UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALEXANDRE FELIN GINDRI

Estudo de Injeção de Falhas para a Máquina Virtual do Sistema Android

Trabalho de Graduação.

Profa. Dra. Taisy Silva Weber
Orientador

Porto Alegre, Julho de 2011.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do ECP: Prof. Sérgio Luis Cechin
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

À UFRGS, pelos ensinamentos que levarei por toda a vida.

Aos colegas, pelo companherismo em todos os momentos desta grande caminhada.

À Taisy, orientadora, pela oportunidade e a ajuda dada neste trabalho.

Aos meus familiares, pela torcida e energia positiva.

À Taciane, namorada, pela motivação direta e indireta, pela compreensão e pela cumplicidade.

Aos meus pais, coautores de tudo que faço, pela paciência, pelo exemplo e pelo amor incondicional durante todos esses anos.

SUMÁRIO

AGRADECIMENTOS **3**
SUMÁRIO **4**
LISTA DE ABREVIATURAS **6**
LISTA DE FIGURAS **7**
LISTA DE TABELAS **8**
RESUMO **9**
1 INTRODUÇÃO **10**
2 ANDROID **12**
2.1 Dalvik Virtual Machine – DVM 13
3 INJEÇÃO DE FALHAS **14**
3.1 Injeção de Falhas em Java 14
3.2 Ativação das falhas 15
4 AMBIENTES MÓVEIS **17**
5 MODELOS DE FALHAS DE COMUNICAÇÃO **19**
5.1 Modelo de Perda Sem Memória de Bernoulli 19
5.2 Modelo de Perdas em Rajada de Gilbert Simplificado 19
5.3 Modelo de Perdas em Rajada de Gilbert 20
5.4 Modelo de Perdas em Rajada de Gilbert-Elliott 20
5.5 Cadeia de Markov com Quatro Estados 21
6 AMBIENTE DE DESENVOLVIMENTO DO ANDROID **23**
6.1 Instalação do Ambiente 23
6.2 Emulador 25
6.3 Construção e Utilização de SDKs Próprios 26
7 IMPLEMENTAÇÃO DO INJETOR DE FALHAS UDP **30**
7.1 Classe DatagramSocket no Android e Principio do Funcionamento do Injetor 30
7.2 Detalhamento da Implementação do Injetor de Falhas UDP 32
8 TESTES DO INJETOR DE FALHAS UDP **38**
8.1 Aplicação Cliente – Servidor 38
8.2 Testes da Simulação dos Modelos 39
8.3 Influência no Desempenho 44
8.4 Influência no Timeout 45

9 CONCLUSÃO..46
REFERÊNCIAS...47
ANEXO – ARTIGO DA PROPOSTA DESTE TRABALHO.............................49
APÊNDICE A: CÓDIGO PARCIAL DO DATAGRAM SOCKET COM O INJETOR DE FALHAS...57
APÊNDICE B: CÓDIGO DA APLICAÇÃO CLIENTE......................................64
APÊNDICE C: CÓDIGO DA THREAD DA APLICAÇÃO SERVIDOR.........65
APÊNDICE D: MANIFEST DA APLICAÇÃO SERVIDOR..............................68

LISTA DE ABREVIATURAS

ADT – Android Development Tool

CPU – Central Processing Unit

DDMS – Dalvik Debug Monitor Server

DVM – Dalvik Virtual Machine

IDE – Integrated Development Environment

IP – Internet Protocol

JDK – Java Development Kit

OHA – Open Handset Alliance

RAM – Read Only Memory

SDK – Software Development Kit

UDP – User Datagram Protocol

VM – Virtual Machine

VoIP – Voice Over IP

WLAN – Wireless Local Area Network

LISTA DE FIGURAS

FIGURA 3.1: CANAIS DA FAIXA DE FREQUÊNCIA 2.4 GHZ......................17

FIGURA 4.2: CADEIA DE MARKOV REPRESENTANDO O MODELO DE BERNOULLI...19

FIGURA 4.3: CADEIA DE MARKOV REPRESENTANDO O MODELO DE GILBERT SIMPLIFICADO...20

FIGURA 4.4: CADEIA DE MARKOV QUE REPRESENTA O MODELO DE GILBERT...20

FIGURA 4.5: MODELO DE GILBERT-ELLIOTT...21

FIGURA 4.6: FLUXO DE TRANSMISSÃO PARA H IGUAL À 0,5.................21

FIGURA 4.7: CADEIA DE MARKOV DE QUATRO ESTADOS [VOIP-T] [SALSANO 2009]...21

FIGURA 8: INTERFACE GRÁFICA DO EMULADOR DO ANDROID.........26

FIGURA 9: O CONSTRUTOR DA CLASSE DATAGRAMSOCKET A) ORIGINAL E B) APÓS A IMPLEMENTAÇÃO DO INJETOR.............................31

FIGURA 10: O MÉTODO RECEIVE() A) ORIGINAL E B) APÓS A IMPLEMENTAÇÃO DO INJETOR...31

FIGURA 11: O MÉTODO SEND() A) ORIGINAL E B) APÓS A IMPLEMENTAÇÃO DO INJETOR...32

FIGURA 12: A) REPRESENTAÇÃO GRÁFICA. B) MATRIZ DE TRANSIÇÕES...33

FIGURA 13: GRÁFICO PARA EXECUÇÃO COM MODELO DE BERNOULLI...40

FIGURA 14: GRÁFICO PARA EXECUÇÃO COM MODELO DE GILBERT SIMPLIFICADO...41

FIGURA 15: GRÁFICO PARA EXECUÇÃO COM MODELO DE GILBERT. 42

FIGURA 16: GRÁFICO PARA EXECUÇÃO COM MODELO DE GILBERT-ELLIOTT...43

FIGURA 17: GRÁFICO PARA EXECUÇÃO COM MODELO DE GILBERT. 44

LISTA DE TABELAS

TABELA 1: TABELA DE PROBABILIDADES PARA O MODELO DE BERNOULLI...34

TABELA 2: TABELA DE PROBABILIDADES PARA O MODELO DE GILBERT SIMPLIFICADO..35

TABELA 3: TABELA DE PROBABILIDADES PARA O MODELO DE GILBERT...35

TABELA 4: TABELA DE PROBABILIDADES PARA O MODELO DE GILBERT-ELLIOT...36

TABELA 5: TABELA DE PROBABILIDADES PARA O MODELO DE QUATRO ESTADOS [VOIP-T] [SALSANO 2009]..36

TABELA 6: TABELA DE PROBABILIDADES PARA A CADEIA DE MARKOV DE QUATRO ESTADOS...37

TABELA 7: RESULTADO DAS EXECUÇÕES SEM INJETOR, COM O INJETOR DESABILITADO E COM INJETOR HABILITADO............................45

TABELA 8: RESULTADO DAS EXECUÇÕES SEM INJETOR, COM O INJETOR DESABILITADO E COM INJETOR HABILITADO............................45

RESUMO

O Android é um sistema de desenvolvimento para aparelhos móveis que vem mantendo a atenção tanto da comunidade tecnológica como dos consumidores em geral. Este sistema de desenvolvimento permite a fácil criação de aplicações que, assim como as aplicações nativas, utilizem toda a capacidade dos aparelhos. Com essa facilidade, surgem diversas aplicações, provenientes de diferentes desenvolvedores. Para se ter confiabilidade nesse cenário é desejável poder testar a tolerância dessas aplicações a falhas. Este trabalho explora a possibilidade de implementar injetores de falhas para o Android no nível da máquina virtual, onde rodam todas as aplicações de usuário, por meio da implementação de um injetor de falhas de comunicação UDP.

Palavras-chave: Android, injeção de falhas, máquina virtual.

1 INTRODUÇÃO

Nos últimos anos os computadores portáteis tem ganho importância frente aos computadores de mesa. A possibilidade de utilizar um notebook em lugar de um desktop em casa e outro na oficina, permite ao funcionário ter seus arquivos e programas a disposição, sem precisar configurar e manter dois sistemas.

As aplicações mais comuns para os usuários, relacionadas a comunicação (e-mail, bate-papo, redes sociais), entretenimento (jogos, conteúdo multimídia como músicas e vídeos), pesquisas e compras, entre outros, já são suportadas por dispositivos moveis de pequeno porte, como são os handhelds: palm tops, smartphones e aparelhos semelhantes. Para o uso profissional, é extremamente interessante para um técnico poder acessar os manuais de um produto quando em campo, ou para um médico obter a ficha de um paciente durante o atendimento domiciliar.

O avanço das tecnologias permite agregar maior poder computacional em dispositivos menores. Porém ainda há alguns pontos que comprometem o desempenho para tais dispositivos. As limitações de tamanho e consumo de potência podem trazer como consequência o uso de componentes com menor tolerância a falhas (menos dispositivos redundantes) e maior taxa de erros (menor tensão nos componentes eletrônicos é mais facilmente modificada por interferência).

Quando colocado em um ambiente móvel, com interferências entre sinais e movimentação, o desafio de se fazer uma aplicação que não apresente defeitos, ou seja, que tenha dependabilidade [AVIZIENIS 2004], torna-se ainda maior. É necessário incluir mecanismos de tolerância a falhas que alcancem esse objetivo quando em operação no ambiente real.

É incontrolável recriar todos os cenários e depender da ocorrência natural de falhas. Para simular a ocorrência das falhas, e assim conseguir testar a tolerância do sistema, é utilizado o método de injeção de falhas [HSUEH 1997], que é um dos mais consolidados na literatura. O método permite que sejam provocados os padrões de falhas semelhantes aos que podem ocorrer durante a operação do dispositivo.

Um sistema que tem crescido de forma notável quanto a utilização nos novos dispositivos móveis é o Android. Este é um sistema desenvolvido e publicado pela Google, e posteriormente pela Open Handset Alliance, como *open source* (código aberto), de forma a permitir que desenvolvedores possam criar aplicações próprias e vendê-las num mercado mundial. Recentemente este mercado, que é suportado pelo próprio projeto Android, incluiu, entre outros 20 países, o Brasil.

Desenvolvedores que queiram criar aplicações tolerantes a falhas precisam de um método de teste para descobrir os pontos críticos do projeto. Usuários que adquirem uma aplicação de uma fonte desconhecida podem eventualmente desejar testar se esta cumpre com os requisitos de dependabilidade que se espera dessa aplicação.

As aplicações de usuário do Android rodam em uma máquina virtual própria do sistema, a Dalvik Virtual Machine, que é uma máquina otimizada para sistemas com limitações de memória, processamento e bateria. Essa máquina é baseada na máquina virtual Java, mas diferencia-se quanto ao formato dos bytecodes, não tendo suporte ao uso de mecanismos de instrumentação usualmente utilizados para injeção de falhas, como JVMTI e Javassist.

A pesar de ser um sistema móvel, a comunicação no Android utiliza a mesma pilha de protocolos IP [TANENBAUM 1996] que outros tipos de sistemas. Esses protocolos são implementados em nível do kernel do sistema, e acessados pelas aplicações através de sockets. Um dos protocolos mais utilizados é o UDP, pois possui um ótimo desempenho, além de permitir comunicação multicast. Em compensação, o protocolo não é confiável, pois não prevê nenhum método de detecção e recuperação de falhas, deixando essas tarefas para serem implementadas pela aplicação, caso necessário.

O objetivo deste trabalho é realizar um estudo quanto a possibilidade de se injetar falhas no nível da máquina virtual do Android, analisando as dificuldades, a flexibilidade e a intrusividade do injetor no desempenho das aplicações.

Foram injetadas falhas na comunicação UDP, modificando a classe *DatagramSocket* da máquina virtual para simular falhas de acordo com os modelos mais comuns de perda de pacotes. Utilizando uma aplicação Cliente – Servidor, o injetor de falhas implementado foi testado de forma qualitativa e quantitativa.

2 ANDROID

O projeto foi iniciado pela Android Inc., e continuado pela Google em 2005, ao adquirir a empresa. O objetivo era fazer uma plataforma de telefone flexível, aberta e de fácil migração para fabricantes. Desde 2007 o projeto pertence a Open Handset Alliance [OHA], um consórcio de varias companhias (incluindo a Google), e foi anunciado que o Android seria o seu primeiro produto: uma plataforma para sistemas móveis construída sobre o Kernel Linux 2.6. Desde outubro de 2008 o código foi publicado sob a Licença Apache (Apache License), permitindo que os desenvolvedores ofereçam extensões proprietárias, sem a necessidade de liberação do código.

No segundo trimestre de 2010, segundo pesquisa divulgada pela empresa Canalys, 34% dos smartphones no mercado americano executam Android, e o crescimento do número de aparelhos que usam Android no mundo nesse mesmo período foi de 886% [DOBLER 2010]. Em setembro de 2010 o Android anunciou em seu site que o Android Market, um sistema de mercado onde os desenvolvedores podem vender suas aplicações aos usuários, adicionou suporte a 20 novos países, incluindo o Brasil. Com isso o Android Market oferece suporte para desenvolvedores de 29 países e usuários consumidores de 32 países.

Esse rápido crescimento é também devido a facilidade de começar a desenvolver aplicações. O sistema de desenvolvimento (SDK) do Android é simples de ser instalado e com abundante documentação. Os requisitos pré instalação são o Java SDK (JDK), e é fortemente recomendada a instalação do Eclipse IDE junto com um plugin Android Development Tool (ADT), todos gratuitos. Também é aconselhado baixar a versão para iniciantes do Android, que contém as ferramentas básicas, podendo ser usado para posteriormente baixar as demais ferramentas.

As aplicações são desenvolvidas em linguagem Java, e cada uma é executada no Android em um processo Linux separado. Cada aplicação tem também uma instância da máquina virtual utilizada, chamada Dalvik Virtual Machine (DVM), que é uma máquina semelhante a Java, mas com otimizações de memória e utilização de recursos em aparelhos móveis. Não é considerada uma máquina virtual Java pois ela não opera o mesmo bytecode, mas ao invés disso o bytecode é transformado por uma ferramenta do Android SDK no formato interpretado pela DVM.

O sistema foi desenvolvido para permitir que as aplicações de terceiros utilizem todos os recursos disponíveis do dispositivo da mesma forma que as aplicações nativas. Além disso, as aplicações podem fazer uso de elementos de outras aplicações, os chamados componentes. O sistema inicia um processo dessa aplicação e instancia os objetos necessários desse componente que será chamado.

Pela perspectiva do desenvolvedor, existem muitas possibilidades para explorar os recursos que o aparelho tem a oferecer, possibilitando a criação de aplicações complexas. Ainda, com a possibilidade de usar componentes de outras aplicações, dos

quais não se possui o código, pode se ter uma fonte de erros que comprometam o funcionamento na presença de falhas.

Pela perspectiva do usuário, as fontes que disponibilizam as aplicações (empresas ou desenvolvedores independentes) são diversas, e não necessariamente confiáveis.

Em ambos os casos é desejável um mecanismo de teste que permita validar as aplicações e garantir o nível de dependabilidade esperado destas.

2.1 Dalvik Virtual Machine – DVM

O projeto do Android foi feito para sistemas com recursos de memória, processamento e consumo limitados. Por esse motivo a utilização de uma máquina virtual convencional não seria uma opção viável.

Criada por Dan Bornstein, a Dalvik Virtual Machine é a máquina virtual que roda no Android. Esta máquina foi desenvolvida com otimizações de uso de CPU, menor gasto de memória e com otimizações para rodar mais de uma instância.

A diferença de máquinas virtuais Java, a DVM não roda bytecodes Java tradicionais. No SDK do Android é utilizada a ferramenta dx, que converte os arquivos de bytecode .class contidos em um arquivo .jar para o formato do arquivo da DVM de formato .dex. Nesse formato, informações repetidas, como por exemplo assinaturas de funções referenciadas de outros arquivos, são minimizadas com a utilização de vários apontadores.

Outra diferença significativa é que a DVM utiliza uma arquitetura baseada em registradores, em lugar de uma pilha de dados. Com essa mudança houve uma diminuição no número de instruções e de unidades de código, e embora haja um aumento no número de bytes no stream de instruções, os bytes são consumidos aos pares.

3 INJEÇÃO DE FALHAS

Existem dois tipos principais de injetores de falhas: os de hardware e os de software. Os implementados em hardware são módulos acoplados que não utilizam recursos do sistema e assim não afetam seu desempenho. Contudo, esse tipo de injetor é geralmente custoso, e a sua utilização é restrita a quantidade de pontos de inserção. Injetores de falhas em software, por outro lado, tem somente o custo de ser implementado e são mais flexíveis, embora tenham maior interferência [ACKER 2010].

O mecanismo injetor de falhas deve, preferencialmente, minimizar a intrusividade no sistema, diminuindo o impacto no desempenho normal deste. Se o injetor influenciar no desempenho de uma aplicação, ao ser retirado implica em um sistema diferente que pode, por consequência, apresentar defeitos diferentes.

Suponha-se que para testar um módulo que processa uma série de dados e coloca cada resultado em uma fila, é adicionado um módulo de injeção de falhas, fazendo com que o processamento tenha o tempo duplicado; ao se retirar este módulo de injeção de falhas a fila irá receber dados no dobro da taxa, podendo eventualmente ficar cheia e apresentar defeito.

Já existem trabalhos onde foi realizado o porte de injetores de falhas no nível do Kernel do Android, e estudos de modelos de injeção de falhas para a plataforma. Este trabalho tem como objetivo estudar a possibilidade de implementar um injetor de falhas a nível da máquina virtual.

3.1 Injeção de Falhas em Java

Para plataforma Java, existem trabalhos que apresentam injetores de falhas, como JACA [MARTINS 2002], FIONA [JACQUES-SILVA 2004], Comform [MENEGOTTO 2010] e FIRMI [VACARO 2006]. As técnicas mais utilizadas para execução do código injetor de falhas na máquina virtual Java serão apresentadas a seguir.

