UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

DESENVOLVIMENTO DE FRAMEWORK PARA JOGOS MULTIPLAYER PARA PLATAFORMA ANDROID

FELIPE GARCIA

BLUMENAU
2011

2011/2-7

FELIPE GARCIA

DESENVOLVIMENTO DE FRAMEWORK PARA JOGOS MULTIPLAYER PARA PLATAFORMA 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. Paulo Cesar Rodacki Gomes, Doutor – Orientador

BLUMENAU
2011

2011/2-7

DESENVOLVIMENTO DE FRAMEWORK PARA JOGOS MULTIPLAYER PARA PLATAFORMA ANDROID

Por

FELIPE GARCIA

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

Presidente:	Prof. Paulo Cesar Rodacki Gomes, Doutor – Orientador, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. Mauro Marcelo Mattos, Doutor – FURB

Blumenau, 12 de dezembro de 2011

Dedico este trabalho a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

A Deus, pelo seu imenso amor e graça.

À minha família, pelo apoio e compreensão.

Aos meus amigos, pelos empurrões e cobranças.

Ao meu orientador, Paulo Cesar Rodacki Gomes, por ter acompanhado a conclusão deste trabalho.

A Silvia Sales e Mauricio Kiniz, por me fornecerem a base para o meus conhecimentos de programação.

A Senior Sistemas, por me ingressar nesta carreira, pelos cursos que me foram fornecidos e pela ajuda com boa parte de minhas mensalidades.

Todos meus professores e outras pessoas que de alguma forma me ajudaram a chegar onde estou hoje.

Nossas dúvidas são traidoras e nos fazem perder o que, com frequência, poderíamos ganhar, por simples medo de arriscar.

William Shakespeare

RESUMO

Este trabalho apresenta a implementação de um framework da camada de rede de jogos multiplayer para a plataforma Android. O framework possibilita a utilização dos protocolos TCP/IP e UDP/IP para efetuar a comunicação entre diferentes dispositivos Android e/ou Java de forma simplicada. Para o envio e recebimento de informações o framework disponibiliza uma classe base chamada Packet do qual os pacotes do usuário irão estender para definir como este será lido e escrito. Esta classe pode ser utilizada em ambos os protocolos sem necessitarem de alteração. O framework também possui a opção de utilizar encriptação no envio dos dados e a possibilidade de controlar os pacotes que podem ser enviados em uma determinada conexão. Por fim será apresentado o desempenho do framework com o auxílio de aplicativos desenvolvidos para teste do framework.

Palavras-chave: Android. Rede. Jogos.

ABSTRACT

This work presents a framework to implement the network layer of multiplayer games for the Android platform. The framework allows the use of TCP / IP and UDP / IP to make the communication among different Android devices and / or Java so simple. For sending and receiving information the framework provides a base class called Packet which will be extended by the user packages to define how his packages will be read and written. This class can be used in both protocols without requiring changes. The framework also has the option to use encryption when sending data and the ability to control what packets can be send in a given connection. Finally, we will present the performance of the framework with the help of applications designed to test the framework.

Key-words: Android. Network. Games.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de principais componentes do Android 16
Figura 2 – Exemplo de rede IP com três *hosts* 19
Figura 3 – Exemplo de transmissão TCP pelo protocolo IP 20
Figura 4 – Cabeçalho IPv4 22
Figura 5 – Técnicas de transmissão: (a) *unicast*. (b) *multicast*. (c) *broadcast*. 23
Figura 6 – Arquiteturas de comunicação: (a) computador único. (b) ponto-a-ponto. (c) cliente-servidor. (d) ponto-a-ponto cliente-servidor híbrido. (e) rede de servidores. 24
Figura 7 – Exemplo de *packet flood* 27
Figura 8 – Diagrama de caso de uso 31
Quadro 1 – Caso de uso UC01 64
Quadro 2 – Caso de uso UC02 65
Quadro 3 – Caso de uso UC03 66
Quadro 4 – Caso de uso UC04 67
Quadro 5 – Caso de uso UC05 68
Quadro 6 – Caso de uso UC06 69
Figura 9 – Diagrama de pacote do *framework* 31
Figura 10 – Diagrama de classe do pacote `connection` 33
Figura 11 – Diagrama de classe do pacote `connection.cipher` 35
Figura 12 – Diagrama de classe do pacote `connection.packet` 36
Figura 13 – Diagrama de classe do pacote `connection.exception` 38
Figura 14 – Diagrama de classe para o cliente TCP do pacote `connection.wireless.tcp` 38
Figura 15 – Diagrama de classe para o servidor TCP do pacote `connection.wireless.tcp` 40
Figura 16 - Diagrama de classe para o cliente UDP do pacote `connection.wireless.udp` 41
Figura 17 - Diagrama de classe para o servidor UDP do pacote `connection.wireless.udp` 43
Figura 18 – Diagrama de sequência do cliente TCP 44
Figura 19 – Diagrama de sequência do cliente UDP 45

Quadro 7 – Método `getConnection` da classe `ConnectionFactory` 48
Quadro 8 - Método `getConnection` da classe `TCPConnectionFactory` 48
Quadro 10 – Exemplo de obtenção do serialVersionUID de um pacote 49
Quadro 11 – Encriptação AES128 50
Quadro 12 – Decriptação AES128 51
Quadro 13 – Execução do `onReceive` dos *listeners* de pacote 52
Quadro 14 – Escritor de pacotes TCP 53
Quadro 15 – Escritor de pacotes UDP 54
Quadro 16 – Leitor de pacote TCP 55
Quadro 17 – *Thread* leitora de datagramas 56
Quadro 18 – Leitor de pacotes UDP 57
Figura 20 – Tela principal do *messenger* 58
Figura 21 – Tela de configuração 59
Figura 22 – A esquerda Motorola Milestone após receber o pacote 6196, a direita início da execução do aplicativo servidor em Java 59
Figura 23 – Jogo snake iniciado no Motorola Milestone à esquerda e no Motorola XOOM à direita. A cobra verde é a cobra controlada pelo jogador e a azul é a do adversário. 60
Quadro 19 – Comparação das funcionalidades básicas de um framework de redes 61

LISTA DE SIGLAS

AES - *Advanced Encryption Standard*

API – *Application Programming Interface*

DDoS - *Distributed Denial of Service*

EDGE - *Enhanced Data rates for Gsm Evolution*

GMS - *Global System for Mobile communication*

GNE - *Game Network Engine*

GPS - *Global Positioning System*

IDE - *Integrated Development Environment*

IP – *Internet Protocol*

ISO - *International Organization for Standardization*

LAN – *Local Area Network*

MD5 - *Message-Digest algorithm 5*

MMO – *Massive Multiplayer Online*

NGP – *Next Generation Portable*

OO – Orientação a Objetos

OpenGL ES - *Open Graphics Library for Embedded Systems*

OSI - *Open Systems Interconnection*

PSP – *PlayStation Portable*

SDK - *Software Development Kit*

TCP – *Transmission Control Protocol*

UDP – *User Datagram Protocol*

UML - *Unified Modeling Language*

VoIP - *Voice over Internet Protocol*

WAN – *Wide Area Network*

Wi-Fi - *Wireless Fidelity*

WPA - *Wi-Fi Protected Access*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 OBJETIVOS DO TRABALHO 13
1.2 ESTRUTURA DO TRABALHO 14
2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 ANDROID 15
2.1.1 Intenções 17
2.2 ARQUITETURA DE REDES PARA JOGOS 17
2.2.1 Protocolos de rede 17
2.2.2 Técnicas de transmissão 22
2.2.3 Arquitetura de comunicação 23
2.2.4 Prevenção de *cheat* 25
2.3 TRABALHOS CORRELATOS 28
3 DESENVOLVIMENTO DO FRAMEWORK 30
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 30
3.2 ESPECIFICAÇÃO 30
3.2.1 Casos de Uso 31
3.2.2 Diagrama de classes 31
3.2.3 Diagrama de sequência 43
3.3 IMPLEMENTAÇÃO 47
3.3.1 Técnicas e ferramentas utilizadas 47
3.3.2 O *framework* 47
3.3.3 Operacionalidade do *framework* 58
3.4 RESULTADOS E DISCUSSÃO 60
4 CONCLUSÕES 62
4.1 EXTENSÕES 62

1 INTRODUÇÃO

A tecnologia de computação fez grandes saltos frente a miniaturização, poder e sofisticação. Alta velocidade de redes internacionais de dados fazem parte da vida moderna, todos os dias no que é chamdo de 'a Internet'. O peculiarmente humano desejo de entretenimento e diversão tem empurrado a fusão e evolução da computação e das tecnologias de rede. Hoje, jogos de computador são um mercado cada vez mais importante cujas receitas anuais já excedem a da indústria de cinema de Hollywood (ARMITAGE, 2006, p. 12).

Essa constante necessidade de evolução dos jogos de computador, leva aos dias atuais onde há consoles de jogos muito poderosos, com alta qualidade gráfica, acesso a internet e capacidade de capturar movimentos corporais. Computadores são utilizados por milhões de pessoas que interajem entre si através de jogos *multiplayers online*. Consoles portáteis como o *Next Generation Portable* (NGP) sucessor do *Playstation Portable* (PSP) e o Nintendo 3DS estão cada vez mais avançados.

Tendo em vista toda essa evolução alavancada pelos jogos, pode-se pensar na próxima estapa desta evolução, que seria unir o console portátil e o *smartphone*. Um exemplo hoje no mercado é o *smartphone* Xperia Play lançado pela Sony com um controle físico para jogos e capaz de executar jogos de outros consoles da marca Sony.

Este trabalho descreve o desenvolvimento de um *framework* para jogos *multiplayer* em rede utilizando a plataforma Android, explorando o potencial dos recursos de rede desta plataforma com os protocolos de comunicação *Transmission Control Protocol* (TCP) sobre *Internet Protocol* (IP) e *User Datagram Protocol* (UDP) sobre IP com possibilidade de encriptação dos dados transferidos pela rede. Avalia também a eficiência dos recursos *Wireless Fidelity* (Wi-Fi) utilizados, considerando os pontos positivos e negativos da tecnologia utilizada.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver um framework para jogos multiplayer na plataforma Android. Foi criada também uma aplicação para efetuar os testes e demonstrar a utilização do framework.

Os objetivos específicos do trabalho são:

a) possibilitar utilização em jogos cujo servidor é centralizado em um disposivo móvel ou computador;
b) possibilitar utilização em jogos onde não há um servidor centralizado;
c) utilizar uma criptografia que forneça segurança das informações transmitidas;
d) disponibilizar uma avaliação da eficiência dos recursos Wi-Fi da plataforma Android com relação aos itens acima (a, b, c).

1.2 ESTRUTURA DO TRABALHO

A estrutura deste documento está apresentada em quatro capítulos, sendo que o segundo capítulo contém a fundamentação teórica necessária para o entendimento deste trabalho.

O terceiro capítulo apresenta aspectos do desenvolvimento do framework na plataforma Android, os casos de uso do framework, os diagramas de classe e toda a especificação que define o framework. Ainda no terceiro capítulo são apresentadas as partes principais da implementação, o aplicativo de teste e também os resultados obtidos.

Por fim, o quarto capítulo refere-se às conclusões do presente trabalho e sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A seção 2.1 apresenta a plataforma Android. A seção 2.2 descreve arquitetura de redes para jogos e suas características. Por fim, a seção 2.3 traz os trabalhos correlatos.

2.1 ANDROID

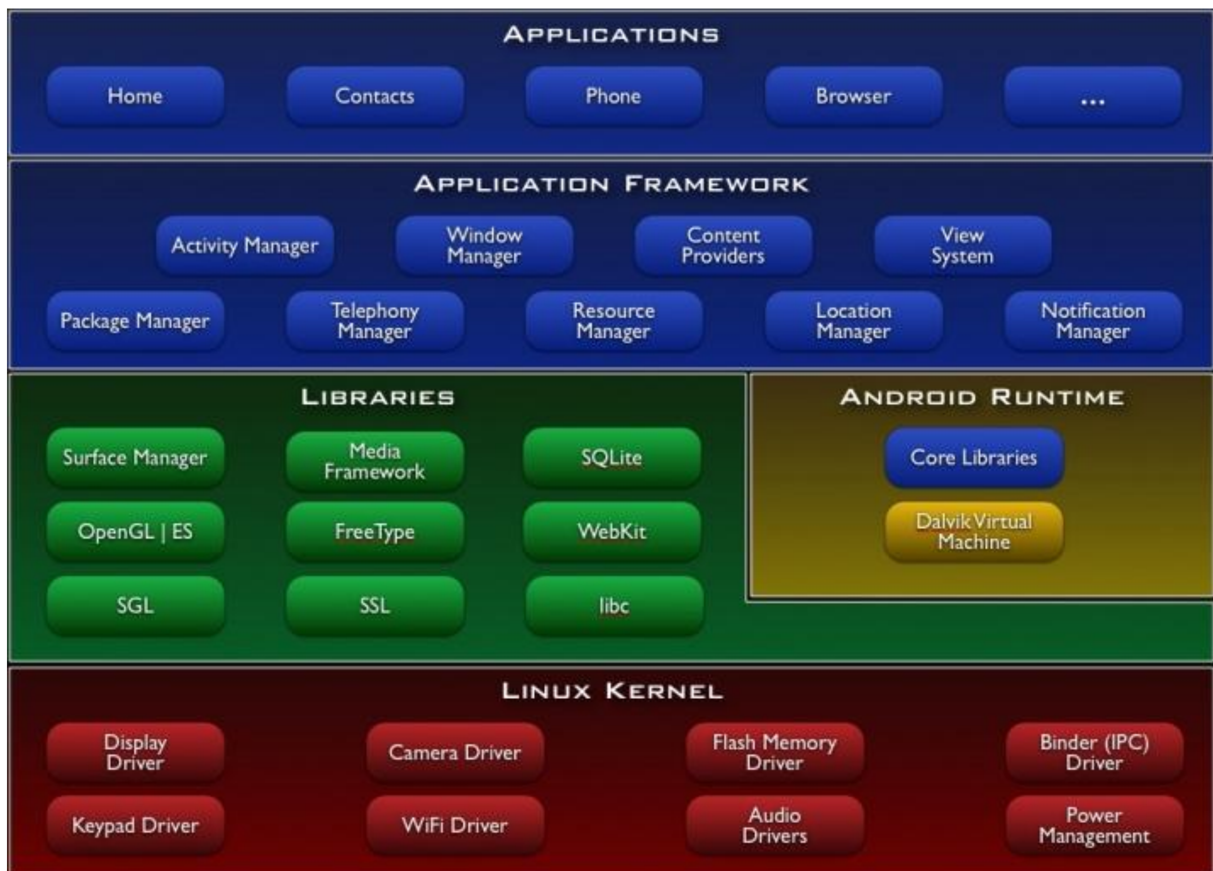
Android é uma plataforma para dispositivos móveis que inclui um sistema operacional, um *middleware* e aplicações chaves (ANDROID DEVELOPERS, 2011a). Ela foi inicialmente criada pela empresa Android Inc. que foi comprada pela Google Inc. em 2005.

As principais funcionalidades existentes no Android são as seguintes (ANDROID DEVELOPERS, 2011a):

a) *framework* da aplicação possibilitando reutilização e substituição de componentes;
b) máquina virtual Dalvik otimizada para dispositivos móveis;
c) *browser* integrado baseado no *open source* WebKit;
d) gráficos otimizados através de uma biblioteca 2D padrão e 3D baseada na especificação Open Graphics Library for Embedded Systems (OpenGL ES) 1.0;
e) SQLite para armazenamento dados estruturados;
f) suporte a mídias comuns de áudio, vídeo e imagem;
g) telefonia *Global System for Mobile Communication* (GSM) (dependente de hardware);
h) *bluetooth*, *Enhanced Data rates for GSM Evolution* (EDGE), 3G e Wi-Fi (dependente de hardware);
i) câmera, *Global Positioning System* (GPS), bússola e acelerometro (dependente de hardware);
j) ambiente de desenvolvimento rico incluindo um simulador de dispositivo, ferramentas para *debug*, *profiler* de memória e *performance*, e um *plugin* para o Eclipse *Integrated Development Environment* (IDE).

