F-SOFIST — Uma ferramenta para teste de protocolos tolerantes a falhas

Marcos Renato Rodrigues Araujo

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

F-SOFIST — Uma ferramenta para teste de protocolos tolerantes a falhas

Marcos Renato Rodrigues Araujo

Outubro de 2000

Banca Examinadora:

- Eliane Martins (Orientadora)
  Instituto de Computação - Universidade Estadual de Campinas
- Taisy Silva Weber
  Instituto de Informática - Universidade Federal do Rio Grande do Sul
- Ricardo de Oliveira Anido
  Instituto de Computação - Universidade Estadual de Campinas
- Edmundo Madeira (Suplente)
  Instituto de Computação - Universidade Estadual de Campinas

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Araujo, Marcos Renato Rodrigues

Ar15f *F-SOFIST* — Uma ferramenta para teste de protocolos tolerantes a falhas / Marcos Renato Rodrigues Araujo – Campinas, [S.P. :s.n.], 2000.

Orientadora : Eliane Martins

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Protocolos de redes de computadores. 2. Tolerância a falhas (Computação). I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

F-SOFIST — Uma ferramenta para teste de protocolos tolerantes a falhas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Marcos Renato Rodrigues Araujo e aprovada pela Banca Examinadora.

Campinas, 30 de Outubro de 2000.

Eliane Martins

Eliane Martins (Orientadora)
Instituto de Computação -
Universidade Estadual de Campinas

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 17 de agosto de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Taisy Silva Weber
UFRGS

Prof. Dr. Ricardo de Oliveira Anido
IC – UNICAMP

Eliane Martins

Profa. Dra. Eliane Martins
IC – UNICAMP

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

À tia Ivone, minha primeira educadora.

Agradecimentos

À minha orientadora, Eliane Martins, pela paciência e confiança e constante incentivo.

Aos meus familiares, pelo apoio constante.

À Marília Dias Vieira Braga, pelo incentivo e cooperação.

Aos professores João Carlos Setubal e João Meidanis, meus atuais chefes, cuja colaboração foi essencial à conclusão deste trabalho.

À professora Cláudia M. Bauzer Medeiros, pelo empurrão final.

Resumo

Esta dissertação discute a implementação de uma ferramenta orientada à execução de testes de protocolos de comunicação. Sua implementação foi baseada em uma arquitetura de testes denominada *Ferry clip*, uma arquitetura de testes desenvolvida para dar suporte à metodologia de testes da ISO.

A principal contribuição deste trabalho está no desenvolvimento de uma nova arquitetura para teste de protocolos que permite injeção de falhas por software, resultando em uma ferramenta flexível e que possui um alto nível de portabilidade.

Abstract

This dissertation discusses the implementation of a tool oriented to execution of tests in communication protocols.Its implementation was based in a protocol test architecture named *Ferry clip*, developed to give support to ISO tests methodology.
The main contribution of this work is developing a new protocol test architecture that provides software fault injection, resulting in a flexible and highly portable tool.

Índice de figuras

Conteúdo

Capítulo 1

Introdução

1.1 Motivação e objetivos

Atualmente observa-se um crescimento do número de sistemas computacionais interconectados através de redes, formando sistemas distribuídos. Com a evolução dos sistemas computacionais, a tendência é que os protocolos de comunicação acompanhem essa evolução, tornando-se cada vez mais complexos e sofisticados. Isso torna essencial a existência de mecanismos que permitam testar de maneira adequada as implementações desses protocolos.

Apresentamos aqui uma arquitetura utilizada no desenvolvimento de uma ferramenta que, integrada a dois outros sistemas, um de geração de testes e outro de análise de resultados [MAS00, StM97], é capaz de realizar testes em protocolos de comunicação tolerantes a falhas. Essa ferramenta foi desenvolvida utilizando como base uma arquitetura denominada *Ferry-injection*, uma extensão da arquitetura *Ferry-clip* [CLP89, CVD92, Fis98], tornando-a capaz de injetar falhas por software.

A arquitetura *Ferry-clip* foi escolhida por apresentar uma série de vantagens:

- Suporte à execução de testes pela metodologia OSI/ISO;
- Suporte à execução de testes de interoperabilidade, permitindo testar a interação entre duas implementações distintas;
- Alto grau de portabilidade, facilitando sua utilização em diversas plataformas;
- Alto grau de modularidade;
- Baixo nível de intrusão no código do protocolo associado ao Sistema em Teste;

A nova arquitetura foi proposta com a seguinte motivação: desenvolver uma ferramenta que aproveitasse as vantagens da arquitetura *Ferry-clip* e fosse capaz de realizar injeção de falhas por software, possibilitando a validação de sistemas de comunicação. Com base na arquitetura proposta foi desenvolvida uma ferramenta de testes que recebeu o nome de F-SOFIST (*Ferry-clip with SOftware Fault-Injection Support Tool*).

1.2 Estrutura da dissertação

Essa dissertação está estruturada da seguinte forma:

- O capítulo 2 dá uma visão geral de todos os conceitos básicos para o desenvolvimento e compreensão da ferramenta;
- O capítulo 3 descreve a nova ferramenta, apresentando a nova arquitetura e todos os aspectos de implementação da F-SOFIST;
- O capítulo 4 mostra os resultados obtidos com a implementação e a utilização da F-SOFIST em um caso real;
- O capítulo 5 faz uma análise comparativa com outras ferramentas que realizam funções semelhantes;
- Finalmente, o capítulo 6 apresenta as conclusões e possíveis extensões futuras a este trabalho.

Ao final desta dissertação encontra-se um glossário descrevendo as principais siglas apresentadas aqui. Recomenda-se sua utilização no caso de qualquer dúvida.

1.3 Convenções

São utilizadas as seguintes convenções nesta dissertação:

- Exemplos de código estão escritos utilizando-se fonte `Courier;`
- Termos em inglês estão escritos em *itálico*;
- Tradução dos termos em inglês, quando disponíveis, estão **em negrito**;
- Textos enfatizados estão sublinhados;

Capítulo 2

Conceitos Básicos e Metodologia

Esta dissertação é resultado da implementação de uma ferramenta voltada à realização de testes em protocolos tolerantes a falhas. Este capítulo introduz os conceitos básicos para compreensão de nossa ferramenta. A seção 2.1 descreve o conceito básico de teste de protocolos, a seção 2.2 descreve algumas metodologias de teste, a seção 2.3 descreve a arquitetura *Ferry clip*, utilizada como base do desenvolvimento da ferramenta, e a seção 2.4 descreve a injeção de falhas.

2.1 Teste de protocolos

Define-se como *Teste de Protocolo* [LCV95] o conjunto de métodos ao qual se submete a implementação de um dado protocolo com o objetivo de verificar se o mesmo foi implementado de acordo com os seus requisitos e encontrar falhas de implementação. O objetivo principal destes testes é garantir a *Conformidade* das implementações [ISO88, LCV95, Lap91, Lap95, Por89, SHR97]. Podemos assim destacar os *Testes de Conformidade* como o conjunto de métodos utilizados para garantir que a implementação de um sistema está de acordo com a sua especificação. O processo de teste [Por89, LCV95] pode ser dividido em quatro partes:

- Geração de Testes *(Test Generation)*: é o processo de extração de casos de teste a partir de uma especificação do protocolo;
- Seleção e Parametrização de Testes *(Test Selection and Parameterization)*: uma implementação nem sempre necessita estar totalmente de acordo com a especificação. Desta maneira é possível selecionar um subconjunto entre todos os casos de teste possíveis de se aplicar à uma implementação para validá-la. O processo de identificação desse subconjunto é denominado Seleção de Testes e o processo de escolha dos valores para os parâmetros de cada caso de teste é denominado Parametrização de Testes;
- Execução de testes: é o ato de aplicar um caso de teste à implementação. O resultado desta execução deve ser armazenado para análise posterior.
- Análise de Resultados *(Test Verification)*: é o processo no qual verifica-se o comportamento de uma dada implementação utilizando-se uma determinada instância de teste, com o objetivo de verificar se aquele determinado caso de teste produziu o comportamento esperado de acordo com sua especificação [Gri99];

Existe uma norma específica que descreve como proceder para realizar testes de conformidade de protocolos denominada ISO IS9646 [ISO91]. Esta norma define 10 tipos de teste, onde podemos destacar quatro tipos de teste associados à conformidade:

- Testes de interconexão básica: verifica se existe conformidade suficiente para que a implementação se comunique com outras implementações do mesmo protocolo;
- Testes de capacidade: determina o que a implementação é capaz de fazer;
- Testes de comportamento: determina qual a extensão dos requisitos dinâmicos que são seguidos pela IUT (de *Implementation Under Test*, explicado na seção seguinte);
- Testes de resolução de conformidade: determina até que ponto uma implementação pode estar de acordo com a especificação fornecida pelo projetista;

Nos dedicaremos aqui aos testes de conformidade e seus métodos.

2.2 Métodos

A norma IS9646 também define métodos utilizados nos testes de conformidade. São divididos em dois grupos: locais e externos. Os testes externos por sua vez são subdivididos em três categorias: distribuídos, coordenados e remotos e a sua característica básica está no fato de um dos testadores se localizar à parte da implementação em teste, comunicando-se com esta através de primitivas de serviço abstrato.

Várias definições são apresentadas aqui. Para facilitar seu entendimento foi criado um glossário de termos, apresentado ao final desta dissertação.

2.2.1 Testes locais

Neste método a aplicação dos testes é feita interceptando-se as Primitivas de Serviço Abstrato (ASP's) da interface superior da Implementação em Teste (IUT), ou seja, suas primitivas de comando. O Testador Superior (UT) utiliza um Ponto de Controle e Observação (PCO) para aplicar os testes. Tanto o Testador Superior (UT) quanto o Testador Inferior (LT) – sendo este o responsável por interceptar as ASP's provenientes da interface inferior da IUT – localizam-se no Sistema de Teste (TS), que é o nó físico responsável pela aplicação dos testes no Sistema em Teste (SUT), que é a máquina onde localiza-se a IUT. Ambos os sistemas podem localizar-se no mesmo nó físico.

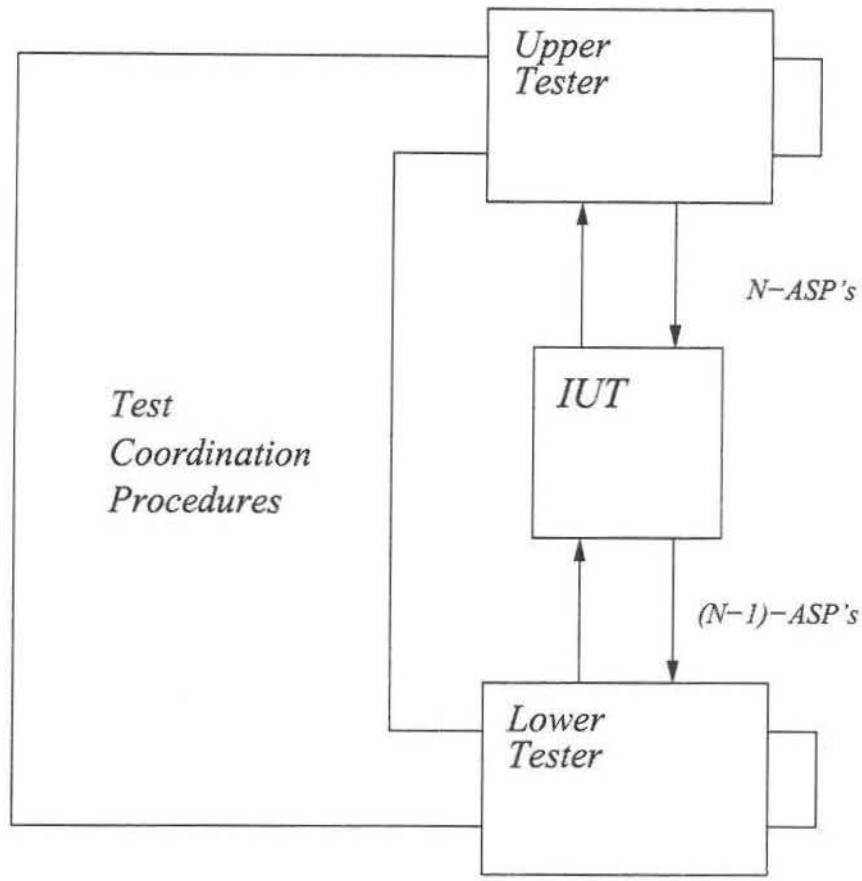

Figura 2-1: Teste Local

2.2.2 Testes distribuídos

Este método é semelhante ao método local para aplicação de testes no que se refere à interceptação das Primitivas de Serviço Abstrato. A diferença básica está no fato de o Testador Superior localizar-se junto à Implementação em Teste ao invés de estar no Sistema de Teste. Neste caso os procedimentos de teste são distribuídos entre o Sistema de Teste, que fica responsável pela Interface Inferior, e o Sistema em Teste, com a Interface Superior. A coordenação entre os testadores neste caso fica sob responsabilidade do usuário e, para isso, deve possuir uma interface que permita ao usuário ter controle sobre a mesma. As PDU's (*Protocol Data Units* – verifique glossário) apresentadas aqui se referem às informações trocadas entre a mesma camada de duas implementações de um mesmo protocolo, e encapsuladas na forma de ASP's.

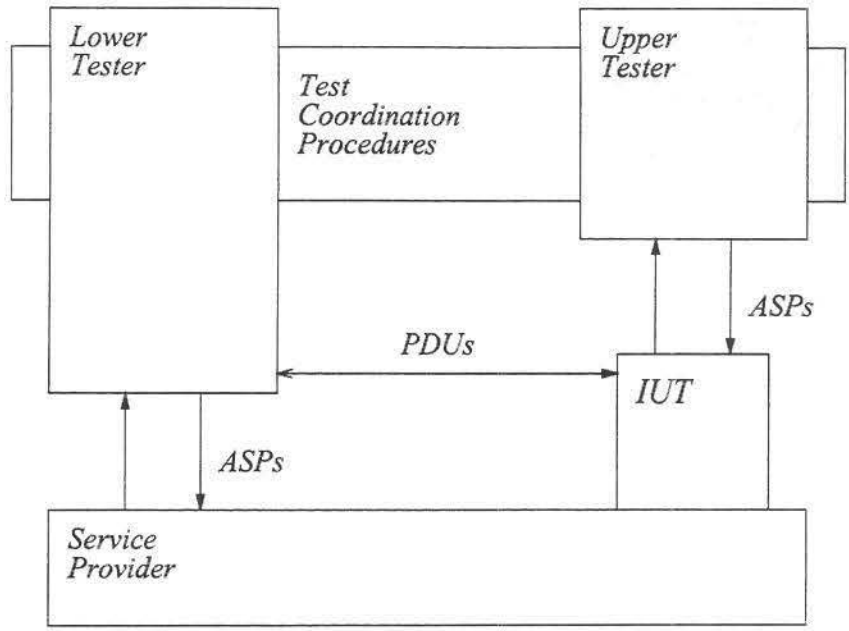

Figura 2-1:Testes Distribuídos

2.2.3 Testes coordenados

Neste método a aplicação dos testes é feita diretamente na interface superior da IUT; sendo *n* a própria IUT, o Testador superior deve ser capaz de gerar todos os eventos da camada (*n* + 1) do protocolo, ao menos nos aspectos relevantes da implementação. Neste caso a coordenação dos testes é feita através de um protocolo para troca de dados entre as partes.

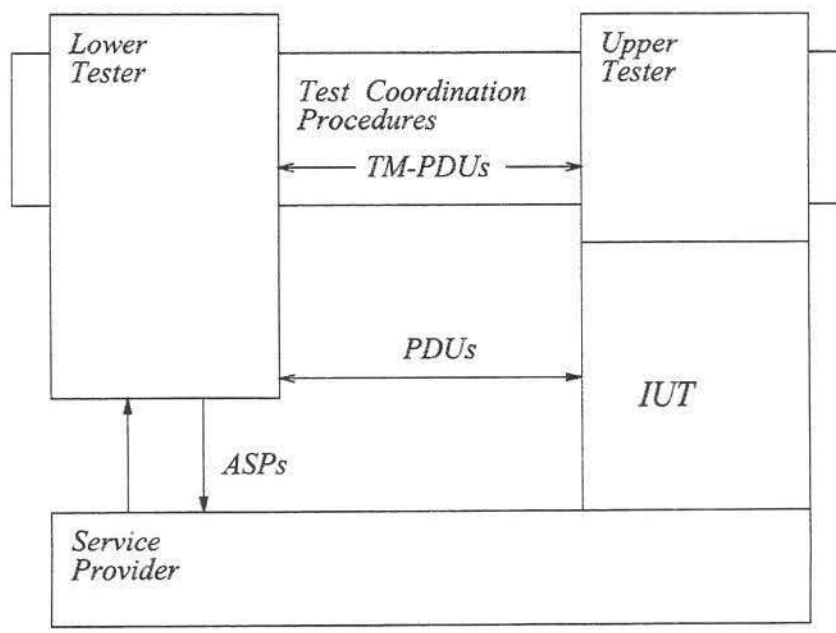

Figura 2-1: Testes Coordenados

Neste método a aplicação dos testes é feita diretamente na interface superior da IUT; sendo *n* a própria IUT, o Testador superior deve ser capaz de gerar todos os eventos da camada (*n* + 1) do

protocolo, ao menos nos aspectos relevantes da implementação. Neste caso a coordenação dos testes é feita através de um protocolo para troca de dados entre as partes.