3.1.1 Javassist

Javassist é uma biblioteca de classe para edição de bytecode Java. Permite que classes sejam criadas durante a execução e a modificar um arquivo de classe quando a máquina virtual for carregá-la.

O pacote Javassist provê classes e métodos que permitem obter containers com o pool (repositório) de classes, recuperar arquivos de classes específicas e seus bytecodes, realizar modificações em classes e salvá-las.

Para injetar falhas de comunicação UDP em uma aplicação utilizando Javassist, pode ser carregada a classe java.net.DatagramSocket e modificar os métodos de envio e recepção para métodos que realizem a injeção das falhas.

Como as alterações são feitas a nível de bytecodes Java, o pacote Javassist não pode ser usado diretamente na DVM.

3.1.2 JVMTI

A JVMTI é uma interface de programação usada em ferramentas de desenvolvimento e monitoração. Permite inspecionar o estado e controlar a execução de aplicações em máquinas virtuais Java.

O JVMTI tem um componente cliente chamado de agente, sendo este uma biblioteca dinâmica que recebe notificações de acontecimento de eventos. Dentre as notificações, o agente pode receber a notificação da carga de uma classe, e chamar funções para alterar o estado da máquina virtual.

No injetor FIONA, o agente JVMTI recebe a notificação da carga da classe java.net.DatagramSocket na inicialização da máquina virtual, e antes da carga, substituindo a imagem da classe por uma imagem instrumentada. A aplicação chama de forma transparente os recursos da classe, não precisando de modificações no código fonte.

O implementação de JVMTI em diferentes máquinas virtuais é suportado pela Oracle, que adquiriu a Sun, criadora da plataforma Java. Para a máquina virtual do Android não há uma implementação de JVMTI.

3.1.3 Modificações no código fonte do sistema

Uma alternativa viável é a modificação do código da base sobre a qual o sistema roda. No caso do Android, podem ser feitas alterações tanto no kernel, como demonstrado no porte do injetor Firmament [DOBLER 2010], como na máquina virtual, sobre a qual rodam todas as aplicações.

O método de edição do código do sistema consiste em inserir trechos de código que simulem a ocorrência de falhas e, após, retomar a operação normal com a presença da falha. Essa injeção de uma falha pode ser provocada, por exemplo, perturbando o valor de variáveis, desviando fluxos de execuções ou adicionando atrasos em pontos estratégicos.

Utilizar o método de edição do código fonte do sistema normalmente se mostra de simples implementação, além de ser uma alternativa para processadores que não tenham hardware de depuração apropriados para usar ferramentas de inserção em execução [LOOKER 2005]. Outra vantagem é não ser necessário ter disponível o código da aplicação que será testada.

A edição do código fonte do sistema tem desvantagens. O método pode não ser válido para a certificação de sistemas, já que os sistemas com e sem o injetor podem ser considerados sistemas diferentes. A pesar disso, é possível implementar injetores com baixa intrusividade e que podem ser utilizados em qualquer aplicação que o sistema original suporta, e ter uma boa estimativa do desempenho da aplicação quando em um cenário com a presença de falhas.

Foi utilizada neste trabalho o método de modificar o código fonte, pois além de não ser possível fazer o porte de injetores que utilizem Javassist ou JVMTI, este método causa pouca alteração no desempenho do sistema como um todo e permite um bom aprendizado quanto a modificações e geração do sistema.

3.2 Ativação das falhas

Quanto ao controle da inserção de uma falha em um sistema, normalmente são utilizados critérios temporais, estabelecendo uma relação de tempo entre a injeção de cada falha, ou sequenciais, onde a falha é inserida após um número de execuções de um trecho. Podem ser utilizados diferentes padrões para decidir quando invocar o código que produzirá a falha, podendo ser de forma pseudoaleatória, seguindo algum modelo, ou que seja controlada diretamente pelo usuário testador do sistema.

4 AMBIENTES MÓVEIS

A operação de dispositivos em ambientes móveis caracteriza-se pela computação com recursos limitados, principalmente por tamanho e consumo de potência, e pela comunicação numa rede não cabeada, com movimentação relativa entre os dispositivos. O padrão 802.11 é o que determina os padrões de implementação das redes sem fio (WLAN).

Com o aumento da utilização de equipamentos eletrônicos que operam ou emitem ondas nas mesmas faixas de frequências, aumenta também o problema da interferência entre sinais. Por exemplo, a faixa de frequência de 2,4 a 2,4835 GHz, onde não é requerida autorização, torna-se problemática para estabelecimento de redes WLAN [BRANQUINHO 2005]. Em zonas onde há grande concentração de redes 802.11, alguns usuários chegam até a alterar o firmware dos seus roteadores para acessar outras faixas [BRAY] fora da regulamentada pelo seu país ou região [WIKIPEDIA].

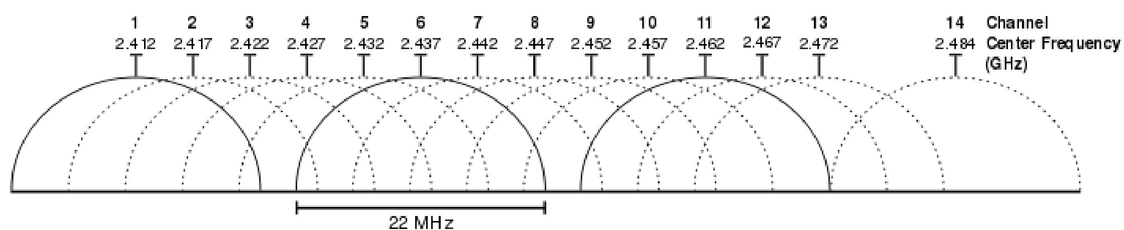

Figura 3.1: Canais da faixa de frequência 2.4 Ghz.

A Figura 3.1 mostra o espectro de canais presente na faixa de frequência de 2.4 GHz. Cada canal tem largura de 22MHz, mas os centros são espaçados por apenas 5MHz. No Brasil, são permitidos todos os canais de 1 a 13 [MORIMOTO 2008]. O canal 14 é somente utilizado no Japão.

Somado ao problema da interferência está o grande dinamismo presente em ambientes móveis [ROCHA 2007]. A alteração na distância dos dispositivos devido a movimentação gera mudanças na atenuação dos sinais, nos atrasos de recepção, nas reflexões, refrações e difrações sofridas pelo sinal durante a propagação, entre outros.

Dispositivos móveis, portanto, estarão susceptíveis a diversas falhas com respeito comunicação de dados;

Perdas de pacotes: podem ser causadas caso haja interferência ou movimentação que atenue o sinal a ponto do hardware não ser capaz de recuperar a mensagem. O estado de erro pode ser transitório e rápido, onde ocorrem perdas simples, ou podem ocorrer as chamadas rajadas ou “bursts” de perda de pacotes (uma sequência de pacotes é perdida).

Períodos de desconexão: o dispositivo pode perder a conexão com a rede, precisando fazer uma renegociação com o roteador ao qual está ligado (ou a outro roteador na área).

Trocas de ordem: as trocas de ordem podem ocorrer devido a atrasos diferentes em alguns pacotes.

Como a comunicação de dados está presente em grande parte de aplicações de dispositivos móveis, uma parte importante no teste destes sistemas é conseguir injetar esses falhas que simulem esses cenários.

5 MODELOS DE FALHAS DE COMUNICAÇÃO

O objetivo de utilizar métodos de injeção de falhas sobre um sistema é verificar se os mecanismos de proteção contra falhas serão suficientes quando esse sistema estiver em um cenário específico. Por essa razão, é fundamental caracterizar a ocorrência de falhas para esses cenários, e assim poder projetar um módulo injetor de falhas adequado.

Na literatura são apresentados diferentes modelos matemáticos que podem ser utilizados para estudar e simular falhas de comunicação em redes sem fio. A partir de medidas em cenários reais podem ser determinados os parâmetros que compõem o modelo.

A seguir serão apresentados alguns dos modelos mais utilizados para perda de pacotes, que serão simulados pelo injetor de falhas implementado neste trabalho.

5.1 Modelo de Perda Sem Memória de Bernoulli

É um dos modelos mais simples de perda. Determina que as perdas de pacotes ocorrem com uma taxa *r*. Analogamente, pode se pensar que cada pacote tem uma probabilidade fixa de ser perdido, independente de estados anteriores [VOIP-T].

Essa taxa ou probabilidade de perda pode ser estimada a partir de uma amostra, dividindo a quantidade de mensagens perdidas pelo total de mensagens enviadas na amostra.

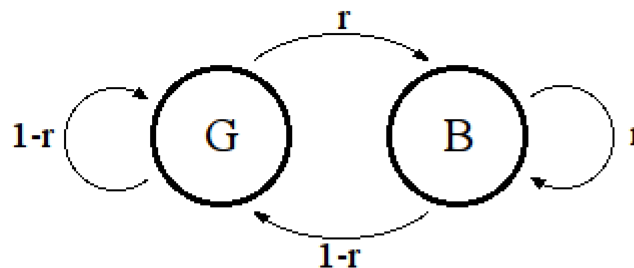

Figura 4.2: Cadeia de Markov representando o Modelo de Bernoulli.

A Figura 4.2 acima mostra como o modelo de Bernoulli pode ser representado com uma cadeia de Markov de dois estados. No estado *G* (“Good” ou “Gap”) as mensagens são entregues, no estado *B* (“Bad” ou “Burst”) as mensagens são perdidas. A probabilidade de ir para o estado de falha e de ficar no estado de falha é igual à taxa *r*.

5.2 Modelo de Perdas em Rajada de Gilbert Simplificado

O modelo de Gilbert simplificado utiliza uma cadeia de Markov semelhante a apresentada acima; possui os mesmos estados de entrega e perda de mensagens, mas com probabilidades diferentes de transições entre estados. É um modelo que permite

simular perdas de pacotes em rajada de forma satisfatória, desde que corretamente configurados seus parâmetros.

À diferença do modelo de Bernoulli, as probabilidades de transição entre os estados *G* e *B* não possuem uma relação matemática.

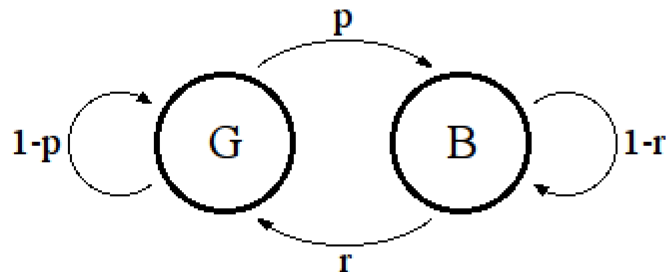

Figura 4.3: Cadeia de Markov representando o Modelo de Gilbert simplificado.

Na Figura 4.3 são apresentadas as probabilidades *p* e *r* propostas por Gilbert. A probabilidade *p* define a frequência de ocorrer uma rajada de perdas, e a probabilidade *r* influencia no tamanho da rajada. Os valores de *p* e *r* devem ser suficientemente pequenos para simular perdas corretamente [GILBERT 1960].

5.3 Modelo de Perdas em Rajada de Gilbert

No modelo de perdas não simplificado de Gilbert [GILBERT 1960], além dos parâmetros *p* e *r* é considerado também o parâmetro *h*, que é a probabilidade de não ocorrência de perdas no estado B.

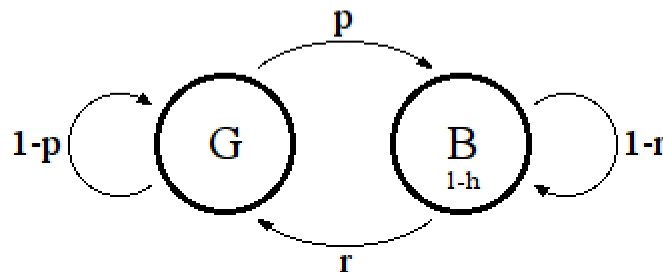

Figura 4.4: Cadeia de Markov que representa o Modelo de Gilbert.

No estado *B*, como observado na Figura 4.4, a probabilidade de perda de mensagens é de 1 - *h*. O estado *B* é analogo ao Modelo de Bernoulli, com a taxa de perda *r* sendo igual a 1 - *h*, podendo este estado ser também modelado com uma cadeia de Markov de dois estados.

5.4 Modelo de Perdas em Rajada de Gilbert-Elliott

No Modelo de Gilbert-Elliott [ELLIOTT 1963] é levada em consideração uma taxa de erro também no estado G, porém menor que no estado B. O valor *k* na Figura 4.5 representa a probabilidade de não ocorrência de perdas.

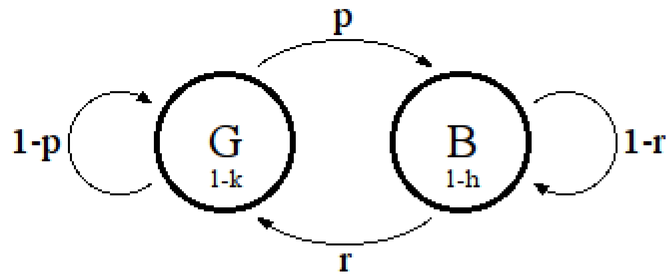

Figura 4.5: Modelo de Gilbert-Elliott.

O modelo permite representar de forma satisfatória dois estados que são característicos em redes sem fio; um estado bom (*G*) onde se tem perdas esporádicas de pacotes, e um estado ruim (*B*) onde ocorrem perdas em rajadas com poucos pacotes que podem transmitidos corretamente.

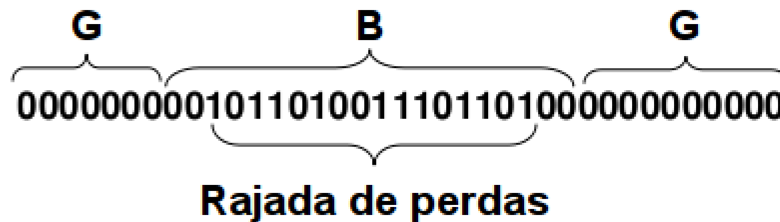

Figura 4.6: Fluxo de transmissão para h igual à 0,5.

A Figura 4.6 [SALSANO 2009], mostra um fluxo de transmissão, onde mensagens transmitidas são simbolizadas com 0 e mensagens perdidas com 1. Para esse fluxo foi utilizado o valor da probabilidade *h* igual à 0,5, como proposta originalmente por Gilbert. O objetivo é mostrar um ponto fraco do modelo; para *h* maior que 0, a duração do estado *B* não corresponde a largura da rajada desde a primeira até a última perda de mensagens.

5.5 Cadeia de Markov com Quatro Estados

O site VoIP Troubleshooter [VOIP-T] e posteriormente [SALSANO 2009] apresentam um modelo que consiste em uma cadeia de Markov de quatro estados, definida por cinco parâmetros não redundantes.

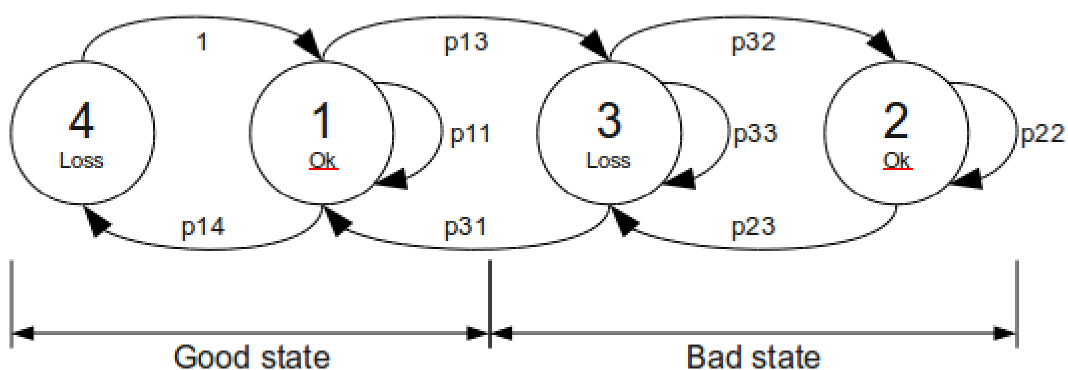

Figura 4.7: Cadeia de Markov de quatro estados [VOIP-T] [SALSANO 2009].

O modelo é semelhante ao de Gilbert-Elliott, tendo também um estado bom (na Figura 4.7 marcado como “Good state”) com poucas perdas e um estado ruim (marcado como “Bad state”) de perdas com algumas transmissões. Cada um desses estados é dividido em outros dois. Os estados 1 e 2 transmitem mensagens e nos estados 3 e 4 as mensagens são perdidas.

Os parâmetros $p13$, $p14$, $p23$, $p31$ e $p32$ são suficientes para caracterizar o modelo, sendo os parâmetros $p11$, $p22$ e $p33$ são calculados como:

$p11 = 1 - p13 - p14$

$p22 = 1 - p23$

$p33 = 1 - p31 - p32$

No estado bom, existe uma pequena probabilidade $p14$ de se passar do estado de transmissão 1 para o estado de perda 4. Para diminuir o número de parâmetros que caracterizam o modelo, a probabilidade de volta do estado 4 para o 1 é igual à 1, que se reflete em ter apenas uma perda por vez. Existe uma probabilidade $p13$ de se passar para o estado ruim de rajada de perdas.

Como não há transições entre os estados 1 e 2, o estado ruim inicia e finaliza no estado 3, onde efetivamente são perdidas mensagens. Assim, o ponto fraco apontado para o modelo de Gilbert-Elliott não ocorre, e o tamanho da rajada é igual a duração do estado ruim. Esta duração do estado ruim é relacionada a probabilidade $p31$ de transição.

A probabilidade $p32$ indica a frequência com a qual ocorrem transmissões durante a rajada de perdas e a probabilidade $p23$ a quantidade de transmissões consecutivas dentro da rajada de perdas.

