UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

MOBILE COMMAND: FERRAMENTA PARA CONTROLE REMOTO DO SISTEMA OPERACIONAL LINUX A PARTIR DE UM SMARTPHONE ANDROID

DIEGO ARMANDO GUSAVA

BLUMENAU
2011

2011/1-14

DIEGO ARMANDO GUSAVA

MOBILE COMMAND: FERRAMENTA PARA CONTROLE REMOTO DO SISTEMA OPERACIONAL LINUX A PARTIR DE UM SMARTPHONE ANDROID

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Mauro Marcelo Mattos, Dr - Orientador

BLUMENAU
2011

2011/1-14

MOBILE COMMAND: FERRAMENTA PARA CONTROLE REMOTO DO SISTEMA OPERACIONAL LINUX A PARTIR DE UM SMARTPHONE ANDROID

Por

DIEGO ARMANDO GUSAVA

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Mauro Marcelo Mattos, Doutor – Orientador, FURB
Membro:	Prof. Francisco Adel Péricas, Mestre – FURB
Membro:	Prof. Antônio Carlos Tavares, Mestre – FURB

Blumenau, 28 de junho de 2011

Dedico este trabalho aos meus pais, Armando e Iolanda, e a todos que me ajudaram diretamente na realização deste trabalho.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

Aos meus pais por sempre me incentivarem a estudar e pelo apoio na realização deste trabalho.

À minha namorada Ellen Laurentino pela força e compreensão neste momento muito importante.

Aos professores que contribuíram para minha formação.

Ao meu orientador, Mauro Mattos, por ter acreditado na conclusão deste trabalho.

A todos que contribuíram indiretamente para a realização deste trabalho.

Algumas pessoas querem que algo aconteça, outras desejam que algo aconteça, outras fazem acontecer.

Michael Jordan

RESUMO

Este trabalho apresenta o desenvolvimento de uma ferramenta que utiliza a arquitetura cliente/servidor, possibilitando a interação entre um dispositivo móvel Android e um computador *desktop*/servidor. A partir de um dispositivo móvel é possível enviar comandos para o *shell* do sistema operacional do computador *desktop* e visualizar o resultado no *smartphone*. São disponibilizadas todas as operações de manipulações de arquivos suportados pelo Java. Além disso, na plataforma Linux, é possível atuar sobre os processos disparando-se instâncias de *shell* com comandos específicos daquele ambiente.

Palavras-chave: *Smartphone*. Linguagem de comandos. Sistema operacional. Acesso remoto.

ABSTRACT

This work presents the development of software in client/server architecture, enabling the interaction between an Android mobile device and a desktop computer/server. From a mobile device can send commands to the shell of the desktop computer operating system and view the result on the smartphone. All file manipulation operations supported by Java are available. In addition, you can act on the processes, firing instances of shell with commands specific to the Linux environment.

Key-words: Smartphone. Commands language. Operating system. Remote access.

LISTA DE ILUSTRAÇÕES

LISTA DE SIGLAS

ADT – Android *Development Tools*

API – *Application Program Interface*

AVD – Android *Virtual Device*

CGI – *Common Gateway Interface*

DVM – Dalvik *Virtual Machine*

FTP – *File Transfer Protocol*

GPRS/EDGE – *General Packet Radio Service/Enhanced Data for Global Evolution*

HTTP – *Hypertext Transfer Protocol*

ICMP – Internet *Control Message Protocol*

IP – Internet *Protocol*

JME – Java *Micro Edition*

RAM – *Random Access Memory*

SDK – *Software Development Kit*

SMTP – *Simple Mail Transfer Protocol*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

URL – *Uniform Resource Locator*

XML - *Extensible Markup Language*

XMPP – *eXtensible Messaging and Presence Protocol*

SUMÁRIO

1 INTRODUÇÃO

Computadores pessoais estão sendo substituídos por *smartphones*, por realizar tarefas que até então só computadores realizavam e principalmente pela mobilidade que eles oferecem. Isso gera interesse por parte das pessoas, e companhias veem isso como uma forma de investir e ganhar dinheiro (VANCE, 2009).

> Hoje, existem 1,5 bilhões de aparelhos de televisão em uso em todo o mundo e um bilhão de pessoas têm acesso a internet. No entanto, quase três bilhões de pessoas têm um telefone celular, tornando o aparelho um dos produtos de consumo mais bem-sucedidos do mundo. Dessa forma, construir um aparelho de celular superior melhoraria a vida de inúmeras pessoas em todo o mundo. (OPEN HANDSET ALLIANCE, 2010).

Com um número crescente de aparelhos móveis, o uso deles torna-se frequente e o número de softwares fabricados aumenta, mas ainda não cobrem todas as necessidades que os usuários desejam. Isto normalmente acontece por falta de espaço, processamento ou ainda por motivos de facilidade e segurança (MACE, 2006).

Conforme Oliveira, Carissimi e Toscani (2003, p. 17), um sistema operacional recebe requisições de usuários através de uma interface de comunicação, a qual implementa uma determinada linguagem, denominada linguagem de comandos (*command language*). Esta linguagem de comunicação é interpretada por um componente do sistema operacional denominado interpretador de comandos.

Diante do que foi exposto, buscando a mobilidade e adicionando a facilidade que a internet oferece de se comunicar com outros computadores, pretende-se desenvolver uma interface de comunicação remota de comandos de um sistema operacional baseado na arquitetura cliente/servidor de tal forma que seja possível a partir de um *smartphone*, disparar comandos no computador *desktop* e receber o resultado no *smartphone*, uma espécie de *shell* remoto.

1.1 OBJETIVOS DO TRABALHO

O objetivo do trabalho é o desenvolvimento de uma ferramenta para possibilitar o envio de requisições de serviços a um sistema operacional *desktop*, disparados a partir de um *smartphone*, que poderá visualizar o resultado gerado.

Os objetivos específicos do trabalho são:

a) disponibilizar um protocolo de comunicação entre o *smartphone* e o *desktop*;
b) disponibilizar uma aplicação no *smartphone* para acesso a comandos do sistema operacional a ser controlado;
c) validar o uso da ferramenta através de um estudo de caso envolvendo operações de gerenciamento de processos de um sistema operacional Linux.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está estruturado em quatro capítulos. No segundo capítulo é apresentada a fundamentação teórica necessária para o desenvolvimento do trabalho. Nele são discutidos sobre a arquitetura cliente/servidor, plataforma Android e *shell* Linux. Por último são apresentados os trabalhos correlatos.

O terceiro capítulo trata sobre o desenvolvimento desta ferramenta, intitulada como Mobile Command. Nele é apresentada a especificação do sistema, iniciando pelos diagramas de caso de uso, assim como digramas de classe e de sequência. Comentários sobre a implementação, as técnicas utilizadas, ferramentas utilizadas e operacionalidade são apresentadas, assim como os resultados obtidos.

O quarto capítulo refere-se às conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este trabalho inicialmente apresenta os conceitos do protocolo *Transmission Control Protocol/Internet* Protocol (TCP/IP) e a arquitetura cliente/servidor, em seguida é apresentada a plataforma Android e os recursos que ela disponibiliza para desenvolver aplicações para dispositivos móveis. Na sequência é apresentado *shell* do Linux. Por fim, são apresentados trabalhos correlatos que auxiliaram no entendimento de aplicativos móveis.

2.1 CONJUNTO DE PROTOCOLOS TCP/IP

O conjunto de protocolos denominado TCP/IP permite que diversos computadores de modelos e fabricantes diferentes, executando sistemas operacionais distintos, comuniquem entre si através da troca de pacotes de informação. Os protocolos estão devidamente organizados em uma estrutura de camadas, em que cada uma é responsável por uma parte da comunicação. Um conjunto de protocolos organizados em camadas lógicas origina uma pilha de protocolos, pelo que se conclui que a pilha de protocolos internet representa a forma como se organizam logicamente os diversos protocolos comumente chamados por TCP/IP (KUROSE, 2006, p. 114).

O conjunto de protocolos TCP/IP é composto por quatro camadas, cada uma com suas funções próprias (PANWAR, 2004, p. 1):

a) camada de interface de rede: inclui os *softwares* de controle das interfaces de rede acoplados aos sistemas operacionais e aos correspondentes dispositivos físicos de interligação, podendo dizer-se que esta camada trata dos aspectos ligados aos equipamentos que utilizam o meio de comunicação (*Ethernet*, ATM, ADSL, etc);
b) camada de internet: gerência a transferência dos pacotes de informação na rede, incluindo-se nesta camada os mecanismos do seu encaminhamento. Alguns dos protocolos da pilha internet que são manipulados nesta camada são o *Internet Protocol* (IP) e o *Internet Control Message Protocol* (ICMP);
c) camada de transporte: fornece controles de fluxo de dados entre dois computadores, essencialmente através dos protocolos de transporte *User Datagram Protocol* (UDP) sem conexão e *Transmission Control Protocol* (TCP)

orientado a conexão;

a) camada de aplicação: a camada de aplicação é a camada que a maioria dos programas de rede usa de forma a se comunicar através de uma rede com outros programas, podendo citar: protocolo para acesso remoto (TELNET), *File Transfer Protocol* (FTP), *Simple Mail Transfer Protocol* (SMTP) e *Hypertext Transfer Protocol* (HTTP).

2.2 MODELO CLIENTE/SERVIDOR

O modelo cliente/servidor é caracterizado pela presença de processos clientes e processos servidores. Os servidores são responsáveis por atender e processar pedidos dos clientes. A comunicação da arquitetura é do tipo requisição e resposta, ou seja, os clientes solicitam algum tipo de serviço aos servidores e esses, por sua vez, atendem aos pedidos dos clientes. Este tipo de comunicação tem um nível de complexidade baixa, e por não se tratar de um protocolo orientado a conexão possui um baixo custo de comunicação (STEVENS, 2004, p. 121). Na arquitetura cliente/servidor é possível que um servidor atue como cliente em determinada situação.

Cada servidor pode disponibilizar um ou vários tipos de serviços aos clientes, sendo divididos em servidores interativos, servidores concorrentes, servidores orientados a conexão, servidores sem conexão, servidores com estado e servidores sem estado. Servidores interativos recebem uma solicitação de um cliente, prepara a resposta e a envia de volta ao cliente para só depois atender uma solicitação de outro cliente. Servidores concorrentes definem que para cada nova solicitação de serviço de um cliente, é criado um novo processo servidor e desta forma, o servidor atende todas as solicitações concorrentemente, minimizando o tempo de resposta aos clientes. Servidores orientados a conexão formam uma comunicação confiável entre um cliente e um servidor e é fornecida pelo uso de um protocolo de comunicação orientado a conexão como o protocolo de transporte TCP da família de protocolos TCP/IP e toda a responsabilidade de verificação da integridade dos dados, controle e ordenamento de mensagens ficam a cargo do protocolo de comunicação. Servidores sem conexão utilizam protocolos de comunicação sem conexão, como o protocolo de transporte UDP, também da família de protocolos TCP/IP, mas não garantem que as mensagens entre os clientes e os servidores sejam entregues de uma forma correta. Para garantir a integridade dos

dados neste caso, é necessário implementar o controle diretamente no servidor. Servidores com estado se recordam das operações realizadas pelos clientes. As informações de estado são geralmente armazenadas em memória *Random Access Memory* (RAM). A principal desvantagem em se utilizar servidores com estado é a possibilidade de ocorrer inconsistência nas informações de estado guardadas pelo servidor. A inconsistência nas informações de estado pode ser causada pela perda de mensagens na rede ou pela queda do cliente ou do servidor. Servidores sem estado são os que não guardam qualquer informação de estado das operações realizadas pelos clientes. A premissa básica desses servidores é que toda operação realizada é dita idempotente, ou seja, qualquer repetição da operação não causará nenhum dano, pois o resultado será sempre o mesmo (COMER; STEVENS, 1997, p. 363).

2.3 PLATAFORMA ANDROID

Em 05 de novembro de 2007, o Open Handset Alliance (OHA) anunciou a plataforma Android. OHA é um grupo de mais de 30 empresas que estão trabalhando juntas para oferecer uma plataforma moderna e flexível para o desenvolvimento de aplicações coorporativas.

2.3.1 Definição

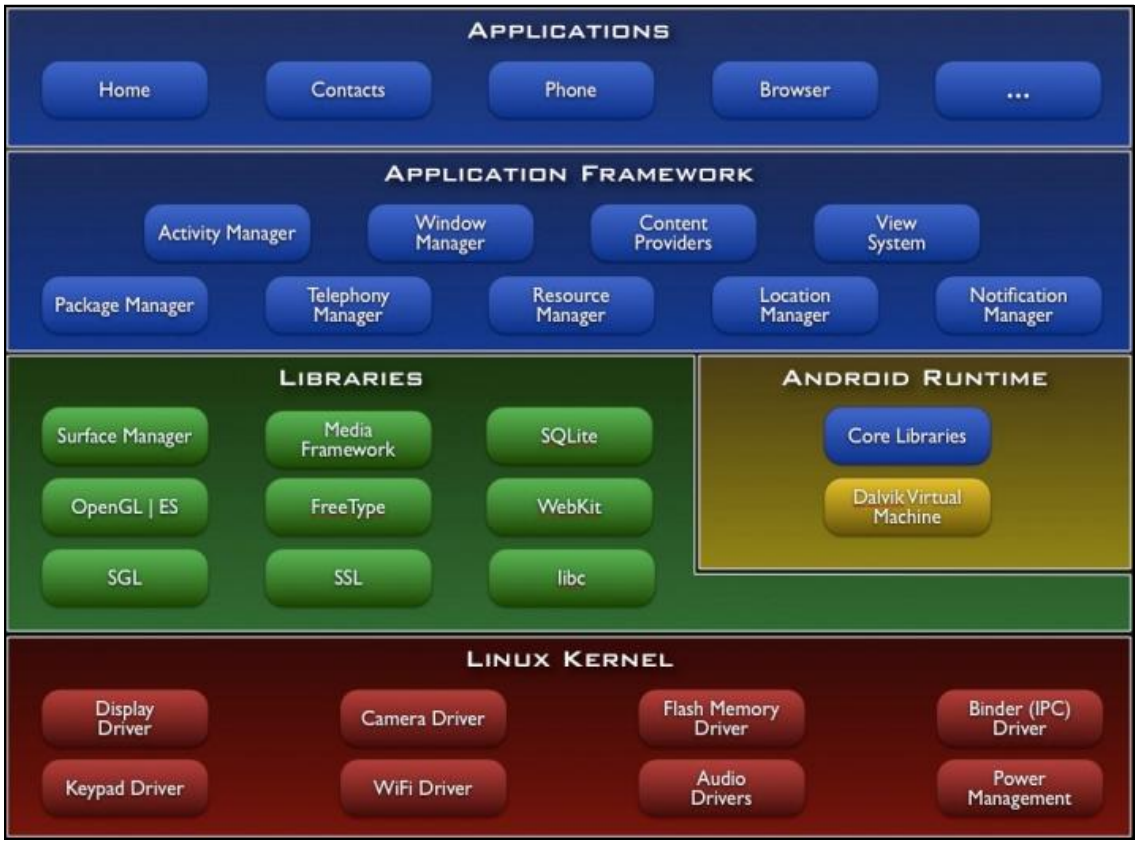
Android segundo Google (2011a), é uma pilha de softwares para dispositivos móveis que inclui um sistema operacional, um *middleware* e um conjunto de aplicações chaves. Os desenvolvedores podem criar aplicações para a plataforma usando o kit de desenvolvimento de software (SDK). As aplicações para essa plataforma são escritas usando a linguagem de programação Java e executam sobre o Dalvik Virtual Machine (DVM), uma máquina virtual própria para dispositivos com restrições de recursos, como pouca capacidade computacional, baixa capacidade de armazenamento e baterias de curta duração.

2.3.2 Arquitetura Android

A arquitetura do sistema operacional Android, segundo Google (2011b), é formada por

quatro camadas: núcleo do sistema, bibliotecas, fr*amework* de aplicação e as aplicações.

