéricas).

2.1.5 Injeção de Falhas e outras Técnicas de Validação

A injeção de falhas pode complementar outras técnicas de validação [SEG 88, ARL 90, MAR 93a] como modelagem analítica e *error logging*. Modelagem analítica é extremamente difícil pois precisa de informação retirada da análise experimental [KAO 93] e a simplificação de assertivas, feitas de modo a fazer a análise tratável, reduz a utilidade dos resultados [KAN 95]. *Error logging* armazena informações sobre a ocorrência de erros no sistema em funcionamento, mas não pode ser empregada em sistemas que requerem extrema confiabilidade, onde teste após o uso do sistema é inapropriado [SEG 88, KAN 95].

Barton [BAR 90] mostra, através dos resultados de experimentos usando FIAT, que há um número limitado de manifestações de falhas nos níveis de abstração superiores ao nível que as falhas reais ocorrem. Isto possibilita que ferramentas de injeção de falhas possam realizar emulações de falhas a níveis de abstração mais altos que os das falhas reais.

2.2 Orientação a Objetos

Esta seção apresenta uma introdução aos conceitos de orientação a objetos segundo BOOCH [BOO 94]. Fowler [FOW 96] salienta que o método de BOOCH é muito baseado em implementação e as suas construções tem relacionamentos diretos com C++, o que justifica o seu uso neste trabalho de mestrado, visto que a linguagem utilizada é C++.

Sistemas complexos devem ser manipulados através da decomposição do problema em hierarquias de abstração, de modo a simplificar a tarefa de desenvolvimento de software. A orientação a objetos é uma forma de visualizar o problema como uma coleção de objetos que se relacionam entre si para fornecer um comportamento para algum nível mais alto de abstração.

O modelo de objetos é composto de quatro elementos principais: abstração, encapsulamento, modularidade e hierarquia:

- A abstração denota as características essenciais de um objeto que o distingue dos demais.
- A encapsulamento serve para separar a interface de uma abstração e a sua implementação.
- A modularidade visa decompor um sistema em um conjunto de módulos coesos e fracamente acoplados.
- A hierarquia é uma ordenação de abstrações.

Um objeto possui estado, comportamento e identidade e corresponde a uma instância de alguma classe. Uma classe corresponde a um conjunto de objetos que compartilham uma estrutura e comportamento comuns. Cada classe possui um conjunto de atributos (variáveis) e um conjunto de métodos (funções membros). Cada objeto pode ser acessado através de uma mensagem (chamada a algum método).

O modelo de objetos proposto por BOOCH possui os seguintes diagramas:

- Diagrama de Classes: mostra a existência de classes e seus relacionamentos na visão lógica de um sistema.
- Diagrama de Transição de Estados: mostra o espaço de estados de um dado contexto, os eventos que provocam uma transição de um estado para um outro, e as ações que resultam.
- Diagrama de Objetos: mostra a existência de objetos e seus relacionamentos na visão lógica de um sistema.
- Diagrama de Interação: traça a execução de um cenário.
- Diagrama de Módulo: mostra a alocação de classes e objetos na visão física do sistema.
- Diagrama de Processo: mostra a alocação de processos a processadores na visão física do sistema.

O capítulo 4 e o anexo 3 ilustram os diagramas de BOOCH através da modelagem de AFIDS e de AFIDS-ip e descrevem em mais detalhes a notação utilizada nesses diagramas.

3 Ferramentas de Injeção de Falhas para Sistemas Distribuídos

Este capítulo apresenta as ferramentas existentes para injeção de falhas implementada por software em sistemas distribuídos. A partir da análise destas ferramentas, características que foram e devem ser consideradas no projeto de novas ferramentas são discutidas. Estas ferramentas foram escolhidas devido ao enfoque desta dissertação, que é a injeção de falhas de comunicação em sistemas distribuídos. Alguns exemplos de ferramentas que abordam apenas falhas de memória e CPU são Xception [CAR 95], ProFI [LOV 93], FERRARI [KAN 95] e FINE [KAO 93].

3.1 Ferramentas Analisadas

Esta seção apresenta o local de desenvolvimento, o sistema destino, o modelo de falhas, a arquitetura e os experimentos realizados em oito ferramentas analisadas: FIAT [SEG 88], EFA [ECH 92], SFI [ROS 93], DOCTOR [HAN 93], PFI [DAW 94], CSFI [CAR 95a], SockPFI [DAW 95] e ORCHESTRA [DAW 96]. Destas ferramentas, FIAT, CSFI e PFI foram as que mais influenciaram o desenvolvimento da arquitetura AFIDS. Mas a princípio todas as ferramentas analisadas poderiam ser modeladas usando AFIDS.

Para os leitores que não estiverem interessados nas particularidades das ferramentas analisadas, sugere-se a leitura direta da seção 3.2, que discute as questões de projeto de novas ferramentas.

3.1.1 Local de Desenvolvimento

O local de desenvolvimento cita a universidade e o país onde cada ferramenta foi desenvolvida.

3.1.1.1 Local de Desenvolvimento de FIAT

FIAT (*Fault Injection-based Automated Testing*) foi desenvolvido pela *Carnegie Mellon University* nos Estados Unidos.

3.1.1.2 Local de Desenvolvimento de EFA

EFA (*Experimental environment for Fault-tolerant Algorithms*) foi desenvolvido pela *Universität Dortmund* na Alemanha.

3.1.1.3 Local de Desenvolvimento de SFI, DOCTOR, PFI, SockPFI e ORCHESTRA

SFI (*Software Fault Injector*), DOCTOR (*An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-Time Systems*), PFI (*Probe/Fault Injection tool*), SockPFI e ORCHESTRA foram desenvolvidos pela *University of Michigan* nos Estados Unidos.

3.1.1.4 Local de Desenvolvimento de CSFI

CSFI (*Communication Software Fault Injector*) foi desenvolvido pela Universidade de Coimbra em Portugal.

3.1.2 Sistema Destino

O sistema destino determina em qual plataforma de hardware e software cada ferramenta foi desenvolvida.

3.1.2.1 Sistema Destino de FIAT

FIAT visa validar sistemas distribuídos de tempo real através da descoberta de deficiências nos mecanismos de detecção e recuperação de erros do sistema, e avaliar quantitativamente a dependabilidade do sistema sob teste. Foi implementado com 4 PC's IBM RT conectados via *token ring*.

3.1.2.2 Sistema Destino de EFA

EFA visa o teste de algoritmos tolerantes a falhas em sistemas distribuídos. EFA possui nodos com 1 a 3 processadores 68000 rodando o S.O. de tempo real *A/ROSE*. Os nodos são conectados por um conjunto de *links* seriais bidirecionais, que podem ser configurados para qualquer topologia até nodos de grau 12.

3.1.2.3 Sistema Destino de SFI

SFI visa validar a dependabilidade do sistema distribuído de tempo real HARTS. Foi implementado no sistema distribuído HARTS que possui nodos multiprocessadores conectados por uma rede de interconexão ponto a ponto. Cada nodo possui processadores de aplicação e processadores de rede (responsáveis pela comunicação), que rodam o sistema operacional HARTOS.

3.1.2.4 Sistema Destino de DOCTOR

DOCTOR visa validar sistemas distribuídos de tempo real se preocupando em minimizar o *overhead* da injeção de falhas e da coleta de dados sobre o sistema destino. Foi implementado no sistema distribuído HARTS.

3.1.2.5 Sistema Destino de PFI

PFI visa testar sistemas distribuídos de forma determinística ao invés da forma randômica. PFI é uma abreviatura para *Probe/Fault Injection tool*. Além disso, foi projetada para o teste de sistemas onde o código fonte do protocolo tolerante a falhas não está disponível. Para tanto foi desenvolvida uma técnica de injeção chamada *script-driven probing and fault injection*. Através desta técnica procura-se alcançar três objetivos: detecção de erros de projeto/implementação de protocolos tolerantes a falhas, identificação de violações de especificações de protocolo e *insight* sobre decisões feitas pelos implementadores. Implementado inicialmente com o *x-kernel* rodando sobre *Mach 3.0* e posteriormente portado para *SunOS*.

3.1.2.6 Sistema Destino de SockPFI e ORCHESTRA

SockPFI e ORCHESTA visam o teste de algoritmos tolerantes a falhas e avaliação de características temporais de aplicações de tempo real distribuídas. Foram implementados sobre o sistema operacional *Real-Time Mach*. ORCHESTRA foi portado para o *SunOS* e *Solaris*.

3.1.2.7 Sistema Destino de CSFI

CSFI visa validar sistemas paralelos/distribuídos. Foi implementado em um sistema baseado em *transputers* T805 executando o sistema operacional *PARIX 1.2*.

3.1.3 Modelo de Falhas

O modelo de falhas determina quais os locais onde as falhas podem ser injetadas com a utilização da ferramenta.

3.1.3.1 Modelo de Falhas de FIAT

FIAT permite injetar falhas de memória, registrador e comunicação.

3.1.3.2 Modelo de Falhas de EFA

EFA permite injetar falhas de comunicação (omissão, temporização, valor e bizantina).

3.1.3.3 Modelo de Falhas de SFI

SFI permite injetar falhas de memória, comunicação (mensagens perdidas, alteradas e atrasadas) e processador.

3.1.3.4 Modelo de Falhas de DOCTOR

DOCTOR permite injetar falhas de processador, memória e comunicação (mensagens perdidas, duplicadas, alteradas e atrasadas).

3.1.3.5 Modelo de Falhas de PFI, SockPFI e ORCHESTRA

PFI, SockPFI e ORCHESTA permitem emular quedas de *links* e de processos e a injeção de falhas de comunicação (omissão de envio, omissão de recebimento, temporização, bizantinos).

3.1.3.6 Modelo de Falhas de CSFI

CSFI permite injetar falhas de comunicação (mensagens duplicadas, alteradas).

3.1.4 Arquitetura

Esta subseção apresenta a arquitetura das ferramentas analisadas.

3.1.4.1 Arquitetura de FIAT

FIAT segue as seguintes fases experimentais:

- Validação do sistema livre de falhas: Traçar as características de desempenho dos componentes de software do sistema e coletar dados.
- Validação do *workload* livre de falhas: Traçar as características de desempenho/funcionalidade do *workload* e coletar dados.
- Experimentação de injeção de falhas: Injetar falhas no *workload* analisado e coletar histórias e registros de erros.

FIAT é um ambiente integrado que utiliza as seguintes abstrações usadas no processo de injeção: *workload*, classes de falhas, experimento e coleta/análise de dados.

O *workload* é um conjunto monitorável de tarefas de tempo real que se comunicam. Para injetar falhas em uma tarefa sem quebrar o mecanismo de proteção do sistema operacional, à tarefa são ligados quatro programas anexos, que servem como um Cavalo de Tróia para injetar falhas de dentro da tarefa após uma requisição externa.

Os programas anexos são:

- Monitor de *Workload*: Fornece a monitorização da execução.
- Injetor de Falhas: Faz a injeção real.
- Relatório e Detecção de erros: Coleta os dados.
- Arquitetura Tolerante a Falhas: É um programa genérico para a implementação de mecanismos de recuperação baseados em software.

A geração de tarefas é mostrada na figura 3.1.

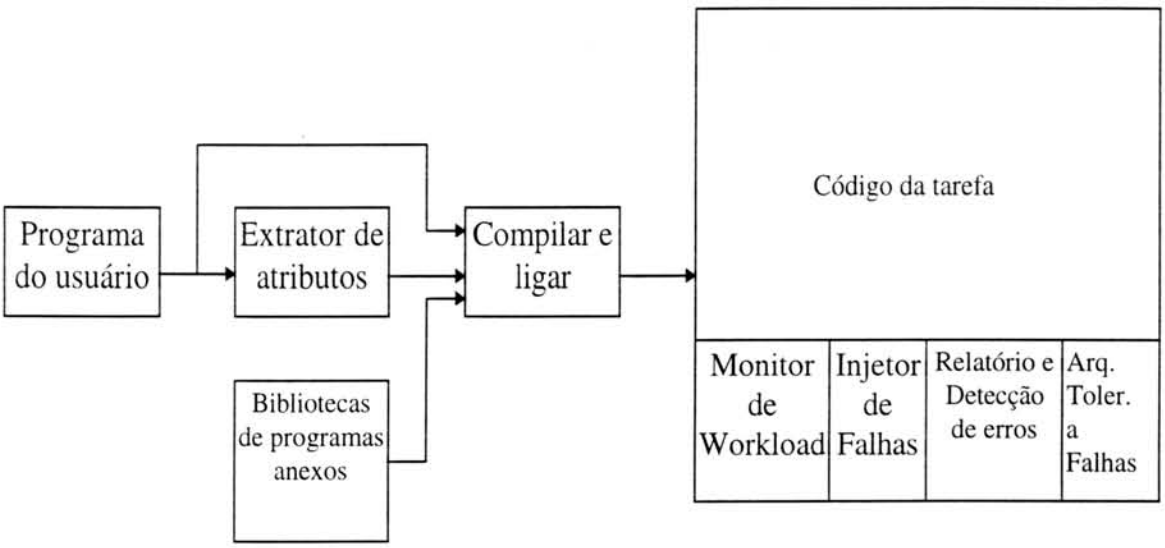


FIGURA 3.1 - Geração de tarefas de FIAT

O **extrator de atributos** extrai do programa do usuário atributos como o conjunto de nomes simbólicos identificando cada tarefa do *workload* bem como os segmentos de código e dados dentro de cada tarefa. Após a extração dos atributos o **extrator de atributos** fornece tabelas conhecidas como **domínios**.

Uma classe de falhas é um padrão que descreve um conjunto de workloads ou modificações do sistema, que são representativos de um grupo de falhas físicas/lógicas que tem propriedades comuns. Alguns dos domínios são extraídos pelo extrator de atributos e alguns são padrões (*defaults*) do sistema. Como em qualquer tipo abstrato, a classe de falhas pode ser instanciada, significando que cada método é aplicado ao domínio associado e uma falha específica (instância de falha) é gerada. Este processo é conduzido automaticamente por uma ferramenta nomeada de Gerador de Instância de Falhas. O conjunto de instâncias de falhas é chamado de Lista de falhas.

FIAT coleta de dados de duas formas:

- Histórias: São registros de eventos (de desempenho e funcionalidade) normais monitorados.
- Relatórios de Erros: São registros de exceções e condições anormais.

FIAT possibilita dois tipos de análise de dados:

- Variedade Fechada: Fornece um conjunto de funções pré-definidas tais como análise do *workload* e estatísticas de cobertura dos erros.
- Variedade Aberta: É uma linguagem de consulta a banco de dados relacional que capacita os usuários a definir as suas próprias metas de análise.

Duas entidades são definidas para os experimentos:

- Descrição do experimento: É uma descrição de alto nível de um fluxo do experimento, que inclui comandos de *workload*, injeção de falhas e coleta de dados.
- *Script* do experimento: É gerado automaticamente por uma ferramenta chamada **EDT** (Tradutor de Descrição de Experimento) a partir da **descrição do experimento**. Contém uma seqüência de comandos detalhada de baixo nível para o controle da injeção de falhas em um sistema distribuído em tempo de execução.

A implementação de hardware e software de FIAT reflete aquela de um sistema distribuído de tempo real.

A estrutura de hardware de FIAT é composta de dois componentes, que são ligados por uma rede de comunicação:

- **FIRE** (Receptores de Injeção de Falhas): Fornecem a plataforma de execução para o sistema distribuído sob teste, através do monitoramento do *workload* e a injeção de falhas.
- **FIM** (Gerente de Injeção de Falhas): Suporta o desenvolvimento do experimento e coleta/análise de dados, além de controle em tempo de execução do experimento.

A arquitetura de hardware de FIAT é mostrada na figura 3.2.

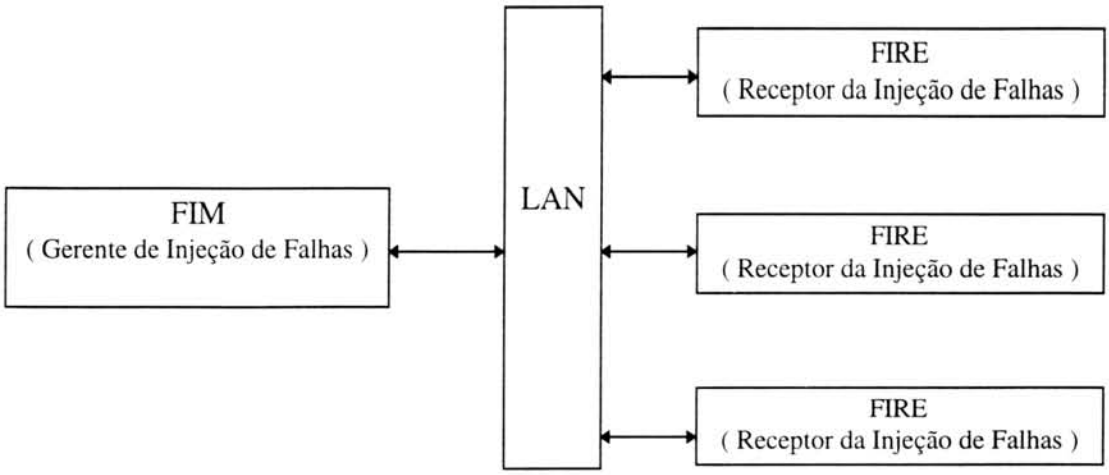


FIGURA 3.2 - Arquitetura de hardware de FIAT

A estrutura de software de FIAT também é dividida em dois componentes:

- **Software FIM:** A arquitetura de software FIM é mostrada na figura 3.3.

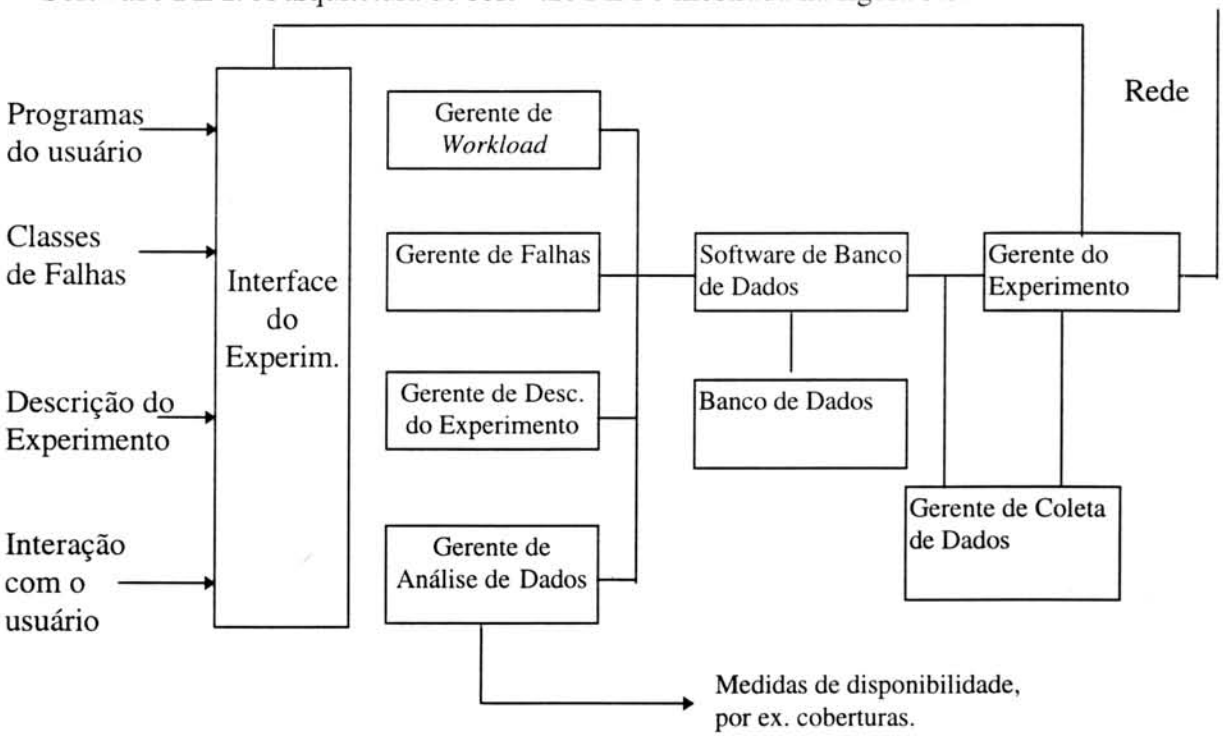


FIGURA 3.3 - Arquitetura do componente de software FIM em FIAT

- **Software FIRE:** Possui dois componentes básicos: Monitor e Controlador FIRE (FMC) e o Kernel do SO Modificado. É mostrado na figura 3.4.

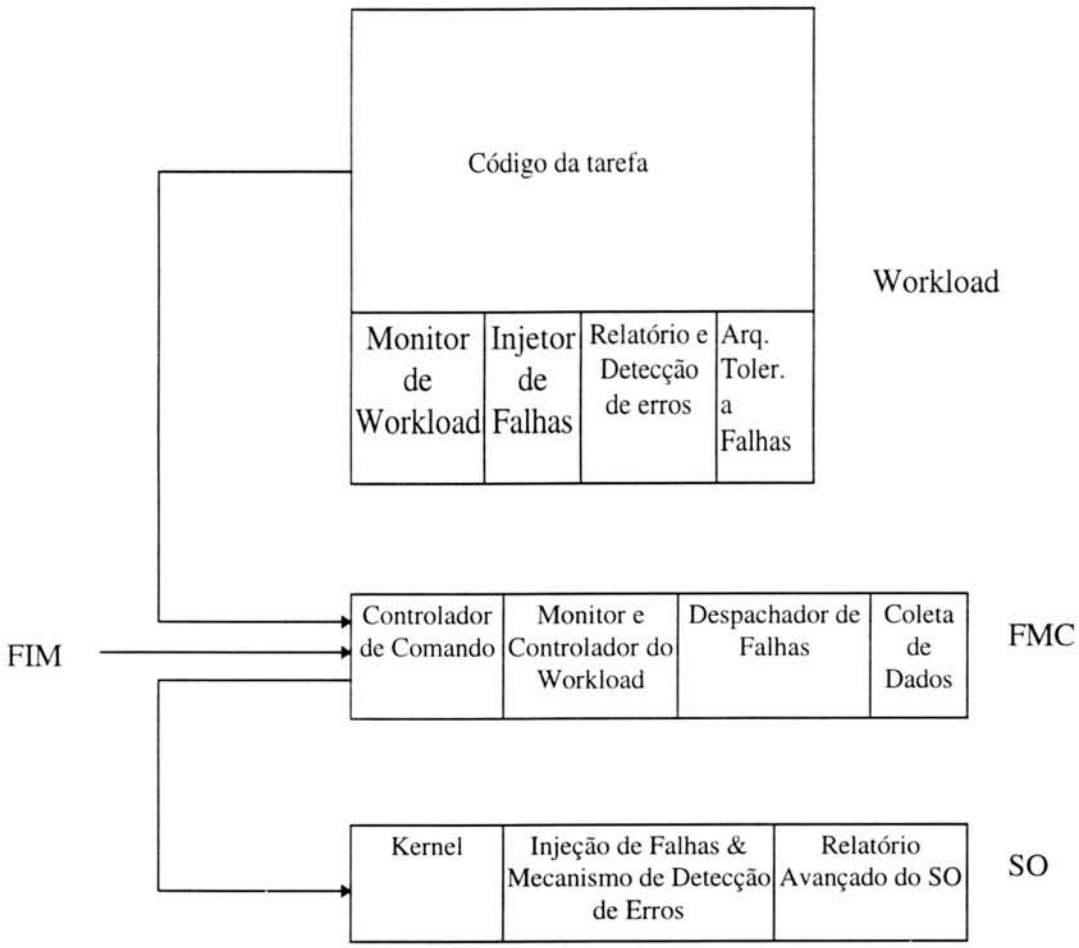


FIGURA 3.4 - Software FIRE detalhado na ferramenta FIAT

3.1.4.2 Arquitetura de EFA

O teste de um sistema distribuído através da injeção de falhas é esquematizado na figura 3.5 segundo o modelo EFA [ECH 92].