A Figura 1 mostra um diagrama dos principais componentes da plataforma Android. Em síntese, a plataforma Android opera sobre um Kernel Linux utilizando a Dalvik Virtual Machine que é a responsável por transformar *bytecodes* Java que possuem a extensão `class`

em *bytecodes* próprios com a extensão `dex` (Dalvik *Executable*), e diversas bibliotecas como, por exemplo, a biblioteca do SQLite. Sobre estas bibliotecas há um *framework* de aplicações que possui diversas *Application Programming Interfaces* (APIs) como a API de tela e a de notificação. Finalmente sobre este *framework* são desenvolvidas as aplicações.



Fonte: Android Developers (2011a).

Figura 1 – Diagrama de principais componentes do Android

O desenvolvimento para a plataforma Android é realizado utilizando-se do *Software Development Kit* (SDK) do Android na versão desejada e do Eclipse IDE com o *plugin* do Android instalado. O *plugin* e o SDK estão disponíveis no site oficial da plataforma Android e o Eclipse IDE pode ser obtido no site oficial do Eclipse. Os softwares citados são gratuitos e podem ser baixados por qualquer pessoa que deseja utilizá-los. A linguagem de desenvolvimento é Java com restrições de algumas bibliotecas e adição de algumas bibliotecas próprias da plataforma Android.

A Google também disponibiliza uma loja virtual chamada Android Market para aplicativos Android onde qualquer desenvolvedor pode postar suas aplicações. Para fazer uso da loja o desenvolvedor deve efetuar um cadastro de desenvolvedor e aceitar um termo de utilização. As aplicações gratuitas podem ser postadas sem nenhum custo, já em aplicações pagas 30% do valor é direcionado para Google e 70% para o desenvolvedor.

2.1.1 Intenções

O Android possui um mecanismo de troca de mensagens entre aplicações através de intenção (representado por uma classe chamada `Intent`). Esta classe permite que uma aplicação solicite de outras aplicações a utilização de determinadas funcionalidades que são comuns entre várias aplicações, como por exemplo, abrir imagens, efetuar ligações, abrir endereços de sites, utilizar o Wi-Fi (ANDROID DEVELOPERS, 2011b).

No caso de existir apenas uma aplicação que suporte a solicitação, o Android se encarrega automaticamente de abrir a aplicação ao ser feita a solicitação. Quando há mais de um aplicativo que suporta a aplicação o Android disponibiliza opções para usuário para que ele se encarregue de escolher a aplicação desejada.

Outro ponto importante a comentar sobre o mecanismo de intenções é a possibilidade de o usuário ter noção dos recursos utilizados pelo aplicativo. Ao instalar um aplicativo no Android a plataforma mostra todas as intenções do mesmo. O usuário então poderá aceitar a instalação do aplicativo caso esteja de acordo com as intenções requeridas, ou cancelar a instalação, caso contrário.

2.2 ARQUITETURA DE REDES PARA JOGOS

Esta seção apresenta os principais tópicos da arquitetura de rede que devem ser de conhecimento para o desenvolvimento de um framework para jogos em rede. É apresentado o funcionamento de protocolos de rede, as técnicas de transmissão da informação, as arquiteturas de comunicação que podem ser utilizadas e por fim algumas técnicas para prevensão de *cheats*.

2.2.1 Protocolos de rede

É uma linguagem que dois ou mais computadores utilizam para compartilhar informações entre eles. Naturalmente, todos os computadores devem falar a mesma linguagem para garantir que a informação seja transmitida com sucesso (MULHOLLAND,

2004, p. 103).

Um protocolo define as regras de comunicação e os formatos dos dados. Protocolos são padrões que funcionam em diferentes computadores, isto é, se dois computadores estão conectados na mesma rede eles poderão compartilhar informações independentes do sistema operacional ou arquitetura que estes possuem (MULHOLLAND, 2004, p. 103).

2.2.1.1 Modelo Open Systems Interconnection (OSI)

A *International Organization for Standardization* (ISO) desenvolveu um modelo de padronização de comunicação de dados no final da década de 70 chamado de OSI. Este modelo é baseado em sete camadas. Cada camada possui uma área unica do processo de comunicação de dados e são conectadas entre si. Entre cada camada existe uma interface que estas camadas utilizam para se comunicar. O padrão OSI então define a função de cada camada e suas interfaces, mas não define como essas camadas são criadas. O fato de não definir como as camadas são criadas torna simples a criação de novas soluções em comunicação de dados utilizando o modelo OSI e a alteração das soluções já existentes. Com isso tem-se a capacidade de escolher entre diferentes opções de implementação de cada camada. Como exemplo pode-se citar TCP e UDP, os dois protocolos são implementações da camada de transporte, é possível escolher um desses protocolos para utilizar na camada de transporte sem a necessidade de se preocupar com as demais camadas devido a interfaces entre as camadas garantirem que os dados fluam corretamente (MULHOLLAND, 2004, p. 104).

A seguir é apresentada uma descrição da função de cada camada (MULHOLLAND, 2004, p. 105):

a) camada física: É a primeira camada e inclui dispositivos eletrônicos como circuitos, cabos e conectores. Basicamente define o meio utilizado para a transmissão de dados;
b) camada de ligação de dados: É a segunda camada e consiste da placa de rede e do driver do dispositivo;
c) camada de rede: É a terceira camada, reposável pelo IP. Ela endereça os pacotes e roteia os pacotes pelas sub-redes até o endereço correto;
d) camada de transporte: É a quarta camada e define o método de transmissão de dados, TCP e UDP pertencem a está camada;

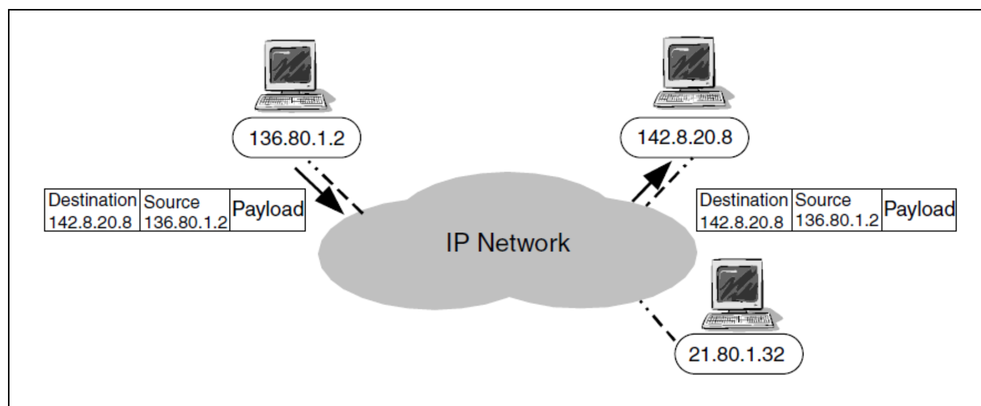
e) camada de sessão: É a quinta camada, responsável pela sincronização da sessão, definindo quando pode receber e transmitir dados;
f) camada de apresentação: É a sexta camada, responsável pela compressão e/ou encriptação dos dados;
g) camada de aplicação: É a sétima camada, define uma aplicação como por exemplo um e-mail.

2.2.1.2 Rede Internet Protocol (IP)

Uma maneira simples de entender uma rede IP é como uma nuvem com dispositivos anexados em torno das suas bordas. Estes dispositivos podem ser qualquer hardware capaz de transmitir ou receber informações. O principal objetivo da rede IP é prover conectividade, ou seja, transmitir dados de uma fonte para um destino. Redes IP costuman ignorar o conteúdo dos pacotes, não oferecer garantia do tempo de entrega dos mesmos e nem mesmo garantia que o pacote irá chegar ao seu destino (ARMITAGE, 2006, p. 42).

Os dispositivos anexados comumente chamados de *host*s são identificados por endereços de IP. Endereços de IP são atualmente utilizados em duas versões IPv4 e IPv6. Endereços IPv4 são identificados por um número de 32 bits representados por quatro áreas, cada uma representando o valor numérico de um byte separado por ponto (128.80.195.7). Já endereços IPv6 são identificados por um número de 128 bits e representado por oito áreas cada uma com 4 números hexadecimais (2001:0db8:85a3:08d3:1319:8a2e:0370:7344) (ARMITAGE, 2006, p. 42).

A Figura 2 apresenta um exemplo de uma rede IP com três *hosts* com um pacote sendo encaminhado do *host* 136.80.1.2 para o *host* 142.8.20.8.



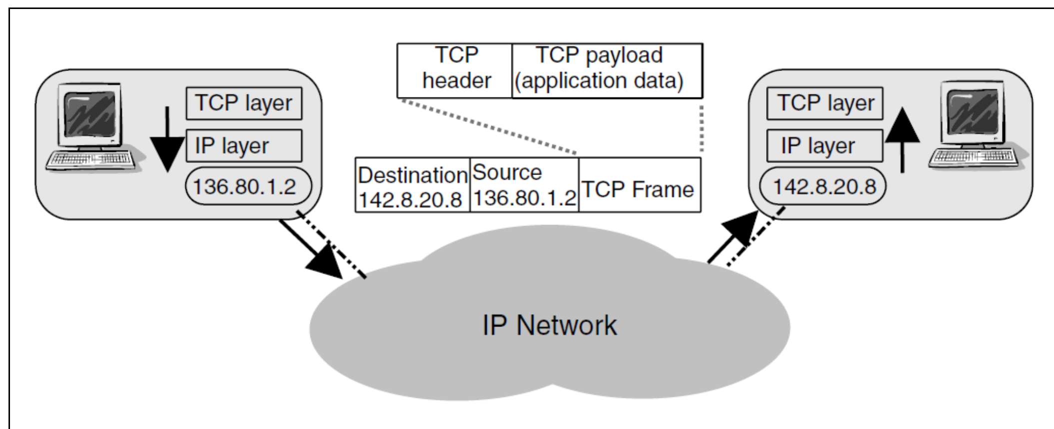
Fonte: Armitage (2006).

Figura 2 – Exemplo de rede IP com três *hosts*

2.2.1.3 Transmission Control Protocol (TCP)

TCP é um protocolo da camada de transporte que está diretamente sobre a camada IP. Seu objetivo é criar um caminho bidirecional entre dois *hosts*. Os dados da aplicação são quebrados em partes e transmitidos dentro de *frames* TCP que são enviados dentro de pacotes IP através da rede até o seu *host* de destino (Figura 3). O *host* de destino notifica explicitamente o recebimento do *frame* garantindo que a camada TCP seja capaz de detectar os frames perdidos na transmissão pela camada IP e retransmiti-lo garantindo que o pacote seja entregue (ARMITAGE, 2006, p. 44).

O fato de o protocolo TCP continuar enviando o *frame* mesmo que a transmissão sofra repetidas perdas de pacotes na camada IP pode fazer com que a aplicação sofra variações de latência imprevisíveis (ARMITAGE, 2006, p. 45).



Fonte: Armitage (2006).

Figura 3 – Exemplo de transmissão TCP pelo protocolo IP

2.2.1.4 User Datagram Protocol (UDP)

UDP é outro tipo de protocolo da camada de transporte que também está diretamente sobre a camada IP. O protocolo UDP não faz verificação de recebimento dos dados enviados, com isso não garante que o pacote chegou ao seu destino. O protocolo UDP também não necessita efetuar uma conexão entre *host* de envio e o de destino como é feito no protocolo IP, ou seja, ele apenas envia a informação para o destino sem saber se o mesmo está apto a receber os dados ou não (ARMITAGE, 2006, p. 45).

2.2.1.5 TCP X UDP

Considerando os dois protocolos citados anteriormente surge uma importante questão: qual seria o protocolo mais adequado para ser utilizado no desenvolvimento de um jogo? Para responder esta pergunta primeiro faz-se uma análise de cada protocolo (FIEDLER, 2008).

Características do protocolo TCP (FIEDLER, 2008):

a) orientado a conexão;
b) garante entrega e ordem dos *frames;*
c) divide automaticamente seus dados em pacotes;
d) garante que os dados não são enviados rápidos demais para a capacidade de transmissão da rede;
e) fácil de utilizar.

Características do protocolo UDP (FIEDLER, 2008):

a) sem conceito de conexão, a menos que seja implementado;
b) não garante entrega ou ordem, os pacote podem não chegar, serem duplicados e até perdidos;
c) é necessário quebrar manualmente seus dados em pacotes antes de envialos;
d) deve-se garantir que os dados não sejam enviados rápidos demais para a capacidade de transmissão da rede;
e) se um pacote é perdido, deve-se implementar uma forma de detectar e reenviar o pacote, se necessário.

A decisão parece clara, TCP faz tudo que é necessário e é mais simples de usar enquando utizar UDP é muito mais complexo e é preciso implementar vários comportamentos (FIEDLER, 2008). Mas o protocolo TCP nem sempre é o mais apropriado. Para decidir qual protocolo utilizar deve-se verificar as características do jogo que será implementado. Na maioria dos casos, jogos com grande tráfego de informações em tempo real, ou seja, tráfego de informações que precisam ser mostradas no mesmo instante no host de destino caso contrário as informações serão inutilizadas, devem utilizar UDP, visto que, TCP pode gerar muita latência retransmitindo pacotes perdidos. Algumas empresas, para reduzir o esforço com implementação utilizando UDP, optam ainda por utilizar um modelo híbrido, utilizando UDP e TCP conforme a necessidade (FIEDLER, 2008).

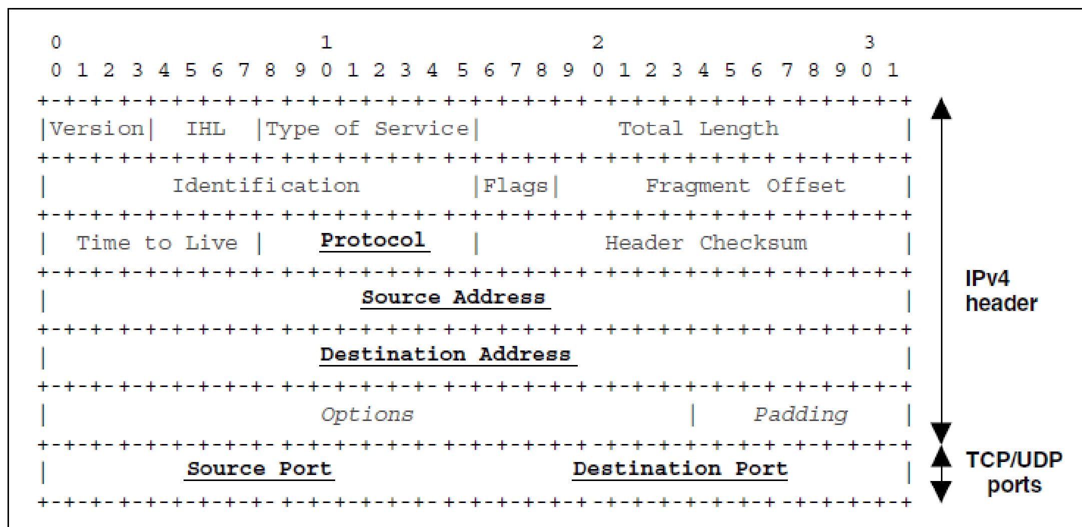
Nos demais jogos que não possuem grande tráfego de informações em tempo real costuma-se utilizar o protocolo TCP pela sua praticidade. Ainda assim, algumas empresas

preferem optar pelo protocolo UDP por motivos de desempenho (FIEDLER, 2008).

2.2.1.6 Multiplexação

Os *frames* TCP e UDP carregam dois números de 16 bits que são chamados de números de porta, estes números identificam a fonte e o destino de um *frame* no seu *host* fonte ou destino. Este recurso permite que diferentes aplicações que residem no mesmo *host* ou em *hosts* diferentes trafeguem dados entre si, sem precisar de vários endereços de IP na rede (ARMITAGE, 2006, p. 46).

A Figura 4 mostra os campos chaves de um cabeçalho IPv4 e os primeiros 32 bits do cabeçalho TCP ou UDP (ARMITAGE, 2006, p. 46):



Fonte: Armitage (2006).