A Figura 1 mostra as camadas que o sistema operacional Android compõe.



Fonte: Aquino (2007, p. 5).

Figura 1 - Arquitetura do Android

De acordo com a figura acima, são apresentadas agora, as camadas que a arquitetura Android possui (AQUINO, 2007, p. 5):

a) primeira camada: é formada pelo núcleo do sistema (*kernel*), funciona como uma camada de abstração entre o hardware e o restante de softwares da plataforma. Utilizando Linux, é responsável pelo gerenciamento de memória, processos, *threads*, pilha de protocolos de rede, módulo de segurança e vários módulos do núcleo de infraestrutura;

b) segunda camada: contém um conjunto de bibliotecas C/C++ usada por vários componentes do sistema e a Dalvik Virtual Machine. A DVM foi escrita de forma que um dispositivo possa executar múltiplas máquinas virtuais concorrentemente de maneira eficiente. Ela usa o *kernel* do Linux para prover a funcionalidade de múltiplas *threads* e gerenciamento de memória de baixo nível. Cada aplicação do Android roda em um processo separado, com sua própria máquina virtual, número do processo e dono do processo. Isso garante que caso a aplicação apresente erros, ela possa ser isolada e removida da memória sem comprometer o resto do sistema.

Ao desenvolver as aplicações em Java, a DVM compila o *bytecode* (`.class`) e converte para o formato *Dalvik Executable* (`.dex`), que representa a aplicação do Android compilada. Depois disso, os arquivos `.dex` e outros recursos do projeto são compactados em um único arquivo com a extensão *Android Package File* (`.apk`), que representa a aplicação final;

c) terceira camada: é composta por *frameworks* de aplicações. Os desenvolvedores têm acesso completo a mesma interface de programação de aplicativos que é usada pelas aplicações essenciais da plataforma. Esta camada fornece um conjunto de serviço tais como *Activity Manager*, que gerência o ciclo de vida das aplicações; *Package Manager*, que mantêm quais aplicações estão instaladas no dispositivo; *Window Manager*, que gerência as janelas das aplicações*; Telephony Manager*, que são os componentes para acesso aos recursos de telefonia; *Content Providers*, que permitem que as aplicações acessem dados de outras aplicações ou compartilhem os seus próprios dados; *Resource Manager*, que fornece acesso aos recursos gráficos e arquivos de layout; *View System*, que é um conjunto rico e extensível de componentes de interface do usuário; *Location Manager*, que gerência a localização do dispositivo; *Notification Manager*, que permite que todas as aplicações exibam alertas na barra de status e *XMPP Service* que é o suporte para o uso do protocolo *eXtensible Messaging and Presence Protocol* (XMPP);

d) quarta camada: é formada pelos aplicativos propriamente ditos, e contempla aplicações como cliente de e-mail, programa de envio de mensagens, calendários, mapas, navegador web, contatos, entre outros.

2.3.3 Componentes básicos

Para Steele e To (2011, p. 23) as aplicações Android são construídas basicamente a partir de cinco tipos de componentes, que são definidos pela própria arquitetura do Android, são eles:

a) componente *activity*: responsável por controlar os eventos da tela e definir qual *Extensible Markup Language* (XML) será responsável por desenhar a interface gráfica do usuário;

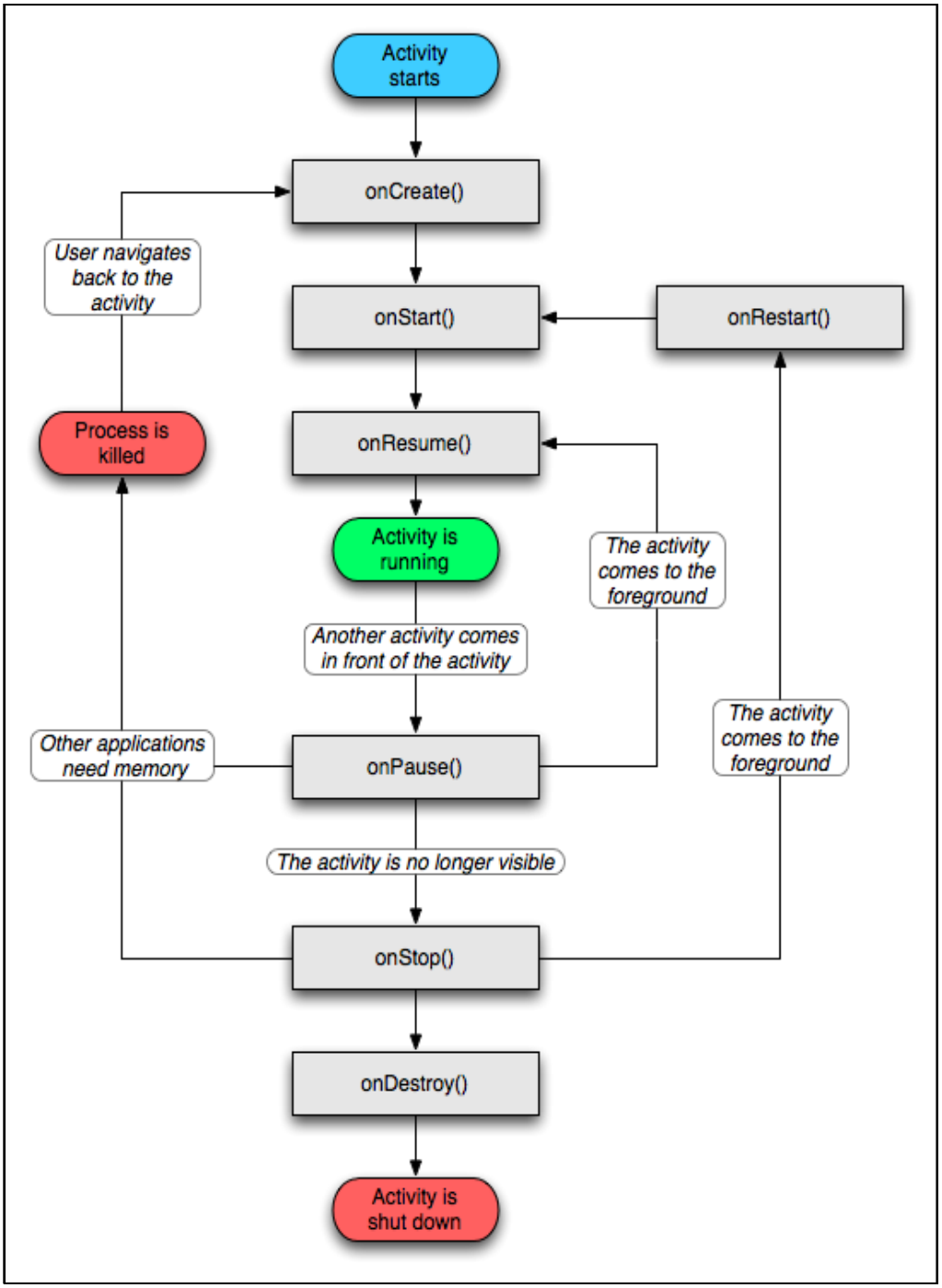
b) componente *intent*: é o coração do Android e representa uma mensagem da aplicação para o sistema operacional, solicitando que algo seja realizado,

possibilitando a integração de diferentes aplicações;

c) componente *service*: utilizado para executar um serviço em segundo plano, geralmente vinculado a algum processo que deve executar por tempo indeterminado e tem um alto consumo de recursos, memória e CPU;
d) componente *broadcast receiver*: parecido com componente s*ervice*, só que este responsável por executar em segundo plano durando pouco tempo e sem utilizar uma interface gráfica. Seu objetivo é receber uma mensagem (componente *intent*) e processá-la sem que o usuário perceba;
e) componente c*ontent provider*: é um provedor de conteúdo. O Android permite armazenar informações de várias formas, utilizando banco de dados, arquivos e o sistema de preferências. Contudo, geralmente essas informações ficam salvas dentro do pacote de sua aplicação, e somente a aplicação que criou o banco de dados ou o arquivo pode ter acesso às informações. Para isso foi criado o componente c*ontent provider*, o qual permite que determinadas informações sejam públicas para qualquer aplicação.

2.3.4 Aplicação Android: ciclo de vida

Uma aplicação Android é executada em um processo único e em sua própria máquina virtual, e seu ciclo de vida é gerenciado pelo sistema operacional, através da pilha de *activity* (*activity stack*), de acordo com as necessidades do usuário e nos recursos disponíveis (GOOGLE, 2011b). Uma aplicação pode conter várias *activitys*, e cada *activity* representa uma tela da aplicação, ela pode-se encontrar em três estados possíveis: executando, temporariamente interrompida em segundo plano ou completamente destruída. Na Figura 2 é usado um fluxograma para exemplificar o ciclo de vida de uma *activity*.



Fonte: Google (2011a).

Figura 2 – Ciclo de vida do componente *activity*

Segundo Google (2011a) existem três subniveis do ciclo de vida principal, que por sua vez ficam se repetindo durante a execução da aplicação. Estes ciclos são chamados de *entire lifetime*, *visible lifetime* e *foreground lifetime*. Cada um desses ciclos se inicia durante a chamada de um dos métodos que controlam o ciclo de vida da *activity* e terminam quando outro método é chamado. Os principais métodos que controlam o ciclo de vida são `onCreate()`, `onStart()`, `onStop()`, `onResume()`, `onPause()` e `onDestroy()`.

Entire lifetime refere-se ao ciclo de vida como um todo, e ocorre apenas uma vez, onde

uma *activity* tem início e fim. Ela acontece entre as chamadas do método `onCreate(bundle)` e `onDestroy()`, os quais são chamados apenas uma única vez, quando a *activity* é criada e destruída, respectivamente.

Visible lifetime a *activity* já iniciou seu ciclo de vida, mas pode tanto estar no topo da pilha interagindo com o usuário como também temporariamente parada em segundo plano. Esse ciclo ocorre entre os métodos `onStart()` e `onStop()`. O método `onStart()` é chamado logo após o método `onCreate()`, que é chamado apenas uma única vez. No método `onStart()` é definido um ciclo que se repete entre as chamadas dos métodos, como mostrado na Figura 3.

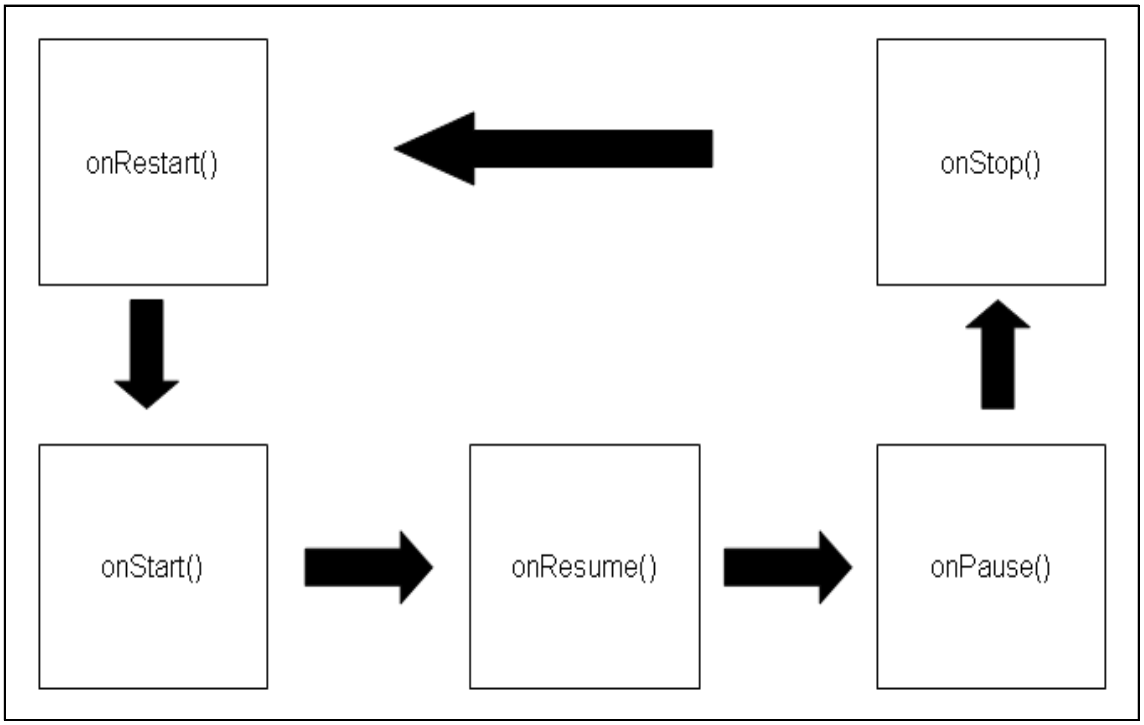


Figura 3 - *Visible lifetime*

Durante esse período o usuário pode estar interagindo com uma *activity*, ou não, dependendo em que estado ela se encontra, talvez outra *activity* possa estar no topo da pilha e em execução, por exemplo. Uma *activity* só ocupa o topo da pilha (*activity stack*) quando o método `onResume()` é executado. Portanto, esse ciclo de vida engloba todo o tempo que a *activity* está no topo ou está em segundo plano esperando outra *activity* terminar. Sempre que o método `onStart()` é chamado, automaticamente o método `onResume()` também é invocado.

Foreground lifetime a *activity* está no topo da pilha interagindo com o usuário. Esse ciclo de vida ocorre entre os métodos `onResume()` e `onPause()`. Uma *activity* pode frequentemente alternar entre os estados executando e pausado. Por exemplo, o método `onPause()` pode ser chamado quando o usuário atende uma ligação telefônica e quando o

usuário retorna, o método `onResume()` é chamado para continuar a aplicação.

Durante a execução dos métodos `onPause()`, `onStop()` e `onDestroy()`, o sistema operacional pode destruir a *activity* caso as condições de memória estejam críticas.

2.3.5 Componente *service*: ciclo de vida

O ciclo de vida do componente *service* é muito similar ao ciclo de vida do componente *activity*, mas existem algumas diferenças importantes a serem destacadas (GOOGLE, 2011b):

a) `onCreate()` e `onStart()`:o serviço pode iniciar quando o cliente chama o método `Context.startService(Intent)`. Se o serviço não estiver em execução, Android inicia a execução do serviço e chama o método `onCreate()` seguido pelo `onStart()`. Se o componente já estiver em execução, o método `onStart()` é invocado novamente, diferentemente do componente *activity* o componente *service* pode chamar o método `onStart()` repetidas vezes;

b) `onResume()`, `onPause()` e `onStop()`: esses métodos não são úteis ao componente *service*, porque ele não possui interface com usuário, e estará sempre rodando em segundo plano;

c) `onBind()`: caso seja necessário se conectar a um serviço, pode-se chamar o método `Contect.bindService()`. Este método cria um serviço se ele não estiver em execução, e chama o método `onCreate()`, mas não chama o método `onStart()`;

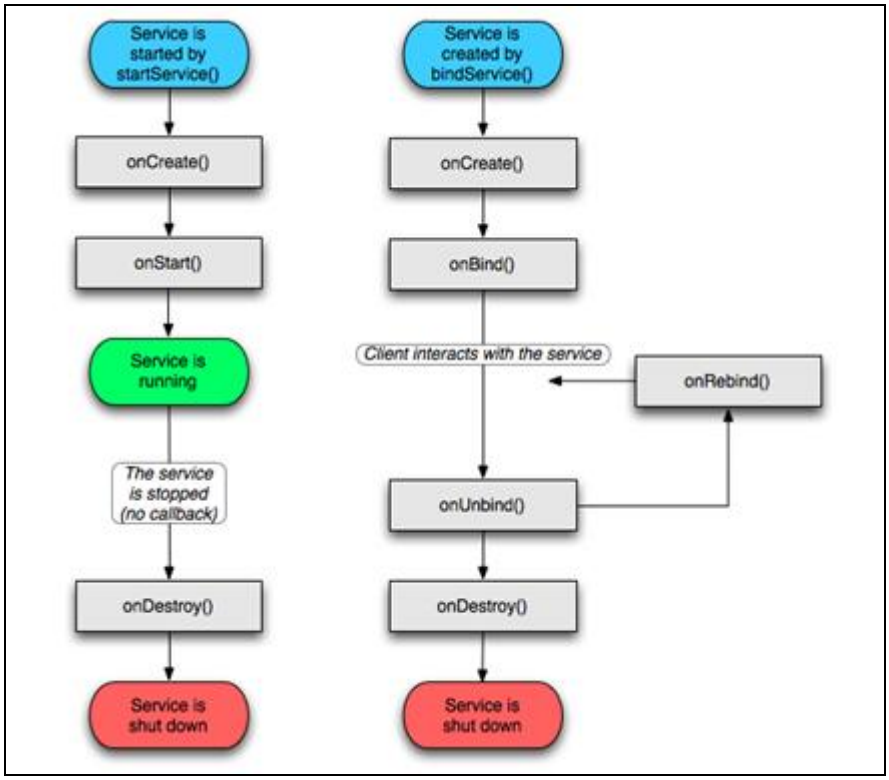
d) `onDestroy()`: Android chama o método `onDestroy()` quando quer destruir o componente, pois não existem mais clientes conectados a ele, ou porque a memória do *smartphone* está muito baixa.