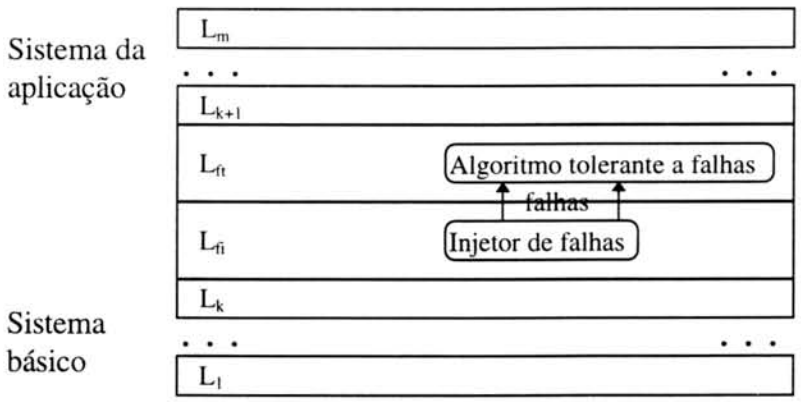


FIGURA 3.5 - Teste de um sistema distribuído tolerante a falhas através da injeção de falhas em EFA

O sistema básico engloba as camadas L_1 a L_k . O sistema da aplicação engloba as camadas L_{k+1} a L_m . A camada L_{ft} é colocada entre duas camadas quaisquer da pilha de camadas do sistema a ser testado, e é a camada onde é implementado o algoritmo

tolerante a falhas. A camada L_{fi} é colocada logo abaixo da camada L_{ft} a fim de injetar as falhas.

O modelo de falhas define os nodos do sistema como regiões de falhas, que podem estar sem falhas ou com falhas. O conjunto de mensagens, que um nodo falho envia durante uma única execução do algoritmo, é definido como um caso de falha. A tarefa do injetor de falhas é a geração de todas as mensagens dos casos de falhas especificados, chamadas de mensagens injetadas. Um exemplo de sistema distribuído que usa este modelo é apresentado na figura 3.6.

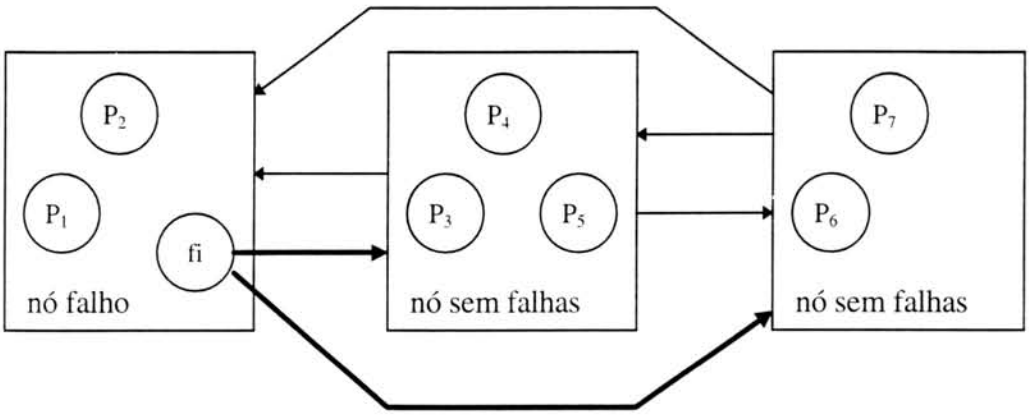


FIGURA 3.6 - Mensagens normais e injetadas em EFA

O injetor de falhas pode ser disparado a qualquer momento através de um evento disparador. O modelo de falhas proposto pela ferramenta EFA suporta falhas de omissão, falhas de temporização, falhas de valor e falhas bizantinas. A especificação de falhas para o injetor de falhas é feita através de programação.

No projeto do injetor de falhas de EFA foram feitas as seguintes decisões estruturais básicas:

- O injetor de falhas é colocado no topo da camada de enlace do sistema de comunicação de forma que as mensagens a serem modificadas possam ser acessadas diretamente.
- Cada nodo possui um injetor de falhas local.
- O programa que o pessoal de teste adiciona ao injetor de falhas é colocado em um módulo chamado **analisador**. Este módulo inspeciona as mensagens de entrada e decide as ações de injeção apropriadas a serem feitas.
- Todos os valores a serem usados para injeções de falhas posteriores devem ser armazenados explicitamente em uma **base de mensagens**.
- Todas as ações dependentes de tempo são feitas por um objeto extra, **a lista de temporização**.
- Os operadores de entrada, saída, envio e recebimento são trocadas por operadores especiais que informam ao **analisador** qual ação tomar.

A figura 3.7 a seguir mostra os componentes e o fluxo de informação em um único injetor de falhas local.

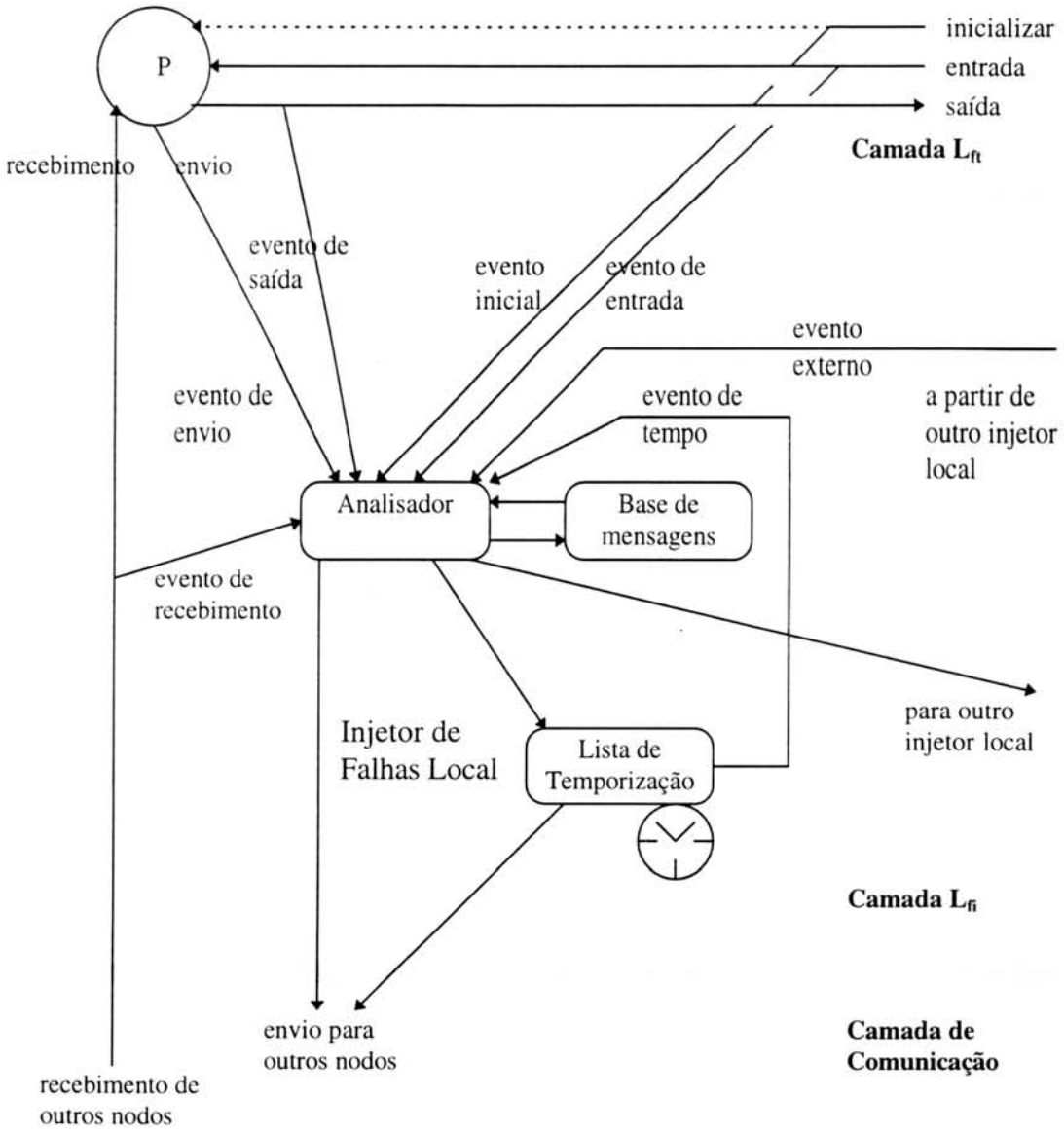


FIGURA 3.7 - Estrutura de um injetor de falhas local usado por EFA

Um **programa de injeção de falhas** é uma coleção de procedimentos de decisão, um para cada injetor de falhas local. Um procedimento de decisão é colocado no seu respectivo **analisador**, onde é chamado sempre que um dos seguintes eventos disparadores ocorrerem:

- evento inicial do início de uma execução de teste para um caso de falhas particular,
- evento de entrada, saída, envio ou recebimento provocado pelo respectivo comando em um processo do algoritmo tolerante a falhas (o processo P na figura 3.7),
- evento de tempo provocado por uma mensagem colocada anteriormente em uma lista de temporização por um procedimento de decisão local,
- evento externo provocado por um injetor de falhas local diferente em algum nodo falho.

O **analisador** toma as suas ações de acordo com o valor do evento disparador e a informação de estado armazenada na base de mensagens. A **base de mensagens** representa a informação de estado do **analisador**, que é organizada como uma pequena

base de dados de mensagens com informações adicionais como rótulo, tempo e número de sequência.

Atualmente, a linguagem de injeção de falhas usada em EFA é uma extensão da linguagem de programação C, com a adição das seguintes primitivas de injeção de falhas: descrição do **evento disparador**, vetor de casos de falhas, acesso a base de mensagens, falsificação do conteúdo das mensagens e transmissão de mensagens.

Assume-se que o pessoal de teste conhece a estrutura das mensagens trocadas entre os processos do algoritmo tolerante a falhas.

Transmissão de mensagens múltiplas é fácil, basta chamar n vezes o operador de envio da camada de comunicação com a mesma mensagem.

Para a falsificação do tempo de transmissão de uma mensagem, a mensagem e alguns parâmetros adicionais como o tempo de transmissão desejado são colocados na **lista de temporização**. Então a lista de temporização é encarregada de enviar a mensagem no tempo especificado.

A transmissão de mensagens é ordenada através do uso de condições de espera, na forma **mensagem A antes de B**. Esta ordenação permite que a execução de um programa seja reproduzível, ou seja, seja determinística. Então pode-se alcançar qualquer grau de determinismo (do não determinismo total ao determinismo total) da execução do programa através da especificação de um conjunto apropriado de condições de espera.

Mensagens espontâneas - tempo de transmissão não depende de nenhum evento induzido pelo procedimento correto do algoritmo tolerante a falhas - podem ser geradas. Com isso é possível provocar envios prematuros de mensagens.

Os esquemas mencionados anteriormente podem ser combinados de modo a obter falhas mais complexas.

O injetor de falhas EFA faz a geração randômica de casos de falhas e também a geração de casos de falhas através da análise do programa.

3.1.4.3 Arquitetura de SFI

SFI é composto de dois componentes principais: o Gerador de Experimento SFI (SEG) e os Módulo de Controle SFI (SCM). O SEG usa um arquivo de descrição de experimento fornecido pelo usuário a fim de criar os arquivos executáveis e os arquivos *scripts* usados para executar os experimentos de injeção de falhas. O SCM consiste de rotinas que fornecem as capacidades de injeção de falhas e modificação do procedimento. O SEG compila as porções apropriadas de SCM com o *workload* para cada nodo. O relacionamento entre os arquivos de SFI é mostrado na figura 3.8.

SFI usa três métodos diferentes de injeção de falhas, de acordo com a acessibilidade dos componentes onde serão injetadas as falhas:

- Injeção Ativa: É realizada por um processo que executa concorrentemente com a execução do *workload*. É utilizada por SFI para injetar erros de memória.
- Alteração do Fluxo de Controle: É usada quando o procedimento do sistema precisa ser alterado. Quando a injeção de falhas é ativada, um programa rodando executa uma seqüência de instruções alternativas, de modo que a função pretendida seja calculada incorretamente. É particularmente útil a nível de sistema operacional ou protocolos de comunicação, onde os serviços disponíveis para programas podem ser alterados de modo que sua funcionalidade difere quando injeção de falhas é ativada. SFI usa esta técnica para injetar defeitos de comunicação.
- Troca de Código: Pode ser usada para injetar falhas em áreas que não são acessíveis quando programas estão executando. Com esta técnica, falhas em uma unidade

funcional podem ser emuladas através da alteração de instruções de programa que usam aquela unidade. SFI usa esta técnica para emular falhas de unidades funcionais do processador.

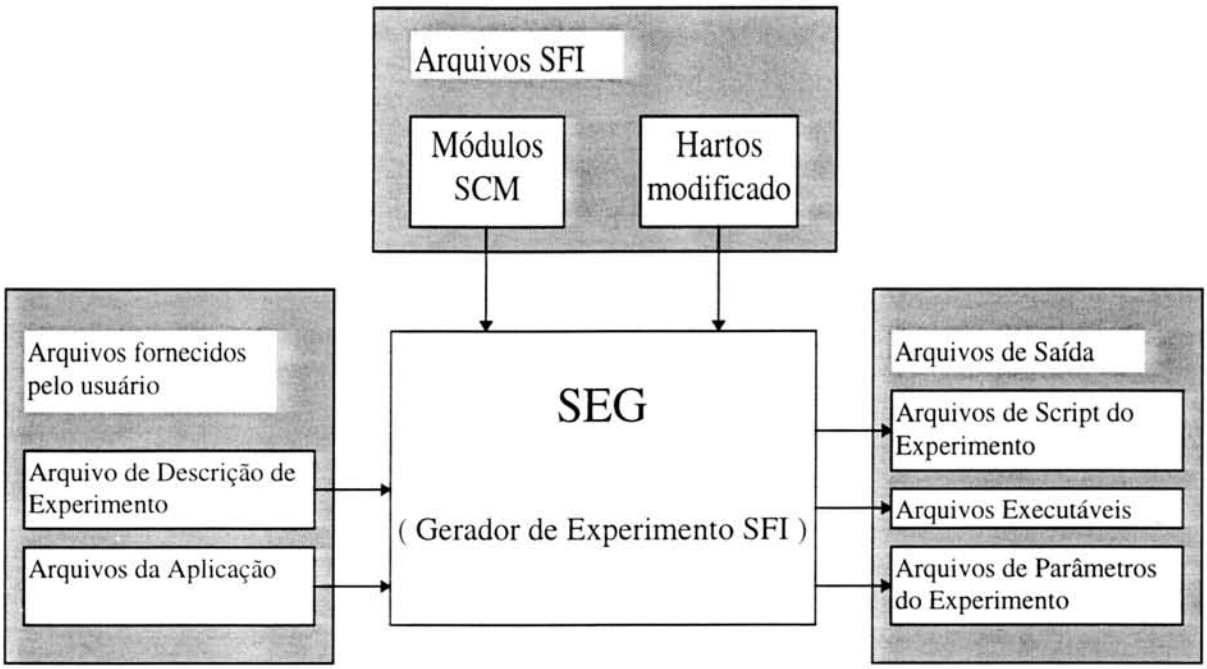


FIGURA 3.8 - Relacionamento entre os arquivos SFI

3.1.4.4 Arquitetura de DOCTOR

DOCTOR possui um arquitetura de software modular, que é dividida em:

- computador hospedeiro: Serve como uma console para controlar a injeção de falhas;
- sistema destino: Corresponde ao sistema a ser testado. É ligado ao computador hospedeiro por uma rede *Ethernet* dedicada.

Os componentes da arquitetura de DOCTOR são:

- **Injetor de Falhas**
  - **EGM** (Módulo de Geração de Experimento)**:** Gera imagens de executáveis que serão posteriormente carregados para o sistema destino. EGM utiliza workloads reais ou gerados por **SWG**. Em qualquer caso, os workloads são compilados, ligados e a informação da tabela de símbolos é extraída para ser referenciada por **ECM**.
  - **ECM** (Módulo de Controle de Experimento)**:** Controla a injeção de falhas, sincronizando o início e o fim de cada experimento em cada nodo, e configura o ambiente pela carga dos executáveis do *workload*, **FIA**s, **DCM**s, e até mesmo do software de sistema se for preciso.
  - **FIA** (Agente de Injeção de Falhas)**:** Recebe comandos de **ECM** via Ethernet e os executa pela injeção de falhas ou fazendo workloads esperar/iniciar/parar.
- **DCM** (Módulo de Coleta de Dados) e **HMON** (Hardware MONitor)**:** Ambas as ferramentas armazenam os eventos gerados pelo objeto monitorado. Para experimentos que requerem alta precisão em medidas temporais é usado **HMON**, que é um monitor implementado em hardware.
- **DAM** (Módulo de Análise de Dados)**:** Faz a análise dos dados obtidos nos experimentos.

- **SWG** (Gerador de *Workload* Sintético)**:** Geram workloads que utilizam os recursos do sistema, de modo a alterar os resultados experimentais.
- **GUI** (Interface Gráfica do Usuário)**.**

A arquitetura de DOCTOR é mostrada na figura 3.9.

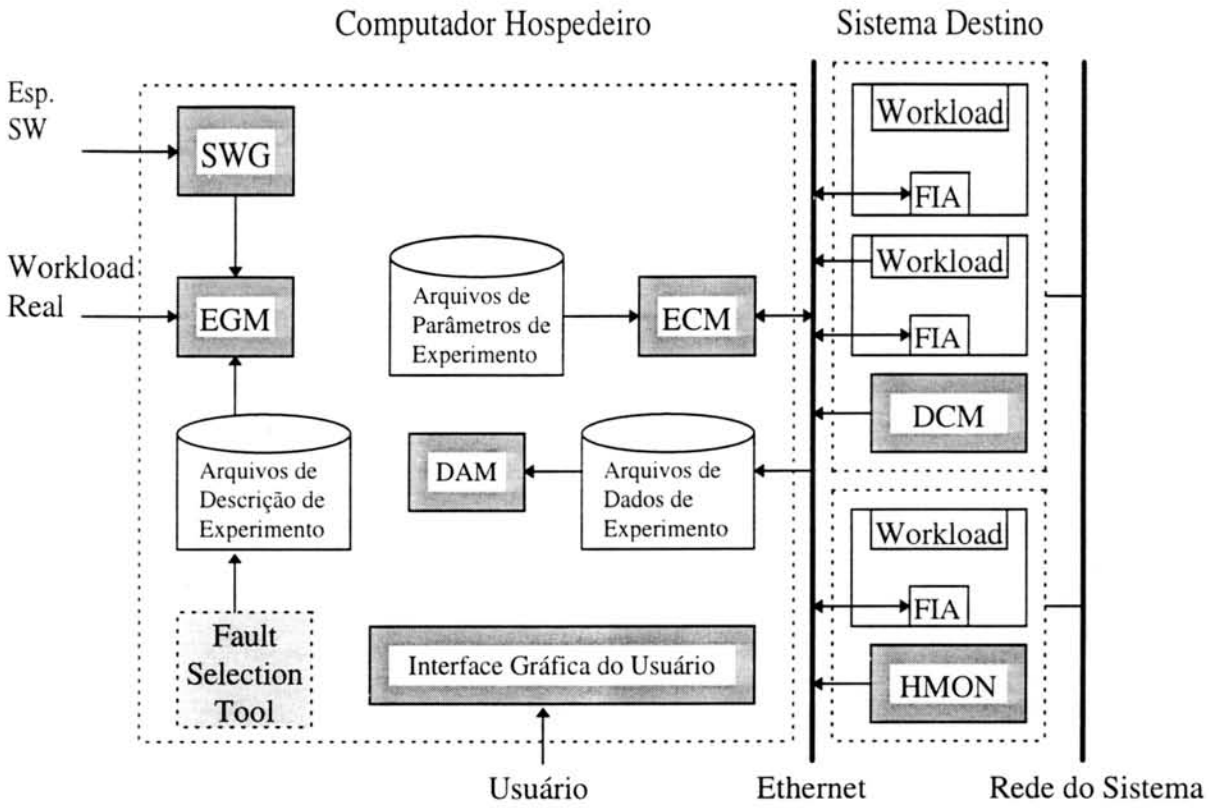


FIGURA 3.9 - Arquitetura de DOCTOR

As falhas de comunicação são injetadas por uma camada de protocolo especial que aceita comandos de **FIA**. Esta Camada de Protocolo Especial (Camada de Injeção):

- constrói a estrutura de história das mensagens
- é colocada geralmente logo abaixo do protocolo ou programa do usuário a ser testado
- é implementada no sistema operacional *x-kernel* [HUT 91]
- intercepta operações UPI (Interface Uniforme de Protocolo) - *send* e *receive* - entre o protocolo sob teste e os protocolos de camadas inferiores

Para cenários de falhas mais complexas várias cópias da camada de injeção podem ser colocadas entre diferentes camadas de protocolo.

3.1.4.5 Arquitetura de PFI

PFI apresenta uma técnica chamada *script-driven probing and fault injection*. Esta técnica visualiza um protocolo distribuído como uma pilha de camadas intercomunicantes, da mesma forma vista pelo *x-kernel*. Ela se baseia na interceptação e filtragem de mensagens entre participantes do protocolo. Para o teste do protocolo destino uma camada *driver* é colocada acima da camada a ser testada, e uma camada *PFI (Probe/fault injector)* é colocada abaixo da camada a ser testada.

Os *scripts* são o coração do método proposto. Eles servem para a especificação de instruções a fim de orquestrar uma computação distribuída para um estado desejado. Além disso, *scripts* permitem a especificação de falhas a serem injetadas no sistema quando um certo estado é alcançado. *Scripts* fazem 3 tipos de operações sobre mensagens:

- Filtragem de mensagens: interceptação e exame das mensagens.
- Manipulação de mensagens: para perda, atraso, reordenação, duplicação ou modificação de uma mensagem.
- Injeção de mensagens: para sondagem (*probing*) de um participante através da introdução de uma nova mensagem no sistema.

A linguagem de especificação de *scripts* é uma linguagem interpretada chamada Tcl. As extensões à linguagem podem ser escritas em C. A recompilação da camada PFI não é requerida quando se faz apenas mudanças nos *scripts* para novos testes, mas é necessária quando as rotinas de biblioteca são mudadas. Pode-se construir várias rotinas de biblioteca genéricas, por ex. atrasar, duplicar, alterar mensagens.

A camada PFI pode gerar apenas mensagens que não alteram o estado do protocolo destino (p. ex. pacotes ACK). Para a geração de mensagens que alteram o estado do protocolo destino (p. ex. pacotes de dados), a geração deve ser feita pela camada *driver*. Os *stubs* são escritos por pessoas que conhecem os formatos dos pacotes do protocolo destino. A arquitetura da ferramenta PFI é mostrada na figura 3.10.