6 AMBIENTE DE DESENVOLVIMENTO DO ANDROID

Para auxiliar os projetistas no desenvolvimento de aplicações Android, a Google disponibiliza um kit de desenvolvimento de software (SDK) de forma gratuita, para os principais sistemas operacionais: Windows, Mac OS e Linux. Além das ferramentas de linha de comando, é disponibilizado um plugin para o Eclipse, uma ferramenta gratuita de desenvolvimento de software que suporta diversas linguagens de programação.

Integrado com o ambiente, um emulador de dispositivos permite carregar e executar as aplicações sobre o Android, simulando o dispositivo alvo.

Como mencionado anteriormente, o código fonte do Android está disponível para os desenvolvedores, e será explicado também como gerar um SDK próprio para ser utilizado no ambiente de desenvolvimento, permitindo testá-lo com o emulador.

Para explicar os passos realizados para este trabalho de graduação, o ambiente foi instalado em um computador com as seguintes características de hardware:

- Processador Intel i5 2500k, com quatro núcleos
- 8 gigabytes de RAM
- Placa de video offboard

Com respeito ao software, foi utilizado:

- Instalação “limpa” do sistema operacional Ubuntu 10.04
- Kernel do Linux versão 2.6.32-28 para arquitetura de 64 bits.

Foram seguidas as indicações no site do Android, na data 09/05/2011.

6.1 Instalação do Ambiente

No site do Android [ANDROID] há todos os requisitos para a instalação do ambiente de desenvolvimento, assim como os passos para instalar suas dependências.

6.1.1 Java SDK - JDK

Primeiro é necessário ter o sistema de desenvolvimento Java, o JDK. Caso o objetivo seja utilizar apenas o ambiente de desenvolvimento, pode ser utilizada tanto a o JDK 5 quanto o JDK 6. Para ter maior flexibilidade, foram instaladas as duas versões.

Em ambos os casos o procedimento de instalação consiste em adicionar os repositórios correspondentes e fazer a instalação com o gerenciador de pacotes aptitude.

Para instalar o JDK 5:

```
$  sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu \
   dapper main multiverse"
$  sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu \
   dapper-updates main multiverse"
$  sudo apt-get update
$  sudo apt-get install sun-java5-jdk
```

Para o JDK 6:

```
$  sudo add-apt-repository "deb http://archive.canonical.com/ lucid
   partner"
$  sudo add-apt-repository "deb-src
   http://archive.canonical.com/ubuntu lucid partner"
$  sudo apt-get update
$  sudo apt-get install sun-java6-jdk
```

O comando *update-alternatives* no terminal mostra a lista de alternativas para um programa, e o usuario seleciona qual versão deseja utilizar. Com isso pode se alternar entre as versões do JDK.

6.1.2 Eclipse IDE

O Eclipse IDE é um ambiente de desenvolvimento gratuito, com suporte a diversas linguagens de programação e extensível com plugins. Com isso, consegue integrar ferramentas externas, como CVS ou GIT para controle de versões, Bugzilla para controle de bugs ou, de principal importância para este trabalho, as ferramentas do Android.

No site do projeto Eclipse [ECLIPSE] está disponível para ser baixada a versão recomendada para o Android, o Eclipse Classic. O download consiste em um arquivo compactado, que deverá ser descompactado em um diretório definida pelo usuário. Para este trabalho, o diretório utilizado foi:

`/home/usuario/programas/eclipse/`

ou, utilizando o caractere especial "~" que é equivalente a `/home/usuario`:

`~/programas/eclipse/`

Dentro desse diretório se encontra o executável, com o nome de *eclipse*, que é uma aplicação Java e, portanto, roda sobre uma máquina virtual. Ainda no mesmo diretório raiz se encontra o arquivo *eclipse.ini*, que permite configurar opções de inicialização, como o tamanho da memória alocada para a VM.

6.1.3 Android Plugin - ADT

O próximo passo descrito no site do Android é instalar o plugin ADT. O plugin é baixado e instalado diretamente pelo Eclipse, no menu:

Help → Install New Software...

No menu "drop down" com o rótulo *Work with:*, deve ser adicionado o repositório:

https://dl-ssl.google.com/android/eclipse/

Com isso deve surgir a opção *Developer Tools* que deve ser selecionada e prosseguir com a instalação. Após aceitar os temos de compromisso, deve se finalizar a instalação e esperar o Eclipse reiniciar. Por causa de um bug, deve se abrir a janela do ADT:

Window → Android SDK and AVD Manager

Na primeira vez, uma janela é aberta, onde se pode permitir ou não a coleta pela Google de dados anônimos para fins de estatística.

6.1.4 Instalação do SDK do Android

O Kit de Desenvolvimento de Software do Android deve ser baixado diretamente do site do Android e descompactado. O diretório utilizado neste trabalho foi:

```
~/programas/android-sdk-linux_x86/
```

Em sequência, o caminho do SDK deve ser ser configurado no Eclipse, através do menu:

Window → Preferences

Na janela de preferências, selecionar o item *Android* e configurar o campo *SDK Location* com o caminho correspondente.

6.1.5 Instalação das Plataformas Android

A cada nova versão do sistema Android que pode ser implantada em um aparelho, é criada uma plataforma. Cada plataforma contém, além das bibliotecas e imagens do sistema, exemplos de código e “skins” para o emulador. O SDK baixado, que consiste em apenas o núcleo das ferramentas, e pode ser utilizado para baixar os componentes restantes, incluindo as plataformas.

Os componentes são divididos de forma modular, separando as diferentes plataformas e as ferramentas que as suportam. Deve ser baixado, via o ADT plugin, pelo menos uma plataforma, sobre a qual serão desenvolvidas as aplicações. No Eclipse, ir no menu:

Window → Android SDK and AVD Manager

Selecionar o item *Available Packages*, e dentre as opções, abrir a caixa Android Repositories, onde serão mostradas todas as plataformas disponíveis que, finalmente, devem ser instaladas.

6.2 Emulador

O emulador do Android simula o dispositivo real no qual será implantado o sistema Android. Para poder executar o emulador, deve ser criado pelo menos um dispositivo virtual com o plugin ADT, a partir das plataformas que tenham sido instaladas. Para isso, no Eclipse, ir no menu:

Window → Android SDK and AVD Manager

Selecionar o item *Virtual Devices* e pressionar o botão com rótulo *New...*, onde será atribuido o nome do dispositivo, selecionada a versão da plataforma alvo e outras opções.

Com o emulador é possível tanto observar a aplicação resultante como depurá-la, bastando ir no menu do Eclipse para rodar a aplicação:

Run → Run

Ou para depurar:

Run → Debug

O emulador é iniciado e, em sequência, a aplicação é enviada, instalada e executada.

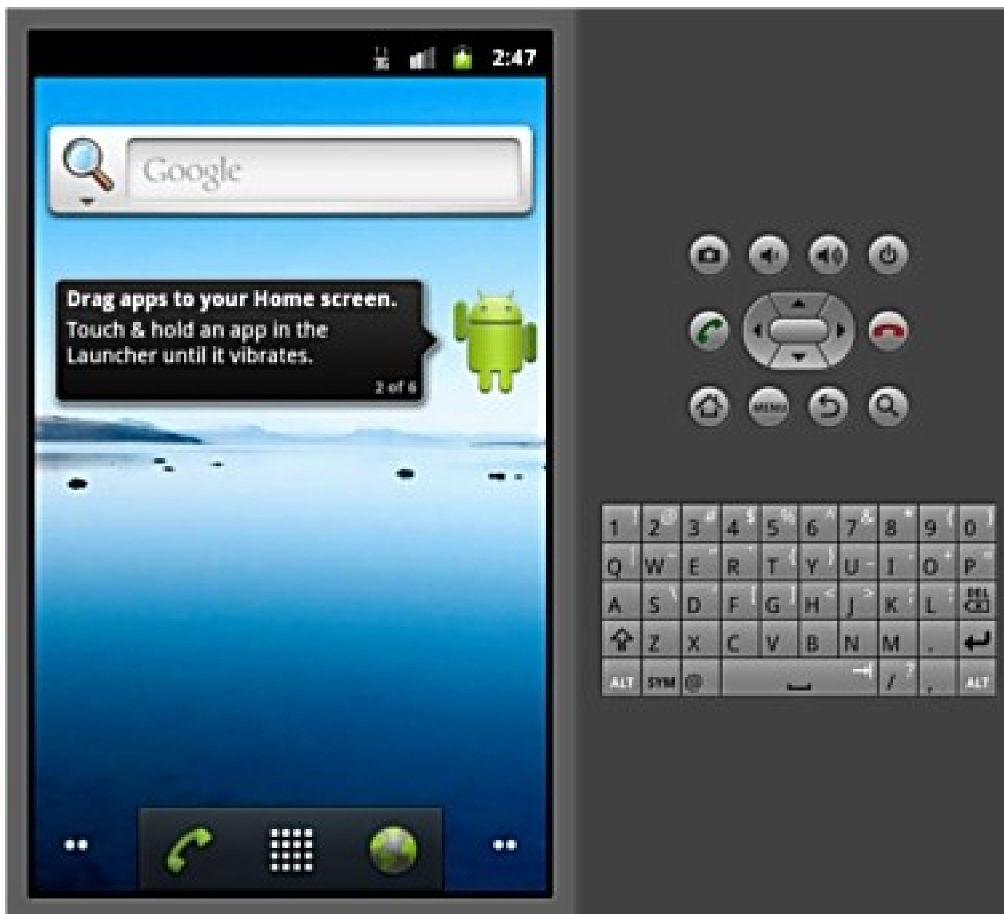

Figura 8: Interface gráfica do emulador do Android.

A interface gráfica do emulador é semelhante a um aparelho real, como pode ser observado na Figura 8. A janela é dividida em duas partes: à esquerda se encontra a tela do dispositivo, que funciona como uma tela sensível ao toque ("touchscreen"); no lado direito estão os botões e o teclado. A tela sensível ao toque é emulada com ajuda do mouse, sendo o clique com o botão esquerdo equivalente a encostar na tela. Além do teclado da interface gráfica, o usuário pode digitar no teclado do computador, ou utilizar o teclado virtual que pode ser invocado pelas aplicações.

As ferramentas do SDK, junto com o plugin para o Eclipse, facilitam a função de depuração das aplicações. No Eclipse, ao iniciar a execução de uma aplicação em modo "Debug", é selecionada uma perspectiva de depuração, permitindo o usuário adicionar "break points", executar passo a passo e monitorar variáveis.

Outra perspectiva de grande utilidade é a "DDMS" (Dalvik Debug Monitor Server), que permite controlar o emulador. Nessa perspectiva são exibidas diversas informações sobre o estado do emulador, as threads que estão sendo executadas, as mensagens do console e logs, entre outros. Além dessas informações, é possível iniciar chamadas, alterar os estados da conexão de voz, dados, taxas e latência, transferir arquivos de e para o emulador.

6.3 Construção e Utilização de SDKs Próprios

Assim como é possível baixar as plataformas do Android que foram lançadas, também existe a possibilidade de criar o próprio SDK, a partir do código fonte. Desta forma, o desenvolvedor consegue testar tanto as suas aplicações quanto alterações ou criação de novas funcionalidades implementadas no sistema Android.

Na continuação serão mostrados os passos realizados neste trabalho para preparar o mesmo computador descrito na instalação do ambiente de desenvolvimento, para logo baixar o código fonte do Android, compilar um novo SDK e utilizá-lo com o emulador no Eclipse. Além das instruções presentes no site do Android [ANDROID], serão mostrados alguns problemas e as soluções encontradas.

6.3.1 Preparação do Sistema

O primeiro passo para preparar o sistema é instalar o JDK correto. Dependendo da versão do Android que se quer compilar, deve se escolher uma das versões do JDK que foi instalada. Para as versões mais recentes, a partir da Gingerbread (nome dado a versão 2.3), deve ser escolhido o JDK 6; para versões mais antigas deve ser utilizado o JDK 5. No caso deste trabalho, foi escolhida a versão 2.3, por isso JDK 6.

Com o comando *update-alternatives* é feita a escolha entre as versões dos componentes do JDK. Em um terminal, executar o comando:

```
$  sudo update-alternatives --config java
```

Dentre as opções fornecidas, selecionar a correspondente ao JDK 6. Repetir o procedimento para os componentes em comum entre as duas versões do JDK, além do *java*: *javac*, *javadoc*, *javah* e *javap*.

Também são necessários o compilador/interpretador *Python* versão 2.4 até 2.7, e o sistema de controle de revisões distribuído *Git* versão 1.5.4 ou mais recente. No caso do compilador/interpretador *Python*, este já estava presente no sistema, verificado no terminal com o comando:

```
$  python --version
Python 2.6.5
```

Da mesma forma, a versão do Git instalado com o sistema é verificado com o comando:

```
$  git --version
git version 1.7.0.4
```

Dependendo da sistema ser de 32 e 64 bits, uma série de pacotes deve ser instalada. Para este trabalho, foi instalada a lista de pacotes para 64 bits, via comando *apt-get install* no terminal:

```
$  sudo apt-get install git-core gnupg flex bison gperf \
   build-essential zip curl zlib1g-dev libc6-dev \
   lib32ncurses5-dev ia32-libs x11proto-core-dev libx11-dev \
   lib32readline5-dev lib32z-dev
```

Apenas as bibliotecas que não estejam presentes no sistema são instaladas. Embora no site do Android sejam indicadas para montar versões mais antigas em sistemas de 64 bits, podem ser necessárias mais pacotes para estabelecer um ambiente de compilação correto, instalados com o seguinte comando no terminal:

```
$  sudo apt-get install gcc-multilib g++-multilib libc6-i386 \
   libc6-dev-i386
```

Seguidos os passos acima, o sistema está pronto para obter o código fonte.

6.3.2 Obtenção do Código Fonte

O Android utiliza a ferramenta *Repo* para facilitar o uso do *Git*. Para facilitar o procedimento, deve ser criado um diretório para executáveis onde será colocado o *Repo*, e o diretório deve ser adicionado na variável de ambiente *PATH*, para poder executar a ferramenta sem precisar fornecer o caminho completo. O diretório é criado com o comando no terminal:

```
$  mkdir ~/bin
```

No arquivo `~/.bashrc`, adicionar a linha a seguir para incluir o diretório no *PATH* cada vez que o terminal for executado:

```
export PATH=~/bin:$PATH
```

Deve ser baixado e dada a permissão ao *Repo* para ser executável, com os comandos:

```
$  curl http://android.git.kernel.org/repo > ~/repo
$  sudo chmod 777 ~/bin/repo
```

Em uma instalação anterior, em um computador com *Python* de versão diferente, ocorreu uma incompatibilidade ao tentar importar a biblioteca *readline*, e o *Repo* foi modificado para incluir a biblioteca *rlcompleter*.

A seguir deve ser criado o diretório para os fontes do Android e ir para o diretório:

```
$  mkdir ~/android
$  cd ~/android
```

Finalmente, o *Repo* deve ser inicializado com o ramo de trabalho, neste caso o *gingerbread*, e o código fonte desse ramo é baixado:

```
$  repo init \
   -u git://android.git.kernel.org/platform/manifest.git \
   -b gingerbread
$  repo sync
```

O ramo da versão *gingerbread* tem aproximadamente 3,3 gigabytes.

6.3.3 Compilação e Utilização do SDK

Após baixar o código, ainda no diretório criado para os fontes do Android, deve ser executado um script de inicialização para preparar o ambiente do terminal, escolher como alvo da compilação a primeira opção *generic-eng* e compilar o Android para gerar o SDK:

```
$  source build/envsetup.sh
$  lunch 1
$  make sdk
```

A compilação pode ser acelerada alterando o número de “jobs” para utilizar mais de um núcleo do processador. O projeto foi preparado para permitir o paralelismo das

tarefas, e o uso desse paralelismo é aconselhado no site do Android. No computador utilizado neste trabalho, foram utilizados quatro “jobs” e o resultado foi satisfatório:

```
$  make sdk -j4
```

Como resultado da compilação é gerado um arquivo comprimido como .zip no diretório:

`~/android/out/host/linux-x86/sdk/`

Esse arquivo .zip contem um SDK como o baixado do site do Android, com a plataforma compilada como se tivesse sido baixada através do plugin. Para ser utilizado no Eclipse e com o emulador, o procedimento é o mesmo apresentado para instalar o SDK, as plataformas do Android e a criação do dispositivo a ser emulado.

7 IMPLEMENTAÇÃO DO INJETOR DE FALHAS UDP

A proposta deste trabalho de graduação é injetar falhas de comunicação UDP em aplicações sob teste, com o objetivo de explorar a injeção de falhas no nível da máquina virtual do Android.

Como não é possível utilizar ferramentas já disponíveis para outras plataformas, como as apresentadas Javassist ou JVMTI, foi decidido explorar a solução de modificação do código da máquina virtual.

Primeiro será apresentada a classe que implementa os *sockets* UDP: a localização do arquivo da classe na estrutura de diretórios dos fontes do Android, o funcionamento dos métodos de interesse da classe e uma visão global de como foi feita a injeção das falhas na classe.

Após, é descrita a implementação do injetor de falhas com maior detalhamento, explicando os motivos para se tomar as decisões dos pontos mais importantes, e é demonstrado como o injetor consegue implementar diferentes modelos de perdas de mensagens.

7.1 Classe DatagramSocket no Android e Principio do Funcionamento do Injetor

A comunicação UDP em sistemas baseados em Java é suportada, em mais alto nível, pela classe java.net.DatagramSocket. Para encontrar o arquivo da classe dentro da estrutura de diretórios do código fonte do Android, foi procurado pelo nome com o comando no terminal:

```
$  find -name DatagramSocket.java
./libcore/luni/src/main/java/java/net/DatagramSocket.java
```

O arquivo DatagramSocket.java contém todos os atributos e métodos da classe. Na implementação do injetor de falhas UDP, foram adicionados atributos e modificados os construtores e os métodos *receive()* e *send()*, que são os métodos para receber e enviar pacotes. O injetor funciona com uma cadeia de Markov de quatro estados, onde dois são de transmissão normal e dois são de perda.

