UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIEGO DEMARCO DE GRANDI

EXPERIMENTOS DE INJEÇÃO DE FALHAS DE COMUNICAÇÃO UDP EM AMBIENTES ANDROID

Trabalho de Graduação

Prof. Dr. Taisy S. Weber
Orientador

Porto Alegre, dezembro de 2012

Agradecimentos

Gostaria de agradecer a algumas pessoas que foram muito importantes para mim durante o meu período de graduação.

Martina, em primeiro lugar, que me convenceu que eu deveria voltar para a UFRGS e terminar a minha graduação. Além de apoio incondicional e palavras de incentivo.

A Taisy Weber, orientadora, pela paciência e dedicação durante a produção dessa dissertação.

A minha família, por todos estes anos de ensino e carinho.

Sumário

Lista de Abreviaturas

OHA – Open Handset Alliance

IDE – Integrated Development Environment

SDK – Software Development Kit

VM – Virtual Machine

ADT – Android Development Tools

AVD Manager – Android Virtual Device Manager

J2SE – Java 2 Standard Edition

JDK – Java Development Kit

APK – Android Package File

UDP – User Datagram Protocol

Lista de Figuras

Lista de Tabelas

Resumo

O Android é o sistema operacional de maior crescimento do mercado de dispositivos móveis, sendo eles telefones celulares e tablets. Este crescimento também aumenta a quantidade de softwares desenvolvidos para esta plataforma, que por vez necessita de maiores garantias que os aplicativos funcionem em ambientes sujeitos a falhas de comunicação. Este trabalho utiliza o aplicativo Evernote instalado em um dispositivo Android virtual como base de testes, injeta falhas na comunicação e analisa como a aplicação responde a erros de comunicação. O trabalho mostra que testes sob falhas é uma estratégia viável para a avaliação do comportamento de aplicativos móveis em ambientes hostis.

Palavras-chave: Android, injeção de falhas, comunicação.

Abstract

Android is the market's most growing mobile operational system, including mobile telephones and tablets. This growth also enhances the amount of software developed to this platform. Sometimes the users need guarantees that these programs operate appropriately in environments that are susceptible to communication faults. This work uses Evernote installed on an Android virtual device as the test case, injects faults while communication and analyzes how the program answers to these communication faults. The work shows that running tests emulating faulty conditions is a viable strategy to evaluate the behavior of mobile applications in hostile environments.

Keywords: Android, failure injection, network.

1. INTRODUÇÃO

Desde o advento dos smartphones, o acesso a informações na Internet vem migrando dos computadores de mesa ou notebooks para esta nova possibilidade mais portável, ágil e conveniente. Não necessitar levar uma mochila para conseguir carregar seu notebook até algum local onde haja *wi-fi* para então conseguir verificar sua caixa de entrada de e-mails tornou o telefone móvel uma ferramenta indispensável no meio corporativo.

Devido a esta crescente necessidade, sistemas operacionais foram desenvolvidos para alcançar o mercado. O Android, sistema inicialmente desenvolvido pela Google [1] e após pela Open Handset Alliance [2] como open-source, é a plataforma mais utilizada pelo mundo, abrangendo celulares de várias operadoras e pela facilidade de desenvolver aplicações e publicá-las no Anroid Marketplace, plataforma de venda, download e disponibilização de aplicações para Android.

Aplicações como e-mais, visualizadores de documentos e aplicativos de escritório são extremamente comuns nos dispositivos móveis, e usam de conexão com internet para acessar os mesmos. Porém, tais aplicações precisam estar preparadas para funcionar e comunicar mesmo quando houver problemas de comunicação.

1.1. Motivação

Buscando mostrar que um sistema é resistente a falhas de comunicação, é possível utilizar mecanismos de tolerância a falhas e testá-las utilizando técnicas que produzam as mesmas. Hsueh [3] cita que, mesmo havendo resultados relevantes utilizando a técnica de medição de carga de trabalho dos sistemas operacionais para prever falhas, a técnica de injeção de falhas é tida como a mais adequada para reproduzir tais cenários e testar o comportamento da aplicação.

Este trabalho apresenta a seguinte proposta: injetar falhas na comunicação no ambiente de sistema do Android mais recente do mercado, o 4.1 (Jelly Bean), usando o método de mutacionar a classe DatagramSocket, que centraliza as rotinas de *send* e *receive* da interface de sockets da comunicação de dados do Android. Para injetar falhas em um programa durante sua execução, será construído o Android com recursos para a injeção de falhas.

Buscando validar e analisar o tipo de falha implementada, uma aplicação cliente-servidor será desenvolvida e seu comportamento será monitorado, provendo dados a serem analisados sobre a injeção de falha na comunicação.

Por fim, uma aplicação de mercado foi escolhida para ter o seu comportamento analisado enquanto trafega dados com falhas de comunicação. O aplicativo escolhido foi o Evernote [4] , utilizando o subaplicativo de mesmo nome, que consiste num serviço de nuvem de anotações em texto.

Vários trabalhos do grupo de tolerância a falhas trataram de injeção de falhas em ambientes móveis, entre eles podem ser citados os trabalhos de Acker [5] e Gindri [6]. O primeiro trabalho citado usa o injetor de falhas Firmament [7, 8] para injetar falhas de comunicação a partir do kernel Linux que compõem o ambiente do Android. Firmament intercepta mensagem na pilha de protocolo do kernel usando recursos de filtros de pacotes. O segundo trabalho atua na máquina virtual. Alterações na máquina virtual do ambiente Android implementam o injetor de falhas de comunicação. Este trabalho segue a mesma proposta de atuar na máquina virtual Android, porém alterando somente uma classe, a *DatagramSocket*, que é a responsável pela comunicação UDP.

Devido a baixa intrusividade do mecanismo de injeção de falhas, que é mais restrita e confinada do que alterar a máquina virtual, é possível injetar a falha somente nas aplicações que utilizarem o socket UDP; se outros serviços do sistema estiverem em operação, eles não serão afetados. O teste fica restrito a aplicação alvo. Esse confinamento é importante para o injetor não prejudicar outras funcionalidades do sistema e assim alterar tanto o comportamento que os resultados dos experimentos de teste perdem o valor.

Ainda, este trabalho utiliza uma aplicação de mercado para testar a funcionalidade das mesmas e analisar o comportamento da aplicação sob falhas.

A melhor estratégia para construir injetores de falhas de comunicação para ambientes como o Android ainda é um problema em aberto. Os trabalhos nesta área exploram várias estratégias diferentes, tal como criar injetores de falhas em hardware ou software ou ainda utilizar de radiação no equipamento afim de produzir uma falha [3]. Cabe a equipe de testes escolher a melhor estratégia para os seus objetivos de teste.

1.2. Proposta

Este trabalho está dividido em dois grandes tópicos: modelagem e implementação; sendo este último com subdivisões também importantes: programa de teste da comunicação e avaliação de uma aplicação existente no mercado.

A injeção de falhas na comunicação será implementada utilizando a técnica de alteração de código em tempo de compilação [3]. Esta técnica modifica trechos de código de classes e rotinas usadas pelo sistema operacional e injeta perturbações no fluxo da mesma.

Em um primeiro momento, a falha será injetada no arquivo de *DatagramSocket* do Android, que é a interface de comunicação de dados do sistema operacional. A falha será

injetada nos métodos *send()* e *receive()*, assim garantindo que toda e qualquer comunicação poderá ser afetada pela falha.

Após será gerado um ambiente do Android com a injeção de falha por alteração de código. Para este ambiente, duas aplicações de teste serão criadas: uma aplicação cliente que transmitirá pacotes de dados via Android com a injeção de falhas, e um servidor que estará na máquina de desenvolvimento esperando a conexão do cliente. Este verifica quantos pacotes foram recebidos desde o momento da primeira transmissão até um timeout de 5 segundos. Com esta transmissão cliente servidor, poderá ser verificado que o injetor de falhas opera conforme desejado.

Por último e ainda utilizando a o ambiente Android modificado com a injeção de falha, a aplicação Evernote será instalada. Então serão feitas transmissões de texto pela mesma aplicação e verificaremos o texto recebido na versão web da aplicação para observar o comportamento da aplicação sob falhas.

2. MODELAGEM

A comunicação móvel vigente no mercado visa trafegar a maior quantidade de dados possível evitando tráfegos tidos como pouco eficientes, tal como confirmação de recepção e monitoramento de dispositivos conectados.

Além disso, ambientes móveis são mais sujeitos a influências climáticas que podem causar falhas na comunicação. Por último, como sistemas móveis têm grande apelo com o consumidor, as falhas devem ser corrigidas com o mínimo de percepção do usuário [9].

2.1. Modelo de injeção de falhas

O modelo de injeção de falhas utilizado neste trabalho foi o de implementação de falhas no software, buscando reproduzir falhas de comunicação, omitindo mensagens transmitidas pela rede [3]. A injeção de falhas visa avaliar estratégias de tolerância a falhas [10, 3]. Através da introdução controlada de falhas, é possível determinar se o software consegue responder de maneira coesa e robusta a eventos tidos como falhas.

Injeção de falhas é tanto uma ferramenta de teste e determinação de métricas (como cobertura de falhas e queda de desempenho sob falhas), como também para benchmark de dependabilidade [11] comparando diferentes implementações de um mesmo sistema sujeitas às mesmas falhas.

A injeção de falhas foi inserida na classe de Socket do Android de acordo com a Arquitetura de Iyer [12], onde cada entidade da arquitetura tem funções definidas e transições possíveis entre as mesmas. A figura 1 abaixo apresenta a arquitetura proposta por Iyer.

- **Alvo:** A aplicação que receberá a carga com erros e será analisado o seu comportamento, também chamado sistema sob teste ou SUT (system under test).
- **Controlador:** aplicação que roda no dispositivo alvo ou algum outro e controla o experimento.
- **Injetor:** implementa a falha no alvo, podendo ser falha de hardware, software ou radiação.
- **Monitor:** acompanha a execução (tanto normal quanto anormal) e ativa o coletor de dados quando necessário.
- **Coletor de dados:** coleta os dados quando necessário durante o tempo de execução.

- **Analisador:** fora do tempo de execução, ele analisa os dados coletados durante o experimento.