2.2.4 Testes remotos

Neste método o Testador Superior é suprimido e os testes são realizados apenas através da interface inferior da Implementação em Teste. É utilizado quando não se tem acesso à interface superior da Implementação em Teste.

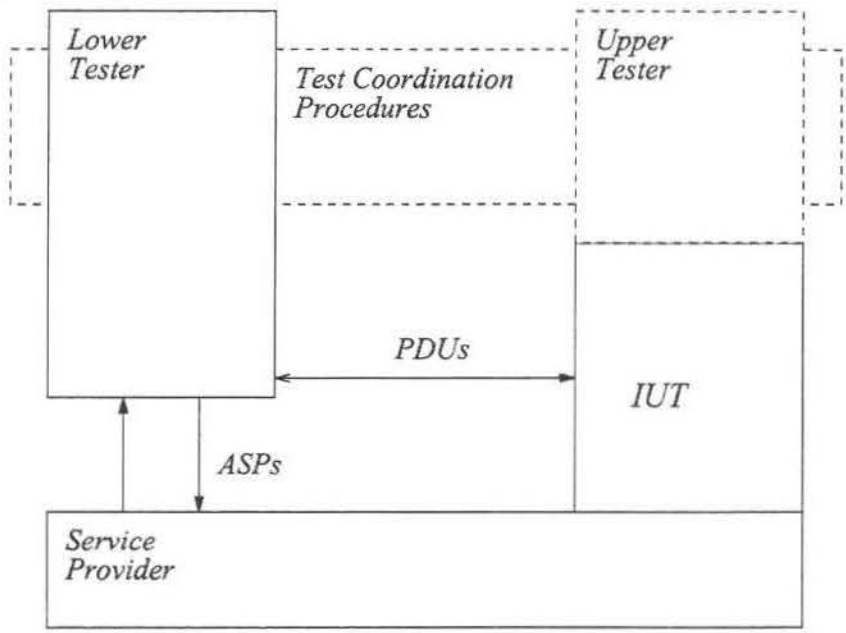

Figura 2-1: Testes Remotos

2.3 Arquitetura Ferry clip

A arquitetura *Ferry clip* [CLP89, CVD92] foi desenvolvida com o objetivo de dar suporte às metodologias de teste definidas pela ISO. A idéia básica é permitir que os dados sejam trocados entre o Sistema em Teste e o Sistema de Teste de maneira transparente de modo a permitir que tanto o Testador Superior quanto o Testador Inferior residam no Sistema de Teste. A implicação imediata disso é a facilidade de sincronização entre os testadores além de diminuir a quantidade de software associada ao Sistema em Teste.

Ela define basicamente dois componentes: o ***Ferry* Ativo** (*Active Ferry*) e o ***Ferry* Passivo** (*Passive Ferry*) - que a partir de agora serão referenciados apenas como AF e PF - que trocam dados entre si. Esta troca de dados é feita através de um protocolo particular de comunicação, denominado **Protocolo de Transferência entre *Ferry*'s** (*Ferry Transfer Medium Protocol*). A característica principal deste protocolo é permitir uma troca de dados confiável entre AF e PF.

As definições utilizadas na figura estão apresentadas no glossário.

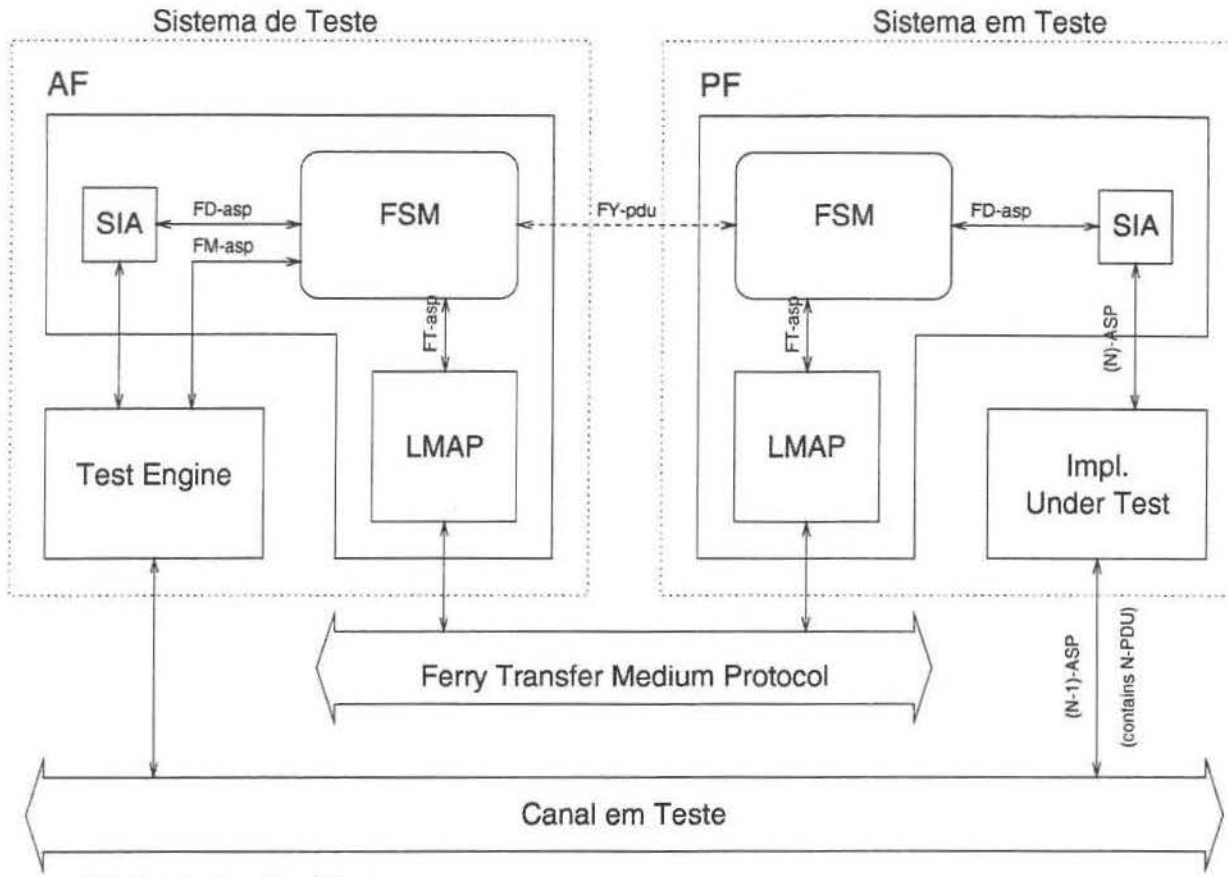

Figura 2-1: Arquitetura Ferry-clip

A Figura 2-1 apresenta esta arquitetura. O Sistema de Teste é formado pelo AF e pela Máquina de Testes (*Test Engine*), que agrupa as funções do Testador Superior e do Testador Inferior, sendo também responsável pelo envio das instruções (comandos) à Máquina de Estados Finita do AF. Estes comandos servem para estabelecer uma conexão com o PF, pedidos de desconexão e outros dados de controle específicos. Já o Sistema em Teste é formado pelo PF e pela Implementação em Teste. As informações são trocadas entre AF e PF através de Primitivas de Serviço Abstrato, aqui descritas:

- *Ferry Data* (FD-asp): primitiva de transporte de dados entre a Máquina de Testes e a Implementação em Teste;
- *Ferry Management* (FM-asp): para envio de comandos às FSM's;
- *Ferry Transport* (FT-asp), para a troca de dados entre AF e PF.

A Máquina de Estados Finita do AF e a do PF trocam dados através de "pacotes de dados" denominados FY-PDU's (*Ferry Protocol Data Units*), encapsuladas pelo *Ferry Transport* (FT-asp).

2.3.1 AF

O AF é a parte da arquitetura que reside no Sistema de Teste. Contém o software necessário para estabelecer, manter uma conexão e trocar dados com o PF, residente no Sistema em Teste. O AF é definido para ser independente de uma Implementação em Teste em particular de modo que a troca de uma Implementação em Teste por outra não afeta diretamente o AF. Isso permite ao Sistema de Teste ser reutilizado em outras Implementações.

Podemos dividir o AF em três partes interdependentes:

- *Lower Mapping Module*: encerra todas as funcionalidades para troca de dados entre AF e PF através do protocolo FTMP. Caso o FTMP seja trocado ou modificado de alguma forma, este é o único módulo afetado no AF;
- *Finite State Machine*: encerra todas as funções independentes do protocolo FTMP. Este módulo é responsável especificamente pelo controle da comunicação entre AF e PF (solicitação de conexão e desconexão, envio e recepção de dados);
- *Service Interface Adapter*[1]: realiza o "transporte" das interfaces da Implementação em Teste. É através deste módulo que a(s) interface(s) da Implementação em Teste fica(m) disponível(is) ao Sistema de Teste. Caso uma Implementação em Teste seja substituída, este é o único módulo afetado no AF.

2.3.2 PF

O PF reside no Sistema em Teste e, tal como o AF, possui o código necessário para estabelecer e manter uma conexão com o AF. O PF é responsável pelo encapsulamento e envio dos dados provenientes da Implementação em Teste ao Sistema de Teste através do AF e de aplicar os testes provenientes do Sistema de Teste na mesma.

O módulo PF é um "espelho" do módulo AF, podendo também ser dividido em três módulos:

- *Lower Mapping Module*: encerra todas as funcionalidades para troca de dados entre AF e PF através do protocolo FTMP. Caso o FTMP seja trocado ou modificado de alguma forma, este é o único módulo afetado no PF, da mesma maneira que no AF;
- *Finite State Machine*: encerra todas as funções independentes do protocolo FTMP. Este módulo é responsável pelo recebimento e processamento das solicitações feitas pelo AF (aceitação de conexão, solicitação de envio de dados), bem como as solicitações feitas pela Implementação em Teste;
- *Service Interface Adapter*: transporta de maneira transparente a(s) interface(s) da Implementação em Teste até o Sistema de Teste através do AF, recebendo dados desta e convertendo-os em um formato compreensível ao Sistema de Teste. No caso de substituição da Implementação em Teste, este é o único módulo substituído.

[1] Como o módulo PF também possui um módulo com a mesma finalidade, este módulo na maioria dos casos é considerado supérfluo. É utilizado basicamente quando se deseja uma representação intermediária entre Sistema de Teste e Sistema em Teste dos dados trocados entre eles. Na ausência desta representação intermediária, este módulo pode ser omitido.

A grande vantagem da arquitetura *Ferry clip* está no fato de não ser necessário desenvolver uma ferramenta nova para cada Implementação em Teste: basta modificar o Adaptador de Interface de Serviço em cada um dos módulos, deixando-se sempre uma interface padrão disponível à Máquina de Testes. Do mesmo modo, basta modificar os módulos LMAP para diferentes canais de conexão entre AF e PF. Desta maneira, pode-se aplicar os mesmos casos de teste em IUT's diferentes sem que haja necessidade de grandes modificações na ferramenta.

2.4 Injeção de Falhas

Antes de dar prosseguimento, é necessário conhecer alguns conceitos básicos: Segurança no Funcionamento, Falha, Erro e Defeito.

Denomina-se **Segurança no Funcionamento** (do inglês, *Dependability*) a propriedade que um sistema computacional possui de dar confiança aos serviços que ele oferece [Lap95]. A Segurança no Funcionamento possui algumas propriedades: disponibilidade (*availability* - estar pronto para o uso), confiabilidade (*reliability* - manter-se funcionando), segurança (*safety* – não causar danos ao sistema), confidencialidade (*confidentiality* – não permitir acesso não autorizado), integridade (*integrity* – não permitir alterações no conteúdo), e mantenabilidade (*maintainability* – permitir correções e evoluir).

Define-se defeito como sendo o resultado de um serviço do sistema que não está de acordo com sua especificação. Um erro é o estado do sistema que levou ou pode levar a um defeito. Uma falha é a causa suposta ou constatada de um defeito.

Injeção de Falhas [Blo97, Mar93, CMS94, Car95, SHR97, AAC90, MAA96, DJM97, Cri91, HTI97] é o nome genérico dado ao conjunto de técnicas utilizadas para acelerar a ocorrência de falhas, erros e defeitos em um sistema [Mar93]. Os principais objetivos são:

- Validação dos procedimentos de verificação;
- Validação dos mecanismos de tolerância a falhas;

A injeção de falhas pode ser feita de três maneiras:

- *Injeção por Hardware*: consiste em injetar falhas diretamente no hardware da Implementação em Teste, tais como sinais elétricos ou distúrbios eletromagnéticos. Estas técnicas em geral possuem alto custo de implementação devido à necessidade de desenvolvimento de um hardware próprio para esta finalidade, além da possibilidade de causar danos físicos à implementação;
- *Simulação de Falhas*: consiste em criar uma representação em software da Implementação em Teste e estudar seu comportamento. Esta técnica está associada a um alto custo de desenvolvimento devido à necessidade de se representar o Sistema em Teste através de software, algo nem sempre possível, além da ausência de garantias de que a simulação corresponderá à sua implementação real;
- *Injeção por Software*: possui duas características básicas: baixo custo de implementação e não causa danos físicos à Implementação. Consiste basicamente de um meio termo entre Injeção Física e Simulação Lógica, partindo do princípio que se pode emular falhas de hardware através de alterações no software da própria implementação. Deste modo não há necessidade de um hardware dedicado à injeção de falhas nem uma representação em

software. É um método muito mais simples de ser implementado a um custo muito mais baixo.

Como podemos observar, a injeção por software possui uma série de vantagens com relação aos outros métodos de injeção de falhas. Embora apresente várias vantagens com relação aos outros métodos, possui algumas desvantagens: alguns tipos de falhas não podem ser representados e esse tipo de injeção pode afetar as características de temporização do sistema, limitando seu uso em sistemas de tempo real.

No caso específico de protocolos de comunicação, uma característica que deve ser observada na injeção de falhas é o grau de intrusão do injetor. Como a injeção, independente do método utilizado, necessita de algum tipo de instrumentação, essa mesma instrumentação pode interferir na execução do Sistema em Teste. Uma solução apresentada em [DJM95, DJM96] é a colocação de uma camada de injeção intermediária imediatamente abaixo da camada do protocolo na qual se deseja injetar falhas, não havendo necessidade de se modificar o código da mesma, e utilizar uma outra estação independente do Sistema em Teste para monitorar os dados são trocados pela IUT.

2.5 Resumo

Este capítulo apresentou brevemente os conceitos básicos para a compreensão dos capítulos subseqüentes desta dissertação: metodologia OSI para teste de protocolos, arquitetura *Ferry-clip* e injeção de falhas.

Capítulo 3

Aspectos de implementação

Uma vez descritos os conceitos necessários, damos prosseguimento neste capítulo à descrição da nossa ferramenta. A ferramenta foi projetada para dar suporte à realização de testes em protocolos de comunicação que possuam mecanismos de tolerância a falhas e foi desenvolvida através da extensão da arquitetura *Ferry-clip*.

A seção 3.1 apresenta a nossa arquitetura proposta para injeção de falhas, a seção 3.2 apresenta detalhes sobre a implementação da nossa ferramenta e a seção 3.3 apresenta a modelagem da ferramenta utilizando-se orientação a objetos.

3.1 A arquitetura Ferry-injection

Para permitir injeção de falhas por software propõe-se aqui uma extensão para a arquitetura *Ferry-clip*. A idéia desta extensão é permitir a injeção lógica de falhas (tal como descrita na seção anterior) em protocolos que possuam mecanismo de recuperação e tolerância a falhas. Esta modificação consiste em acrescentar um injetor de falhas junto à IUT e um mecanismo simples de controle deste injetor junto ao Sistema de Teste. O mecanismo de troca de mensagens é o mesmo utilizado para enviar dados à IUT. As modificações deitas na arquitetura *Ferry-clip* estão mostradas na Figura 3-1.