7.1.1 Métodos Construtores

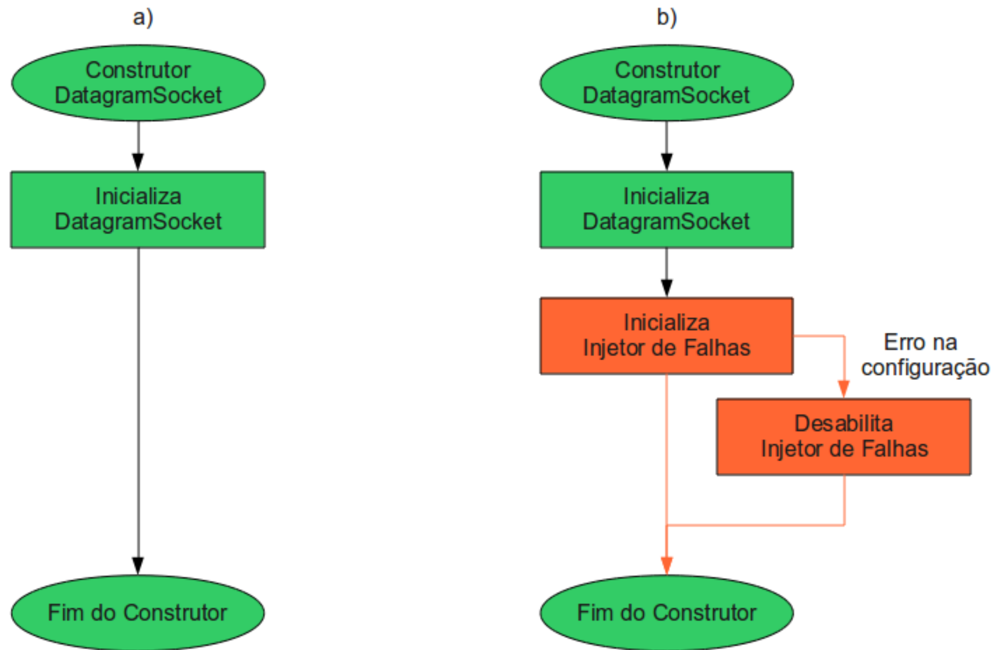

Figura 9: O construtor da classe DatagramSocket a) original e b) após a implementação do injetor

A classe *DatagramSocket* possui cinco construtores, nos quais foi adicionada ao final uma chamada à função de inicialização do injetor, como pode ser visto na Figura 9. Nessa função é feita a leitura de um arquivo de configuração e são aplicadas as configurações para o *DatagramSocket* recém-criado. Caso o arquivo de configuração não esteja completo ou tenha alguma incoerência, o injetor não é habilitado.

7.1.2 Método *receive()*

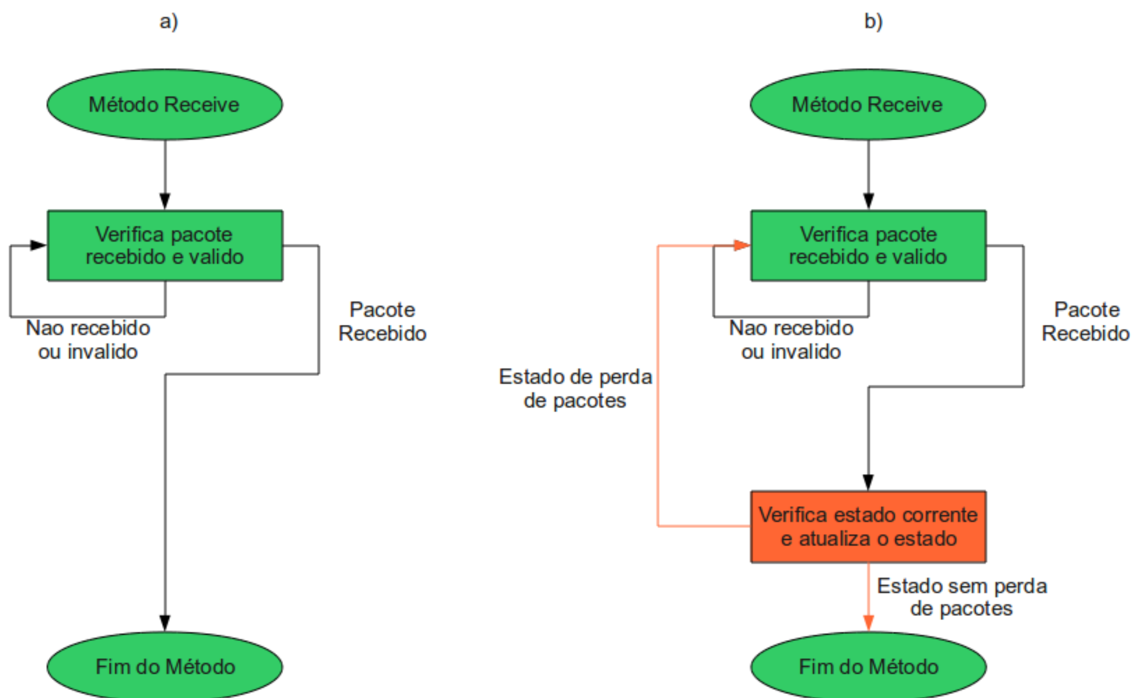

Figura 10: O método *receive()* a) original e b) após a implementação do injetor.

O *receive()* consiste, basicamente, em um laço, onde o método fica preso até verificar a chegada de um pacote válido para o *socket*. Quando o pacote for recebido é

preenchida com as informações do pacote uma estrutura do tipo DatagramPacket, que é passada como parâmetro ao método. Para perder pacotes na recepção, foi adicionado um laço que engloba o laço original da função, onde o estado corrente é verificado e atualizado para o próximo estado. Caso se esteja em um estado de perda, ocorre uma nova iteração, esperando o próximo pacote; caso contrário, a condição de saída do laço do injetor vai ser alcançado e a chamada do método *receive()* é finalizada. Este comportamento é ilustrado na Figura 10.

Para tentar diminuir o tempo adicional utilizado para manutenção do injetor, se optou por minimizar a quantidade de testes (e por consequência a quantidade de desvios), com o ônus de duplicar trechos de código. O maior exemplo disso é que, logo após a declaração e inicialização das variáveis, é feito o teste se o injetor está habilitado para recepção; caso este esteja desabilitado, o restante do código será idêntico ao original do método.

7.1.3 Método *send()*

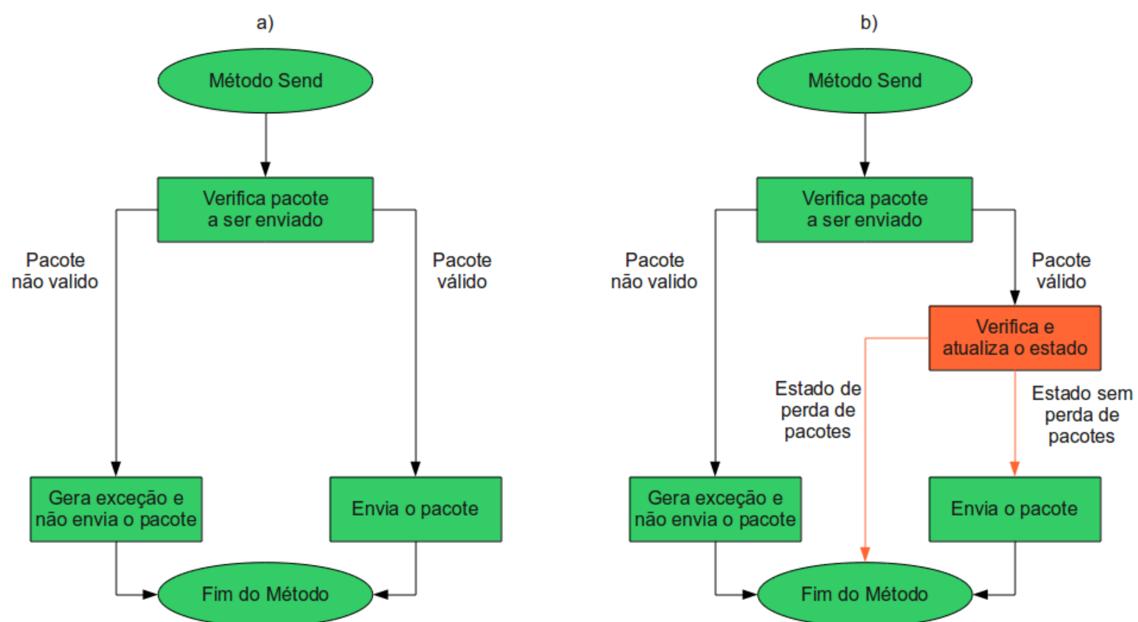

Figura 11: O método *send()* a) original e b) após a implementação do injetor

No método *send()*, o pacote passado como argumento é verificado e, caso o pacote seja não válido, e gerada uma exceção correspondente e o pacote não é enviado. Se nenhuma exceção for gerada, o pacote valido é enviado. Como é possível observar na Figura 11, no método *send()* modificado, os pacotes válidos são enviados apenas se o injetor estiver no estado sem perdas de pacotes.

7.2 Detalhamento da Implementação do Injetor de Falhas UDP

O injetor foi implementado com uma cadeia de Markov de quatro estados, de forma a simular os cinco modelos apresentados anteriormente neste trabalho. Os estados são numerados de zero à três, sendo os estados “0” e “1” sem perda de pacotes, e os estados “2” e “3” de perda de pacotes.

7.2.1 Implementação da Cadeia de Markov

Dentre os atributos da classe *DatagramSocket*, foi adicionada uma matriz de valores em ponto flutuante de dimensão quatro por quatro, que contém as probabilidades de transição acumuladas, sendo o primeiro índice o estado atual e o segundo índice o próximo estado. A Figura 12 a seguir ilustra a cadeia de Markov com todas as transições

entre estados e a matriz de probabilidades acumuladas, onde cada linha representa as transições com o mesmo estado de origem e a coluna o estado destino.

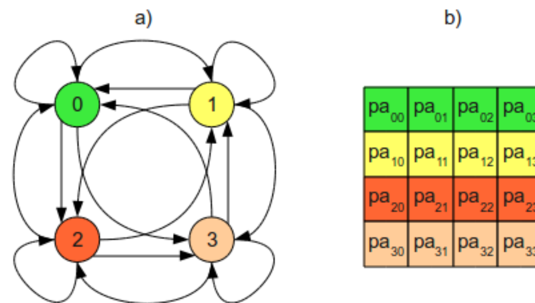

Figura 12: a) representação gráfica. b) matriz de transições.

As probabilidades de transição devem estar contidas no intervalo [0,1], sendo que a soma de todas as probabilidades deve resultar em 1. As probabilidades de transição de cada estado são acumuladas para que um valor escolhido dentro do intervalo [0,1] represente somente uma das transições. Exemplo:

p_{00} = 0,05 ; p_{01} = 0,20 ; p_{02} = 0,35 ; p_{03} = 0,40

As probabilidades acumuladas serão:

pa_{00} = 0,5 ; pa_{01} = 0,25 ; pa_{02} = 0,60 ; pa_{03} = 1,00

Isto é, caso o estado corrente for “0”, o próximo estado será:

“0”, se o número escolhido for menor que 0,5.

“1”, se o número escolhido for maior ou igual a 0,5 e menor que 0,25

“2”, se o número escolhido for maior ou igual a 0,25 e menor que 0,60

“3”, se o número escolhido for maior ou igual a 0,60 e menor que 1,00

Além da matriz, foi adicionado um atributo inteiro como identificador do estado atual. Também foram adicionados mais dois atributos, do tipo booleano, para indicar se a injeção de falhas será habilitada para o método *send()* e para o *receive()*. Apesar do injetor poder ser habilitado simultaneamente para os dois métodos, é utilizada a mesma cadeia de Markov para ambos.

No construtor do *DatagramSocket*, os atributos que foram adicionados são inicializados a partir de arquivos de configuração. Dentro dos métodos *send()* e *receive()*, se o estado corrente for menor que “2”, o pacote é enviado / recebido, caso contrário será perdido. Para calcular o próximo estado, é sorteado um valor randômico no intervalo [0,1), e este valor é comparado com as probabilidades de transição acumuladas na matriz, como mostrado no exemplo. O código modificado da classe *DatagramSocket* está disponível no apêndice.

7.2.2 Ajustes do *timeout* para o Método *receive*

O *receive()* é, por padrão, um método bloqueante, que espera até ser recebido um novo pacote. Para modificar esse comportamento é possível atribuir um valor de *timeout*, que consiste em um tempo máximo de espera pela recepção de um pacote antes de ser gerada uma exceção.

Originalmente, o controle desse *timeout* é realizado em nível mais baixo que a implementação do *DatagramSocket* da máquina virtual, causando que a cada novo pacote recebido reinicie a contagem de tempo, mesmo que o injetor tenha simulado a perda desse pacote. Essa diferença de comportamento pode inserir erros que não

ocorreriam mesmo na perda real de pacotes; a exceção de *timeout* não será gerada mesmo que o injetor simule uma perda total de conexão, onde todos os pacotes são perdidos, enquanto se esteja recebendo pacotes *socket*.

Para solucionar esse problema foi feito um ajuste do *timeout*. A classe DatagramSocket modificada mantém o valor de tempo original do *timeout* que foi configurado pela aplicação, e este é configurado em mais baixo nível no início de cada chamada do método *receive()*. Também no início da chamada do *receive()*, é pego um valor de referência de tempo: cada vez que um pacote é descartado, o tempo transcorrido desde a referência é decrementado do *timeout* original. Caso esse decremento for maior ou igual ao *timeout* original, é gerada a exceção de *timeout* para a aplicação.

7.2.3 Configuração Inicial do Injetor

Na inicialização do *DatagramSocket*, o primeiro arquivo a ser lido é o de configuração inicial, comum a todos os modelos, que está localizado dentro do sistema de arquivos do emulador em:

```
/data/local/tmp/UDP_loss_injector.cfg
```

O arquivo é composto por quatro números inteiros. Os dois primeiros valores habilitam, caso diferentes de "0", a injeção de falha no envio e recepção, respectivamente. O terceiro valor indica qual modelo será configurado, com valores válidos de "0" até "5" representando, respectivamente, os modelos de Bernoulli, Gilbert Simplificado, Gilbert, Gilbert-Elliott, cadeia de Markov de quatro estados [VOIP-T] [SALSANO 2009], e uma cadeia de Markov de quatro estados genérica onde todas as probabilidades de transição podem ser configuradas. O quarto valor indica o estado inicial, cujo significado depende do modelo.

A transformação dos modelos para a cadeia de Markov do injetor serão mostrados a seguir, e o código de inicialização se encontra no apêndice deste trabalho, para maior detalhamento.

7.2.4 Configuração para o Modelo de Bernoulli

Para configurar o modelo de Bernoulli é lido o arquivo:

```
/data/local/tmp/Bernoulli.cfg
```

O único parâmetro que precisa ser configurado é a taxa de perda r. O injetor configura o estado "0" para transmissão e o estado "2" para perda.

Tabela 1: Tabela de probabilidades para o modelo de Bernoulli.

		Próximo Estado			
		0	1	2	3
Estado de origem	0	$1-r$	0	r	0
	1	0	0	0	0
	2	$1-r$	0	r	0
	3	0	0	0	0

A Tabela 1 acima mostra as probabilidades não acumuladas de transição configuradas no injetor. O estado inicial é definido na configuração comum como zero para sem perdas e diferente para com perdas.

7.2.5 Configuração para o Modelo de Gilbert Simplificado

Na configuração do modelo de perdas de Gilbert Simplificado, são lidos as probabilidades *p* e *r*, contidas no arquivo:

```
/data/local/tmp/SimpleGilbert.cfg
```

Da mesma forma que no modelo de Bernoulli, são utilizados os estados “0” e “2” para comunicação sem e com perdas, respectivamente.

Tabela 2: Tabela de probabilidades para o modelo de Gilbert Simplificado.

		Próximo Estado			
		0	1	2	3
Estado de origem	0	1-*p*	0	*p*	0
	1	0	0	0	0
	2	*r*	0	1-*r*	0
	3	0	0	0	0

As probabilidades de transição para configuração do modelo são apresentadas na Tabela 2. O estado inicial é configurado da mesma forma que o modelo anterior.

7.2.6 Configuração para o Modelo de Gilbert

No modelo de perdas de Gilbert são definidos os parâmetros *p*, *r* e *h*, configurados no arquivo:

```
/data/local/tmp/Gilbert.cfg
```

Neste caso, para o estado bom (sem perdas) ainda é utilizado o estado “0”, e para o estado ruim são utilizados os estados “1” e “2”, por haver a probabilidade *h* de se transmitir pacotes nesse estado.

Tabela 3: Tabela de probabilidades para o modelo de Gilbert.

		Próximo Estado			
		0	1	2	3
Estado de origem	0	1-*p*	(*p*) * (*h*)	(*p*) * (1-*h*)	0
	1	*r*	(1-*r*) * (*h*)	(1-*r*)*(1-*h*)	0
	2	*r*	(1-*r*) * (*h*)	(1-*r*)*(1-*h*)	0
	3	0	0	0	0

Como além da probabilidade de se ir para o estado ruim, existe também a probabilidade de se perder ou não os pacotes, as probabilidades finais de transição de estados são compostas, e estão definidas na Tabela 3. O estado inicial pode ser configurado como zero para o estado bom, e para o estado ruim, caso diferente de zero, será utilizada a probabilidade h para definir o estado inicial “1” ou “2”.