A Figura 4 mostra dois ciclos de vida do componente *service*, pois existem duas formas de se iniciar um serviço, pelos métodos `startService()` e `bindService()`:

a) `startService()`: método utilizado para iniciar um serviço, que ficará executando por um tempo indeterminado, até que o método `stopService()` ou `stopSelf()` seja chamado explicitamente;

b) `bindService()`: método pode iniciar um serviço se ele ainda não está executando, ou pode também conectar-se a um serviço já em execução, que é o principal objetivo deste método.

A diferença entre iniciar o serviço através do método `bindSerice()` é que o processo do serviço estará vinculado a quem se conectou a ele, por exemplo, uma *activity*. Então se a *activity* for destruída, o serviço também será. O mesmo não acontece quando o cliente quer iniciar o processo chamando o método `startService()`, pois este ficará executando por tempo indeterminado até que alguém chame o método `stopService()`, o que permite que o serviço fique em execução mesmo depois que a aplicação termine (figura 4).



Fonte: Tistory (2011).

Figura 4 – Ciclo de vida do componente *service*

2.4 SHELL LINUX

Shell é um interpretador de comandos. Realiza a leitura de comandos escritos por um teclado ou por um arquivo *shell script* e interpreta. Resumindo, é uma maneira de efetuar uma interface entre o usuário e o sistema operacional. *Shell Script* é o conjunto de comandos escritos em uma linguagem *script*, que permite, por exemplo, o uso de variáveis e estruturas de repetição, facilitando a escrita de programas mais complexos. A Figura 5 mostra como o *shell* Bash interpreta os comandos ou arquivos *shell script*:

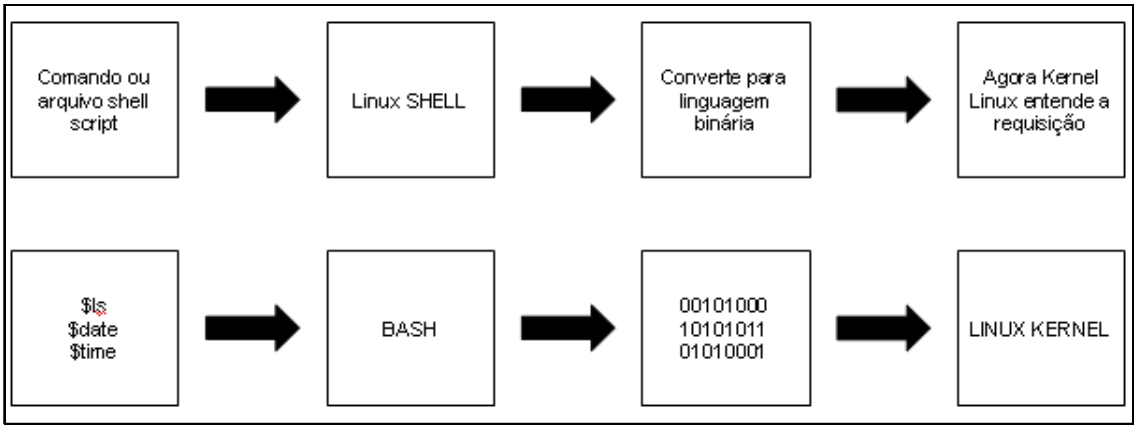


Figura 5 – Como o *shell* interpreta comandos

Bash *(Bourne-Again Shell)* é o *shell* encontrado na maioria das distribuições Linux, desenvolvido por Brian Fox e Chet Ramey.

De acordo com Jargas (2009) é possível realizar diversas tarefas com *shell*, tais como:

a) rotinas de cálculos;

b) instaladores de software;

c) manipulação de banco de dados;

d) *Common Gateway Interface* (CGI);

e) rotinas de *backup*.

2.5 TRABALHOS CORRELATOS

Algumas ferramentas desempenham papel semelhante ao proposto neste trabalho. Dentre elas, foram selecionadas: ResMo (MATOS, 2009), Ignition (LOGMEIN, 2011) e PhoneMyPc (SOFTWAREFORME, 2010).

2.5.1 ResMo

Segundo Matos (2009), ResMo é um sistema cliente/servidor desenvolvido com o objetivo de ajudar as pessoas que trabalham na gerência de redes de computadores Linux,

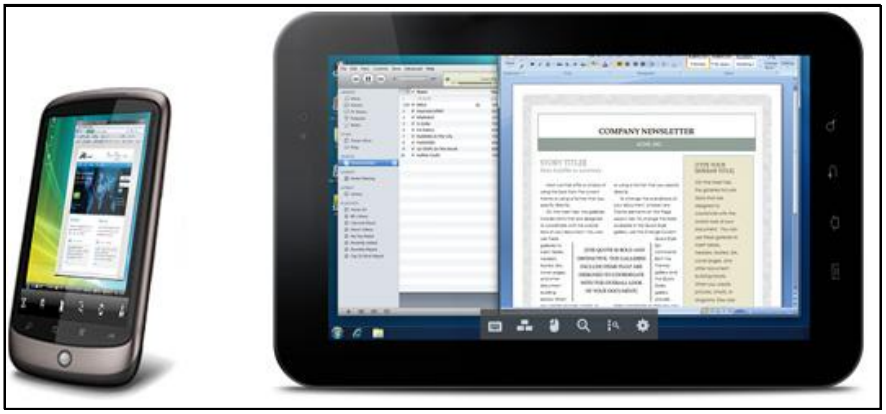
usando a *Java Micro Edition* (JME) com os protocolos de conexão GPRS/EDGE[1], sendo possível realizar consultas e enviar comandos para serem executadas no servidor. A grande vantagem do sistema é que problemas de redes podem ser diagnosticados e resolvidos rapidamente, como por exemplo, a parada de serviços ou afogamento de processador ou memória. Além disso, pode-se prever com um planejamento de falhas, a implementação de *scripts* de solução para prováveis erros, controlando assim as falhas do servidor.

2.5.2 Ignition

O sistema Ignition oferece acesso remoto entre aparelhos móveis e computadores *desktop*. Através de uma aplicação Android, pode-se visualizar o computador *desktop* no *smartphone*. De acordo com a empresa LogMeIn (2011) este sistema oferece as seguintes funcionalidades:

a) acesso ao computador pessoal: acessar a área de trabalho, arquivos e aplicativos remotos e os recursos da rede;
b) acesso a área de trabalho do computador, podendo mover o mouse, assim como escutar o som reproduzido no computador pessoal;
c) controle de dispositivos em rede virtual: ligar dispositivos em redes virtuais, de ponto a ponto;
d) disponível para Android e Iphone.

A Figura 6 mostra o aplicativo Ignition em execução no *smartphone* e *tablet*.



Fonte: LogMeIn (2011).

Figura 6 – Aplicação Ignition para Android

1 *General Packet Radio Service/Enhanced Data for Global Evolution*

2.5.3 PhoneMyPC

O sistema PhoneMyPC é um sistema que, segundo SoftwareForMe (2011), oferece acesso remoto entre dispositivos móveis com o sistema operacional Android e computadores com o sistema operacional Windows.

Os recursos oferecidos pelo sistema são:

a) controle do mouse e teclado: controla-se o mouse e teclado do computador através de comandos enviados pelo *smartphone*;
b) imagens de alta qualidade da tela do computador: consegue-se ver no celular a atual tela do computador;
c) interação em tempo real: interage-se com o computador através da tela do *smartphone*;
d) execução de ações: executa qualquer aplicação do computador que esteja na lista exibida no *smartphone*;
e) controle de processos: lista das aplicações executadas em tempo real no computador, podendo fechar, maximizar/minimizar qualquer aplicação.

A Figura 7 mostra a aplicação PhoneMyPC em execução no *smartphone* equipado com o sistema operacional Android.

Fonte: SoftwareForMe (2011).
Figura 7 – PhoneMyPC em execução no Android

3 DESENVOLVIMENTO

Neste capítulo são abordadas as etapas do desenvolvimento do projeto. São ilustrados os principais requisitos, a especificação, a implementação e por fim são listados os resultados e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos apresentados abaixo se encontram classificados em Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF).

O sistema deverá:

e) permitir que o software cliente navegue entre diretórios e arquivos do sistema operacional no qual o software servidor está instalado (RF);
f) permitir que o software cliente envie comandos para o software servidor (RF);
g) permitir que o software cliente receba arquivos do software servidor, contendo o resultado dos comandos executados (RF);
h) permitir que o software servidor realize *downloads* de dados da internet através de comandos enviados pelo software cliente (RF);
i) permitir que o software servidor execute e finalize aplicativos no *desktop* através de comandos enviados pelo software cliente (RF);
j) permitir que o software cliente visualize os aplicativos em execução no sistema operacional onde o software servidor está instalado (RF);
k) oferecer ao software cliente uma lista amigável de serviços que podem ser executados no software servidor (RNF);
l) ser implementado na linguagem Java (RNF);
m) ser implementado usando o ambiente de desenvolvimento Eclipse (RNF);
n) possibilitar a transferência de dados criptografados (RNF);
o) que o software cliente execute no sistema operacional Android (RNF);
p) que o software servidor execute no sistema operacional Linux (RNF).

3.2 ESPECIFICAÇÃO

Para detalhar a especificação do sistema foram utilizados os diagramas de casos de uso, classes e sequência, os quais serão apresentados nas próximas seções.

3.2.1 Diagramas de casos de uso

Na aplicação existem dois cenários. O primeiro é o servidor e o segundo o cliente. A seguir serão apresentados os diagramas de caso de uso do servidor e do cliente, detalhando os principais.

3.2.1.1 Módulo Servidor

A Figura 8 apresenta os casos de uso do servidor, que são executados pelo módulo servidor quando recebem alguma requisição do módulo cliente. Os casos de uso que são descritos: `Receber uma lista de diretórios e arquivos` (quadro 01), `Realizar o download de um arquivo` (quadro 02), `Enviar URL` (quadro 03), `Enviar um comando customizado` (quadro 04), `Visualizar processos em execução` (quadro 05), `Finalizar processo` (quadro 06).

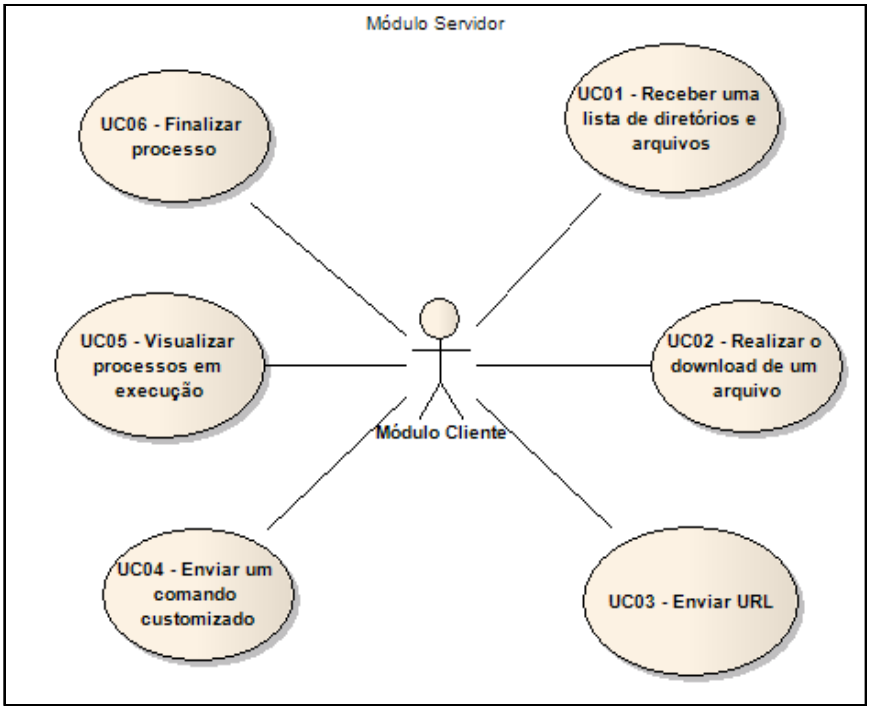


Figura 8 – Diagrama de casos de uso do módulo servidor executados pelo módulo cliente

UC01 – Receber uma lista de diretórios e arquivos	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição de visualizar os diretórios e arquivos. 02) O servidor monta uma lista de todas as pastas e arquivos do diretório requisitado. 03) O servidor envia a lista através da conexão já aberta pelo cliente.
Exceção 01	No passo 03, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Lista das pastas e arquivos enviado.

Quadro 1 – Caso de uso 01

UC02 – Realizar o d*ownload* de um arquivo	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição para enviar um arquivo. 02) O servidor verifica se existe o arquivo. 03) O servidor envia o arquivo requisitado.
Exceção 01	No passo 02, caso o arquivo não exista, é enviada uma mensagem ao módulo cliente avisando que o arquivo não existe mais.
Exceção 02	No passo 03, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Arquivo enviado.

Quadro 2 – Caso de uso 02

UC03 – Enviar URL	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição para realizar o *download* da URL. 02) O servidor se conecta a *Uniform Resource Locator* (URL) passada, e realiza o *download*. 03) O servidor envia uma mensagem ao módulo cliente indicando que o *download* foi iniciado com sucesso.
Exceção 01	No passo 02, caso o servidor não consiga se conectar a URL passada, é enviado uma mensagem de erro ao servidor.
Exceção 02	No passo 03, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Mensagem de sucesso enviada.

Quadro 3 – Caso de uso 03

UC04 – Enviar um comando customizado	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição para executar um comando. 02) O servidor executado o comando passado utilizando o *shell* do Linux. 03) O servidor monta uma lista com todos os resultados recebidos do *shell* do Linux. 04) O servidor envia a lista dos resultados obtidos ao módulo cliente.
Exceção 01	No passo 04, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Lista dos resultados obtidos enviado.

Quadro 4 – Caso de uso 04

UC05 – Visualizar processos em execução	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição para enviar os processos que estão executando no módulo servidor. 02) O servidor executa um arquivo *shell script*, que busca todos os processos em execução. 03) O servidor monta uma lista com todos processos em execução 04) O servidor envia a lista dos processos em execução obtidos ao módulo cliente.
Exceção 01	No passo 04, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Lista dos processos que estão em execução no servidor.

Quadro 5 – Caso de uso 05

UC06 – Finalizar processo	
Pré Condições	Estar com uma conexão aberta com o cliente.
Cenário Principal	01) O módulo servidor recebe uma requisição para finalizar um processo. 02) O servidor executado um comando *shell* para finalizar o processo. 03) O servidor executa um arquivo *shell script* para obter todos os processos em execução. 04) O servidor monta uma lista com todos os processos em execução. 05) O servidor envia uma lista dos processos em execução obtidos para o módulo cliente.
Exceção 01	No passo 05, caso a conexão seja perdida, o caso de uso se encerra.
Pós-Condição	Lista dos processos que estão em execução no servidor.