Figura 4 – Cabeçalho IPv4

2.2.2 Técnicas de transmissão

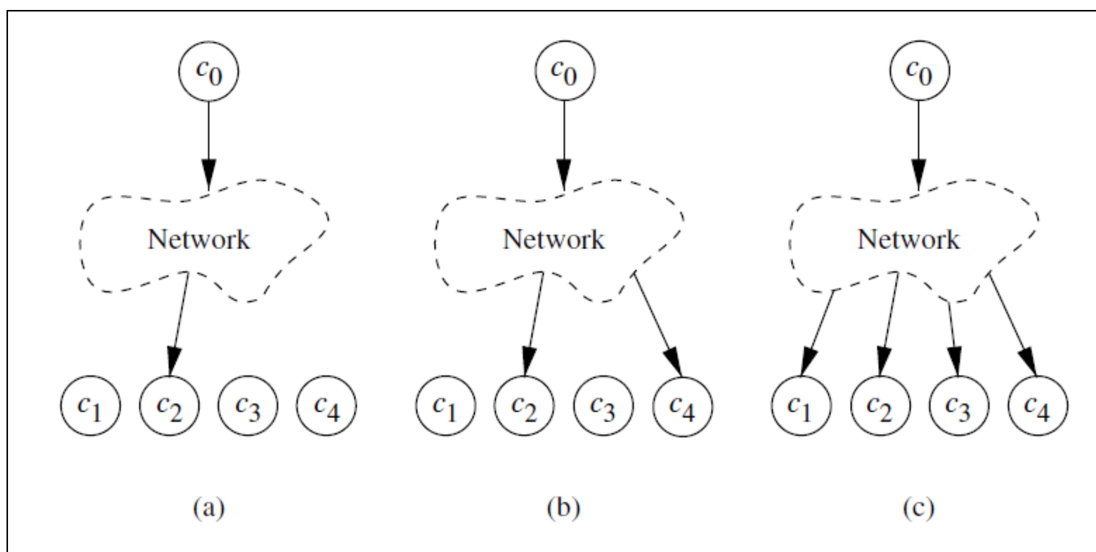
Técnicas de transmissão são maneiras como o transmissor de uma informação se comunica com o receptor da informação. Elas podem ser divididas em três tipos (Figura 5) (SMED, 2006, p. 174):

a) *unicast*: é uma comunicação entre um transmissor e um único receptor e permite trafego direto ponta a ponto, se a informação tem o objetivo de ser enviada a vários receptores a transmissão unicast desperdiça conexão por enviar mensagens

redundantes;

b) *multicast*: é a comunicação entre um transmissor e vários receptores. Ela permite que os receptores se inscrevam para receber informações de seu interesse. O transmissor não precisa conhecer os receptores inscritos mas irá mandar uma única mensagem, que será recebida por todos os receptores inscritos;

c) *broadcast*: é a comunicação entre um transmissor e todos os receptores. Todo receptor terá que receber e processar a mensagem. Isso pode ser um problema a medida que o número de receptores aumente, por esse motivo algumas redes bloqueiam transmissões *broadcast*.

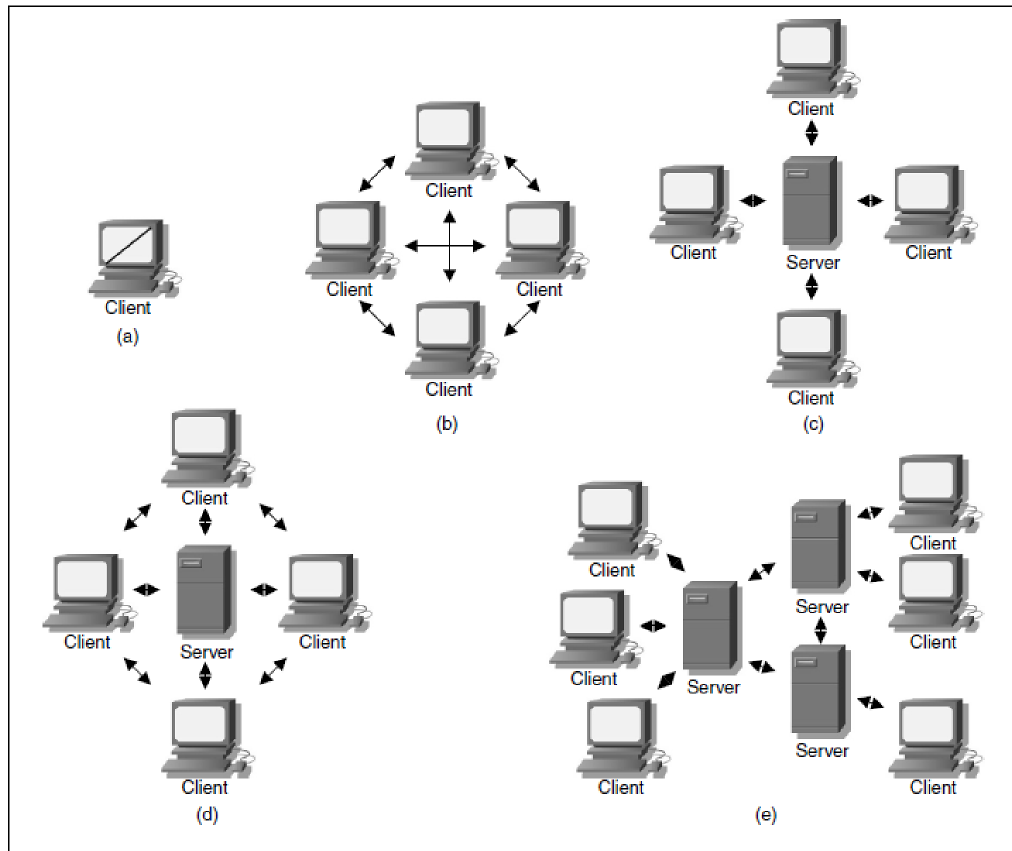


Fonte: Smed (2006).

Figura 5 – Técnicas de transmissão: (a) *unicast*. (b) *multicast*. (c) *broadcast*

2.2.3 Arquitetura de comunicação

Durante a evolução dos jogos *multiplayer* foram consideradas diversas arquiteturas para fazer a comunicação entre os jogadores. As alternativas de comunicação são mostradas na Figura 6.



Fonte: Armitage (2006).

Figura 6 – Arquiteturas de comunicação: (a) computador único. (b) ponto-a-ponto. (c) cliente-servidor. (d) ponto-a-ponto cliente-servidor híbrido. (e) rede de servidores

No inicio dos jogos *multiplayer* não havia uma rede entre os jogadores, a funcionalidade *multiplayer* era alcançada tendo vários jogadores interagindo no mesmo computador. Essa interação era realizada com todos os jogadores na mesma tela ou dividindo-se a tela em várias áreas sendo uma para cada jogador (ARMITAGE, 2006, p. 16).

Na arquitetura ponto-a-ponto, cada cliente processa em um ponto e nenhum ponto tem mais controle sobre o jogo que outro. Arquiteturas ponto a ponto são populares em jogos *multiplayer* jogados em *Local Area Network* (LAN) devido ao suporte a broadcast e de possuir geralmente um número menor de jogadores que participam de um jogo. Mesmo que arquiteturas ponto-a-ponto possam ser utilizadas em *Wide Area Network* (WAN) como a internet, elas não conseguem ser escaláveis suficiente sem uma hierarquia adicional (ARMITAGE, 2006, p. 16).

Na arquitetura cliente-servidor, um computador faz o papel do servidor comunicando-se com cada computador cliente e fazendo papel do mediador. Os clientes não se comunicam diretamente entre si. Nessa arquitetura o servidor é a parte crítica, pois deve ter capacidade de comunicação e de computação necessários, caso contrário a jogabilidade pode degradar para todos os clientes. Se o cliente não consegue se comunicar com o servidor ele é desconectado

do jogo até conseguir recuperar a comunicação. Esta é a arquitetura mais popularmente utilizada em jogos *online* comerciais (ARMITAGE, 2006, p. 17).

Na arquitetura ponto-a-ponto cliente-servidor híbrido, o servidor serve de mediador para as informações base enviadas pelos clientes como no cliente-servidor tradicional, mas os clientes também podem se comunicar com os outros clientes como na arquitetura ponto-a-ponto tradicional. A comunicação entre os clientes são geralmente informações não críticas para o funcionamento do jogo, como por exemplo, um jogador se comunicar com o outro por *Voice over Internet Protocol* (VoIP) (ARMITAGE, 2006, p. 17).

Por fim a arquitetura de rede de servidores onde há uma cadeia de servidores, a comunicação entre os servidores pode ser ponto-a-ponto onde cada servidor tem a mesma função ou no modelo cliente-servidor onde os servidores se comunicam com servidores mestre obtendo-se assim uma arquitetura hierarquica de servidores. Separando os clientes através de varios servidores reduz a necessidade de servidores muito potentes e aumenta a escalabilidade da arquitetura do jogo. Em contrapartida a arquitetura de rede de servidores é um mecanismo mais complicado e difícil de manter as informações consistentes (ARMITAGE, 2006, p. 17).

2.2.4 Prevenção de *cheat*

*Cheat*ers são jogadores que atacam jogos de computador em rede geralmente motivados por vandalismo ou dominância. Entretanto, apenas uma minoria dos *cheaters* é motivada por vandalismo. A grande maioria tem o objetivo de dominar, possuir uma posição sobre-humana ou adquirir vantagens com relação a outros jogadores (SMED, 2006, p. 213).

Com o aumento da lucratividade dos jogos online, os *cheaters* passaram a ser realizados também para ganhos financeiros realizados com a venda de items pagos e a venda de personagens já prontos. Naturalmente, as perdas financeiras causadas diretamente ou indiretamente por *cheater*s são a maior preocupação dos produtores de jogos *online* e a principal motivação para implementar medidas contra *cheaters*. Por outro lado, algumas vezes, os jogos podem acabar postergando a correção de uma falha devido a possibilidade desta falha atrair mais jogadores para o jogo (SMED, 2006, p. 213).

A prevenção de *cheat* possui três objetivos distintos:

a) proteger informações importantes;
b) prover um jogo justo para todos os jogadores;

c) manter o jogo desafiador.

É importante citar que alguns tipos de *cheat* não podem ser totalmente prevenidos, muitas vezes a prevenção dá-se em dificultar o *cheat* a ponto em que o custo de invadir o sistema seja mais alto do que os benefícios provenientes da invasão.

Nas seções a seguir são citadas algumas das práticas de *cheat* mais comuns.

2.2.4.1 Packet Tampering

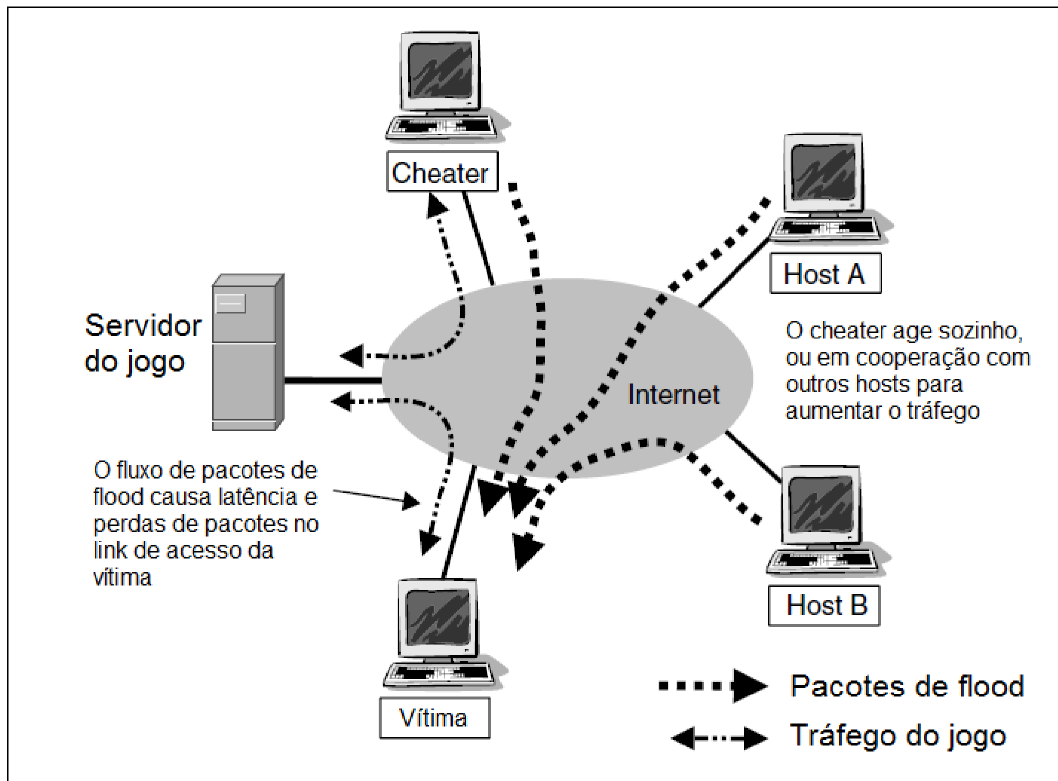
Consiste na modificação de bytes de pacotes transmitidos para ver o que acontece. A primeira defesa contra esse tipo de ataques é uma simples função de *hash*. O algoritmo de *hash* irá embaralhar uma cadeia de bytes de forma que seja impossível decodificar essa cadeia para o valor anterior. A cadeia de bytes é enviada junto com o pacote. Ao receber o pacote o receptor executa a função *hash* sobre a mesma informação e verifica se as cadeias de byte são idênticas. Caso as cadeias de bytes sejam diferentes o pacote é rejeitado. Um bom algoritmo para geração de *hash* é chamado de Message-Digest algorithm 5 (MD5). É um algoritmo bem testado, diponível publicamente e rápido suficiente para ser utilizado em jogos (DELOURA, 2000, 105).

2.2.4.2 Packet Replay

Consiste em capturar um pacote do cliente e então reenviá-lo múltiplas vezes. Esta tática pode ser utilizada para executar comando mais rápido do que o jogo permite, mesmo que o jogo faça uma checagem de tempo o *packet replay* ainda pode ser utilizado para automatizar a execução de um comando. Uma maneira de prevenir este problema seria enviar uma informação que sempre seja diferente mesmo que o pacote seja idêntico, por exemplo, a geração de um número aleatório através de uma regra. No caso de envios sem garantia de entrega o número será válido ser for um número possível da sequência (DELOURA, 2000, 105).

2.2.4.3 Packet Flood

É causado pela geração excessiva de pacotes para um determinado jogador, esses pacotes podem ser gerados apenas pelo *cheater*, pelo *cheater* em cooperação com outros jogadores ou até em sistemas mais complexos como aplicações *Distributed Denial of Service* (DDoS) . A Figura 7 demonstra como o *packet flood* é realizado (ARMITAGE, 2006, p. 115).



Fonte: Armitage (2006).

Figura 7 – Exemplo de *packet flood*

Essa geração excessiva de pacotes faz com que o jogador que está sendo atacado tenha sua jogabilidade degradada ou em níveis mais extremos pode até causar uma desconexão do jogo devido a concorrência dos pacotes na sua rede (ARMITAGE, 2006, p. 115).

Esse problema pode ser solucionado se toda a comunicação do cliente for realizada através dos servidores, sem possuir conexão direta com outro cliente e sem que os servidores divulguem os endereços de IP dos outros jogadores (ARMITAGE, 2006, p. 115).

2.2.4.4 Engenharia Reversa

Outro problema muito difícil de resolver é o problema da engenharia reversa. Como o

cliente tem acesso a todos os fontes da aplicação cliente, inclusive o algoritmo de encriptação, ele sempre poderá fazer engenharia reversa (DELOURA, 2000, 107).

A melhor solução para esse problema é tornar a engenharia reversa do sistema mais difícil, tornando assim o custo de realizá-la proibitiva. Seguem alguns artifícios que podem ser utilizados (DELOURA, 2000, 107):

a) remover todas as informações de depuração e comentários;
b) não isolar os algoritmos de encriptação e decriptação. Em vez disso, combinar esses algoritmos com outros códigos do sistema, reduzindo a manutenibilidade mas aumentando a segurança;
c) computar números mágicos na execução em vez de colocar os valores diretamente no código;
d) incluir uma boa chave de encriptação em cada versão do cliente, inclusive nas versões *beta*.