7.2.7 Configuração para o Modelo de Gilbert-Elliott

O modelo de Gilbert-Elliott inclui a probabilidade *k* de transmissão no seu estado bom. Os parâmetros *p*, *r*, *h* e *k* devem ser fornecidos, nessa ordem, no arquivo:

```
/data/local/tmp/GilbertElliott.cfg
```

De forma semelhante ao realizado na configuração do modelo anterior, o estado bom é também dividido nos dois estados "0" e "3". As probabilidades resultantes das transformações são apresentadas na Tabela 4.

Tabela 4: Tabela de probabilidades para o modelo de Gilbert-Elliot.

		Próximo Estado			
		0	1	2	3
Estado de origem	0	(1-*p*) * (*k*)	(*p*) * (*h*)	(*p*) * (1-*h*)	(1-*p*)*(1-*k*)
	1	(*r*) * (*k*)	(1-*r*) * (*h*)	(1-*r*)*(1-*h*)	(*r*) * (1-*k*)
	2	(*r*) * (*k*)	(1-*r*) * (*h*)	(1-*r*)*(1-*h*)	(*r*) * (1-*k*)
	3	(1-*p*) * (*k*)	(*p*) * (*h*)	(*p*) * (1-*h*)	(1-*p*)*(1-*k*)

O estado inicial bom é configurado pelo usuário com o valor zero, e o estado ruim com um valor diferente, e o estado na cadeia de Markov é escolhido de acordo com as probabilidades *k* para o estado bom, e *h* para o estado ruim.

7.2.8 Configuração para o Modelo de Quatro Estados

A cadeia de Markov [VOIP-T] [SALSANO 2009] tem os seus parâmetros *p13*, *p14*, *p23*, *p31* e *p32* configurados nessa ordem no arquivo:

```
/data/local/tmp/FourStateModel.cfg
```

O modelo também é representado com uma cadeia de Markov de quatro estados, da mesma forma que o injetor foi implementado, resultando na configuração apresentada na Tabela 5.

Tabela 5: Tabela de probabilidades para o modelo de quatro estados [VOIP-T] [SALSANO 2009].

		Próximo Estado			
		0	1	2	3
Estado de origem	0	1-*p11*-*p13*	0	*p13*	*p14*
	1	0	*1-p23*	*p23*	0
	2	*p31*	*p32*	1-*p31*-*p32*	0
	3	1	0	0	0

Os estados do modelo de 1 a 4 são mapeados para os estados de "0" a "3" do injetor, e o estado inicial é configurado diretamente com o índice, iniciando em "0".

7.2.9 Configuração para a Cadeia de Markov Genérica de Quatro Estados

Com esta configuração do injetor de falhas, o usuário pode trabalhar com outros modelos, ajustando as probabilidades das transições e o estado inicial. O arquivo de configuração neste caso é:

`/data/local/tmp/FourStateMarkovChain.cfg`

São configurados, em sequência, as três primeiras probabilidades de transição de cada estado de origem. A quarta quarta probabilidade é omitida, pois, ao acumular as probabilidades, os valores acumulados da última coluna devem ser iguais à 1.

Tabela 6: Tabela de probabilidades para a cadeia de Markov de quatro estados.

		Próximo Estado			
		0	1	2	3
Estado de origem	0	1-*p11*-*p13*	0	*p13*	*p14*
	1	0	*1-p23*	*p23*	0
	2	*p31*	*p32*	1-*p31*-*p32*	0
	3	1	0	0	0

A Tabela 6 mostra como a sequência de doze probabilidades definida no arquivo de configuração é preenchida na tabela do injetor de falhas UDP. O estado inicial é configurado de forma direta a partir do índice.

8 TESTES DO INJETOR DE FALHAS UDP

No capítulo anterior foi apresentado um injetor de falhas para *sockets* UDP que simula a perda de pacotes utilizando diferentes modelos de perda. Neste capítulo será explicado como o injetor foi testado, para avaliar o seu desempenho de forma qualitativa e quantitativa.

8.1 Aplicação Cliente – Servidor

Dentre os testes realizados para validação do porte do injetor de falhas de comunicação Firmament para o Android [DOBLER 2010], foi utilizada uma aplicação simples do tipo Cliente-Servidor. A aplicação Servidor é executada no emulador do Android, e o Cliente roda na máquina virtual no mesmo computador.

Para que as aplicações Cliente e Servidor possam se comunicar, é necessário escolher a porta que será utilizada pelo *socket* UDP e habilitá-la no emulador. Após o emulador iniciar, ele deve ser acessado por *telnet* no endereço local, na porta do emulador (a porta padrão do emulador é a 5554) e habilitar a porta com o comando de redirecionamento. Neste caso foi utilizada a 3333 para o *socket* do Servidor. No terminal, executar:

```
$  telnet localhost 5554
redir add udp:3333:3333
exit
```

Além de habilitar a porta no emulador, também deve se habilitar a permissão de acesso a internet no arquivo AndroidManifest.xml do projeto do Servidor. O arquivo AndroidManifest.xml se encontra no apêndice deste trabalho.

Ambas aplicações serão descritas a seguir, junto com as modificações necessárias para realizar os testes do injetor.

8.1.1 Cliente

Originalmente, a aplicação Cliente gera pacotes com um número de sequência e envia para o Servidor. Isso é feito com um laço do tipo *for*, sendo a variável de controle um inteiro, que é utilizado como número de sequência. Dentro do laço são criados o *buffer* para os dados do pacote, o pacote, e alguns *streams* para manipular o número de sequência e copiá-lo para dentro dos primeiros quatro bytes de dados pacote. Após o pacote estar pronto, este é enviado e a aplicação espera cinco milissegundos antes da próxima iteração do laço.

Esta aplicação Cliente foi rearranjada, reduzindo o número de instruções dentro do laço ao utilizar sempre o mesmo pacote. No laço de repetição são feitas as operações de atualização do índice no mesmo pacote, envio de pacote e espera. Além dessa alteração,

nos testes do injetor implementado neste trabalho, foi necessário diferenciar dois tipos de Cliente.

O primeiro, para o teste qualitativo, envia os pacotes com um tempo de espera suficiente para serem processados pelo Servidor. O tempo utilizado foi de 10 milissegundos, e foram enviados 10.000 pacotes.

No segundo, o tempo de espera foi deixado no valor mínimo, para provocar o envio dos pacotes em rajada. A espera funciona como uma interrupção na execução do Cliente, diminuindo a influência do escalonamento de processos do sistema operacional do computador. Neste caso, o Cliente envia 100.000 pacotes.

8.1.2 Servidor

O Servidor é dividido em duas classes: a aplicação Android principal e uma *thread* de recepção de pacotes. A aplicação principal apenas inicializa a *thread* de recepção e passa o controle. A *thread* de recepção inicia criando o *socket* UDP e a variável para receber os pacotes. Após, entra em um laço infinito de execução, interrompido apenas quando a aplicação for encerrada pelo usuário.

Dentro desse laço, são inicializadas as estruturas para guardar os números de sequência dos pacotes recebidos, duplicados e fora de ordem, e também é inicializado o *timeout* do *socket* como zero, para desabilitar a exceção por *timeout*. A partir desse ponto, o código que recebe os pacotes está dentro de um bloco *try-catch*, que captura a interrupção por *timeout* do *socket*.

Com a exceção desabilitada, o Servidor permanece preso, esperando a chegada do primeiro pacote. Logo que o primeiro pacote é recebido, o *timeout* é setado para 10 segundos, o primeiro pacote é processado e é iniciado um laço infinito de recepção e processamento dos pacotes. Quando o Cliente deixar de enviar pacotes, o laço de recepção é interrompido pela exceção de *timeout*, e no tratamento da exceção são exibidas no console as informações coletadas da sequência de pacotes recebida.

Como o injetor implementado efetua falhas apenas do tipo perda de pacotes, a aplicação Servidor foi modificada, removendo as checagens de pacotes duplicados e fora de ordem, e com isso, reduzindo o número de operações dentro do laço de recepção. Ainda para diminuir o número de instruções dentro do laço de recepção, a obtenção do número de sequência foi alterada para consumir menos instruções, fazendo a manipulação direta nos quatro primeiros bytes e formando o valor inteiro.

Para o teste qualitativo, além de mostrar os resultados no console, o Servidor gera um arquivo, onde cada linha contém o número de pacotes recebidos a cada intervalo de cinquenta pacotes. Isto é, a primeira linha contém quantos pacotes foram recebidos com número de sequência entre 0 e 49, a segunda linha os recebidos entre 50 e 99, e assim sucessivamente. O arquivo de saída é salvo no sistema de arquivos do emulador em:

```
/mnt/sdcard/server_out.txt
```

Para a aplicação conseguir gravar no cartão de memória, foi necessário, além da permissão para internet, dar permissão de escrita no seu arquivo AndroidManifest.xml, dentro do projeto da aplicação.

8.2 Testes da Simulação dos Modelos

O primeiro teste realizado é a simulação dos modelos. Consiste em configurar o injetor para os diferentes modelos, executar as aplicações Cliente e Servidor e analisar o seu desempenho através do arquivo de saída do Servidor. Como mencionado

anteriormente, para este teste foi utilizado o Cliente com o período de espera de 10 milissegundos entre envios de pacotes, com total de 10.000 pacotes.

Os valores para os parâmetros de configuração dos modelos não foram retirados de análise de redes reais, foram escolhidos de forma a realizar uma análise de forma qualitativa, de acordo com a sequência dos pacotes recebidos.

8.2.1 Simulação do Modelo de Bernoulli

Neste primeiro teste, o injetor foi configurado habilitando as falhas no método *receive()* e configurando o modelo como para o de Bernoulli, com a taxa de erro *r* igual a 0,30 (equivalente a 30% de erro). O conteúdo do arquivo de configuração comum foi:

```
0
1
0
0
```

No arquivo de configuração Bernoulli.cfg:

```
0.30
```

Foi gerado o gráfico com o arquivo de saída, que pode ser visto na Figura 13.

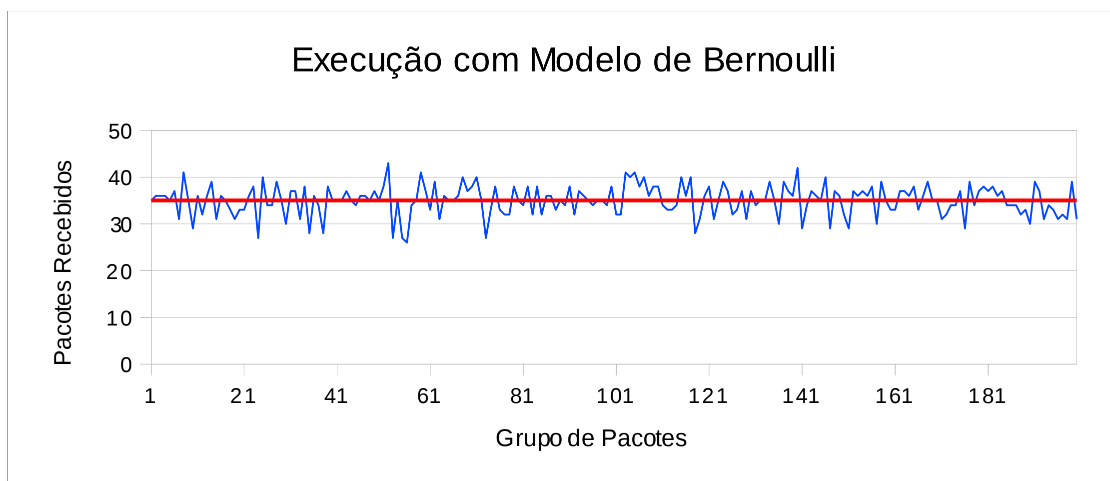

Figura 13: Gráfico para execução com Modelo de Bernoulli.

Na Figura 13, é possível observar, em azul, a quantidade de pacotes recebidos a cada 50 pacotes consecutivos e, em vermelho, a média de pacotes recebidos durante toda a execução. Embora a recepção tenha sido de forma aleatória, os valores estão próximos da média de 35 pacotes, equivalente aos 70%. Isso está de acordo com a taxa de erro de 30% configurados no modelo. O Servidor recebeu 69,99% dos pacotes enviadas pelo Cliente.

8.2.2 Simulação do Modelo de Gilbert Simplificado

Neste teste o injetor foi configurando para utilizar o modelo de Gilbert Simplificado, com o parâmetro *p* igual a 0,001 e o parâmetro *r* igual a 0,005. Essa configuração pode ser interpretada como uma probabilidade de 0,1% de ocorrer uma rajada de erros, que deve finalizar com probabilidade de 0,5%. O conteúdo do arquivo de configuração comum foi:

```
0
1
1
0
```

No arquivo de configuração SimpleGilbert.cfg:

```
0.001
0.005
```

O gráfico dos pacotes recebidos é apresentado na figura Figura 14.

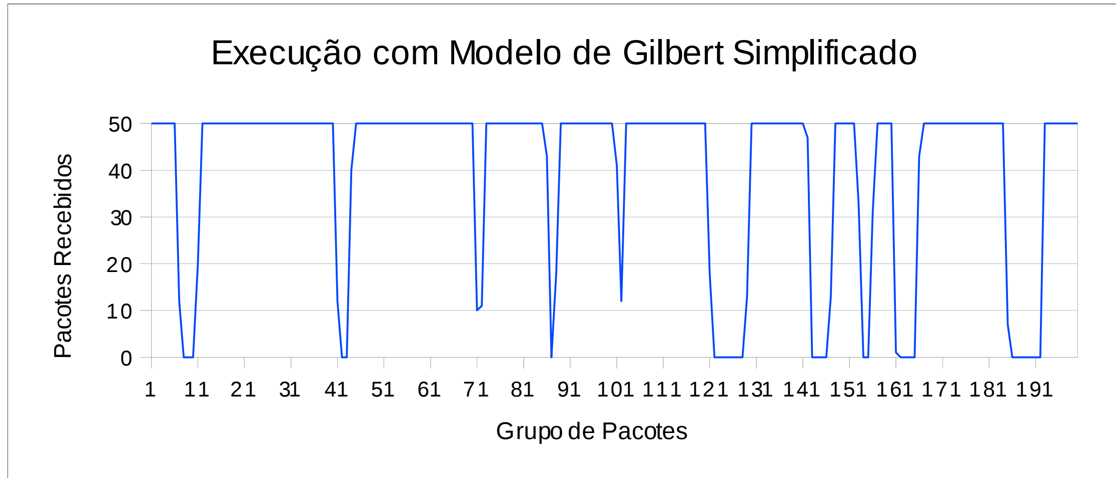

Figura 14: Gráfico para execução com Modelo de Gilbert Simplificado.

Como pode ser visto no gráfico, ocorrem rajadas de erro, onde nenhum pacote é recebido. No caso das rajadas de perda próximas aos grupos de pacotes 70 e 100, são rajadas curtas que estão divididas em dois grupos, e por isso não alcançam o valor 0.

Nessa execução o Servidor recebeu 79,75% dos pacotes enviadas pelo Cliente.

8.2.3 Simulação do Modelo de Gilbert

Para o teste do modelo de Gilbert, foi mantida a configuração anterior para o parâmetro *p*, o parâmetro *r* foi diminuído para aumentar o tamanho da rajada e foi configurado a taxa de envio em estado de erro *h* igual a 0,20. Com isso, em lugar de não se receber nenhum pacote durante as rajadas de perda, serão recebidos cerca de 20% dos pacotes, com a ocorrência de rajadas semelhante mas de maior duração. O conteúdo do arquivo de configuração comum foi:

```
0
1
2
0
```

No arquivo de configuração Gilbert.cfg:

```
0.001
0.002
0.20
```

O gráfico dos pacotes recebidos é apresentado na figura Figura 15.

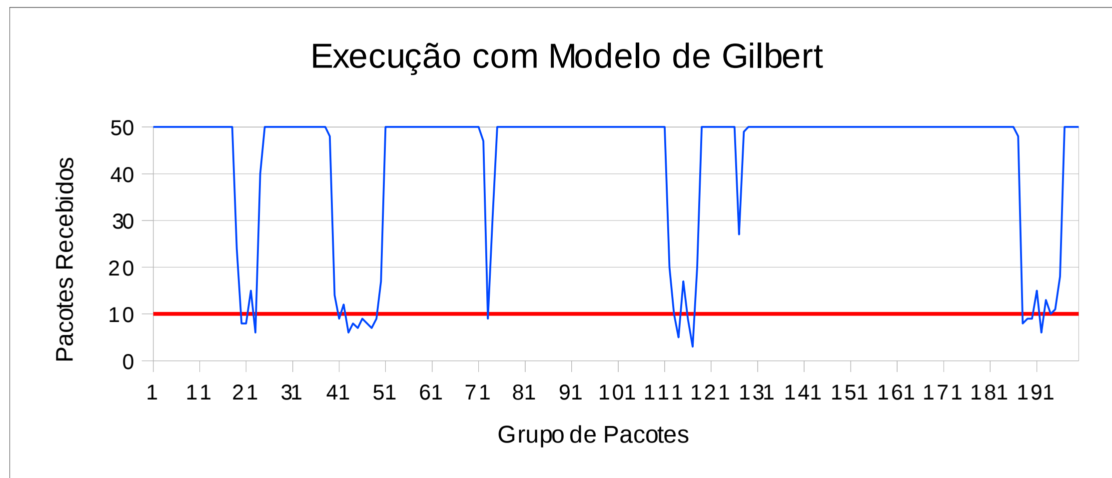

Figura 15: Gráfico para execução com Modelo de Gilbert.