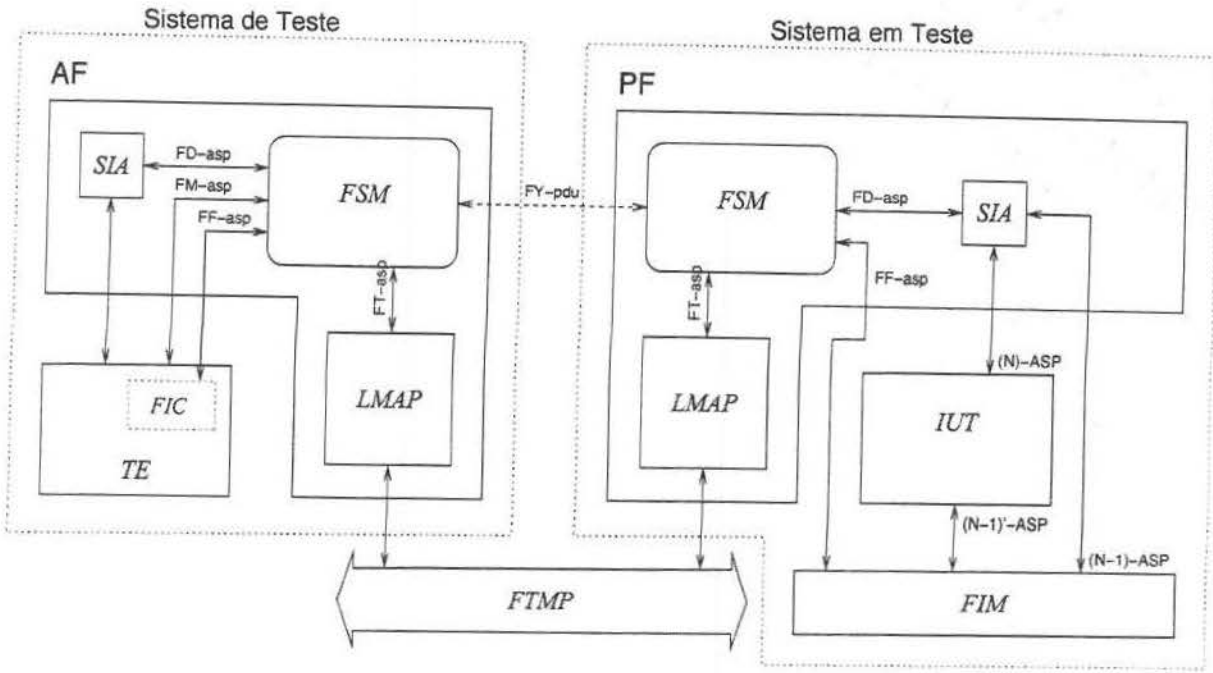

Figura 3-1: Arquitetura *Ferry-injection*

Comparando-se com a Figura 2-1 (capítulo anterior), pode-se verificar as seguintes alterações:

- Criação de outro canal lógico de transporte entre a Máquina de Testes[2] e o módulo injetor de falhas: FF ("*Ferry Faults*" – veja glossário);
- Um submódulo FIM acrescentado ao Sistema em Teste desempenha a função de injeção de falhas (este esquema será explicado adiante);
- Um submódulo FIC é acrescentado à Máquina de Testes;

Na seqüência apresentamos a funcionalidade de cada módulo.

3.1.1 Controle dos testes

Tal como explicado no capítulo anterior, este módulo agrupa as duas funções de teste UT e LT e a coordenação da conexão entre AF e PF. Na nossa proposta, este acumula também a função de controle do Módulo Injetor de Falhas (*Fault Injection Module* – FIM), localizado no Sistema em Teste, através do Controlador do Injetor de Falhas (*Fault Injection Controller*– FIC).

As funções de teste UT e LT estão agrupadas aqui em um submódulo interno denominado *Test Suite Processor* (TSP) uma vez que não há necessidade de existir dois módulos distintos realizando a mesma função. O papel do TSP é simplesmente coordenar e observar o que se passa nas interfaces superior e inferior da IUT. Também é nele que estão definidas as funções utilizadas pelo interpretador TCL para envio e recepção de dados.

[2] *Test Engine*; verifique glossário.

A Máquina de Testes também agrupa a interface com o usuário (no momento através de linha de comando) e um interpretador *TCL* [Ous94] utilizado para o processamento de scripts de teste, metodologia já utilizada em outras ferramentas do mesmo gênero [DJM95, DJM96, DWJ95]. No caso o script TCL contém todos os comandos para estabelecimento de uma conexão entre AF e PF, todos os procedimentos e dados de teste, comandos a serem enviados ao Injetor de Falhas e informações específicas relativas à IUT, ao PF e ao AF. Outro módulo é o *Logger*, responsável pela geração do relatório com os resultados do teste. Este relatório contém todas as informações sobre comportamento da IUT durante os procedimentos de teste e serve de entrada para uma outra ferramenta responsável pela análise dos resultados [StM97].

3.1.2 Módulos AF e PF

Estes dois módulos formam o *Ferry clip* tal como concebidos no capítulo anterior. Contém todos os procedimentos para a troca de dados entre IUT e *Test Engine*.

3.1.3 Módulo Injetor de Falhas (FIM)

O módulo injetor de falhas foi desenvolvido de modo a ser o mais simples possível, possuindo apenas quatro classes de falha. Definindo-se *pacote* como a unidade mínima de transporte de dados de um protocolo, podemos descrever estas classes como:

- *Atraso*: forçar um pacote a chegar fora de ordem ao destino;
- *Omissão*: suprimir um pacote para que ele não chegue ao destino;
- *Duplicação*: forçar um pacote a chegar repetidas vezes ao destino;
- *Corrupção*: corromper o conteúdo de um pacote;

Além disso, cada falha pode ser classificada, de acordo com a persistência, em:

- *Permanente*: ocorre em todos os pacotes enviados ao destino;
- *Transiente*: ocorre em um único pacote ou ocasionalmente;
- *Intermitente*: ocorre periodicamente;

Apesar da simplicidade, este modelo cobre quase todos os tipos de falhas que geralmente ocorrem em protocolos de comunicação. De fato, dentro da bibliografia pesquisada [Blo97, SoW97, DJM96, KaI94, HRS93, TsI95, SVS88, ECL92, DWJ95, DJM95] o nosso modelo não cobre apenas a geração de mensagens espontâneas [ECL92], uma variação da duplicação de mensagens que consiste em inserir uma mensagem aleatória na comunicação. Uma variante deste tipo de falha pode ser feita através de uma combinação de duas classes de falhas: corrupção e duplicação de mensagens. Como esta implementação é desnecessária no momento, será deixada para trabalho posterior.

Este módulo está integrado com o interpretador TCL, definindo para ele as funções de injeção e controle de falhas.

3.2 Detalhes de implementação

A ferramenta foi totalmente escrita em C++ e compilada utilizando-se o GNU g++. A Máquina de Testes e o AF foram compilados em uma estação SUN Sparc Ultra 450 com 1Gb de memória e plataforma Solaris 2.6, e o PF foi compilado em um PC dual Pentium III 450MHz com 512 Mb de memória e plataforma Linux.

Segue-se adiante com uma descrição completa da ferramenta.

3.2.1 Entrada

Nossa ferramenta é responsável apenas pela execução dos testes. Cada caso de teste será gerado por outra ferramenta cujo nome é CONDADO [MAS00]. Esta ferramenta foi desenvolvida para gerar casos de teste para protocolos cuja especificação foi feita utilizando-se Máquinas de Estado Finitas estendidas e cuja idéia básica é exercitar cada transição da máquina de estados finita ao menos uma vez, dando cobertura à parte de controle da especificação e retornando ao estado original.

Código 3-1: exemplo de saída da CONDADO

```
...
?U.SENDrequest !L.CR
?L.CC !U.SENDconfirm ?U.DATArequest
?L.RESUME
?L.TOKENgive !L.DT
?L.ACK !U.MONITOR_COMPLETE !L.TOKEN_RELEASE !L.DISrequest
?L.BLOCK
?L.DISrequest !U.DISindication
...
```

No Código 3-1, o símbolo '?' representa um dado que o testador está enviando à IUT e '!' indica um dado que o testador está esperando da IUT, 'U' e 'L' representam as interfaces superior e inferior e a seqüência que segue é uma instrução específica da IUT. Esses resultados são aplicados a uma ferramenta que gera um script em linguagem TCL a partir do resultado da ferramenta CONDADO, tal como mostrado no Código 3-2.

Código 3-2: código TCL referente ao Código 3-1

```
reset
senddata Upper SENDrequest
recvdata Lower CR
senddata Lower CC
recvdata Upper SENDconfirm
senddata Upper DATArequest
senddata Lower RESUME
senddata Lower TOKENgive
recvdata Lower DT
senddata Lower ACK
recvdata Upper MONITOR_COMPLETE
recvdata Lower TOKEN_RELEASE
recvdata Lower DISrequest
senddata Lower BLOCK
senddata Lower DISrequest
```

```
recvdata Upper DISindication
```

Os comandos do Código 3-2 não são comandos nativos da linguagem TCL, estando descritos no Apêndice A.

3.2.2 Interpretador TCL e gerador de LOG

São os dois módulos básicos de entrada e saída da Máquina de Testes. A ferramenta CONDADO produz uma série de comandos no formato descrito no Código 3-1. Esse código é convertido em uma série de comandos na linguagem TCL pela ferramenta de conversão mencionada, produzindo um script tal como descrito no Código 3-2. O interpretador TCL processa esse script, com eventuais modificações por parte do usuário. Este script contém todas as instruções que devem ser enviadas à IUT e que devem ser esperadas por ela, tal como descrito anteriormente. Simultaneamente ao processamento do script, o gerador de Log registra o comando que foi enviado à IUT e registra todos os dados provenientes da IUT para posterior análise de traço. Esta análise será feita por outra ferramenta [StM97].

3.2.3 AF/FSM e PF/FSM

As Máquinas de Estado Finitas são responsáveis pelo controle da conexão entre AF e PF. Seu funcionamento e seus estados fundamentais estão descritos nas tabelas disponíveis no Apêndice A. Os eventos são gerados ou pela Máquina de Testes ou pelo módulo LMAP (indicando que é um evento remoto).

Caso ocorra um evento não descrito na tabela, automaticamente é gerado um evento `FT-ERROR.ind` e é ativado o tratamento de exceção: uma mensagem de erro é enviada ao *Logger* (que a registra) e a conexão é encerrada unilateralmente.

3.2.4 AF/LMAP e PF/LMAP

São os dois módulos responsáveis pela troca de dados entre AF e PF através de um protocolo de comunicação confiável (FTMP). O protocolo escolhido é o TCP/IP, um protocolo bastante difundido comercialmente com o advento da Internet e disponível em uma grande variedade de plataformas. A troca de dados é feita através de FY-pdu's, tal como descrito no capítulo 2 e seu formato está descrito a seguir:

Figura 3-1: FY pdu para troca de dados (FD-ASP's)

Header 1 byte	Conn ID 1 byte	Length 2 bytes	String n bytes

Figura 3-2: FY pdu para troca de comandos (FM-ASP's)

Header 1 byte	Conn ID 1 byte	EXTRA 2 bytes

Os campos são:

- *Header*: cabeçalho. Contém a informação sobre o tipo de ASP sendo transportada. Os possíveis pacotes estão descritos no Apêndice A;
- *Conn ID*: identificador da conexão. É utilizado para identificar qual a conexão AF/PF que está sendo utilizada (no caso de múltiplas IUT's);
- *Length*: informa o tamanho do campo;
- *Extra*: informações adicionais para execução de um comando contido em um FM-ASP;
- *String*: campo de dados de tamanho *n*, onde *n* é o valor do campo *Length*;

Exemplo 1: código C++ que define FY-pdu

```
struct fy_pdu
{
  unsigned char control;
  unsigned char conn_id;
  union
  {
    unsigned short P_lenght;
    unsigned short P_extra;
  } P_le;

  char * stream;

#define p_lenght P_le.P_lenght
#define p_extra  P_le.P_extra
};
```

3.2.5 FIM

O funcionamento do injetor é simples: uma instrução proveniente do FIC é transmitida através de primitivas do tipo *FF* (FF-asp – verifique glossário) e enviada ao FIM contendo a seguinte informação: identificação do pacote onde a falha será injetada, método a ser utilizado, tipo de falha (0 para falhas transientes, 1 para falhas permanentes e $n > 1$ para falhas intermitentes, com n identificando o número de mensagens que circulam como intervalo de intermitência) e informações adicionais (tal como máscara de corrupção no caso de corrupção de pacotes ou tempo de espera até lançar o pacote novamente na comunicação no caso de duplicação ou atraso).

A Figura 3-1 mostra um esquema simplificado do FIM. Temos quatro funções básicas internas: *push* (retira o pacote da comunicação e armazena no buffer), *pop* (retira o pacote do buffer e joga na comunicação), *dup* (duplica o conteúdo do buffer e joga uma cópia na comunicação) e *clear* (limpa o buffer). O buffer em questão é alocado com o tamanho do pacote e um contador conta o número de pacotes que passam através dele, servindo o resultado desse contador como referência para identificação dos pacotes. Através deste modelo podemos exemplificar a injeção de falhas:

- *Atraso*: o FIM aloca memória em seu buffer onde mantém uma cópia do pacote, omitindo-o temporariamente (*push*), e deixando que o pacote seguinte seja processado pela IUT. Após a passagem de um determinado número de pacotes, o pacote retido é liberado (*pop*) e processado pela IUT. Os parâmetros deste método são então a mensagem na qual deve ser aplicada (de acordo com o contador) e o tempo de atraso (em número de mensagens).

- *Omissão*: identificar o pacote a ser suprimido e retirá-lo (*push* e *clear*) da comunicação com a IUT;
- *Duplicação*: basta identificar o pacote a ser duplicado (*push* e *dup*) e o tempo de atraso em número de pacotes trocados até que o pacote seja liberado (*dup* ou *pop*) novamente;
- *Corrupção*: a corrupção deve ser feita bit a bit. Além da identificação da mensagem é necessária uma máscara de bits (*Mask*) identificando quais os bits a serem modificados no buffer antes do seu envio;

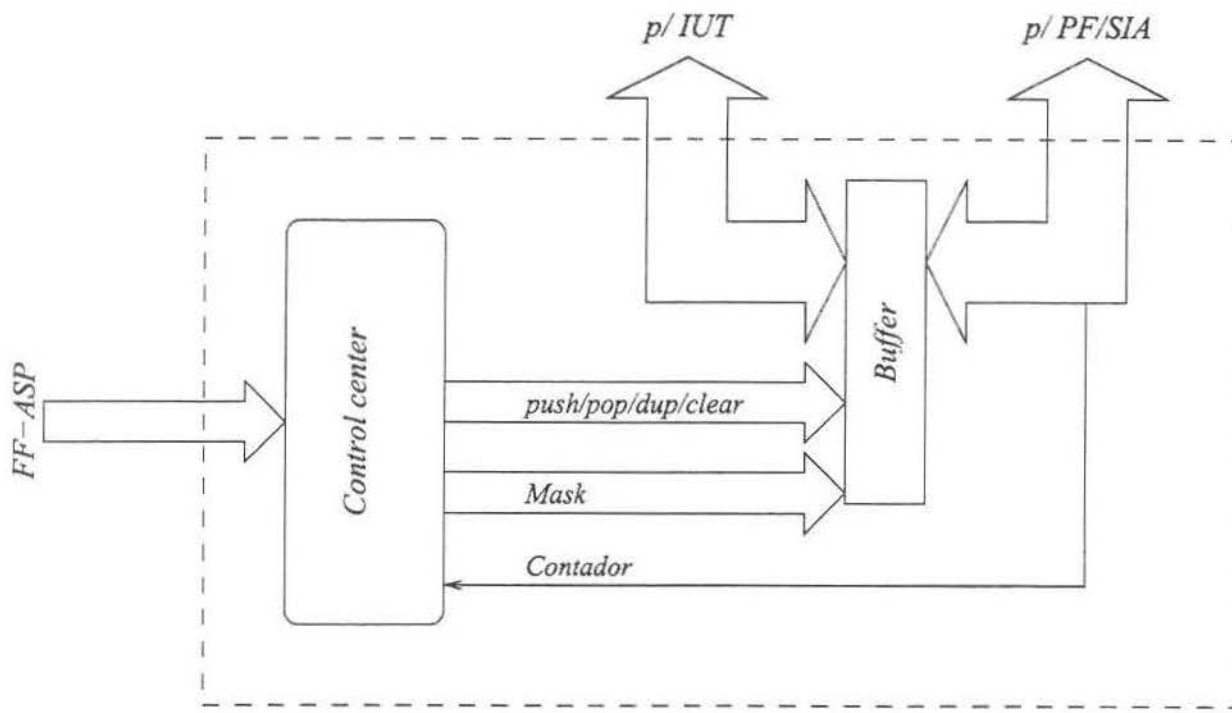

Figura 3-1: FIM

Essas falhas são armazenadas individualmente no injetor. Um limite é estabelecido para que o injetor não armazene instruções demais uma vez que normalmente a quantidade de memória disponível no Sistema em Teste ou mesmo na implementação é limitada. Para isso definiu-se a instrução `reset` cuja finalidade é limpar o conteúdo do injetor antes do próximo caso de teste.