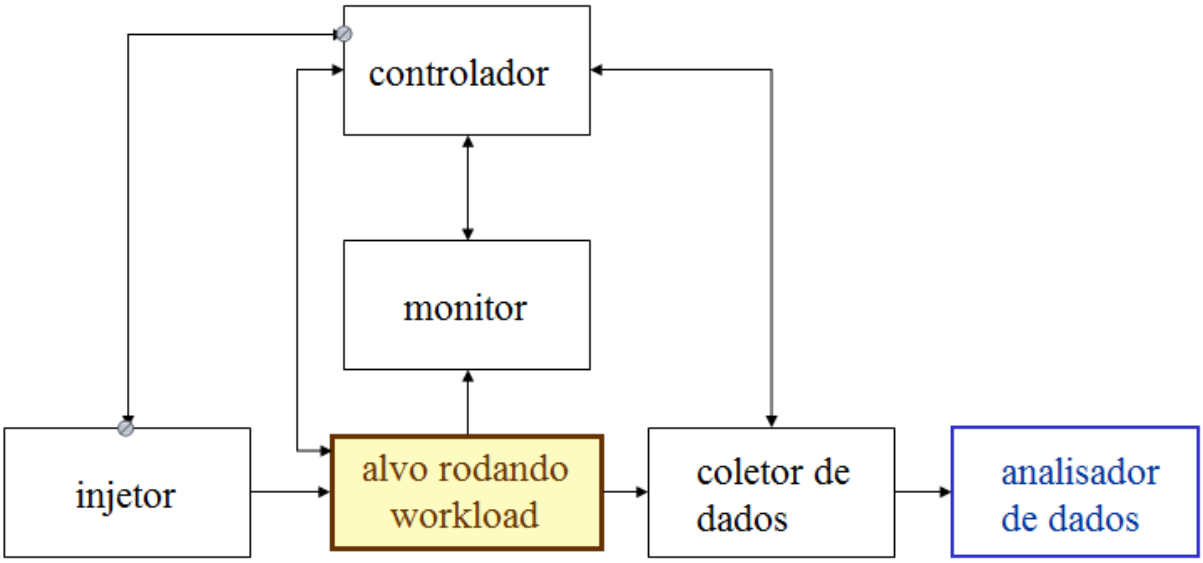

Figura 1: A arquitetura proposta por Iyer.

De acordo com a arquitetura sugerida acima, as seguintes entidades relativas a este trabalho foram possíveis de serem identificadas:

- **Alvo:** A aplicação Evernote;
- **Controlador:** a máquina de desenvolvimento que monitora a aplicação web do Evernote;
- **Injetor:** O injetor é o método *ableToCommunicate* da classe *DatagramSocket*, que avalia se o datagrama que será enviado via socket terá falhas ou não.
- **Monitor e Coletor de Dados:** Estas duas entidades estão representadas por outras já citadas anteriormente. Ambos também são verificados pela aplicação web do Evernote.
- **Analisador:** A aplicação WinMerge captura o texto recebido pela aplicação web do Evernote recebida e compara com o texto padrão.

Este trabalho focou na entidade Injetor, entidade que proporcionava maior facilidade em inserção de código, e o alvo é uma aplicação consolidada no mercado e com comportamento constante e esperado. As entidades menores (monitor e coletor) não tiveram aplicações que as representam implementadas, tendo em vista que a aplicação Evernote é um

software fechado e sem possibilidade de modificar o seu código buscando inserir as entidades faltantes.

O injetor foi modelado de acordo com o princípio de injeção de falha em software em tempo de execução simulando falhas de rede [12]. Os injetores por software são preferidos em relação aos em hardware pois têm um custo menor, menor complexidade, maior adaptabilidade e menos interferência no sistema. A sua desvantagem é que nem todas as falhas são possíveis de serem injetadas via software, pois existem recursos que não são passíveis de terem o seu comportamento alterado via software, como por exemplo, a modulação dos dados enviados pelo sistema de antena.

Como este trabalho tem a possibilidade de injetar a falha a partir do software, mais especificamente na classe *DatagramSocket* que é a responsável pela comunicação via protocolo UDP no Android, a injeção produz erros que alteram o estado do sistema alvo, simulando falhas de comunicação. O fluxo normal na comunicação de dados, exemplificado na figura 2, é:

- Quando enviando dados (método *send*): A aplicação solicita envio do datagrama. Após a classe de comunicação (DatagramSocket) monta o datagrama, envia o mesmo e devolve o controle para a aplicação.

- Quando recebendo dados (método *receive*): A aplicação escuta um socket previamente configurado. A classe de comunicação (DatagramSocket) espera receber algum dado via socket. Quando há dados a serem recebidos, o DatagramSocket recebe os dados, repassa para a aplicação e devolve o controle para a mesma.

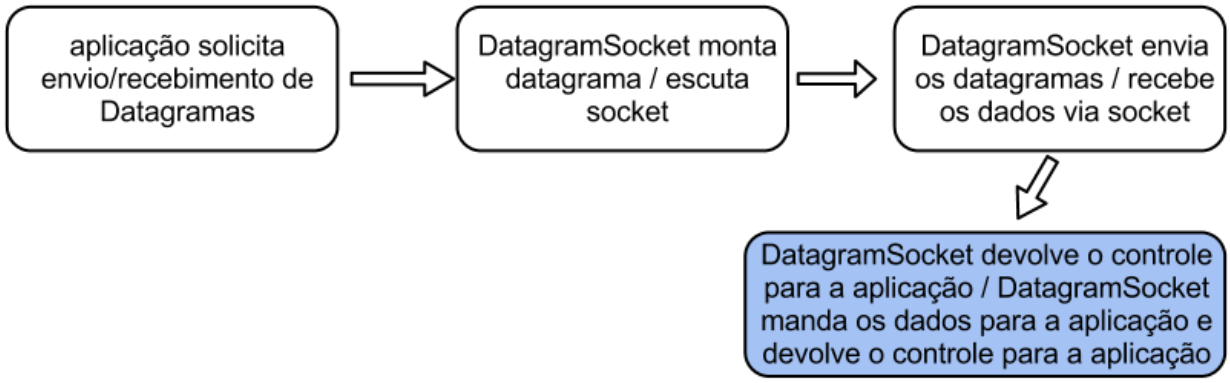

Figura 2: Fluxo normal da comunicação de dados.

O processo de comunicação do Android com injeção de falhas, exemplificado na figura 3, é diferente:

- Quando enviando dados (método *send*): A aplicação solicita envio do datagrama. Então é testada a chance de injeção de falhas. Caso negativo (1), a classe de comunicação (DatagramSocket) monta o datagrama, envia o mesmo e devolve o controle para a aplicação. Caso Positivo (2), o controle é devolvido para a aplicação sem comunicar dados.

- Quando recebendo dados (método *receive*): A aplicação escuta um socket previamente configurado. A classe de comunicação (DatagramSocket) espera receber algum dado via socket. Quando há dados a serem recebidos, é testada a chance de injeção de falhas. Caso negativo (1), o DatagramSocket recebe os dados, repassa para a aplicação e devolve o controle para a mesma. Caso positivo (2), o controle é repassado para a aplicação devolvendo dados nulos como recebidos.

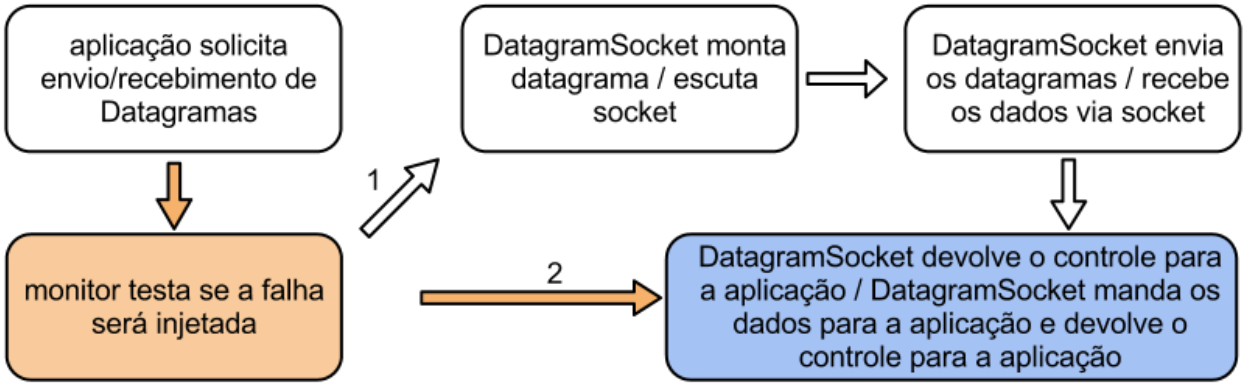

Figura 3: Fluxo de comunicação com injeção de falhas.

2.2. Modelo de avaliação de comunicação

A aplicação Evernote baseia-se num conjunto de aplicações de produtividade para uso corporativo, e sincronizado em nuvem para poder acessar em qualquer dispositivo. A aplicação utilizada captura o texto inserido no dispositivo móvel e envia para a nuvem, disponibilizando no repositório. Para preencher o texto na aplicação, será colado um texto dentro do mobile e sincronizado com a nuvem, enviando o input para o servidor.

Utilizando o Android compilado com a injeção de falha na comunicação, o esperado que após enviar o texto via o emulador Android, a aplicação Evernote será aberta fora do emulador e o texto recebido na nuvem será comparado ao texto enviado pela aplicação do mobile. A diferença entre o servidor e a aplicação mobile será analisada para saber se houve falhas ou não.

O grupo de pesquisa de tolerância a falhas em Android com o qual este trabalho é integrante utilizou abordagens distintas enquanto injetando suas falhas O trabalho de Alexandre Gindri [6] também utilizou modificação de código fonte para criar o injetor, porém utilizou modelos de injeção de falhas de Bernoulli e modelo de perdas em rajadas de Gilber-Elliot. Estes dois modelos utilizam máquinas de estados determinísticas para injetar a falha. Eduardo Acker [5] utilizou uma ferramenta desenvolvida pela UFRGS, o Firmament. Esta ferramenta avalia sistemas distribuídos e protocolos de comunicação, além de inserir o conceito de faultlet, sendo este uma aplicação que emula o comportamento das falhas atuando também sobre registradores de uso geral que estão ligados ao fluxo da ferramenta.