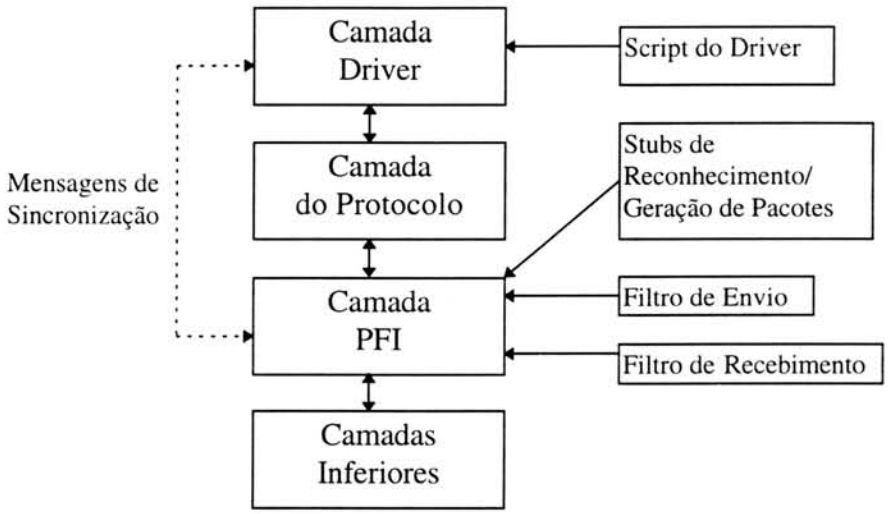
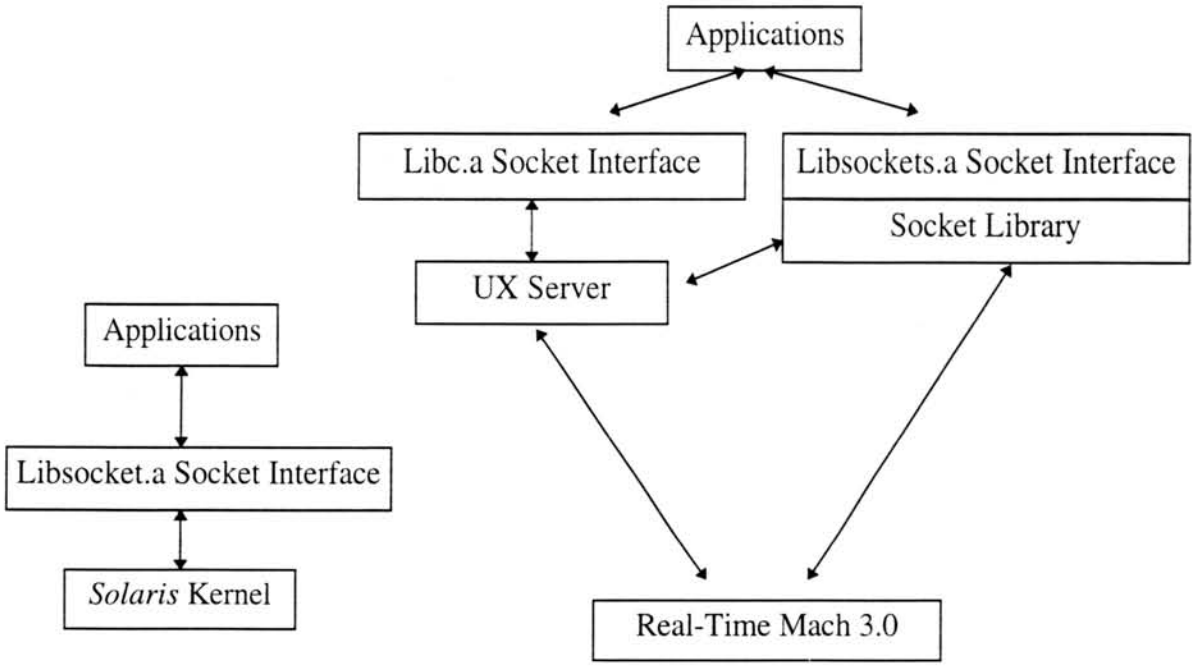


FIGURA 3.10 - Interação entre os scripts em PFI

3.1.4.6 Arquitetura de SockPFI e ORCHESTRA

SockPFI e ORCHESTRA utilizam o modelo da ferramenta PFI mencionada anteriormente. A figura 3.11b apresenta a estrutura utilizada por SockPFI. Ambas as figuras 3.11a e 3.11b representam as estruturas usadas por ORCHESTRA. Uma biblioteca de injeção de falhas é ligada à aplicação. Após executar os procedimentos de injeção da biblioteca, os procedimentos da biblioteca de *sockets* original são chamados. Para compensar a interferência dos mecanismos de injeção nas características temporais, peculiaridades do *Real-Time Mach* são utilizadas. Um dos mecanismos usados é compensar esse *overhead* através da alocação de recursos extras de CPU para as atividades de comunicação. Isto porque a maior parte do *overhead* da técnica PFI ocorre no sistema de comunicação.



a) *Solaris* **b) *Real-Time Mach***

FIGURA 3.11 - Estrutura de *Sockets* em SockPFI e ORCHESTRA

3.1.4.7 Arquitetura de CSFI

CSFI possui os seguintes componentes:

- **EDM** (Módulo de Definição do Experimento): Permite ao usuário definir arquivos com grandes quantidades de descritores de falhas para experimentos múltiplos ou, arquivos com apenas um descritor de falhas para experimentos *ad hoc*.
- **EMM** (Módulo de Gerência do Experimento): Permite ao usuário automatizar o processo de injeção de falhas e fazer experimentos múltiplos sem intervenções adicionais do operador. O operador deve fornecer apenas o nome do arquivo de descrição das falhas, a linha de comando para *benchmark* e o nome do arquivo de saída do *benchmark*. O *benchmark* é então executado sem nenhuma injeção de falhas e alguns dados da execução são coletados. Estes dados são comparados com os dados obtidos a partir da injeção das falhas e armazenados em um arquivo de resultados da injeção.
- **IA** (Agente de Injeção): Para utilizar o CSFI deve-se colocar no código da aplicação duas funções: StartCSFI(...) e StopCSFI() que estão arquivadas em uma biblioteca C para serem ligadas à aplicação do usuário. StartCSFI(...) ativa o IA e inicia a monitorização dos *links*. StopCSFI() termina o processo de injeção e faz a coleta de lixo de CSFI.
- Roteador PARIX alterado.

Os dois primeiros módulos executam sobre o computador *host*, enquanto os dois últimos executam no sistema destino. EDM e EMM podem ser quase diretamente usados com outros sistemas destinos sem modificações. IA deve ser recompilado para o novo sistema destino com poucas alterações no código. O roteador PARIX alterado deve ser totalmente reescrito para o novo sistema destino por ser dependente de sistema. Os dados coletados são armazenados em arquivos no formato Microsoft Excel. A arquitetura de CSFI é mostrada na figura 3.12.

As falhas de comunicação são emuladas em CSFI por um camada modificada do protocolo de comunicação de PARIX. Esta escolha permite obter mais generalidade e independência da aplicação pois não é necessário alterar a aplicação, apenas as mensagens.

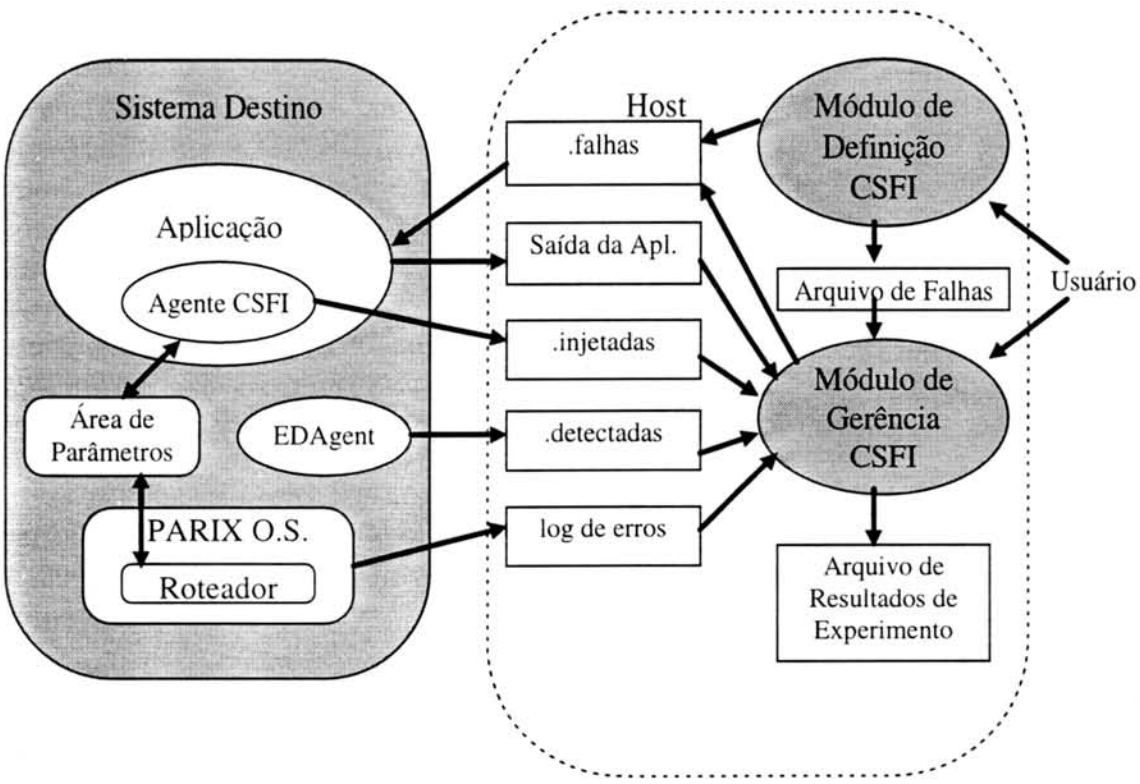


FIGURA 3.12 - Interfaces de CSFI

3.1.5 Experimentos realizados

Esta seção apresenta os experimentos realizados pelas ferramentas analisadas.

3.1.5.1 Experimentos realizados com FIAT

Os experimentos iniciais apresentados por Segall [SEG88] foram dois: teste de um sistema de *checkpointing* distribuído em tempo real tolerante a falhas e teste de uma estratégia *duplicate-match*.

Posteriormente Barton [BAR90] apresentou resultados de outros experimentos. Foram realizados experimentos utilizando multiplicação de matrizes e ordenação por seleção para representarem tarefas de aplicação. E uma estratégia de detecção de erros *duplicate and compare* para representar uma arquitetura tolerante a falhas.

3.1.5.2 Experimentos realizados com EFA

Aplicação em protocolos de mascaramento de falhas, concordância e ordenação (*ordering*) [ECH 94].

3.1.5.3 Experimentos realizados com SFI

Foram realizados dois experimentos usando SFI: teste de um modelo para prever o efeito de defeitos de comunicação intermitentes nos tempos de entrega (*delivery*) de mensagens entre dois nodos HARTS adjacentes e, o efeito de perdas de mensagens em diferentes algoritmos de roteamento.

3.1.5.4 Experimentos realizados com DOCTOR

Para ilustrar o funcionamento de DOCTOR foram conduzidos dois experimentos de injeção de falhas: medida da latência de erro de um mecanismo de detecção de erros *duplicate-match* e avaliação de um algoritmo de diagnóstico probabilístico distribuído.

3.1.5.5 Experimentos realizados com PFI

Foram conduzidos experimentos com implementações de TCP (*Transmission Control Protocol*) e GMP (*Group Membership Protocol*).

3.1.5.6 Experimentos realizados com SockPFI

Foram realizados experimentos sobre uma implementação de um protocolo de cópia primária de tempo real (*real-time primary backup*) rodando sobre *Real-Time Mach*.

3.1.5.7 Experimentos realizados com ORCHESTRA

Foram realizados experimentos para a avaliação de duas implementações de protocolos. A primeira implementação avaliada foi um protocolo assíncrono de *group membership* desenvolvido sobre o sistema operacional *Solaris*. A segunda implementação foi uma aplicação de áudio-conferência de tempo real (*RT-Phone*), desenvolvida pela CMU (*Carnegie Mellon University*) rodando sobre *Real-Time Mach*.

3.1.5.8 Experimentos realizados com CSFI

O objetivo dos experimentos apresentados foi analisar a porcentagem de falhas que levam a aplicação paralela a produzir resultados errados em um sistema paralelo comercial típico sem técnicas de tolerância a falhas. Foram realizados experimentos com 4 implementações:

- Nqueens: O número de possíveis locais de N rainhas em um tabuleiro de xadrez de forma que elas não se afetem. Aplicação mestre-escravo.
- Cálculo do π: Cálculo numérico do valor aproximado de π. Aplicação mestre-escravo.
- Multiplicação de Matrizes: Aplicação mestre-escravo.
- Concordância Bizantina: Concordância bizantina entre 3 processos.

3.2 Questões de Projeto de Ferramentas

As oito ferramentas (FIAT, EFA, SFI, DOCTOR, PFI, CSFI, SockPFI, ORCHESTRA) serviram de base para a determinação das características principais que devem ser consideradas no projeto de uma ferramenta de injeção de falhas, características estas que foram seguidas no desenvolvimento de AFIDS. A comparação entre as ferramentas analisadas e a arquitetura proposta AFIDS é realizada em cada uma das seções seguintes.

3.2.1 Interferência mínima sobre o sistema destino

A interferência da ferramenta de injeção de falhas sobre o sistema destino deve ser o mínimo possível de modo que não prejudique a precisão das medidas obtidas nos experimentos. FIAT, SFI, DOCTOR, CSFI consideram esta questão separando a arquitetura da ferramenta em duas partes: a) hospedeiro - faz a gerência do processo de injeção de falhas e b) sistema destino - o sistema que será testado. Interferência mínima é essencial em sistemas de tempo real, onde as características temporais devem preservadas. SockPFI e ORCHESTRA consideram o aproveitamento de características

do sistema operacional de tempo real (*Real-Time Mach*) para minimizar o impacto da ferramenta de injeção de falhas. AFIDS permite a divisão das ferramentas em hospedeiro e sistema destino.

3.2.2 Fácil expansibilidade para novos tipos de falhas

Importante porque capacita a construção de cenários complexos de falhas baseada em um conjunto de falhas primitivas. Todas as ferramentas analisadas consideram esta questão. AFIDS permite a construção de novos tipos de falhas.

3.2.3 Automação dos experimentos

Esta característica visa acelerar o processo de injeção de falhas, e é realizada usando três aspectos importantes: a) uso de arquivos para a descrição dos experimentos de injeção de falhas, b) geração automática de casos de falhas e c) geração de *workloads* sintéticos. *Workloads* sintéticos [HAN 93] são programas gerados especificamente para possibilitar a avaliação sistemática da ação das características operacionais sobre as características de dependabilidade do sistema a ser testado. Todas as ferramentas analisadas usam arquivos para descrição dos experimentos de injeção de falhas. A única ferramenta que gera automaticamente casos de falhas é EFA. Somente DOCTOR gera *workloads* sintéticos. AFIDS utiliza apenas arquivos para a automação dos experimentos, os itens b) e c) discutidos acima são sugestões para trabalho futuro.

3.2.4 Acesso ao código fonte do protocolo a ser testado no sistema destino

As únicas ferramentas que não precisam do código fonte do protocolo disponível para os experimentos são PFI, SockPFI e ORCHESTRA. SockPFI e ORCHESTRA são refinamentos de PFI e podem ser consideradas versões de um mesma ferramenta. É necessário o conhecimento do formato do pacote de mensagens do protocolo sob teste. SockPFI e ORCHESTRA precisam do acesso ao código objeto do protocolo sob teste. AFIDS permite a construção de ferramentas que acessam ou não o código fonte, de forma similar a PFI, SockPFI e ORCHESTRA.

3.2.5 Injeção de Falhas Determinística

A injeção de falhas determinística, em contraste com a randômica, possibilita guiar o sistema sob teste a estados "difíceis de alcançar" [DAW 94], reproduzir experimentos de injeção de falhas [ECH 92, CAR 95a] e alcançar uma alta cobertura de erros de projeto com um número menor de casos [ECH 91, ECH 94]. As ferramentas EFA, PFI, SockPFI, CSFI e ORCHESTRA fornecem esta característica. AFIDS permite a injeção de falhas determinística mas precisa definir um gerador de casos de falhas que gere um conjunto de falhas mínimo que leve a um alto grau de cobertura.

3.2.6 Portabilidade da ferramenta

Algumas ferramentas de injeção foram construídas especificamente para um sistema distribuído particular, enquanto outras são mais portáveis. SFI e DOCTOR foram projetadas baseadas no sistema distribuído HARTS. CSFI foi implementado para uma rede de transputers, mas pode ser usado em outras plataformas com poucas modificações. EFA foi projetada e implementada sobre o sistema operacional de tempo real A\ROSE. PFI foi inicialmente desenvolvida sobre o *x-kernel* [HUT 91] rodando sobre Mach 3.0 e depois foi portado para o *SunOS*. SockPFI foi desenvolvida sobre

Real-Time Mach. ORCHESTRA roda em *Real-Time Mach*, *SunOS* e *Solaris*. AFIDS está implementado em C++ sobre o sistema operacional Linux. Mas a arquitetura orientada a objetos de AFIDS permite portar facilmente para outras plataformas.

3.2.7 Transparência para a aplicação

Há três formas de implementação do mecanismo de injeção de falhas de comunicação:

a) Adição de um objeto no código fonte: O pessoal de teste ou a própria ferramenta de injeção deve adicionar um objeto de injeção de falhas no código fonte do protocolo tolerante a falhas. Esta é a forma utilizada no estado atual de implementação de uma ferramenta baseada em AFIDS.

b) Colocação de uma camada de injeção entre duas camadas do sistema de comunicação a nível de sistema operacional: O pessoal de teste deve ter acesso ao formato dos pacotes das mensagens e ao núcleo do sistema operacional. As ferramentas EFA, CSFI e SFI usam esta forma. DOCTOR usa esta forma através do sistema operacional *x-kernel* [HUT 91] rodando sobre HARTS. AFIDS permite a construção de ferramentas com esta configuração.

c) Colocação de uma camada de injeção entre duas camadas do sistema de comunicação a nível de usuário: O pessoal de teste deve ter acesso ao formato dos pacotes de mensagens e ao código objeto do aplicação. SockPFI e ORCHESTRA usam esta forma para aplicações que utilizam comunicação através de *sockets*. O código objeto do protocolo é religado com a biblioteca de injeção, que utiliza a biblioteca de *sockets* original. FIAT é implementado de forma semelhante. AFIDS permite a construção de ferramentas com esta configuração.

4 AFIDS

A motivação para o desenvolvimento de AFIDS se deve principalmente à dificuldade enfrentada nas implementações de técnicas de tolerância a falhas, ADC [BAR 95] e FIX [TEI 95], recentemente concluídas. Nessas implementações, que abordam comunicação confiável e recuperação de processos em sistemas distribuídos, a maior dificuldade é determinar o comportamento sob falhas das técnicas implementadas. No sistema FIX foi previsto, mas não implementado, um módulo de injeção de falhas acionado diretamente a partir de chamadas de sistema do Linux. Esse módulo forneceria primitivas que poderiam ser usadas para a construção de ferramentas de injeção de falhas em protocolos de recuperação de processos. No ADC adotou-se outra abordagem. Como o sistema de avaliação de protocolos de difusão roda em um ambiente simulado sob controle de um módulo central, as falhas são injetadas por esse módulo de acordo com uma probabilidade definida por um modelo de entrada. Entretando, apenas uma pequena classe de falhas é tratada. As duas soluções citadas para injeção de falhas são específicas para cada uma das implementações.

AFIDS pretende ser um *framework* para a construção destas ferramentas. Segundo [BOO 96]: "Através do uso de *frameworks* maduros, o esforço de desenvolvimento torna-se mais fácil, porque os principais elementos funcionais podem ser reutilizados".

4.1 Características de AFIDS

AFIDS visa possibilitar a construção de ferramentas de injeção de falhas implementada por software para sistemas distribuídos através do fornecimento de uma biblioteca de classes básica e genérica em C++, que disponibilize uma estrutura para suportar a geração de parâmetros de falhas, a injeção efetiva da falha e o seu controle e, finalmente, a coleta de dados dos experimento e a sua análise. Visa também permitir a validação de protocolos tolerantes a falhas para sistemas distribuídos, seja através de eliminação ou prevenção de falhas.

AFIDS está restrita à emulação de falhas de comunicação em sistemas distribuídos, suportando os seguintes tipos de falhas sobre mensagens: omissão de recebimento, omissão de envio, valor, queda de canal de comunicação, queda de processo, tempo e bizantina. Estas falhas são as mesmas suportadas por PFI [DAW 94]. AFIDS fornece flexibilidade na injeção de falhas pois permite a injeção em vários níveis do protocolo. A princípio pode ser utilizada em qualquer plataforma UNIX com o Compilador GCC da Free Software Foundation.

Atualmente AFIDS é um protótipo. Isto é enfatizado porque, segundo [BOO 96]: "Um *framework* só começa a alcançar maturidade após a sua aplicação em pelo menos três ou mais aplicações distintas".

Qualquer estação de trabalho executando sistemas operacionais UNIX pode suportar AFIDS, visto que foi desenvolvida em C++ e *sockets* sem utilizar características específicas de algum sistema em particular. AFIDS está sendo implementada em PC's rodando o sistema operacional Linux.

AFIDS permite construir ferramentas para testar vários protocolos tolerantes a falhas. Inicialmente está sendo usada para construir uma ferramenta para a validação de um protocolo de comunicação de grupo tolerante a falhas chamado Newtop [EZH 95]. Este protocolo foi implementado em *SunOS* por Macedo [EZH 95]. A validação de Newtop irá auxiliar a refinar AFIDS.

Depois desta fase, AFIDS será usada para construir ferramentas para validar outros protocolos tolerantes a falhas. Já que AFIDS é orientada a objetos, é possível melhorar os mecanismos implementados em AFIDS através da modificação da estrutura de classes que os implementa.

A comparação entre AFIDS e as outras ferramentas existentes está na seção 3.2.

4.2 Componentes de AFIDS

AFIDS fornece mecanismos para gerenciar os principais componentes das ferramentas de injeção de falhas implementada por software para sistemas distribuídos, como ilustrado nas figuras 4.1 até 4.6.

O arquivo *fault_parameters* corresponde ao conjunto *F* (falhas) nos conjuntos FARM (Falhas, Ativações, Resultados e Medidas) [ARL 90]. As ações de envio e recebimento dos objetos *injector* correspondem ao conjunto *A* (Ativações). O arquivo *readouts_data* corresponde ao conjunto *R* (Resultados). O arquivo *measures_data* corresponde ao conjunto *M* (Medidas).

O hospedeiro (*host*) é uma máquina que controla o processo de injeção de falhas. Os objetos *manager*, *generator*, *controller*, *collector*, *analyser* e *name_server* e os arquivos *fault_parameters*, *standard_data*, *readouts_data* e *measures_data* são colocados no hospedeiro. O sistema destino corresponde aos nodos onde os processos do protocolo tolerante a falhas rodam. Os objetos *injector* e o arquivo *faults_base* são colocados no sistema destino.

4.2.1 Objetos

Os principais objetos de AFIDS são:

a) *manager* (figura 4.1) gerencia os outros componentes da injeção de falhas.

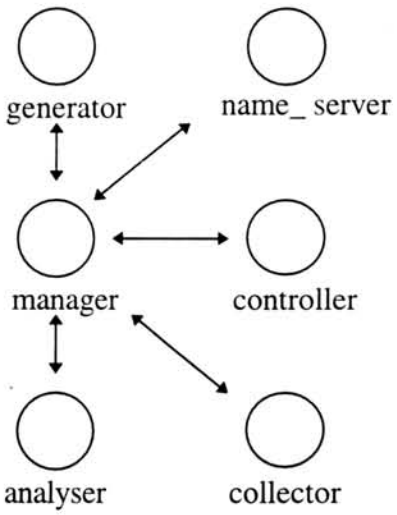


FIGURA 4.1 - Os objetos gerenciados pelo objeto *manager*

b) *generator* (figura 4.2) gera um arquivo de parâmetros de falhas (*fault_parameters*).

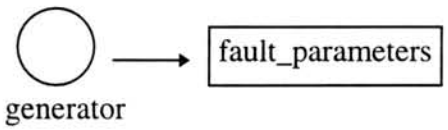


FIGURA 4.2 - O objeto *generator*

c) *name_server* (figura 4.3) é relaciona os objetos *injector*, os processos, os nodos e grupos de processos se houver. Este relacionamento é realizado durante a geração do arquivo de parâmetros de falhas. O arquivo *name_base* é utilizado para armazenar os identificadores. A seção 4.2.3 ilustra em detalhes o objeto *name_server*.

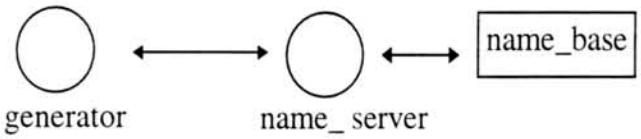


FIGURA 4.3 - O objeto *name_server*

d) *controller* (figura 4.4) controla a conexão entre os objetos *injector*, quando eles são criados ou destruídos, e envia aos objetos *injector* a informação dos parâmetros de falhas.