3.3 Modelos de Objetos

Os modelos de objetos foram definidos a partir do modelo apresentado na Figura 3-1 e serviu como parâmetro para definição das classes utilizadas em nossa ferramenta. A descrição de cada classe segue abaixo:

- TCL Interpreter: processa o script do usuário com as funções setFile e processFile. A função createCommand é utilizada para definir os comandos da ferramenta definidos no Apêndice A; a função sendCmd é utilizada para definir a função `sendcmd` do interpretador TCL;

- Test Suite Processor: agrupa as funções do *Upper Tester* e *Lower Tester*; a variável UpperInterface contém o endereço da interface superior da Implementação em Teste e a variável LowerInterface contém o endereço da interface inferior da Implementação em Teste; as funções sendData e recvData são utilizadas pelo *TCL Interpreter* para definir as funções `senddata` e `recvdata`;
- Fault Injection Controller: envia comandos ao FIM; a função sendFcmd é utilizada pelo TCL Interpreter para definir a função `sendfcmd` do interpretador TCL;
- PortUnix: define um ponto de comunicação com duas interfaces; é utilizado por associação pelos outros módulos para troca de dados com o AF;
- Logger: armazena todas as informações provenientes dos outros módulos em um arquivo local;

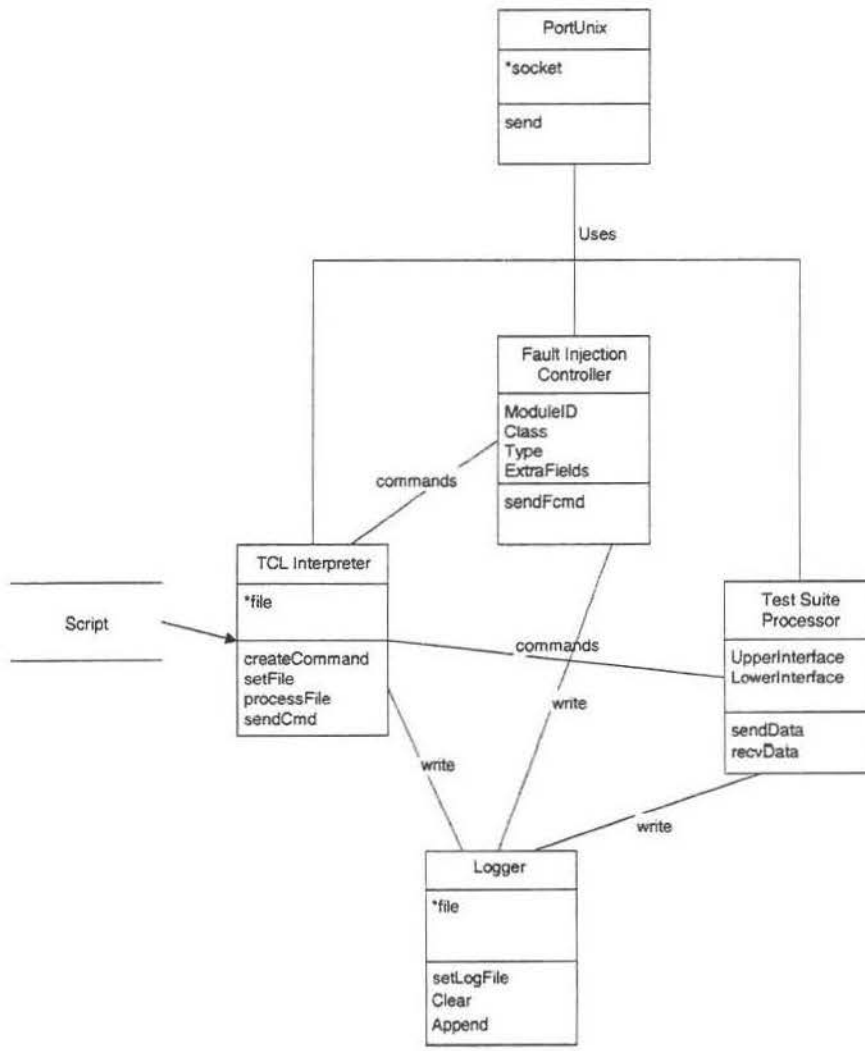

Figura 3-1: diagrama de objetos da Máquina de Testes

A Figura 3-2 mostra o diagrama de objetos para o AF. Observe que a classe portUnix é utilizada aqui também, uma vez que ela é necessária para a interface entre AF e *Test Engine*, feita através

de duas interfaces: via SIA (FD asp's) via FSM (FM e FF asp's). A classe FSM consiste da Máquina de Estados Finita cujos estados estão descritos no Apêndice A. A classe SIA consiste da interface de dados com a Máquina de Testes e contém as funções para troca de dados com ela. A classe LMAP herda todas as suas funcionalidades da classe FTMP, que consiste de uma interface com o FTMP (*Ferry Transfer Medium Protocol*).

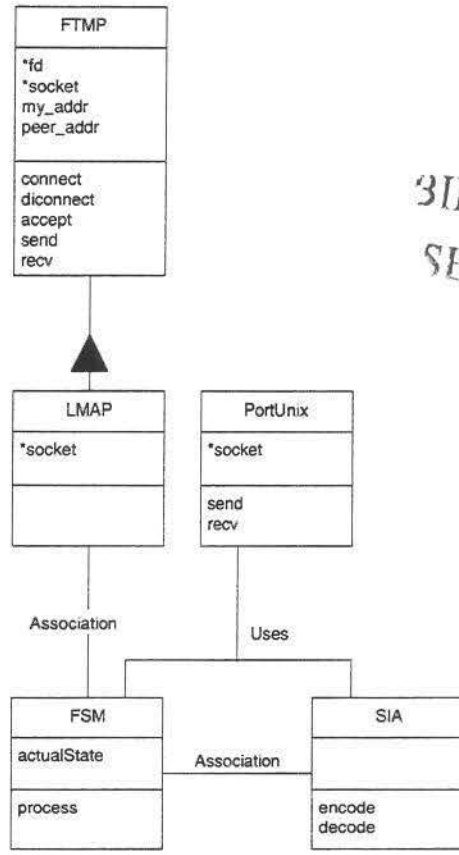

Figura 3-2: modelo de objetos para o AF

A Figura 3-3 mostra o diagrama de objetos do PF. Este diagrama é muito semelhante ao AF mas possui algumas diferenças fundamentais: o FSM reconhece os estados e eventos descritos no Apêndice A, a classe portUnix não é utilizada, o FIM herda todas as funcionalidades do SIA e a SIA usa por associação a interface com a IUT que é dependente da IUT.

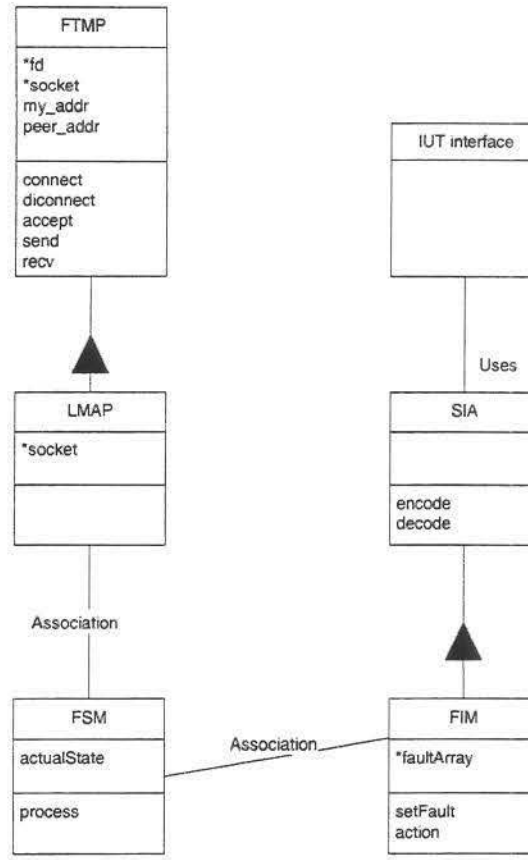

Figura 3-3: modelo de objetos para o PF

O esquema utilizado pela classe FIM permite que, no caso de testes SEM injeção de falhas, esta classe seja facilmente substituída pela classe SIA sem perda de funcionalidade.

3.4 Resumo

Neste capítulo apresentamos os diversos aspectos envolvidos no desenvolvimento de nossa ferramenta. Inicialmente foi introduzida a arquitetura *Ferry-injection*, uma versão modificada da arquitetura *Ferry-clip* que permite a injeção de falhas por software em protocolos de comunicação e seus aspectos de funcionamento. Na seqüência foram mostrados os detalhes de implementação desta ferramenta utilizando a arquitetura proposta. Finalmente foram apresentados os modelos de objetos utilizados na implementação da ferramenta, modelos os quais serviram de base para a codificação da mesma utilizando-se a linguagem C++.

Capítulo 4

Validação da ferramenta

Nosso objetivo inicial foi validar nossa ferramenta. Para tanto foi necessário escolher um protocolo para execução de testes; no nosso caso foi escolhido o protocolo SNMPv3. Este capítulo descreve os procedimentos de validação da ferramenta e como o SNMPv3 foi utilizado neste processo. Na seção 4.1 são apresentados os testes individuais de validação e integração da ferramenta; na seção 4.2 apresentamos o protocolo SNMPv3 e suas características; na seção 4.3 mostramos como esse protocolo foi utilizado como Implementação em Teste (IUT); na seção 4.4, uma vez integrada a ferramenta, são demonstrados os testes realizados sobre esta IUT.

4.1 Suíte de testes de validação da ferramenta

Nossa ferramenta está dividida em módulos e cada módulo foi submetido a testes de validação de maneira independente. Uma vez feito isso conjunto completo foi integrado e também submetido a testes de validação. Desta maneira, definiu-se assim uma suíte de testes que permitisse:

- Validar a Máquina de Testes;
- Validar isoladamente AF e PF e comunicação entre eles;
- Validar o Módulo Injetor de Falhas;
- Integrar os módulos;
- Aplicar a ferramenta em uma situação real;

Esta seção descreve o que foi feito para validar cada módulo e os procedimentos de integração da ferramenta, não necessariamente na ordem em que foram executados. A aplicação em uma situação real está descrita na seção 4.2.

4.1.1 Validação da Máquina de Testes

A Máquina de Testes está dividido em várias partes: *Fault Injection Controller* (FIC), o *Test Suite Processor* (TSP), o interpretador TCL e o *Logger*. O diagrama de blocos deste módulo está mostrado na Figura 4-1.

O FIC foi validado por partes: primeiro na geração de comandos para o FIM e depois no envio de comandos para o FIM, que é feito através do AF/PF. Para efeito de testes o FIM foi compilado juntamente com o FIC e o teste foi feito através de chamada de procedimento. Mais tarde, já com

o AF/PF pronto, foi possível verificar a troca de dados entre os dois módulos através de uma conexão de dados. Detalhes deste procedimento estão disponíveis na subseção 4.1.3.

O interpretador TCL é a peça mais importante da nossa ferramenta. Ele gera instruções para o FIC e para o TSP e foram escritas instruções específicas na forma de procedimentos para geração das chamadas a estes módulos. Esses procedimentos estão descritos no Apêndice A.

O TSP foi validado com ajuda do arquivo binário gerado pelo interpretador TCL. Este módulo foi considerado válido se o encaminhamento dos pacotes destinados à interface superior da IUT e os destinados à interface inferior estavam seguindo seu caminho correto.

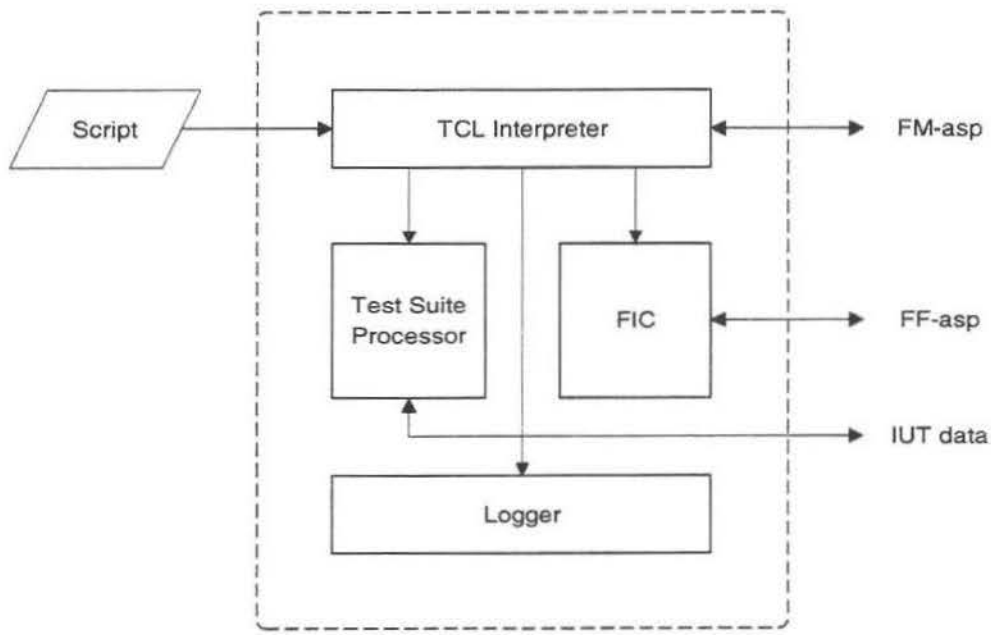

Figura 4-1: diagrama de blocos da Máquina de Testes

Como o papel do *Logger* é simples, ele foi considerado validado no momento em que foi capaz de armazenar as informações submetidas a ele.

Para efeito de integração, a Máquina de Testes foi considerado válido no momento que foi capaz de se conectar ao AF e trocar dados com ele.

4.1.2 Validação do AF e do PF

Para validação dos dois módulos que compõem o *Ferry clip* validou-se primeiro os seus submódulos SIA, FSM e LMAP.

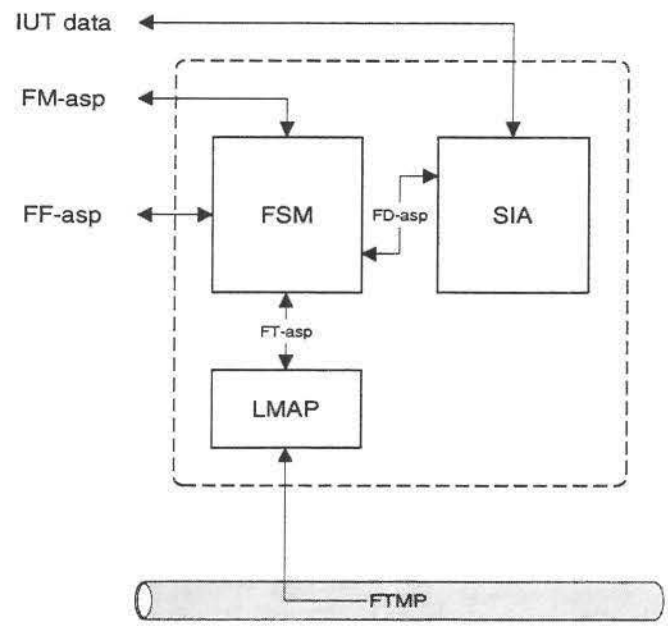

Figura 4-1: diagrama de blocos do AF

A validação dos módulos LMAP foi simples: bastou verificar se eles eram capazes de estabelecer uma conexão entre si e trocar dados adequadamente. Isso foi feito com um programa auxiliar que trocava dados utilizando este módulo. O LMAP foi considerado válido quando se verificou que o mesmo era capaz de trocar dados com seu par.

A validação dos módulos FSM também foi simples: AF/FSM e PF/FSM foram compilados juntos em um programa auxiliar. Os três estados possíveis da AF/FSM e os dois estados possíveis de PF/FSM (verifique Apêndice A) foram avaliados em conjunto e os módulos foram considerados válidos no momento em que todos os estados foram exercitados e responderam adequadamente. Como procedimento final os módulos foram integrados ao LMAP correspondente e verificou-se o estabelecimento de conexão entre os módulos utilizando-se o FSM para gerenciar essa conexão.

A validação dos módulos SIA depende muito do protocolo em teste, uma vez que estes módulos são os únicos dependentes da IUT. Para efeito de validação simples foram escritos dois SIA's "vazios" (funções *encode* e *decode* que simplesmente trocavam dados, sem manipulá-los) para verificar a troca de dados. Depois esses módulos foram integrados à FSM e ao LMAP correspondente. Os módulos foram considerados válidos quando permitiram essa troca de dados.

O *Ferry clip* foi considerado válido quando permitiu que dois programas trocassem dados entre si utilizando o *Ferry clip* como canal de troca.