No gráfico foi marcada, em vermelho, a linha correspondente a 10 pacotes, que equivale a 20% dos 50 pacotes por grupo. A não ser na rajada de perdas próxima ao grupo 130, que foi a mais curta da execução, as demais rajadas tiveram o número de pacotes recebidos próximos aos 20% configurados no parâmetro *h*.

Nessa execução o Servidor recebeu 86,49% dos pacotes enviadas pelo Cliente.

8.2.4 Simulação do Modelo de Gilbert-Elliott

O teste realizado com o modelo de Gilbert-Elliott mantém o valor dos parâmetros *p* e *r* do teste anterior, o valor do parâmetro *h* foi diminuído para 0,1 e o valor do parâmetro *k* do foi configurado em 0,85. Com essa configuração, há uma diferença razoável entre as taxas de recepção no estado bom e no estado ruim, facilitando a diferenciação de ditos estados. O conteúdo do arquivo de configuração comum foi:

```
0
1
2
0
```

No arquivo de configuração GilbertElliot.cfg:

```
0.001
0.002
0.10
0.85
```

O gráfico dos pacotes recebidos é apresentado na figura Figura 16.

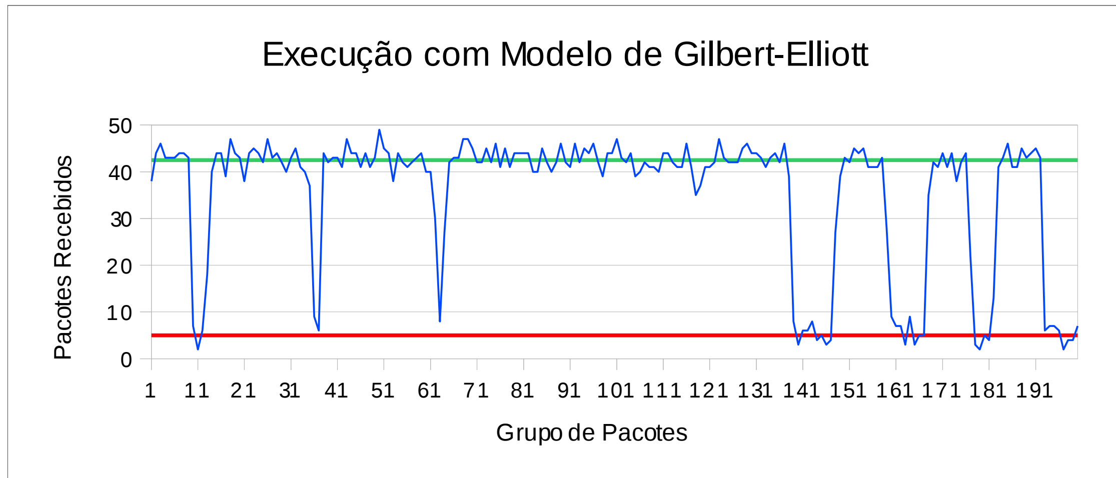

Figura 16: Gráfico para execução com Modelo de Gilbert-Elliott.

Foram marcadas no gráfico as taxas de recepção esperadas no estado bom, em verde, e do estado ruim, em vermelho. O comportamento está coerente com o esperado.

Nessa execução o Servidor recebeu 70,99% dos pacotes enviadas pelo Cliente.

8.2.5 Simulação do Modelo de Quatro Estados

Para este teste os parâmetros do modelo de quatro estados foram configurados de forma a ocorrerem rajadas de perda de forma semelhante ao modelo de Gilbert e de Gilbert-Elliot, mas com taxa pequena de perdas no estado bom e uma taxa de transmissão no estado ruim um pouco maior, e de maior duração. Os valores atribuídos aos parâmetros foram *p13* igual a 0,001, *p14* igual a 0,005, *p23* igual a 0,25, *p31* igual a 0,002 e *p32* igual a 0,006. O conteúdo do arquivo de configuração comum foi:

```
0
1
4
0
```

No arquivo de configuração Gilbert.cfg:

```
0.001
0.005
0.25
0.002
0.006
```

O gráfico dos pacotes recebidos é apresentado na figura Figura 17.

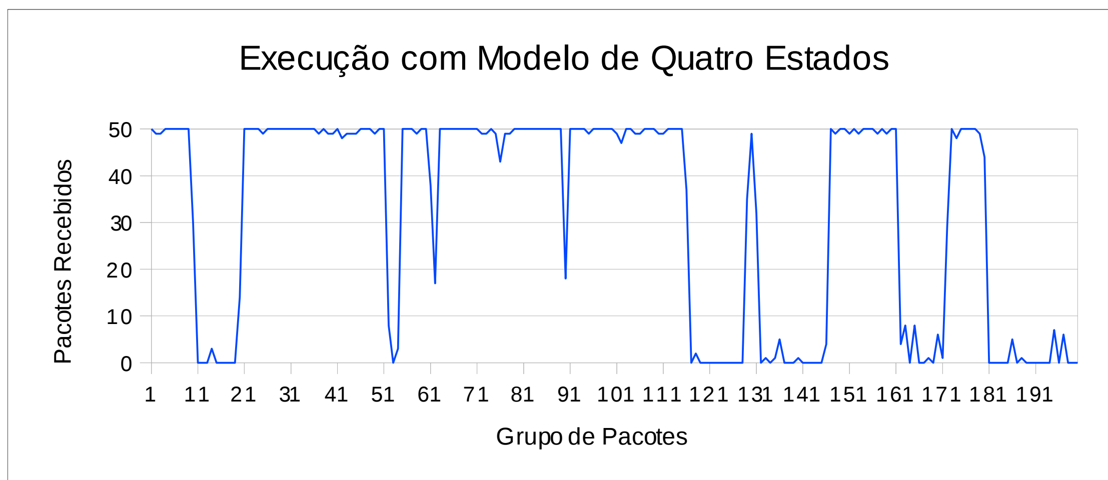

Figura 17: Gráfico para execução com Modelo de Gilbert.

No gráfico é possível observar os erros simples no estado bom, embora ocorram mais de um dentro de um mesmo grupo em alguns pontos. Também podem ser vistos poucos pacotes transmitidos no estado ruim. Com essa configuração ocorreram mais perdas e as rajadas tiveram maior duração.

Nessa execução o Servidor recebeu 63,76% dos pacotes enviadas pelo Cliente.

8.3 Influência no Desempenho

O objetivo deste teste foi comparar o desempenho quanto ao tempo de processamento da chamada ao método *receive()* da classe *DatagramSocket*, para o sistema Android sem o injetor, com o injetor instalado, porém desabilitado, e com o injetor habilitado. Além do SDK com o injetor de falhas de comunicação, foi compilado um SDK com o código fonte original do Android 2.3.

Para ter uma estimativa do tempo de processamento, foi testado quantos pacotes de uma rajada a aplicação Servidor consegue processar. Na aplicação Servidor modificada neste trabalho, a quantidade de instruções além da chamada ao método *receive()* foi reduzida o tanto quanto possível. Assim, a influência do *receive()* no tempo de processamento dos pacotes é maximizada, tendo-se uma medida mais próxima da variação de tempo inserida pelo injetor de falhas.

Neste teste foi utilizada a aplicação Cliente com o tempo de espera mínimo de 1 nanossegundo, para gerar uma rajada constante de pacotes com uma taxa suficiente para encher a fila de pacotes UDP.

Para verificar a influência com o injetor desabilitado, basta desabilitar a injeção de falhas na recepção no arquivo comum de configuração. No caso do injetor habilitado, como não se quer perder pacotes, foi utilizado o modelo de Gilbert com *p* igual a 1, *r* igual a 0 e *h* igual a 1. Com essa configuração, o estado seguinte do injetor sempre será o estado “1”, fazendo com que não sejam causadas perdas, ao mesmo tempo fazendo que o processamento para atualizar o estado seja maior do que caso o estado seguinte fosse “0”.

Foram feitas 5 execuções das aplicações Cliente e Servidor para cada caso, e foi armazenado o total de pacotes processados. Com esses valores foram calculadas as médias, utilizadas na comparação do desempenho.

Tabela 7: Resultado das execuções sem injetor, com o injetor desabilitado e com injetor habilitado.

Execução	Sem injetor	Injetor Desabilitado	Injetor Habilitado
1	98542	98558	98455
2	98421	98479	98272
3	98525	98433	98327
4	98434	98494	98414
5	98408	98454	98274
Media	98466.0	98483.6	98348.4
Taxa Relativa (%)	100.000	100.018	99.881

A Tabela 7 mostra a quantidade de pacotes processados em cada execução, a média e a taxa de processamento relativa ao sistema Android sem o injetor. Observando os resultados, é possível concluir que a alteração no desempenho não é significativa, sendo de 0,018% para o injetor desabilitado e de 0,119% com o injetor habilitado.

8.4 Influência no Timeout

Também foi feito o teste quanto a influência no tempo de espera para gerar a exceção de *timeout* do método *receive()*. O Cliente envia apenas um pacote no caso do sistema sem injetor, com o injetor instalado e desabilitado e com o injetor habilitado.

Para o caso do injetor habilitado, este recebe apenas o primeiro pacote e logo vai para o estado de perda, onde permanece pelo resto da execução. Para este caso também foi feito o teste com o Cliente enviando uma rajada de mensagens.

Tabela 8: Resultado das execuções sem injetor, com o injetor desabilitado e com injetor habilitado.

Execução	Sem injetor	Injetor Desabilitado	Injetor Habilitado	Injetor Habilitado (rajada)
1	9999173128	9993223521	10002434781	10001590371
2	10000681680	9995664259	9995734680	10001370612
3	9995327400	9991914406	9998942913	10000727281
4	9998079121	10000808992	10001224928	10001536365
5	9999987230	9995506474	9999080667	10001678918
6	9993424708	9997301854	9998368349	10001594215
7	9994957630	9998722746	9994794921	10000625881
8	9995366181	9997659053	10002200016	10005689828
9	9996625209	9991913142	9997061730	10001216467
10	9994024082	9994452296	9996197777	10001458127
Media	9996764636.9	9995716674.3	9998604076.2	10001748806.5
Tempo Relativo (%)	100.000	99.990	100.018	100.050

A Tabela 8 mostra o tempo transcorrido entre a recepção do primeiro pacote e a geração da exceção, medido pela aplicação Servidor para todos os casos de teste.

Foi observado que a maior diferença no tempo de espera, comparando com o sistema sem o injetor, foi de 0,050% e ocorreu com o injetor habilitado e o Cliente enviando uma rajada de pacotes. Isto ocorreu porque é o caso onde são feitos mais ajustes no *timeout*.

9 CONCLUSÃO

O Android é um sistema que vem constantemente ganhando mercado. Ao ser um sistema de código aberto, com vasta documentação e de simples utilização, até mesmo desenvolvedores menos experientes se vem interessados em criar aplicações. Somando isso com a facilidade de se disponibilizar as aplicações através do mercado do Android, é uma quantidade muito grande de aplicações vindas das mais diversas fontes, muitas delas desconhecidas e, portanto, pouco confiáveis.

Além disso, os dispositivos móveis, que são o alvo do sistema Android, estão sujeitos a operar em condições em que há uma grande probabilidade de interferências, atenuações e perdas de sinais, o que se traduz em uma grande probabilidade de ocorrência de falhas.

Por causa disso, é fundamental ter mecanismos de injeção de falhas, tanto pela perspectiva do desenvolvedor, que deve verificar a tolerância de suas aplicações a falhas, quanto pela perspectiva do usuário, que tem interesse em saber se o aplicativo que vai adquirir é confiável.

Neste trabalho, foram pesquisadas as opções para injetar falhas a nível da máquina virtual do Android, onde rodam todas as aplicações. Se descobriu que as ferramentas mais comumente utilizadas para injeção de falhas em Java não podem ser utilizadas, devido às diferenças nos bytecodes e na arquitetura da DMV, máquina virtual do Android, com respeito às máquinas virtuais Java convencionais.

Ao ser o projeto do Android disponibilizado de forma gratuita, foi possível explorar a solução de alterar as bibliotecas do núcleo do Java utilizado no sistema Andorid, onde se conseguiu implementar um injetor de falhas de comunicação UDP. Foi estudado todo o processo de preparação do sistema, obtenção do código fonte, alteração do código e compilação de um SDK, para poder ser testado no emulador de dispositivos.

Mesmo sem experiência na área de injeção de falhas, dedicando tempo para projetar, analisar e melhorar a implementação, foi possível criar um injetor de falhas razoavelmente versátil, que consegue simular de forma satisfatória alguns dos modelos mais difundidos de falhas de comunicação do tipo perda de pacotes. Além de ser flexível, o injetor mostrou baixa alteração no desempenho do sistema, que é importante para não se mascarar outras possíveis falhas no sistema.

REFERÊNCIAS

[1] ACKER, E. V. ; WEBER, T. S. ; CECHIN, S. L.; “Injeção de falhas para validar aplicações em ambientes móveis”. In: Workshop de Testes e Tolerância a Falhas, 11., 2010, Gramado. XI Workshop de Testes e Tolerância a Falhas. Porto Alegre : Sociedade Brasileira de Computação, 2010, volume 1, pp. 61-74.

[2] AVIZIENIS, A. ; LAPRIE, J. ; RANDELL, B. ; LANDWEHR, C. ; “Basic concepts and taxonomy of dependable and secure computing”. In: IEEE trans. on dependable and secure computing, 2004, volume 1, pp. 11–33.

[3] BRANQUINHO, O. C.; REGGIANI, N.; ANDREOLLO, A. G., “Redes de comunicação de dados sem fio - uma análise de desempenho”. In: ISA Show, 2005

[4] DOBLER, R. J. ; WEBER, T. S. ; CECHIN, S. L. ; “Porte do Injetor de Falhas Firmament para o Ambiente Android”. In: Escola Regional de Redes de Computadores, 2010, Alegrete, RS, 2010, volume 1, pp. 9-15.

[5] ELLIOTT, E.O. ; "Estimates of Error Rates for Codecs on Burst-Noise Channels", In: Bell System Technical Journal 42, 1963. pp. 1977-1997

[6] GILBERT, E. N. ; “Capacity of a Burst-Noise Channel”. In: Bell System Technical Journal, vol.39, Setembro 1960.

[7] HOHLFELD, O., GEIB, R., HaßLINGER, G.; “Packet Loss in Real-Time Services: Markovian Models Generating QoE Impairments”. In: Proceedings of IWQoS, 2008. pp. 239-248

[8] HSUEH, M.; TSAI, T. K.; IYER, R. K. ; “Fault Injection Techniques and Tools” ; Computer, 1997, pp. 75-82.

[9] JACQUES-SILVA, G. ; JUNG DREBES, R. ; GERCHMAN, J. ; WEBER, T. S. ; “FIONA: a fault injector for dependability evaluation of Java-based network applications”, In: IEEE Symposium on Network Computing and Applications, 2004

[10] JACQUES-SILVA, G. et al. ; “A network-level distributed fault injector for experimental validation of dependable distributed systems,” in Proc. of the 30th COMPSAC, Chicago: IEEE, 2006, pp. 421–428.

[11] LOOKER, N. ; MUNRO, M. ; XU, J. ; "A Comparison of Network Level Fault Injection with Code Insertion”. In: Computer Software and Applications Conference, 2005, volume 1, pp. 479 - 484

[12] MARMITT, H. F. ; CECHIN, S. L. ; WEBER, T. S. ; “Modelos para injeção de falhas em ambiente móveis”. In: Escola Regional de Redes de Computadores, 2010, Alegrete, RS. ERRC - 8. Escola Regional de Redes de Computadores. Alegrete : Unipampa, 2010. volume 1, pp. 1-8.

[13] MARTINS, E.; RUBIRA, C.M.F.; LEME, N.G.M.; “Jaca: a reflective fault injection tool based on patterns”, International Conference on Dependable Systems and Networks, Inst. of Comput., UNICAMP, 2002

[14] MENEGOTTO, C. C. ; WEBER, T. S. ; “Injeção de falhas de comunicação em aplicações java multiprotocolo,” in Anais WTF 11, Gramado: Porto Alegre: SBC, 2010, pp. 75–88.

[15] ROCHA, T. ; TOLEDO, M. B. F. de ; “Mecanismos de Adaptação para Transações em Ambientes de Computação Móvel”. In: Revista IEEE Latin America Transactions, 2007

[16] SALSANO, S. ; LUDOVICI, F. ; ORDINE, a. ; “Definition of a general and intuitive loss model for packet networks and its implementation in the Netem module in the Linux kernel”. Technical report, University of Rome “Tor Vergata”, October 2009.

[17] TANENBAUM, A. S. ; “Computer Networks”. Prentice Hall, terceira edição, 1996.

[18] VACARO, J. C. ; WEBER, T. S. ; “Injeção de falhas na fase de teste de aplicações distribuídas,” in Anais do Simpósio Brasileiro de Engenharia de Software 20, SBC, 2006, volume. 1, pp. 161–176.

[19] WEBER, T. S. ; “Tolerância a falhas: conceitos e exemplos”. In: Programa de Pós-Graduação em Computação, UFRGS, 2003.