2.3 TRABALHOS CORRELATOS

Foram encontrados diversos projetos de *framework* de rede para jogos, tais como GNet (GAMATRA, 2009), Game Network Engine (GNE) (WINNEBECK, 2011) e RakNet (JENKINS SOFTWARE, 2011). Especificamente para dispositivos móveis há o Photon Network Engine (EXIT GAMES, 2011).

GNet é uma *engine* de jogos em rede desenvolvido pela empresa Gamantra criado especificamente para jogos no estilo Massive Multiplayer Online (MMO). Ela suporta desenvolvimento nas linguagens C++ e Java. O GNet não se propõe apenas a fornecer a funções básicas de rede, mas também possui funcionalidades como chat, sistema de clã e persistência em banco de dados (GAMATRA, 2009).

GNE é projeto *open source* desenvolvido em C++. O GNE até a conclusão deste trabalho, implementa toda a parte de transmissão e tratamento de erros utilizando o protocolo UDP com ou sem garantia de entrega, e a arquitetura de comunicação ponto-a-ponto, fazendo com que desenvolvedor não precise se preocupar com a camada mais baixo nível da comunicação. Existe a previsão que futuramente o projeto também possuirá suporte a arquitetura cliente-servidor, comunicação entre jogadores por texto e a atribuição de identificadores para cada jogador (WINNEBECK, 2011).

RakNet é uma engine de jogos em rede desenvolvido pela Jenkins Software LLC. Ele suporta o desenvolvimento nas linguagens C++ e C#. Dentre suas funcionalidades pode-se citar a replicação de objetos entre os computadores, segurança da conexão por diversos algoritmos de encriptação, mecanismo auto-atualização dos arquivos do jogo e comunicação por voz (JENKINS SOFTWARE, 2011).

Por último o Photon Network Engine é uma engine de jogos desenvolvido pela Exit Games exclusivamente para plataformas móveis como o iOS, Android e Windows Phone (EXIT GAMES, 2011).

O Photon implementa o protocolo UDP com ou sem garantia de entrega e o protocolo TCP. Um ponto a ser observado é que mesmo o cliente podendo ser implementado na plataforma Android, iOS ou Windows Phone o servidor deve ser implementado obrigatoriamente na linguagem C# (EXIT GAMES, 2011).

3 DESENVOLVIMENTO DO FRAMEWORK

Neste capítulo serão detalhadas a etapas do desenvolvimento do framework. São apresentados os requisitos principais, a especificação, a implementação e ao final são apresentados os resultado e discussão.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O *framework* deve:

a) permitir o tráfego de informações entre cliente e servidor (Requisito Funcional - RF);
b) possibilitar o tráfego de informações por *Transmission Control Protocol* (TCP) e por *User Datagram Protocol* (UDP) sobre o protocolo *Internet Protocol* (IP) (Requisito Não Funcional - RNF);
c) fazer uso da conexão Wi-Fi (RNF);
d) utilizar a linguagem de programação Java para plataforma Android (RNF);
e) utilizar a Eclipse IDE como ambiente de desenvolvimento (RNF);
f) prover segurança das informações trafegadas (RF);
g) possuir extensibilidade para implementação de outros tipos de protocolos e outras formas de conexões (RNF).

3.2 ESPECIFICAÇÃO

A especificação deste trabalho foi desenvolvida através da ferramenta Gliffy e da ferramenta StarUML utilizando conceitos de Orientação a Objetos (OO) através dos diagramas de caso de uso, classe e atividade da *Unified Modeling Language* (UML).

3.2.1 Casos de Uso

Esta seção descreve todos os casos de uso da aplicação. Foi identificado apenas o ator desenvolvedor. Ele faz uso das ferramentas disponibilizadas pelo *framework* para possibilitar o desenvolvimento da camada de rede do seu jogo. Na Figura 8 é apresentado o diagrama de caso de uso do framework.

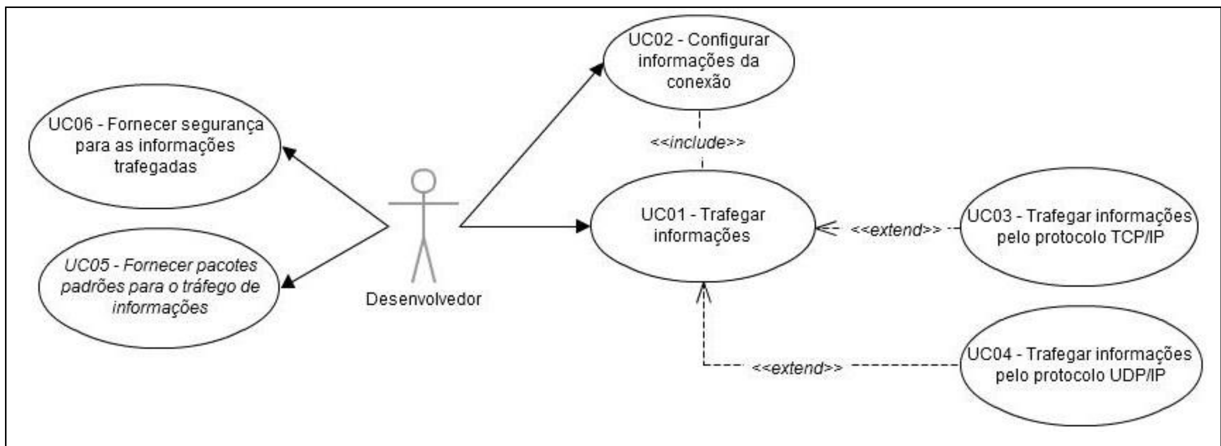


Figura 8 – Diagrama de casos de uso

3.2.2 Diagrama de classes

A seguir são descritas as classes necessárias para o desenvolvimento do framework. Para melhorar o entendimento, primeiramente a Figura 9 apresenta um diagrama de pacotes com os pacotes do *framework* e suas dependências. Neste diagrama não são apresentadas as classes contidas em cada pacote para facilitar o entendimento do mesmo.

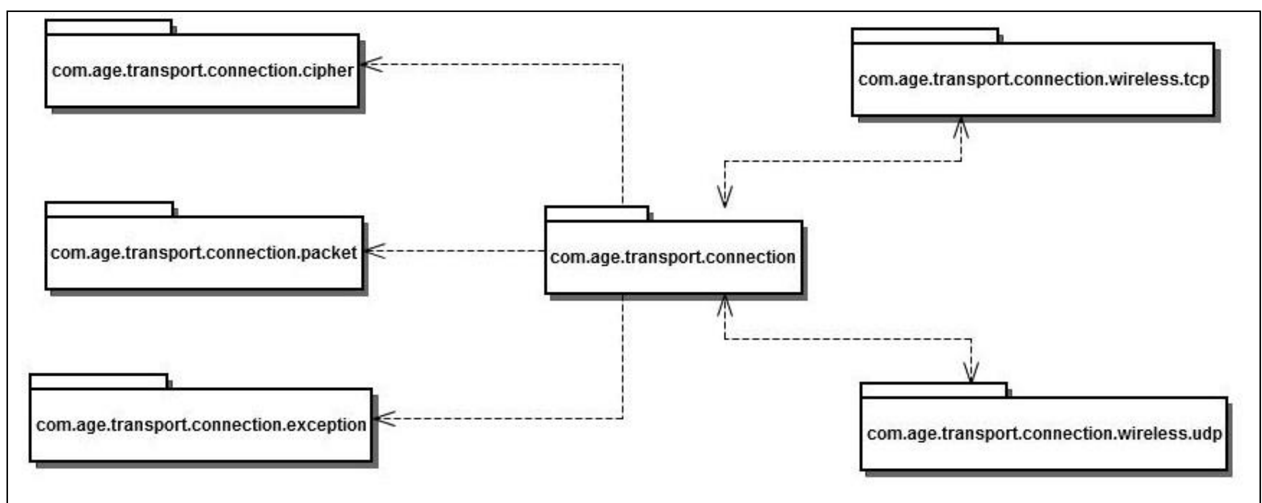


Figura 9 – Diagrama de pacote do *framework*

3.2.2.1 Pacote com.age.transport.connection

O pacote `com.age.transport.connection` (Figura 10) é o principal pacote da aplicação. Nele estão contidas as principais interfaces do *framework* e a classe de configuração.

A interface IConnection é a base do framework, ela representa uma conexão de um computador com um ou mais computadores, qualquer meio de comunicação que se queira implementar no *framework* deve implementar essa interface. O framework já possui a implementação desta interface para conexões TCP/IP e UDP/IP. Esta interface possui o métodos `start` e `close` que inicia e finaliza uma conexão, o método `sendPacket` que envia pacote sem encriptação, o método `sendEncryptedPacket` que envia um pacote com a encriptação passada no parâmetro `cipher` ou sem encriptação caso seja informado `null` neste parâmetro, o método `getPacketReceivedListeners` retorna a lista de *listeners* de pacote desta conexão, o método `addPacketReceivedListener` adiciona um *listener* de pacote a conexão e o método `removePacketReceivedListener` remove o *listener* de pacote informado como parâmetro, e finalmente os métodos `setAllowedPacket` e `getAllowedPacket` informa e retorna respectivamente o mapa de pacotes permitidos para a conexão.

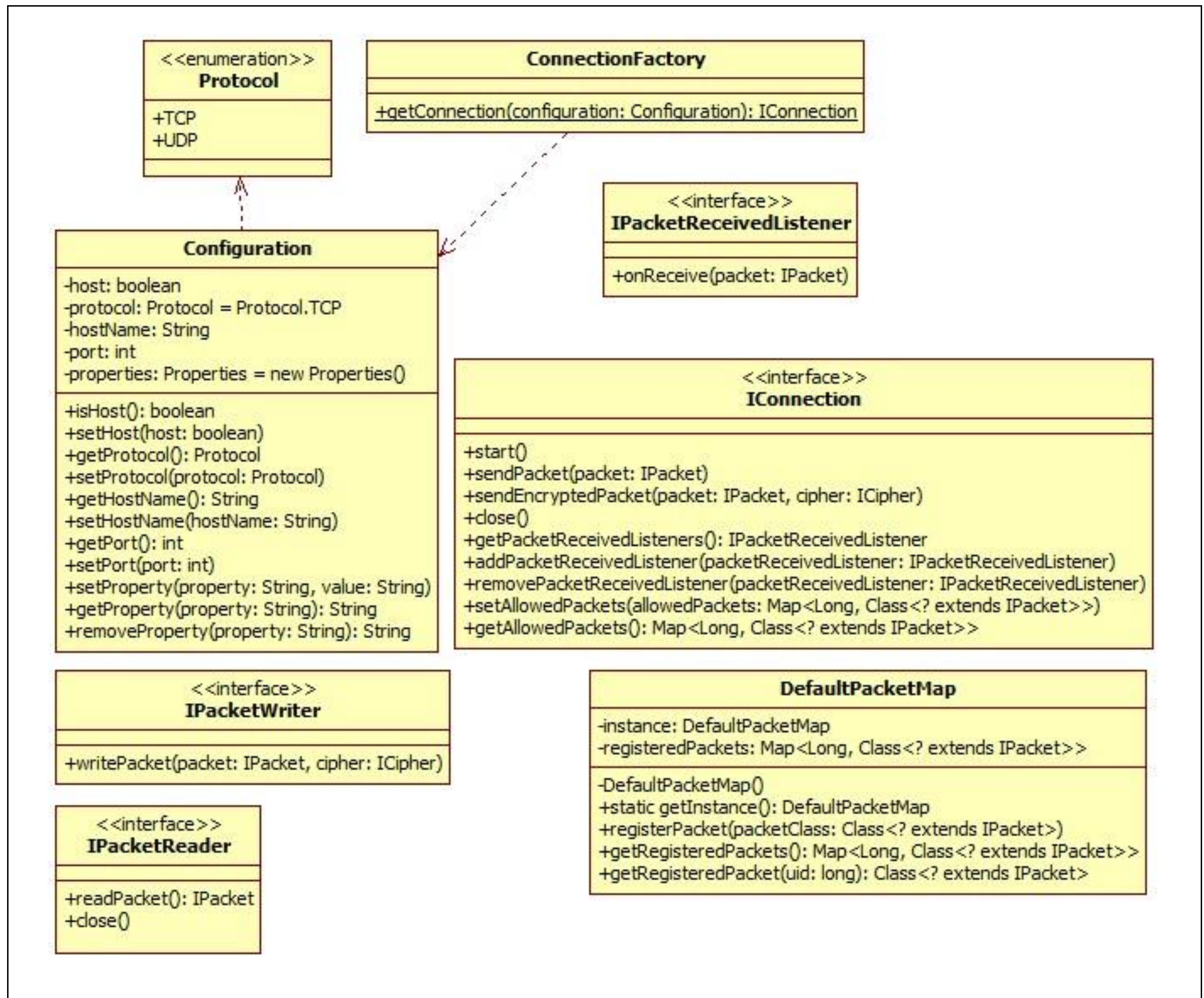


Figura 10 – Diagrama de classe do pacote `connection`

A interface `IPacketReader` é a interface base para a implementação de um leitor de pacotes. Ela possui o método `readPacket` que retorna o pacote lido pela conexão e é utilizada pelas implementações da classe `IConnection` dentro de uma `Thread` para ficar escutando todos os pacotes que chegam à conexão, a exceção `CipherExcetion` caso haja algum problema com a decodificação da encriptação, `AccessViolationException` caso o pacote não seja permitido na conexão, `ConnectionException` caso haja algum problema na conexão como uma queda de conexão ou `PacketException` caso haja problema na leitura do pacote. Além deste método também há o método `close` que faz as finalizações do leitor de pacote caso seja necessário, como é o caso da implementação para o protocolo UDP que será apresentada mais adiante.

A interface `IPacketWriter` faz o inverso da interface `IPacketReader`, ou seja, ela envia os pacotes para todos os receptores da conexão, o envio é realizado pelo método `writePacket` que envia o pacote passado como parâmetro utilizando a encriptação passa

também como parâmetro ou sem encriptação caso o parâmetro seja null. Se houver algum problema na encriptação do pacote ela retorna CipherException e se houver problema na transformação do pacote para um array de bytes ela retorna PacketException.

A interface IPacketReceivedListener é a interface base das classes que irão escutar os pacotes lidos pela conexão. Nenhuma classe do *framework* implementa essa interface. Sua implementação é de responsabilidade do desenvolvedor que irá utilizar o framework, assim como a associação desse *listener* com a conexão. Essa interface possui apenas o método onReceive que é chamado a cada pacote recebido pela conexão para todas a implementações do *listener* associadas à conexão.

A classe DefaultPacketMap é um mapa padrão de pacotes permitidos. Este mapa é utilizado caso não seja informado nenhum mapa de pacotes permitidos na conexão. Esta classe utiliza o padrão de projeto *singleton*, portanto qualquer inclusão ou remoção de pacotes do mapa afetará todas as conexões que utilizam o mapa de pacotes padrão. Por padrão este mapa é criado sem nenhuma permissão de pacote. Esta classe possui o atributo instance onde é armazenada a única instância da classe, o construtor privado para proibir a criação de instâncias fora da classe, o método getInstance que retorna a única instância da classe, e os métodos registerPacket, getRegisteredPackets e getRegisteredPacket que respectivamente registram, obtém os pacotes registrados e retorna o pacote com o uid informado do atributo registeredPackets.

A classe *Configuration* tem por objetivo servir como a classe de configuração padrão para qualquer tipo de conexão. Ela possui o atributo host que identifica se a conexão é cliente no caso de ter o valor false ou servidor no caso de ter o valor true, o atributo protocol que identifica o tipo de protocolo da configuração, o atributo hostName que identifica o endereço de ip do servidor no caso de ser uma conexão cliente, o atributo port que armazena a porta utilizada pela conexão e o dicionário properties utilizado para a inclusão de propriedades adicionais, além dos seus respectivos *getters* e *setters*. O dicionário properties também possui o método removeProperty utilizado para remover uma propriedade informada no método setProperty.