Uma das diferenças entre este trabalho e anteriores do grupo de tolerância a falhas é que neste foi escolhida uma aplicação de mercado como alvo de avaliação afim de avaliar a robustez da mesma, além de utilizar um método probabilístico para injeção da falha, diferenciando-se dos outros trabalhos do grupo. Qualquer aplicação da plataforma Android que comunique via protocolo UDP poderia ser utilizada como alvo.

3. AMBIENTE

Visando facilitar o desenvolvimento e a participação da comunidade na concepção de novas versões e aplicativos para o Android , a Google disponibiliza kits de desenvolvimento de software (SDK, Software Development Kit) com suporte para os principais sistemas operacionais do mercado (Windows, MacOs e Linux).

O SDK utilizado neste trabalho foi o ADT (Android development Tools): um plug-in da IDE Eclipse [13] que facilita a criação e o desenvolvimento de novas aplicações do Android.

Para gerenciar as diferentes versões do Android que serão construídas no decorrer do trabalho foi utilizado o plug-in da IDE Eclipse AVD Manager (Android Virtual Device) [14]. Este software concentra as configurações das diferentes versões do Android disponíveis na máquina de desenvolvimento. É possível baixar versões de mercado do Android, criar dispositivos virtuais emulados com diferentes características de tamanho de tela, processamento e espaço de armazenamento e ainda criar um dispositivo virtual utilizando um ambiente customizado, que é o objetivo deste trabalho.

Este trabalho criou dois dispositivos virtuais Android:

- Um com a versão 4.1 do Android padrão.
- Um com a versão 4.1 do Android modificado com o Injetor de falhas.

3.1. Ambiente de Desenvolvimento

A máquina onde foi desenvolvido este trabalho tem as seguintes características de hardware:

- Intel i5 2500k 3.3Ghz (4 cores, 1 thread por core e cada core funcionando a 4Ghz via overclock)
- 8Gb DDR3 1600Mhz de Memória
- Placa de Vídeo GeForce STi GTX560 ti

O software instalado e utilizado para o desenvolvimento deste trabalho foi o seguinte:

- Windows 7 Professional 64-bits, com todos as atualizações de alta prioridade instalados até o dia 08/10/2012. Instalação não limpa.

Para o desenvolvimento da injeção de falhas, foram utilizados os seguintes softwares:

- Eclipse Indigo Service Release 2, build 2012216 - 1857.
- Android AVD, obtido pela Eclipse Marketplace.
- Android SDK, também obtido via Eclipse Marketplace

Para construir o Android personalizado, é necessário utilizar uma máquina com UNIX como sistema operacional. Então, foi instalado uma máquina virtual com o sistema operacional Linux Ubuntu instalado:

- Oracle VM VirtualBox 4.2.0-r80737
- Ubuntu 10.10 64-bits (única versão do ubuntu em que o Android tem testes periódicos para as suas versões de Android acima do 3.1) [15].

3.2. Instalação do ambiente de desenvolvimento e construção do Android 4.1

A partir de uma instalação clean do Ubuntu 10, é necessário instalar pacotes de bibliotecas utilizadas pelo Android. Os passos enumerados a seguir são necessários para construir o Android.

3.2.1. Instalação do Java requerido pelo Android

É descrito no site do Android [1] que as versões até 2.3 do sistema operacional são específicas da plataforma Java 5, enquanto as versões posteriores do Android usam a versão 6. Para baixar esta máquina virtual, foi utilizado o Java 6.36 [16]. Apesar de não ser a versão mais atual do Java, é a versão tida como “estável” pelo Android.

Buscando instalar o Java 6.36, primeiro foi feito o download do seu instalador na página oficial do Java Oracle. Após o download, foram executados os seguintes comandos no terminal do Linux para dar permissão de execução ao instalador Java e então propriamente executar a instalação da plataforma Java versão 6.36:

```
$ chmod 777 jdk-6u36-Linux-x64.bin
$ ./jdk-6u36-Linux-x64.bin
```

Após rodar este comando, foi criado um diretório chamado `jdk1.6.0_36` com os arquivos do Java descompactados. Para movê-los, foi necessário executar o comando *mover* (*mv)* com permissão de supervisor (*sudo*):

```
$ sudo mv jdk1.6.0_32 /usr/lib/jvm/
```

Neste ponto os arquivos do Java foram movidos para um diretório padrão da *Máquina Virtual Java* no Linux. Agora falta adicionar o caminho da nova instalação do Java nos registros de sistema, para então utilizar a JVM.

```
$ sudo update-alternatives --install "/usr/bin/Java" "Java"
"/usr/lib/jvm/jdk1.6.0_32/bin/Java" 1

$ sudo update-alternatives --install "/usr/bin/Javac" "Javac"
"/usr/lib/jvm/jdk1.6.0_32/bin/Javac" 1
```

Caso uma versão do Java tenha sido instalada antes de começar os passos mencionados por este trabalho, é necessário atualizar as configurações do Linux para o novo Java:

```
$ sudo update-alternatives --config Java
$ sudo update-alternatives --config Javac
```

Enfim, o Java está instalado na máquina virtual [17].

3.2.2. Instalação das bibliotecas Linux requeridas pelo Android

Com o Java já instalado, é necessário instalar uma série de bibliotecas que serão necessárias para a obtenção, instalação e construção do Android no ambiente de desenvolvimento. Estas bibliotecas envolvem desde aplicações de gerência de repositórios, tal como *git-core*, até bibliotecas de linguagens que o Android usa enquanto constrói o seu sistema, como o *python-markdown*.

Para obter estas bibliotecas, é necessário rodar o seguinte comando dentro do terminal Linux:

```
$ 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 \
    libgl1-mesa-dev g++-multilib mingw32 tofrodos python-markdown \
    libxml2-utils xsltproc
```

Por utilizar o Ubuntu 10.10, é necessário uma lib extra:

```
$ sudo ln -s /usr/lib32/mesa/libGL.so.1 /usr/lib32/mesa/libGL.so
```

3.3. Obtenção dos fontes do Android

Depois de obter todas as bibliotecas necessárias, o ambiente está apto a receber, construir e rodar o Android. Então passamos para a próxima etapa: configurar um repositório para obter o source code do Android [17].

Primeiro, criamos um diretório que servirá como repositório dos arquivos do Android:

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

Após é feito o download do *Script Repo* [18]. Este script foi criado pela Google para facilitar a sincronização e obteção dos arquivos do seu projeto entre os seus desenvolvedores. Após o término do download, é necessário dar permissão de execução ao *Repo Script.*

```
$ curl https://dl-ssl.google.com/dl/googlesource/git-repo/repo >
   ~/bin/repo
$ chmod a+x ~/bin/repo
```

Com isto, a máquina tem os softwares necessários para instalar o repositório Android. Agora é necessário buscar os fontes do Android e guardá-los em algum local.

```
$ mkdir Android_source
$ cd Android_source
```

O comando acima cria um diretório denominado "*Android_source*". Neste diretório, o script repo configurará um repostório *Git* [19] na máquina. É possível escolher uma única

versão do Android para efetuar o download, e foi esta a opção utilizada neste trabalho. A versão de Android escolhida foi o 4.1, também conhecido no mercado como *Jelly Bean*.

```
$ repo init -u https://Android.googlesource.com/platform/manifest -b
    Android-4.0.1_r1
```

O comando repo init configura arquivos de repositório necessários para a obtenção dos fontes. Porém ainda não foi efetuado o download. Buscando baixar os arquivos, é necessário sincronizar o repositório local com o repositório remoto, vulgo repositório Android da Google [17].

```
$ repo sync
```

O comando *sync* verificará que o repositório local está vazio, logo ele obterá cópias dos arquivos do repositório remoto e salvará no local. Logo os dois repositórios estarão sincronizados e iguais.

3.4. Construção do sistema Android

Após aproximadamente 3.2Gb de download, os fontes do Android 4.1 estarão no diretório *Android_sources*, e pronto para construir uma imagem do sistema Android. Se algum arquivo for alterado, ele irá para a imagem. Em primeiro momento, uma imagem será criada com o Android puro, sem alterações, e posteriormente a injeção e falhas será inserida nos fontes e então uma segunda imagem também será construída. Para construir uma imagem do Android, é necessário efetuar os seguintes comandos:

Para inicializar o ambiente de construção:

```
$ .build/envsetup.sh
```

Para organizar qual a versão que será construída:

```
$ lunch full-user
```

Para construir a imagem do Android:

```
$ make -j4
```

O parâmetro do comando *make* é *t* número de threads alocadas para a construção. Comumente é usado o valor 1 ou 2 para cada núcleo da máquina de desenvolvimento. Como a

máquina tem 4 cores, porém 2 alocados para a máquina virtual, foi utilizado o comando *-j4*, alocando 4 threads para a construção do sistema.

3.5. Injeção das falhas

Existem 2 tipos principais de injeção de falhas: a injeção baseada em simulação, comumente usada durante o desenvolvimento das aplicações e a injeção de falhas baseada em protótipos, comumente utilizada para testar aplicações em sua versão funcional ou final. [3]. Neste trabalho utilizaremos a injeção baseada em protótipos, injetando a falha em software. A injeção de falhas baseada em protótipo pressupõem a existência de um sistema alvo pronto ou parcialmente pronto. Ela é útil para a avaliação de sistemas de terceiros e não necessita do código fonte da aplicação alvo.

Durante o processo de construção dos dois Android citados no tópico acima, foi ressaltado que num dado ponto do mesmo o arquivo *DatagramSocket.java* com a injeção de falhas é inserido junto dos demais arquivos para então construir o sistema com o injetor. O *DatagramSocket* com a possibilidade de injeção de falha deve ser colocado dentro do diretório *sources/java/net/*. Tendo em vista que toda comunicação via sockets do Android passa por esta classe é necessário alterar o código desta classe somente.