Quadro 6 – Caso de uso 06

3.2.1.2 Módulo Cliente

A Figura 9 apresenta os casos de uso do módulo cliente.

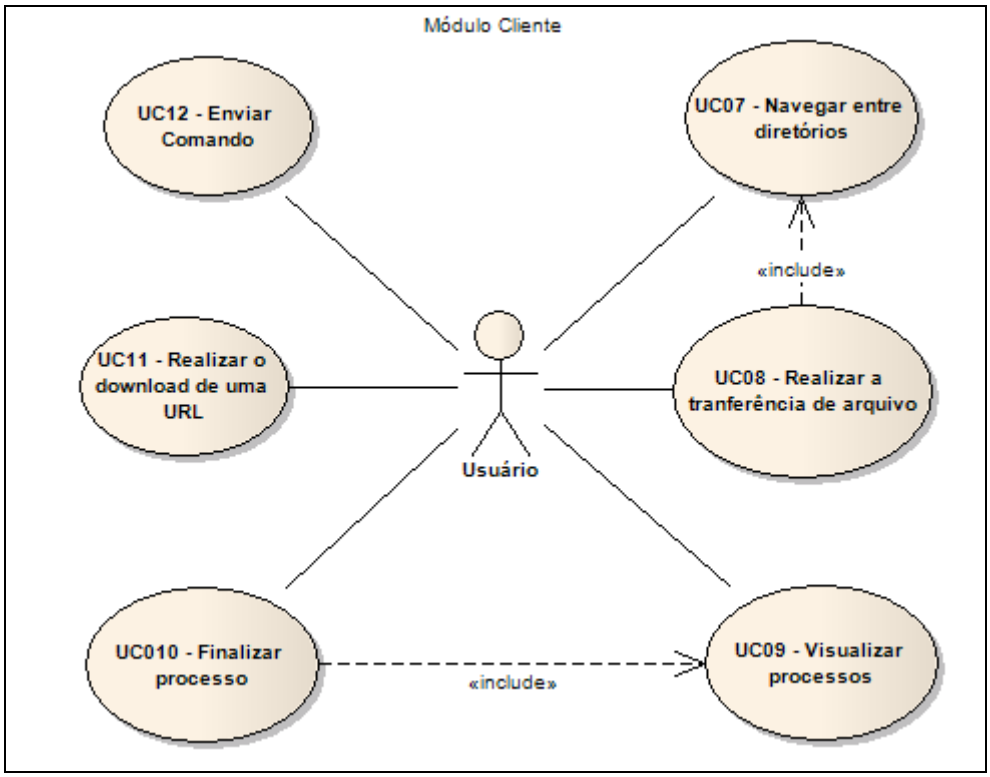


Figura 9 – Diagrama de casos de uso do módulo cliente executados pelo usuário

O caso de uso do módulo cliente possui apenas um ator, que é o usuário do sistema. Detalhando os casos de uso, o UC07 `Navegar entre diretórios` é descrito no quadro 7 apresentando um fluxo alternativo e uma exceção. O UC08 `Realizar a transferência de arquivo` necessita do UC07 para executar, pois é necessário primeiro visualizar o arquivo para poder ser escolhido e transferido, este caso de uso apresenta apenas um fluxo alternativo e uma exceção e é explanado no quadro 8. Caso o usuário decida visualizar os processos que estão ocorrendo no servidor, o caso de uso 09, conhecido como `Visualizar processos`, descreve o processo no quadro 9, apresentando apenas uma exceção. Para finalizar é apresentado o caso de uso 12, chamado de `Enviar comando` no quadro 10.

UC07 – Navegar entre diretórios: visualizar os diretórios e arquivos existentes no servidor.	
Pré-Condições	Cliente deve estar conectado ao servidor.
Cenário Principal	01) Usuário solicita a visualização de diretórios e arquivos. 02) O módulo cliente busca no `properties` de configuração do sistema o diretório inicial, que se deseja visualizar. 03) O módulo cliente envia uma solicitação ao módulo servidor para enviar os diretórios e arquivos do diretório escolhido. 04) O módulo cliente recebe do servidor uma lista contendo o nome dos diretórios e arquivos. 05) O sistema apresenta a lista que contêm o nome dos diretórios e arquivos. 06) Usuário seleciona o diretório que quer visualizar, e retorna ao passo 03.
Fluxo alternativo 01	No passo 06 caso o usuário retorne ao menu, o caso de uso se encerra.
Exceção 01	No passo 03 e 04 caso a conexão seja perdida o caso de uso se encerra.
Pós-Condição	O sistema exibe com sucesso os resultados.

Quadro 7 – Caso de uso 07

UC08 – Realizar a transferência de arquivo	
Pré-Condições	Módulo cliente deve estar conectado ao módulo servidor.
Cenário Principal	01) include::UC07. 02) O usuário seleciona um arquivo. 03) O módulo cliente pergunta ao usuário se ele deseja fazer o *download* do arquivo selecionado. 02) Usuário solicita a transferência do arquivo. 03) O módulo cliente abre uma nova conexão com o módulo servidor 04) O módulo cliente solicita ao módulo servidor a transferência do arquivo. 05) O módulo cliente recebe pedaços do arquivo, e vai escrevendo o arquivo na pasta apropriada. 06) Quando o *download* do arquivo é finalizado, é exibida uma notificação ao usuário e o caso de uso se encerra.
Fluxo alternativo 01	No passo 02 o usuário cancela a transferência do arquivo e retorna ao passo 01.
Exceção 01	No passo 03,04,05 caso a conexão seja perdida o caso de uso se encerra.
Pós-Condição	O módulo cliente recebe o arquivo.

Quadro 8 – Caso de uso 08

UC09 – Visualizar processos	
Pré-Condições	Módulo cliente deve estar conectado ao módulo servidor.
Cenário Principal	01) Usuário solicita a visualização dos processos que estão em execução no servidor. 02) O módulo cliente envia uma solicitação ao módulo servidor de visualização dos processos em execução 03) O módulo cliente recebe uma lista do nome dos processos que estão em execução no servidor. 04) O módulo cliente apresenta a lista dos processos. 05) O caso de uso encerra quando o usuário retorna ao menu.
Exceção 01	No passo 02,03 caso a conexão seja perdida o caso de uso se encerra.
Pós-Condição	O sistema exibe com sucesso os resultados.

Quadro 9 – Caso de uso 09

UC12 – Enviar comando	
Pré-Condições	Módulo cliente deve estar conectado ao módulo servidor.
Cenário Principal	01) Usuário solicita o envio de comandos. 02) O sistema apresenta um campo em branco para a inserção dos comandos. 03) O usuário informa o comando e seleciona a opção de enviar ao servidor. 04) O sistema apresenta o resultado do comando executado no servidor. 05) O usuário clica em "Ok" e retorna ao passo 2. 06) O caso de uso encerra quando o usuário retorna ao menu.
Fluxo alternativo 01	No passo 02,03 e 04 caso o usuário retorne ao menu, o caso de uso se encerra.
Exceção 01	No passo 02,03 e 04 caso a conexão seja perdida o caso de uso se encerra.
Pós-Condição	O sistema exibe com sucesso os resultados.

Quadro 10 – Caso de uso 12

3.2.2 Diagrama de classes

O diagrama de classes fornece uma visão de como as classes estão estruturadas e relacionadas. De forma a facilitar a estruturação e a relação entre elas, são apresentados dois diagramas de classes no módulo servidor e dois diagramas de classes no módulo cliente.

3.2.2.1 Módulo servidor

O módulo servidor apresenta várias classes divididas em quatro pacotes, que são explanados no quadro 11.

Definição dos pacotes do módulo servidor	
Pacote	**Definição**
`br.com.mobilecommand`	define a classe `Main`, que é onde se inicia duas *threads* do pacote `br.com.mobilecommand.connection`
`br.com.mobilecommand.connection`	este pacote contém as classes responsáveis por criar conexões seguras e inseguras
`br.com.mobilecommand.helper`	este pacote contém as classes que são responsáveis por gerenciar as requisições feitas pelo módulo cliente assim como executar e enviar o resultado para o módulo cliente
`br.com.mobilecommand.model`	possui duas classes, que são usadas remotamente para troca de mensagens entre os dois módulos

Quadro 11 – Pacotes do módulo servidor

São explanados a partir de agora três dos quatro pacotes existentes, o pacote `br.com.mobilecommand` possui apenas a classe `Main` que é simples e já foi explicada no

quadro acima.

A figura 10 ilustra as classes do pacote `br.com.mobilecommand.connection`.

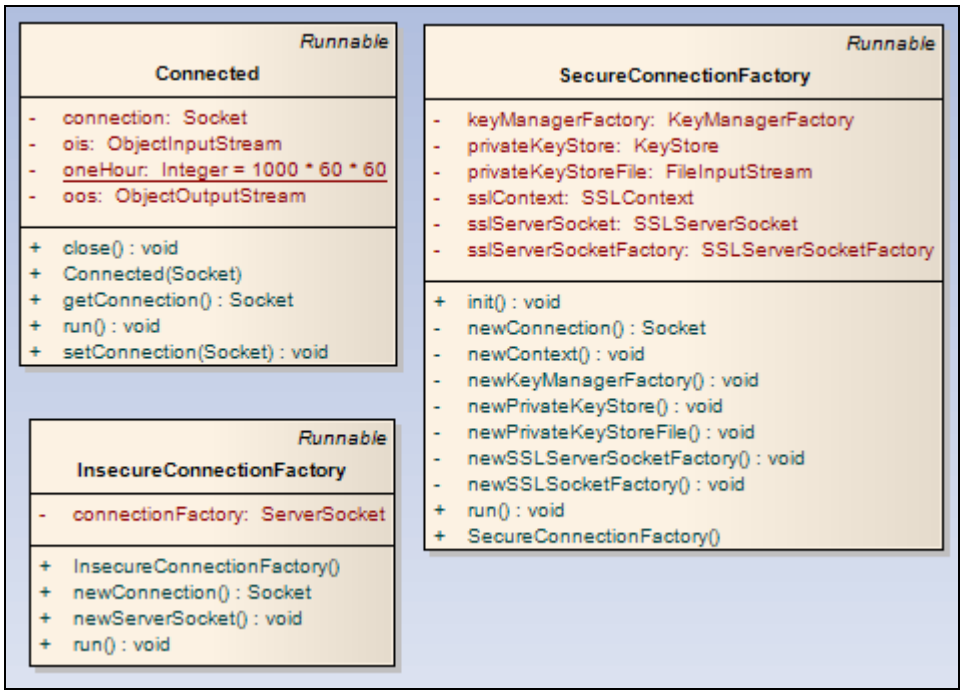


Figura 10 – Pacote `br.com.mobilecommand.connection`

A classe `Connected`, implementa `Runnable`, recebe em seu construtor uma instância de `Socket` que é a conexão já estabelecida com o cliente, e fica responsável por aguardar requisições feitas pelo módulo cliente, assim como fechar a conexão caso ocorra algum erro ou *time out*. Quadro 12 mostra os principais métodos da classe `Connected`.

Connected	
método	**Descrição**
`getConnection`	retorna o `Socket` que contém a conexão aberta com o cliente
`close`	fecha a conexão com o cliente
`run`	é o principal método da classe, responsável por aguardar requisições do módulo cliente e trata-las

Quadro 12 – Principais métodos da classe `Connected`

A classe `SecureConnectionFactory`, que implementa `Runnable`, é uma fábrica de conexões seguras. Ela é responsável por criar um `SSLSocket` e instanciar a classe `Connected`, passando o `Socket` criado, podendo assim iniciar uma nova `Thread` que executará o método `run()` da classe `Connected`.

SecureConnectionFactory	
método	**Descrição**
`newConnection`	escuta requisições para abrir uma nova conexão segura, retornando um `Socket`
`newSSLSocketFactory`	cria a fabrica de `Socket` seguro em determinada porta
`run`	é o principal método da classe, instancia a classe `Connected` passando o `Socket` criado.

Quadro 13 – Principais métodos da classe `SecureConnectionFactory`

A classe `InsecureConnectionFactory`, que implementa `Runnable`, é uma fábrica de conexões inseguras. Ela é responsável por criar um `Socket` e instanciar a classe `Connected`, passando o `Socket` criado, iniciando uma nova `Thread` que executará o método `run()` da classe `Connected`.

InsecureConnectionFactory	
método	**Descrição**
`newConnection`	escuta requisições para abrir uma nova conexão, retornando um `Socket`
`newServerSocket`	cria a fabrica de `Socket` em determinada porta
`run`	é o principal método da classe, instancia a classe `Connected` passando o `Socket` criado.

Quadro 14 – Principais métodos da classe `InsecureConnectionFactory`

Para o tratamento das mensagens recebidas, assim como execução dos comandos remotos enviados, temos o pacote `br.com.mobilecommand.helper` que nos auxilia nestes processos. A figura 11 detalha um pouco mais as classes do pacote `br.com.mobilecommand.helper`.

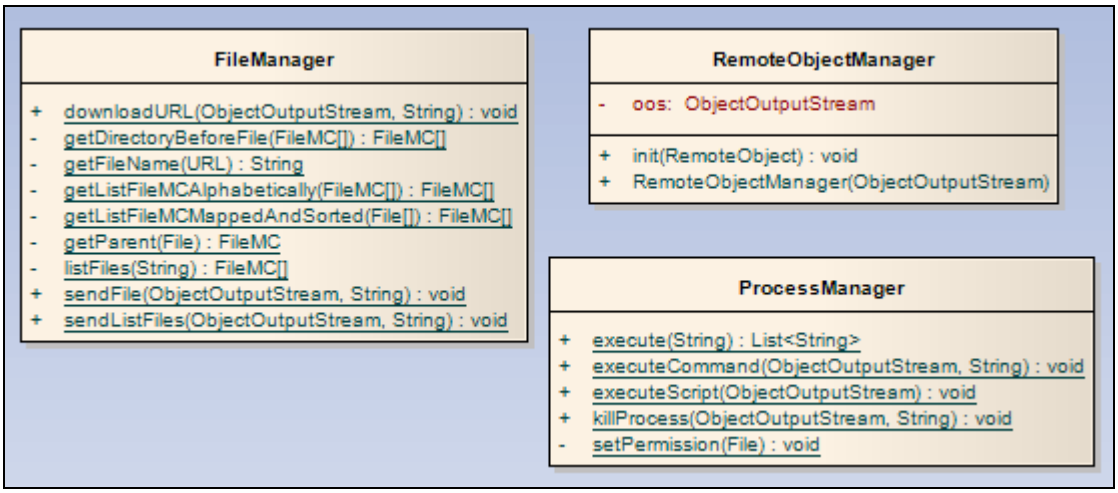


Figura 11 – Pacote `br.com.mobilecommand.helper`

A classe `RemoteObjectManager` é responsável por gerenciar o objeto remoto (`RemoteObject`) que é enviado nas requisições. O método `init()` recebe como parâmetro um `RemoteObject` que contém as váriaveis `type` e `command`. A variável `type` define o tipo de comando que se deseja executar, por exemplo, o tipo pode se referir a visualizar arquivos, ou finalizar algum processo e assim por diante. A variável `command` define o que se deseja executar, por exemplo, caso a variável `type` defina que se deseja visualizar os diretórios e arquivos, a variável `command` irá conter qual o diretório se deseja visualizar.

A classe `FileManager` é responsável por gerenciar todo o tipo de solicitação que se refira a arquivos físicos. As solicitações que desejam visualizar diretórios e arquivos, fazer o *download* de um arquivo físico para o *smartphone* ou que o módulo servidor inicie o

download de uma URL, são todas gerenciadas pela classe `FileManager`. O quadro 15 mostra os principais métodos desta classe.