A classe ConnectionFactory é uma implementação do padrão de projeto *abstract factory* utilizada para encontrar a conexão mais adequada para a configuração realizada. Ela possui apenas o método getConnection que busca a conexão ideal de acordo com o tipo de protocolo ou lança a exceção ConnectionException caso não encontre uma conexão.

A enumeração Protocol identifica os protocolos que podem ser utilizados, atualmente

possui apenas o valor `TCP` e o valor `UDP`, mas poderá ter outro valor caso seja implementada uma nova forma de conexão.

3.2.2.2 Pacote com.age.transport.connection.cipher

O pacote `com.age.transport.connection.cipher` (Figura 11) é responsável pela encriptação dos pacotes.

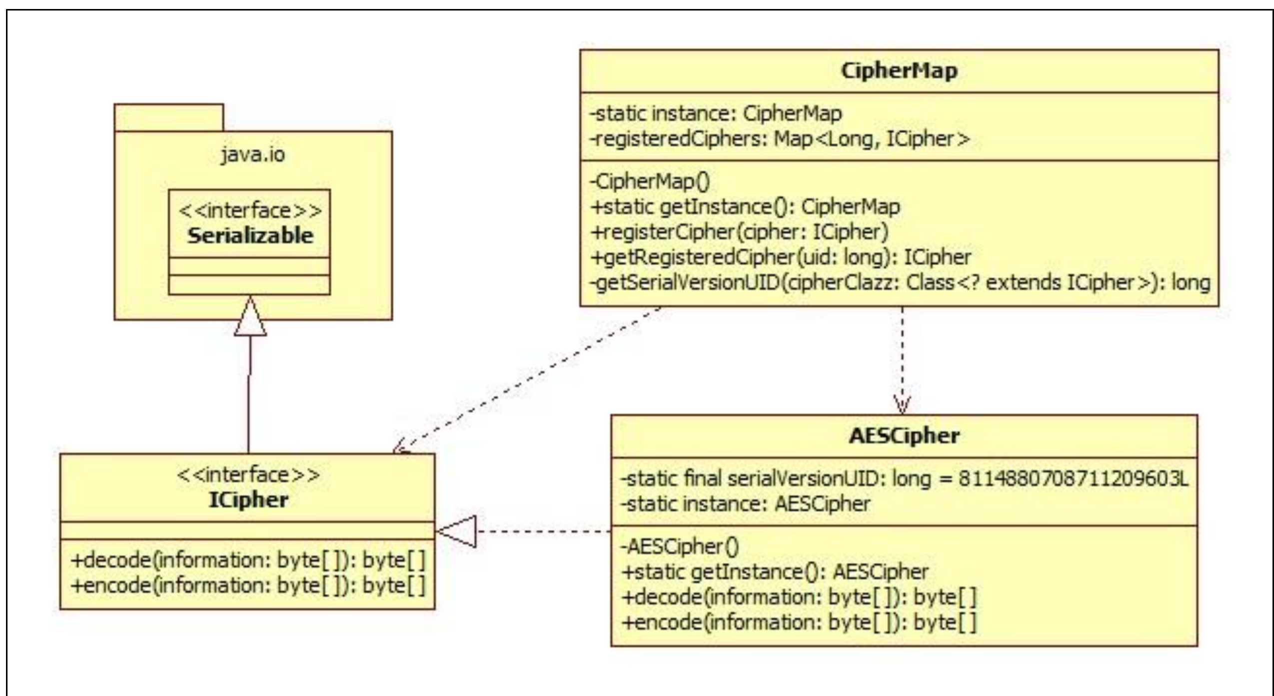


Figura 11 – Diagrama de classe do pacote `connection.cipher`

A interface `ICipher` é a interface base da implementação de algoritmos de encriptação. Ela possui o método `encode` que recebe um array de bytes e encripta este array retornando um array de bytes encriptado, e o método `decode` que faz o contrário, recebe um array de bytes encriptado e retorna o array de bytes decriptado. Ambos lançam `CipherException` caso ocorra algum erro. A interface `ICipher` estende a interface `Serializable` para possuir um atributo `serialVersionUID` que identifique cada implementação da interface com um id único.

A classe `AESCipher` é uma implementação da interface `ICipher` utilizando o algoritmo de encriptação AES 128bits. Essa classe serve como encriptador padrão caso não haja a necessidade de implementar outro, mas também serve de exemplo de implementação da interface. Como não há necessidade de possuir mais de uma instância da classe `AESCipher` foi utilizado o padrão de projeto Singleton. Ela possui atributo `serialVersionUID` que é um

atributo necessário quando a classe implementa `Serializable`, o atributo `instance` com a única instância da classe, o construtor privado para garantir que não sejam criadas novas instâncias, o método `getInstance` para obter a instância única, e as implementações dos métodos `decode` e `encode` da interface `ICipher`.

A classe `CipherMap` é uma Singleton que gerencia um mapa que deve conter todas as implementações da interface `ICipher`. Ao implementar a interface `ICipher` o desenvolvedor deve registrar sua classe neste mapa, caso contrário não conseguirá utilizá-la. Devido ao `AESCipher` ser nativo do framework esta classe já é registrada no `CipherMap` na construção da instância. Ela possui o atributo `instance`, o construtor privado e o método `getInstance` padrões de classes Singleton, o atributo `registeredCiphers` onde são armazenados as classes registradas, o método `registerCipher` que registra uma nova implementação de `ICipher`, o método `getRegisteredCipher` que obtém a classe com o `serialVersionUID` passado como parâmetro e o método privado `getSerialVersionUID` utilizado para obter o `serialVersionUID`.

3.2.2.3 Pacote com.age.transport.connection.packet

O pacote `com.age.transport.connection.packet` (Figura 12) é o pacote onde está contida a interface `IPacket`, sua implementação para `String` e `Object`, e uma classe utilitária para obter o `serialVersionUID` do pacote.

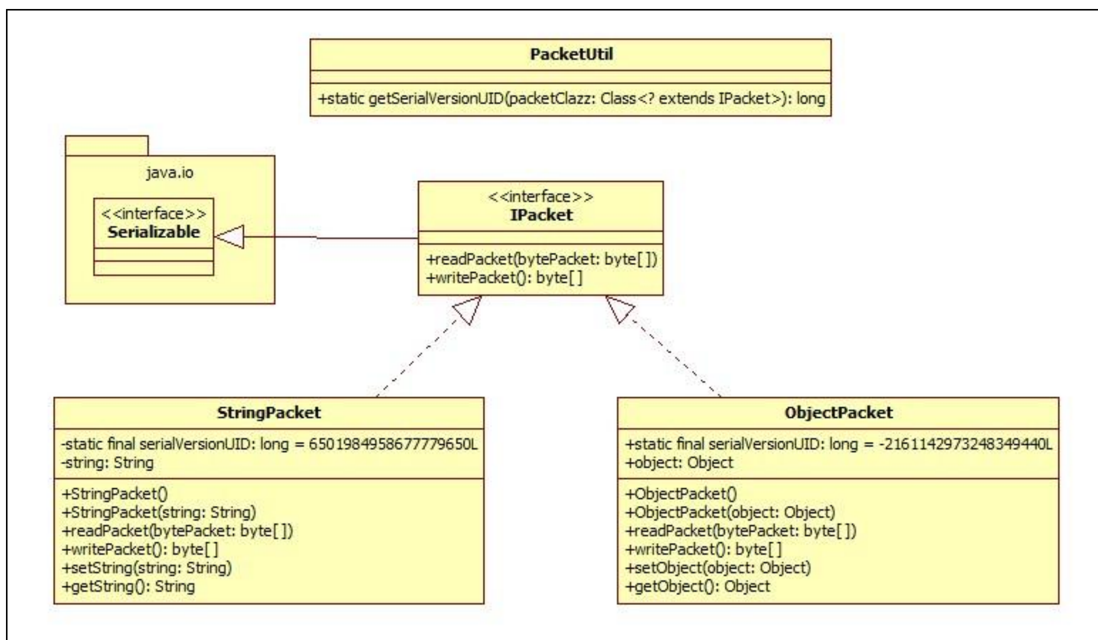


Figura 12 – Diagrama de classe do pacote `connection.packet`

A interface IPacket é a interface base para a implementação de pacotes. Muito similar a interface ICipher esta interface também estende a interface Serializable para obter um identificador único para cada implementação de IPacket. Ela também possui o método readPacket que faz a leitura de um pacote a partir de um array de bytes atribuindo os dados lidos aos atributos da classe e o método writePacket que retorna um array de bytes contendo as informações dos atributos da classe.

A classe StringPacket é uma implementação da interface IPacket para efetuar a transferência pela rede de informações do tipo String. Ao enviar um pacote do tipo StringPacket o atributo string é convertido para um array de bytes enviado pela rede e transformado novamente para uma String no receptor do pacote. A classe possui o serialVersionUID para identificar o pacote, um atributo string com seu *getter* e *setter* para armazenar a informação que deve ser enviada, o construtor vazio utilizado pelo *framework* via reflexão no receptor, o construtor com uma string como parâmetro para simplificar a criação do pacote pelo desenvolvedor, e os métodos readPacket e writePacket que fazem a transformação da informação.

A classe ObjectPacket é muito similar a classe StringPacket com a diferença que a classe ObjectPacket aceita todo tipo de objeto e não apenas String. Internamente a classe utiliza ObjectInputStream e ObjectOutputStream, portanto, os objetos transmitidos por esse pacote devem obrigatoriamente implementar Serializable.

A classe PacketUtil é uma classe utilitária que possui apenas o método getSerialVersionUID que pode ser utilizado pelo desenvolvedor para facilitar a obtenção do id único do pacote.

3.2.2.4 Pacote com.age.transport.connection.exception

O pacote com.age.transport.connection.exception (Figura 13) é onde estão armazenadas todas as exceções do framework. Este pacote é utilizado apenas por organização visto que as exceções poderiam estar nos outros pacotes sem problemas.

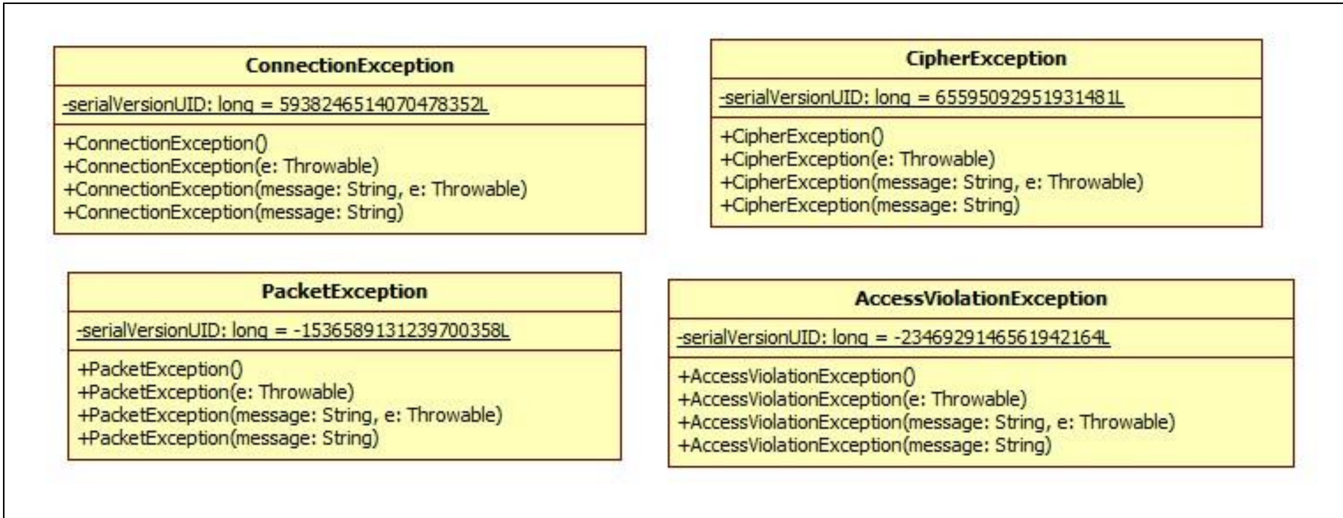


Figura 13 – Diagrama de classe do pacote `connection.exception`

Todas as classes de exceção do pacote `com.age.transport.connection.exception` estendem a classe `Exception`, que é utilizada em Java para representar exceções checadas e implementam todos os construtores padrões da classe pai.

3.2.2.5 Pacote com.age.transport.connection.wireless.tcp

O pacote `com.age.transport.connection.wireless.tcp` (Figuras 14 e 15) é o pacote onde está localizada a implementação da conexão TCP para cliente (Figura 14) e para servidor (Figura 15).

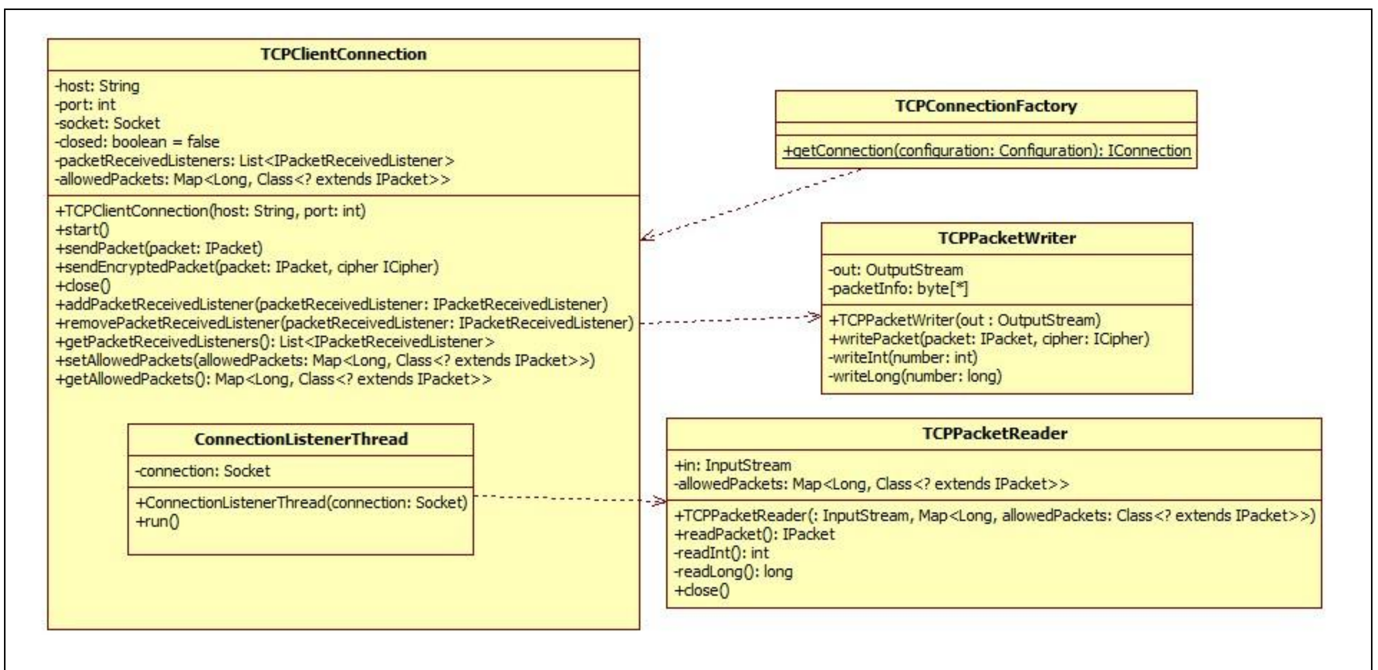


Figura 14 – Diagrama de classe para o cliente TCP do pacote `connection.wireless.tcp`

A classe TCPConnectionFactory é uma implementação do padrão *abstract factory* para decidir qual implementação de conexão TCP utilizar, se a implementação cliente, ou a implementação servidor. Esta classe será chamada pela classe ConnectionFactory, após perceber que a conexão é do tipo TCP. A classe ConnectionFactory passa a responsabilidade de obter a conexão TCP mais adequada para a classe TCPConnectionFactory.