3.6. Emulador Android

Após construir o Android, o path do emulador é inserido automaticamente no path do sistema construído. Logo basta chamar o comando de inicialização do emulador que um dispositivo móvel virtual rodando o sistema construído será executado [18].

```
$ emulator
```

3.7. Instalar a aplicação Evernote no emulador Android

Com o emulador rodando, é necessário instalar a aplicação Evernote dentro da mesma. Tendo em vista que este sistema vem sem algumas funcionalidades do Android, como por exemplo o Android Market, aplicativo onde procuramos novos aplicativos para instalar no dispositivo móvel, é necessário instalar manualmente a aplicação Evernote.

A aplicação Evernote é a aplicação sob teste também chamada alvo do sistema de injeção de falhas. Essa aplicação sofrerá o efeito da injeção de falhas de comunicação e seu comportamento será observado durante o teste.

Para instalá-la, em primeiro momento é necessário obter a aplicação compactada no formato Android Package File, .apk. A mesma foi baixada do site Android Drawer [20]. A versão da aplicação é a 4.2, vide figura 4.

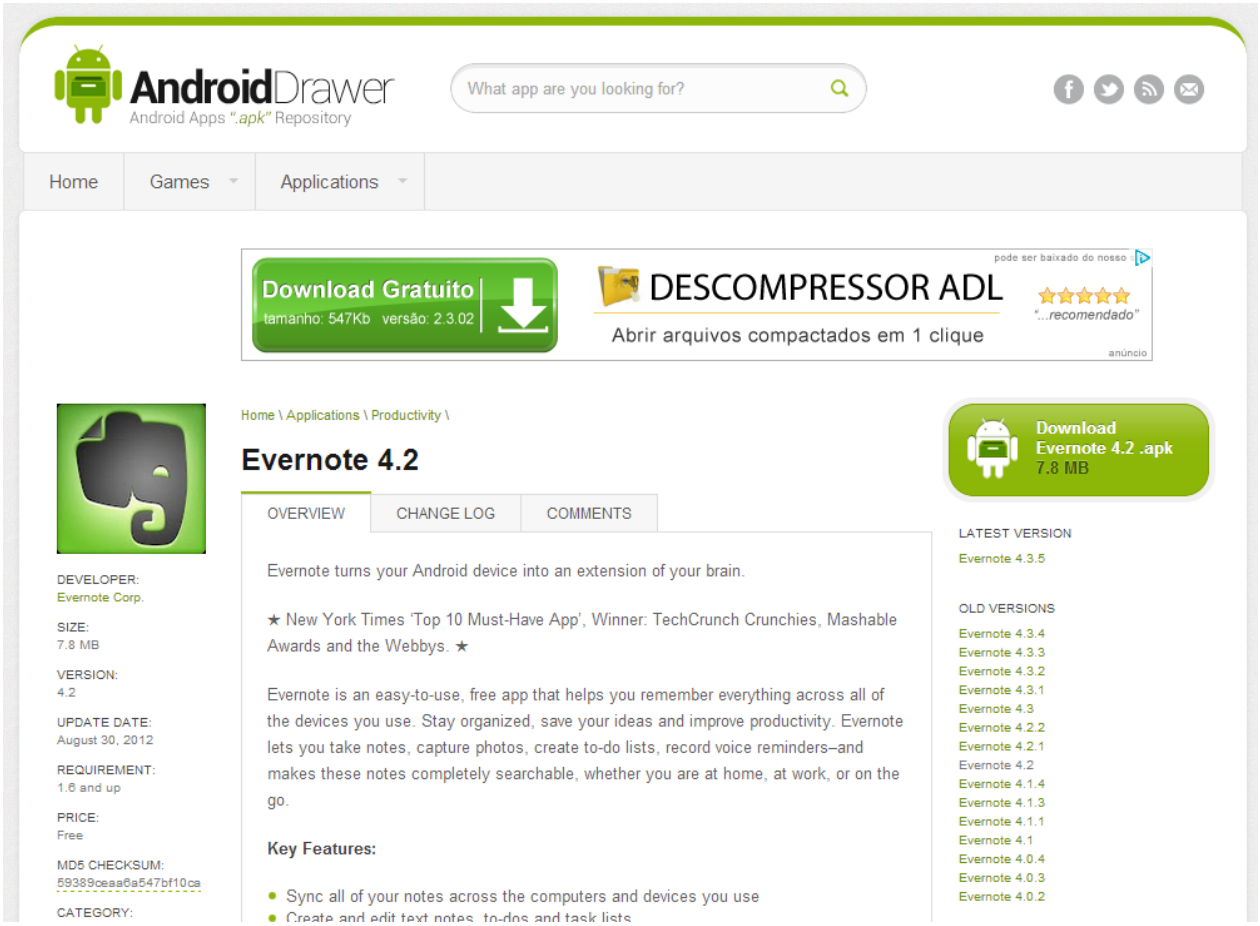

Figura 4: Efetuando download da aplicação Evernote

Com o Android Package da aplicação, agora é necessário instalá-lo no emulador.

Acessando o *promt* de comando, é necessário navegar até o diretório onde está o arquivo com a extensão *.apk*. Então deve-se rodar o seguinte comando:

```
$ adb install -r app.apk
```

No exemplo acima, o aplicativo comprimido está representado como *app.apk*.

Após o término da instalação (poucos segundos), o emulador estará com a aplicação aberta e pronta para uso, exemplificado na figura 5 abaixo:

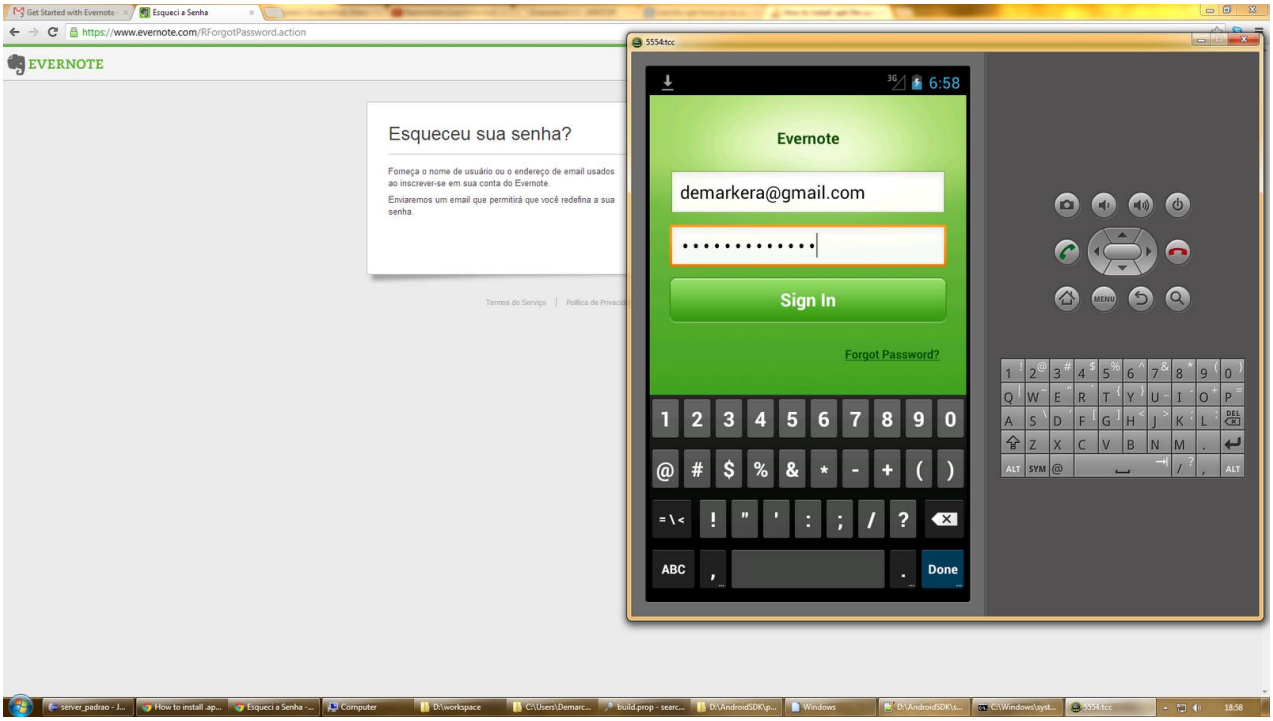

Figura 5: Aplicação Evernote instalada no emulador.

4. IMPLEMENTAÇÃO DO INJETOR DE FALHAS NAS CLASSES DO ANDROID

O Android tem concentrado em uma única classe o envio e recebimento dos datagramas via sockets UDP: a classe DatagramSocket. Esta classe, além dos métodos *send* e *receive*, concentra rotinas de estabelecimento de conexões via sockets tal como também o encerramento das mesmas e propriedades dos sockets. Os sockets abertos por esta classe trafegam dados somente pelo protocolo UDP [1]. Existem outras classes responsáveis pelo tráfego de dados via protocolo TCP.

Analisando o código da classe, é possível identificar que os métodos *send* e *receive* estão bem encapsulados e não comprometem a confiabilidade do código mesmo sendo alterados; somente a comunicação será afetada.

A injeção da falha será implementada via alteração de código [21] buscando simular falhas de rede [3], alterando o fluxo dos métodos para somente prosseguir se o cálculo de chance de falha for satisfeito.

A carga de falhas escolhida é fixa de 2% de datagramas trafegados via socket. Isto quer dizer que, em média, 2% de todos datagramas não alcançarão o destino esperado quando tentando enviar datagramas ou que 2% dos datagramas que seriam recebidos quando recebendo dados via comunicação não serão processados nem repassados ao processo requisitante. Dentro do socket, a taxa foi implementada de forma fixa, impossibilitando a sua alteração sem recompilar o Android.

Uma taxa considerada aceitável para a falha nas comunicações é de 2% [22], tendo em vista que o encontrado no Brasil oscila entre 8% e 1%, dependendo das operadoras de telefonia móvel.