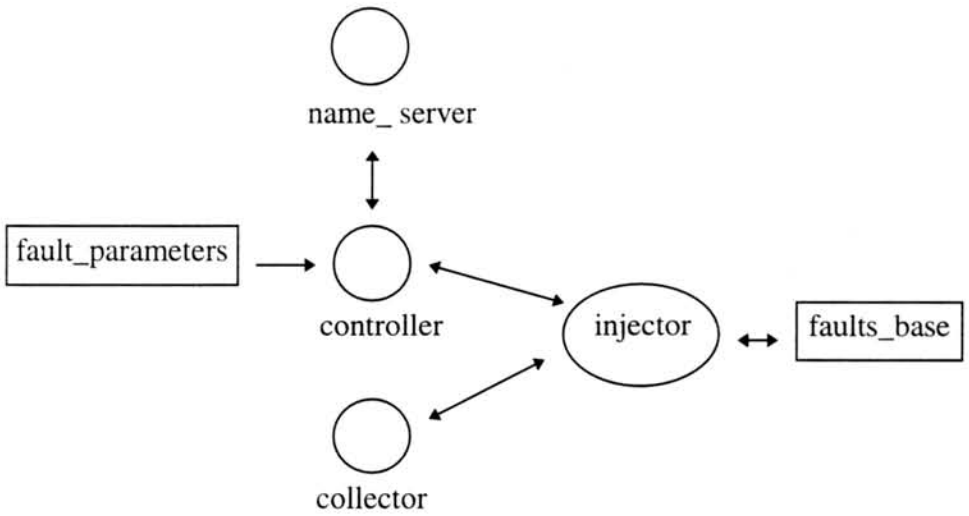


FIGURA 4.4 - O objeto *controller*

e) *collector* (figura 4.5) coleta os dados padrões (a partir da execução do protocolo sem injeção de falhas) colocando os dados no arquivo *standard_data*. Posteriormente coleta os resultados da injeção de falhas (a partir da execução do protocolo com a injeção de falhas, colocando os dados no arquivo *readouts_data*). Controla também a conexão com os objetos *injector* quando eles são criados ou destruídos.

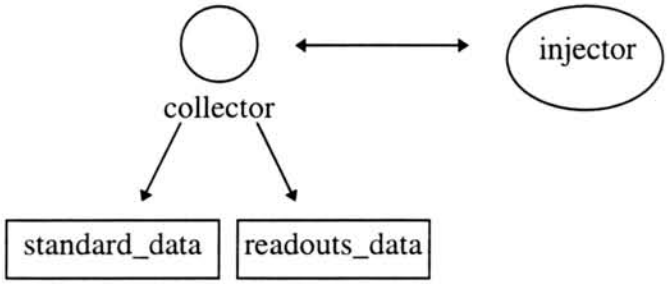


FIGURA 4.5 - O objeto *collector*

f) *injector* (figura 4.4) recebe os parâmetros das falhas que este objeto deve injetar. Estes parâmetros são armazenados em uma base de falhas (*faults_base*) pelo objeto *injector*, que é sempre consultado para que a falha seja injetada no seu devido tempo. Ele também se comunica com os objetos *controller* e *collector* para informá-los sobre a sua criação ou destruição.

g) *analyser* (figura 4.6) avalia o arquivo *standard_data* e o arquivo *readouts_data* de modo a obter medidas de dependabilidade, que são colocadas no arquivo *measures_data*. As medidas de dependabilidade devem ser definidas pelo pessoal de teste.

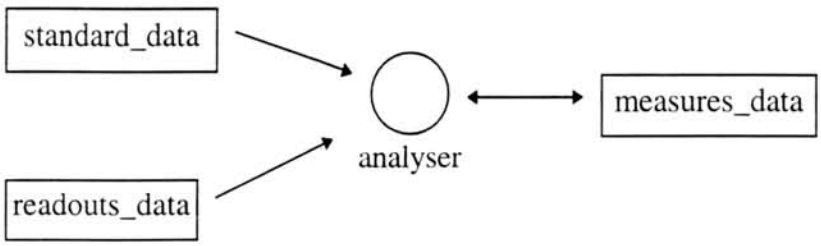


FIGURA 4.6 - O objeto *analyser*

O anexo 2 possui detalhes adicionais sobre os protocolos das classes que definem os objetos especificados nesta seção.

Para a comunicação entre os objetos de AFIDS podem ser utilizadas quaisquer interfaces de comunicação, seja *sockets*, RPC [BIR 84], PVM [SUN 90] ou alguma outra plataforma de comunicação distribuída.

4.2.2 Arquivos

Os principais arquivos de AFIDS:

a) *fault_parameters*: É gerado pelo objeto *generator* ou é escrito pelo pessoal de teste. Armazena os parâmetros das falhas a serem injetadas. A seção 4.2.4 ilustra em detalhes o arquivo *fault_parameters*.

b) *faults_base*: Armazena os parâmetros das falhas que serão injetadas por um objeto *injector* em particular. Ou seja, cada objeto *injector* deve possuir um arquivo *faults_base*. Este arquivo elimina a necessidade de comunicação adicional entre o hospedeiro e o sistema destino após a criação do objeto *injector*.

c) *name_base*: Armazena os identificadores dos processos, grupos, nodos e objetos *injector*. É utilizado pelo objeto *name_server*.

d) *standard_data*: Armazena os dados (valores de variáveis e mensagens) resultantes da execução do protocolo tolerante a falhas sem a injeção das falhas.

e) *readouts_data*: Armazena os dados (valores de variáveis e mensagens) resultantes da execução do protocolo tolerante a falhas com a injeção das falhas.

f) *measures_data*: Armazena as medidas de dependabilidade resultantes da análise dos arquivos *standard_data* e *readouts_data* pelo objeto *analyser*. Exemplos de medidas são a cobertura e a latência dos mecanismos tolerantes a falhas.

Para acelerar o acesso aos dados, os arquivos *name_base* e *faults_base* podem ser implementados em estruturas de dados armazenadas na memória principal em vez de armazená-las na memória secundária.

O anexo 1 possui detalhes sobre as estruturas de dados que armazenam os dados dos arquivos descritos nesta seção.

4.2.3 O Objeto *Name_Server*

O objeto *name_server* é um servidor de nomes que armazena identificadores (no arquivo *name_base*) para os objetos *injector* de modo a facilitar a relação entre o nodo, grupo ou processo onde uma falha deve ser injetada. É composto pela seguinte estrutura de dados:

- Identificação do objeto *injector* (*InjectorId*): uma cadeia de caracteres que identifica o objeto *injector*.
- Endereço do objeto *injector* (*InjectorAddress*): o endereço IP do nodo.
- Nodos responsáveis pelo objeto *injector* (*NodesList*): os nomes dos nodos.
- Grupos responsáveis pelo objeto *injector* (*GroupsList*): os identificadores dos grupos que o objeto *injector* é responsável.
- Processos responsáveis pelo objeto *injector* (*ProcessList*): os identificadores dos processos que o objeto *injector* é responsável.

As operações básicas são (baseadas no modelo SNS (*Simple Name Service*) [COU94]):

- *Bind* (*InjectorId, InjectorAddress, NodesList, GroupsList, ProcessList*) ({*Success, AlreadyExist*}: cria um entrada no *name_server* que armazena as informações sobre cada objeto *injector*.
- *LookupId* (*InjectorId*) ({*Success, NotFound*}: procura uma identificação de um objeto *injector* e retorna os atributos se a identificação é encontrada.
- *LookupAttributes* (*InjectorAddress, NodesList, GroupsList, ProcessList*) ({*Success, NotFound*}: procura uma entrada com os atributos e retorna a identificação do objeto *injector* correspondente.
- *Unbind* (*InjectorId*) ({*Success, NotFound*}: deleta uma entrada no *name_server*.

4.2.4 O arquivo *fault_parameters*

Como exemplo de parâmetros de falhas do arquivo *fault_parameters* podem ser citados:

- Tempo da Injeção de Falhas: Indica em qual mensagem a falha será injetada. É usado o número do pacote da mensagem. Este tempo pode ser escolhido de forma determinística ou randômica. Na implementação apresentada no capítulo 5 este tempo é definido de forma determinística.
- Localização das Falhas: cabeçalho ou os dados da mensagem.
- Duração das Falhas: Pode ser temporário, permanente ou intermitente. Para falhas temporárias, a duração é o número de pacotes de mensagens. Para falhas intermitentes, é um número obtido de uma função de distribuição de probabilidade ou um número dado pelo pessoal de teste. Para falhas permanentes, a injeção de falhas começa a partir do tempo determinado pelo parâmetro "tempo da injeção de falhas" até a destruição do objeto *injector* corrrespondente.
- Nodo onde as falhas serão injetadas: Nome do nodo ou o endereço IP do nodo.
- Processo onde as falhas serão injetadas: É alguma identificação do processo.
- Grupo de processos onde as falhas serão injetadas: É a identificação do grupo de processos que receberão a injeção das falhas. É importante em protocolos que envolvem grupos de processos (por exemplo, comunicação em grupo e replicação de dados).
- Tipo de Falha que será usado: Tipo de falhas de comunicação (todas implementadas pelas funções de envio e recebimento dos objetos *injector*).

- Padrão de Falha: É uma cadeia de caracteres que irá ser usada para modificar o conteúdo das mensagens.

4.3 Opções de projeto de ferramentas baseada em AFIDS

As seções 4.3.1 e 4.3.2 apresentam duas opções de projeto básicas para o desenvolvimento de novas ferramentas de injeção baseadas em AFIDS. Estas opções são apenas duas entre muitas. Outras opções podem ser imaginadas através da manipulação dos objetos e os arquivos de AFIDS de acordo com os objetivos do desenvolvedor. As figuras 4.7 e 4.8 mostram somente quatro processos e três nodos no sistema destino, mas AFIDS permite manipular o número total de processos e nodos necessários para a execução do protocolo tolerante a falhas sob teste.

4.3.1 Com o componente de injeção em cada processo do protocolo tolerante a falhas

A organização dos componentes da ferramenta baseada em AFIDS esboçada na figura 4.7 não foi utilizada por nenhuma das ferramentas analisadas (FIAT, EFA, SFI, DOCTOR, CSFI, PFI, SockPFI, ORCHESTRA).

Esta forma de organização será referida no texto como AFIDS-ip (ferramenta baseada em AFIDS com 1 objeto *injector* por processo do protocolo sob teste) e acopla um objeto injetor dentro do código fonte do(s) processo(s) do protocolo a ser testado. Mas a idéia de acoplar um objeto injetor em cada processo do protocolo sob teste é similar àquela de FIAT, onde o objeto injetor era ligado ao processo do protocolo. E similar à implementação de SockPFI e ORCHESTRA, onde o código objeto do protocolo sob teste é ligado a uma biblioteca de *sockets* alterada (biblioteca responsável pela injeção das falhas), que por sua vez usa a biblioteca de *sockets* original. Ou seja, AFIDS-ip utiliza um objeto injetor explicitamente dentro código fonte do protocolo, enquanto FIAT, SockPFI e ORCHESTRA usam um objeto injetor ligado ao código objeto do protocolo.

Neste projeto de AFIDS-ip, o código do protocolo tolerante a falhas a ser validado tem que estar disponível para ser modificado da seguinte forma:

- Cada processo do protocolo tolerante a falhas que envia ou recebe mensagens incorpora um objeto *injector*, que será colocado no início do processo do protocolo tolerante a falhas.
- As funções de envio e recebimento de cada processo do protocolo tolerante a falhas devem ser substituídas por uma função membro de envio ou recebimento do objeto *injector*. A função membro do objeto *injector* encapsula a função original do processo do protocolo de modo a injetar falhas.

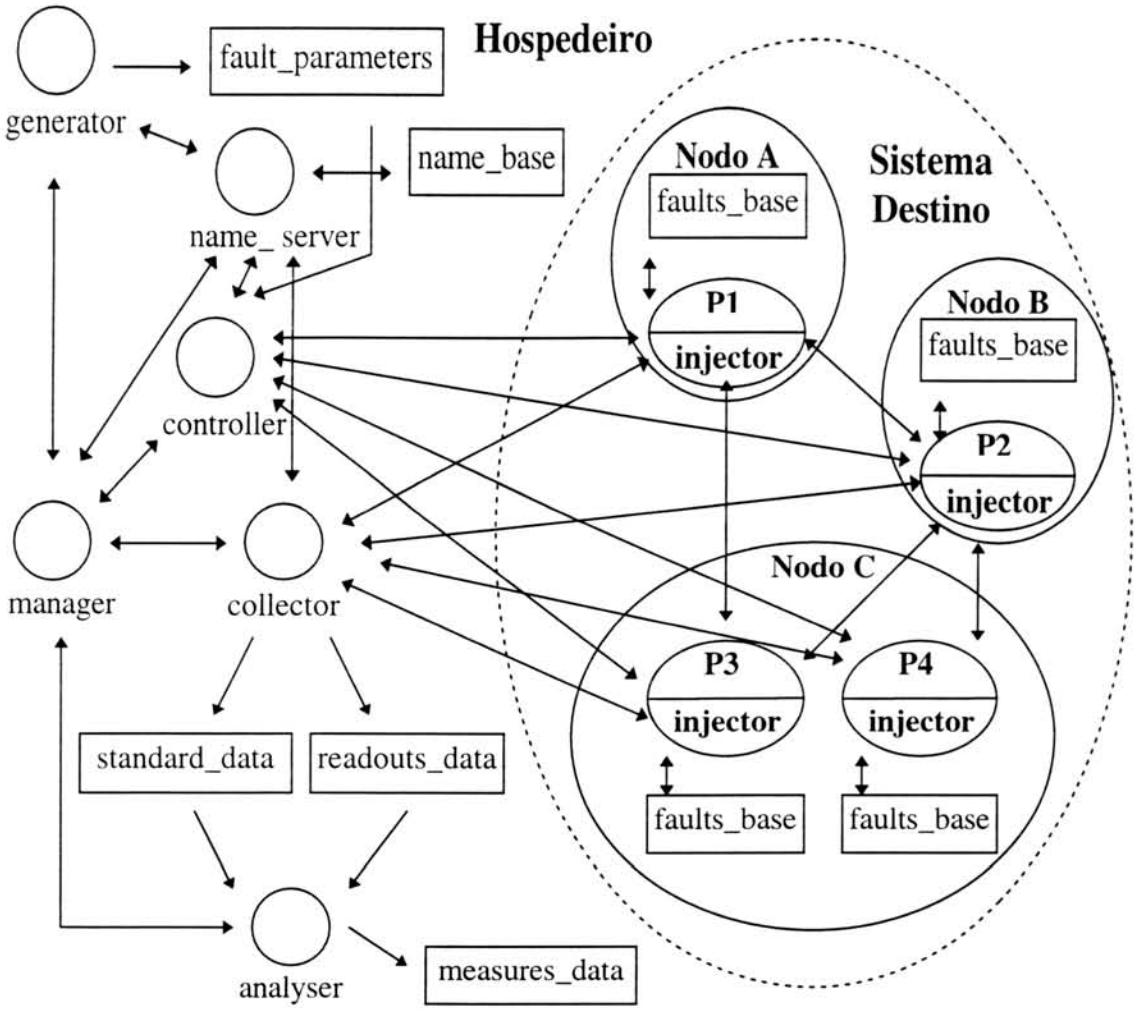


FIGURA 4.7 - Objetos principais de AFIDS-ip (1 objeto *injector* por processo).

4.3.2 Com o componente de injeção entre duas camadas do sistema de comunicação

Devido à sua natureza orientada a objetos, é possível estender AFIDS para suportar a organização em camadas, onde o componente injetor (objeto *injector*) é colocado entre duas camadas de comunicação, da forma que está esboçada na figura 4.8. Nesta forma de organização há a necessidade de acesso ao kernel para incluir a camada de injeção entre duas camadas de comunicação, como em EFA e CSFI. Ou então utilizar o *x-kernel* a nível de S.O. e incluir a camada de injeção dentro do sistema de camadas de comunicação do *x-kernel*, como em DOCTOR. Uma outra forma, seria utilizar o filtro de pacotes existente no *SunOS* (NIT [SUN 87]). A forma da figura 4.8 será referida no texto como AFIDS-in (ferramenta baseada em AFIDS com 1 objeto *injector* por nodo).

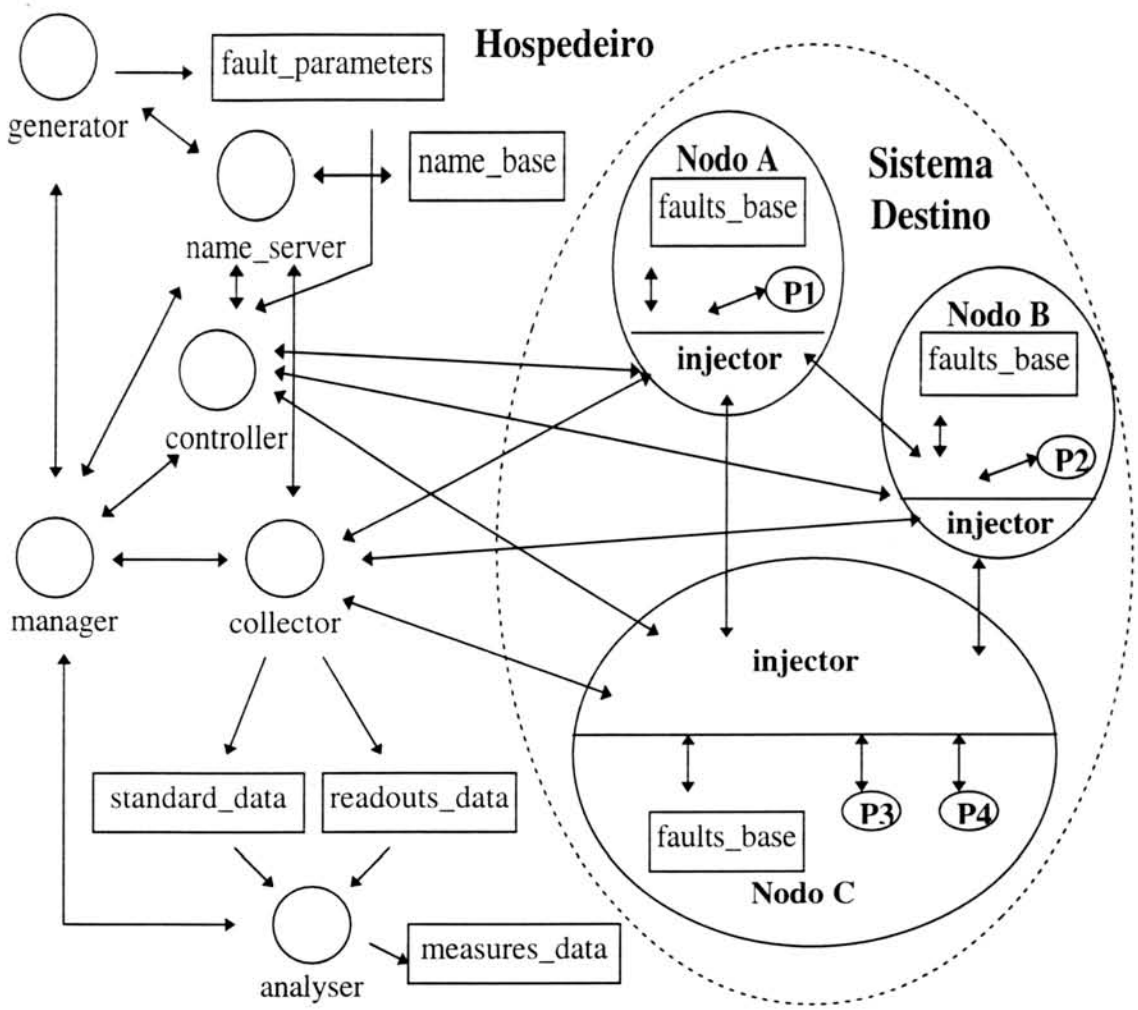


FIGURA 4.8 - Objetos principais de AFIDS-in (1 objeto *injector* por nodo).

4.4 O Diagrama de Classes do Nível mais Alto de AFIDS

O diagrama atual de classes de AFIDS está ilustrado na figura 4.9.

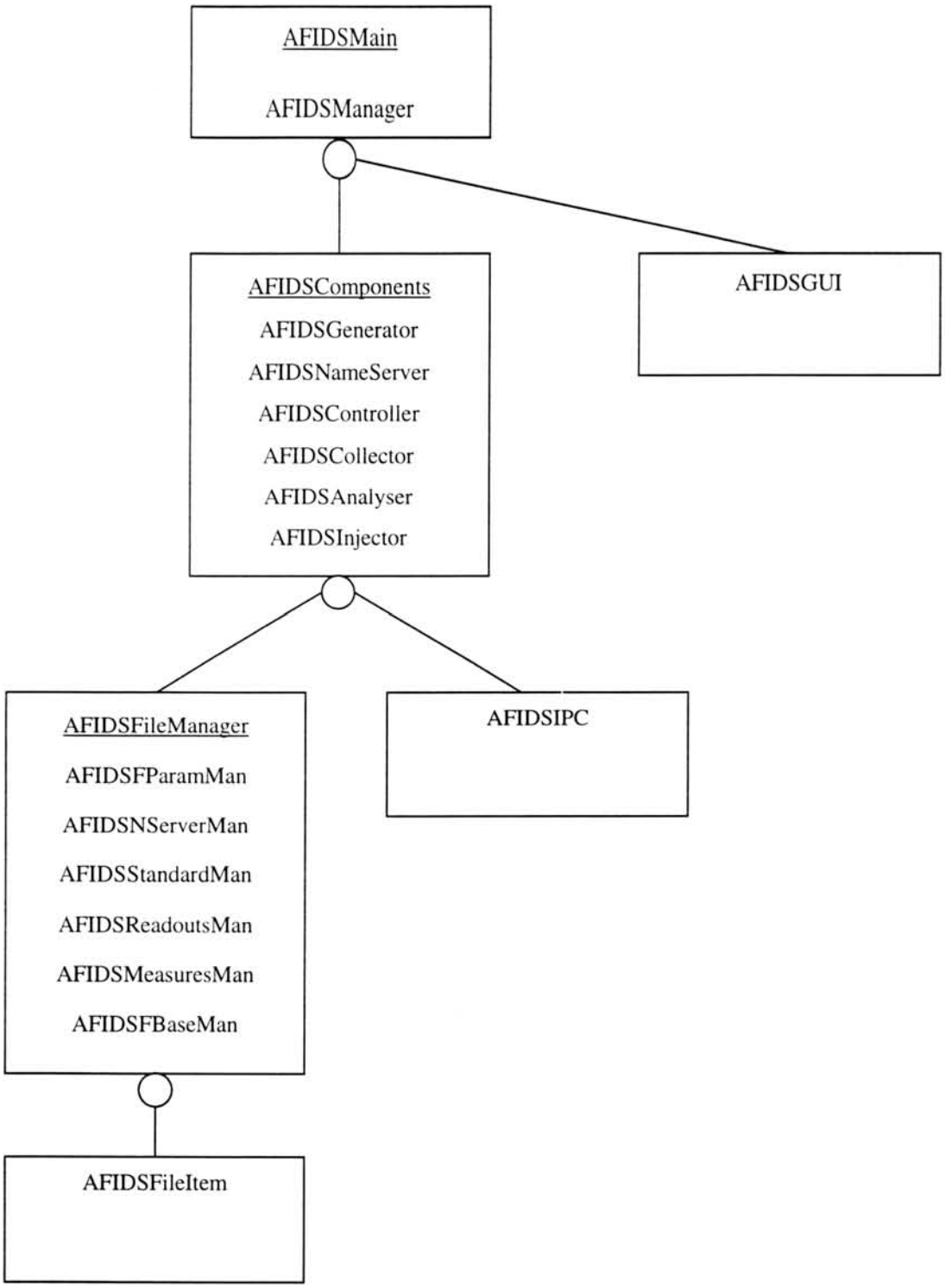


FIGURA 4.9 - Classes do nível mais alto de AFIDS.

Os retângulos esboçados na figura 4.9 representam categorias de classes. As categorias de classes servem para particionar o modelo lógico de um sistema. Uma categoria de classe contém um conjunto de classes ou outras categorias de classes e contribui com operações indiretamente através das classes contidas na categoria. O

anexo 3 apresenta detalhes referentes à notação utilizada. No diagrama da figura 4.9, há seis categorias de classes que apresentam a visão arquitetural de AFIDS:

- AFIDSMain: corresponde ao topo da arquitetura de AFIDS e contém as classes que fazem a gerência dos experimentos. No estado atual de AFIDS apenas a classe AFIDSManager é parte desta categoria. Como mostra o diagrama, as classes pertencentes a AFIDSMain utilizam as categorias AFIDSComponents e AFIDSGUI.
- AFIDSComponents: contém as classes que cooperam entre si na execução dos experimentos de injeção de falhas. As classes desta categoria utilizam as categorias AFIDSFileManager e AFIDSIPC.
- AFIDSGUI: é responsável pela interface gráfica com o usuário. No estado atual de AFIDS esta categoria de classes não foi definida. Trata-se de um trabalho futuro a ser realizado.
- AFIDSFileManager: contém as classes que manipulam os arquivos de AFIDS.
- AFIDSIPC: é responsável pela comunicação através de trocas de mensagens entre os componentes de AFIDS.
- AFIDSFileItem: corresponde às estruturas de dados dos itens que definem os arquivos de AFIDS.