A classe TCPPacketWriter é uma implementação da interface IPacketWriter para conexões TCP. Ela é responsável por enviar os pacotes através da camada TCP. Ela possui o atributo out que é o canal de saída de informações da conexão TCP, o atributo packetInfo que armazena o pacote transformado em bytes. O método writeInt e o método writeLong são métodos auxiliares para transformação de valores int e long em bytes. O método writePacket é responsável por transformar o pacote em bytes, aplicar a criptografia e enviar o pacote.

A classe TCPPacketReader é uma implementação da interface IPacketReader para conexões TCP. Ela é responsável por receber os pacotes pela camada TCP. Ela possui o atributo in do qual são lidas as informações recebidas, o atributo allowedPackets que armazena os pacotes que podem ser recebidos, o método readInt e o método readLong que são métodos auxiliares que transformam bytes em int e long, o método readPacket que decripta os bytes recebidos e transforma os bytes em um pacote, o método close é implementado sem executar nenhuma instrução devido ao leitor de pacotes TCP não necessitar executar nenhuma finalização.

A classe TCPClientConnection é a implementação da interface IConnection para clientes TCP. Seu construtor possui os parâmetros host e port que devem possuir o IP e a porta do servidor TCP. Estes parâmetros são armazenados no atributo host e port. O atributo socket possui o *socket* aberto para o servidor, o atributo closed é uma *flag* para indicar quando a conexão foi finalizada, o atributo packetReceivedListeners armazena os *listeners* que irão escutar os pacotes recebidos por esta conexão, o atributo allowedPacket possui o mapa de pacotes permitidos. Os métodos são todos implementações dos métodos definidos na interface, portanto, possuem a função previamente citada na interface.

A classe interna ConnectionListenerThread é uma Thread responsável por chamar constantemente o método readPacket de classe TCPPacketReader e disparar os eventos de pacote recebido aos *listeners*. Caso seja percebido que a conexão foi perdida ao executar o método readPacket a classe TCPPacketReader lançará uma ConnectionException, esta

será tratada pelo *listener* que irá finalizar a conexão com o transmissor que perdeu a conexão.

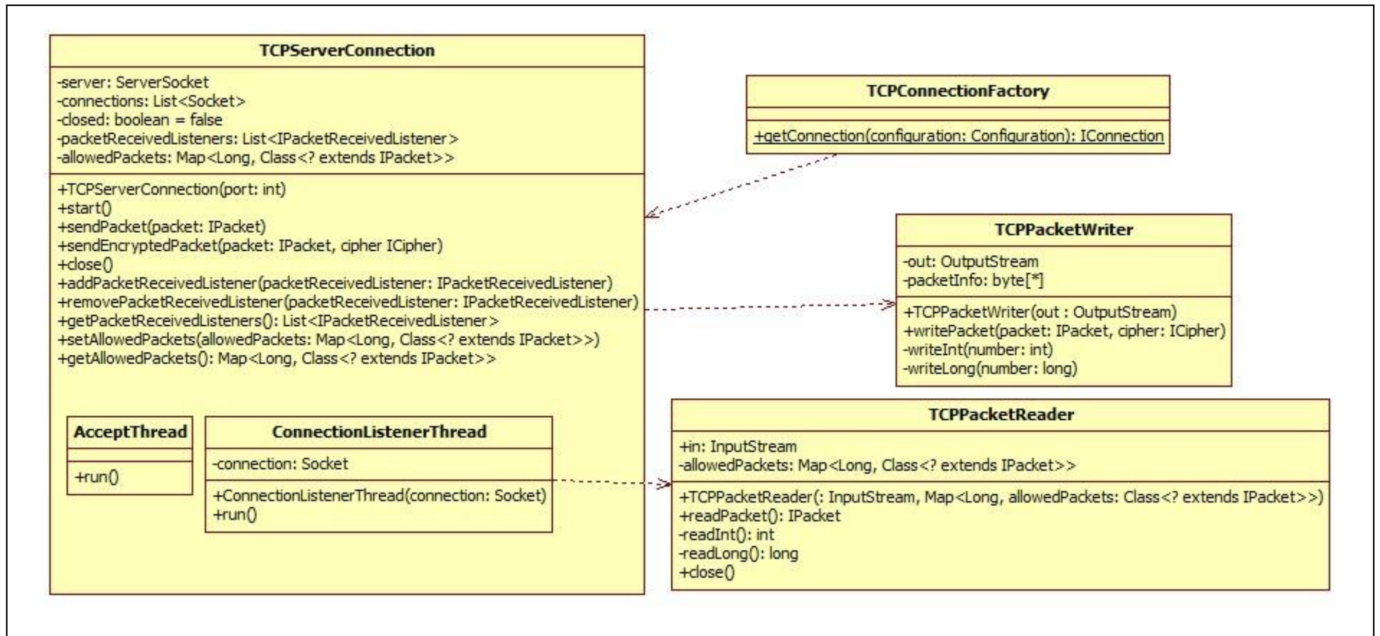


Figura 15 – Diagrama de classe para o servidor TCP do pacote `connection.wireless.tcp`

A classe `TCPServerConnection` é a implementação da interface `IConnection` para servidores TCP. Esta classe possui apenas o parâmetro `port` no construtor que deve conter a porta que o servidor utilizará no computador local, os demais métodos, atributos e a classe interna `ConnectionListenerThread` seguem as mesmas funções da classe `TCPClientConnection` com exceção do atributo `connection` que armazena todas as conexões ativas de clientes.

A classe interna `AcceptThread` é uma `Thread` responsável por aceitar as conexões de clientes ao servidor. Esta Thread faz uso da classe ServerSocket que é uma classe contida na API do Java responsável por adminitrar o recebimento de informações de todos os clientes, desta forma o framework não precisa efetuar a verificação de recebimento de dados em cada cliente. Ele precisa apenas verificar na classe ServerSocket.

3.2.2.6 Pacote com.age.transport.connection.wireless.udp

O pacote `com.age.transport.connection.wireless.udp` (Figuras 16 e 17) é o pacote onde está localizada a implementação da conexão UDP para cliente (Figura 16) e para servidor (Figura 17).

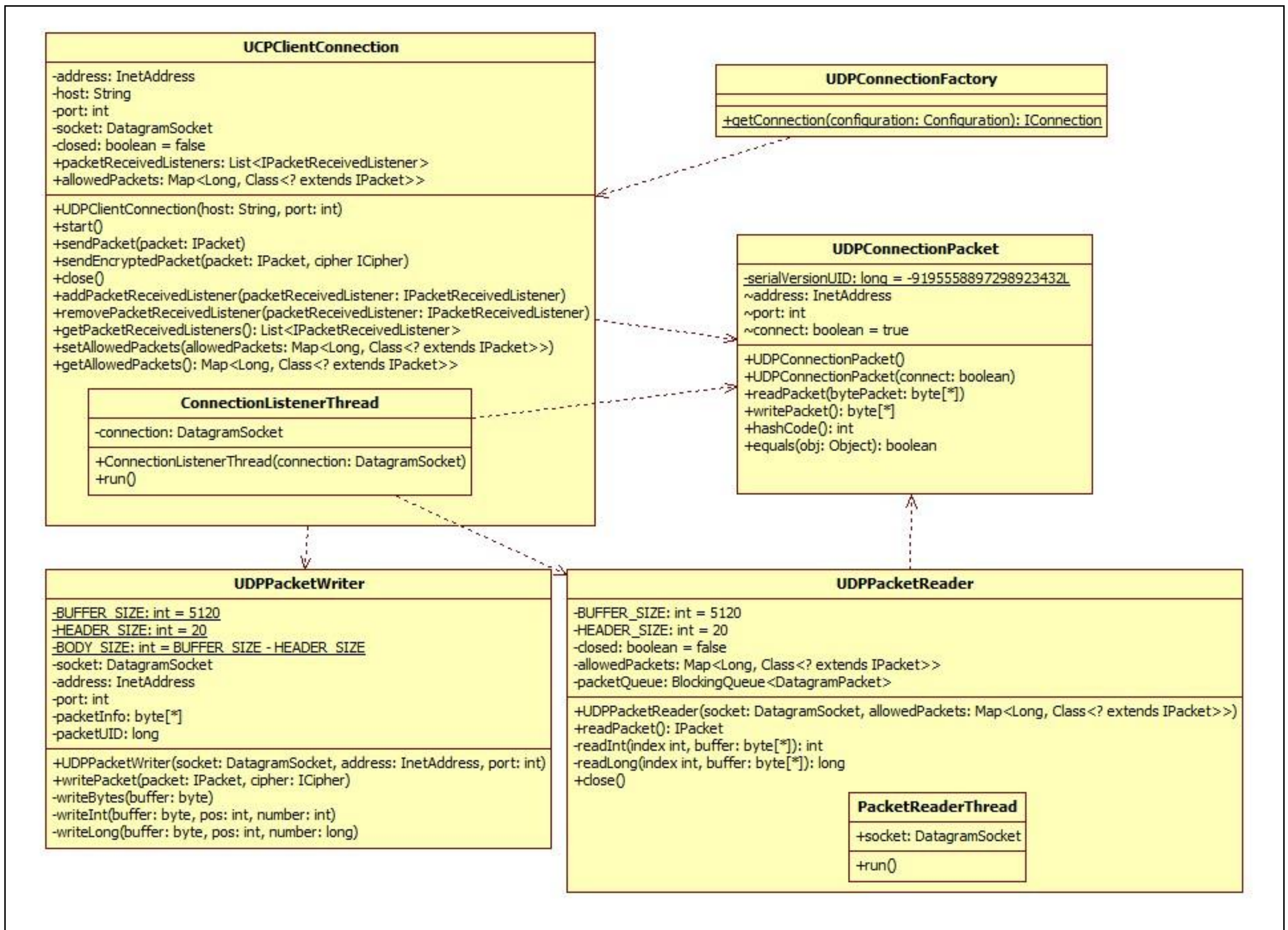


Figura 16 - Diagrama de classe para o cliente UDP do pacote `connection.wireless.udp`

A classe `UDPConnectionFactory` é uma implementação de *abstract factory* que tem o objetivo de decidir qual implementação de UDP utilizar, se é a implementação para cliente ou para servidor.

A classe `UDPPacketWriter` é a implementação da interface `IPacketWriter` para conexões UDP. Responsável por escrever os pacotes nas conexões pelo protocolo UDP, o escritor de pacotes UDP possui as constantes `BUFFER_SIZE`, `HEADER_SIZE` e `BODY_SIZE` que armazenam o tamanho do buffer, o tamanho do cabeçalho e o tamanho do corpo do pacote respectivamente, o atributo `socket` que armazena o socket do receptor, os atributos `address` e `port` que armazenam o endereço IP e a porta do receptor, o atributo `packetInfo` que armazena o pacote em `bytes`, e o atributo `packetUID` que armazena o id único do pacote. Esta classe possui o método `writePacket` que transforma o pacote em bytes, encripta e envia o pacote. Este método utiliza os métodos auxiliares `writeBytes`, `writeInt` e `writeLong` que escrevem respectivamente `bytes`, `int` e `long` no `array` que será enviado pelo datagrama.

A classe `UDPPacketReader` é a implementação da interface `IPacketReader` para conexões UDP. Responsável por receber os pacotes nas conexões pelo protocolo UDP, ele

possui a constante BUFFER_SIZE que armazena o tamanho do buffer, a constante HEADER_SIZE que armazena o tamanho do cabeçalho, o atributo closed que identifica se o leitor está fechado ou não, o atributo allowedPackets que possui a lista de pacotes permitidos, e o atributo packetQueue que é uma fila *thread-safe* responsável por receber os datagramas recebidos pela PacketReaderThread. Esta classe também possui os métodos readInt e readLong que são utilizados para ler variáveis do tipo int e long do datagrama, o método readPacket que decripta e transforma os bytes em um pacote novamente, e o método close que finaliza a Thread PacketReaderThread.

A classe interna PacketReaderThread é uma Thread responsável por ficar constantemente recebendo os datagramas UDP. Após ler o datagrama o mesmo é armazenado na fila de datagramas da classe UDPPacketReader.

A classe UDPConnectionPacket implementa a interface IPacket, é utilizada para sinalizar conexões e desconexões no protocolo UDP, visto que este pacote é utilizado apenas pelas conexões UDP, este foi mantido no pacote UDP. O pacote também é incluído automaticamente na lista de pacotes permitidos, fazendo com que o desenvolvedor não tenha que se preocupar com o pacote ao utilizar conexões UDP.

A classe UDPClientConnection é a implementação da interface IConnection para clientes UDP. Ela possui o atributo address que é obtido através do atributo host ao iniciar a conexão, o atributo socket armazena o DatagramSocket deste cliente e os demais atributos que tem o mesmo objetivo dos atributos explicados anteriormente na conexão TCP. Os métodos são todos implementações dos métodos da interface IConnection para a conexão UDP cliente.

A classe interna ConnectionListenerThread também segue a mesma funcionalidade das classes internas utilizadas nas conexões TCP cliente e servidor só que com algumas particularidades da conexão UDP cliente.

A classe UDPServerConnection é a implementação de interface IConnection para servidores UDP. Ela possui o atributo port que define a porta utilizada pelo servidor, o atributo socket que possui o DatagramSocket do servidor, o atributo connections é uma coleção de UDPConnetionPacket onde não pode haver mais de um UDPConnectionPacket com o mesmo endereço IP e porta. Isto é garantido pelo fato da coleção ser do tipo *set*.

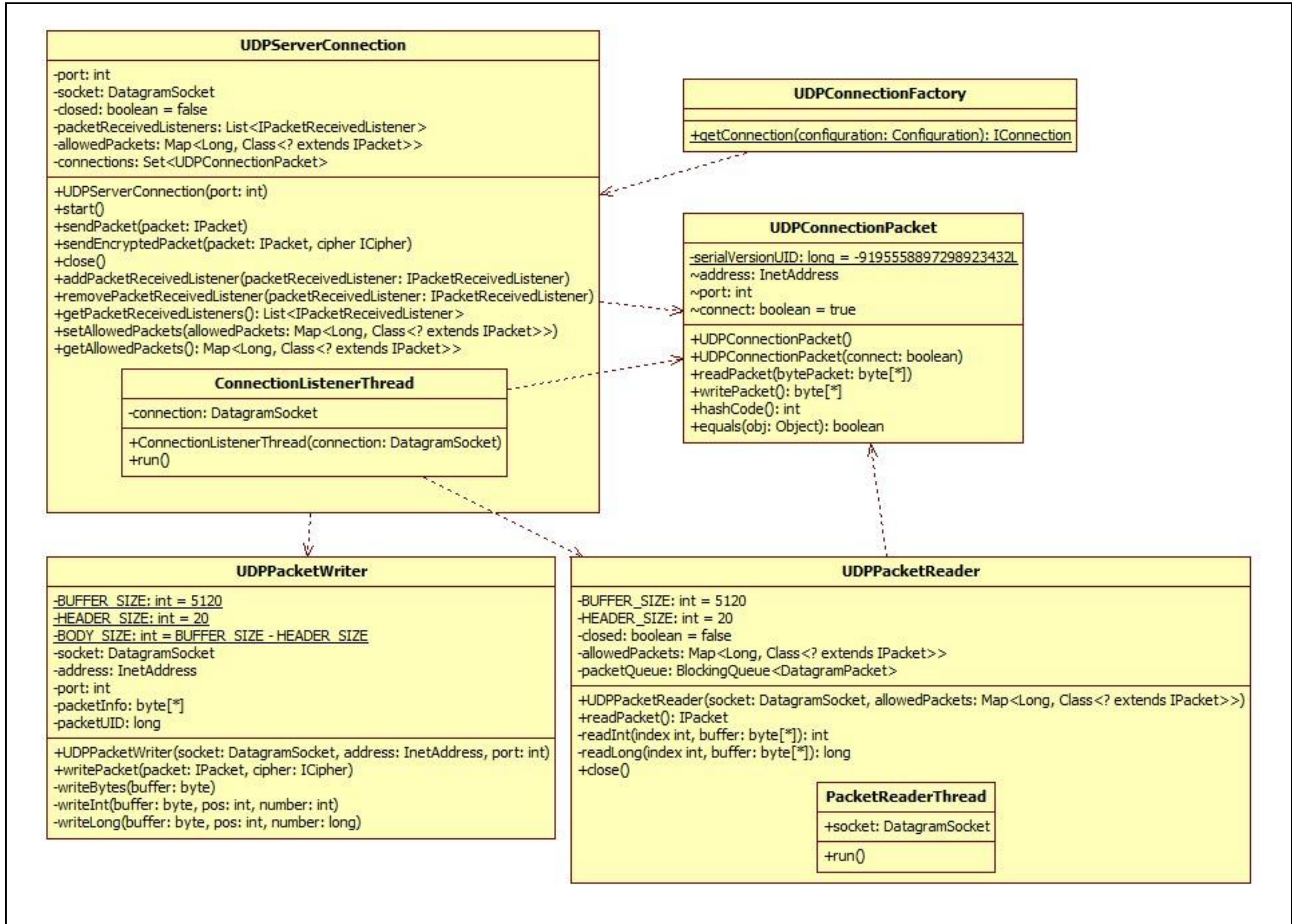


Figura 17 - Diagrama de classe para o servidor UDP do pacote `connection.wireless.udp`

3.2.3 Diagramas de sequência

Os diagramas de sequência (Figura 18 e 19) mostram como o desenvolvedor utilizará o framework na sua aplicação cliente, desde o momento da configuração até o momento do envio da primeira mensagem ao servidor para conexões do tipo TCP e UDP.