FileManager	
método	**Descrição**
`sendListFiles`	recebe uma `String` como parâmetro que significa o diretório que se deseja visualizar. Monta um *array* de `FileMC` ordenado que contêm todos os diretórios e arquivos do diretório que se deseja visualizar. Depois do *array* montado envia ao módulo cliente
`sendFile`	recebe uma `String` como parâmetro que significa o arquivo que o módulo cliente deseja fazer o *download*
`downloadURL`	recebe uma `String` como parâmetro que significa a URL que se deseja conectar para fazer o *download* de algum arquivo. O método se conecta a URL passada e realiza o *download* do arquivo, enviando ao módulo cliente uma mensagem que o *download* foi iniciado com sucesso

Quadro 15 – Principais métodos da classe `FileManager`

A classe `ProcessManager` é responsável por gerenciar as solicitações referentes a visualização e finalização de processos que estão em execução no servidor. Ela também é responsável por executar os comandos customizados enviados pelo módulo cliente. O quadro 16 explica em detalhes os principais métodos desta classe.

ProcessManager	
método	**Descrição**
`executeScript`	quando o módulo cliente deseja visualizar os processos que estão em execução no servidor, o módulo servidor executa esse *script* escrito em *shell script*, que retorna todos os processos em execução. O módulo servidor monta um *array* de `String` contendo o nome do processo e envia ao módulo cliente
`executeCommand`	recebe uma `String` como parâmetro que significa o comando que se deseja executar no *shell* do Linux. O módulo servidor executa o comando no *shell* e monta um *array* de `String` que é enviado ao módulo cliente
`killProcess`	recebe uma `String` como parâmetro que significa o nome do processo que se deseja matar. O método se utilizando de um comando *shell* Linux mata o processo e executa o método `executeScript()` que enviará ao módulo cliente os processos que estão em execução.

Quadro 16 – Principais métodos da classe `ProcessManager`

3.2.2.2 Módulo cliente

O módulo cliente apresenta várias classes dividas entre nove pacotes, que são apresentados e explanados no quadro 17.

Definição dos pacotes do módulo cliente	
Pacote	**Definição**
`br.com.mobilecommand`	define a classe `Main` da aplicação
`br.com.mobilecommand.activity`	este pacote contém as classes responsáveis por interagir com o usuário, como definir a tela que será exibida ao usuário assim como controlar os eventos que ocorram.
`br.com.mobilecommand.adapter`	este pacote contém as classes que auxiliam no processo de exibir uma lista de itens ao usuário
`br.com.mobilecommand.connection`	este pacote contêm as classes responsáveis por criar conexões seguras e inseguras
`br.com.mobilecommand.exception`	contêm duas classes responsáveis por lançar erros, caso algo não saia como o desejado
`br.com.mobilecommand.helper`	define a classe `Conf`, que lê as configurações escritas do arquivo `conf.properties`
`br.com.mobilecommand.manager`	contêm as classes que são responsáveis por enviar as requisições feitas pelo usuário ao módulo servidor e tratar o que recebe do módulo servidor
`br.com.mobilecommand.service`	contêm uma classe e uma interface que são responsáveis por iniciar um serviço em segundo plano, podendo assim fazer o download de um arquivo e ao mesmo tempo poder atender uma chamada telefônica, por exemplo.
`br.com.mobilecommand.model`	contêm as classes `FileMC` e `RemoteObject`, que são os objetos que trafegam na rede.

Quadro 17 – Pacotes do módulo servidor

As classes do pacote `br.com.mobilecommand.activity` são apresentados na figura 12.

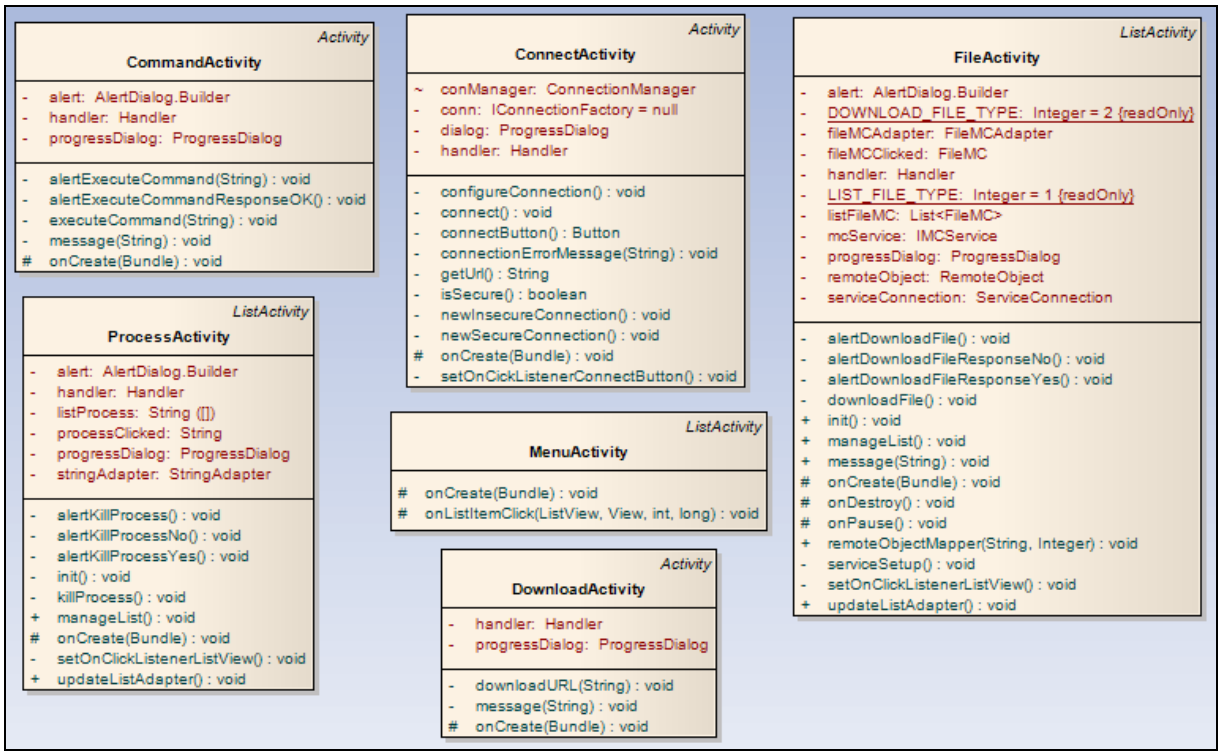


Figura 12- Pacote `br.com.mobilecommand.activity`

Cada classe é responsável por uma única tela. A classe `ConnectActivity` é

responsável por obter os dados de conexão do servidor, e após a conexão ser concluída com sucesso, redirecionar o usuário para a tela do menu.

ConnectActivity	
método	**Descrição**
`connect`	inicia um nova `Thread`, que verifica se o usuário quer um conexão segura ou não, e instância uma fabrica de `Socket`, passando a URL de conexão com o servidor. Após o `Socket` criado e a conexão estabelecida é chamado o método `configureConnection()`
`configureConnection`	instancia a classe `ConnectionManager` passando o `Socket` criado no método `connect`, que ficará responsável por manter apenas uma conexão aberta. Após isso, o usuário é redirecionado a tela do menu.

Quadro 18 – Principais métodos da classe `ConnectActivity`

A classe `MenuActivity` é responsável pela tela de menu, que contêm as opções de comandos que podem ser executados no servidor, como é uma classe relativamente simples, que apenas redireciona o usuário a tela solicitada, não é explanada em maiores detalhes.

A classe `FileActivity` disponibiliza ao usuário a visualização dos diretórios e arquivos do servidor, assim como a opção de poder fazer o *download* do arquivo se assim desejado. O quadro 19 fornece maiores informações sobre os principais métodos desta classe.

FileActivity	
método	**Descrição**
`manageList`	começa exibindo ao usuário um alerta de aguarde, para então iniciar uma nova `Thread` que fica responsável em chamar o método `getDiretorios()` da classe `FileMCManager`, que irá buscar no servidor os diretórios e arquivos do diretório solicitado. Após receber a lista de diretórios e arquivos chama o método `updateListAdapter()`
`updateListAdapter`	inicia uma nova `Thread` que limpa a tela de itens exibidos ao usuário e mostra os novos itens (diretórios e arquivos) que foram solicitados pelo usuário, finalizando o alerta de aguarde
`downloadFile`	inicia uma nova `Thread` solicitando que o serviço já criado inicie o *download* de um arquivo lhe passando uma fabrica de `Socket`, que será necessário para abrir uma nova conexão com o servidor.
`serviceSetup`	inicia um serviço ou se conecta ao serviço caso ele já tenho sido iniciado, que será responsável por fazer *download* de um arquivo em segundo plano.

Quadro 19 – Principais métodos da classe `FileActivity`

A classe `ProcessActivity`, é responsável por apresentar os processos que estão em execução no servidor assim como disponibilizar a opção ao usuário de poder finaliza-los. O quadro 20 apresenta os principais métodos desta classe.

ProcessActivity	
método	**Descrição**
manageList	Inicia exibindo ao usuário um alerta de aguarde, para então iniciar uma nova Thread que fica responsável em chamar o método getProcessRunning() da classe ProcessManager, que irá buscar no servidor os processos em execução. Após receber a lista de dos processos chama o método updateListAdapter()
updateListAdapter	limpa a tela de itens exibidos ao usuário e mostra os novos itens (processos) que foram solicitados pelo usuário, finalizando o alerta de aguarde.
killProcess	um alerta de aguarde é exibido ao usuário, logo após é iniciado uma nova Thread que chamará o método killProcess() da classe ProcessManager, passando o processo que foi clicado na lista de itens exibida e que se deseja finalizar, após a execução o mesmo método retornará uma lista atualizada dos processos em execução, chamando assim o método updateListAdapter()

Quadro 20 – Principais métodos da classe ProcessActivity

O pacote br.com.mobilecommand.connection contêm uma interface e duas classes, no qual implementam esta interface. A interface IConnectionFactory foi criada para abstrair a criação de uma conexão segura e insegura, que é implementada pelas classes SecureConnection e InsecureConnection respectivamente.

As classes do pacote br.com.mobilecommand.manager gerenciam e tratam as requisições que são realizadas ao módulo servidor, assim como as respostas que recebe. A figura 13 apresenta as classes do pacote br.com.mobilecommand.manager.

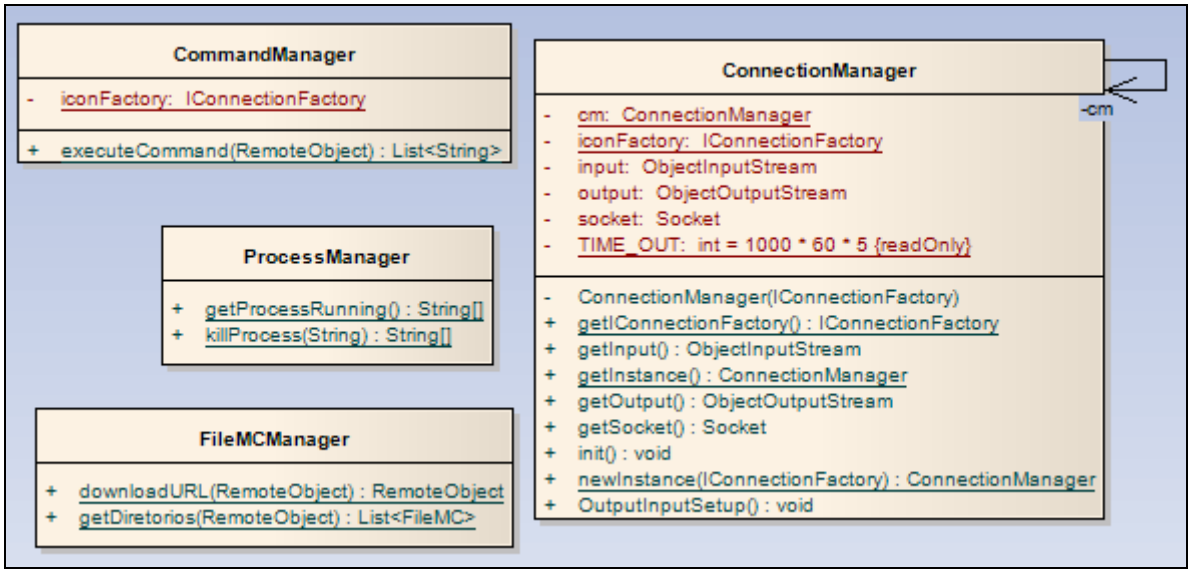


Figura 13 – Classes o pacote br.com.mobilecommand.manager

A classe ConnectionManager é responsável por abrir uma conexão com o servidor, assim como manter apenas uma conexão aberta, possui um construtor privado e só pode ser instanciada se chamado o método newInstance(). Seus principais métodos são apresentados no quadro 21.

ConnectionManager	
método	**Descrição**
`newInstance`	método `static` que instancia `ConnectionManager` passando `IConnectionFactory`, que é a interface que o `SecureConnectionFactory` e `InsecureConnectionFactory` implementam
`getIConnectionFactory`	método `static` que retorna `IConnectionFactory` que foi passada na hora da chamada do método `newInstance()`
`init`	chama o método `getConnection()` da instância passada no método `newInstance()` que implementa `IConnectionFactory`, estabelecendo assim apenas uma conexão aberta
`getOutput`	retorna uma instância de `ObjectOutputStream`, necessária para escrever um objeto para o servidor
`getInput`	retorna uma instância de `ObjectInputStream`, necessária para ler um objeto que venha do servidor
`getInstance`	método `static` que retorna a instância de `ConnectionManager` criada no método `newInstance()`

Quadro 21 – Principais métodos da classe `ConnectionManager`

A classe `CommandManager` gerência as requisições de envio de comando, a classe `ProcessManager` tem como objetivo buscar os processos em execução no módulo servidor, assim como matar o processo.

CommandManager	
método	**Descrição**
`getProcessRunning`	método `static` que retorna um *array* de `String` contendo o nome dos processos que estão rodando no servidor
`killProcess`	método `static` que solicita ao servidor que finalize um processo e retorna um *array* de `String` atualizado com o nome dos processos que estão rodando no servidor.

Quadro 22 – Principais métodos da classe `CommandManager`

A classe `FileMCManager` fica responsável por gerenciar requisições de visualizar diretórios e arquivos assim como o envio de URL para o módulo servidor realizar *download*.

FileMCManager	
método	**Descrição**
`getDiretorios`	método `static` que envia uma solicitação ao módulo servidor de visualizar diretórios e arquivos existentes no diretório passado. Recebe como resposta do servidor uma lista `FileMC` que representa os diretórios e arquivos solicitados
`downloadURL`	método `static` que envia uma solicitação ao módulo servidor requisitando que ele se conecte e inicie o *download* da URL passada

Quadro 23 – Principais métodos da classe `FileMCManager`

O pacote `br.com.mobilecommand.service` contém uma interface e uma classe. A classe `MCService` estende o componente `Service` do Android, ela é executada em segundo plano, para que quando o usuário deseje realizar o *download* de algum arquivo do servidor

por exemplo, a transmissão do arquivo não seja perdida caso ele tenha que atender uma ligação telefônica.

MCService	
método	**Descrição**
`downloadFile`	método responsável por receber uma fabrica de `Socket` e abrir uma nova conexão com o servidor, com a conexão já estabelecida chama o método `getFile()` para fazer o *download* propriamente dito do arquivo e chama o método `notification()` para notificar o usuário que o arquivo foi finalizado com sucesso e o método `stopSelf()` do componente `Service`, para destruir o serviço.
`getFile`	método responsável por fazer o *download* e escrever no cartão de memória o arquivo físico
`notification`	alerta o usuário que o *download* do arquivo foi finalizado com sucesso

Quadro 24 – Principais métodos da classe `MCService`

3.2.3 Diagrama de sequência

Esta seção apresenta dois diagramas de sequência que representam o conjunto de passos que o programa executa para realizar determinada tarefa, com base nas ações do usuário. O primeiro diagrama de sequência ilustra o que ocorre no módulo servidor quando o módulo cliente envia uma requisição para executar um comando de *shell* escrito pelo usuário.