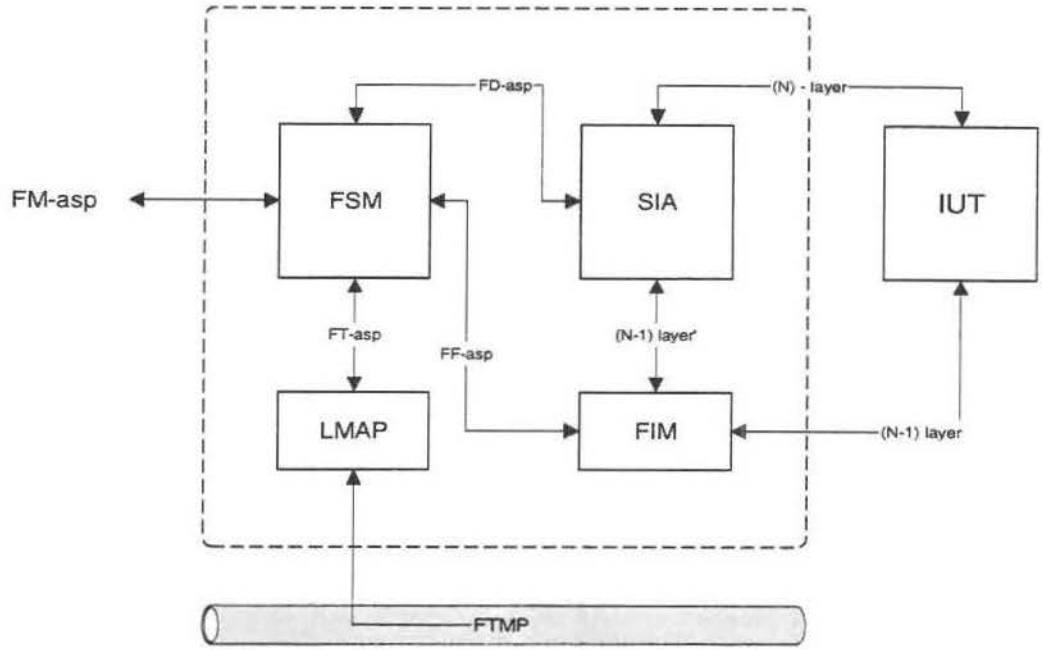

Figura 4-2: diagrama de blocos do PF com o FIM

4.1.3 Validação do FIM

Na ausência de um protocolo na qual fosse possível aplicar testes, este módulo só pode ser totalmente validado após a escolha de uma IUT. Para efeito de teste o módulo injetor foi compilado junto com o FIC de modo que o FIC chamasse o FIM como uma simples chamada de procedimento. Uma "PDU" artificial (um buffer contendo uma seqüência conhecida) foi passada ao módulo e submetida a cada caso de falha previsto, com o resultado sendo disponibilizado *on line* em intervalos regulares de um segundo. O módulo foi considerado válido quando todas as falhas previstas foram aplicadas com sucesso.

4.1.4 Integração dos módulos

A integração dos módulos foi feita de maneira gradual e por partes. O primeiro passo foi integrar *Test Engine* e AF. Como descrito anteriormente existem três canais de comunicação entre *ambos os módulos*: FD (troca de dados), FM (gerenciamento do *Ferry Clip*) e FF (envio de instruções de falhas do FIC ao FIM). Para efetuar essa troca de dados entre os módulos foi definido um esquema que permitisse no futuro prever testes de interoperabilidade e entre múltiplas partes [CVD92]. Prevendo isso, a comunicação entre ambos os módulos foi esquematizada para ser feita através de *sockets*, um para cada canal de comunicação, criados através da chamada de sistema *socketpair*. A Máquina de Testes cria um AF através da chamada de sistema *fork* e passa como parâmetro as três interfaces criadas. A Figura 4-1 mostra em detalhes este esquema. Caso se deseje utilizar a ferramenta para testes de interoperabilidade e/ou testes entre múltiplas partes, basta criar um novo conjunto de canais de comunicação e um novo AF para cada IUT (já associada a seu respectivo PF) a qual se deseja monitorar.

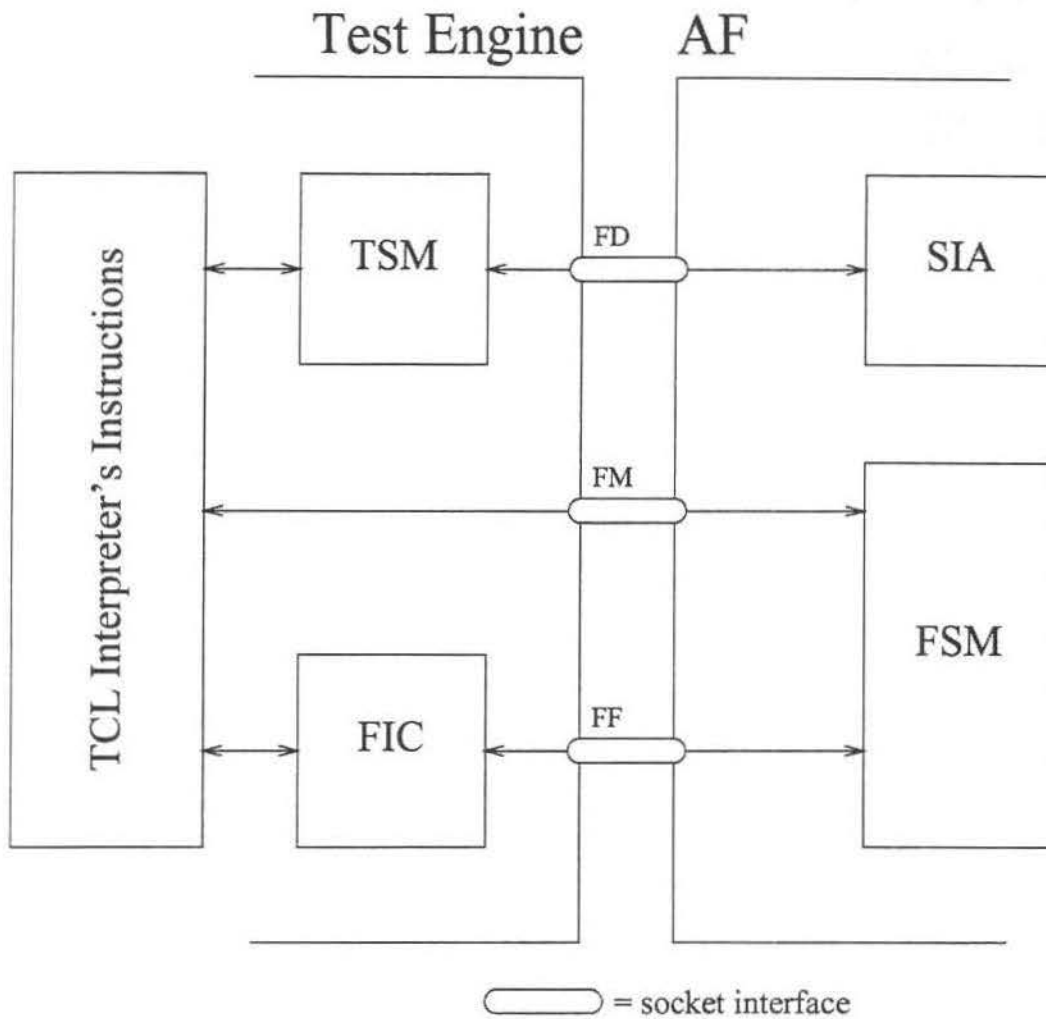

Figura 4-1: interface entre Test Engine e AF

Embora todos os recursos para implementar esse tipo de trabalho estejam prontos, este tipo de teste não foi realizado com nossa ferramenta, sendo deixado para trabalho futuro.

Em segundo lugar foi necessário verificar se AF e PF conseguiam trocar dados adequadamente. Para fazer isso se colocou o PF/FSM em modo *loopback* e estabeleceu-se uma conexão entre os dois lados. O modo *loopback* faz com que o PF simplesmente envie de volta ao AF qualquer informação que lhe chegue.

Em terceiro lugar foi necessário verificar se FIC e FIM conseguiam trocar dados através do *Ferry clip*. Para verificar isso bastou verificar se o FIM recebia as instruções do FIC, pois o funcionamento já foi testado individualmente. Essa verificação foi feita monitorando-se as ações do FIM para cada comando recebido.

Por último foi feito um esquema para verificar a integração geral da ferramenta. Para isso desenvolveu-se uma IUT "vazia" com duas interfaces, onde os dados recebidos por uma interface eram simplesmente copiados para a outra interface e verificava-se o resultado no *Logger*. Concluídos os testes de integração, passou-se aos testes em uma situação real, escolhendo-se uma IUT para aplicar testes.

4.2 Descrição da IUT

Podemos definir um protocolo de comunicação como um conjunto de regras que são utilizadas para coordenar a troca de dados entre dois equipamentos computacionais. O protocolo SNMP refere-se a "Protocolo Simples de Gerenciamento de Rede" (do inglês *Simple Network Management Protocol*). Sua finalidade básica é monitorar e controlar serviços prestados por uma determinada máquina como, por exemplo, monitorar filas de impressão, instalar programas, gerenciar bases de dados distribuídas, entre outras aplicações.

Sua especificação [RFC2570] define quatro entidades básicas:

- Agentes (*SNMP Agent*): cada Agente está associado a um dispositivo conectado à rede e oferece acesso remoto aos dados de gerenciamento daquele dispositivo;
- Estações Gerenciadoras (*SNMP Management Station*): responsável pelo gerenciamento e controle dos dados disponibilizados pelos Agentes;
- Protocolo de Gerenciamento (*SNMP Management Protocol*): responsável pela troca de dados entre uma Estação Gerenciadora e os Agentes ou entre os Agentes;
- Informação para Gerenciamento: informações sendo armazenadas;

Essas informações estão armazenadas em bases de dados denominadas "Bases de Informação para Gerenciamento" (*SNMP Management Information Bases* – MIB's). Essas bases são mantidas pelas Estações Gerenciadoras.

A troca de dados é feita através de SNMP PDU's sobre o protocolo UDP/IP, um protocolo de rede bastante simples que não requer conexão entre as partes. As PDU's básicas implementadas por este protocolo são:

- `GET.req`: realiza uma procura exata na MIB, retornando ao conteúdo que casou exatamente com a expressão de busca;
- `GET_NEXT.req`: realiza uma procura na MIB a partir de um nó específico;
- `GET_BULK.req`: realiza uma busca potencialmente grande na MIB, retornando todas as entradas na tabela que estão de acordo com o padrão de busca;
- `GET.rsp`: resposta a uma das solicitações anteriores;
- `SET.req`: realiza uma procura exata na MIB, modificando seu valor atual;
- `INFORM.req`: utilizado para troca informações entre estações gerenciadoras;
- `SNMPv2.trap`: utilizado para indicar que houve uma situação de exceção;
- `REPORT`: não definido, devendo ser definido pelo implementador;

Devido a uma série de problemas de segurança dos dados disponibilizados pelos Agentes o protocolo SNMP foi reformulado e uma nova versão foi concebida, denominada SNMPv2. Entre outras inovações, esta nova versão apresenta facilidades como:

- Notificação de eventos;
- Tratamento de erros;

- Autenticação de dados;
- Confidencialidade;
- Controle de acesso;

Infelizmente a nova versão do protocolo não foi bem aceita na comunidade comercial, principalmente devido à problemas de compatibilidade com a versão anterior do protocolo. Para tentar solucionar tais problemas, uma nova versão foi concebida, denominada SNMPv3, na verdade uma versão simplificada do SNMPv2. Entre outros objetivos desta nova concepção, podemos destacar:

- Acomodar ambientes operacionais distintos com diferentes demandas de gerenciamento;
- Facilitar a transição de versões anteriores do protocolo;
- Facilitar a manutenção das bases de dados e as diversas atividades desenvolvidas pelo protocolo;

As facilidades apresentadas no SNMPv2 já caracterizam um protocolo passível de ser testado pela nossa ferramenta. Contudo o SNMPv3 é uma versão mais recente, o que orientou nossa escolha como protocolo a ser testado.

4.3 Conexão da IUT à ferramenta

Para realização dos testes utilizamos uma implementação de domínio público do protocolo SNMPv3 denominada "ucd-snmp", versão 3.6.2, desenvolvida na *University of California at Davis*. Esta implementação consiste de um gerenciador e uma série de aplicativos que desempenham funções básicas utilizando o protocolo SNMP. O gerenciador é um *daemon* (*snmpd*) executando permanentemente em uma máquina a qual se deseja monitorar e que, como mencionado anteriormente, responde a conexões através de uma porta de comunicação utilizando o protocolo UDP/IP, sob a porta de número 161.

As informações disponibilizadas pelo gerenciador são passíveis de configuração e a implementação escolhida já apresenta uma série de itens pré-configurados, de modo que a configuração do Gerenciador é desnecessária.

Os aplicativos trabalham em conjunto com o Gerenciador providenciando uma interface por linha de comando que permite acesso às funções básicas do protocolo SNMPv3. Uma biblioteca de funções (*libsnmp*) também é disponibilizada caso o usuário deseje escrever seus próprios aplicativos. Esta biblioteca permitiu desenvolver o código do AF/SIA e do PF/SIA, tal como descrito na próxima subseção.

4.3.1 Implementação dos SIA's

Como definido anteriormente nossa IUT gerou durante sua compilação uma biblioteca de funções que possibilitam o desenvolvimento de aplicativos que utilizem o SNMP. Como o nosso objetivo neste trabalho é validar nossa ferramenta, em princípio a utilização desta biblioteca para desenvolver nossa interface com a IUT não atrapalha nosso objetivo. Por outro lado seria uma inclusão desnecessária que sobrecarregaria a interface com a IUT ainda mais. Como descrito anteriormente, nosso objetivo é desenvolver uma ferramenta que seja facilmente portável entre

diversas plataformas e trabalhe bem com limitações do Sistema em Teste. Assim sendo foram feitos dois tipos de teste: sem a utilização desta biblioteca e com a utilização da biblioteca, e os resultados de desempenho foram analisados.

No primeiro caso utilizaram-se os aplicativos que vieram junto com o pacote ucd-snmp. Esses aplicativos produziam os dados (PDU's) já no formato que deveriam ser enviados para a IUT, dados que posteriormente eram encapsuladas pelo *Ferry clip* e aplicadas diretamente à IUT. O papel da AF/SIA e da PF/SIA foi simplesmente repassar esses pacotes à IUT e enviar a resposta de volta.

No segundo caso a biblioteca de funções descrita na subseção anterior foi incorporada ao AF/SIA, permitindo que este criasse dinamicamente suas PDU's utilizando as instruções contidas no script. Neste caso a entrada de dados da ferramenta tornou-se mais clara.

Como descrito anteriormente, cada SIA possui duas interfaces: *encode* e *decode*. Pode-se dividir os procedimentos de geração de PDU's entre elas:

- Captura requisição da Máquina de Testes (encode);
- Filtra os argumentos recebidos da Máquina de Testes (encode);
- Cria uma requisição com esses argumentos (encode);
- Envia requisição ao PF/SIA (encode);
- Captura resposta do PF/SIA (decode);
- Processa resposta do PF/SIA (decode);
- Envia informação resultante de volta aa Máquina de Testes (decode);

O papel de envio dos dados diretamente à IUT continua pertencendo ao PF/SIA. Os procedimentos estão relacionados abaixo:

- Captura PDU do AF/SIA (decode);
- Envia requisição à IUT (decode);
- Captura a resposta da IUT (encode);
- Envia resposta ao AF/SIA (encode);

Uma vez definidos os parâmetros, pode-se dar continuidade aos testes.

4.3.2 Formato das PDU's

O formato da PDU é genérico para qualquer tipo de pacote trocado entre as partes é definido utilizando-se as informações disponíveis nas estruturas definidas abaixo. A estrutura *snmp_pdu* é utilizada para armazenar as informações sobre a requisição. Os campos são:

- Version: versão do SNMP (no nosso caso é v3);
- Address: endereço da máquina para consulta;
- SrcParty, dstParty, context: utilizado para autenticação dos dados;
- Community: comunidade à qual se deseja consultar;

- Command: comando a ser enviado;
- Reqid: identificação da solicitação;
- Errstat: último código de erro;
- Errindex: último índice de erro;
- Variables: estrutura com a requisição;

Alguns conceitos mostrados aqui devem ser configurados no momento em que o pacote ucd-snmp é compilado ou é instalado. A *comunidade*, por exemplo, é definida durante a instalação do pacote em um arquivo de configuração (a maneira como isso é feito será descrito logo adiante). Para nós, a comunidade a qual se deseja consultar depende dessa configuração.