Esta organização através de categorias de classes possibilita que AFIDS utilize qualquer interface de comunicação interprocessos, ilustrado através da categoria de classe AFIDSIPC. Na implementação atual, AFIDS usa *sockets*.

Os formatos dos itens que se relacionam com os arquivos utilizados em AFIDS, que correspondem à categoria AFIDSFileItem descrita acima, estão definidos no Anexo 1. Os protocolos das classes pertecentes às outras categorias descritas nesta seção estão no Anexo 2.

5 Uma Implementação baseada em AFIDS-ip

A implementação inicial de AFIDS, que será usada para avaliar o protocolo NewTop, considera o modelo de uma ferramenta com 1 objeto *injector* por processo por ser mais simples de implementar do que AFIDS-in (ferramenta baseada em AFIDS com 1 objeto *injector* por nodo). Como consequência disto, a biblioteca AFIDS pode ser construída mais rapidamente, provocando um entendimento mais rápido do procedimento de injeção. Apesar de ser necessária a intromissão no código, possui flexibilidade para possibilitar experimentos de injeção de falhas praticamente em qualquer nível de abstração no sistema de comunicação. Por exemplo, um protocolo de comunicação de grupo que utilize *sockets* pode ser validado no mais alto nível de abstração (injeção de falhas nas primitivas de comunicação fornecidas pelo protocolo) ou no mais baixo nível de abstração (injeção de falhas nas primitivas de comunicação a nível de *sockets*).

AFIDS-ip (ferramenta baseada em AFIDS com 1 objeto *injector* por processo do protocolo sob teste) supõe que a camada de transporte é confiável, ou seja, as mensagens são entregues sem corrupção ou perda e em sequência a quaisquer dois processos pertencentes à AFIDS-ip. Para tanto a implementação da comunicação é realizada através da utilização de *sockets* com o protocolo TCP/IP utilizando TCP. Esta implementação possui um mínimo de tolerância a falhas.

Uma implementação futura de AFIDS-in irá reutilizar as classes de AFIDS-ip e irá realimentar a biblioteca AFIDS com novas classes e métodos genéricos. A implementação de AFIDS-in diminuiria a carga existente em AFIDS-ip, onde há um objeto *injector* em cada processo do protocolo. Isto porque haveria apenas um objeto *injector* por nodo. Mais ainda, com AFIDS-in, há a possibilidade de não ser necessária a intromissão no código fonte do protocolo, por exemplo, de forma similar a PFI ou EFA.

5.1 Componentes de AFIDS-ip

A figura 5.1 refina a figura 4.7 criando o objeto *controller-collector* através da agregação dos objetos *controller* e *collector*. Isto para que haja uma diminuição na comunicação entre o sistema destino e o hospedeiro.

AFIDS-ip tem duas características importantes: gerenciamento flexível da criação e destruição de processos e comunicação reduzida entre os objetos *controller-collector* e *injector*.

O gerenciamento flexível da criação e destruição dos processos do protocolo tolerante a falhas, permitindo que os objetos *injector* informem sobre a suas próprias criações e destruições ao objeto *controller-collector*, possibilita que os objetos *injector* sejam criados e destruídos dinamicamente durante a execução do protocolo tolerante a falhas.

A comunicação é reduzida entre o objeto *controller-collector* e os objetos *injector*, porque quando um objeto *injector* é criado, o objeto *controller-collector* envia a ele os parâmetros das falhas e a identificação única gerada pelo objeto *name_server*. Então o objeto *injector* armazena no arquivo *faults_base* os parâmetros de falhas recebidos e que serão injetadas por ele. Depois disso, os objetos *injector* não precisam esperar por uma mensagem do objeto *controller-collector* a respeito das falhas a serem injetadas, reduzindo a comunicação entre eles.

A modelagem de AFIDS-ip usando BOOCH está descrita no Anexo 3. O formato dos pacotes de comunicação entre os componentes de AFIDS-ip estão descritos no Anexo 4.

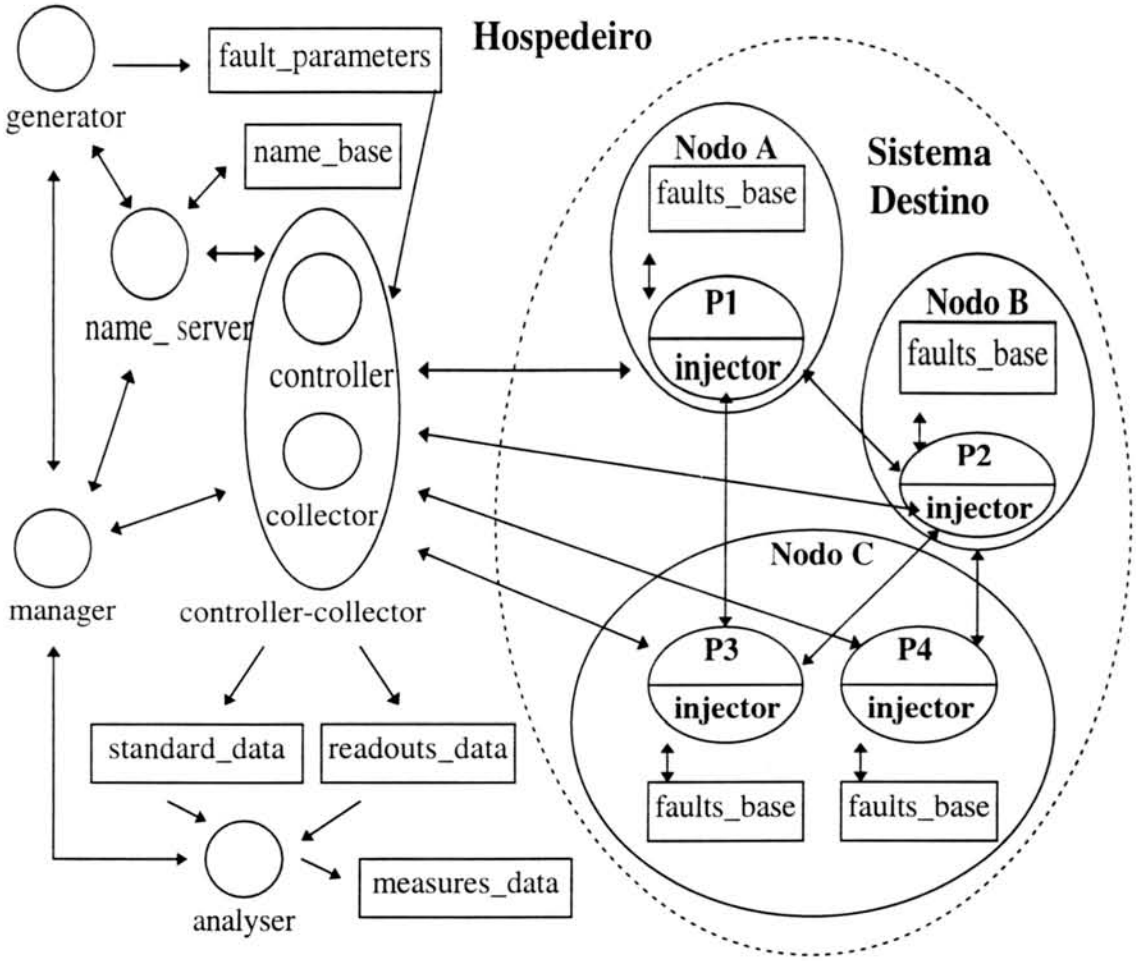


FIGURA 5.1 - AFIDS-ip modificada

5.2 O Cenário de Execução de AFIDS-ip

Um cenário típico da injeção de falhas utilizando a arquitetura da figura 5.1 é:

1. O objeto *manager* é criado. O construtor do objeto *manager* instancia os objetos *name_server*, *generator*, *controller-collector* e *analyser*.

2. Uma função membro do objeto *manager* pergunta ao usuário sobre quatro opções:

- Iniciar o processo de injeção de falhas. Esta opção supõe que um arquivo *fault_parameters* esteja disponível. Vá para o Passo 3.
- Gerar o arquivo *fault_parameters*. O objeto *generator* interage com o usuário e cria o arquivo *fault_parameters*. Após os parâmetros de falhas serem gerados, o objeto *generator* invoca o objeto *name_server* que armazena os identificadores no arquivo *name_base*. Vá para o Passo 2.
- Chamar o objeto *analyser* que analisa os arquivos *standard_data* e *readouts_data* e gera o arquivo *measures_data*. Vá para o Passo 2.
- Terminar o processo de injeção de falhas. Vá para o Passo 7.

3. O objeto *controller-collector* inicia o gerenciamento de sinais (mensagens) dos objetos *injector*. Ele fica esperando por dois possíveis sinais provenientes dos objetos *injector*:

- No recebimento de um sinal de criação, ele estabelece a conexão para o novo objeto *injector* e requisita ao objeto *name_server* uma identificação para o objeto *injector* recém-criado. Além disso, o objeto *controller-collector* lê o arquivo *fault_parameters* e obtém os parâmetros de falhas que o objeto *injector* recém-criado deve injetar. Então o objeto *controller-collector* envia ao objeto *injector* os parâmetros de falhas de interesse do novo objeto *injector* e um identificador único para o objeto *injector*.
- No recebimento de um sinal de destruição, ele fecha a conexão com o objeto *injector*.

4. O objeto *controller-collector* então fica esperando pelos dados dos objetos *injector* para gerar o arquivo *readouts_data*. Para cada criação de um processo do protocolo tolerante a falhas, cada construtor do objeto *injector* envia um sinal de criação para o objeto *controller-collector*. Para cada destruição de um processo do protocolo tolerante a falhas, cada destrutor do objeto *injector* envia um sinal de destruição para o objeto *controller-collector*.

5. O protocolo tolerante a falhas é executado e os dados da execução são coletados pelo objeto *controller-collector*, que gera o arquivo *readouts_data*.

6. Quando o objeto *controller-collector* percebe que a execução do protocolo tolerante a falhas terminou (discutido na seção 5.3), ele envia um sinal (mensagem) para o objeto *manager* informando-o sobre o término da execução do experimento de injeção. Vá para o Passo 2.

7. O objeto *manager* é destruído. O destrutor do objeto *manager* destrói os objetos *name_server*, *generator*, *controller-collector* e *analyser*.

5.3 Detecção da Terminação da Execução do Protocolo Tolerante a Falhas

De modo a permitir que o objeto *controller-collector* saiba do término da execução do protocolo tolerante a falhas, seja sem ou com injeção de falhas, o seguinte procedimento é usado:

a) São usadas duas variáveis auxiliares no objeto *controller-collector*:

- int creationCounter: é um contador de objetos *injector* criados. Inicialmente é inicializado com 0.
- char someInjectorCreated: é uma variável lógica que indica se algum objeto *injector* já foi criado (1) ou não (0). Inicialmente é inicializado com 0.

b) Cada vez que o objeto *controller-collector* recebe um sinal de criação de algum objeto *injector*, creationCounter é aumentado de 1 unidade. Na primeira vez que creationCounter é incrementado, a variável lógica someInjectorCreated é setada para 1.

c) Cada vez que o objeto *controller-collector* recebe um sinal de destruição de algum objeto *injector*, creationCounter é diminuído de 1 unidade.

d) Quando someInjectorCreated for igual a 1 e creationCounter for igual a 0, então o objeto *controller-collector* sabe que a execução do protocolo já terminou e retorna o controle do experimento para o objeto *manager*.

5.4 Implementação das Falhas de Comunicação

De acordo com o tipo de falha a ser injetado, os seguintes procedimentos são executados (todas implementados pelas funções membro de envio e recebimento dos objetos *injector*):

- omissão de recebimento: A função membro de recebimento simplesmente ignora a mensagem recebida.
- omissão de envio: A função membro de envio simplesmente não envia a mensagem.
- valor: A função membro de envio e recebimento modifica o valor do cabeçalho ou o segmento de dados da mensagem.
- queda de canal de comunicação: Este tipo de falha usa as falhas de omissão de envio e de recebimento. É usado sobre um canal de comunicação para evitar o envio e o recebimento de mensagens sobre ele.
- queda de processo: Este tipo de falha usa a falha de queda de canal de comunicação. É usada em todos os canais de comunicação para um processo, evitando o envio e recebimento de mensagens sobre o processo.
- queda de nodo: Este tipo de falha usa a falha de queda de processo. É usada em todos os processos de um nodo evitando o envio e o recebimento de mensagens sobre o nodo.
- temporização: O envio ou o recebimento de uma mensagem é atrasada.
- bizantino: Combinação das falhas especificadas acima.

6 Utilização da Ferramenta AFIDS-ip

Para a utilização de AFIDS-ip (ferramenta baseada em AFIDS com 1 objeto *injector* por processo do protocolo sob teste), deve-se seguir os seguintes passos: 1) decidir em qual nível no sistema de comunicação as falhas serão injetadas, 2) construir um arquivo de parâmetros de falhas, 3) inserir um objeto *injector* em cada processo do protocolo tolerante a falhas, 4) encapsular as funções de envio e recebimento do protocolo dentro dos métodos da classe de injeção, 5) executar os experimentos e 6) analisar os dados coletados. Este capítulo discute cada um destes passos.

6.1 Escolha do nível do sistema de comunicação para a injeção de falhas

Em protocolos ou sistemas onde se tem acesso ao código fonte, e cujas implementações possuem mais de um nível de abstração, é possível escolher em qual deles a injeção de falhas será realizada.

Por exemplo, em um protocolo de comunicação de grupo confiável, que é implementado usando *sockets*, há no mínimo dois níveis de abstração em relação à comunicação:

- o nível mais baixo correspondente ao nível de *sockets*: onde as funções de envio e recebimento entre os processos são executadas via chamadas a rotinas de biblioteca de *sockets*. Neste caso, a injeção de falhas poderia ser implementada através do encapsulamento das funções de envio e recebimento de *sockets* (*read*, *recv*, *write*, *send*, etc).
- o nível do protocolo correspondente à utilização de grupos: a injeção de falhas poderia ser implementada através do encapsulamento das funções responsáveis pelo envio e recebimento de mensagens de grupos de processos. Por exemplo, dois grupos, A e B, e as funções de envio (A.send() e B.send()) e recebimento (A.receive() e B.receive()).

Esta possibilidade de escolha auxiliaria na detecção de possíveis erros de projeto/implementação do protocolo nos diferentes níveis de abstração.

6.2 Construção de um arquivo de parâmetros de falhas

Estudos para a geração de parâmetros de falhas, por exemplo, baseados no código fonte do protocolo [ECH 91] ou em modelos formais (por ex. redes de Petri) [ECH 94], preocupam-se em aumentar a cobertura sobre os erros de projeto com um número mínimo de casos de falhas. O estado atual de implementação AFIDS-ip se preocupa apenas em especificar as falhas. A utilização de métodos sofisticados é deixado como trabalho futuro.

Para exemplificar, admita um protocolo de comunicação de grupo confiável com os seguintes nodos (Sirius e Rigel), grupos (A, B e C) e processos (A1, A2, A3, B1, B2, C1 e C2), especificados no arquivo de parâmetros de falhas (*fault_parameters*):

```
Node Sirius{
        Groups (processes): A (A1, A2, A3), B (B1, B2);
        Injector Id : IA1, IA2, IA3, IB1, IB2;
};
Node Rigel{
        Groups (processes): C (C1, C2);
```

```
        Injector Id : IC1, IC2;
};
```

Os parâmetros de falhas , ainda no arquivo *fault_parameters*, são especificados da seguinte forma:

```
FaultParameters {
        Fault 0{
                InjectionTime: 10;
                FaultLocation: header;
                FaultDuration: temporary (5);
                Node: Sirius;
                Process: A1;
                FaultPattern: "lixo no cabeçalho";
        }

        Fault 1{
                InjectionTime: 30;
                FaultLocation: header;
                FaultDuration: intermitent (10);
                Node: Sirius;
                Process: B2;
                FaultPattern: "lixo no cabeçalho";
        }

        Fault 2{
                InjectionTime: 20;
                FaultLocation: data;
                FaultDuration: permanent;
                Node: Rigel;
                Group: C;
                Process: C1, C2;
                FaultPattern: "lixo nos dados";
        }

};
```

Cada falha tem um número (Fault **number**) correspondente à posição em um vetor de falhas. O tempo de injeção (InjectionTime: **number**) corresponde a um número do pacote de mensagem do protocolo sob teste. A localização da falha (FaultLocation: **header** | **data**) indica se a falha será injetada no cabeçalho ou na área de dados do pacote de mensagem.

A duração da falha (FaultDuration) pode ser temporária, permanente ou intermitente. A falha temporária (temporary (**number**)) é injetada desde o pacote definido em InjectionTime até o pacote definido em **number**. A falha permanente (permanent) é injetada desde o pacote definido por InjectionTime até o fim do experimento). A falha intermitente (intermitent (**number**)) ocorre periodicamente desde InjectionTime a intervalos definidos por **number**.

O nodo (Node: **name**) define qual o nodo que a falha será injetada. O grupo (Group: **name**) define em qual grupo de processos a falha será injetada. A lista de

processos (Process: **process_list**) define os processos onde a falha será injetada. O padrão de falha (FaultPattern: **pattern**) define qual a cadeia de caracteres que será utilizada para realizar a injeção da falha.

Com a criação do arquivo de parâmetros de falhas (*fault_parameters*), o objeto *generator* envia os identificadores no servidor de nomes (objeto *name_server*).

A especificação acima será utilizada nas seções seguintes. A definição da gramática e a implementação da linguagem para esta especificação é um trabalho futuro, os testes atuais definem os parâmetros estaticamente no código.

6.3 Inserção de um objeto *injector* em cada processo do protocolo

Para possibilitar a utilização dos parâmetros de falhas especificados na seção 6.2, o pessoal de teste deve inserir nos processos (A1, B2, C1 e C2) do protocolo objetos *injector* no início da função principal. Exemplo usando o processo A1:

```
main ()
{
        AFIDSipInjector IA1;
        Corpo original da mensagem
}
```

Desta forma, quando o processo A1 for criado na execução normal do protocolo, então o objeto *injector* será instanciado e ficará ativo.

6.4 Encapsulamento das funções de envio e recebimento do protocolo

As funções de envio e recebimento destes processos devem ser encapsuladas pelo pessoal de teste nas funções de envio e recebimento do objeto *injector* inserido. Exemplo usando o processo A1:

```
main ()
{
        AFIDSipInjector A1Injector;
        ...
        A1Injector.send (função de envio original);
        ...
        A1Injector.receive (função de recebimento original);
        ...
}
```

A identificação para o objeto A1Injector irá ser recebida pelo processo A1 quando o construtor de A1Injector enviar o sinal de criação ao objeto *controller*. Com o recebimento do sinal de criação, o objeto *controller* consulta o objeto *nameServer* e percebe que a identificação é IA1 e envia para objeto *injector*. Desta forma a questão da identificação de objetos *injector*, nodos, processos e grupos é resolvida.

6.5 Execução dos experimentos

Detalhes da execução de AFIDS-ip podem ser encontrados na seção 5.2. Primeiramente é realizada a execução do protocolo sem a injeção de falhas e o arquivo *standard_data* é gerado. Para uma melhor escolha dos parâmetros de falhas, pode-se

avaliar o arquivo *standard_data* e então modificar os parâmetros de falhas especificados no arquivo *fault_parameters*. A seguir se procede a execução com a injeção efetiva dos parâmetros de falhas, o que irá gerar o arquivo *readouts_data*.

O estado atual da implementação simplesmente lista as mensagens trocadas entre os processos que possuem os objetos *injector* inseridos. Isto para ambos os arquivos *standard_data* e *readouts_data*. O aperfeiçoamento desta implementação é deixada como trabalho futuro.

6.6 Análise dos dados coletados

A análise automática dos arquivos *standard_data* e *readouts_data* para gerar um arquivo com medidas comparativas (arquivo *measures_data*) é uma tarefa não trivial e também é deixada como sugestão de trabalho futuro.

O estado atual da implementação simplesmente não faz a análise automática. A análise dos arquivos *standard_data* e *readouts_data* é realizada de forma manual pelo pessoal de teste que estiver utilizando a ferramenta.

7 Trabalhos Futuros e Conclusão

Este capítulo apresenta sugestões de trabalhos futuros relacionados a AFIDS e a conclusão deste trabalho.

7.1 Trabalhos Futuros

Trabalhos futuros que podem ser considerados a respeito de AFIDS e da injeção de falhas implementada por software:

- Término da implementação do protótipo de AFIDS-ip: os itens que estão em aberto e que foram mencionados no capítulo 6 serão estudados com maior profundidade e tratados futuramente.
- Validação de Newtop utilizando AFIDS-ip: após a solidificação da implementação de AFIDS-ip a validação de Newtop será possível.
- Aplicação das ferramentas baseadas em AFIDS para validação de uma variedade de protocolos tolerantes a falhas em sistemas distribuídos, por exemplo, protocolos de replicação de dados, recuperação de processos e comunicação em grupo confiável.
- Desenvolvimento de geradores de parâmetros de falhas que se baseiem:
  - no código fonte do protocolo [ECH 91], ou
  - em algum modelo formal do protocolo [ECH 94], ou
  - nas mensagens trocadas durante a execução do protocolo sem injeção de falhas.
- Desenvolvimento de analisadores de experimentos de injeção de falhas.
- Interface gráfica para utilização de AFIDS-ip.
- Implementação de uma ferramenta baseada em AFIDS que utilize apenas um objeto injetor por nodo (AFIDS-in) visando diminuir o *overhead* do objeto *injector* sobre o sistema destino e refinar ainda mais AFIDS.
- Mecanismos necessários para a construção de ferramentas de injeção de falhas que sejam utilizados em sistemas de tempo real: algumas ferramentas como sockPFI e ORCHESTRA se preocupam com questões de tempo real. Esta preocupação deve ser adicionada à AFIDS porque sistemas de tempo real são cada vez mais comuns.
- Obtenção de medidas de desempenho de modo a comparar as várias ferramentas de injeção baseadas em AFIDS.

7.2 Conclusão

A injeção de falhas em sistemas distribuídos é uma opção de complemento a outras técnicas de validação de protocolos em sistemas distribuídos. A sua aplicação em protótipos do protocolo possibilita a detecção e correção de erros de projeto e/ou implementação. Além disso, medidas de dependabilidade como cobertura e latência dos mecanismos do protocolo podem ser obtidas.

A orientação a objetos através dos conceitos de abstração, encapsulamento, modu-laridade e hierarquia possibilita a construção de estruturas de código genéricas e facilmente reutilizáveis, diminuindo o esforço de novos desenvolvimentos e do código final gerado.

Nenhuma ferramenta existente na literatura de injeção de falhas em software para sistemas distribuídos apresenta explicitamente uma abordagem orientada a objetos. De modo a auxiliar a suprir esta lacuna, AFIDS visa fornecer uma biblioteca orientada a objetos para construir ferramentas de injeção de falhas implementada por software em sistemas distribuídos. AFIDS capacita construir rapidamente novas ferramentas com

mecanismos melhorados ou adicionados. Esta característica permite otimizar, passo a passo, a organização e a estrutura de AFIDS, visando minimizar a interferência da ferramenta de injeção de falhas sobre o sistema destino e disponibilizar uma biblioteca de classes estável, simples de utilizar e consistente.

Entre as mais recentes ferramentas para injetar falhas implementadas por software em sistemas distribuídos, somente PFI e ORCHESTRA executam em estações de trabalho rodando UNIX (*SunOS* e *Solaris*). AFIDS é mais uma opção para injetar falhas em protocolos tolerantes a falhas em sistemas operacionais UNIX.

Sendo construída usando C++ e *sockets* sobre Linux, sem o uso de características específicas de Linux, é fácil portar AFIDS para outros sistemas operacionais UNIX compatíveis. Mais ainda, a interferência sobre o sistema destino é reduzida pela separação entre hospedeiro e sistema destino, da mesma forma que as ferramentas de injeção existentes, e por não usar nenhuma aplicação ou sistema adicional para injetar falhas. É possível até mesmo utilizar AFIDS em outras plataformas que não utilizem *sockets*, visto que a forma de comunicação dos componentes de AFIDS é uma opção do projetista da ferramenta.