[20] Open Handset Alliance website, www.openhandsetalliance.com, acessado em junho de 2011

[21] Android website, seções market e developers, www.android.com, acessado em junho de 2011

[22] Javassist, Shigeru Chiba website, http://www.csg.is.titech.ac.jp/~chiba/javassist/, acessado em junho de 2011

[23] JVMTI, Oracle website, http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html, acessado em junho de 2011

[24] BRAY, J. ; COLLINS, B. ; KOBIE, N. ; “What's Killing your Wi-Fi?”. In: PC Pro website, http://www.pcpro.co.uk/features/367672/whats-killing-your-wi-fi, acessado em junho de 2011

[25] VoIP Troubleshooter website ; “Packet Loss Burstiness”. In: http://www.voiptroubleshooter.com/indepth/burstloss.html, acessado em junho de 2011

[26] Wikipedia, “IEEE 802.11”. In: http://en.wikipedia.org/wiki/IEEE_802.11, acessado em junho de 2011

[27] MORIMOTO, C. E. ; “Redes Wireless, Parte 2: padrões”, In: http://www.hardware.com.br/tutoriais/padroes-wireless/pagina3.html, 2008, acessado em junho de 2011

[28] Eclipse website, seção downloads: http://www.eclipse.org/downloads/, acessado em junho de 2011

ANEXO – ARTIGO DA PROPOSTA DESTE TRABALHO

Estudo de Injeção de Falha para a Máquina Virtual do Sistema Android

Alexandre Felin Gindri[1], Taisy Weber[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{afgindri,taisy}@inf.ufrgs.br

Resumo. *O Android é um sistema de desenvolvimento para aparelhos móveis que vem mantendo a atenção da comunidade tecnológica. Este sistema de desenvolvimento permite a fácil criação de aplicações que, assim como as aplicações nativas, utilizem toda a capacidade dos aparelhos. Com essa facilidade, surgem diversas aplicações, provenientes de diferentes desenvolvedores. Para se ter confiabilidade nesse cenário é desejável poder testar a tolerância dessas aplicações a falhas. Neste relatório preliminar serão apresentadas as possíveis abordagens para a implementação de um injetor de falhas para o Android em nível da máquina virtual, onde rodam as aplicações.*

Introdução

Nos últimos anos os computadores portáteis tem ganho lugar frente aos computadores de mesa. A possibilidade de utilizar um notebook em lugar de um desktop em casa e outro na oficina, permite ao funcionário ter seus arquivos e programas a disposição, sem precisar configurar e manter dois sistemas.

As aplicações mais comuns para os usuários, relacionadas a comunicação (e-mail, bate-papo, redes sociais), entretenimento (jogos, conteúdo multimídia como músicas e vídeos), pesquisas e compras, entre outros, já são suportadas por dispositivos moveis de pequeno porte, como são os handhelds: palm tops, smartphones e aparelhos semelhantes. Para o uso profissional, é extremamente interessante para um técnico poder acessar os manuais de um produto quando em campo, ou para um médico obter a ficha de um paciente durante o atendimento domiciliar.

O avanço das tecnologias permite agregar maior poder computacional em dispositivos menores. Porém ainda há alguns pontos que comprometem a performance para tais dispositivos. As limitações de tamanho e consumo de potência podem trazer como consequência o uso de componentes com menor tolerância a falhas (menos dispositivos redundantes) e maior taxa de erros (menor tensão nos componentes eletrônicos é mais facilmente modificada por interferência).

Quando colocado em um ambiente móvel, com interferências entre sinais e movimentação, o desafio de se fazer uma aplicação que não apresente defeitos, ou seja, que tenha dependabilidade [AVIZIENIS], se torna ainda maior. É necessário incluir mecanismos de tolerância a falhas que alcancem esse objetivo quando em operação no ambiente real.

É inviável recriar todos os cenários e depender da ocorrência natural de falhas. Para simular a ocorrência das falhas, e assim conseguir testar a tolerância do sistema, é utilizado o método de injeção de falhas, que é um dos mais consolidados na literatura. O método permite que sejam provocados os padrões de falhas semelhantes aos que podem ocorrer durante a operação do dispositivo.

Um sistema que tem crescido de forma notável quanto a utilização nos novos dispositivos móveis é o Android. Este é um sistema desenvolvido e publicado pela Google (e posteriormente pela Open Handset Alliance) como open source, de forma a permitir que desenvolvedores possam criar aplicações próprias e vendê-las num mercado mundial. Recentemente este mercado, que é suportado pelo próprio projeto Android, incluiu, entre outros 20 países, o Brasil.

Desenvolvedores que queiram criar aplicações tolerantes a falhas precisam de um método de teste para descobrir os pontos críticos do projeto. Usuários que adquirem uma aplicação de uma fonte desconhecida podem eventualmente desejar testar se esta cumpre com os requisitos de dependabilidade que se espera dessa aplicação.

As aplicações do Android rodam em uma máquina virtual própria do sistema, a Dalvik Virtual Machine, que é uma máquina otimizada para sistemas com limitações de memória, processamento e bateria.

O objetivo deste trabalho é realizar um estudo quanto a possibilidade de se injetar falhas no nível da máquina virtual do Android, tentando diminuir a intrusividade do injetor no desempenho das aplicações.

Android

O projeto foi iniciado pela Android Inc., e continuado pela Google em 2005, ao adquirir a empresa. O objetivo era fazer uma plataforma de telefone flexível, aberta e de fácil migração para fabricantes. Desde 2007 o projeto pertence a Open Handset Alliance, um consórcio de varias companhias (incluindo a Google), e foi anunciado que o Android seria o seu primeiro produto: uma plataforma para sistemas móveis construída sobre o Kernel Linux 2.6. Desde outubro de 2008 o código foi publicado sob a Licença Apache (Apache License), permitindo que os desenvolvedores ofereçam extensões proprietárias, sem a necessidade de liberação do código.

No segundo trimestre de 2010, segundo pesquisa divulgada pela empresa Canalys, 34% dos smartphones no mercado americano executam Android, e o crescimento do número de aparelhos que usam Android no mundo nesse mesmo período foi de 886% [DOBLER]. Em setembro de 2010 o Android anunciou em seu site que o Android Market, um sistema de mercado onde os desenvolvedores podem vender suas aplicações aos usuários, adicionou suporte a 20 novos países, incluíndo o Brasil. Com isso o Android Market oferece suporte para desenvolvedores de 29 países e usuários consumidores de 32 países.

Esse rápido crescimento é também devido a facilidade de começar a desenvolver aplicações. O sistema de desenvolvimento (SDK) do Android é simples de ser instalado e com abundante documentação. Os requisitos pré instalação são o Java SDK (JDK), e é fortemente recomendada a instalação do Eclipse junto com um plugin Android Development Tool (ADT). Também é aconselhado baixar a versão para iniciantes do Android, que contém as ferramentas básicas, podendo ser usado para posteriormente baixar as demais ferramentas.

As aplicações são desenvolvidas em linguagem Java, e cada uma é executada no Android em um processo Linux separado. Cada aplicação tem também uma instância da máquina virtual utilizada, chamada Dalvik Virtual Machine (DVM), que é uma máquina semelhante a Java, mas com otimizações de memória e utilização de recursos em aparelhos móveis. Não é considerada uma máquina virtual Java pois ela não opera o mesmo bytecode, mas ao invés disso o bytecode é transformado por uma ferramenta do Android SDK no formato interpretado pela DVM.

O sistema foi desenvolvido para permitir que as aplicações de terceiros utilizem todos os recursos disponíveis do dispositivo da mesma forma que as aplicações nativas. Além disso, as aplicações podem fazer uso de elementos de outras aplicações, os chamados componentes. O sistema inicia um processo dessa aplicação e instancia os objetos necessários desse componente que será chamado.

Pela perspectiva do desenvolvedor, existem muitas possibilidades para explorar os recursos que o aparelho tem a oferecer, possibilitando a criação de aplicações complexas. Ainda, com a possibilidade de usar componentes de outras aplicações, dos quais não se possui o código, pode se ter uma fonte de erros que comprometam o funcionamento na presença de falhas.

Pela perspectiva do usuário, as fontes que disponibilizam as aplicações (empresas ou desenvolvedores independentes) são diversas, e não necessariamente confiáveis.

Em ambos os casos é desejável um mecanismo de teste que permita validar as aplicações e garantir o nível de dependabilidade esperado destas.

Dalvik Virtual Machine - DVM

O projeto do Android foi feito para sistemas com recursos de memória, processamento e consumo limitados. Por esse motivo a utilização de uma máquina virtual convencional não seria uma opção viável.

Criada por Dan Bornstein, a Dalvik Virtual Machine é a máquina virtual que roda no Android. Esta máquina foi desenvolvida com otimizações de uso de CPU, menor gasto de memória e com otimizações para rodar mais de uma instância.

A diferença de máquinas virtuais Java, a DVM não roda bytecodes Java tradicionais. No SDK do Android é utilizada a ferramenta dx, que converte os arquivos de bytecode .class contidos em um arquivo .jar para o formato do arquivo da DVM de formato .dex. Nesse formato, informações repetidas, como por exemplo assinaturas de funções referenciadas de outros arquivos, são minimizadas com a utilização de vários apontadores.

Outra diferença significativa é que a DVM utiliza uma arquitetura baseada em registradores, em lugar de uma pilha de dados. Com essa mudança houve uma diminuição no número de instruções e de unidades de código, e embora haja um aumento no número de bytes no stream de instruções, os bytes são consumidos aos pares.

Ambientes móveis

A operação de dispositivos em ambientes móveis se caracteriza pela computação com recursos limitados, principalmente por tamanho e consumo de potência, e pela comunicação numa rede não cabeada, com movimentação relativa entre os dispositivos.

Com o aumento da utilização de equipamentos eletrônicos que operam ou emitem ondas nas mesmas faixas de frequências, aumenta também o problema da interferência entre sinais. Por exemplo, a faixa de frequência de 2,4 a 2,4835 GHz, onde não é requerida autorização, se torna problemática para estabelecimento de redes WLAN [BRANQUINHO].

Somado ao problema da interferência está o grande dinamismo presente em ambientes móveis [ROCHA]. A alteração na distância dos dispositivos devido a movimentação gera mudanças na atenuação dos sinais, nos atrasos de recepção, nas reflexões, refrações e difrações sofridas pelo sinal durante a propagação, entre outros.

Dispositivos móveis, portanto, estarão susceptíveis a diversas falhas com respeito comunicação de dados; perdas de mensagens, períodos de desconexão, corrompimento de mensagens, trocas de ordem de mensagens. Equipamento com limitações (Potência), causa pra aplicação perdas de conexão, alterações nas taxas de transmissão (banda), perdas de mensagens, alteração na ordem, corrompimento dos dados.

Como a comunicação de dados está presente em grande parte de aplicações de dispositivos móveis, uma parte importante no teste destes sistemas é conseguir injetar esses falhas que simulem esses cenários.

Injeção de falhas

Existem dois tipos principais de injetores de falhas: os de hardware e os de software. Os implementados em hardware são módulos acoplados que não utilizam recursos do sistema e assim não afetam seu desempenho. Contudo, esse tipo de injetor é geralmente custoso, e a sua utilização é restrita a quantidade de pontos de inserção. Injetores de falhas em software, por outro lado, tem somente o custo de ser implementado e são mais flexíveis, embora tenham maior interferência [ACKER].

O mecanismo injetor de falhas deve, preferencialmente, minimizar a intrusividade no sistema, diminuindo o impacto no desempenho normal deste. Se o injetor influenciar no desempenho de uma aplicação, ao ser retirado implica em um sistema diferente que pode, por consequência, apresentar defeitos diferentes.

Suponha-se que para testar um módulo que processa uma série de dados e coloca cada resultado em uma fila, é adicionado um módulo de injeção de falhas, fazendo com que o processamento tenha o tempo duplicado; ao se retirar este módulo de injeção de falhas a fila irá receber dados no dobro da taxa, podendo eventualmente ficar cheia e apresentar defeito.

Já existem em andamento trabalhos sobre portes de injetores de falhas para o Kernel do Android, e estudos de modelos de injeção de falhas para a plataforma. Este trabalho tem como objetivo estudar a possibilidade de implementar um injetor de falhas a nível da máquina virtual, a DVM.

Injeção de falhas em java

Para plataforma Java, existem trabalhos que apresentam injetores de falhas, como JACA [MARTINS], e FIONA [JACQUES-SILVA]. Duas técnicas usadas em injeção de falhas na máquina virtual Java serão apresentadas a seguir.

Javassist

Javassist é uma biblioteca de classe para edição de bytecode Java. Permite que classes sejam criadas durante a execução e a modificar um arquivo de classe quando a máquina virtual for carregá-la.

O pacote Javassit provê classes e métodos que permitem obter containers com o pool (repositório) de classes, recuperar arquivos de classes específicas e seus bytecodes, realizar modificações em classes e salvá-las.

Para injetar falhas de comunicação UDP em uma aplicação utilizando Javassist, pode ser carregada a classe java.net.DatagramSocket e modificar os métodos de envio e recepção para métodos que realizem a injeção das falhas.

Como as alterações são feitas a nível de bytecodes Java, o pacote Javassist não pode ser usado diretamente na DVM.

JVMTI

A JVMTI é uma interface de programação usada em ferramentas de desenvolvimento e monitoração. Permite inspecionar o estado e controlar a execução de aplicações em máquinas virtuais Java.

O JVMTI tem um componente cliente chamado de agente, sendo este uma biblioteca dinâmica que recebe notificações de acontecimento de eventos. Dentre as notificações, o agente pode receber a notificação da carga de uma classe, e chamar funções para alterar o estado da máquina virtual.

No injetor FIONA, o agente JVMTI recebe a notificação da carga da classe java.net.DatagramSocket na inicialização da máquina virtual, e antes da carga, substituindo a imagem da classe por uma imagem instrumentada. A aplicação chama de forma transparente os recursos da classe, não precisando de modificações no código fonte.

O implementação de JVMTI em diferentes máquinas virtuais é suportado pela Oracle, que adquiriu a Sun, criadora da plataforma Java. Para a máquina virtual do Android não há uma implementação de JVMTI.

Proposta do trabalho

A proposta deste trabalho de graduação é injetar falhas de comunicação UDP na aplicação sob teste a partir da máquina virtual do Android.

Na primeira etapa do trabalho, apresentada neste artigo, foi feita uma pesquisa para embasamento teórico sobre o sistema alvo, as técnicas utilizadas, ferramentas semelhantes.

Como não é possível utilizar ferramentas já disponíveis, como as apresentadas Javassist ou JVMTI, para a etapa seguinte deve ser feito um estudo mais aprofundado dos bytecodes dos arquivos .dex utilizados pela máquina virtual do Android.

Será necessário saber como são feitas as definições das entidades, como tipos, métodos, variáveis, para assim conseguir incluir novas entidades ou modificar as existentes. Também deve se entender como é feita a chamada a um método, para poder localizar e desviar uma chamada para outro método.

Alternativamente pode se procurar ferramentas prontas para a manipulação de bytecodes Dalvik que realizem as operações necessárias.

Com esse conhecimento dos bytecodes da DVM, seria possível procurar pelas chamadas aos métodos de send e receive da classe de sockets UDP da máquina virtual, e substituir por chamadas aos métodos que implementam o comportamento com falha.

Finalmente, deve ser implementado um método que simule algum padrão de falha, como perdas de mensagens, alteração da ordem de recepção ou alteração de conteúdo.

Planejamento das atividades

A seguir será apresentado o planejamento por mês. Durante todo o período será feito em paralelo o aprimoramento do texto, incluindo as informações encontradas e os resultados das atividades. No final

Janeiro

Durante o mês se espera encontrar ferramentas de edição de bytecodes Dalvik que possam colaborar com o desenvolvimento do injetor de falhas.

Fevereiro

Deve ser feito um estudo dos bytecodes Dalvik, com o objetivo de entender sua estrutura e poder localizar as entidades que devem ser modificadas.

Março

Na primeira metade do mês, somando o conhecimento das ferramentas disponíveis com o dos bytecodes, deve se fazer uma aplicação simples, com alguma chamada de função definida no próprio código da aplicação, para ser interceptada e modificada.

Conseguindo interceptar e modificar essa chamada a função, o próximo passo será criar ou reutilizar uma aplicação que tenha comunicação através de transmissão de datagramas com sockets UDP, e de forma semelhante conseguir interceptar chamadas a função de recepção / envio.

Abril

Se dará continuidade à tarefa do mês anterior de interceptar chamadas de sockets UDP. Como resultado é esperado poder injetar algum tipo de falha, como perda ou alteração de conteúdo dos datagrama, em uma aplicação própria que tenha diversas chamadas as funções de socket UDP.

Maio

Elaborar uma ferramenta que faça a localização e a injeção da falha.

Junho

Realizar testes sobre as aplicações próprias e sobre aplicações de outra fonte. Incluir os resultados para finalizar o texto e preparar a apresentação do trabalho.

Referências

DOBLER, R. J. ; WEBER, T. S. ; CECHIN, S. L. ; “Porte do Injetor de Falhas Firmament para o Ambiente Android”. In: Escola Regional de Redes de Computadores, 2010, Alegrete, RS, 2010. v. 1. p. 9-15.

ACKER, E. V. ; WEBER, T. S. ; CECHIN, S. L.; “Injeção de falhas para validar aplicações em ambientes móveis”. In: Workshop de Testes e Tolerância a Falhas, 11., 2010, Gramado. XI Workshop de Testes e Tolerância a Falhas. Porto Alegre : Sociedade Brasileira de Computação, 2010. v. 1. p. 61-74.

MARMITT, H. F. ; CECHIN, S. L. ; WEBER, T. S. ; “Modelos para injeção de falhas em ambiente móveis”. In: Escola Regional de Redes de Computadores, 2010, Alegrete, RS. ERRC - 8. Escola Regional de Redes de Computadores. Alegrete : Unipampa, 2010. v. 1. p. 1-8.

AVIZIENIS, A. ; LAPRIE, J. ; RANDELL, B. ; LANDWEHR, C. ; “Basic concepts and taxonomy of dependable and secure computing”. In: IEEE trans. on dependable and secure computing, 2004. v. 1 p. 11–33.