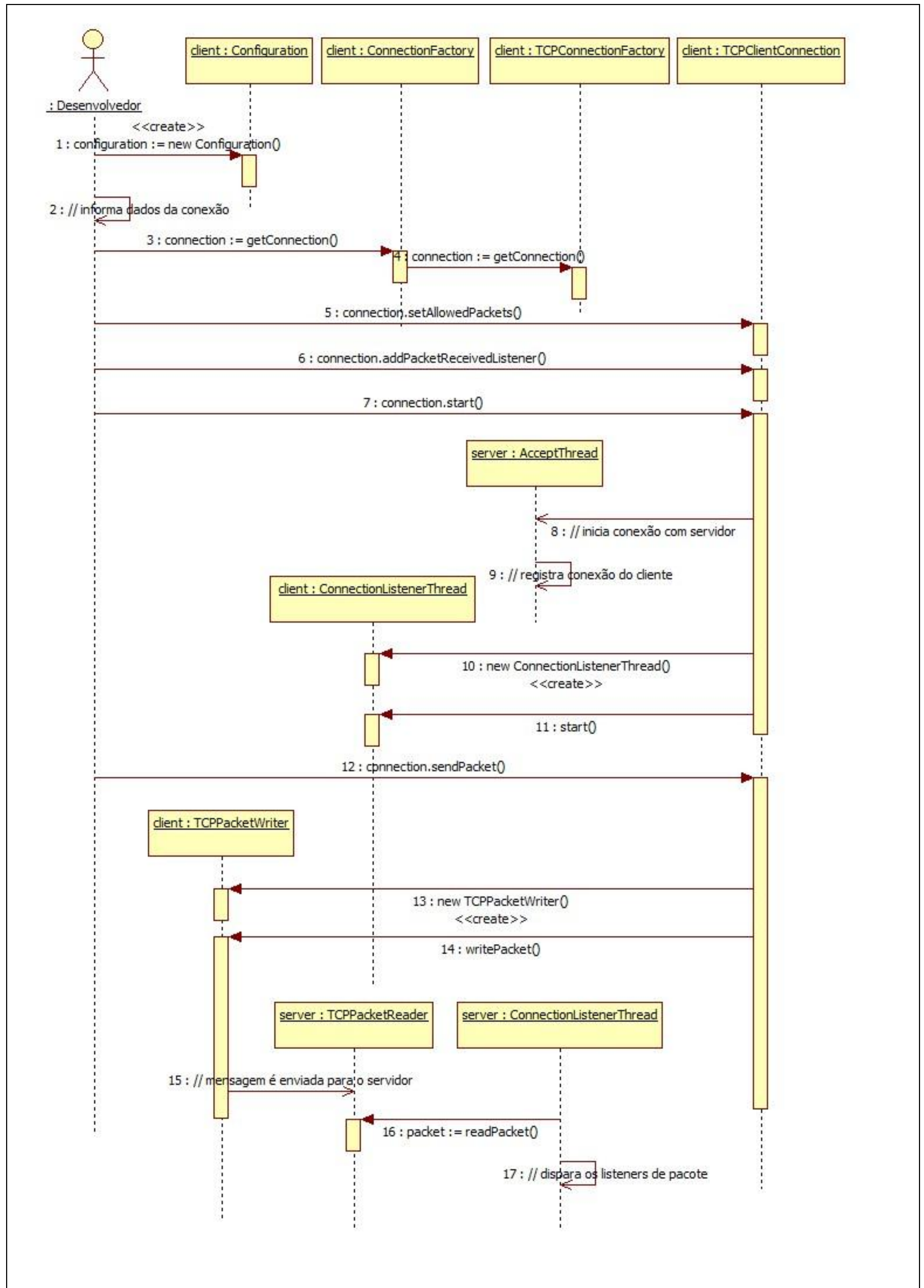


Figura 18 – Diagrama de sequência do cliente TCP

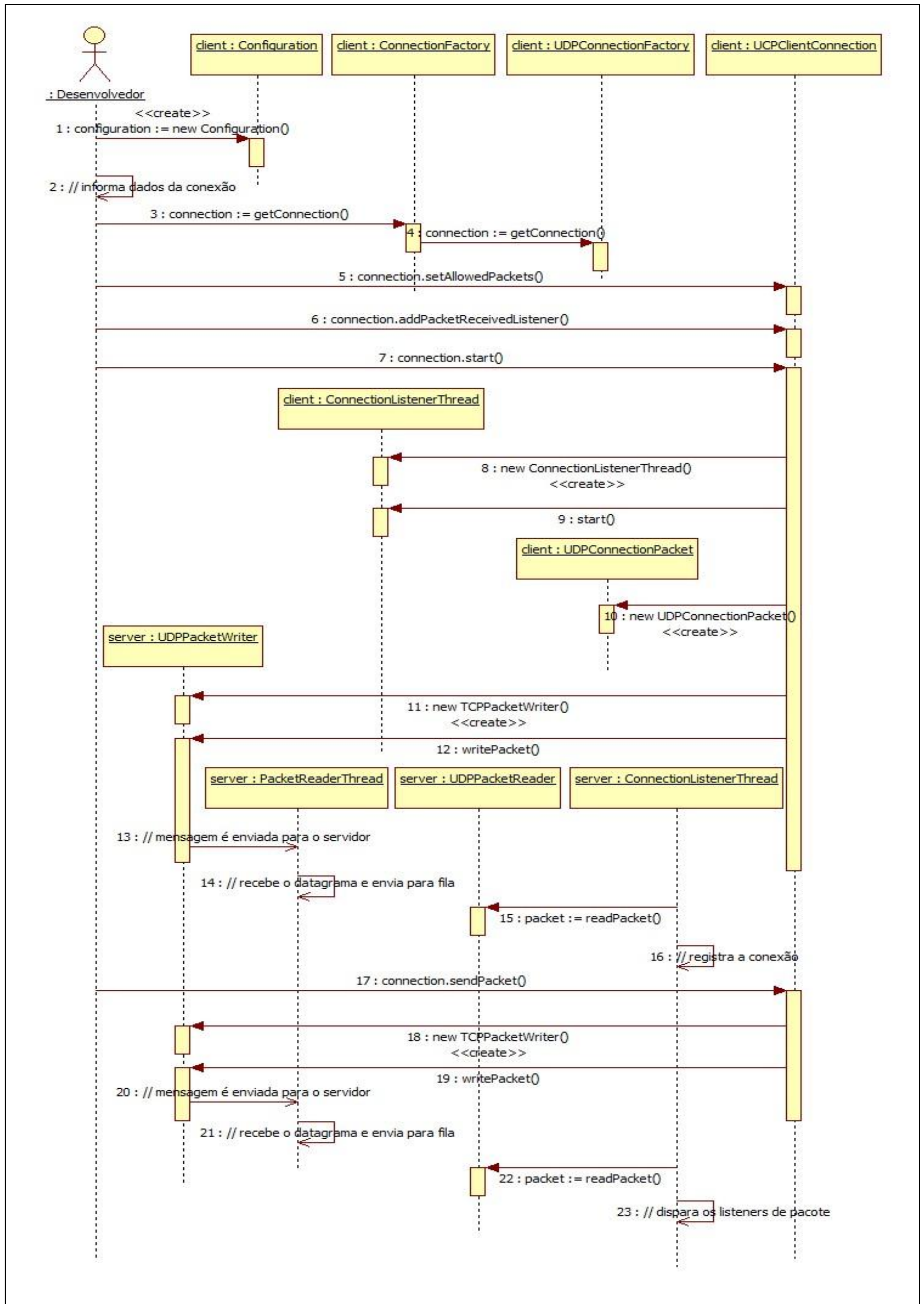


Figura 19 – Diagrama de sequência do cliente UDP

Como pode ser observado, as chamadas realizadas pelo desenvolvedor são exatamente idênticas em ambas as conexões. Como o método `getConnection` da classe `ConnectionFactory` retorna um `IConnection`, o desenvolvedor pode criar uma tela para realização da configuração e utilizar sua conexão sem nem mesmo saber se o cliente do seu sistema optou por utilizar uma conexão UDP ou TCP.

Por mais que as chamadas do desenvolvedor sejam iguais, há varias diferenças entre as duas conexões, como a forma que a conexão é realizada, e a forma em que os pacotes são recebidos.

Para iniciar uma conexão pela conexão TCP a conexão cliente inicia um *socket* que fará com que a classe `AcceptThread` receba uma solicitação de conexão e coloque essa nova conexão na lista de clientes, após isso a thread `ConnectionListenerThread` é iniciada para escutar os pacotes que são enviados pelo servidor. Na conexão UDP diferentemente da conexão TCP, o servidor UDP pode receber pacotes sem previamente aceitar uma conexão cliente. Para prevenir que pacotes de clientes não registrados se comuniquem com o servidor foi implementado um pacote responsável por criar o mesmo comportamento da conexão TCP. O pacote `UDPConnectionPacket` é o único pacote que pode ser recebido pelo servidor UDP até o momento em que o cliente esteja registrado pelo mesmo. Sua função é exatamente esta, registrar a conexão cliente no servidor.

Para receber um pacote o servidor TCP simplesmente obtém os bytes da conexão transforma os bytes em pacotes e envia para os listeners. Já na conexão UDP como a conexão não armazena os dados recebidos em um buffer, é utilizada a thread `PacketReaderThread` para constantemente receber os pacotes na forma de datagramas e armazená-los em uma fila para futuramente ele serem consumidos pelo leitor de pacotes que transformará os `bytes` do datagrama em um pacote e os enviará para os listeners. Devido à necessidade desta segunda thread a interface do leitor de pacote houve a inclusão do método `close` que faz a finalização da *thread*.

Também pode-se perceber nos diagramas de sequência que a conexão UDP, por mais que seja relativamente mais rápida na transmissão de dados, exige muito mais trabalho de implementação, conforme já havia sido comentado no seção 2.2.1.5 - TCP X UDP.

3.3 IMPLEMENTAÇÃO

A seguir são mostradas as técnicas e ferramentas utilizadas e a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O desenvolvimento do *framework* foi feito inteiramente na linguagem Java sem necessidade de nenhuma classe da plataforma Android, tornando-o assim utilizável como cliente ou servidor tanto em aplicativos Java como em aplicativos na plataforma Android. Já o desenvolvimento dos aplicativos de teste foi feito utilizando as bibliotecas da plataforma Android.

O ambiente de desenvolvimento utilizado foi o Eclipse com o conjunto de *plugins* do Android Development Tools (ADT), que possui uma série de recursos para criação, execução e depuração de aplicações Android de forma a facilitar o desenvolvimento.

Para a execução e depuração foram utilizados dois dispositivos móveis e um notebook. O primeiro dispositivo utilizado foi o *smartphone* Motorola Milestone que possui um processador de 600Mhz e 256MB de memória RAM. O segundo dispositivo foi o *tablet* Motorola XOOM que possui um processador de 1.000Mhz e 1GB de memória RAM. Por fim, o notebook utilizado foi um DELL Vostro com processador Core i5 com dois núcleos de 2.66Ghz e 4GB de memória RAM.

3.3.2 O *framework*

A seguir serão apresentados alguns fragmentos de códigos elencados por importância para aplicação e complexidade.

3.3.2.1 ConnectionFactory

O primeiro fragmento a ser detalhado diz respeito a como a classe

`ConnectionFactory`, a classe `TCPConnectionFactory` e a classe `UDPConnectionFactory`, que usam o padrão de projeto *abstract factory*, se comunicam para obter a implementação da interface `IConnection` mais apropriada para a configuração criada.

A classe `ConnectionFactory` escolherá a partir da configuração criada qual protocolo será utilizado,. Isto é realizado no método `getConnection` (Quadro 7).

```
public static IConnection getConnection(Configuration configuration)
throws ConnectionException
{
	switch(configuration.getProtocol()){
	case TCP:
		return TCPConnectionFactory.getConnection(configuration);
	case UDP:
		return UDPConnectionFactory.getConnection(configuration);
	}
	throw new ConnectionException("Connection type not found...");
}
```

Quadro 7 – Método `getConnection` da classe `ConnectionFactory`

A classe `TCPConnectionFactory` é responsável por definir se a conexão TCP será do tipo cliente ou servidor (Quadro 8).

```
public static IConnection getConnection(Configuration configuration)
{
	if (configuration.isHost()){
		return new TCPServerConnection(configuration.getPort());
	}
	return new TCPClientConnection(configuration.getHostName(),
	                               configuration.getPort());
}
```

Quadro 8 - Método `getConnection` da classe `TCPConnectionFactory`

Por último a classe `UDPConnectionFactory` é responsável por definir se a conexão UDP será do tipo cliente ou servidor (Quadro 9).

```
public static IConnection getConnection(Configuration configuration)
throws ConnectionException
{
	if (configuration.isHost()){
		return new UDPServerConnection(configuration.getPort());
	}
	return new UDPClientConnection(configuration.getHostName(),
	                               configuration.getPort());
}
```

Quadro 9 - Método `getConnection` da classe `UDPConnectionFactory`

É interessante observar que caso o desenvolvedor opte por utilizar somente um tipo de protocolo, seja ele TCP ou UDP, ele pode utilizar a *factory* do tipo de protocolo escolhido diretamente, sem ter que obrigatoriamente utilizar a classe `ConnectionFactory`.

3.3.2.2 Id único de pacotes e encriptadores

Um problema encontrado durante o desenvolvimento é como identificar os pacotes e encriptadores utilizados tanto no receptor como no transmissor da informação de forma que:

a) o desenvolvedor seja obrigado a informá-lo para suas implementações de pacotes e encriptadores;
b) o identificador pudesse ser obtido rapidamente.

Foi decidido então fazer a interface `IPacket` (interface base dos pacotes) e a interface `ICipher` (interface base dos encriptadores) estenderem a interface `Serializable` do Java. Utilizando este artifício a própria IDE do desenvolvedor obriga-o a informar o atributo estático do tipo `long serialVersionUID` para qualquer classe que implemente `IPacket` ou `ICipher`. Conforme a especificação da interface, para qualquer classe que implemente `Serializable` caso o atributo `serialVersionUID` não seja declarado pelo desenvolvedor o compilador gera um `serialVersionUID` para a classe baseado nas características da mesma. Outro fator importante é que a maioria da IDEs Java cria o atributo `serialVersionUID` para o desenvolvedor utilizando um gerador de id único, tornando assim o valor deste atributo único para cada classe a menos que o próprio desenvolvedor informe valores duplicados.