Código 4-1: estrutura snmp_pdu

```
struct snmp_pdu {
 int version;

 /* Address of peer */
 snmp_ipaddr address;

 oid *srcParty;
 int srcPartyLen;
 oid *dstParty;
 int dstPartyLen;
 oid *context;
 int contextLen;

 /* community for outgoing requests. */
 u_char *community;

 /* Length of community name. */
 int community_len;

 /* Type of this PDU */
 int command;

 /* Request id */
 long reqid;

 /* Error status (non_repeaters in GetBulk) */
 long errstat;

 /* Error index (max_repetitions in GetBulk) */
 long errindex;

 /* TRAP INFORMATION */

 /* System OID */
 oid *enterprise;
 int enterprise_length;

 /* address of object generating trap */
 snmp_ipaddr agent_addr;

 /* trap type */
 long trap_type;

 /* specific type */
 long specific_type;
```

```
 /* Uptime */
 u_long time;

 /* MIB tree */
 struct variable_list *variables;
};
```

A estrutura *snmp_session* define os parâmetros sobre a comunicação, tal como endereço da máquina, porta para envio da solicitação, tempo de espera e porta de resposta, entre outros.

Observa-se uma redundância nos campos. Na verdade uma estrutura é utilizada para criar a outra, o que explica a redundância.

- Community: mesmo parâmetro definido na estrutura anterior;
- Retries: número de tentativas de envio antes de acusar erro de *timeout*;
- Timeout: tempo máximo de espera (em μS);
- Peername: endereço IP da máquina à qual se deseja enviar a requisição;
- Remote_port: porta para envio de requisições;
- Local_port: porta para recebimento da resposta;
- Authenticator: função para autenticação dos dados;
- Callback: função para tratamento dos dados que são recebidos;
- Callback_magic: apontador para dados relevantes resultantes da função anterior;
- Version: versão da aplicação (no nosso caso é v3);
- SrcParty, dstParty, context: utilizado para autenticação dos dados;
- Snmp_synch_state: armazena o código de erro ou de sucesso da requisição e, no caso de problemas, armazena a estrutura *snmp_pdu* que causou o erro;
- S_errno: código de erro retornado pelo sistema;
- S_snmp_errno: código de erro retornado pela requisição;

Código 4-2: estrutura snmp_session

```
struct snmp_session {
 /* community for outgoing requests. */
 u_char *community;

 /* Length of community name. */
 int community_len;

 /* Number of retries before timeout. */
 int retries;

 /* Number of uS until first timeout, then exponential backoff */
 long timeout;

 /* Domain name or dotted IP address of default peer */
 char *peername;

 /* UDP port number of peer. */
```

```
 u_short remote_port;

 /* My UDP port number, 0 for default, picked randomly */
 u_short local_port;

 /* Authentication function or NULL if null authentication is used */
 u_char *(*authenticator) (u_char *, int *, char *, int);

 /* Function to interpret incoming data */
 int (*callback) (int, struct snmp_session *, int, struct snmp_pdu *, void *);

 /* Pointer to data that the callback function may consider important */
 void *callback_magic;

 int version;
 oid *srcParty;
 int srcPartyLen;
 oid *dstParty;
 int dstPartyLen;
 oid *context;
 int contextLen;
 struct synch_state * snmp_synch_state;

 /* copy of system errno */
 int s_errno;
 /* copy of library errno */
 int s_snmp_errno;
};
```

A estrutura *variable_list* é a estrutura que armazena os parâmetros de busca propriamente ditos e que armazenam a resposta da solicitação. Os dados que interessam estão armazenados em *name* e em *val*.

Código 4-3: estrutura variable_list

```
struct variable_list {
 /* NULL for last variable */
 struct variable_list *next_variable;

 /* Object identifier of variable */
 oid *name;

 /* number of subid's in name */
 int name_length;

 /* ASN type of variable */
 u_char type;

 /* value of variable */
 union {
 long *integer;
 u_char *string;
 oid *objid;
 u_char *bitstring;
 struct counter64 *counter64;
 } val;
 int val_len;
};
```

A biblioteca de funções *libsnmp*, descrita anteriormente, contém funções que manipulam estas estruturas convertendo-as na PDU utilizada para troca de dados. O formato final é o resultado da serialização da estrutura genérica abaixo (representada no formato ASN.1) [RFC1157]:

Código 4-4: representação de uma PDU no formato ASN.1

```
-- request/response information
RequestID ::=
      INTEGER

ErrorStatus ::=
      INTEGER {
         noError(0),
         tooBig(1),
         noSuchName(2),
         badValue(3),
         readOnly(4)
         genErr(5)
      }

ErrorIndex ::=
      INTEGER

-- variable bindings
VarBind ::=
      SEQUENCE {
         name
            ObjectName,
         value
            ObjectSyntax
      }

VarBindList ::=
      SEQUENCE OF
         VarBind
```

Por exemplo, o formato da PDU GET.req, descrita na seção anterior, é:

Código 4-5: representação de GET.req no formato ASN.1

```
GetRequest-PDU ::=
      [0]
      IMPLICIT SEQUENCE {
            request-id
               RequestID,
            error-status -- always 0
               ErrorStatus,
            error-index  -- always 0
               ErrorIndex,
            variable-bindings
               VarBindList
      }
```

Com base nesses dados é possível estabelecer os procedimentos para injeção de falhas.

4.3.3 Troca de dados entre PF/SIA e IUT

A troca de dados entre as partes é feita através do protocolo UDP/IP, tal como descrito anteriormente. O protocolo UDP/IP efetua a troca de dados sem necessidade de se estabelecer uma conexão com a entidade que o recebe. A unidade de informação desse protocolo é denominada *datagrama*. Ao contrário do protocolo TCP/IP, o protocolo UDP/IP não possui nenhum mecanismo de recuperação de erros, o que o torna suscetível a erros que possam ocorrer na troca de dados, mas, em compensação, é um protocolo muito mais eficiente em termos de velocidade de transmissão e processamento. Assim, a eventual recuperação de erros deve ser feita pelo software que está utilizando o protocolo. No nosso caso a recuperação de erros deve ser feita pelo próprio protocolo SNMP através de rotinas de recuperação de erros.

Por outro lado isso causa um problema: para efeito de testes com injeção de falhas, como garantir que uma falha não foi causada pelo próprio meio físico? Nossa solução é simples: o PF/SIA e a IUT residem na mesma máquina e trocam informações através de sua interface interna (`localhost`), sem comunicação com o meio físico. Nestas condições o protocolo UDP é altamente confiável. Mesmo assim é necessário tomar alguns cuidados:

- Há um limite no número de pacotes que podem ficar esperando no buffer para serem lidos: todos os pacotes que chegarem após esse limite serão sumariamente descartados;
- Há um limite no tamanho do pacote: se esse limite for ultrapassado o pacote será dividido, sendo necessárias várias leituras no buffer para recuperar o pacote, causando uma eventual queda de desempenho;

Esses limites são impostos pelo sistema operacional e variam de um para outro. No caso específico de nossa ferramenta essas limitações não irão interferir no processo de teste, pois o tamanho dos pacotes trocados entre SIA e IUT é limitado e as requisições são processadas imediatamente após seu recebimento.

4.3.4 Injeção de falhas

Para que a injeção de falhas seja possível é necessário um mecanismo de interceptação das PDU's que saem do PF/SIA para a IUT e vice-versa. No caso da nossa ferramenta essa interceptação é feita dentro da própria SIA utilizando-se o módulo FIM, descrito anteriormente.

Por questão de simplicidade e para manter a modelagem da ferramenta o mais semelhante possível à estrutura *Ferry clip* original, optou-se por utilizar o mecanismo de *herança* para inserir o módulo FIM, aproveitando-se do fato que a ferramenta foi modelada orientada a objetos.

Observando-se novamente a modelagem da ferramenta apresentada no capítulo anterior, verifica-se que a classe PF/SIA é superclasse de PF/FIM. Isso foi feito para que, quando necessário, a classe PF/SIA substitua a classe PF/FIM de maneira transparente no caso de não necessidade deste recurso, sem prejuízos à implementação da mesma.

4.4 Testes com o SNMPv3

Para validação completa da ferramenta o pacote ucd-snmp 3.6.2 teve de ser compilado na plataforma onde os testes seriam feitos. A compilação foi feita sobre uma plataforma Linux executando em uma máquina Dual Xeon 300MHz com 512 Mb RAM, máquina que serviu para

nós como Sistema em Teste, consistindo do Gerenciador e o PF compilado sobre esta mesma plataforma. Este sistema é formado pela Máquina de Testes e o AF sobre uma plataforma Solaris 2.6, executando em uma máquina Ultra SPARC 366 MHz com 1Gb RAM e dois processadores. A conexão entre as duas máquina é feita através de uma rede *FastEthernet* e protocolo FTMP utilizado é o TCP/IP.

Para efeito de teste uma outra máquina foi colocada monitorando o FTMP. Essa máquina possui plataforma Linux em um Pentium II 233MHz com 128 Mb RAM. Como o papel dessa máquina é simplesmente monitorar a conexão entre AF e PF, ela recebeu o nome de *Monitor*.

A seqüência de testes consistiu basicamente de diversos scripts em TCL utilizados para validar cada comando da ferramenta. Cada script foi executado diversas vezes e o tempo médio foi dado como a média das diferentes execuções.

4.4.1 Configuração da IUT

A configuração do SNMPv3 é simples. No caso do ucd-snmp uma configuração básica já vem pré-estabelecida, cabendo ao usuário fazer as modificações que achar conveniente. Foi configurada uma única comunidade para efeito de teste que recebeu o nome de "*public*". O arquivo de configuração está mostrado abaixo.

```
####
# First, map the community name "public" into a "security name"
#       sec.name  source      community
com2sec public    default     public

####
# Second, map the security name into a group name:
#       groupName    securityModel securityName
group   publicGroup v2c           public

####
# Third, create a view for us to let the group have rights to:
#       name           incl/excl     subtree         mask(optional)
view    systemview     included      .1
view    systemview     excluded      .1.3

####
# Finally, grant the group read-only access to the systemview view.
#       group          context sec.model sec.level prefix read   write  notif
access  publicGroup    ""      any       noauth    exact  systemview none none

syslocation Institute of Computation/Unicamp
syscontact Root <root@lbi.ic.unicamp.br>
```

Código 4-1: configuração do gerenciador SNMP (snmpd)

Este exemplo deve ser descrito por partes. Todas as linhas iniciadas por "#" são ignoradas.

A entrada iniciada com "*com2sec*" serve para mapear uma comunidade a um nome. No caso definimos o nome "*public*" para esta comunidade, "*default*" indica que o nível de segurança é o nível padrão (acesso irrestrito).

A entrada iniciada por "*group*" associa o nome definido na seção *com2sec* a um grupo e a um nível de segurança (no caso v2c[3]). Vários nomes podem ser associados a um único grupo.

A entrada iniciada por "*view*" associa o nome "*systemview*" a um nível de acesso que inclui toda a sub-árvore "*.1*" mas exclui "*.1.3*".

Finalmente a entrada iniciada por "*access*" configura o nível de acesso à MIB pelo grupo "*publicGroup*" (definido na entrada "*group*"). Respectivamente a entrada diz que todas as PDU's trocadas utilizam contexto[4] nulo, que vale para qualquer modelo de segurança (v1, v2, v2c ou usm), que o nível de segurança é irrestrito ("*noauth*"), que o prefixo da PDU enviada pelo agente deve casar exatamente com o contexto definido (no caso é nulo), que a leitura deve utilizar o nível de acesso "*systemview*", não é permitido modificar a MIB ("*none*") e não ocorrem notificações[5].

4.4.2 Exemplo de execução

O objetivo deste experimento básico é introduzir a sintaxe do script e não utiliza nenhuma verificação de erros. Neste exemplo simples a ferramenta envia 1000 GET.req e bloqueia a execução do script até a chegada de um GET.rsp para cada solicitação. Não é feito nenhum tratamento de erro. A sintaxe do script é simples e foi apresentada na seção 3.2.1. Os detalhes de cada comando podem ser encontrados no Apêndice B.

Entrada

```
# INITIALIZATION

# timeout is 1 second (1000000 uS)
set timeout 1000000

# show current date
puts -nonewline "#Date: "; date

# open connection with peer PF #1 (host lua at port 2345)
openconn { 1 lua 2345 }

# send 1000 GET.req to community public, MIB entry "system.sysDescr.0"
time {
    senddata { 1 Lower GETreq "public 1.1.0" }
    recvdata { 1 Lower GETrsp "^system.sysDescr"}
} 1000

# close connection
closeconn { 1 }
```

[3] V1, V2, V2C e USM identificam três modelos de segurança distintos e refere-se respectivamente às normas definidas para o protocolo SNMPv1, SNMPv2, SNMPv2 estendido e SNMPv3.
[4] Padrão de letras que deve estar contido no cabeçalho de todas as PDU's. Utilizado apenas no nível de segurança USM.
[5] Notificações são ações que o gerente executa no caso de um acesso não autorizado.

Saída obtida no Logger

As linhas iniciadas por ">" indicam que foi enviada uma PDU ao PF, e linhas iniciadas por "<" indicam que chegou uma PDU do PF. Na seqüência vem o número que identifica o PF (especificado na instrução "*openconn*"). A segunda coluna identifica a interface que enviou/recebeu a PDU. A terceira coluna identifica o comando enviado seguido pelo parâmetro entre parênteses. No final da linha é mostrado o tempo decorrido na execução do comando.

```
#Date: Fri Jul  2 02:49:14 BRT 2000
#connection stablished: PF#1 (143.106.73.50,2345)
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 31ms
>1 Lower GETreq ("public 1.1.0") 31ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 33ms
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 34ms
>1 Lower GETreq ("public 1.1.0") 34ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 36ms

(...)

>1 Lower GETreq ("public 1.1.0") 31ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 32ms
64438 microsseconds per iteration
#connection closed: PF#1 (duration: 79,6 seconds)
```

Código 4-1: script TCL para enviar 1000 GET.req e aguardar resposta

4.4.3 Consulta básica

Uma consulta básica consiste de um GET.req e um GET.rsp. Neste caso específico o script envia um comando "senddata", fazendo uma requisição à comunidade public e ao nó system.sysDescr.0, ou numericamente 1.1.0[6] (system = 1, sysDescr = 1). Este nó específico da MIB contém a descrição do sistema operacional utilizado. A instrução "recvdata" é utilizada para bloquear a execução do script até a chegada da resposta.

Entrada

```
# open connection with peer PF #1 (host lua at port 2345)
openconn { 1 lua 2345 }

# senddata command: senddata <PF#> <Interface> <message>
# send GET.req to community public, MIB entry "system.sysDescr.0"
senddata { 1 Lower GETreq "public 1.1.0" }

# recvdata command: recvdata <PF#> <Interface>
```

[6] Cada nó da MIB possui um código numérico e um código alfanumérico (exceto nas extremidades) que é definido durante a inicialização da MIB através de diversos arquivos de configuração. Como esses arquivos de configuração são extensos, optou-se por fazer referências a esses códigos apenas quando necessário.

```
if { recvdata { 1 Lower GETrsp "^system.sysDescr"} } then
{ puts "Error in answer; exiting\n"; exit }

# close connection with peer PF #1
closeconn 1
```

Saída obtida no Logger

```
#connection stablished: PF#1 (143.106.73.50,2345)
>1 Lower GETreq ("public 1.1.0") 35ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 33ms
#connection closed: PF#1 (duration: .078 seconds)
```

Código 4-1: consulta GET.req básica

4.4.4 Consulta completa

Uma consulta mais complexa pode ser feita fazendo-se uso de variáveis auxiliares. O objetivo deste teste é mostrar algumas vantagens da utilização da linguagem TCL como script de controle. A busca começa a partir do nó snmp.snmpInPkts.0 – representado pelo código 11.1.0 – e prossegue buscando por 11.2.0, 11.3.0 etc. até que seja atingido o último nó da sub-árvore snmp.

Entrada

```
# open connection with peer PF #1 (host lua at port 2345)
openconn { 1 lua 2345 }

# constant values
set node1 11
set node3 0
set counter 0

# should cause error when answer not matches "snmp";
# this is an indication that the subtree "snmp" is finished.
for { set node2 1 }
    { ![ senddata { 1 Lower GETreq "public $node1.$node2.$node3" } ] &&
      ![ recvdata { 1 Lower GETrsp "^snmp"} ] }
    { incr $node2 } {
    incr counter
}

decr counter # last count is failed
puts "#Total nodes in subtree 11: $counter"
# close connection with peer PF #1
closeconn 1
```

Código 4-1: script TCL para enviar 100 GET.req e aguardar resposta

Saída

```
#connection stablished: PF#1 (143.106.73.50,2345)
>1 Lower GETreq ("public 11.1.0") 35ms
<1 Lower GETrsp ("snmp.snmpInPkts.0 = 13145") 34ms
>1 Lower GETreq ("public 11.2.0") 33ms
```

```
<1 Lower GETrsp ("snmp.snmpOutPkts.0 = 13075") 34ms
>1 Lower GETreq ("public 11.3.0") 32ms
<1 Lower GETrsp ("snmp.snmpInBadVersions.0 = 30") 35ms
>1 Lower GETreq ("public 11.4.0") 33ms
<1 Lower GETrsp ("snmp.snmpInBadCommunityNames.0 = 0") 34ms
>1 Lower GETreq ("public 11.5.0") 36ms
<1 Lower GETrsp ("snmp.snmpInBadCommunityUses.0 = 0") 33ms
>1 Lower GETreq ("public 11.6.0") 33ms
<1 Lower GETrsp ("snmp.snmpInASNParseErrs.0 = 10") 34ms
>1 Lower GETreq ("public 11.7.0") 34ms
<1 Lower GETrsp ("(noSuchName)") 35ms
#Total nodes in subtree 11: 6
#connection closed: PF#1 (duration: .078 seconds)
```

O resultado da execução do script mostrou que apenas seis entradas foram encontradas antes do fim da execução ("`#Total nodes in subtree 11: 6`"). Concluiu-se que a sub-árvore possui apenas seis entradas e que o SNMP utiliza a chamada GETrsp para enviar a resposta mesmo na ocorrência de erros.