4.1. O método de cálculo de probabilidade ableToCommunicate

Utilizando o princípio de Distribuição Uniforme das probablidades [23] W, onde a probabilidade de a variável aleatória X ocorrer no intervalo infinitesimal [x*, x*+dx] é 1/(b-a) se x estiver entre a e b, e zero em caso contrário, um número aleatório generatedNumber que respeita esta distribuição, com média de 2% (0.02) com limite inferior de 0 e limite superior de 1. Quando o número aleatório for menor que 0.02 (2%), o método retornará falso, caso contrário, retornará verdadeiro. A figura 6 abaixo exemplifica o método:

```
private boolean ableToCommunicate() {
        final int minChance = 0;
        final int maxChance = 100;
        int generatedNumber = minChance
                + (int) (Math.random() * ((maxChance - minChance) + 1);
        final int prob = 2;
        if (generatedNumber > prob)
                return true;
        else {
                return false;
        }
}
```

Figura 6: O método *ableToCommunicate*.

Caso fosse necessário modificar a probabilidade de falha na comunicação, é necessário modificar apenas a variável *prob*, variando entre 0 e 100. Caso o valor da variável seja maior que 100, toda e qualquer comunicação será falha. Se o valor de prob seja 0 ou menor, nunca serão injetadas falhas na comunicação.

Outra abordagem de configuração de taxa de falhas seria escrever um arquivo de configuração, colocá-lo dentro do espaço de disco reservado ao Android no dispositivo móvel virtual e ler o valor do mesmo antes de cada novo experimento de teste.

4.2. Injeção da falha no método send

O método send é o responsável pelo envio de dados via sockets no Android. Porém antes de enviar os dados, ele verifica se ainda está aberto o socket de comunicação e configura o DatagramPacket com os dados entrados pelo programa do usuário, no caso deste trabalho, o Evernote [18].

Para alterar o mínimo possível de código e manter sua coerência, foi alterado somente uma linha do código original do método: a chamada *impl.send(pack)* foi envolvida por um condicional if, que recebe um booleano do método que calcula a probabilidade de injeção da falha. A figura 7 demonstra as alterações no método send da classe DatagramSocket.

```
public void send(DatagramPacket pack) throws IOException {
        checkOpen();
        ensureBound();
        InetAddress packAddr = pack.getAddress();
        if (address != null) { // The socket is connected
                if (packAddr != null) {
                        if (!address.equals(packAddr) || port != pack.getPort()) {
                                throw new IllegalArgumentException("Packet
                                address mismatch with connected address");
                        }
                } else {
                pack.setAddress(address);
                pack.setPort(port);
                }
        } else {
        // not connected so the target address is not allowed to be null
                if (packAddr == null) {
                        throw new NullPointerException("Destination address is
                        null");
                }
        }
        if(ableToCommunicate()){ impl.send(pack); }
}
```

Figura 7: O método *send*.

4.3. Injeção da falha no método receive

O método receive é o responsável pelo recebimento de dados via sockets no Android. Porém antes de enviar os dados, ele verifica se ainda está aberto o socket de comunicação. Caso ainda esteja aberto, ele escuta o socket esperando dados.

Buscando alterar o mínimo possível de código e manter sua coerência, foi alterada somente uma linha do código original do método: a chamada *impl.receive(pack)* foi envolvida por um condicional if, que recebe um booleano do método que calcula a probabilidade de injeção da falha. O método *receive* está exemplificado na figura 8 abaixo:

```
public synchronized void receive(DatagramPacket pack) throws
IOException {
        checkOpen();
        ensureBound();
        if (pack == null) {
                throw new NullPointerException();
        }
        if (pendingConnectException != null) {
                throw new SocketException("Pending connect failure",
                pendingConnectException);
        }
        pack.resetLengthForReceive();
        if(ableToCommunicate()){  impl.receive(pack);  }
}
```

Figura 8: O método *receive*.

4.4. Comentários sobre o capítulo

Este capítulo visou ilustrar como foi implementada a injeção de falhas a partir dos métodos responsáveis pela comunicação e de que maneira era calculado se a falha seria ativada durante a comunicação.

De acordo com a proposta que as falhas podem ocorrer em qualquer momento, tanto enviando quando recebendo dados, a injeção foi implementada também nos dois processos: envio e recebimento.

5. TESTES DO INJETOR DE FALHAS

Para testar se o injetor de falhas simula um comportamento esperado de falhas no Android, foi implementada uma aplicação cliente - servidor para testá-lo. Um fluxo de dados será gerado do cliente instalado no emulador Android para o servidor que estará rodando na máquina. O servidor foi implementado na linguagem Java.

O teste baseia-se num princípio simples: o cliente deve enviar uma quantidade fixa de datagramas via *socket* UDP para a aplicação servidor e esta contará quantos datagramas foram recebidos. A aplicação servidor esperará datagramas com um timeout de 10 segundos após o último datagrama recebido. Então será feita a contagem de datagramas e a obtenção da porcentagem de recebimento.

Esta aplicação de teste foi construída na IDE Eclipse de desenvolvimento como um projeto de uma aplicação Android tendo um dos ambientes previamente construídas (com injetor de falhas e sem injetor de falhas) como biblioteca de execução. Os testes foram verificados dentro da console da IDE.

5.1. Aplicação cliente de teste

A aplicação cliente tenta estabelecer uma conexão UDP com a aplicação servidor. Caso seja estabelecida, começa a transmissão de 1000 datagramas. Quando estes 1000 datagramas foram enviados, a aplicação encerra e para de transmitir.

Para o funcionamento correto desta aplicação como um Projeto Android dentro do emulador rodando na IDE Eclipse, foi necessário adicionar permissões de utilização de rede para a aplicação. Para configurar estas permissões, foi adicionado um comando dentro do *AndroidManifest.xml* [18] da aplicação de teste:

```
<uses-permission Android:name="Android.permission.INTERNET">
</uses-permission>
```

Este comando garante que a aplicação terá acesso a rede internet do emulador Android.

5.2. Aplicação Servidor de teste

A aplicação servidor foi desenvolvida na linguagem Java. Ela foi desenvolvida em 5 estágios, sendo um para cada estado da comunicação, sendo o anterior necessário para ocorrer o próximo:

1. Tenta criar um socket UDP com uma porta previamente configurada;
2. Espera alguém estabelecer a conexão;
3. Recebe o primeiro datagrama;
4. Recebe os demais datagramas;
5. Encerra comunicação ao expirar o timeout (10 s).

Ao término da comunicação, ela exibe as seguintes informações no console:

1. Tempo de comunicação em ns;
2. Quantidade de pacotes (datagramas) recebidos e taxa de recebimento em comparação ao esperado (1000 pacotes);

Após o timeout, a aplicação servidor espera um novo cliente conectar e recomeçar o processo.

5.3. Resultado dos testes

Tendo em vista que foram testados dois ambientes com características distintas em relação a comunicação UDP (um sem o injetor de falhas enquanto o outro tem o injetor), é esperado que hajam resultados com diferenças sensíveis. Nos próximos subcapítulos estão os resultados obtidos utilizando a aplicação cliente-servidor implementada previamente.

5.3.1 Socket sem Injeção de falhas

Testando a aplicação sem o injetor de falhas, o resultado foi igual ao esperado: todos os datagramas enviados pela aplicação cliente foram recebidos pelo servidor, vide tabela 1 e figura 9 abaixo:

Teste:	Tempo de comunicação	Porcentual de recebimento
1	3,40 s	100%
2	5,11 s	100%
3	4,12 s	100%

Tabela 1: Testes da comunicação sem injeção de falhas.

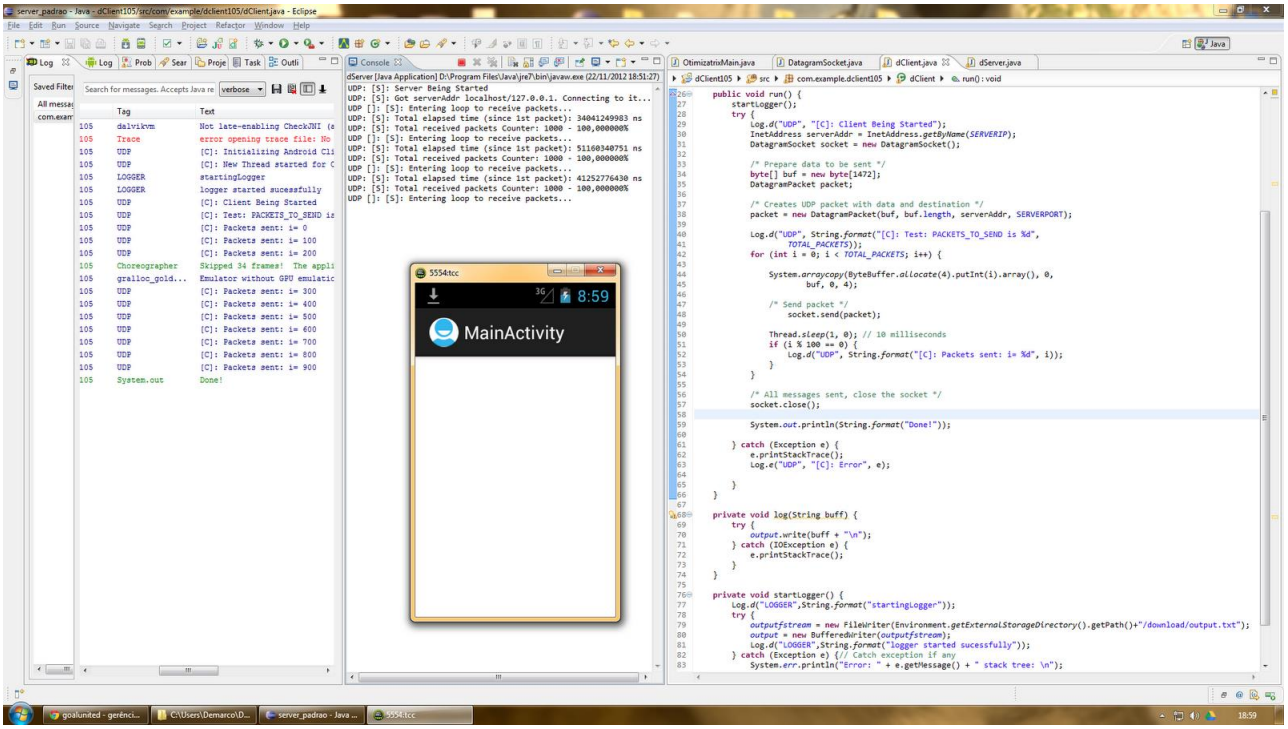

Figura 9: Teste de comunicação utilizando ambiente sem injeção de falhas.