A implementação de AFIDS-ip mostra que a construção de ferramentas de injeção de falhas é uma tarefa que necessita de muita articulação dos projetistas para a definição de uma arquitetura de ferramenta ideal para os objetivos deles. E tal tarefa está sendo facilitada com o desenvolvimento de AFIDS, que se apresenta em constante evolução.

Particularmente, na UFRGS, alguns trabalhos sobre protocolos tolerantes a falhas em sistemas distribuídos sobre sistemas operacionais UNIX (Linux, *SunOS*) têm sido desenvolvidos. Esses trabalhos envolvem replicação de dados, recuperação de processos e protocolos de difusão. AFIDS será usado para validar estes protocolos. Entretanto, de modo a refinar a arquitetura, o protocolo de comunicação de grupo tolerante a falhas Newtop [EZH 95] será o primeiro a ser validado.

EFA validou protocolos de mascaramento de falhas, concordância e ordenação, PFI validou protocolos de comunicação (TCP e GMP) e DOCTOR validou algoritmos de diagnóstico distribuído. No momento, AFIDS aborda somente o modelo de falhas de comunicação, que é o bastante para validar uma vasta gama de protocolos tolerantes a falhas em sistemas distribuídos. Entretanto, para tornar a arquitetura mais abrangente, seria conveniente adicionar modelos de falhas de processador e memória, de modo a suportar o teste de algoritmos distribuídos, que não podem ser testados somente através do uso do modelo de falhas de comunicação.

ANEXOS

Esta seção de anexos é dividida em quatro partes. O anexo 1 aborda as estruturas de dados que são utilizadas para armazenar os dados dos arquivos de AFIDS. O anexo 2 apresenta os protocolos das classes de AFIDS codificados em C++. O anexo 3 apresenta a modelagem dos componentes principais de AFIDS-ip (ferramenta baseada em AFIDS com 1 objeto *injector* por processo do protocolo sob teste). O anexo 4 apresenta o formato dos pacotes das mensagens enviadas entre os componentes de AFIDS-ip.

Anexo 1 Formato das Estruturas de Dados relacionadas com os Arquivos de AFIDS

Os formatos das estruturas de dados que armazenam as informações dos arquivos utilizados por AFIDS refletem a implementação atual de AFIDS-ip e podem ser modificados de acordo com as necessidades do projetista de novas ferramentas. Estes formatos são referenciados no Anexo 2 e no Anexo 4.

Definição da estrutura relacionada ao arquivo *fault_parameters* (struct Fparam)

Em FParam, o campo injectionTime (número de sequência das mensagens do protocolo sob teste) indica em qual mensagem será aplicada a injeção. O campo faultLocation indica se a falha será aplicada no cabeçalho ou na área de dados da mensagem. O campo faultDuration (quantidade de pacotes de mensagens do protocolo sob teste) indica a duração da aplicação das falhas. O campo faultType indica qual o tipo de falha de comunicação a ser injetada: 0 - omissão de recebimento, 1 - omissão de envio, 2 - valor, 3 - queda de canal de comunicação, 4 - queda de processo, 5 - queda de nodo e 6 - temporização. O campo patternSize indica o tamanho do campo faultPattern. O campo patternStart indica a partir de qual posição o padrão de falhas (campo faultPattern) será instalado. O campo faultPattern é uma cadeia de caracteres que irá ser usada para modificar o cabeçalho ou os dados das mensagens. O formato em C++ é:

```
struct FParam
{
    long injectionTime;
    char faultLocation;
    int faultDuration;
    long faultType;
    int patternSize;
    int patternStart;
    char *faultPattern;
};
```

Definição do item do arquivo *name_base* (struct Nbase)

Em Nbase, o campos injectorId, injectorAddress, nodesNumber, nodesList, groupsNumber, groupsList, processNumber e processList, armazenam respectivamente, uma cadeia de caracteres que identifica o objeto *injector*, o endereço IP do nodo, a

quantidade de nodos, a lista de nodos, a quantidade de grupos, a lista dos grupos, a quantidade de processos e a lista de processos.

```
struct Nbase{
        char injectorId[MAXLABELSIZE];
        char injectorAddress[MAXLABELSIZE];
        int nodesNumber;
        char **nodesList;
        int groupsNumber;
        char **groupsList;
        int processNumber;
        char **processList;
};
```

Definição do item do arquivo *standard_data* (struct Standard)

Em Standard, o campo messageType é inicializado com 4 para identificar a mensagem como uma mensagem normal do protocolo sob teste, ou com 8 para identificar o valor de alguma variável. O campo ilabel identifica o objeto *injector* que enviou a mensagem. O campo contentsSize indica o tamanho da mensagem normal do protocolo se messageType for 4, ou indica o tamanho do valor da variável se messageType for 8. O campo variableName armazena o nome da variável. O campo messageContents contém o conteúdo da mensagem normal do protocolo ou o valor da variável (transformada em uma *string*). O formato em C++ é:

```
struct Standard
{
    char messageType;
    char ilabel[MAXLABELSIZE];
    int contentsSize;
    char variableName[MAXLABELSIZE];
    char *messageContents;
};
```

Definição do item do arquivo *readouts_data* (struct Readouts)

No estado de implementação atual de AFIDS-ip, o arquivo *readouts_data* utiliza a mesma estrutura de dados que *standard_data*. A estrutura Readouts é definida em C++ da seguinte forma: Standard Readouts;

Definição do item do arquivo *measures_data* (struct Measures)

A estrutura Measures não está definida e é deixada como trabalho futuro, conforme foi explicado na seção 6.6.

Definição do item do arquivo *faults_base* (struct Fbase)

A estrutura do arquivo *faults_base* utiliza a estrutura Fparam definida neste anexo. É definida em C++ como: Fparam FBase;

Anexo 2 Os Protocolos das Classes de AFIDS

Neste anexo são apresentados os protocolos das classes pertecentes a cada uma das categorias descritas na seção 4.4. Os protocolos estão descritos em C++.

Todas as classes definidas nesta seção são chamadas classes abstratas ou classes base, e não podem ser instanciadas diretamente para produzir objetos. Para que elas possam ser utilizadas, classes filhas devem ser criadas, tendo como pai a classe abstrata desejada. Essas classes filhas irão possuir realmente a implementação das funções membros definidas na classe abstrata (classe pai). Todas as funções membros que possuem a palavra chave *virtual* são ditas virtuais e devem por isso ser definidas pelo projetista da ferramenta.

Todas as funções membros que possuem o mesmo nome dado à classe são ditas construtores e são chamadas automaticamente quando a classe for instanciada. Ex: na classe AFIDSManager, o construtor é a função membro AFIDSManager (void).

Todas as funções membros que possuem o mesmo nome dado à classe e iniciam com o til (~) são ditas destrutores e são chamadas automaticamente quando o objeto instaciado for destruído. Ex: na classe AFIDSManager, o destrutor é a função membro ~AFIDSManager (void).

Toda classe que possuir a palavra chave *template* é uma classe parametrizável, ou seja tem uma outra classe como parâmetro. Ex: a classe AFIDSGenerator possui como parâmetro uma classe FParamMan. Em uma implementação usando a classe AFIDSGenerator, o parâmetro que poderia ser usado no lugar da classe FParamMan poderia ser uma classe FaultParameters, que seria uma classe que teria funções membro e atributos específicos de um tipo particular de arquivo de parâmetros de falhas. Ex: main(void) { AFIDSGenerator <FaultParameters> generator; }. Isto permite, usando este exemplo, que tipos diferentes de arquivos de parâmetros de falhas possam ser definidos através de classes diferentes, possibilitando flexibilidade na criação e manipulação de arquivos de parâmetros de falhas.

Protocolo da Classe AFIDSManager

A classe AFIDSManager é responsável pela gerência dos experimentos de injeção de falhas. A função membro chooseOption é responsável pela escolha da opção de gerência do experimento (geração do arquivo de parâmetros de falhas, execução do protocolo tolerante a falhas sem a injeção de falhas, execução do protocolo tolerante a falhas com a injeção de falhas e análise dos dados resultantes dos experimentos de injeção). A função membro managerRun é a função principal responsável pela gerência.

```
class AFIDSManager
{
public:
        AFIDSManager (void);
        virtual ~AFIDSManager (void);
        virtual void chooseOption (void);
        virtual void managerRun (void);
};
```

Protocolo da Classe AFIDSGenerator

A classe AFIDSGenerator é responsável pela geração do arquivo de parâmetros de falhas. A função membro generatorRun é a implementação de algum método de geração de parâmetros de falhas. O atributo fault_parameters é responsável pela manipulação do arquivo de parâmetros de falhas.

```
template <class FParamMan>
class AFIDSGenerator
{
protected:
        FParamMan fault_parameters;
public:
        AFIDSGenerator (void);
        virtual ~AFIDSGenerator (void);
        virtual void generatorRun (void);
};
```

Protocolo da Classe AFIDSNameServer

A classe AFIDSNameServer é o servidor de nomes de AFIDS. As funções membros Bind, LookupId, LookupAttributes e Unbind correspondem às operações definidas na seção 4.2.3. O atributo name_base é responsável pela manipulação do arquivo do servidor de nomes. A função membro nameServerRun é responsável pela execução do servidor de nomes.

```
template <class NServerMan>
class AFIDSNameServer
{
protected:
        NServerMan name_base;
public:
        AFIDSNameServer (void);
        virtual ~AFIDSNameServer (void);
        void nameServerRun (void);
        void Bind (char *InjectorId, int InjectorAddress, int *NodesList,
        char **GroupsList, char **ProcessList);
        void LookupId (char *InjectorId);
        void LookupAttributes (int InjectorAddress, int *NodesList,
        char **GroupsList, char **ProcessList);
        void Unbind (char *InjectorId);
};
```

Protocolo da Classe AFIDSController

A classe AFIDSController é responsável pela percepção da criação de novos objetos injetores e pelo fornecimento dos parâmetros de falhas a cada um dos objetos injetores. A função membro controllerHandler é responsável pela comunicação com os objetos *injector*. A função membro controllerRun é a função principal da classe. O atributo fault_parameters é responsável pela manipulação do arquivo de parâmetros de

falhas. A função membro sendFaultParameters envia os parâmetros das falhas correspondentes para o objeto *injector* que enviou o sinal de criação ao *controller*.

```
template <class FParamMan>
class AFIDSController
{
protected:
        FParamMan fault_parameters;
public:
        AFIDSController (void);
        virtual ~AFIDSController (void);
        virtual void sendFaultParameters (void);
        virtual int controllerHandler (void);
        virtual void controllerRun (void);
};
```

Protocolo da Classe AFIDSCollector

A classe AFIDSCollector é responsável pela coleta dos dados enviados pelos objetos injetores durante os experimentos de injeção de falhas. Os atributos standard_data e readouts_data são, respectivamente, responsáveis pela manipulação do arquivo de resultados da execução sem a injeção de falhas e pela manipulação do arquivo de resultados da execução com a injeção de falhas. A função membro collectorHandler manipula a comunicação com os objetos *injector*. A função membro collectorRun é responsável pela execução do *collector* e pela geração dos arquivos *standard_data* e *readouts_data*.

```
template <class StandardMan, class ReadoutsMan >
class AFIDSCollector
{
protected:
        StandardMan standard_data;
        ReadoutsMan readouts_data;
public:
        AFIDSCollector (void);
        virtual ~AFIDSCollector (void);
        virtual int collectorHandler (void);
        virtual void collectorRun (void);
};
```

Protocolo da Classe AFIDSInjector

A classe AFIDSInjector é responsável pela injeção das falhas. O atributo faults_base é responsável pela gerência do arquivo de base de falhas (*faults_base*). A função membro receiveFaultParameters recebe os parâmetros de falhas do objeto *controller* e grava no arquivo *faults_base*. A função membro sendSignal envia os sinais de criação e destruição do objeto *injector*. As funções membro send e receive encapsulam, respectivamente, as funções de envio e recebimento originais do protocolo sendo testado. A função membro injectorRun é a função principal da classe.

```
template <class FBaseMan>
class AFIDSInjector
{
protected:
        FBaseMan faults_base;
public:
        AFIDSInjector (void);
        virtual ~AFIDSInjector (void);
        virtual void receiveFaultParameters (void);
        virtual void sendSignal (char type);
        virtual void send (void);
        virtual void receive (void);
        virtual injectorRun (void);
};
```

Protocolo da Classe AFIDSAnalyser

A classe AFIDSAnalyser é responsável pela análise dos arquivos resultantes dos experimentos de injeção de falhas. Os atributos standard_data, readouts_data e measures_data são responsáveis respectivamente, pelos arquivos *standard_data*, *readouts_data* e *measures_data*. A função membro analyserRun é a responsável pela análise dos arquivos *standard_data* e *readouts_data* e pela geração do arquivo *measures_data*.

```
template <class StandardMan, class ReadoutsMan, class MeasuresMan>
class AFIDSAnalyser
{
protected:
        StandardMan standard_data;
        ReadoutsMan readouts_data;
        MeasuresMan measures_data;
public:
        AFIDSAnalyser (void);
        virtual ~AFIDSAnalyser (void);
        virtual void analyserRun (void);
};
```

Protocolos das outras Classes

A classe AFIDSFileMan é uma classe parametrizável que é base para todas as classes manipuladoras de arquivos em AFIDS (utilizando estruturas de dados que definem os arquivos - FParam, Nbase, Standard, Readouts, Measures e FBase, que estão definidos no Anexo 1):

- AFIDSFParamMan: manipula o arquivo *fault_parameters*. É instanciado da seguinte forma: AFIDSFileMan<FParam> AFIDSFParamMan;
- AFIDSNServerMan: manipula o arquivo *name_base*. É instanciado da seguinte forma: AFIDSFileMan<Nbase > AFIDSNServerMan;
- AFIDSStandardMan: manipula o arquivo *standard_data*. É instanciado da seguinte forma: AFIDSFileMan<Standard> AFIDSStandardMan;

- AFIDSReadoutsMan: manipula o arquivo *readouts_data*. É instanciado da seguinte forma: AFIDSFileMan<Readouts> AFIDSReadoutsMan;
- AFIDSMeasuresMan: manipula o arquivo *measures_data*. É instanciado da seguinte forma: AFIDSFileMan<Measures> AFIDSMeasuresMan;
- AFIDSFBaseMan: manipula o arquivo de *faults_base*. É instanciado da seguinte forma: AFIDSFileMan<Fbase> AFIDSFBaseMan;

Em AFIDSFileMan os atributos são currentItem, fileName e FileItem, que são respectivamente, o número do item corrente dentro do arquivo, o nome do arquivo e uma referência à estrutura de dados referente ao arquivo a ser manipulado. As funções membro são openFile, closeFile, seekFile, writeFile e readFile, que são responsáveis respectivamente pela, abertura do arquivo, fechamento do arquivo, posicionamento dentro do arquivo, escrita no arquivo e leitura do arquivo.

```
template <class FileItem>
class AFIDSFileMan
{
protected:
        long currentItem;
        char *fileName;
        FileItem& object;

public:
        AFIDSFileMan (void);
        virtual ~AFIDSFileMan (void);
        void openFile (char *file);
        void closeFile (void);
        virtual int seekFile (int objectNumber);
        virtual int writeFile (FileItem *object);
        virtual FileItem& readFile (void);
};
```

Anexo 3 Modelagem de AFIDS-ip usando BOOCH

Este anexo apresenta os modelos estáticos lógico (diagramas de classes e de objetos) e físico (modelos de módulos e de processos). Além disso são descritos os modelos dinâmicos (diagramas de estados e de interação). São esboçados apenas a modelagem dos componentes principais de AFIDS-ip. A notação utilizada é aquela do método de Booch [BOO94].

Diagrama de Classes

Mostra o relacionamentos entre as classes que compõe a visão lógica do sistema. Devido à impossibilidade de mostrar o diagrama completo, com todos os relacionamentos entre as classes, os vários diagramas de classe esboçados a seguir enfocam uma classe em particular.

A notação utilizada é ilustrada na figura A3.1.

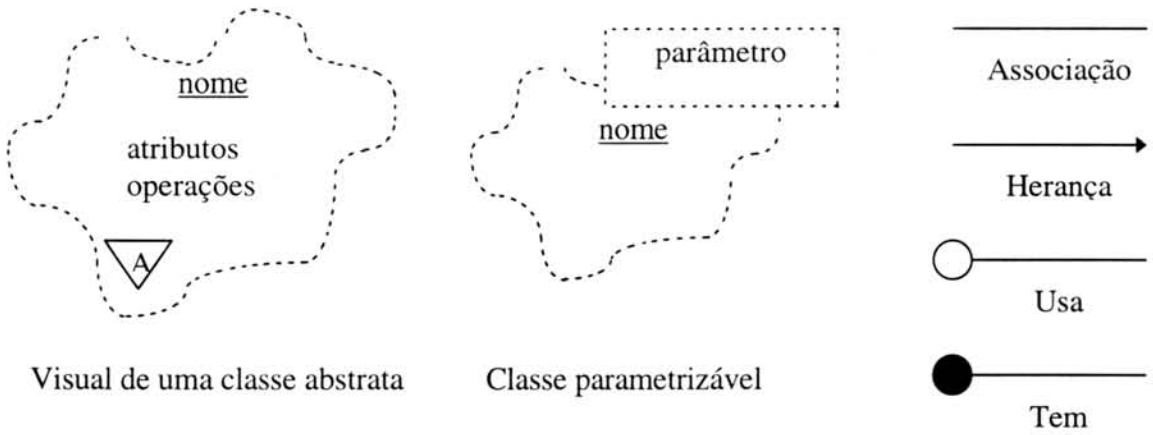


FIGURA A3.1 - Notação utilizada em Diagramas de Classes

Diagrama de Classes tendo como foco principal a classe ipManager

O diagrama da figura A3.2 mostra apenas as relações entre a classe ipManager e as outras classes.

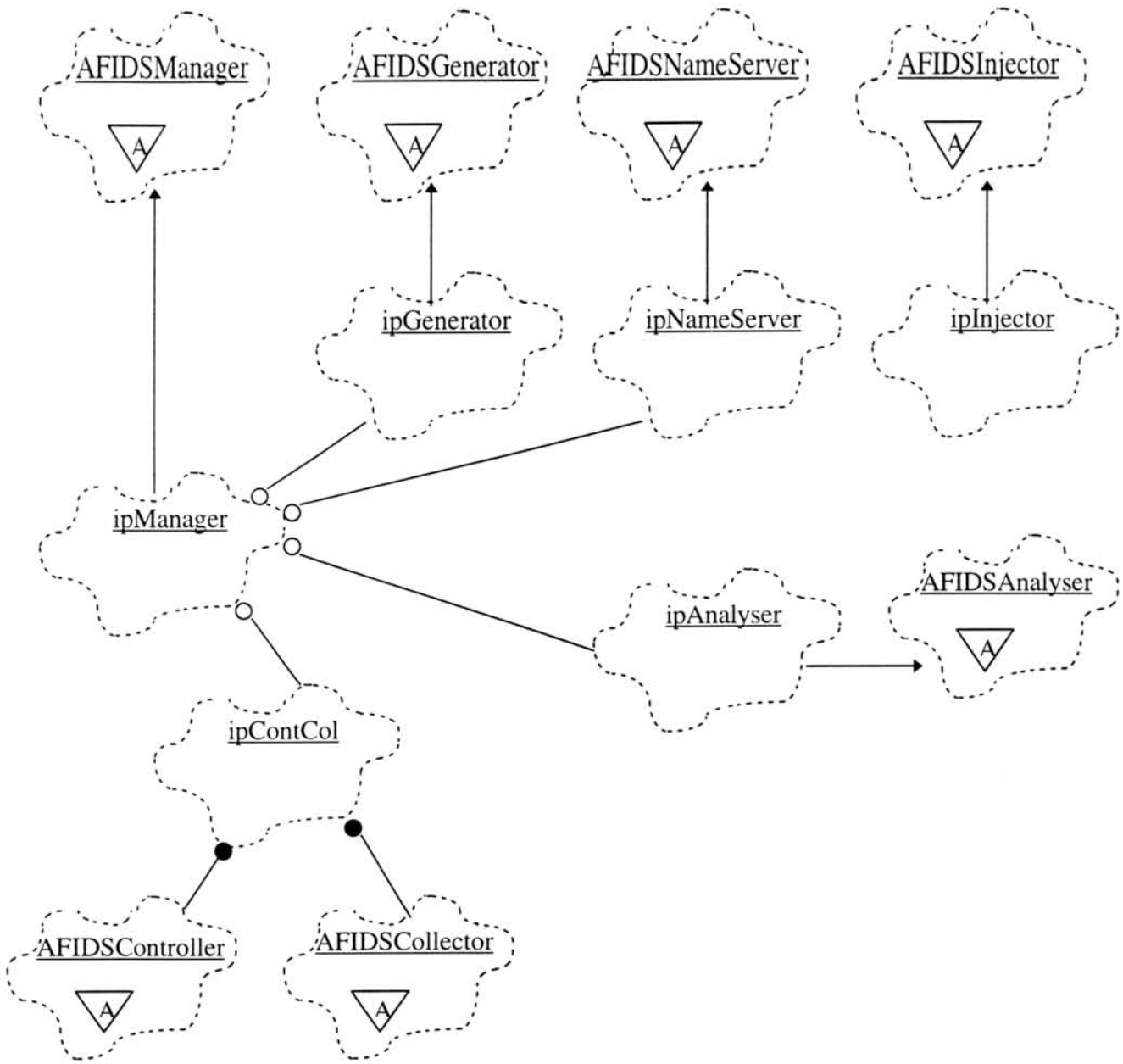


FIGURA A3.2 - Diagrama de Classes com Enfoque na Classe ipManager

Diagrama de Classes tendo como foco principal a classe AFIDSFileMan

O diagrama da figura A3.3 mostra apenas as relações entre a classe AFIDSFileMan e as outras classes. Todas as classes instanciadas a partir de AFIDSFileMan manipulam os arquivos de AFIDS-ip. Os nomes dados foram os mesmos para as classes definidas na seção 4.4. Detalhes adicionais podem ser encontrados no Anexo 2.

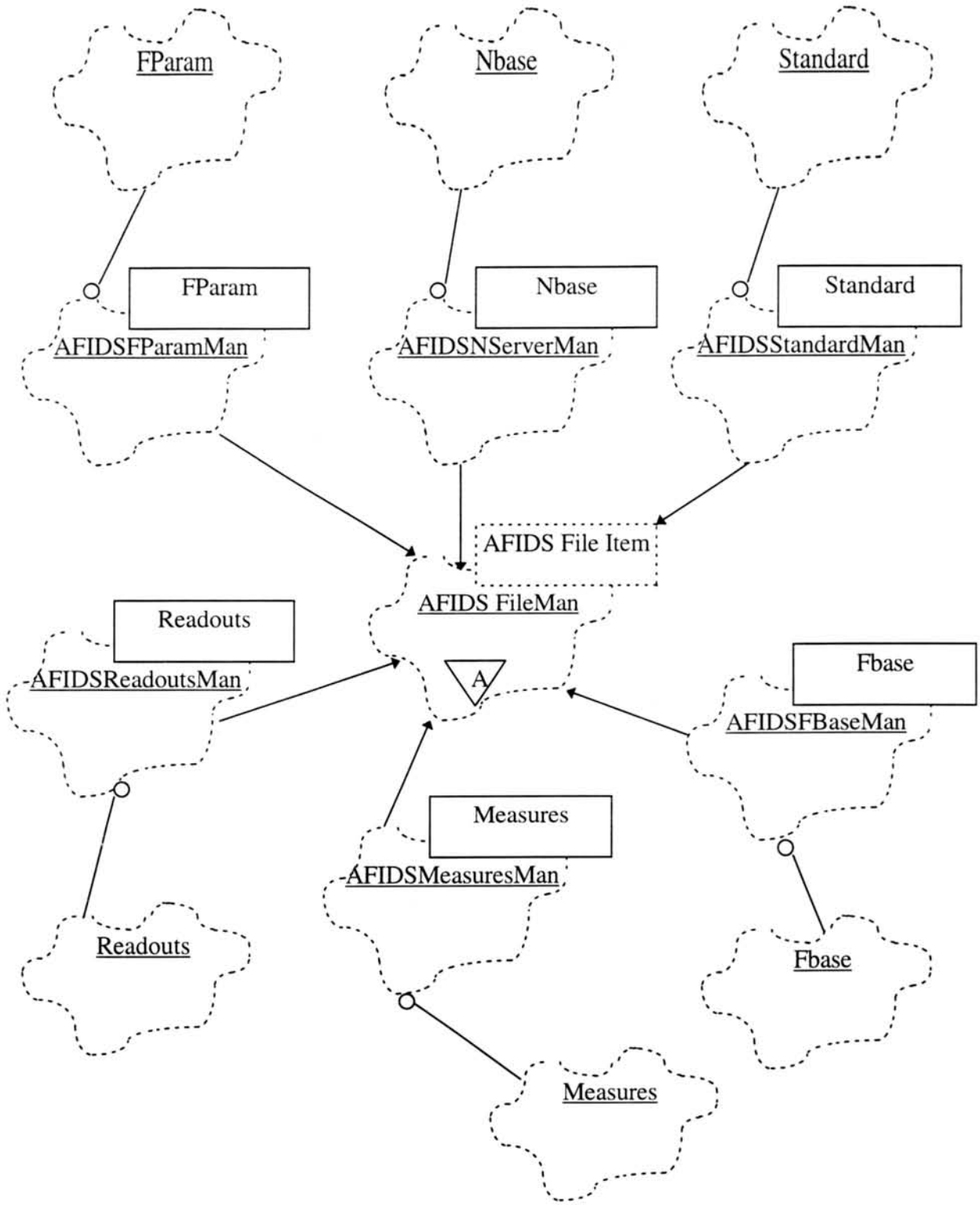


FIGURA A3.3 - Diagrama de Classes com Enfoque na Classe AFIDSFileMan

Diagrama de Classes enfocando a Classe AFIDSFParamMan

O diagrama da figura A3.4 mostra apenas as relações entre a classe AFIDSFParamMan e outras classes.