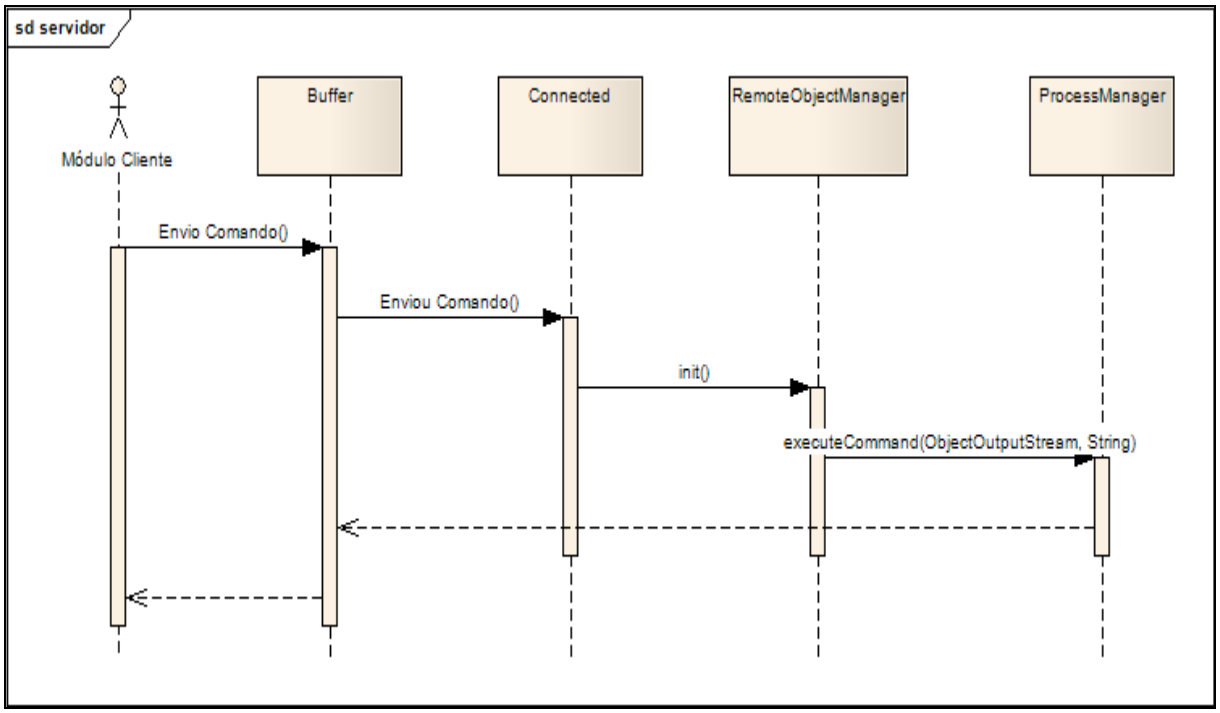


Figura 14 – Diagrama de seqüência do envio de comando *shell* pelo módulo cliente

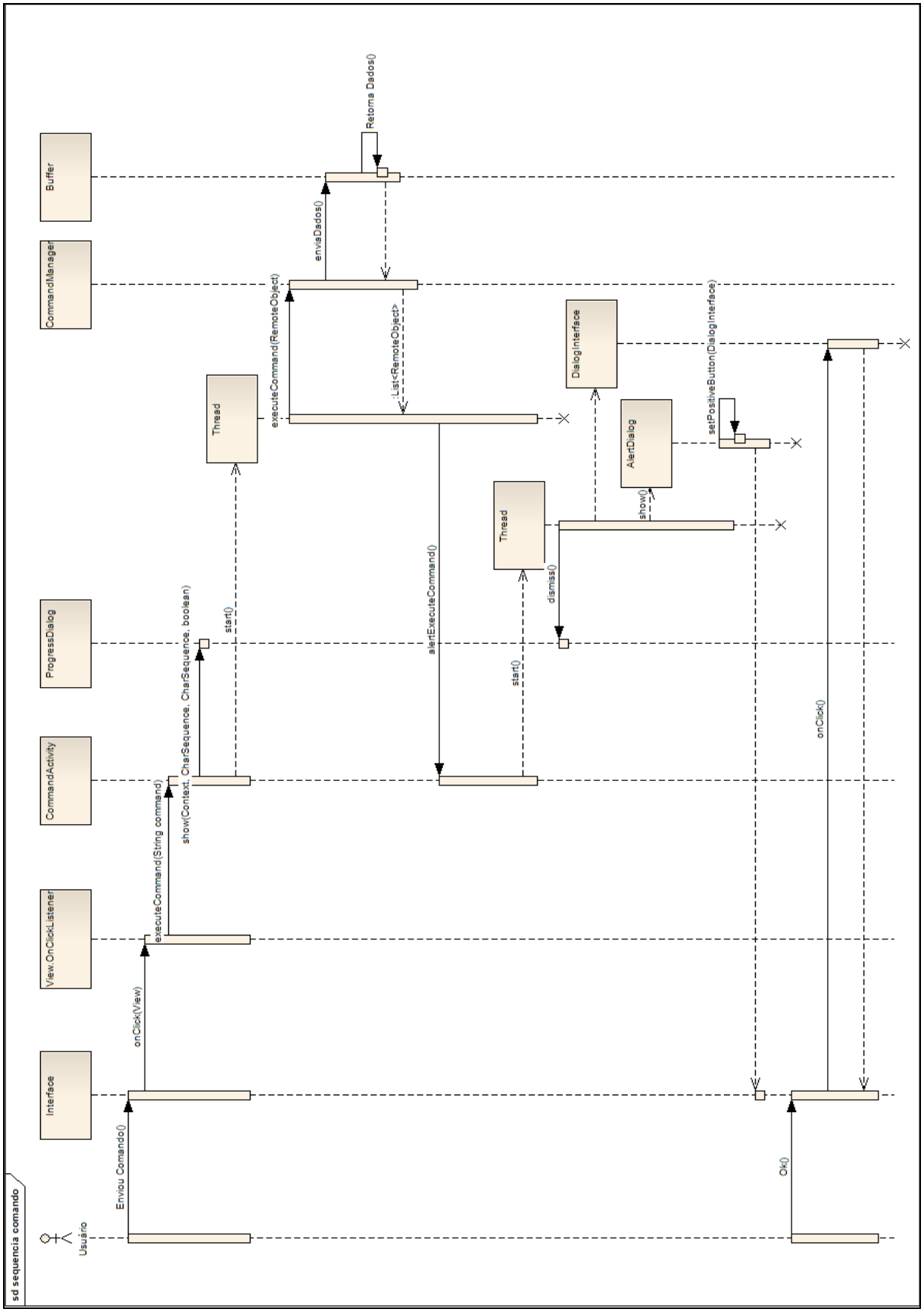


Figura 15 – Diagrama de sequência do envio de comando *shell* ao módulo servidor

A figura 15 ilustra o envio de comando *shell* pelo usuário ao servidor.

3.3 IMPLEMENTAÇÃO

A seguir são apresentadas informações sobre as técnicas e ferramentas utilizadas para a implementação da aplicação.

3.3.1 Técnicas e ferramentas utilizadas

A aplicação Mobile Command foi implementada sob o paradigma da orientação a objetos, usando a linguagem Java na versão 6, assim como o ambiente de desenvolvimento Eclipse IDE. Em conjunto com os recursos da linguagem, foi utilizado o kit de desenvolvimento que o Android disponibiliza, na versão 2.3.1.

Optou-se por utilizar juntamente com o ambiente de desenvolvimento Eclipse IDE, um *plugin* conhecido como *Android Development Tools* (ADT), disponibilizado pela GOOGLE.

3.3.1.1 Android SDK e plugin ADT

O Android SDK disponibiliza várias ferramentas para facilitar o desenvolvimento de aplicações Android. E essas ferramentas podem ser acessadas através de um *plugin* ADT, e está disponível para o ambiente de desenvolvimento Eclipse IDE, de forma totalmente integrada, facilitando a criação, execução, depuração e exportação de aplicações Android.

Os principais componentes que o Android SDK disponibiliza são:

a) *Framework* de aplicação;
b) Ferramentas, como por exemplo, emulador;
c) Aplicações exemplo.

3.3.1.1.1 Emulador Android

O emulador permite testar aplicações Android sem o uso de um aparelho físico. Ele imita quase todo tipo de recurso que um aparelho físico possui. Usando o mouse e o teclado

do computador, pode-se pressionar a tela do emulador e gerar eventos para navegar, iniciar aplicativos, etc. Para facilitar o teste de aplicações em diferentes tipos de aparelhos, o emulador utiliza *Android Virtual Device* (AVD), que permite definir algumas configurações de hardware. Uma vez a aplicação rodando no emulador, é possível utilizar o componente `Service` para se comunicar com outras aplicações que estão instaladas no emulador, pode-se também reproduzir música ou vídeo, assim como escrever e recuperar dados. Outra facilidade é o depurador, que possibilita, por exemplo, simular interrupções, como a chegada de SMS[2], recebimento de telefonemas ou mesmo visualizar o que está ocorrendo no *kernel* do sistema operacional. O emulador é apresentado na figura 16 e algumas funções no quadro 25.

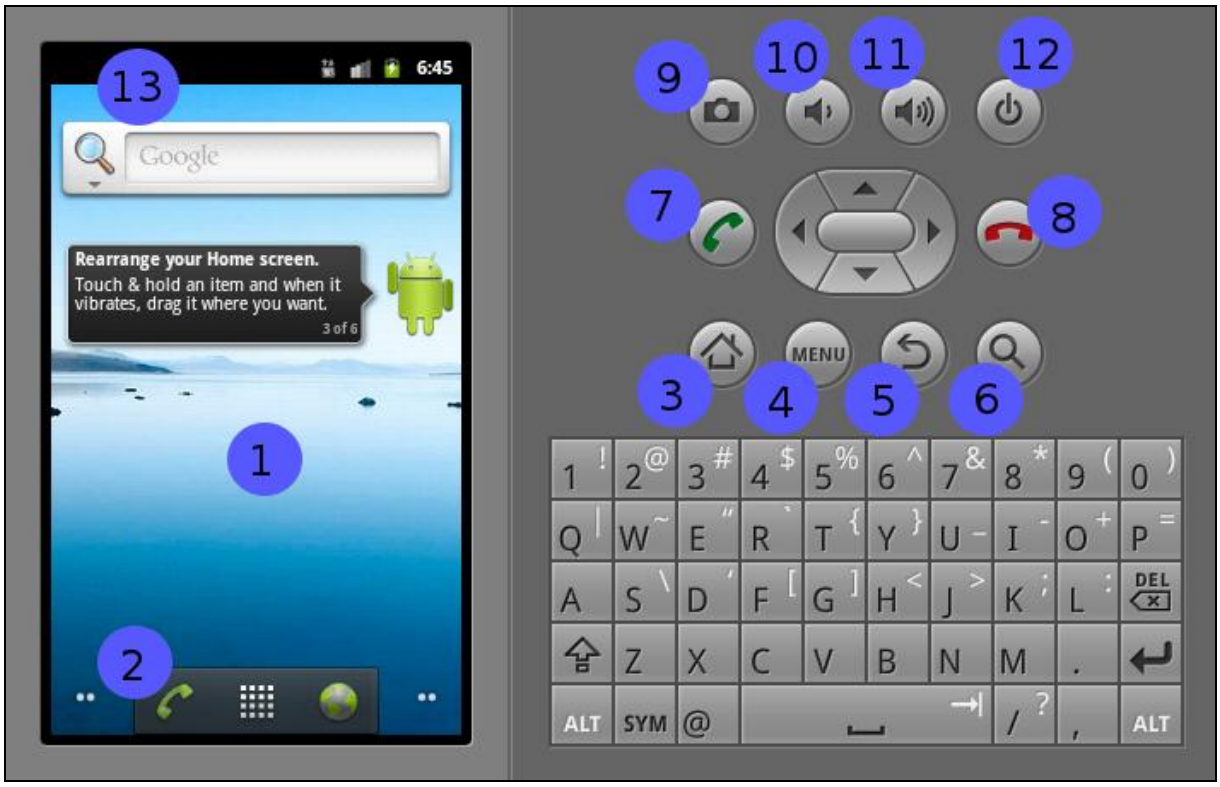


Figura 16 – Emulador Android

2 *Short Message Service*

Número	Descrição
1	Tela inicial
2	Menu
3	Botão *Home*, direciona para a tela inicial
4	Apresenta menu
5	Retorna
6	Pesquisar
7	Receber chamada telefônica
8	Desligar chamada telefônica
9	Tirar foto
10	Abaixar volume
11	Aumentar volume
12	Desligar
13	Notificações

Quadro 25 – Teclado de funções do emulador

Pode-se usar algumas teclas do teclado do computador para controlar comportamento e o ambiente que o Android disponibiliza. O quadro 26 resume o uso das funções mais comuns:

Teclas emulador	Teclas do teclado
Home	HOME
Menu	F2 ou botão *Page Up*
Asterisco	*Shift*-F2 ou *Page Down*
Voltar	ESC
Botão para ligar	F3
Botão para finalizar ligação	F4
Procurar	F5
Botão ligar	F7
Botão para aumentar volume	Ctrl-5
Botão para diminuir volume	Ctrl-F6
Botão câmera	Ctrl-F3

Quadro 26 – Teclas comuns utilizadas no emulador

O emulador, na versão r10, possui algumas limitações, como:

a) não possibilita o fazer e receber chamadas telefônicas, apenas simular;

b) não suporta conexões USB[3];

c) não suporta a captura de imagem ou vídeo;

d) não suporta simulação com fones de ouvido;

e) não possibilita simular o uso da bateria;

f) não suporta *Bluetooth*.