5.3.2. Socket com Injeção de falhas

Para averiguar a funcionalidade do injetor de falhas, foram rodados 5 testes utilizando a comunicação cliente-servidor de testes. Esta comunicação avaliou a funcionalidade do injetor de falhas quando enviando datagramas via socket, com uma taxa de falha de 2%. O resultado dos testes pode ser averiguado na tabela 2 e na figura 10 abaixo:

Teste:	Tempo de comunicação	Porcentual de recebimento
1	4,07 s	98,4%
2	5,55 s	98,2%
3	2,42 s	98,2%
4	4,40 s	98,4%
5	3,41 s	97,8%

Tabela 2: Testes da comunicação com injeção de falhas.

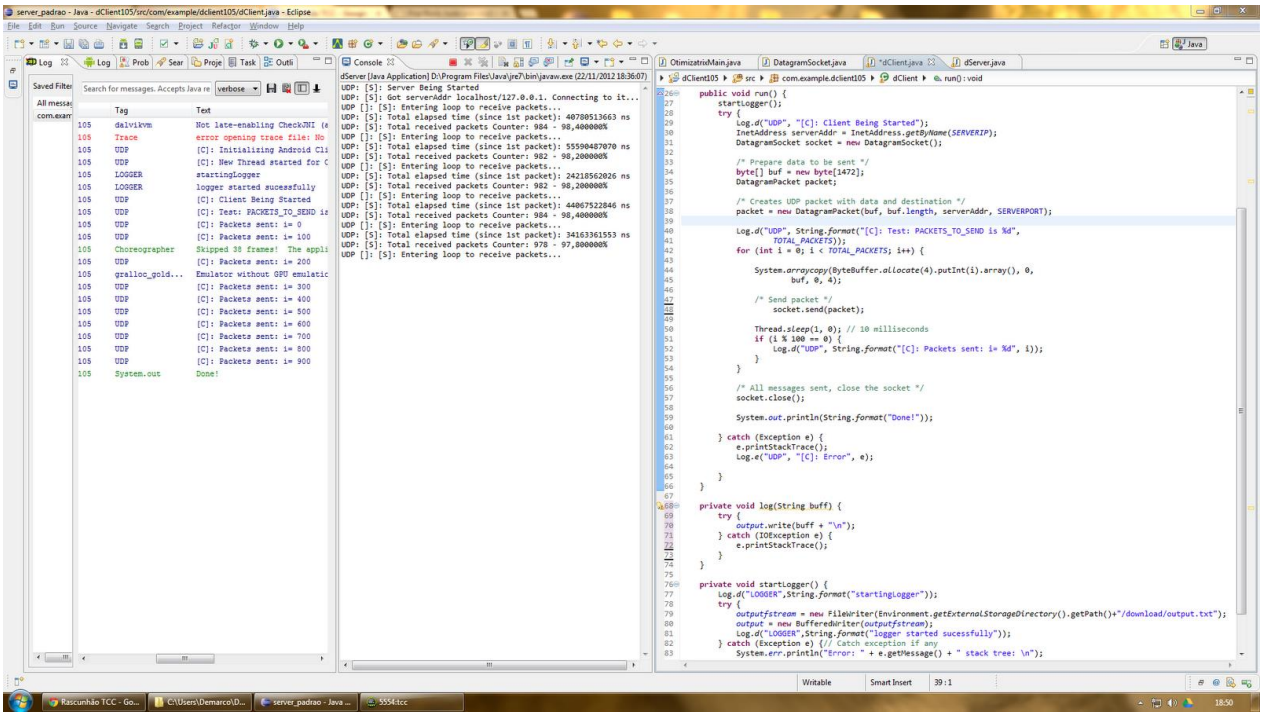

Figura 10: Log da comunicação cliente-servidor com injeção de falhas.

De acordo com estes testes, foi verificada e confirmada a funcionalidade correta do injetor de falhas.

6. AVALIAÇÃO DA APLICAÇÃO EVERNOTE

Neste experimento, ao contrário do anterior, foi usado uma aplicação real como alvo.

Para realizar a comparação entre o envio de dados enquanto usando o Android sem o injetor de falhas e o ambiente Android com a falha injetada, foi realizado o seguinte teste:

Usando o Android default, foi enviada uma mensagem e verificado o conteúdo da mensagem na aplicação Evernote Web [4], via browser no computador onde estava rodando o emulador. A mensagem tem 747 carateres, 119 palavras e está exemplificada na figura 11 abaixo:

> Bem-vindo ao Evernote! Estamos contentes que você esteja aqui. Gaste alguns minutos utilizando este guia para aprender algumas noções sobre o Evernote. Iremos apresentá-lo a variadas funcionalidades e conceitos do Evernote que irão ajudá-lo a começar a se lembrar de tudo.
> Antes que nós comecemos, estão aqui alguns termos que serão usados durante todo este documento:
> Nota : Um elemento único guardado no Evernote.
> Bloco de notas : Um recipiente para notas.
> Sincronizar : O processo pelo qual as notas do Evernote são mantidas atualizadas em todos os seus computadores, telefones, dispositivos e na Web.
> Conta : Um nome de usuário e senha que permite ao Evernote identificar as suas notas e mantê-las disponíveis para você em qualquer lugar.

Figura 11: Texto padrão utilizado nos testes.

O texto recebido pela aplicação web foi comparado com o texto padrão utilizando a ferramenta WinMerge [24], este sendo um analisador de conteúdo de arquivos que indica onde há divergências entre dois arquivos buscando fundi-los. Utilizaremos a função de identificar as diferenças e estas serão anotadas como resultado da transmissão.

6.1. Android padrão

A mensagem recebida foi exatamente a mesma enviada, como é possível verificar na figura 11 abaixo:

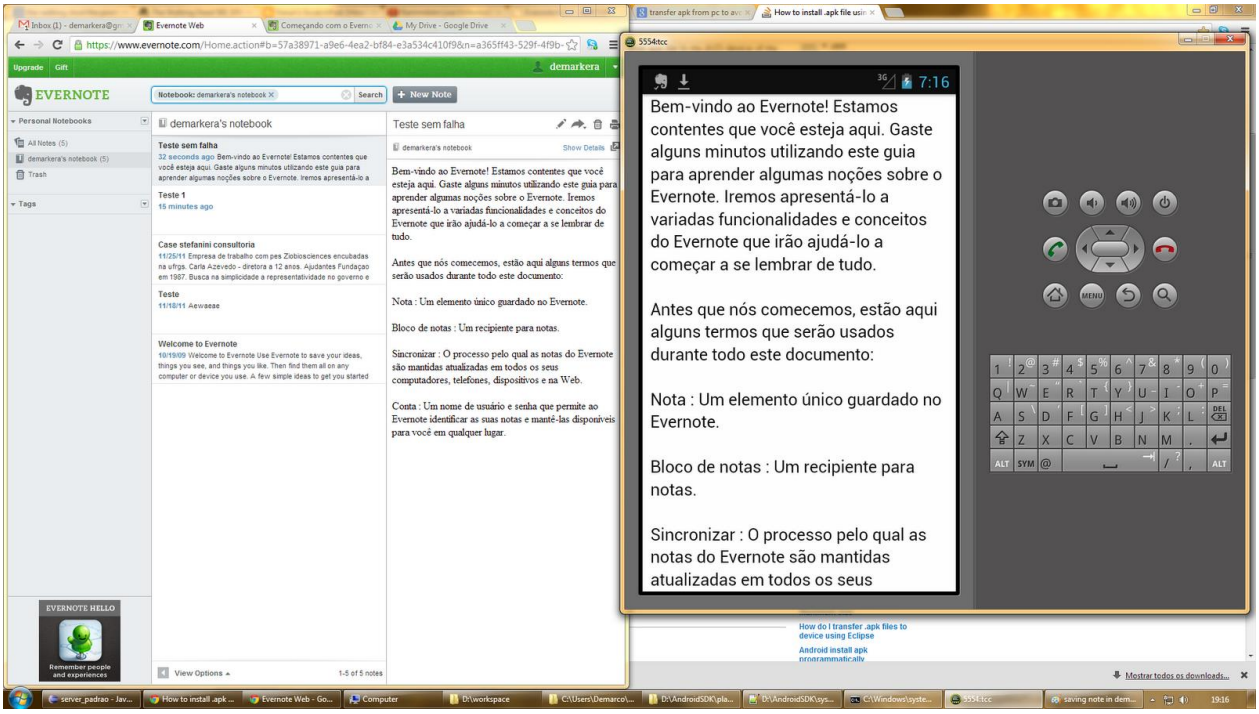

Figura 12: Confirmação de recebimento sem erros pela aplicação Evernote Web.

6.2. Android com injeção de falhas

Utilizando o Android com a injeção de falhas, é esperado que 2% das mensagens se percam. Se a aplicação não recupera mensagens perdidas, perdas de pacotes vão se refletir em perdas de informação no texto transmitido. Foram realizados 20 testes com o Android com o injetor. Destes 20 testes, 2 foram eliminados por impactarem numa interferência estatística gerada pela aplicação Evernote. Ao detectar falhas, ela inseriu caracteres extras no texto com a seginte frase: *"#some text missing#"*.