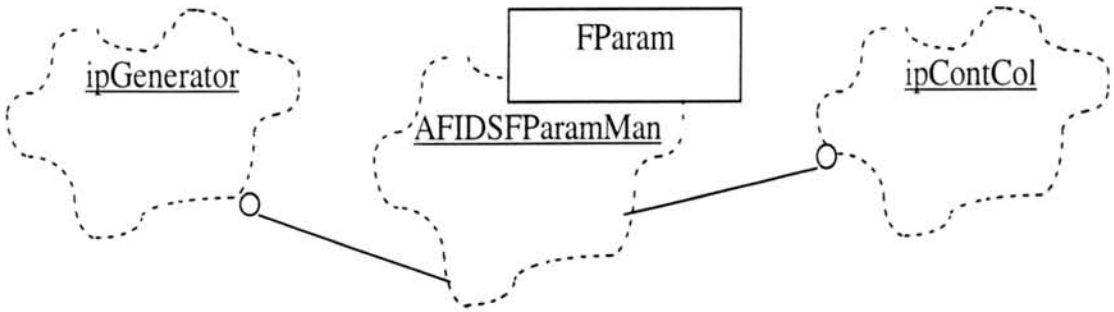


FIGURA A3.4 - Diagrama de Classes com Enfoque na Classe AFIDSFParamMan

Diagrama de Classes enfocando a Classe AFIDSNServerMan

O diagrama da figura A3.5 mostra apenas as relações entre a classe AFIDSNServerMan e outras classes.

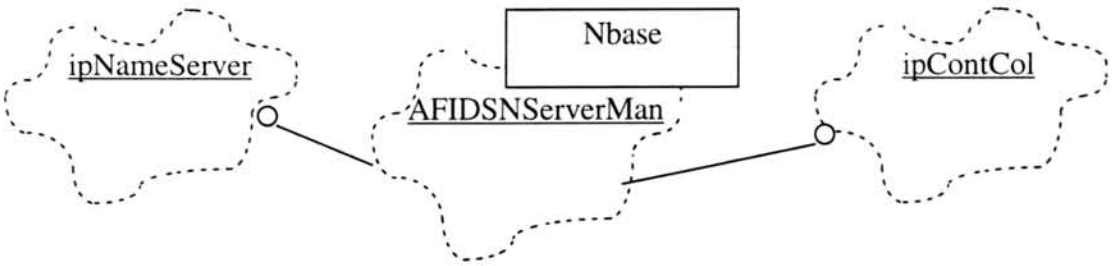


FIGURA A3.5 - Diagrama de Classes com Enfoque na Classe AFIDSNServerMan

Diagrama de Classes enfocando a Classe AFIDSStandardMan

O diagrama da figura A3.6 mostra apenas as relações entre a classe AFIDSStandardMan e outras classes.

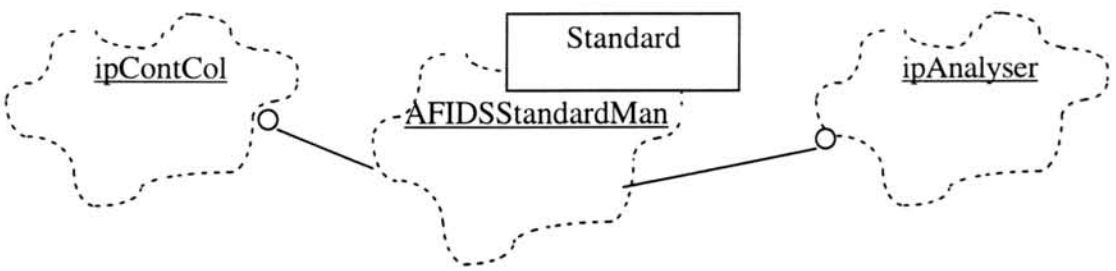


FIGURA A3.6 - Diagrama de Classes com Enfoque na Classe AFIDSStandardMan

Diagrama de Classes enfocando a Classe AFIDSReadoutsMan

O diagrama da figura A3.7 mostra apenas as relações entre a classe AFIDSStandardMan e outras classes.

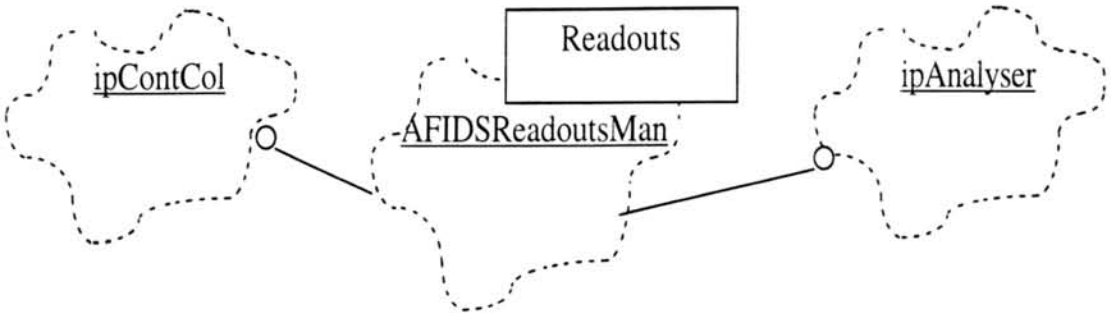


FIGURA A3.7 - Diagrama de Classes com Enfoque na Classe AFIDSReadoutsMan

Diagrama de Classes enfocando a Classe AFIDSMeasuresMan

O diagrama da figura A3.8 mostra apenas as relações entre a classe AFIDSMeasuresMan e outras classes.

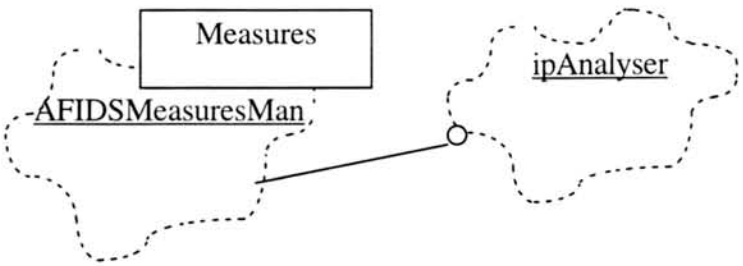


FIGURA A3.8 - Diagrama de Classes com Enfoque na Classe AFIDSMeasuresMan

Diagrama de Classes enfocando a Classe AFIDSFBaseMan

O diagrama da figura A3.9 mostra apenas as relações entre a classe AFIDSFBaseMan e outras classes.

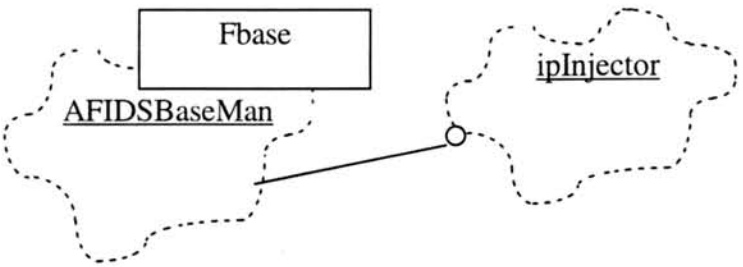


FIGURA A3.9 - Diagrama de Classes com Enfoque na Classe AFIDSFBaseMan

Diagrama de Objetos

Mostra o relacionamento entre os objetos no projeto lógico do sistema. Os diagramas de objetos esboçados a seguir apresentam o cenário de execução discutido na seção 5.3.

Um objeto é representado como mesmo ícone utilizado no diagrama de classes, a diferença é que a linha que define o ícone é cheia em vez de tracejada. Uma das formas de nomear um objeto é A:C, onde A é o nome do objeto e C é o nome da classe. A invocação de métodos de um objeto é indicada por uma linha que une os dois objetos. As setas colocadas junto à linha que une os objetos indica o sentido da invocação de métodos (objeto invocador (objeto invocado). Junto às setas há um número de sequência de invocação e o método que está sendo invocado.

Diagrama de Objetos na Criação de Objetos *Injector*

A figura A3.10 ilustra a sequência de invocação dos métodos entre os objetos na criação de objetos *injector*:

1 O objeto Inj recém-criado envia uma mensagem (contendo informações como nodo e processo) ao objeto ipCC indicando uma criação de objeto *injector*. O método creationRequest do objeto ipCC manipulará esta mensagem.

2 O objeto ipCC envia uma mensagem ao servidor de nomes (NS) requisitando a identificação do objeto Inj recém-criado. O método lookupAttributes() do objeto NS manipulará esta mensagem.

3 O objeto NS envia uma mensagem ao objeto ipCC com os parâmetros de falhas referentes ao objeto Inj recém-criado e a identificação do objeto Inj. O método parametersAnswer() do objeto ipCC manipulará esta mensagem.

O objeto ipCC envia uma mensagem ao objeto Inj com os parâmetros de falhas e o identificador do objeto Inj. O método creationAnswer() do objeto Inj manipulará esta mensagem.

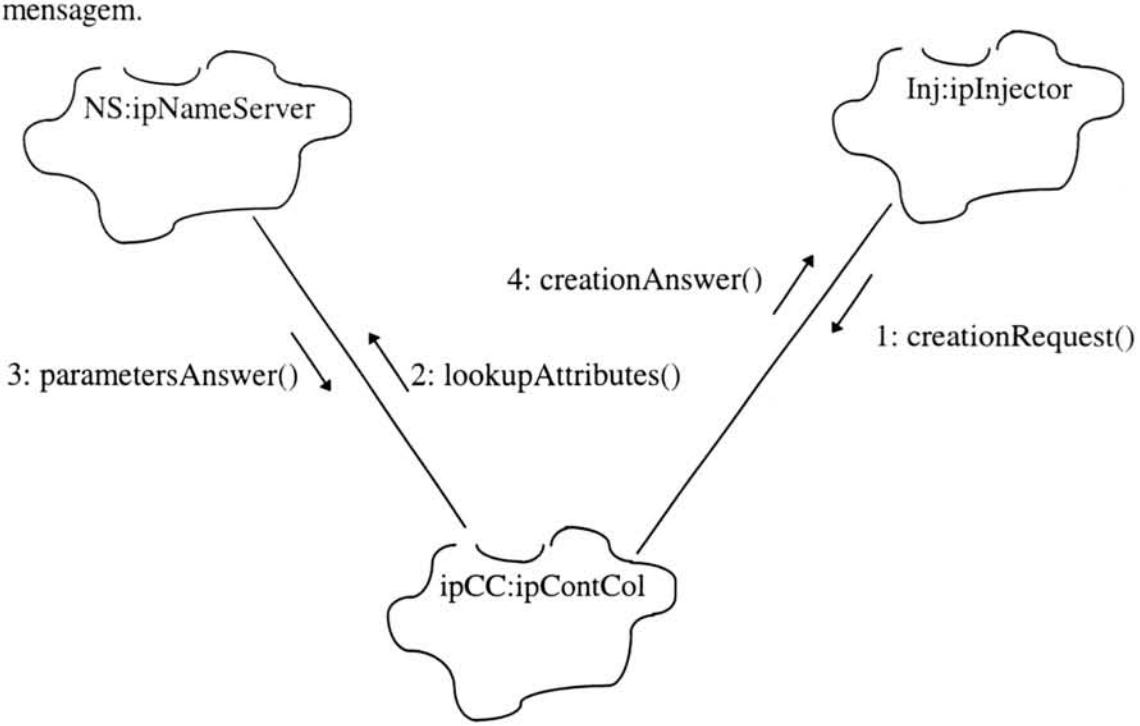


FIGURA A3.10 - Diagrama de Objetos na Criação de Objetos Injector

Diagrama de Objetos na Coleta de Dados do Experimento

A figura A3.11 mostra quando o objeto Inj envia uma mensagem (contendo os dados a serem coletados) ao objeto ipCC. O método collectData do objeto ipCC manipulará esta mensagem.

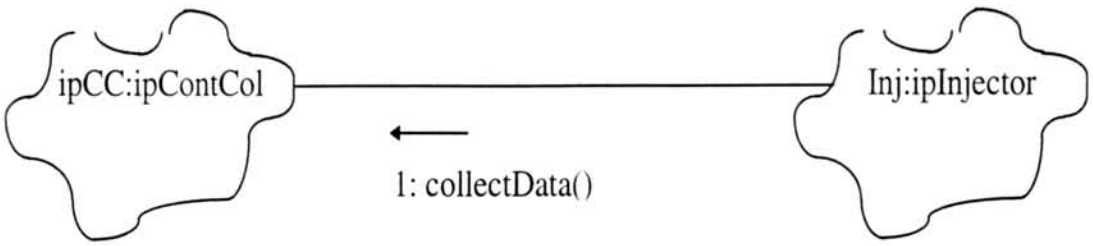


FIGURA A3.11 - Diagrama de Objetos na Coleta de Dados do Experimento

Diagrama de Objetos na Destruição de Objetos *Injector*

A figura A3.12 mostra quando o destrutor do objeto Inj envia uma mensagem (contendo o identificador de Inj) ao objeto ipCC indicando a sua destruição. O método destructionRequest do objeto ipCC manipulará esta mensagem.

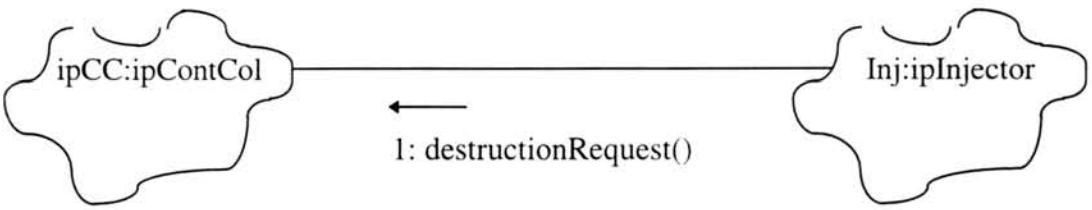


FIGURA A3.12 - Diagrama de Objetos na Destruição de Objetos *Injector*

Diagrama de Interação

É usado para traçar a execução de um cenário no mesmo contexto que o diagrama de objetos e é uma forma alternativa de representar um diagrama de objetos.

Um diagrama de interação aparece na forma tabular. As entidades de interesse são escritas horizontalmente no topo do diagrama. Uma linha vertical tracejada é desenhada abaixo de cada objeto. Mensagens (que podem denotar eventos ou a invocação de operações) são mostradas horizontalmente usando a mesma sintaxe dos diagramas de objetos. Os pontos finais de ícones de mensagens se conectam com as linhas verticais tracejadas, que conectam com as entidades no topo do diagrama e são desenhados do cliente até o servidor. Ordenamento é indicado pela posição vertical, com a primeira mensagem mostrada no topo do diagrama, e a última mensagem mostrada no fundo.

Diagrama de Interação na Criação de Objetos *Injector*

A figura A3.13 mostra a interação dos objetos na criação de objetos *injector*.

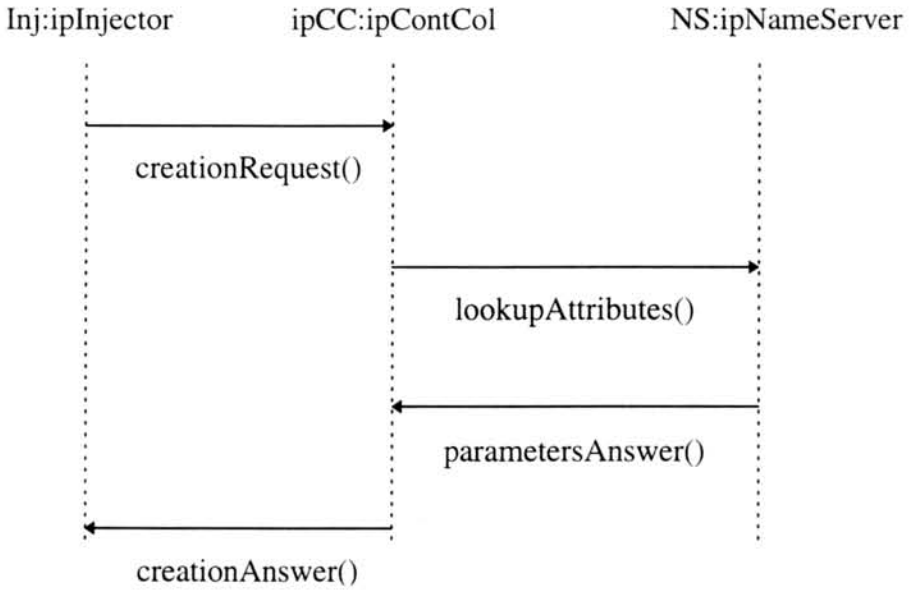


FIGURA A3.13 - Diagrama de Interação na Criação de Objetos *Injector*

Diagrama de Interação na Coleta de Dados do Experimento

A figura A3.14 mostra a interação dos objetos na coleta de dados do experimento.

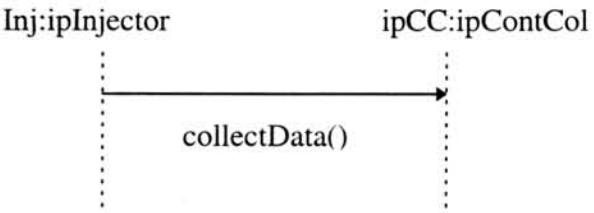


FIGURA A3.14 - Diagrama de Interação na Coleta de Dados do Experimento

Diagrama de Interação na Destruição de Objetos *Injector*

A figura A3.15 mostra a interação dos objetos na destruição de objetos *injector*.

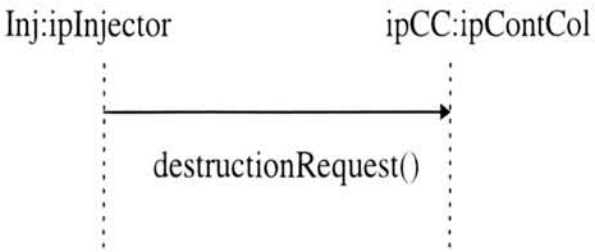


FIGURA A3.15 - Diagrama de Interação na Destruição de Objetos *Injector*

Diagrama de Estados

É usado para mostrar o espaço de estados de uma dada classe, os eventos que causam uma transição de um estado para o outro, e as ações que resultam de uma mudança de estado. Nem toda classe possui um comportamento ordenado de eventos, e por isso deve-se utilizar diagramas de transição de estados somente para aquelas classes que exibem tal comportamento. O diagrama de estados do método Booch utiliza a notação usada por Harel.

Diagrama de Estados da Classe ipContCol

A figura A3.16 mostra o diagrama de estados da classe ipContCol.

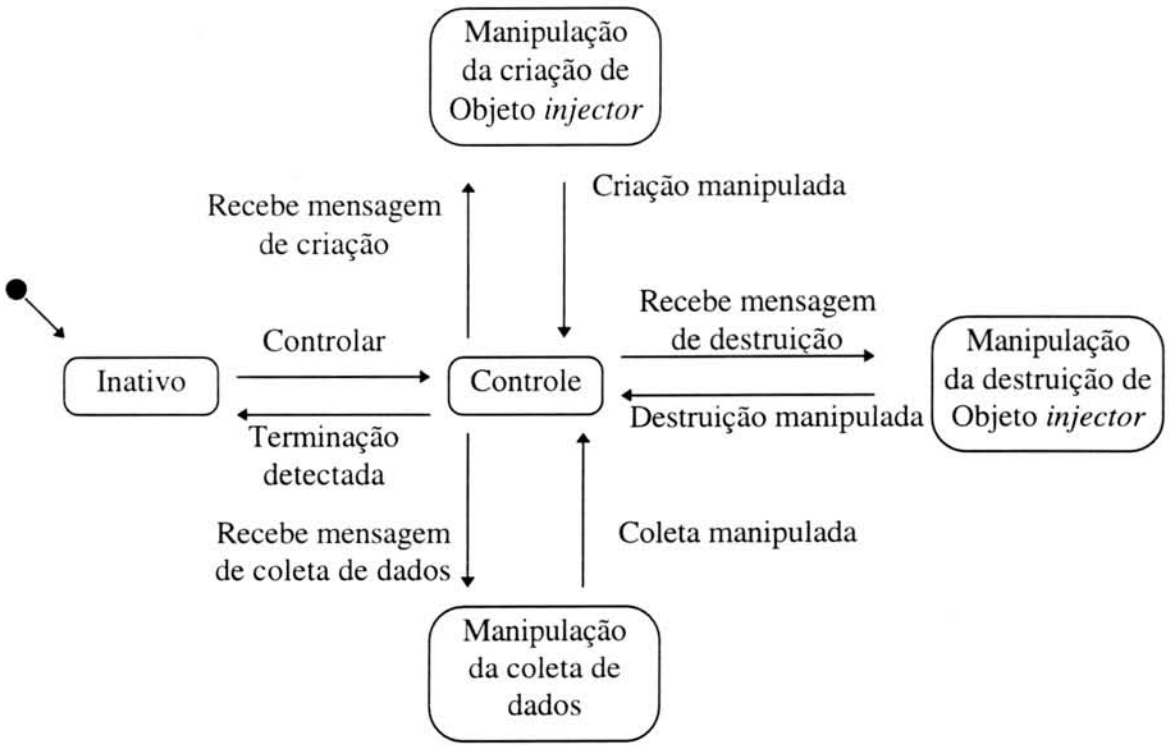


FIGURA A3.16 - Diagrama de Estados da Classe ipContCol

Diagrama de Estados da Classe ipInjector

A figura A3.17 mostra o diagrama de estados da classe ipInjector.