3 *Universal Serial Bus*

3.3.1.2 Módulo servidor: gerenciamento de conexão

A aplicação do módulo servidor disponibiliza ao módulo cliente dois tipos de conexões possíveis, conexão segura e insegura. Para disponibilizar esses dois tipos de conexão o módulo servidor executa duas `Thread`, uma `Thread` da classe `SecureConnectionFactory` que escuta requisições para a abertura de conexões seguras, e a outra `Thread` da classe `InsecureConnectionFactory` que escuta requisições de abertura de conexão insegura. As duas `Thread` são iniciadas no método `main()` da classe `Main` do pacote `br.com.mobilecommand`. O quadro 27 mostra o método `main()` do módulo servidor.

```
public static void main(String[] args) {
		Thread secure = new Thread(new SecureConnectionFactory());
		Thread insecure = new Thread(new InsecureConnectionFactory());
		secure.start();
		insecure.start();
}
```

Quadro 27 – Método `main()` da classe `Main` do módulo servidor

Os métodos `run()` das duas classes são iguais, pois realiza a mesma operação quando uma nova conexão é aberta. A operação como mostra o quadro 28 é escutar requisições de abertura de conexão enquanto a aplicação estiver executando, com a conexão já estabelecida, é instanciada a classe `Connected` passando o `Socket` que é retornado pelo método `newConnection()`, ilustrado no quadro 28, e iniciar uma nova `Thread` da classe `Connected`.

```
public void run() {
		while(true){
			Connected connected = new Connected(newConnection());
			new Thread(connected).start();
		}
}
```

Quadro 28 – Método `run()` das classes `SecureConnectionFactory e InsecureConnectionFactory`

O método `newConnection()` da classe `InsecureConnectionFactory`, retorna um `Socket` criado pela classe `ServerSocket`, disponibilizado pela API [4]do Java, que implementa `Socket` para servidor.

```
public Socket newConnection(){
		Socket connection = null;
		try {
			connection = connectionFactory.accept();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return connection;
}
```

Quadro 29 – Método `newConnection()` da classe `InsecureConnectionFactory`

A classe `SecureConnectionFactory`, implementa o método `newConnection()` de

forma semelhando a implementação do quadro 29, só que, quem cria o `Socket` é a classe `SSLServerSocket`, como mostrado no quadro 30.

```
private Socket newConnection(){
		Socket connection = null;
		try {
			connection =  this.sslServerSocket.accept();
		} catch (IOException e) {
			e.printStackTrace();
		}
		return connection;
}
```

Quadro 30 – Método `newConnection()` da classe `SecureConnectionFactory`

3.3.1.3 Módulo servidor: gerenciamento de requisições

Após a conexão estabelecida, e a `Thread Connected` é que fica responsável por gerenciar as requisições enviadas pelo módulo cliente. A classe `Connected` foi escrita implementando a classe `Runnable`, para assim, o módulo servidor aceitar conexões simultâneas. O método `run()` desta classe é apresentado no quadro 31, e basicamente espera por `RemoteObject` que será enviado pelo módulo cliente. Após o objeto recebido, a classe `RemoteObjectManager` é que vai identificar o tipo de comando, e qual o comando que se deseja executar no servidor. O tipo de comando pode ser exemplificado como, visualizar diretório e arquivos, e o comando significa qual é o diretório que se deseja visualizar.

```
public void run() {
	try {
		RemoteObject remoteObject = (RemoteObject)ois.readObject();
		RemoteObjectManager roManager = new RemoteObjectManager(this.oos);
		while(true){
			roManager.init(remoteObject);
			remoteObject = (RemoteObject)ois.readObject();
		}
	}catch (SocketTimeoutException ste){
		System.out.println("TIME OUT");
		this.close();
	}catch(java.io.EOFException eof){
		System.out.println("Client close connection");
	}catch (IOException e) {
		e.printStackTrace();
	} catch (ClassNotFoundException e) {
		e.printStackTrace();
	}
}
```

Quadro 31 – Método `run`() da classe `Connected`

4 *Application Program Interface*

```
public void init(RemoteObject ro){
             Integer value = ro.getType();
             switch (value) {
                    case DIRECTORY_LIST: {
                           FileManager.sendListFiles(oos, ro.getCommand());
                           break;
                    }
                    case SEND_FILE: {
                           FileManager.sendFile(oos, ro.getCommand());
                           break;
                    }
                    case DOWNLOAD_FILE: {
                           FileManager.downloadURL(oos,ro.getCommand());
                           break;
                    }
                    case PROCESS_RUNNING:{
                           ProcessManager.executeScript(oos);
                           break;
                    }
                    case KILL_PROCESS:{
                           ProcessManager.killProcess(oos,ro.getCommand());
                           break;
                    }
                    case EXECUTE_COMMAND:{
                           ProcessManager.executeCommand(oos,ro.getCommand());
                           break;
                    }
                    default:{
                           break;
                    }
             }
       }
```

Quadro 32 – Método `init()` da classe `RemoteObjectManager`

O método `init()`, apresentado no quadro 32, é responsável por identificar o que o objeto `RemoteObject` deseja executar no servidor.

3.3.1.4 Módulo servidor: enviar lista de diretórios e arquivos

A classe `FileManager` disponibiliza alguns métodos para tratar o envio dos arquivos e diretórios existentes em um diretório no qual o usuário deseja visualizar. O método `sendListFiles()` (quadro 33) com a ajuda do método `listFiles()` (quadro 34) envia ao módulo cliente um *array* da classe `FileMC`, no qual representa os diretórios e arquivos existentes no caminho (diretório) que se desejava visualizar.

```
public static void sendListFiles(ObjectOutputStream oos,String path){
	FileMC[] listFile = listFiles(path);
	try {
		oos.writeObject(listFile);
		oos.flush();
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

Quadro 33 – Método `sendListFiles`() da classe `FileManager`

```
/**
* Lista todos os diretórios e arquivos referente ao caminho passado.
* Os diretórios e arquivos estão ordenados, onde os diretórios são os primeiros
* da do array em ordem alfabética.
*/
private static FileMC[] listFiles(String path){
	File[] listFile = new File(path).listFiles();
	return getListFileMCMappedAndSorted(listFile);
}
```

Quadro 34 – Método `listFiles`() da classe `FileManager`

3.3.1.5 Módulo servidor: executar comandos *shell*

O método `executeCommandShell()` da classe `ProccessManager` é responsável por executar comandos *shell* utilizando duas classes auxiliares `SystemCommandExecutor` e `ThreadedStreamHandler`, desenvolvidas por Alexander (2011). As duas classes abstraem a invocação *shell* do Linux para a execução dos comandos *shell*. O quadro 35 demonstra como é implementado o método `executeCommandShell()`. Primeiro se faz necessário a criação de uma lista no qual indica o *shell* que se deseja invocar, neste caso o *shell* Bash é invocado, em seguida é indicado o comando shell que se deseja executar, passado esta lista para uma instância de `SystemCommandExecutor`, podendo-se assim executar o comando no *shell* do Linux.

```
public static List<String> executeCommandShell(String command){
	List<String> commands = new ArrayList<String>();
	commands.add("/bin/bash");
	commands.add("-c");
	commands.add(command);

	 SystemCommandExecutor commandExecutor = new SystemCommandExecutor(commands);

	try {
		commandExecutor.executeCommand();
	} catch (IOException e) {
		e.printStackTrace();
	} catch (InterruptedException e) {
		e.printStackTrace();
	}

	StringBuilder stdout = commandExecutor.getStandardOutputFromCommand();

	List<String> resultList = new ArrayList<String>();
	resultList.add(stdout.toString());
	return resultList;
}
```

Quadro 35 – Método `executeCommandShell()` da classe `FileManager`

3.3.1.6 Módulo servidor: processos em execução

O método `executeScript()` da classe `ProcessManager` (quadro 36) fica responsável por executar o script `process.sh` (quadro 37) que identifica todos os processos em execução do usuário corrente que está executando o programa módulo servidor. Com o auxilio do método `execute()`, que executa o script propriamente dito (quadro 38), retorna uma lista de `String` contendo todos os processos em execução. Esta lista de processos é enviada ao módulo cliente, finalizando assim o caso de uso.

```
public static void executeScript(ObjectOutputStream oos){
	File script = new File("confs/process.sh");
	setPermission(script);
	List<String> list = execute(script.getAbsolutePath());
	String[] listArray =  list.toArray(new String[0]);
	try {
		oos.writeObject(listArray);
		oos.flush();
	} catch (IOException e) {
		e.printStackTrace();
	}
}
```

Quadro 36 – Método `executeScript()` da classe `ProcessManager`

```
#!/bin/bash
usuario=$USERNAME
pgrep -u $usuario -l | awk '{print $2}'
```

Quadro 37 – Arquivo `process.sh`