A tabela 3 a seguir corresponde a informações extraídas da transmissão do texto com 747 caracteres:

Índice teste	Caracteres recebidos	Porcentual de recepção de caracteres	Falhas detectadas
1	736	98,52%	a palavra “recipiente” (linha 3) nao foi encontrada
2	734	98,25%	“mos” na linha 2 e “dentifi” linha 6 e “te” da palavra Evernote na linha 3
3	740	99,06%	a palavra “mantidas” na linha 5 foi suprimida
4	729	97,59%	“Estamos contentes” na linha 1 não foi encontrada
5	734	98,25%	“computadores, “na linha 5
6	735	98,39%	“alguns minutos” , na linha 1
7	733	98,12%	“noções sobre o”, linha 1
8	733	98,12%	“identificar as”, linha 6
9	739	98,92%	“ e na Web”, linha 5
10	725	97,05%	“para você em qualquer “, linha 6
11	721	96,51%	“nós comecemos, estão aqui “, linha 2
12	712	95,31%	“para”, linha 1 e “este documento”, linha 2, “permite”, linha 6
13	734	98,25%	“Um recipiente“, linha 4
14	733	98,12%	“: Um nome de “, linha 6
15	735	98,39%	“Bem-vindo”, linha 1 e “os “ linha 5
16	731	97,85%	“estão aqui” linha 2 e “Conta” linha 6
17	730	97,72%	“Evernote” linha 3 e “telefones” linha 5
18	732	97,99%	“computadores” linha 5 e “notas” linha 6

Tabela 3: Falhas encontradas no texto comunicado entre o Evernote do dispositivo móvel virtual e a aplicação Web do Evernote.

Tendo em vista a quantidade de caracteres enviados e a quantidade de caracteres recebidos pela aplicação web, foi possível extrair o seguinte gráfico de porcentual de recepção de caracteres por envio exemplificado na figura 13.

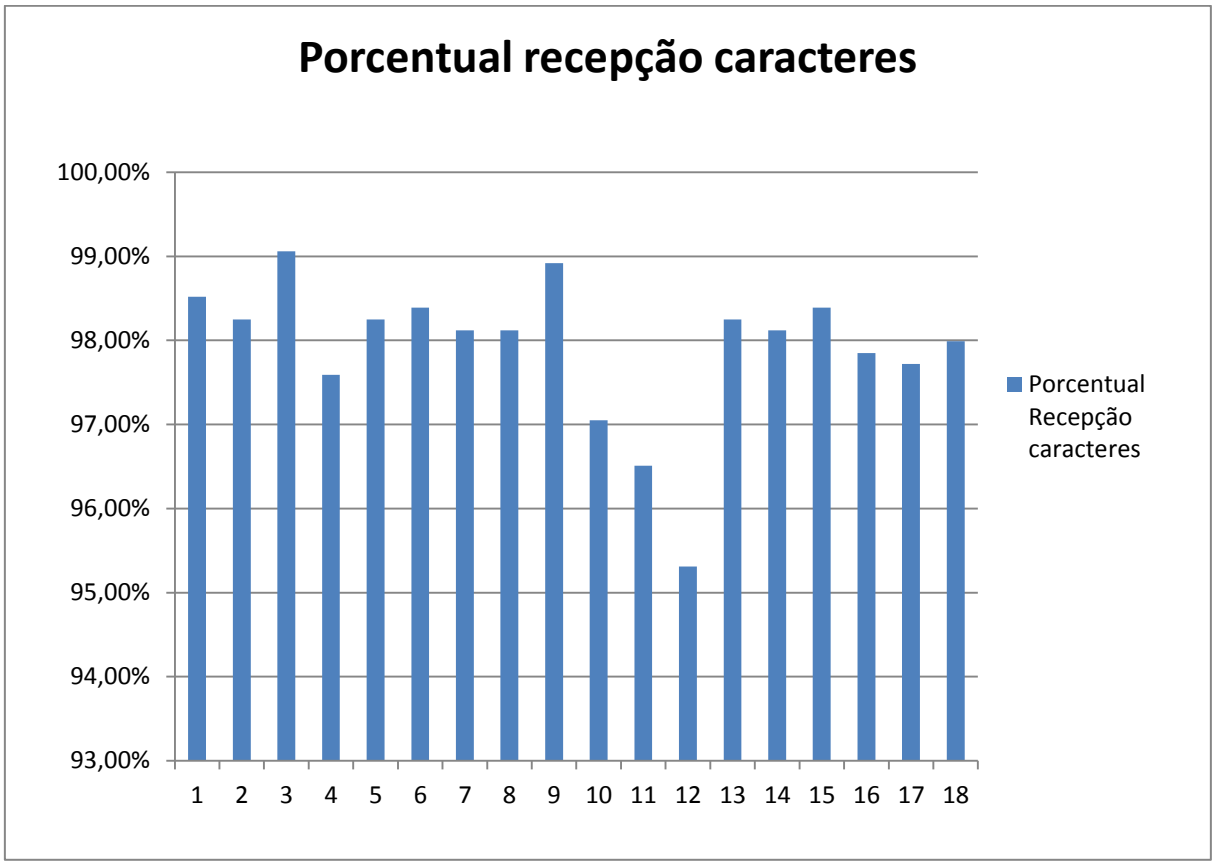

Figura 13: Gráfico de porcentual de recepção de caracteres por envio.

A média de caracteres recebidos pelas 18 transmissões da aplicação Evernote no dispositivo móvel para o Evernote na web foi de 731,4 caracteres, que representa uma média de porcentual de recepção de caracteres de 97,91%.

Analisando as falhas nas transmissões, foi possível identificar que as falhas, por padrão, suprimem uma palavra. Logo cada datagrama envolve uma palavra do texto. Com este conhecimento, é possível esperar que 2%, em média, das palavras do texto serão suprimidas. Como o texto tem 119 palavras, é possível inferir que o tamanho da palavra média é de 6,27 caracteres. Das 119 palavras, o esperado é que 2,38 palavras (2% do total aproximadamente) sejam suprimidas.

Com os dados citados acima, foi possível retirar os seguintes dados estatísticos descritos na tabela 4:

Índice teste	Caracteres Recebidos	Porcentual de recepção de caracteres	Palavras recebidas	Porcentual palavras recebidas
1	736	98,52%	118	99,5%
2	734	98,25%	116	97,4%
3	740	99,06%	118	99,5%
4	729	97,59%	118	99,5%
5	734	98,25%	118	99,5%
6	735	98,39%	117	98,3%
7	733	98,12%	116	97,4%
8	733	98,12%	117	98,3%
9	739	98,92%	116	97,4%
10	725	97,05%	115	96,6%
11	721	96,51%	115	96,6%
12	712	95,31%	115	96,6%
13	734	98,25%	117	98,3%
14	733	98,12%	116	97,4%
15	735	98,39%	117	98,3%
16	731	97,85%	116	97,4%
17	730	97,72%	117	98,3%
18	732	97,99%	117	98,3%

Tabela 4: Porcentuais de recepção de caracteres e palavras.

Analisando os resultados da tabela 4, foi obtido uma média de 98,03% de porcentual das palavras recebidas pela aplicação web do Evernote, que representa 1,97% de falhas na transmissão de palavras. A média de caracteres recebidos foi de 97,91%, que representa 2,09% de falhas no recebimento de caracteres. Este resultado encontrado está dentro do esperado, tendo em vista que a probabilidade de falha de envio ou recebimento de datagrama é de 2%.

De acordo com os resultados obtidos durante os experimentos da aplicação Evernote utilizando o Android com injetor de falhas, foi possível concluir que a aplicação não tem recursos para recuperação de erros, porém apresenta recursos para detecção dos mesmos. Esta avaliação é valiosa para a equipe de desenvolvimento do software buscar melhorias e proteções contra falhas de comunicação.

7. CONCLUSÃO

O Android tem sido o sistema operacional de dispositivos móveis que mais vem crescendo em fatias de mercado [25], alcançando, em Novembro de 2011 maioria (52,5%), dos dispositivos comercializados no mundo. Este crescimento tem sido proporcionado pela facilidade de criar aplicações para o sistema operacional e de modificar os fontes do Android para adequá-los aos dispositivos móveis construídos pelos diferentes fabricantes.

Por existirem muitos fabricantes de telefones móveis, os valores dos mesmos vem tornando-os acessíveis a grande parcela da população mundial. Com a disseminação de dispositivos, é notável e comum que haja interferência e falhas na comunicação móvel. Devido a esta interferência pela superpopulação de telefones móveis, torna-se necessário entender como que as aplicações reagem quando num meio que nem toda mensagem chega ao seu destino.

Tendo em vista que o Android é um projeto open-source e de fácil acesso aos seus fontes, foi possível utilizar a solução de alterar os fontes do Android, injetando as falhas com o mínimo de interferência na performance do mesmo. Também foi de muito valia o acesso aos gerenciadores de dispositivos virtuais (AVD Manager) e o kit de desenvolvimento do Android (Android SDK) acelerando o processo e propiciando que um desenvolvedor sem experiência neste sistema operacional desenvolva aplicações de forma rápida, sucinta e efetiva.

Buscando entender como aplicativos e o próprio sistema operacional reagem, o trabalho focou em emular um ambiente com falhas na comunicação de um dispositivo móvel virtual via injeção de falhas na comunicação.

O injetor de falhas mostrou-se transparente e de impacto praticamente nulo na performance do dispositivo virtual, abrindo a possibilidade de reuso do mesmo em futuras aplicações e trabalhos.

Porém foi possível observar dificuldades durante o processo de execução deste trabalho, principalmente enquanto construindo o ambiente: o Android dá suporte total a plataforma de construção do Android no Ubuntu 10.10, porém foi possível observar erros inesperados durante a configuração do sistema operacional de construção (Ubuntu) e durante a construção do mesmo sistema. Entretanto, no término da construção com erro, ao recomeçar a construção, esta ocorreu sem problemas sem modificar configurações.

Existem melhorias que podem ser implementadas em trabalhos futuros, como por exemplo permitir variar a taxa de falhas na comunicação, seja via arquivo de configuração ou outro método, e também explorar falhas em outros protocolos, como por exemplo o TCP.

8. REFERÊNCIAS BIBLIOGRÁFICAS

[1] ANDROID, website oficial, http://www.android.com/, acessado em Agosto de 2012.

[2] OPEN HANDSET ALLIANCE, seção Android, http://www.openhandsetalliance.com/android_overview.html, acessado em Setembro de 2012.

[3] HSUEH, M. C.; TSAI, T. K.; IYER, R. K. **Fault Injection Techniques and Tools**. Computer, Los Alamitos, CA, EUA, v.30, n.4, p.75-82, Abril. 1997.

[4] EVERNOTE, website official, http://www.evernote.com/, acessado em Outubro de 2011.

[5] 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. p. 61–74.