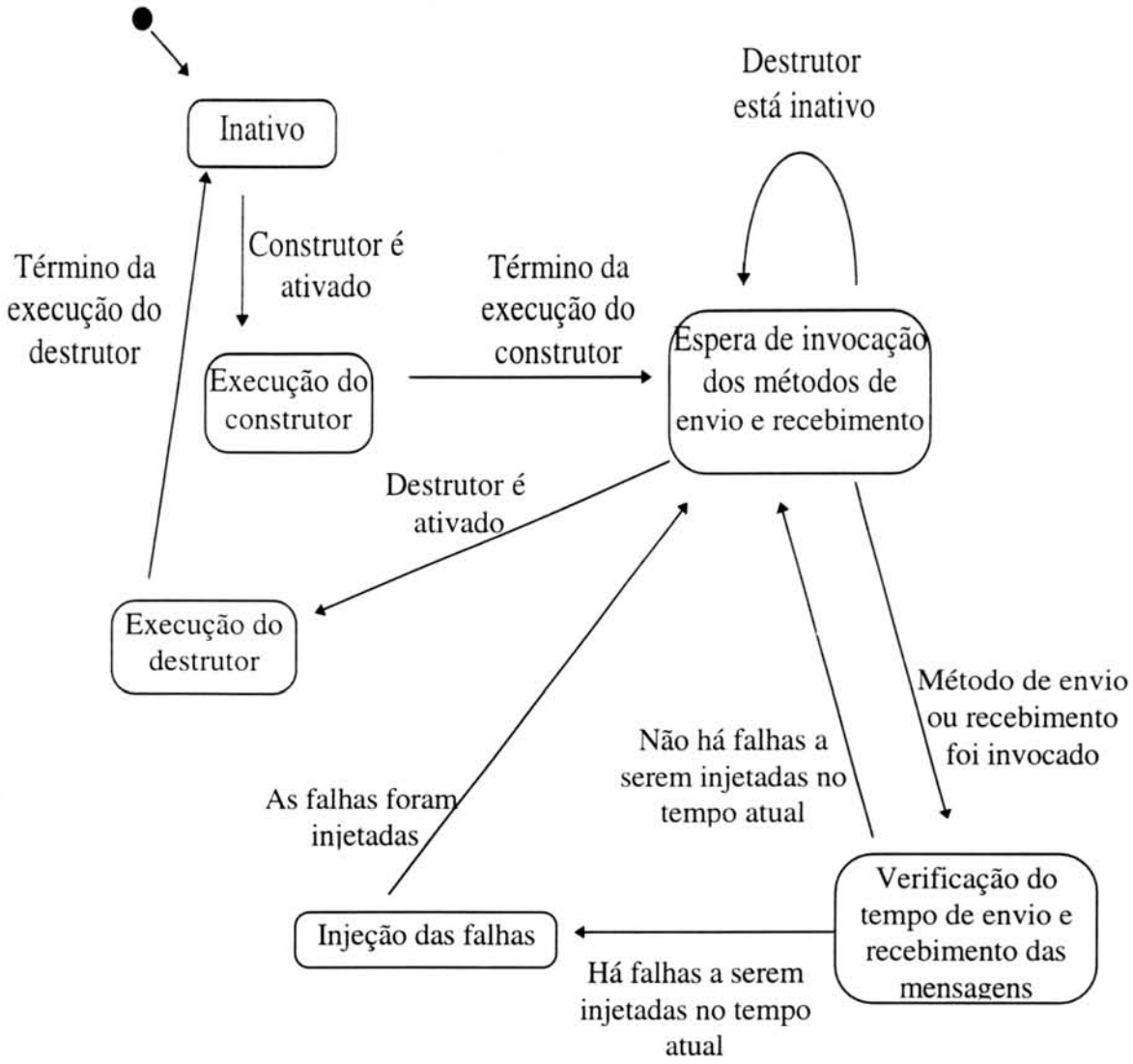


FIGURA A3.17 - Diagrama de Estados da Classe ipInjector

Diagrama de Módulos

É utilizada para mostrar a alocação de classes e objetos para módulos na visão física do sistema. A figura A3.18 apresenta a arquitetura física de AFIDS-ip com apenas o diagrama de módulo do nível mais alto mapeando a arquitetura lógica esboçada na figura 4.9. Cada retângulo da figura corresponde a um subsistema contendo vários módulos:

- O subsistema ipHost corresponde aos módulos que estão restritos ao hospedeiro:
  - * Manager: contém a classe ipManager.
  - * Generator: contém a classe ipGenerator.
  - * ContCol: contém a classe ipContCol.
  - * NServer: contém a classe ipNameServer.
  - * Analyser: contém a classe ipAnalyser.
- O subsistema ipTargetSystem corresponde ao módulo que está restrito ao sistema destino: Injector, que contém a classe ipInjector.
- O subsistema AFIDSFileManager corresponde aos módulos que são responsáveis pela manipulação dos arquivos de AFIDS-ip:
  - * FaultPM: contém a classe AFIDSFParamMan.
  - * StdM: contém a classe AFIDSStandardMan.
  - * ReadM: contém a classe AFIDSReadoutsMan.
  - * MeasM: contém a classe AFIDSMeasuresMan.
  - * FaultBM: contém a classe AFIDSFBaseMan.
  - * NameSM: contém a classe AFIDSNServerMan.
- O subsistema AFIDSFileItem corresponde aos módulos que possuem as estruturas de dados utilizadas para armazenar os dados dos arquivos:
  - * Fparam: contém a struct Fparam.
  - * Nbase: contém a struct Nbase.
  - * Standard: contém a struct Standard.
  - * Readouts: contém a struct Readouts, que no estado de implementação atual é igual à struct Standard.
  - * Measures: no estado atual de implementação não está definido.
  - * Fbase: contém a struct Fbase, que é idêntica a FParam.
- O subsistema ipComm corresponde aos módulos que implementam a comunicação entre os objetos de AFIDS-ip através de *sockets*:
  - * Sockets: contém a classe Sockets. Utiliza as bibliotecas-padrão de *sockets*.
  - * ISTCP: contém a classe TCP, que implementa a comunicação via TCP. É a implementação utilizada em AFIDS-ip porque permite um grau de confiabilidade melhor do que a implementação que utiliza UDP.
  - * ISUDP: contém a classe UDP.

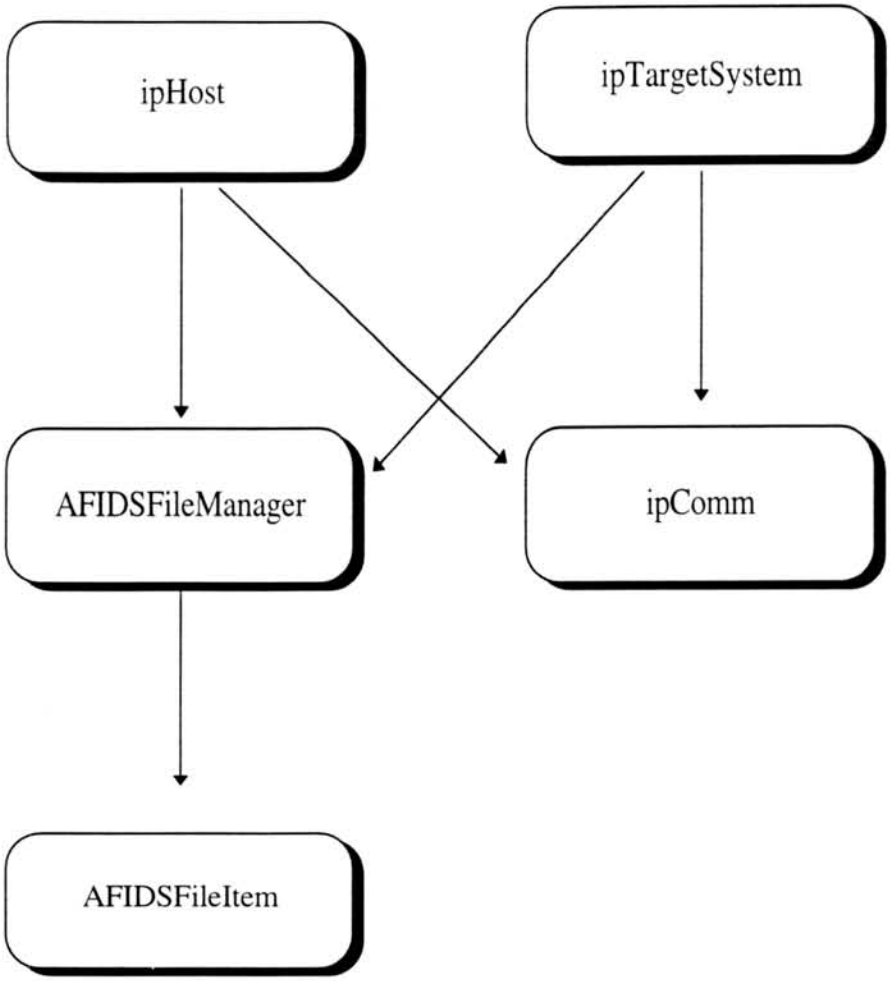


FIGURA A3.18 - Diagrama de Módulos com os subsistemas de AFIDS-ip

Diagrama de Processos

Mostra a alocação de processos a processadores no projeto físico de um sistema. A figura A3.19 mostra o diagrama de processos de AFIDS-ip. A estação de trabalho hospedeira (*host workstation*) interage com as N estações de trabalho do sistema destino (*target workstations*). As N estações no sistema destino também interagem entre si. A seguir é ilustrado quais os processos que existem em cada um dos processadores, no estado de implementação atual de AFIDS-ip:

- *host workstation*: possui o processo Manager, que engloba as classes definidas no subsistema ipHost do Diagrama de Módulos.
- *target workstation*: possui o objeto *injector* encapsulado em cada processo do protocolo que está sendo testado.

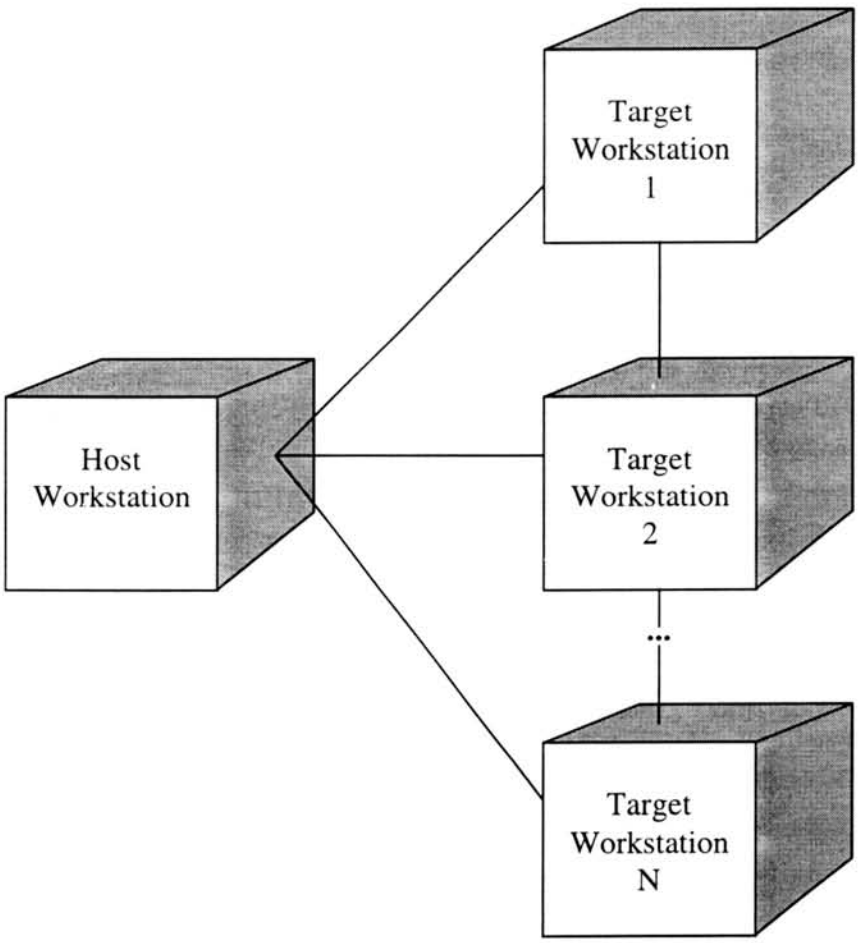


FIGURA A3.19 - Diagrama de Processos de AFIDS-ip

Anexo 4 Formato dos Pacotes de Mensagens entre os Componentes de AFIDS-ip

Este anexo apresenta o formato dos pacotes das mensagens que são trocadas entre os componentes de AFIDS-ip. Todos os formatos de pacotes de mensagens definidos a seguir utilizam a estrutura header. A constante MAXLABELSIZE indica o tamanho máximo dos identificadores utilizados. O campo targetprocess indica qual o processo que receberá a mensagem. O campo sourceprocess indica o processo que envia a mensagem. O campo label é um número de sequência da mensagem. O formato em C++ é:

```
struct header
{
        char targetprocess[MAXLABELSIZE];
        char sourceprocess[MAXLABELSIZE];
        long label;
};
```

Formato dos Pacotes na Comunicação entre os Objetos *injector* e o *controller-collector*

a) Do objeto *injector* para o controller-collector na criação de *injector*:

O campo messageType é inicializado com 0 para identificar a mensagem como um sinal de criação do objeto *injector*. O campo nodo identifica qual o nodo que se encontra o processo que instanciou o objeto *injector*. O campo plabel identifica o processo que instanciou o objeto *injector*. O formato em C++ é:

```
struct InjectorCreationMessage
{
    header h;
    char messageType;
    char nodo[MAXLABELSIZE];
    char plabel[MAXLABELSIZE];
};
```

b) Do objeto *injector* para o *controller-collector* na destruição de *injector*:

O campo messageType é inicializado com 2 para identificar a mensagem como um sinal de destruição do objeto *injector*. O campo ilabel identifica o objeto *injector* que enviou a mensagem. O formato em C++ é:

```
struct InjectorDestructionMessage
{
    header h;
    char messageType;
    char ilabel[MAXLABELSIZE];
};
```

c) Do objeto *injector* para o *controller-collector* no envio de dados durante a Execução do Protocolo:

O formato da mensagem é a mesmo definido no Anexo 1 para os itens do arquivo *standard_data*, através da estrutura Standard. Este pacote é definido em C++ da seguinte forma: Standard CollectedDataMessage.

d) Do objeto *controller-collector* para o *injector* na criação de *injector*:

A estrutura FParam está definida no Anexo 1 e contém as informações de cada falha e é utilizada pela estrutura faultParametersMessage.

Em faultParametersMessage, o campo messageType é setado para 16, indicando que a mensagem leva parâmetros de falhas. O campo ilabel armazena o identificador único do objeto *injector* que é obtido do objeto *name_server*. O campo faultsQuantity indica a quantidade de parâmetros de falhas que estão sendo enviados. O campo faultsList é a lista de parâmetros de falhas a serem injetadas. O formato em C++ é:

```
struct faultParametersMessage
{
    header h;
    char messageType;
    char ilabel;
    int faultsQuantity;
    FParam **faultsList;
};
```

Formato dos Pacotes na Comunicação entre os Objetos *controller-collector*, *generator* e *name_server*

a) Do objeto *generator* para o *name_server* na geração do arquivo *fault_parameters*:

O campo messageType é setado para 10 para indicar que nameSetMessage é uma mensagem de atribuição de um nome ao servidor de nomes. O campo nodo é o identificador do nodo onde se localiza o processo que instanciará o objeto *injector*. O campo plabel é o identificador do processo que instanciará o objeto *injector*. O campo glabel é o identificador do grupo que instanciará o objeto *injector*. O campo ilabel é o identificador do objeto *injector*. O formato em C++ é:

```
struct nameSetMessage
{
    header h;
    char messageType;
    char nodo[MAXLABELSIZE];
    char plabel[MAXLABELSIZE];
    char glabel[MAXLABELSIZE];
    char ilabel[MAXLABELSIZE];
};
```

b) Do objeto *controller-collector* para o *name_server* na criação dos objetos *injector*:

O campo messageType é setado para 20 para indicar que nameRequestMessage é uma mensagem de requisição de um nome ao servidor de nomes. O campo requestNumber indica o número de sequência de requisição. O campo nodo é o identificador do nodo onde se localiza o processo que instanciou o objeto *injector*. O

campo plabel é o identificador do processo que instanciou o objeto *injector*. O campo glabel é o identificador do grupo que instanciou o objeto *injector*. O formato em C++ é:

```
struct NameRequestMessage
{
    header h;
    char messageType;
    int requestNumber;
    char nodo[MAXLABELSIZE];
    char plabel[MAXLABELSIZE];
    char glabel[MAXLABELSIZE];
};
```

c) Do objeto *name_server* para o *controller-collector* na criação de *injector*:

O campo messageType é setado para 30 para indicar que NameMessage é uma mensagem de resposta com a identificação do objeto *injector*. O campo requestNumber indica o número de sequência de requisição. O campo nodo é o identificador do nodo onde se localiza o processo que instanciou o objeto *injector*. O campo ilabel é o identificador do objeto *injector*. O formato em C++ é:

```
struct NameMessage
{
    header h;
    char messageType;
    int requestNumber;
    char ilabel[MAXLABELSIZE];
};
```

Bibliografia

[ARL 90] ARLAT, J. et al. Fault Injection for Dependability Validation: A Methodology and Some Applications. **IEEE Transactions on Software Engineering**, New York, v.16, n.2, p.166-182, Feb. 1990.

[BAR 90] BARTON, J. H. et al. Fault Injection Experiments Using FIAT. **IEEE Transactions on Computers**, New York, v.39, n.4, p.575-582, Apr. 1990.

[BAR 95] BARCELOS, P. P. A.; WEBER, T. S. Experimentação e avaliação de protocolos de difusão confiável. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995. Canela. **Anais...** Porto Alegre: SBC, 1995. p. 277-290.

[BIR 84] BIRREL, A. D.; NELSON, B. J. Implementing Remote Procedure Calls. **ACM Transactions on Computer Systems**, New York, v.2, n.1, p.39-59, Feb. 1984.

[BOO 94] BOOCH, G. **Object-Oriented analysis and design with applications.** 2. ed. Menlo Park: The Benjamin/Cummings, 1994. 589p.

[BOO 96] BOOCH, G. **Object solutions:** Managing the object-oriented project. Menlo Park: The Addison-Wesley. 1996. 323p.

[CAR 95] CARREIRA, J. et al. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In: WORKING CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, 1995, Urbana-Champaign, 1995. **Proceedings...** USA: [s.n.], 1995.

[CAR 95a] CARREIRA, J. et al. Assessing the Effects of Communication Faults on Parallel Applications. In: INTERNATIONAL COMPUTER AND DEPENDABILITY SYMPOSIUM, 1995, Erlangen, 1995. **Proceedings...** Germany: [s.n.], 1995. p.214-223.

[CHI 89] CHILLAREGE, R.; BOWEN, N. S. Understanding large system failures - A fault injection experiment. In: INTERNATIONAL SYMPOSIUM FAULT-TOLERANT COMPUTING, 19., 1989. **Proceedings...** New York: IEEE Computer Society Press, 1989. p.356-363.

[CHO 92] CHOI, G. S.; IYER, R. K. FOCUS: An Experimental Environment for Fault Sensivity Analysis. **IEEE Transactions on Computers**, New York, v.41, n.12, Dec. 1992.

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

[COU 94] COULOURIS, G; DOLLIMORE, J.; KINDBERG, T. **Distributed Systems: Concepts and Design.** 2. ed. Harlow: The Addison-Wesley, 1994. p.253-268.

[DAW 94] DAWSON, S.; JAHANIAN, F. **Probing and Fault Injection of Protocol Implementations.** USA: The University of Michigan, 1994. (Technical Report CSE-TR-217-94).

[DAW 95] DAWSON, S. et al. A Software Fault Injection Tool on Real-Time Mach. In: IEEE REAL-TIME SYSTEMS SYMPOSIUM, 1995. **Proceedings...** [S.l. : s.n.], 1995.

[DAW 96] DAWSON, S. et al. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 26., 1996. **Proceedings...** New York: IEEE Computer Society Press, 1996.

[ECH 91] ECHTLE, K.; CHEN, Y. Evaluation of Deterministic Fault Injection for Fault-Tolerant Protocol Testing. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 21., 1991. **Proceedings...** New York: IEEE Computer Society Press, 1991. p.418-425.

[ECH 92] ECHTLE, K.; LEU, M. The EFA Fault Injector for Fault-Tolerant Distributed System Testing. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT AND DISTRIBUTED SYSTEMS, 1992. **Proceedings...** New York: IEEE Computer Society Press, 1992 .p.28-35.

[ECH 94] ECHTLE, K.; LEU, M. Test of Fault Tolerant Distributed Systems by Fault Injection. In: IEEE WORKSHOP ON FAULT-TOLERANT PARALLEL AND DISTRIBUTED SYSTEM, 1994. **Proceedings...** [S.l.: s.n.], 1994.

[EZH 95] EZHILCHELVAN, P. et al. Newtop: A Fault-Tolerant Group Communication Protocol. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 25., 1995. **Proceedings...** New York: IEEE Computer Society Press, 1995. p.286-306.

[FOW 96] FOWLER, M. A Survey of Object-Oriented Analysis ans Design Methods. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1996, Linz, 1996. **Proceedings...** Austria: [s.n.], 1996.

[HAN 93] HAN, S. et al. **DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-time Systems.** [S.l.]: The University of Michigan, 1993. (Technical Report CSE-TR-192-93).

[HUT 91] HUTCHINSON, N. C.; PETERSON, L. L. The *x-kernel*: An Architecture for Implementing Network Protocols. **IEEE Transactions on Software Engineering**, New York, v.17, n.1, p.64-76, Jan. 1991.

[JOH 96] JOHNSON, R. How to Develop Frameworks. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1996, Linz, 1996. **Proceedings...** Austria: [s.n.], 1996.

[KAN 95] KANAWATI, G. A. et al. FERRARI: A Flexible Software-Based Fault and Error Injection System. **IEEE Transactions on Computers**, New York, v.44, n.2, p.248-260, Feb. 1995.

[KAR 94] KARLSSON, J. et al. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. **IEEE Micro**, New York, v.14, n.1, p.8-23, Feb. 1994.

[KAO 93] KAO, W. et al. FINE: A Fault Injection and Monitoring Environment for Tracing the UNIX System Behaviour under Faults. **IEEE Transactions on Software Engineering**, New York, v.19, n.11, p.1105-1118, Nov. 1993.

[LOV 93] LOVRIC, T.; ECHTLE, K. ProFI: Processor Fault Injection for Dependability Validation. In: IEEE INTERNATIONAL WORKSHOP ON FAULT AND ERROR INJECTION FOR DEPENDABILITY VALIDATION OF COMPUTER SYSTEMS, 1993, Gothenburg, 1993. **Proceedings...** Sweden: [s.n.], 1993.

[MAR 93] MARTINS, E. Validação dos mecanismos de tolerância a falhas em sistemas distribuídos por injeção de falhas. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 5., 1993. **Anais...** [S.l.: s.n.], 1993. p.233-251.

[MAR 93a] MARTINS, E. Validação Experimental da Tolerância a Falhas: A Técnica de Injeção de Falhas. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 5., 1993. **Anais...** [S.l.: s.n.], 1993. p.56-70.

[ROS 93] ROSENBERG, H. A.; SHIN, K. G. Software Fault Injection and its Application in Distributed Systems. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 23., 1993. **Proceedings...** [S.l.: s.n.], 1993. p.208-217.

[SEG 88] SEGALL, Z. et al. FIAT - Fault Injection Based Automated Testing Environment. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 18., 1988. **Proceedings...** New York: IEEE Computer Society Press, 1988. p.102-107.

[STE 90] STEVENS, W. R. **UNIX network programming.** New Jersey: Prentice-Hall. 1990. 772p.

[SUN 87] SUN MICROSYSTEMS INC. **NIT(4P).** *SunOS* 4.1: Reference Manual. USA: SUN Microsystems, 1987.

[SUN 90] SUNDERAN, V. S. PVM: A Framework for Parallel Distributed Computing. **Practice and Experience**, v.2, n.4, p.315-339, Dec. 1990.

[TEI 95] TEIXEIRA JR., C. A.; WEBER, T. S. FIX - uma ferramenta para recuperação de aplicações distribuídas no sistema operacional UNIX. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995. Canela. **Anais...** Porto Alegre: SBC, 1995. p. 333-344.

Informática  UFRGS

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

AFIDS- Arquitetura para Injeção de Falhas em Sistemas Distribuídos.

por

Irineu Sotoma

Dissertação apresentada aos Senhores:

Eliane Martins

Profa. Dra. Eliane Martins (IC-UNICAMP)

Maria Lúcia Lisbôa

Profa. Dra. Maria Lúcia Blanck Lisbôa

Raul F. Weber

Prof. Dr. Raul Fernando Weber

Vista e permitida a impressão.
Porto Alegre, 31/08/97.

Profa. Dra. Taisy Silva Weber,
Orientador.

Profa. Carla Maria Dal Sasso Freitas
Coordenadora Substituta do Curso de
Pós-Graduação em Ciência da Computação
Instituto de Informática - UFRGS