```
public static List<String> execute(String command){
	String result= null;
	List<String> resultList = new ArrayList<String>();
	try{
		Runtime rt = Runtime.getRuntime();
		Process p = rt.exec(command);
		BufferedReader br = new BufferedReader(new
InputStreamReader(p.getInputStream()));
		while( (result = br.readLine()) != null){
			resultList.add(result);
		}
		p.waitFor();
	}catch (InterruptedException  e) {
		e.printStackTrace();
	} catch (IOException e) {
		e.printStackTrace();
	}
	return resultList;
}
```

Quadro 38 – Método `execute()` da classe `ProcessManager`

3.3.2 Operacionalidade da implementação

A operacionalidade do sistema será apresentada através da simulação de casos de utilização do sistema, ou seja, serão detalhadas as operações necessárias para a utilização dos recursos do sistema. De uma forma geral a aplicação Mobile Command tem por objetivo disponibilizar acesso ao módulo servidor através de um *smartphone* com o sistema operacional Android.

A figura 17 apresenta a tela inicial do módulo cliente, que é a tela do formulário de conexão. É informado o endereço do servidor e se a conexão com o servidor será feita de forma segura ou não. Após a conexão ser realizada com sucesso, o usuário é redirecionado a tela de menu (figura 18). No quadro 39 é apresentada a descrição de cada componente da tela do formulário de conexão.

Número	**Descrição**
Figura 17 (1)	URL do servidor, no qual o módulo cliente irá se conectar
Figura 17 (2)	Caso marcado, a conexão será feita de modo seguro, utilizando criptografia.
Figura 17 (3)	Botão que dispara a requisição para se conectar ao módulo servidor.

Quadro 39 – Descrição da figura 16

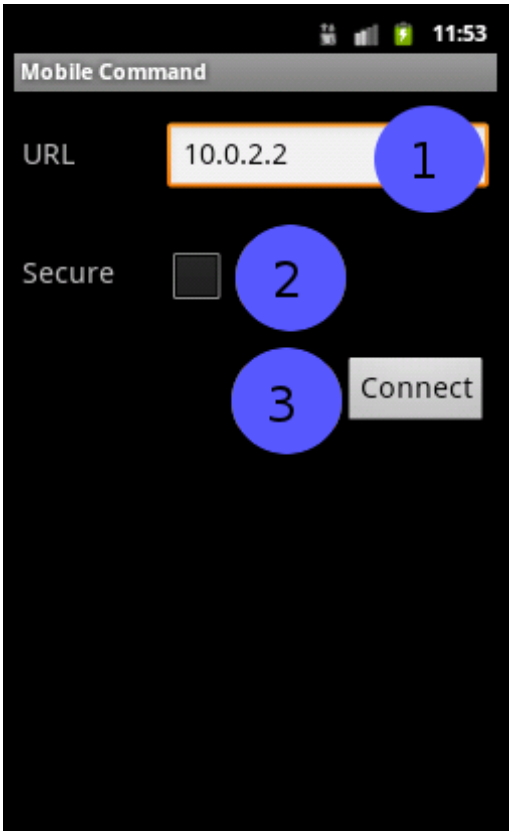


Figura 17 – Formulário de conexão

A tela do menu (figura 18) apresenta algumas opções do que pode ser realizado no módulo servidor.

Menu	**Descrição**
Directory and Files (figura 18(1))	Visualizar os diretórios e arquivos do servidor, e realiza o *download* de arquivos.
URL Download (figura 18(2))	Enviar uma URL ao módulo servidor, para que ele faça *download* dos dados.
Process Running (figura 18(3))	Visualiza os processos que estão rodando no servidor e finaliza os mesmos de acordo com o nome do processo.
Execute Command (figura 18(4))	Executa algum comando no servidor.
Exit (figura 18(5))	Fecha a conexão estabelecida e volta a tela inicial do programa.

Quadro 40 – Descrição da figura 18

Escolhendo a opção *Directory and Files* (figura 18 (1)), será apresentado uma lista contendo todos os diretórios e arquivos do servidor que pertencem ao diretório especificado no arquivo `conf.properties`. A Figura 19 exibe a tela da opção *Directories and Files*.

Figura 18 – Tela do menu

Figura 19 – Tela *Directories and Files*

Opção	Descrição
figura 19(1)	diretório pai do diretório atual que está sendo visualizado
figura 19(2) e figura 19(3)	diretório ou arquivo do diretório atual que está sendo visualizado

Quadro 41 – Descrição da figura 19

Caso seja escolhida a opção dos três pontos (figura 19 (1)), será direcionado ao diretório pai do diretório atual que está sendo visualizado. O mesmo não acontece caso o diretório atual seja `o especificado no arquivo conf.properties`, pois é definido que este é o diretório de maior abrangência que o usuário pode chegar. Os outros itens da lista, que poderão ser diretórios ou arquivos, caso seja escolhido um diretório, a aplicação se encarregará de buscar no servidor as pastas e arquivos que o diretório escolhido possui. Se um arquivo for escolhido, será exibida uma notificação de *download*. Caso a opção escolhida seja fazer o *download* do arquivo (figura 20 (2)), uma nova conexão com o servidor será aberta, de forma totalmente transparente, o *download* do arquivo será realizado em *background*, podendo assim, por exemplo, atender uma ligação telefônica, sem perder os dados, caso a aplicação seja finalizada. Ao final, quando o *download* for finalizado, uma notificação (figura 21) é exibida ao usuário informando o sucesso da ação. Caso o usuário escolha não fazer o *download* (figura 20 (1)), a notificação desaparece e a tela que o usuário se encontrava é exibida novamente.

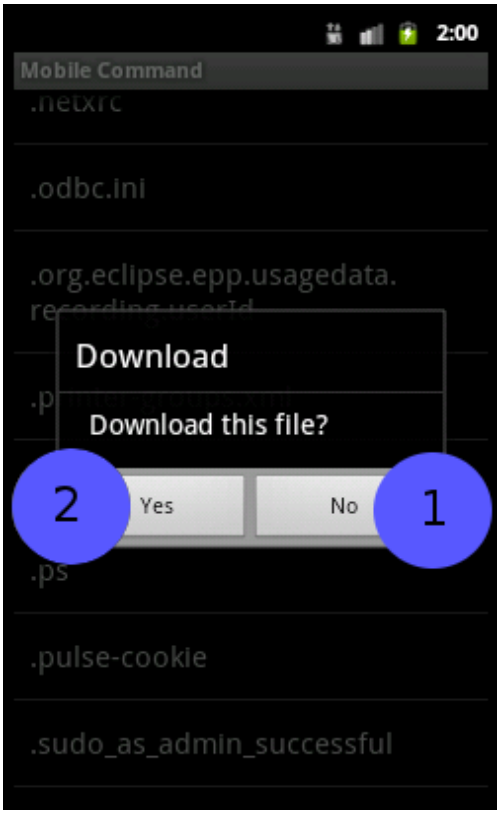


Figura 20 – Alerta de *download* de arquivo

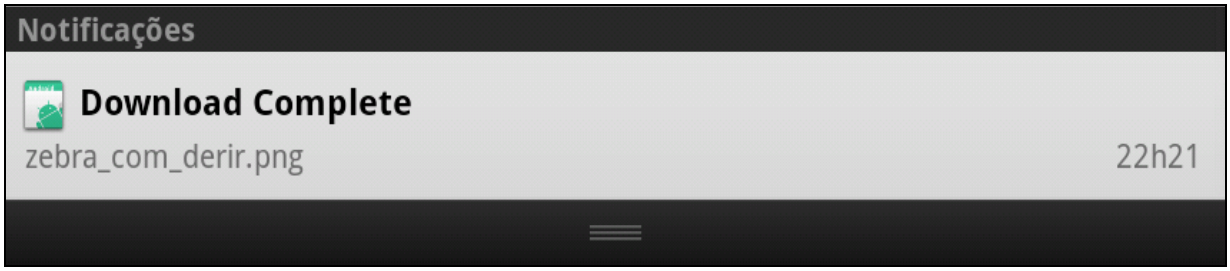


Figura 21 – Notificação de sucesso do *download*

A figura 22 mostra a tela de URL Download.

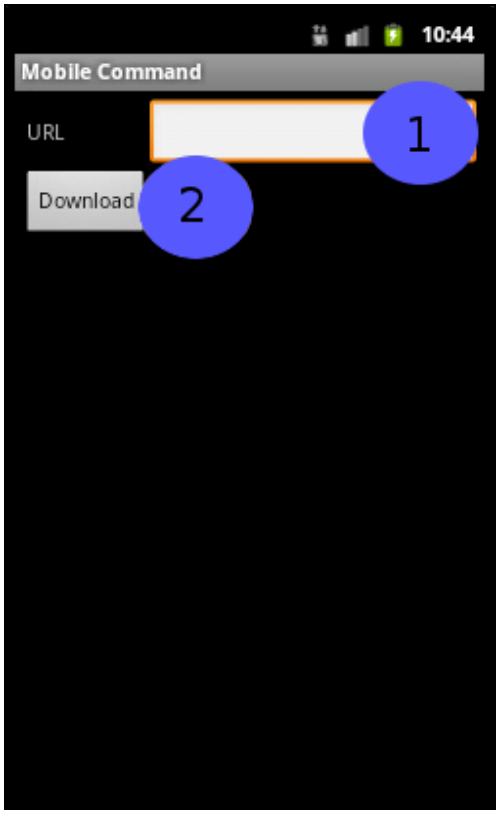


Figura 22 – Tela URL *Download*

Ao informar a URL (figura 22 (1)) e enviar (figura 22 (2)), o servidor se encarrega de realizar e avisar o usuário que o *download* está sendo realizado (figura 23 (3)), caso a URL esteja no formato correto. Se a URL não estiver no formato correto, é apresentada uma notificação na tela com o erro. O arquivo de *download* será armazenado em um diretório no qual é definido pelo usuário no arquivo `conf.properties` do sistema.

A Figura 23 mostra a notificação, caso o *download* tenha começado com sucesso.

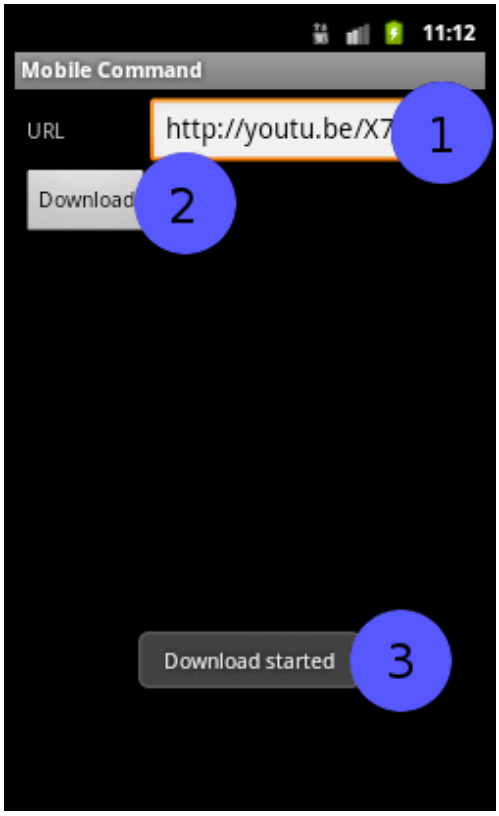


Figura 23 – Tela URL *Download*, resultado de sucesso

Para visualizar todos os processos que estão rodando no servidor, é necessário clicar no item *Running Process* da tela Menu. A figura 24 exibe a tela *Running Process*.

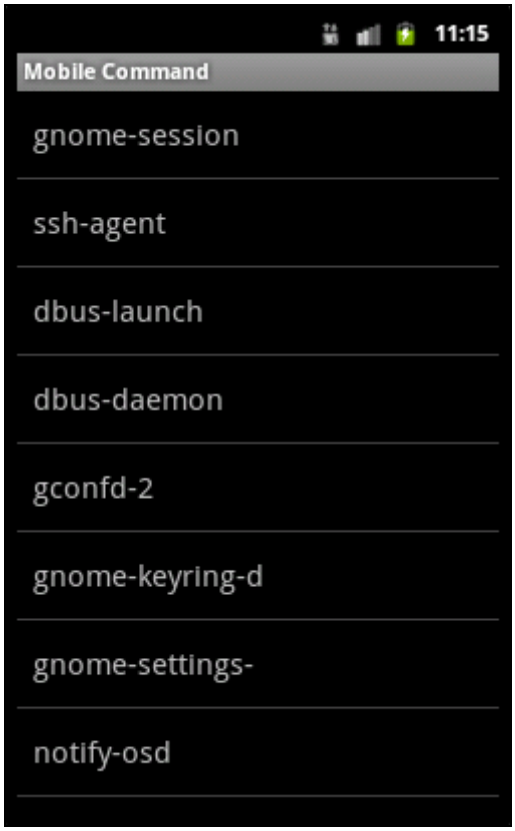


Figura 24 – Processos em execução no servidor

Ao clicar sobre um processo (item da figura 24), é apresentado um alerta se deseja finalizar ou não o processo, caso a escolha seja positiva (figura 25 (1)), o processo é finalizado e a tela é atualizada com todos os processos que estão atualmente rodando no servidor, caso a escolha seja negativa (figura 25 (2)), nada acontece e retorna a tela dos processos (figura 24).

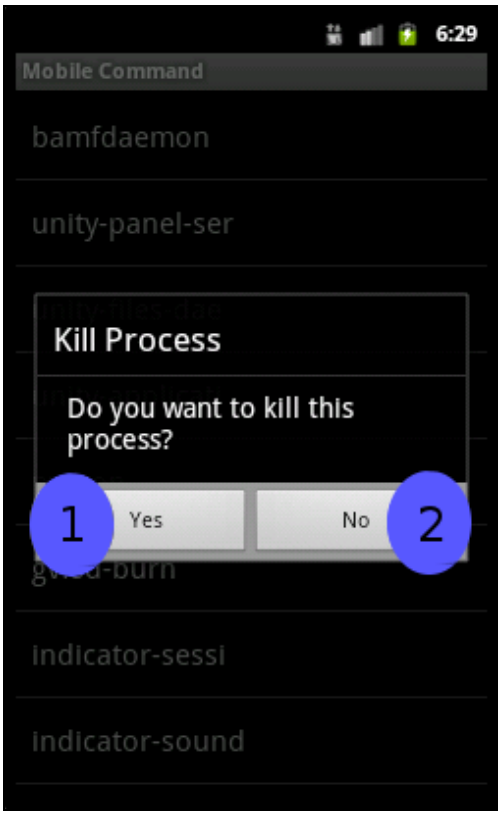


Figura 25 – Alerta para finalizar processo

O envio de comandos personalizados pelo usuário pode ser feito clicando no item *Execute Command* da tela menu. Ao ser escolhida a opção o usuário é redirecionado a tela *Execute Command*, que é apresentado na figura 26. Um formulário é exibido, possibilitando a escrita de um comando (figura 26 (1)) e o envio do mesmo comando para execução no servidor (figura 26 (2)). Ao ser executado com sucesso o comando enviado, um alerta é apresentado ao usuário. Na figura 27 é exibido o alerta com o resultado (figura 27 (1)) do comando `uname` que foi enviado ao servidor. Ao clicar no "OK" (figura 27 (2)) o alerta desaparece, e a tela *Execute Command* (figura 26) é exibida.

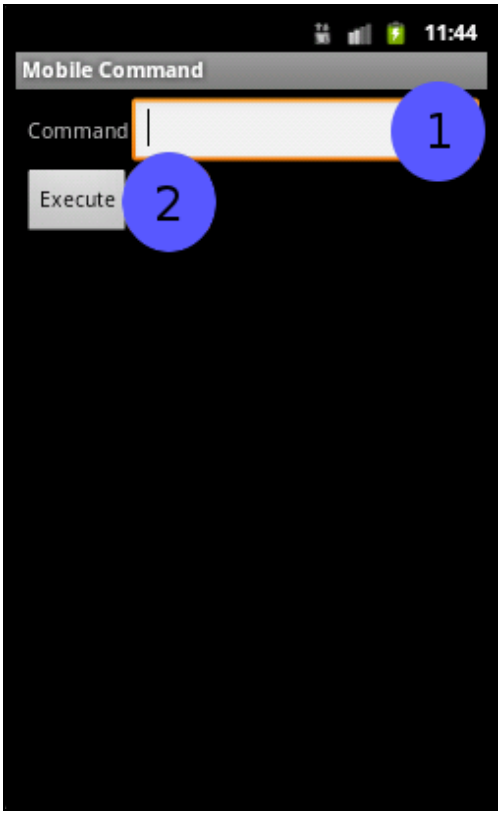


Figura 26 – Tela *Execute Command*

A Figura 26 mostra o resultado do comando enviado ao servidor.

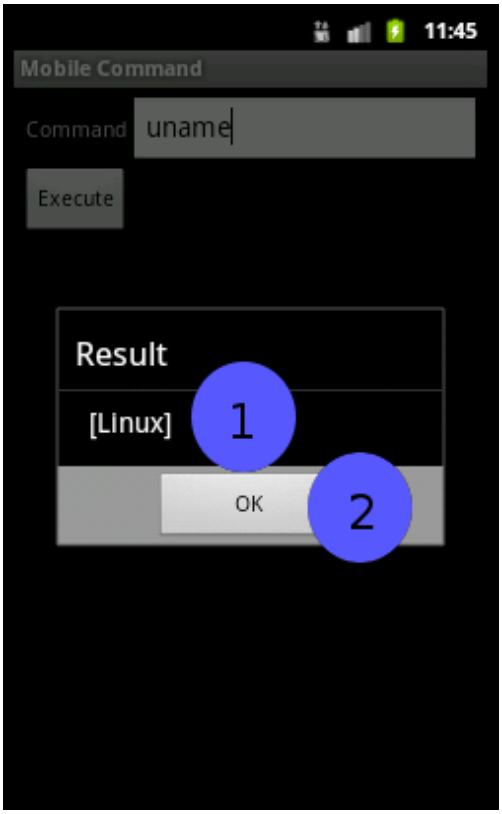


Figura 27 – Alerta do resultado do comando enviado

3.4 RESULTADOS E DISCUSSÃO

Os resultados encontrados com o término do trabalho puderam comprovar com a utilização da aplicação, que é possível acessar o servidor, navegar entre os diretórios, visualizar arquivos e fazer *download* dos mesmos, assim como executar comandos remotos, disponibilizando um protocolo de comunicação entre o *smartphone* e um computador *desktop*, alcançando assim o objetivo inicial.

No Quadro 42 é apresentado o comparativo da aplicação executando sobre diferentes plataformas. O módulo servidor foi desenvolvido para rodar sobre o sistema operacional Linux, mas graças a *Java Virtual Machine* (JVM) é possível rodar algumas funções da aplicação no Windows e Mac.

Funções	**Linux**	**Windows**	**Mac**
Directories and Files	X	X	X
URL Download	X	X	X
Process Runing	X		
Execute Command	X		

Quadro 42 – Comparativo do Mobile Command em diferentes plataformas

No quadro 43 são apresentados as diferenças entre este trabalho e os trabalhos correlatos

Funções	**MobileCommand**	**Ignition**	**ResMo**	**PhoneMyPC**
Acesso a área de trabalho do computador		X		X
Controle do mouse do computador		X		X
Executar comandos remotos	X	X	X	X
Visualizar diretórios e arquivos	X	X	X	X
Download de uma URL	X	X	X	X
Finalizar processos	X	X	X	X
Transferência de arquivo	X			
Baixa transferência de dados (Não visualiza a área de trabalho)	X		X	

Quadro 43 – Diferenças entre os trabalhos correlatos

4 CONCLUSÕES

O trabalho desenvolvido permite enviar comandos remotos para o computador *desktop*, disparados através de um *smartphone*.

Usando o SDK de desenvolvimento do Android, em conjunto com o ambiente de desenvolvimento Eclipse IDE, possibilitou o desenvolvimento de forma adequada dos requisitos estabelecidos neste trabalho. O emulador Android foi necessário para desenvolver, entender e realizar os testes, pois o mesmo disponibiliza quase todos os recursos que um celular possui. A plataforma Android também contempla, bibliotecas para o estabelecimento entre as partes envolvidas em uma comunicação segura, sendo possível trocar dados criptografados.

Em relação aos trabalhos correlatos apresentados, verifica-se que o trabalho proposto atende parte das funcionalidades existentes nas ferramentas disponíveis no mercado, mas de uma forma diferente de visualização, umas vez que não mostra a área de trabalho do sistema operacional no qual executa o módulo servidor, apenas comandos customizados, gerando uma troca menor de *bytes* entre as partes envolvidas, possibilitando rapidez na execução dos comandos e na visualização dos resultados, e esta torna-se a principal diferença e atrativo desta aplicação.

Por fim, este trabalho possibilitou trabalhar com uma nova plataforma, no caso a plataforma Android, assim como obter novas ideias para aplicativos usando dispositivos móveis, o qual é uma tendência nos dias atuais, tanto o uso de *smartphone* como o uso de *tablets*.

4.1 EXTENSÕES

Existem pontos que podem ser agregados ou melhorados na aplicação. Como sugestão pode-se citar:

a) controlar o módulo cliente através de comandos enviados pelo módulo servidor;
b) disponibilizar funções para o *smartphone* servir como um controle remoto do servidor, podendo abrir, executar e parar vídeos, músicas, etc;
c) disponibilizar o módulo cliente para os *smartphones* que possuam outro sistema operacional instalado, além do Android.

REFERÊNCIAS BIBLIOGRÁFICAS

ALEXANDER, Alvin. **Java exec**: execute system processes with Java ProcessBuilder and Process. [S.l.], 2010. Disponível em: <http://www.devdaily.com/java/java-exec-processbuilder-process-1>. Acesso em: 20 mar. 2011.

AQUINO, Juliana F. S. **Plataformas de desenvolvimento para dispositivos móveis**. 2007. 14 f. Monografia (Pós Graduação em Informática) – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro.

COMER, David E.; STEVENS, Douglas L. **Internetworking with TCP/IP**: client-server programming and applications. USA: Prentice-Hall, 1997.

GOOGLE. **Activity**. [S.l.], 2011a.Disponível em: <http://developer.android.com/reference/android/app/Activity.html>. Acesso em: 10 maio 2011.

_____. **What is Android**. [S.l.], 2011b. Disponível em: <http://developer.android.com/guide/basics/what-is-android.html>. Acesso em: 01 maio 2011.

_____. **Service**. [S.l.], 2011c. Disponível em: <http://developer.android.com/reference/android/app/Service.html>. Acesso em: 12 maio 2011.

JARGAS, Aurélio. **Shell script**. [S.l.], [2009?]. Disponível em: <http://www.aurelio.net/shell/>. Acesso em: 26 maio 2010.

KUROSE, James F.; ROSS, Keith W. **Redes de computadores e a internet**: uma abordagem top-down. São Paulo: Pearson Addison Wesley, 2006.

LOGMEIN. **Ignition**. [S.l.], 2010. Disponível em: <https://secure.logmein.com/products/Ignition/android/>. Acesso em: 20 maio 2011.

MACE, Michael M. **Will smartphone kill PC**. 2006. Disponível em: <http://mobileopportunity.blogspot.com/2006/10/will-smartphone-kill-pc.html>. Acesso em: 27 mar. 2010.

MATOS, Clésio R. **ResMo**: um software para a gerência de servidores Linux em dispositivos móveis usando a linguagem de programação J2ME e o protocolo GPRS/EDGE. 2009. 66 f. Monografia (Especialização em Administração em Redes Linux) - Centro de Ciências Exatas e Naturais, Universidade Federal de Lavras, Lavras.

OLIVEIRA, Rômulo. S.; CARISSIMI, Alexandre S.; TOSCANI, Simão S. **Sistemas operacionais**. Porto Alegre: Instituto de Informática da UFRGS, 2000.

OPEN HANDSET ALLIANCE. **Overview**. [S.l.], 2010. Disponível em: <http://www.openhandsetalliance.com/oha_overview.html>. Acesso em: 29 mar. 2010.

PANWAR, Shivendra S.; MAO, Shiwen; RYOO, Jeong-dong. **TCP/IP essentials**: a lab-based approach. USA: Cambridge University Press, 2004.

SOFTWAREFORME. **PhoneMyPc**. 2010. Disponível em: <http://softwareforme.com/phonemypc/>. Acesso em: 27 mar. 2010.

STEELE, James.; TO, Nelson. **The Android developer`s cookbook**: Building Applications with the Android SDK. Boston: Addison-Wesley, 2011.

STEVENS, Willian R.; FENNER, Bill; RUDOFF, Andrew M. **Unix network programming**: the sockets networking API. USA: Addison-Wesley, 2004.

TISTORY. **Android service**. [S.l.], 2011.Disponível em: <http://jjoong2.tistory.com/category/%EA%B0%9C%EB%B0%9C%EA%B4%80%EB%A0%A8/%EC%95%88%EB%93%9C%EB%A1%9C%EC%9D%B4%EB%93%9C%20%EC%A0%95%EB%A6%AC%20%28%20update%20file%20%29?page=3>. Acesso em: 16 maio 2011.

VANCE, Ashlee. **Fabricantes de PCs avançam no mercado de smartphones**. 2009. Disponível em: <http://tecnologia.terra.com.br/interna/0,,OI3641839-EI4796,00-Fabricantes+de+PCs+avancam+no+mercado+de+smartphones.html>. Acesso em: 29 mar. 2010.