4.4.5 Injeção de falhas

Este teste consiste em modificar o cabeçalho da PDU, transformando a PDU em informação inválida.

```
# timeout is 1 seconds (1000000 uS)
set timeout 1000000

# open connection with peer PF #1 (host lua at port 2345)
openconn { 1 lua 2345 }

# reseting fault counter
resetFaults { }

# GET req is 54 bytes; changing 2 bits at header, iteration #2
sendFcmd { 1 C 2 "180000000000000000000000000000000000000000000000000000" }

# senddata command: senddata <PF#> <Interface> <message>
# send GET.req to community public, MIB entry "system.sysDescr.0"
senddata { 1 Lower GETreq "public 1.1.0" }

# recvdata command: recvdata <PF#> <Interface>
if { [ recvdata { 1 Lower GETrsp } ] == 1 } {
   puts "#Error in answer; exiting\n"; exit
}
senddata { 1 Lower GETreq "public 1.1.0" } # Fault inserted here

# recvdata command: recvdata <PF#> <Interface>
if { [ recvdata { 1 Lower GETrsp } ] == 1 } {
   puts "#Error in answer; exiting\n"; exit

# close connection with peer PF #1
closeconn 1
```

Saída

```
#connection stablished: PF#1 (143.106.73.50,2345)
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 33ms
!1 corruption ("1F00000000000000000000000000000000000000000000000000000000")
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower TIMEOUT
#Error in answer; exiting
```

O teste foi considerado falho uma vez que uma nova PDU não foi enviada. Através do monitor foi possível verificar que a PDU fora enviada e através do registro de *log* verifica-se que a injeção foi feita. O que houve de errado então? Verificou-se que o *timeout* escolhido foi muito curto (apenas um segundo) e que não houve tempo de a IUT receber uma nova PDU com a informação corrigida. Neste caso é necessário considerar o tempo de espera da *libsnmp*[7], utilizada para geração de PDU's, antes de se considerar o erro de *timeout*. Bem, esse tempo de espera foi configurado com o valor default de 1 segundo. Observar que: (i) ou o erro está na programação do agente ou (ii) o *timeout* realmente deveria ser ajustado devido ao retardo introduzido pela ferramenta de testes.

Assim um novo teste foi feito utilizando-se o script anterior modificando-se o timeout para 5 segundos.

```
# timeout is 5 seconds (5000000 uS)
set timeout 5000000

# open connection with peer PF #1 (host lua at port 2345)
openconn { 1 lua 2345 }

# reseting fault counter
resetFaults { }

# GET req is 54 bytes; changing 5 bits at header, iteration #2
sendFcmd { 1 C 2 "1F00000000000000000000000000000000000000000000000000000000" }

# senddata command: senddata <PF#> <Interface> <message>
# send GET.req to community public, MIB entry "system.sysDescr.0"
senddata { 1 Lower GETreq "public 1.1.0" }

# recvdata command: recvdata <PF#> <Interface>
if { [ recvdata { 1 Lower GETrsp } ] == 1 } {
   puts "Error in answer; exiting\n"; exit
}

senddata { 1 Lower GETreq "public 1.1.0" } # Fault inserted here

# recvdata command: recvdata <PF#> <Interface>
if { [ recvdata { 1 Lower GETrsp } ] == 1 } {
   puts "Error in answer; exiting\n"; exit
}

# close connection with peer PF #1
closeconn 1
```

[7] O reenvio de PDU's deveria ser previsto no script e corrigido por este, não pelo AF/SIA, módulo no qual está sendo utilizada a biblioteca *libsnmp*. Devido à falta de tempo hábil este problema não foi corrigido, sendo deixado para trabalho futuro.

Saída

```
#connection stablished: PF#1 (143.106.73.50,2345)
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 33ms
!1 corruption ("1F000000000000000000000000000000000000000000000000000000")
>1 Lower GETreq ("public 1.1.0") 32ms
<1 Lower GETrsp ("system.sysDescr.0 = Linux lua 2.2.14-5.0smp #1 SMP Tue Mar 7
21:01:40 EST 2000 i686") 1189ms
#connection closed: PF#1 (duration: 3.33 seconds)
```

Verifica-se que uma nova solicitação foi enviada à IUT e a resposta foi recebida com sucesso, com um timeout mais longo que o habitual. Pode-se considerar este teste um sucesso.

4.4.6 Medidas de desempenho

Aqui são mostrados alguns testes específicos para medir o desempenho da ferramenta, a fim de se ter uma base de comparação com outras ferramentas que desempenham o mesmo papel.

Medição do atraso na interpretação de um script: o objetivo deste teste é verificar o atraso causado pelo interpretador TCL na execução da ferramenta. Para isso foram utilizados dois scripts: um script vazio e um script que executava um *loop* de 10000 atribuições. Verificou-se que a execução do script vazio demorava em média 137ms para ser interpretado e que o script com 10000 atribuições demorava em média 208ms para ser processado. Supondo-se que a primeira medida resulte no *overhead* do interpretador, deduz-se que o tempo real de execução da 10000 atribuições foi de 208 – 137 =71ms, o que nos leva a concluir que o tempo médio gasto em cada operação de atribuição foi aproximadamente de 71/10000 = 7,1µs. Modificando-se o script para realizar 100000 atribuições retornou um tempo médio de 751ms (751 – 137 = 684ms; 6,8µs/atribuição). Reduzindo-se para realizar 1000 atribuições retornou um tempo médio de 141ms (141 – 137 = 7ms; 7µs/transação). Conclui-se que o tempo médio de uma atribuição está na faixa de 7µs.

```
set x 0
while {$x < 10000} {
   incr x
}
```

Código 4-1: script com 10000 atribuições

Código 4-1	Tempo de execução
Script vazio	137ms
Código 4-1 com 1000 atribuições	141ms
Código 4-1 com 10000 atribuições	208ms
Código 4-1 com 100000 atribuições	684ms

Tabela 4-1: script com 10000 atribuições

Medição do atraso na troca de dados com o AF: o objetivo deste teste é medir o atraso causado pela comunicação entre *Test Engine* e AF. As mensagens são trocadas entre esses dois módulos através de três *pipes* criados com a instrução *socketpair*. Partindo da suposição que todos os *pipes* possuem a mesma velocidade de transmissão, basta medir o atraso em um deles. O canal escolhido foi o FM, utilizado para enviar comandos ao AF/FSM. Assim sendo, foram enviadas 1000 instruções FD-DATA.req ao AF com o AF/PF em modo *Ocioso*, no qual o AF responde com um FM-DISC.ind. Tomando como base de cálculo o tempo gasto para enviar um buffer de 1000 bases chega-se ao valor de 13μs/base (26550 – 13353 = 13197 ms p/ enviar 1000 bytes 1000 vezes). Descontado o tempo de uma atribuição (7μs, medido), deduz-se que o tempo necessário para transportar um byte por uma conexão é de 7μs. Na tabela são apresentados os resultados; entre parênteses estão as diferenças de cada medida em relação à medida de um buffer vazio.

```
# 10 bytes
set buffer "XXXXXXXXXX"
time {
    senddata Lower buffer

} 1000
```

Código 4-2: script para teste de conexão entre *Test Engine* e AF

Código 4-2	Tempo médio de execução
Buffer vazio	13,353 s
Buffer com 10 bytes	13,458 s (103 ms/1000 vezes, 1,03 ms/transação)
Buffer com 100 bytes	19,993 s (6640 ms/1000 vezes)
Buffer com 1000 bytes	26,550 s (13197 ms/1000 vezes)

Tabela 4-2: resultado para *Test Engine* e AF

Medição do atraso entre AF e PF: o objetivo deste teste é medir o tempo de resposta na conexão entre AF e PF. Esta conexão é feita através do nosso protocolo FTMP (escolhido como TCP/IP). A medida de atraso neste caso é feita utilizando-se o FM-CNTL.req com o AF/PF em modo *Conectado* e realizando uma troca de dados com PF em modo *loopback*. Observa-se um atraso maior aqui em relação às outras medidas já tomadas. Isso já era esperado devido à utilização de um meio físico para transporte de dados e um protocolo orientado à conexão, no nosso caso o FTMP. O atraso médio da conexão é, descontado o tempo gasto na troca de dados entre *Test Engine* e AF (100 bytes[8]) é de 37 – 6,6 ≈ 30 ms/transação.

```
openconn 1 lua 2345
sendcmd 1 Loopback

# 10 bytes buffer
time {
    senddata 1 Lower "####################"
    recvdata 1 Lower
} 1000
closeconn 1
```

[8] A escolha deste valor se deve a ser esta a ordem de tamanho de uma PDU.

Código 4-3: Medição de atraso entre AF e PF

Código 4-3	Tempo de execução
Buffer nulo	20,89 s (21 ms/transação)
Com 100 bytes	57,75 s (52,75 – 20,89 ≈ 37 ms/100 bytes)
Com 1000 bytes	63,89 s (43 ms/1000 bytes)
Com 100000 bytes	86,10 s (65 ms/100000 bytes)
Com 1000000 bytes	244,63 s (225 ms/1000000 bytes)

Tabela 4-3: Atraso entre AF e PF

Medição do atraso total: o objetivo deste teste é medir o tempo de resposta total da ferramenta. Neste caso o tempo de resposta é o tempo medido durante a execução normal dos testes. Para fazer estas medições utilizamos o Código 4-1, já apresentado no início da subseção 4.3.1. Deduz-se que o tempo médio gasto entre o PF e a IUT (incluindo o processamento necessário) é de 79.6 – (30 x 2) ≈ 9ms.

Código 4-1	Tempo de execução
Tempo de execução do script	79,6 s (1000 interações)

Tabela 4-4: atraso total da ferramenta

4.5 Resumo

Este capítulo apresentou as metodologias utilizadas para a validação de nossa ferramenta, incluindo testes de integração, testes reais e testes de desempenho com a ferramenta. Também apresentou o protocolo SNMPv3, utilizado nos testes reais.

Capítulo 5

Considerações finais

Este capítulo apresenta conclusões e contribuições do trabalho de dissertação, e discute possíveis extensões à arquitetura e à implementação da F-SOFIST.

5.1 Conclusões e Contribuições

Esta dissertação apresentou a arquitetura *Ferry-injection*, uma extensão à arquitetura *Ferry-clip* que permite a injeção de falhas por software, e uma ferramenta, a F-SOFIST, desenvolvida utilizando-se a arquitetura apresentada. A ferramenta apresentada aqui considera que os testes realizados podem ser do tipo "caixa-preta", onde o código-fonte da Implementação em Teste não está disponível.

Embora existam outras ferramentas que permitam a injeção de falhas por software (por exemplo, a ferramenta ORCHESTRA [DJM96]), geralmente elas estão voltadas unicamente para esse tipo de teste, sem dar suporte a outros testes de validação. Desta maneira, podemos dizer que esta dissertação apresenta um trabalho original de especificação e implementação de uma ferramenta para teste de protocolos tolerantes a falhas.

A principal vantagem na utilização da arquitetura *Ferry-clip* está na reutilização de código. Como o sistema está dividido em duas partes, apenas a parte do sistema associada à Implementação em Teste (PF e FIM) devem ser compilados na nova plataforma e, eventualmente, o sub-módulo AF/SIA. A própria modelagem da ferramenta permitiu que a parte mais complexa da codificação fosse associada sempre ao Sistema de Teste (*Test Engine* e AF), sistema que, salvo algum caso específico, não precisará ser compilado em outra plataforma. O Sistema de Teste foi modelado para ser o mais simples possível, agrupando e criando uma interface única (PF/SIA) para o código dependente da Implementação em Teste.

A ferramenta F-SOFIST corresponde ao Sistema de Suporte à Execução de uma arquitetura que está sendo desenvolvida conjuntamente no Instituto de Computação da UNICAMP e a Divisão de Sistemas de Solo do INPE denominada ATIFS (Ambiente de Testes e Injeção de Falhas por Software) [MAA96]. A ferramenta de geração de testes [MRS00] e ferramenta de análise de resultados [StM97], ambas mencionadas anteriormente, também fazem parte dessa arquitetura.

5.2 Avaliação dos resultados experimentais

A especificação da arquitetura *Ferry-injection* e a implementação da F-SOFIST foram feitas visando a integração do ambiente ATIFS. Alguns experimentos foram realizados para analisar o

uso dessa ferramenta em protocolos e alguns modelos foram desenvolvidos para os testes dessa natureza. O capítulo 4 mostra a aplicação da ferramenta em um protocolo real, um dos requisitos para a validação completa da ferramenta.

Devido a algumas limitações do protocolo SNMPv3 não foi possível realizar todos os testes possíveis com a ferramenta, principalmente a injeção de falhas. Isso porque esse protocolo trabalha apenas com requisições e respostas, de modo que a única classe de falhas possível de ser testada foi a classe de corrupção de dados.

Apesar desses problemas, o que motivou a escolha deste protocolo para ser testado? A resposta é simples: o objetivo dos testes é validar a ferramenta, não o protocolo. No nosso caso o protocolo escolhido apresenta recursos de tolerância a falhas que, apesar de muito simples, são também passíveis de serem testados e tais recursos foram testados com sucesso. Através dos testes escolhidos foi possível inclusive determinar o esquema de tolerância a falhas deste protocolo:

- As PDU's são validadas utilizando um código CRC;
- Caso a PDU com a solicitação de um serviço enviada pelo Agente contenha algum erro ela é sumariamente descartada pelo Gerente, fazendo com que o Agente envie uma nova solicitação;

É possível que novos testes ajudassem a compreender melhor a implementação, mas para efeito de validação da ferramenta os testes apresentados no capítulo 4 bastaram.

5.3 Extensões

Esta seção discute algumas extensões passíveis de serem aplicadas à implementação da ferramenta F-SOFIST. Embora a arquitetura *Ferry-injection* possua um alto grau de dinamismo, nem todo esse dinamismo pode ser mostrado na implementação do F-SOFIST. Para avaliar adequadamente esse dinamismo podemos discutir dois pontos de vista: extensões à arquitetura *Ferry-injection* e extensões à ferramenta F-SOFIST.

5.3.1 Extensões à arquitetura *Ferry-injection*

A arquitetura proposta apresentada nesta dissertação pode ser ampliada em alguns pontos principais:

- A arquitetura proposta considera que os testes envolvem apenas uma IUT. O ideal seria expandir a arquitetura para dar suporte à testes envolvendo mais de uma IUT, permitindo assim executar testes de interoperabilidade e testes multi-partes [CVD92]. Embora o suporte básico a testes desse gênero tenha sido implementado, algumas mudanças são necessárias no formato utilizado pelos pacotes que são trocados entre as diversas partes IUT's, além de ser necessário um sistema eficiente de gerenciamento de conexões, uma vez que diversas conexões, envolvendo diversos *Ferry-clips*, serão utilizadas.
- A arquitetura *Ferry-clip* original prevê a utilização de um canal em teste para testar uma implementação que utilize o meio físico para troca de dados (verifique a Figura 2-1). Neste caso o Injetor de Falhas deve utilizar um esquema de interceptação de pacotes diferente do utilizado na nossa implementação. Um esquema possível de ser utilizado neste caso é o apresentado pela ferramenta ORCHESTRA [DJM96]. Neste esquema uma

biblioteca modificada substitui a biblioteca de acesso ao meio físico na Implementação em Teste; no entanto tal esquema necessita acesso ao código fonte dessa implementação, o que não é necessário na atual arquitetura *Ferry-injection*. Tal requisito é discutido em detalhes na arquitetura AFIDS [SoW97a, SoW97b]

- O protocolo FTMP pode ser trocado por um protocolo mais eficiente que o utilizado na nossa implementação (TCP/IP), que é altamente eficiente nesse tipo de conexão segura, mas pode influenciar testes que envolvam sistemas de tempo real uma vez que o custo para garantir essa conexão segura é alto (PDU's grandes). Um exemplo de protocolo rápido e simples é o UDP/IP. Observe que neste caso as rotinas de recuperação de erros devem ser implementadas pelo usuário.