[6] GINDRI, A. F.; WEBER, T. S.; CECHIN S. L. **Aplicando Cadeias de Markov para injeção de perdas de pacotes no sistema Android.** Dissertação de Conclusão de Curso, UFRGS, 2012.

[7] DREBES, R.; JAQUES-SILVA, G.; TRINDADE, J.; WEBER, T. A kernel-based communication fault injector for dependability testing of distributed systems. **Hardware and Software, Verification and Testing**, p.177–190, 2006.

[8] SIQUEIRA, T.; FISS, B.; WEBER, R.; CECHIN, S.; WEBER, T. Applying FIRMAMENT to test the SCTP communication protocol under network faults. **In Test Workshop**, 2009. LATW '09. 10th Latin American, p. 1–6, 2009.

[9] KRISHNA, P.; VAIDYA, N.; PRADHAN, D.; "Recovery in distributed mobile environments". **In proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems**, pp. 83-88, 1993.

[10] ARLAT, J.; AGUERA, M.; AMAT, Y.; CROUZET, J.; LAPRIE, J. C.; MARTINS, E.; POWELL, D.; "Fault injection for dependability validation: A methodology and some applications". **Software Engineering, IEEE Transacions**, v. 16, n. 2, p. 166-182, 1990.

[11] KANOUN, K.; SPAINHOWER, L. Dependability Benchmarking for Computer Systems. **Wiley-IEEE Computer Society**, 2008.

[12] IYER, R. K.; TANG, D.; Experimental Analysis of Computer System Dependability, in **Fault-Tolerant Computer System Design**, D.K. Pradhan, ed., Prentice-Hall. Prof. Tech. Ref., Upper Saddle River, N.J., p. 282-392.

[13] ANDROID DEVELOPERS, ADT plugin, http://developer.android.com/tools/sdk/eclipse-adt.html, acessado em Agosto de 2012.

[14] ANDROID DEVELOPERS, Managing Virtual Devices, http://developer.Android.com/tools/devices/index.html, acessado em Agosto de 2012.

[15] UBUNTU, Ubuntu version 10.04.4 LTS (Lucid Lynx), http://releases.ubuntu.com/lucid/, acessado em Setembro de 2012.

[16] JAVA, Java 6.36 SE Downloads, http://www.oracle.com/technetwork/java/javase/downloads/index.html, acessado em Agosto de 2012.

[17] ANDROID DEVELOPERS, "Initializing a build environment", Downloading the Source, Building and Running e Building for Devices. http://source.android.com/source/initializing.html, acessado em Agosto de 2012.

[18] ANDROID OPEN SOURCE REPOSITORY, Version control with Repo and Git, http://source.android.com/source/version-control.html, acessado em Agosto de 2012.

[19] GIT, http://git-scm.com/, acessado em Agosto de 2012.

[20] ANDROID DRAWER, Downloading Evernote 4.2, http://www.Androiddrawer.com/6023/download-evernote-4-2-app-apk/#.UKPRaIdfAkQ, acessado em Outubro de 2012.

[21] WIKIPEDIA, Fault Injection, http://en.wikipedia.org/wiki/Fault_injection, acessado em Outubro de 2012.

[22] PIVARO, G. F.; **Redução da taxa de queda de chamada em rede celular GSM por meio de ajustes dos parametros de cobertura**. 2008. Dissertação (Mestrado em Engenharia Elétrica) - Universidade de São Paulo Escola de engenharia de São Carlos, São Paulo.

[23] WIKIPEDIA, Distribuição Uniforme, http://pt.wikipedia.org/wiki/Distribui%C3%A7%C3%A3o_uniforme, acessado em Outubro de 2012.

[24] WINMERGE, http://winmerge.org/, acessado em Outubro de 2012.

[25] MOBILE MARKET SHARE, http://en.wikipedia.org/wiki/Mobile_operating_system, acessado em Novembro de 2012.

[26] BIBLIOTECA INF UFRGS, http://www.inf.ufrgs.br/biblio/index.php?option=com_content&view=article&id=57&Itemid=63, acessado em Dezembro de 2012.

[27] MARMITT, H.F.; CECHIN, S. L.; WEBER T.S.; **Modelos para injeção de falhas em ambientes móveis**. In: Escola Regional de Redes de Computadores, 2010, Alegrete, RS. ERRC – 8. Escola Regional de Redes de Computadores. Alegrete: Unipampa, v. 1, p.1-8, 2010.

[28] TANENBAUM, A. S.; **Computer Networks**. Prentice Hall, terceira edição, 1996.

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

Apêndice A: Código do cliente da aplicação de teste

Este apêndice exemplifica a aplicação cliente do teste do injetor de falhas.

```
package com.example.dclient105;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.nio.ByteBuffer;

import android.os.Environment;
import android.util.Log;


public class dClient implements Runnable {

	/* Parametros do trabalho */
	static String SERVERIP = "10.0.2.2";
	int SERVERPORT = 54654;
	static int TOTAL_PACKETS = 1000;

	private static String diretorio = "D:\\androidLog";
	static BufferedWriter output;
	private static FileWriter outputfstream;

	public void run() {
		startLogger();
		try {
			Log.d("UDP", "[C]: Client Being Started");
			InetAddress serverAddr =
InetAddress.getByName(SERVERIP);
			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, SERVERPORT);

			Log.d("UDP", String.format("[C]: Test: PACKETS_TO_SEND
is %d",
					TOTAL_PACKETS));
			for (int i = 0; i < TOTAL_PACKETS; i++) {
```

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

                              socket.send(packet);

                        Thread.sleep(1, 0); // 10 milliseconds
                        if (i % 100 == 0) {
//                            Log.d("UDP", String.format("[C]: Packets
sent: i= %d", i));
                        }
                  }

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

                  System.out.println(String.format("Done!"));

            } catch (Exception e) {
                  e.printStackTrace();
                  Log.e("UDP", "[C]: Error", e);

            }
      }

      private void startLogger() {
            Log.d("LOGGER",String.format("startingLogger"));
            try {
                  outputfstream = new
FileWriter(Environment.getExternalStorageDirectory().getPath()+"/download
/output.txt");
                  output = new BufferedWriter(outputfstream);
                  Log.d("LOGGER",String.format("logger started
sucessfully"));
            } catch (Exception e) {// Catch exception if any
                  System.err.println("Error: " + e.getMessage() + " stack
tree: \n");
                  e.printStackTrace();
            }
      }
}
```

Apêndice B: código do servidor da aplicação de teste

Este apêndice exemplifica a aplicação servidor do teste do injetor de falhas.

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.SocketTimeoutException;
import java.util.BitSet;

public class dServer {
	/* Emulator's IP */
	public static final String SERVERIP = "localhost";

	/* Server's port */
	public static final int SERVERPORT = 54654;

	/* Total expected packets */
	public static final int TOTAL_PACKETS = 1000;

	public static void main(String[] args) {
//		startLogger();
		try {
			System.out.println(String.format("UDP: [S]: Server
Being Started"));

			/* Search server by IP address */
			InetAddress serverAddr =
InetAddress.getByName(SERVERIP);
			System.out.println(String.format(
					"UDP: [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 */
		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);

		System.out.println(String.format("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();
			int i = 0;
```

```
                                    do {

                                          /* Receive next packets */
                                          socket.receive(packet);
                                          i++;
                                          /* 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));
                                          if (i % 100 == 0)
                                                System.out.println("UDP:
                                                Received packets: " + i);
                                    } while (true);
                              } catch (SocketTimeoutException e) {
                              /* Get final time reference and */
                              fin_time = System.nanoTime();

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

                              System.out.println(String.format("UDP: [S]: Total
                              received packets Counter: %d -
                              %f%%",receivedPacketsCounter,100*(double)
                              receivedPacketsCounter  / (double)
                              TOTAL_PACKETS));

                              }
                        } while (true);
                  } catch (Exception e) {
                        System.out.println(String.format("UDP: [S]: Error",
                  e));
                  }
            }
}
```

Apêndice C: DatagramSocket alterado (somente trechos entre receive e send)

Abaixo temos as alterações efetuadas na classe *DatagramSocket* do Android com o injetor de falhas. Somente o trecho de código entre o método *receive* e *send* estão descritos abaixo.

```
    /* Receives a packet from this socket and stores it in the argument {@code
     * pack}. All fields of {@code pack} must be set according to the data
     * received. If the received data is longer than the packet buffer size it
     * is truncated. This method blocks until a packet is received or a timeout
     * has expired.
     *
     * @param pack
     *            the {@code DatagramPacket} to store the received data.
     * @throws IOException
     *                if an error occurs while receiving the packet.
     */
    public synchronized void receive(DatagramPacket pack) throws IOException {
        checkOpen();
        ensureBound();
        if (pack == null) {
            throw new NullPointerException();
        }
        if (pendingConnectException != null) {
            throw new SocketException("Pending connect failure",
pendingConnectException);
        }
        pack.resetLengthForReceive();
        if(ableToCommunicate()){  impl.receive(pack);  }
    }

    /*
     * Calculates if the datagram is able to be communicated
     */
    private boolean ableToCommunicate() {
      final int minChance = 0;
      final int maxChance = 100;
      int generatedNumber = minChance
                  + (int) (Math.random() * ((maxChance - minChance)) + 1);
      final int prob = 2; // 2%
      if (generatedNumber > prob)
                  return true;
      else {
                  return false;
      }
    }
```

```
    /**
     * Sends a packet over this socket.
     *
     * @param pack
     *            the {@code DatagramPacket} which has to be sent.
     * @throws IOException
     *                if an error occurs while sending the packet.
     */
    public void send(DatagramPacket pack) throws IOException {
        checkOpen();
        ensureBound();

        InetAddress packAddr = pack.getAddress();
        if (address != null) { // The socket is connected
            if (packAddr != null) {
                if (!address.equals(packAddr) || port != pack.getPort())
{
                    throw new IllegalArgumentException("Packet address
mismatch with connected address");
                }
            } else {
                pack.setAddress(address);
                pack.setPort(port);
            }
        } else {
            // not connected so the target address is not allowed to be
null
            if (packAddr == null) {
                throw new NullPointerException("Destination address is
null");
            }
        }
        if(ableToCommunicate()){ impl.send(pack); }
    }
```