MARTINS, E.; RUBIRA, C.M.F.; LEME, N.G.M.; Jaca: a reflective fault injection tool based on patterns, International Conference on Dependable Systems and Networks, Inst. of Comput., UNICAMP, 2002

BRANQUINHO, O. C.; REGGIANI, N.; ANDREOLLO, A. G., “Redes de comunicação de dados sem fio - uma análise de desempenho”. In: ISA Show, 2005

ROCHA, T. ; TOLEDO, M. B. F. de ; “Mecanismos de Adaptação para Transações em Ambientes de Computação Móvel”. In: Revista IEEE Latin America Transactions , 2007

WEBER, T. S. ; “Tolerância a falhas: conceitos e exemplos”. In: Programa de Pós-Graduação em Computação, UFRGS, 2003

JACQUES-SILVA, G., JUNG DREBES, R., GERCHMAN, J., WEBER, T. S., “FIONA: a fault injector for dependability evaluation of Java-based network applications”, In: IEEE Symposium on Network Computing and Applications, 2004

Open Handset Alliance website, www.openhandsetalliance.com.

Android website, market and developers subsection, www.android.com.

Javassist, Shigeru Chiba website, http://www.csg.is.titech.ac.jp/~chiba/javassist/

JVMTI, Oracle website,

http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

APÊNDICE A: CÓDIGO PARCIAL DO DATAGRAM SOCKET COM O INJETOR DE FALHAS

Atributos adicionados ao DatagramSocket:

```
  // [Gindri]: used for a 4 state Markov chain model
  private double[][] markovChain = {
    { 0.0, 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 0.0, 0.0 },
    { 0.0, 0.0, 0.0, 0.0 }
  };

  // [Gindri]: initialize with long Good state (if not given in config
file)
  private int currentStateID = 0;

  // [Gindri]: enables / disables fault injection on each direction
  private boolean injectFaultsOnSend = false;
  private boolean injectFaultsOnReceive = false;

  // [Gindri]: stores socket's actual receive timeout (not the current
on SO)
  private int actualTimeout = 0;
```

Um dos construtores DatagramSocket:

```
  public DatagramSocket(int aPort, InetAddress addr) throws
SocketException {
    super();
    checkListen(aPort);
    createSocket(aPort, null == addr ? Inet4Address.ANY : addr);

    // [Gindri]: Initialize parameters used in fault injection models
    initFaultModels();
  }
```

Função de inicialização do Injetor:

```
  void initFaultModels() {
    // [Gindri]: will store the result of any model initialization
    int ret = -1;

    // [Gindri]: aux for configuring model type
    int modelType = 0;

    // [Gindri]: aux for configuring direction
```

```
    int auxInjectSend = 0;
    int auxInjectReceive = 0;

    // [Gindri]: aux for configuring initial state (as it may have
different meanings)
    int initialState = 0;

    // [Gindri]: aux for reading from file
    byte[] line;
    FileInputStream fStream;
    BufferedReader in = null;

    injectFaultsOnSend = false;
    injectFaultsOnReceive = false;

    // [Gindri]: initialize probabilities from config file
    try {
      fStream = new
FileInputStream("/data/local/tmp/UDP_loss_injector.cfg");
      in = new BufferedReader(new InputStreamReader(fStream));

      // [Gindri]: break from this do-while if our input is not ready
      do {

        // [Gindri]: get Fault Injection direction
        if (!in.ready()) break;
        line = in.readLine().getBytes();
        auxInjectSend = Integer.parseInt(new String(line));

        if (!in.ready()) break;
        line = in.readLine().getBytes();
        auxInjectReceive = Integer.parseInt(new String(line));

        // [Gindri]: Check if injection is at least enabled in one
direction
        if ((auxInjectSend == 0) && (auxInjectReceive == 0)){
          break;
        }

        // [Gindri]: get model type
        if (!in.ready()) break;
        line = in.readLine().getBytes();
        modelType = Integer.parseInt(new String(line));

        // [Gindri]: get initial state
        if (!in.ready()) break;
        line = in.readLine().getBytes();
        initialState = Integer.parseInt(new String(line));

        // [Gindri]: test wich model is it to know how to initialize
it
        switch (modelType){
          case 0: // Bernoulli model
            ret = initBernoulli(initialState);
            break;
          case 1: // Simple Gilbert model
            ret = initSimpleGilbert(initialState);
            break;
```

```
          case 2: // Gilbert model
            ret = initGilbert(initialState);
            break;
          case 3: // Gilbert-Elliott model
            ret = initGilbertElliott(initialState);
            break;
          case 4: // VoIP Troubleshooter or Salsano model
            ret = initFourStateModel(initialState);
            break;
          case 5: // custom 4 state markov chain
            ret = initFourStateMarkovChain(initialState);
            break;
          default:
            ret = -1;
            break;
        }

        if (ret == 0) {
          // [Gindri]: if ret is 0, everything is ok. Enable send
and/or receive fault
          if ((auxInjectSend != 0)){
            injectFaultsOnSend = true;
          }
          if ((auxInjectReceive != 0)){
            injectFaultsOnReceive = true;
          }
        }

      } while (false);

    } catch (IOException e) {
      //ret = -1;
    } catch (NumberFormatException eNum){
      //ret = -1;
    } finally {
      if (in != null){
        try{
          in.close();
        } catch (Exception ignored) {}
      }
    }
  }
```

Função de inicialização do modelo de Gilbert-Elliot:

```
  int initGilbertElliott(int initialState){
    int ret = 0;

    double p, r, h, k;

    // [Gindri]: aux for reading from file
    byte[] line;
    FileInputStream fStream;
    BufferedReader in = null;

    try {
      fStream = new
FileInputStream("/data/local/tmp/GilbertElliott.cfg");
```

```
    in = new BufferedReader(new InputStreamReader(fStream));

    if (!in.ready()) ret = -1;
    line = in.readLine().getBytes();
    p = Double.parseDouble(new String(line));

    if (!in.ready()) ret = -1;
    line = in.readLine().getBytes();
    r = Double.parseDouble(new String(line));

    if (!in.ready()) ret = -1;
    line = in.readLine().getBytes();
    h = Double.parseDouble(new String(line));

    if (!in.ready()) ret = -1;
    line = in.readLine().getBytes();
    k = Double.parseDouble(new String(line));

    if (ret != -1) {

      // [Gindri]: Good (with no loss) state probability transitions
      markovChain[0][0] = (1-p)*(k);
      markovChain[0][1] = (p)*(h);
      markovChain[0][2] = (p)*(1-h);
      markovChain[0][3] = (1-p)*(1-k);

      // [Gindri]: Bad (with no loss) state probability transitions
      markovChain[1][0] = (r)*(k);
      markovChain[1][1] = (1-r)*(h);
      markovChain[1][2] = (1-r)*(1-h);
      markovChain[1][3] = (r)*(1-k);

      // [Gindri]: Bad (with loss) state probability transitions
      markovChain[2][0] = (r)*(k);
      markovChain[2][1] = (1-r)*(h);
      markovChain[2][2] = (1-r)*(1-h);
      markovChain[2][3] = (r)*(1-k);

      // [Gindri]: Good (with loss) state probability transitions
      markovChain[3][0] = (1-p)*(k);
      markovChain[3][1] = (p)*(h);
      markovChain[3][2] = (p)*(1-h);
      markovChain[3][3] = (1-p)*(1-k);


      // [Gindri]: Accumulate probabilities now to avoid
subtractions later on send/receive
      for (int state = 0; state < 4; state++) {
        markovChain[state][1] += markovChain[state][0];
        markovChain[state][2] += markovChain[state][1];
        markovChain[state][3] = 1; // Sum of all probabilities is 1
      }

      if (initialState == 0){
        if (Math.random() < k){
          currentStateID = 0;
        } else {
          currentStateID = 3;
```

```
          }

        } else {
          if (Math.random() < h){
            currentStateID = 1;
          } else {
            currentStateID = 2;
          }
        }
      }

    } catch (IOException e) {
      ret = -1;
    } catch (NumberFormatException eNum){
      ret = -1;
    } finally {
      if (in != null){
        try{
          in.close();
        } catch (Exception ignored) {}
      }
    }

    return ret;
  }
```

Método Receive, apenas o laço do injetor:

```
  // [Gindri]: a randomly generated value, to compare with lossProb
  double randVal = 0;

  // [Gindri]: before starting, get reference time in milliseconds
  start_time_ref = System.currentTimeMillis();

  if (actualTimeout > 0){
    // [Gindri]: set the actual timeout, as it may be smaller
    impl.setOption(SocketOptions.SO_TIMEOUT,
Integer.valueOf(actualTimeout));
  }
  // [Gindri]: We are going to loop while we want to lose packets
  do {

    if (address != null || security != null) {
      // The socket is connected or we need to check security
permissions

      // Check pack before peeking
      if (pack == null) {
        throw new NullPointerException();
      }

      // iterate over incoming packets
      while (true) {
        copy = false;

        // let's get sender's address and port
        try {
          // if we peaked and there is no timed out set, there is data
```

```
          senderPort = impl.peekData(tempPack);
          senderAddr = tempPack.getAddress();
        } catch (SocketException e) {
          if (e.getMessage().equals(
              "The socket does not support the operation")) {
            // receive packet to temporary buffer
            tempPack = new DatagramPacket(new
byte[pack.getCapacity()],
                pack.getCapacity());
            impl.receive(tempPack);
            // tempPack's length field is now updated, capacity is
unchanged
            // let's extract address & port
            senderAddr = tempPack.getAddress();
            senderPort = tempPack.getPort();
            copy = true;
          } else {
            throw e;
          }
        }

        if (address == null) {
          // if we are not connected let's check if we are allowed to
          // receive packets from sender's address and port
          try {
            security.checkAccept(senderAddr.getHostName(),
                senderPort);
            // address & port are valid

            break;
          } catch (SecurityException e) {
            if (!copy) {
              // drop this packet and continue
              impl.receive(tempPack);
            }
          }
        } else if (port == senderPort && address.equals(senderAddr)) {
          // we are connected and the packet came
          // from the address & port we are connected to
          break;
        } else if (!copy) {
          // drop packet and continue
          impl.receive(tempPack);
        }
      }
    }

    if (copy) {
      System.arraycopy(tempPack.getData(), 0, pack.getData(), pack
          .getOffset(), tempPack.getLength());
      // we shouldn't update the pack's capacity field in order to be
      // compatible with RI
      pack.setLengthOnly(tempPack.getLength());
      pack.setAddress(tempPack.getAddress());
      pack.setPort(tempPack.getPort());
    } else {
      pack.setLength(pack.getCapacity());
      impl.receive(pack);
```

```
      // pack's length field is now updated by native code call;
      // pack's capacity field is unchanged
    }

    // [Gindri]: If we are here, we received a packet. Test if we
should inject a fault (loss)


    if (currentStateID < 2) {
      // [Gindri]: get a random value in range [0,1)
      randVal = Math.random();

      for (int i = 0; i < 4; i++){
        // [Gindri]: if we are in the probability to go to i state,
then set new currentStateID and break
        if (randVal < markovChain[currentStateID][i]){
          currentStateID = i;
          break; // for
        }
      }
      break; // do-while(true)
    } else {
      // [Gindri]: get a random value in range [0,1)
      randVal = Math.random();

      for (int i = 0; i < 4; i++){
        // [Gindri]: if we are in the probability to go to i state,
then set new currentStateID and break
        if (randVal < markovChain[currentStateID][i]){
          currentStateID = i;
          break; // for
        }
      }
      if (actualTimeout > 0){
        // [Gindri]: this message do not count, discount from timeout
        elapsed_time = (int)(System.currentTimeMillis() -
start_time_ref);

        // [Gindri]: make sure they are not equal, they would disable
the timeout
        if (elapsed_time >= actualTimeout){
          throw new SocketTimeoutException();
        } else {
          // [Gindri]: set the timeout discounting elapsed time
          impl.setOption(SocketOptions.SO_TIMEOUT,
Integer.valueOf(actualTimeout - elapsed_time));
        }
      }

    }

  } while (true);

  // [Gindri]: If we are here, we did not loss the last received
packet. This call will return
```

APÊNDICE B: CÓDIGO DA APLICAÇÃO CLIENTE

```
import java.nio.ByteBuffer;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class AverageClient {

  /* Total amount of messages to send */
  public static final int PACKETS_TO_SEND = 10000;

  public static void main(String[] args){
    try {
      InetAddress serverAddr = InetAddress.getByName("localhost");
      DatagramSocket socket = new DatagramSocket();

      /* Prepare data to be sent */
      byte[] buf = new byte[1472];
      DatagramPacket packet;

      /* Creates UDP packet with data and destination */
      packet = new DatagramPacket(buf, buf.length, serverAddr, 3333);

      System.out.println(String.format("Test: PACKETS_TO_SEND is %d",
PACKETS_TO_SEND));

      for (int i = 0; i < PACKETS_TO_SEND; i++) {

        System.arraycopy(ByteBuffer.allocate(4).putInt(i).array(), 0,
buf, 0, 4);

        /* Send packet */
        socket.send(packet);

        Thread.sleep(10, 0); // 10 milliseconds
      }

      /* All messages sent, close the socket */
      socket.close();

    } catch (Exception e) {
      e.printStackTrace();
    }
    System.out.println(String.format("Done!"));
  }
}
```

APÊNDICE C: CÓDIGO DA THREAD DA APLICAÇÃO SERVIDOR

```
package com.example.GindriServer;

import java.io.FileOutputStream;
import java.io.PrintStream;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketTimeoutException;
import java.util.BitSet;
import android.os.Environment;
import android.util.Log;

public class Server implements Runnable{

  /* Emulator's IP */
  public static final String SERVERIP = "10.0.2.15";

  /* Server's port*/
  public static final int SERVERPORT = 3333;

  /* Total expected packets*/
  public static final int TOTAL_PACKETS = 10000;

  public void run(){
    try{
      Log.d("UDP", "[S]: I, Server, started");

      /* Search server by IP address */
      InetAddress serverAddr = InetAddress.getByName(SERVERIP);
      Log.d("UDP",
        String.format("[S]: Got serverAddr %s. Connecting to it...",
          serverAddr.toString()));

      DatagramSocket socket = new DatagramSocket(SERVERPORT,
serverAddr);

      /* Define maximum length of UDP packets */
      byte[] buf = new byte[1472];

      /* Prepare the UDP packet for received data */
      DatagramPacket packet = new DatagramPacket(buf, buf.length);

      do {
```

```
        /* Counter for received packets (first one already counted) */
        int receivedPacketsCounter = 1;

        /* Create a BitSet for received packets, indexed by packet
sequence number */
        BitSet receivedPackets = new BitSet(TOTAL_PACKETS);

        /* Array for storing packer data, in order to get the packet
sequence number */
        byte[] seqBA;

        /* Time references */
        long ini_time = 0, fin_time;

        /* Initialize the socket timeout to be disabled */
        socket.setSoTimeout(0);

        Log.d("UDP", "[S]: Entering loop to receive packets...");

        try {

          /* Receive the first packet */
          socket.receive(packet);

          /* Set a timeout for socket */
          socket.setSoTimeout(10000);

          /* Get the packet data, to analyze first 4 bytes */
          seqBA = packet.getData();

          /* The sequence number is got by manipulating the first 4
bytes to an integer */
          receivedPackets.set( ((seqBA[0] & 0xff) << 24) | ((seqBA[1]
& 0xff) << 16) | ((seqBA[2] & 0xff) << 8) | (seqBA[3] & 0xff) );

          /* Get an initial time reference */
          ini_time = System.nanoTime();

          do {
            /* Receive next packets */
            socket.receive(packet);

            /* Increment counter for received packets */
            receivedPacketsCounter++;

            /* Get the packet data, to analyze first 4 bytes */
            seqBA = packet.getData();

            /* The sequence number is got by manipulating the first 4
bytes to an integer */
            receivedPackets.set( ((seqBA[0] & 0xff) << 24) |
((seqBA[1] & 0xff) << 16) | ((seqBA[2] & 0xff) << 8) | (seqBA[3] &
0xff) );

          } while (true);
        } catch (SocketTimeoutException e) {

          /* Get final time reference and */
```

```
          fin_time = System.nanoTime();

          Log.d("UDP", String.format(
              "[S]: Total elapsed time (since 1st packet): %d ns",
              fin_time - ini_time
          ));

          /* Print reports to Android (for behavior tests) */
          Log.d("UDP", String.format(
              "[S]: Total received packets: %d - %f%%",
              receivedPackets.cardinality(),
              100*(double)receivedPackets.cardinality()/
(double)TOTAL_PACKETS
          ));

          /* Print reports to Android (for performance test) */
          Log.d("UDP", String.format(
              "[S]: Total received packets: %d - %f%%",
              receivedPacketsCounter,
              100*(double)receivedPacketsCounter/
(double)TOTAL_PACKETS
          ));

/* output to file */
          FileOutputStream out;
          PrintStream prt;
          try {
            /* Check if we are able to write on device external
storage */
            String state = Environment.getExternalStorageState();
            if (Environment.MEDIA_MOUNTED.equals(state)) {
              out = new FileOutputStream("/sdcard/server_out.txt");
              prt = new PrintStream(out);

              int receivedBuffer = 0;

              for (int i = 0; i < TOTAL_PACKETS; i++) {
                if (receivedPackets.get(i)) receivedBuffer++;

                if ((i+1) % 50 == 0){
                  prt.println(receivedBuffer);
                  receivedBuffer = 0;
                }
              }
              prt.close();
            } else {
              Log.d("UDP", "[S]: not mounted");
            }
          } catch(Exception file_error) {
            Log.e("UDP", "[S]: Error", file_error);
          }
/* end of output to file */
        }
      } while (true);
    } catch (Exception e) {
      Log.e("UDP", "[S]: Error", e);
    }
  }
```

```
}
```

APÊNDICE D: MANIFEST DA APLICAÇÃO SERVIDOR

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.GindriServer"
      android:versionCode="1"
      android:versionName="1.0">


    <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name="GindriServer"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
    <uses-permission
android:name="android.permission.INTERNET"></uses-permission>
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"></uses-
permission>

</manifest>
```