5.3.2 Extensões à ferramenta F-SOFIST

Esta seção discute algumas extensões que podem ser aplicadas à ferramenta F-SOFIST, além das modificações propostas na seção anterior.

- Atualmente a ferramenta possui uma única interface com o usuário, por linha de comando, onde o usuário simplesmente passa como argumento o script com os testes que se deseja aplicar, com todos os resultados sendo armazenados no arquivo de Log. Uma interface mais eficiente, possivelmente gráfica, seria muito mais clara do ponto de vista do usuário.
- Tentou-se padronizar o FTMP para utilizar a porta de comunicação número 2345 (TCP/IP). Este esquema se revela inadequado caso a ferramenta aplique testes envolvendo diversas Implementações em Teste. Neste caso a melhor solução seria um sistema de alocação dinâmico de portas e um local onde essas informações possam ser consultadas pelas partes envolvidas para troca de dados.

Apêndice A: Implementação do *Ferry clip*

Máquinas de Estado Finitas do Ferry clip

Os estados apresentados aqui correspondem aos estados possíveis encontrados durante a abertura de conexão entre AF e PF. A tabela abaixo apresenta a máquina de estados relacionada ao AF/FSM e a tabela seguinte apresenta a máquina de estados relacionada ao PF/FSM. As duas máquinas são complementares e trabalham em conjunto.

Estados e transições da AF/FSM

Estado	Evento	Ação	Próximo estado
Ocioso	FM-CONN.req	FT-CONN.req	Conectando
	FM-DISC.req	Nenhuma	Ocioso
	FM-CNTL.req	FM-DISC.ind	
	FD-DATA.req		
	FT-DISC.ind	Nenhuma	
	FT-ERROR.ind	FT-DISC.req	
Conectando	FT-CONN.cnf	FM-CONN.cnf	Conectado
	FM-DISC.req	FT-DISC.req	Ocioso
	FT-DISC.ind	FM-DISC.ind	
	FT-ERROR.ind	Exceção	
Conectado	FM-CNTL.req	FT-DATA.req	Conectado
	FD-DATA.req		
	FT-DATA.ind	FD-DATA.ind	
	FT-DISC.ind	FM-DISC.ind	Ocioso
	FT-ERROR.ind	Exceção	

O tratamento de exceções envolve encerrar a conexão entre AF e PF, gravar uma mensagem no registro de Log indicando (se possível) a causa do erro e encerrar a execução da ferramenta.

Estados e transições da PF/FSM

Estado	Evento	Ação	Próximo estado
Ocioso	FT-CONN.ind	FT-CONN.rsp	Conectado
		FT-DISC.req	Ocioso
	FT-DISC.ind	Nenhuma	
	FT-ERROR.ind	Exceção	
	FD-DATA.req	Nenhuma	
Conectado	FT-DISC.ind	Nenhuma	

	FT-ERROR.ind	Exceção	
	FT-DATA.ind	FT-DATA.req (1)	Conectado
		FD-DATA.ind (2)	
		Ações de controle (3)	
	FD-DATA.req	FT-DATA.req (4)	
		Nenhuma (1)	

(1) Modo *loopback* (todos os dados que chegam ao PF são enviados de volta)

(2) Chegou um dado que deve ser enviado à IUT

(3) Solicita um comando à AF/FSM

(4) IUT solicita envio de dados ao AF

Descrição dos eventos envolvidos

Os eventos aqui apresentados correspondem aos eventos trocados entre os módulos.

Descrição dos Eventos

Estado	Descrição
FM-CONN.req	Solicitar uma nova conexão
FM-CONN.ind	Chegou pedido de conexão
FM-CONN.rsp	Aceitar conexão
FM-CONN.cnf	Conexão foi aceita
FM-DISC.req	Solicitar uma desconexão
FM-CNTL.req	Enviar um comando à FSM remota
FD-DATA.req	Enviar um dado
FT-ERROR.ind	Erro
FT-CONN.cnf	Conexão aceita
FT-DISC.ind	Conexão encerrada
FT-DATA.ind	Receber um dado

Apêndice B: Extensão do interpretador TCL

Comandos adicionados ao interpretador TCL

Os comandos apresentados aqui correspondem aos comandos adicionados ao interpretador TCL, com a finalidade específica de permitir o controle do Sistema em Teste.

Comando	Parâmetros	Descrição
`senddata`	Interface, Comandos	Transmite a seqüência de bytes à IUT utilizando o canal de dados do *Ferry* (utilizando FD-asp's), aguardando a confirmação de envio durante *timeout* milissegundos. Retorna zero em caso de sucesso e 1 em caso de erro e o erro é indicado no arquivo de Log.
`recvdata`	Interface, Comandos	Bloqueia o script durante *timeout* milissegundos até a chegada de uma seqüência de dados proveniente da IUT. Retorna zero em caso de sucesso e 1 em caso de erro e o erro é indicado no arquivo de Log.
`sendcmd`	Instruções	Envia uma instrução de comando ao *Ferry* (utilizando FM-asp's), aguardando a confirmação do comando durante *timeout* milisegundos. Retorna zero em caso de sucesso e 1 em caso de erro e o erro é indicado no arquivo de Log.
`sendFcmd`	Instruções	Envia uma instrução de comando ao FIM (utilizando FF-asp's), aguardando a confirmação do comando durante *timeout* segundos. Retorna zero em caso de sucesso e 1 em caso de erro e o erro é indicado no arquivo de Log.
`reset`		Reinicializa o injetor de falhas e indica o início de um novo caso de teste, aguardando a confirmação do comando durante *timeout* milisegundos. Retorna zero em caso de sucesso e 1 em caso de erro e o erro é indicado no arquivo de Log.

Variáveis especiais

Aqui são apresentadas as variáveis globais utilizadas no interpretador TCL e que afetam o funcionamento da ferramenta.

Variável	Tipo	Descrição
`timeout`	Inteiro	Tempo de espera do script até que seja acusado um erro.
`errorStop`	Boolean	Se setada interrompe a execução do script caso alguma das funções retorne erro; o valor default é zero.

Glossário

AF: *Active Ferry* ou Ferry Ativo. É o objeto da arquitetura *Ferry clip* que faz o papel de cliente na conexão entre AF e PF.

ASP: *Abstract Service Primitives* ou Primitivas de Serviço Abstrato. São os eventos gerados pela implementação tal como observados nos Pontos de Observação e Controle (PCO's).

FD-asp: *Ferry Data*. Primitivas de transporte de dados cia *Ferry clip*.

FF-asp: *Ferry Fault*. Primitivas de transporte de instruções ao FIM.

FIC: *Fault Injection Controller* ou Controlador do Injetor de Falhas. Módulo responsável pelo controle remoto do Módulo Injetor de Falhas (FIM).

FIM: *Fault Injection Module* ou Módulo Injetor de Falhas. Módulo responsável pela injeção de falhas.

FM-asp: *Ferry Management*. Primitivas de transporte de comandos ao *Ferry clip*.

FSM: *Finite State Machine*. Módulo do *Ferry clip* responsável pelo controle de conexão entre AF e PF.

FT-asp: *Ferry Transport*. Primitivas de transporte entre AF e PF.

FTMP: *Ferry Transfer Medium Protocol* ou Protocolo de Transferência entre Ferry's. Designação genérica do protocolo de comunicação confiável escolhido para realizar a troca de dados entre AF e PF.

IUT: *Implementation Under Test* ou Implementação em Teste. É a implementação do protocolo que se encontra sob teste.

LMAP: *Lower Mapping Module*. Módulo do *Ferry clip* responsável pela troca de dados entre AF e PF.

LT: Lower Tester ou Testador Inferior. Responsável pela interceptação de Primitivas de Serviço Abstrato (ASP's) provenientes da interface inferior da implementação.

PCO: *Point of Control and Observation* ou Ponto de Observação e Controle. São os locais onde são coletados e aplicados os dados de teste.

PDU: *Protocol Data Unit* ou Unidade de Dados do Protocolo. É a unidade mínima de informação trocada entre duas implementações de um protocolo.

PF: *Passive Ferry* ou Ferry Passivo. É o objeto da arquitetura *Ferry clip* que faz o papel de servidor na conexão entre AF e PF.

SIA: *Service Interface Adapter*. Módulo do *Ferry clip* responsável pelo transporte de PDU's através da conexão entre AF e PF;

SUT: *System Under Test* ou Sistema em Teste. Sistema no qual serão aplicados os testes.

TCL: *Toolkit Control Language*. Linguagem desenvolvida por J. K. Ousterhout [Ous94] para execução de comandos.

Test Coordination Procedures: são os procedimentos de teste tal como aplicados nos testadores superior e inferior.

TE: *Test Engine* ou Máquina de Testes. Responsável pela geração e coordenação dos testes.

TS: *Test System* ou Sistema de Teste. Sistema que integra a Máquina de Testes e o AF.

UT: *Upper Tester* ou Testador Superior. Responsável pela interceptação de Primitivas de Serviço Abstrato (ASP's) provenientes da interface superior da implementação.

BIBLIOGRAFIA

[AAC90] Arlat, J; Aguera, M; Crouzet, Y; Fabre, J C; Martins, E; Powell, D. *Experimental Evaluation of the Faults Tolerance of an Atomic Multicast System*, IEEE Transactions on Reliability, vol 39, no 4, October, 1990

[Blo97] Blough, D M; Torii, T. *Fault-Injection-Based Testing of Fault-Tolerant Algorithms in Message-Passing Parallel Computers,* Digest of Papers from FTCS-27 (27th International Symposium on Fault-Tolerant Computing), pg 258-267, IEEE Computer Society, 1997, ISBN 0-8186-7831-3, June 24-27, 1997; Seattle, Washington, USA

[Car95] Carreira, J F V. *Software Fault Injection in parallel Systems*, Dissertation submitted to the Univ. of Coimbra in partial fulfillment of the requirements for the degree: Master in Systems and Technology of information, area of system architecture, Julho, 1995; Coimbra, Portugal

[CLP89] Chanson, S T; Lee, B P; Parakh, N J; Zend, H X. *Design and Implementation of a Ferry Clip Test System*, Prox. 9th IFIP Symposium on Protocol Specification Testing & Verification, June 1989; Enschede, Netherlands

[CMS94] Carreira, J F V; Madeira, H; Silva; J G. *Assessing the Effects of Communication Faults on Parallel Applications*, Proceedings of IPDS'95, pp 214-223, April 24-26, 1994; Erlangen, Germany

[Cri91] Cristian, F. *Understanding Fault-Tolerant Distributed Systems*, Communications of the ACM, vol. 34, no. 2, February 1991

[CVD92] Chanson, S T; Vuong, S; Dany, H. *Multi-party and interoperability testing using the Ferry Clip approach*, Computer communications vol 15, no 3, April 1992

[DJM95] Dawson, S; Jahanian, F; Mitton, T. *A Software Fault Injection Tool on Real-Time Mach*, Proc. IEEE Real-Time Systems Symposium, December 1995

[DJM96] Dawson, S; Jahanian, F; Mitton, T. *ORCHESTRA: A Fault Injection Environment for Distributed Systems*, University of Michigan Technical Report CSE-TR-318-96, EECS Department, 1996

[DJM97] Dawson, S D; Jahanian, F; Mitton, T; Tung, T L. *Testing of Fault-Tolerant and Real-Time Distributed Systems via protocol Fault Injection*, FTCS-27, The 27th Annual International Symposium on Fault-Tolerant Computing, June 24-27, 1997; Seattle, Washington, USA

[DWJ95] Dawson, S; Jahanian, F. *Probing and Fault Injection of Protocol Implementations*, Proc. Int. Conf. on Distributed Computer Systems, May 1995

[ECL92] Echtle, K; Leu, M. *The EFA Fault Injector for Fault-Tolerant Distributed System Testing*, Fault-Tolerant Parallel and Distributed Systems, Conf. Proc., IEEE Press, pp. 28-35, 1992

[Fis98] Fischer, K P. *Position Statement to IWPTS 1989*, 2nd International Workshop on Protocol Test Systems, October 3-6, 1989; Berlin (West), Germany

[Gri99] Griffin, J L. *Testing Protocol Implementation Robustness*, 29th Annual International Symposium on Fault-Tolerant Computing, June 15-18, 1999; Madison/Wisconsin, USA

[HRS93] Han, S; Rosenberg, H A; Shin, K G. *DOCTOR: An Integrated Software Fault Injection Environment*, University of Michigan Technical Report CSE-TR-192-93, EECS Department, 1993

[HTI97] Hsueh, M-C; Tsai, T K; Iyer, R K, *Fault Injection Techniques and Tools*, Computer, Vol. 30, No. 4, pp 75-82, April 1997

[ISO88] ISO/IS 9646. *OSI Conformance Testing Methodology and Framework – Part 1: General Concepts* Edit Meeting, Stockolm, 14-21 July 1988

[Kal94] Kao, W I; Iyer, R K. *DEFINE: A Distributed Fault Injection and Monitoring Environment*, Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, June 1994.

[Lap91] Laprie, J C. *Dependability concepts*, Delta-4: A Generic Architecture for Dependable Distributed Computing, Project 818/2252 – Delta 4 – Volume 1, 1991; Berlin, Germany

[Lap95] Laprie, J C. *Dependability – Its Attributes, Impairments and Means*, Predictably Dependable Computing Systems, chapter II, pages 27-40, Springer Verlag, 1995.

[LCV95] Loureiro, Antonio A F; Chanson, Samuel T; Vuong, Son T. *A Critical Assessment of Testability in Communication Protocols,* Anais do 13° Simpósio Brasileiro de Redes de Computadores, UFMG/Belo Horizonte (MG), Maio 1995

[MAA96] Martins, E; Ambrósio, A M; Araujo, M R. *A framework for developing a software-based fault injection tool for the test of communication systems*, ISACC IV – Fourth International Symposium on Applied Corporate Computing, October 30-November 1, 1996; Monterrey/NL, México

[Mar93] Martins, E. *Validação experimental da tolerância a falhas: a técnica de injeção de falhas*, V Simpósio de Computadores Tolerantes a Falhas – Mini-cursos, 4-6 outubro, 1993; ITA - São José dos Campos/SP, Brasil

[MRS00] Martins, E; Ambrósio, A M; Sabião, S B. *ConData: a Tool for Automating Specification-based Test Case Generation for Communiaction Systems*, Proceedings of the 33rd Hawaii International Conference on System Sciences, January, 2000; Hawaii, USA

[Ous94] Ousterhout, J K. *Tcl and the Tk Toolkit*, Addison-Wesley, 480 pages (ISBN 020163337X), Published April 1994

[Por89] Porto, I J. *Testabilidade*, III Simpósio de Computadores Tolerantes a Falhas – Mini-cursos, 20-22 setembro, 1989; Rio de Janeiro/RJ, Brasil

[RFC2570] Request for Comments no. 2570. *Introduction to Version 3 of the Internet-standard Network Management Framework*

[SHR97] Stott, D T; Hsueh, M C; Ries, G L; Iyer, R K. *Dependability Analysis of a Commercial High-Speed Network*, Digest of Papers from FTCS-27 (27th International Symposium on Fault-Tolerant Computing), pg 248-257, IEEE Computer Society, 1997, ISBN 0-8186-7831-3, June 24-27, 1997; Seattle, Washington, USA

[SHR97] Stott, D T; Hsueh, M C; Ries, G L; Iyer, R K. *Dependability Analysis of a Commercial High-Speed Network*, FTCS-27, The 27th Annual International Symposium on Fault-Tolerant Computing, June 24-27, 1997; Seattle, Washington, USA

[SoW97a] Sotoma, I; Weber, T S. *AFIDS – Arquitetura para Injeção de Falhas em Sistemas Distribuídos*, Anais do XV Simpósio Brasileiro de Redes de Computadores, 19-22 maio, 1997; São Carlos/SP – Brasil

[SoW97b] Sotoma, I; Weber, T S. *Desenvolvimento de uma Ferramenta para Injeção de Falhas em Sistemas Distribuídos baseada em AFIDS*, Anais do VII Simpósio de Computadores Tolerantes a Falhas, 2-4 julho, 1997; Campina Grande/RS – Brasil

[StM97] Stefani, M R; Martins, E. *Análise de Traço e Geração de Diagnósticos para Testes Baseados em Injeção de Falhas por Software*, Anais do VII Simpósio de Computadores Tolerantes a Falhas, 02-04 julho, 1997; Campina Grande/PB – Brasil

[SVS88] Segall, Z; Vrsalovic, D; Siewiorek, D; Yaskin, D; Kownacki, J; Barton, J; Rancey, D; Robinson A; Lin, T. *FIAT - Fault Injection-based Automated Testing Environment*, 18th Int. Symposium on Fault-Tolerant Computing (FTCS-18), 1988; Tokyo, Japan

[TsI95] Tsai, T K; Iyer, R K. *FTAPE: A Fault Injection Tool to measure Fault Tolerance*, Proceedings of AIAA Computing in Aerospace 10, March 28-30, 1995; San Antonio/TX, USA