Devido a este id ser privado para a aplicação, foi utilizado reflexão conforme mostrado no Quadro 10.

```
// Obtém o campo serialVersionUID
Field field = packet.getClass().getDeclaredField("serialVersionUID");
// Torna o campo acessível
field.setAccessible(true);
// Obtém o valor do campo
packetUID = field.getLong(null);
```

Quadro 10 – Exemplo de obtenção do `serialVersionUID` de um pacote

3.3.2.3 Encriptação AES128

Um fator interessante na implementação do encriptador AES128 é que devido ao Java já possuir uma implementação do encriptador, foi necessário apenas utilizar esta implementação na classe `CipherAES`, tornando a implementação do encriptador extremamente simples.

O Quadro 11 mostra a implementação do método de encriptação.

```
@Override
public byte[] encode(byte[] information) throws CipherException{
      try {
          // Obtém uma instância do gerador de chaves AES
          KeyGenerator kgen = KeyGenerator.getInstance("AES");
          // Define que a chave terá 128bits
          kgen.init(128);

          // Gera a chave de 128bits
          SecretKey skey = kgen.generateKey();
          // Obtém um array com a chave gerada
          byte[] raw = skey.getEncoded();

          // Cria uma especificação de chave que será utilizada na
          //encriptação
          SecretKeySpec skeySpec = new SecretKeySpec(raw, "AES");

          // Obtém um instância de encriptador AES
          Cipher cipher = Cipher.getInstance("AES");
          // Informa a chave e que será utilizada a encriptação ao
          //encriptador
          cipher.init(Cipher.ENCRYPT_MODE, skeySpec);

          // Obtém os dados encriptados
          byte[] encripted = cipher.doFinal(information);
          // Cria um array de retorno
          byte[] ret = new byte[encripted.length + raw.length];
          // Incluí a chave no array de retorno
          System.arraycopy(raw, 0, ret, 0, raw.length);
          // Incluí os dados encriptado no array de retorno
          System.arraycopy(encripted, 0, ret,
                          raw.length, encripted.length);

          return ret;
      } catch (NoSuchAlgorithmException e) {
            throw new CipherException(e);
      } catch (NoSuchPaddingException e) {
            throw new CipherException(e);
      } catch (IllegalBlockSizeException e) {
            throw new CipherException(e);
      } catch (BadPaddingException e) {
            throw new CipherException(e);
      } catch (InvalidKeyException e) {
            throw new CipherException(e);
      }
}
```

Quadro 11 – Encriptação AES128

O Quadro 12 mostra a implementação do método de decriptação.

```
@Override
public byte[] decode(byte[] information) throws CipherException{

	try {
		// Obtém a chave de encriptação
		SecretKeySpec skeySpec = new SecretKeySpec(information, 0,
                                                  16, "AES");

		// Obtém um instância de encriptador AES
		Cipher cipher = Cipher.getInstance("AES");

		// Informa a chave e que será utilizada a decriptação ao
		//encriptador
		cipher.init(Cipher.DECRYPT_MODE, skeySpec);

		// Retorna o array de bytes decriptados
		return cipher.doFinal(information, 16, information.length - 16);
	} catch (NoSuchAlgorithmException e) {
		throw new CipherException(e);
	} catch (NoSuchPaddingException e) {
		throw new CipherException(e);
	} catch (InvalidKeyException e) {
		throw new CipherException(e);
	} catch (IllegalBlockSizeException e) {
		throw new CipherException(e);
	} catch (BadPaddingException e) {
		throw new CipherException(e);
	}
}
```

Quadro 12 – Decriptação AES128

3.3.2.4 *Listener* receptor de pacote

O *listener* receptor de pacote é uma implementação do padrão de projeto *observer*. Utilizando este padrão o *listener* de pacote permite que o desenvolvedor se cadastre na conexão para receber os pacotes pela mesma. Para se cadastrar o desenvolvedor deve criar uma classe que extenda a interface `IPacketReceivedListener` e adicioná-la uma instância desta classe a conexão pelo método `addPacketReceivedListener`. Depois de cadastrado na conexão, cada vez que a *thread* `ConnectionListenerThread` da conexão receber um novo pacote ela irá chamar o método `onReceive` da instância de `IPacketReceivedListener` adicionada à conexão. Isto é realizado pelo trecho de código do Quadro 13 que está contido no método *run* da *thread* `ConnectionListenerThread` de todas as conexões.

```
// lê o pacote
packet = packetReader.readPacket();

// Dispara eventos de pacote recebido
for (int i = 0; i < packetReceivedListeners.size(); i++) {
	packetReceivedListeners.get(i).onReceive(packet);
}
```

Quadro 13 – Execução do `onReceive` dos *listeners* de pacote

3.3.2.5 Escritor de pacotes TCP

O escritor de pacote TCP é quem faz a transformação do pacote em `bytes` e a sua transmissão pela rede utilizando protocolo TCP, isto é realizado no método `writePacket` da classe `TCPPacketWriter` (Quadro 14). Após enviar um pacote este método termina, mas é chamado novamente pela thread `ConnectionListenerThread`.

São enviados pela comunicação o uid do encriptador, o uid do pacote, a quantidade de `bytes` da informação e a informação.

```
public void writePacket(IPacket packet, ICipher cipher)
throws PacketException, CipherException
{
	try {
		packetInfo = packet.writePacket();

		// Escreve o uid do encriptador e encripta os bytes
		// ou escreve 0 se não houver encriptador
		if (cipher != null) {
			Field field = cipher.getClass()
						.getDeclaredField("serialVersionUID");
			field.setAccessible(true);
			writeLong(field.getLong(null));
			packetInfo = cipher.encode(packetInfo);
		} else {
			writeLong(0);
		}

		// Escreve o UID do pacote
		Field field = packet.getClass()
						.getDeclaredField("serialVersionUID");
		field.setAccessible(true);
		writeLong(field.getLong(null));

		// Escreve o tamanho do pacote
		writeInt(packetInfo.length);

		// Escreve a informação
		out.write(packetInfo);
	} catch (IllegalArgumentException e) {
		throw new PacketException(e);
	} catch (SecurityException e) {
		throw new PacketException(e);
	} catch (IllegalAccessException e) {
		throw new PacketException(e);
	} catch (NoSuchFieldException e) {
		throw new PacketException(e);
	} catch (IOException e) {
		throw new PacketException(e);
	}
}
```

Quadro 14 – Escritor de pacotes TCP

3.3.2.6 Escritor de pacotes UDP

O escritor de pacote UDP é quem faz a transformação do pacote em `bytes` e a sua transmissão pela rede utilizando protocolo UDP. Isto é realizado no método `writePacket` da classe `UDPPacketWriter` (Quadro 15). Após enviar um pacote este método termina, mas é chamado novamente pela thread `ConnectionListenerThread`.

Também são enviados pela comunicação o uid do encriptador, o uid do pacote, a

quantidade de `bytes` da informação e a informação. Diferentemente do protocolo TCP em que o limite de tamanho do pacote é o tamanho máximo da memória RAM destinado a aplicação, no protocolo UDP os pacotes podem possuir no máximo 5KB. Isto ocorre, pois caso contrário os pacotes teriam que ser divididos em partes ao serem enviados pela rede. A implementação desta funcionalidade não foi realizada por ser uma funcionalidade relativamente complexa de se implementar e não agregar vantagens ao *framework*, além de piorar muito o desempenho nas transmissões UDP.

```
public void writePacket(IPacket packet, ICipher cipher)
throws PacketException, CipherException
{
	try {
		byte[] buffer = new byte[BUFFER_SIZE];
		packetInfo = packet.writePacket();
		// Escreve o uid do encriptador e encripta os bytes
		// ou escreve 0 se não houver encriptador
		if (cipher != null) {
			Field field = cipher.getClass()
					.getDeclaredField("serialVersionUID");
			field.setAccessible(true);
			writeLong(buffer, 0, field.getLong(null));
			packetInfo = cipher.encode(packetInfo);
		} else writeLong(buffer, 0, 0);
		// Verifica se o pacote atingiu o limite de tamanho
		if (packetInfo.length > BODY_SIZE){
			throw new PacketException("UDP packets can't have more"
					+ "than 5120 bytes.");
		}
		// Escreve o uid do pacote
		Field field = packet.getClass()
				.getDeclaredField("serialVersionUID");
		field.setAccessible(true);
		packetUID = field.getLong(null);
		writeLong(buffer, 8, packetUID);
		// Escreve o tamanho do pacote
		writeInt(buffer, 16, packetInfo.length);
		// Escreve a informação
		writeBytes(buffer);
		// Envia o buffer
		DatagramPacket dataPacket = new DatagramPacket(buffer,
					buffer.length, address, port);
		socket.send(dataPacket);
	} catch (IOException e) {
		throw new PacketException();
	} catch (IllegalArgumentException e) {
		throw new PacketException();
	} catch (SecurityException e) {
		throw new PacketException();
	} catch (IllegalAccessException e) {
		throw new PacketException();
	} catch (NoSuchFieldException e) {
		throw new PacketException();
	}
}
```

Quadro 15 – Escritor de pacotes UDP

3.3.2.7 Leitor de pacotes TCP

O leitor de pacotes TCP faz a transformação dos bytes recebidos pela rede para pacotes novamente. Isto é realizado pelo método `readPacket` da classe `TCPPacketReader` (Quadro 16).

```
public IPacket readPacket()
throws PacketException, AccessViolationException, ConnectionException,
CipherException
{
	try {
		// Obtém o id do cipher
		long ciphered = readLong();
		// Obtém o id do pacote
		long packetUID = readLong();
		// Obtém a quantidade de bytes do pacote
		int packetSize = readInt();

		// Copia os bytes da informação para um novo array de bytes
		byte[] bytePacket = new byte[packetSize];
		in.read(bytePacket);

		// Decodifica os bytes se estes estiverem encriptados
		if (ciphered != 0){
			ICipher cipherClass = CipherMap.getInstance()
						.getRegisteredCipher(ciphered);
			bytePacket = cipherClass.decode(bytePacket);
		}

		// Verifica se a conexão tem permissão para
		// receber pacotes com este id
		Class<? extends IPacket> packetClass = allowedPackets
							.get(packetUID);
		if (packetClass == null) {
			throw new AccessViolationException("Packet UID not"
								+ "allowed.");
		}
		// Cria uma instância do pacote e chama o readPacket
		IPacket packet = packetClass.newInstance();
		packet.readPacket(bytePacket);
		return packet;
	} catch (IOException e) {
		throw new ConnectionException("Connection end.", e);
	} catch (InstantiationException e) {
		throw new PacketException(e);
	} catch (IllegalAccessException e) {
		throw new PacketException(e);
	}
}
```

Quadro 16 – Leitor de pacote TCP

3.3.2.8 Leitor de pacotes UDP

O leitor de pacotes UDP faz a transformação dos bytes recebidos pela rede para pacotes novamente. A leitura dos datagramas UDP recebidos pela rede é realizada na *thread* `PacketReaderThread` a qual, após ler o datagrama, adiciona este na fila *thread-safe* `packetQueue` (Quadro 17). Esta *thread* executa enquanto o `UDPPacketReader` não for finalizado pelo método `close` e se faz necessária, pois em conexões UDP se o datagrama não for lido no momento em que é enviado pelo transmissor ele é descartado.

```
@Override
public void run()
{
	while (!closed) {
		try {
			// Recebe o datagrama
			DatagramPacket dataPacket = new DatagramPacket(
						new byte[BUFFER_SIZE], BUFFER_SIZE);
			socket.receive(dataPacket);
			// Coloca-o na fila thread-safe
			packetQueue.put(dataPacket);
		} catch (IOException e) {
			e.printStackTrace();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}
	}
}
```

Quadro 17 – *Thread* leitora de datagramas

Após ser adicionado na fila o datagrama agora precisa ser transformado em pacote novamente. Isto é realizado pelo método `readPacket` da classe `UDPPacketReader` (Quadro 18).

```java
public IPacket readPacket()
throws PacketException, AccessViolationException, CipherException
{
	try {
		DatagramPacket datagram;
		while (true) {
			try { // Obtém um datagrama da fila thread-safe
				datagram = packetQueue.take();
				break;
			} catch (InterruptedException e) {e.printStackTrace();}
		}
		// Obtém os bytes do datagrama
		byte[] buffer = datagram.getData();
		// Obtém o id do cipher
		long ciphered = readLong(0, buffer);
		// Obtém o id do pacote
		long packetUID = readLong(8, buffer);
		// Obtém a quantidade de bytes do pacote
		int packetSize = readInt(16, buffer);
		// Copia os bytes da informação para um novo array de bytes
		byte[] bytePacket = new byte[packetSize];
		System.arraycopy(buffer, HEADER_SIZE,
					bytePacket, 0, packetSize);
		// Decodifica os bytes se estes estiverem encriptados
		if (ciphered != 0) {
			ICipher cipherClass = CipherMap.getInstance()
						.getRegisteredCipher(ciphered);
			bytePacket = cipherClass.decode(bytePacket);
		}
		// Verifica se a conexão tem permissão para
		// receber pacotes com este id
		Class<? extends IPacket> packetClass = allowedPackets
								.get(packetUID);
		if (packetClass == null) {
			throw new AccessViolationException("Packet UID not"
								+ "allowed.");
		}
		// Cria uma instância do pacote e chama o readPacket
		IPacket packet = packetClass.newInstance();
		packet.readPacket(bytePacket);
		// Se o pacote for do tipo conexão popula ele manualmente
		if (packetClass == UDPConnectionPacket.class){
			UDPConnectionPacket connectionPacket =
						(UDPConnectionPacket) packet;
			connectionPacket.address = datagram.getAddress();
			connectionPacket.port = datagram.getPort();
		}
		return packet;
	} catch (IllegalArgumentException e) {
		throw new PacketException(e);
	} catch (InstantiationException e) {
		throw new PacketException(e);
	} catch (IllegalAccessException e) {
		throw new PacketException(e);
	} catch (IOException e) {
		throw new PacketException(e);
	}
}
```

Quadro 18 – Leitor de pacotes UDP

3.3.3 Operacionalidade do *framework*

A operacionalidade do *framework* será apresentada através da apresentação de dois aplicativos de teste que fazem uso do *framework*.

3.3.3.1 Aplicativo de teste 1: protótipo de *messenger*

O aplicativo de protótipo de *messenger* possui basicamente duas telas. A tela principal possui uma lista com as mensagens recebidas e enviadas e uma área de digitação onde se pode enviar mensagens escritas nesta área através do botão *send* (Figura 20).

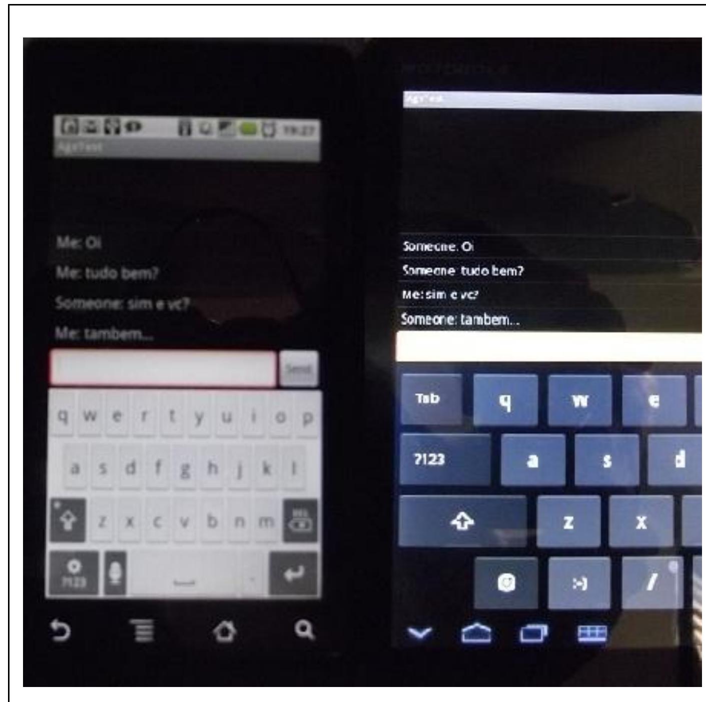


Figura 20 – Tela principal do *messenger*

A outra tela é a tela de configuração. Esta tela serve para o usuário da aplicação definir como será realizada a comunicação da aplicação, se será UDP ou TCP e se este aplicativo funcionará como servidor ou cliente (Figura 21).

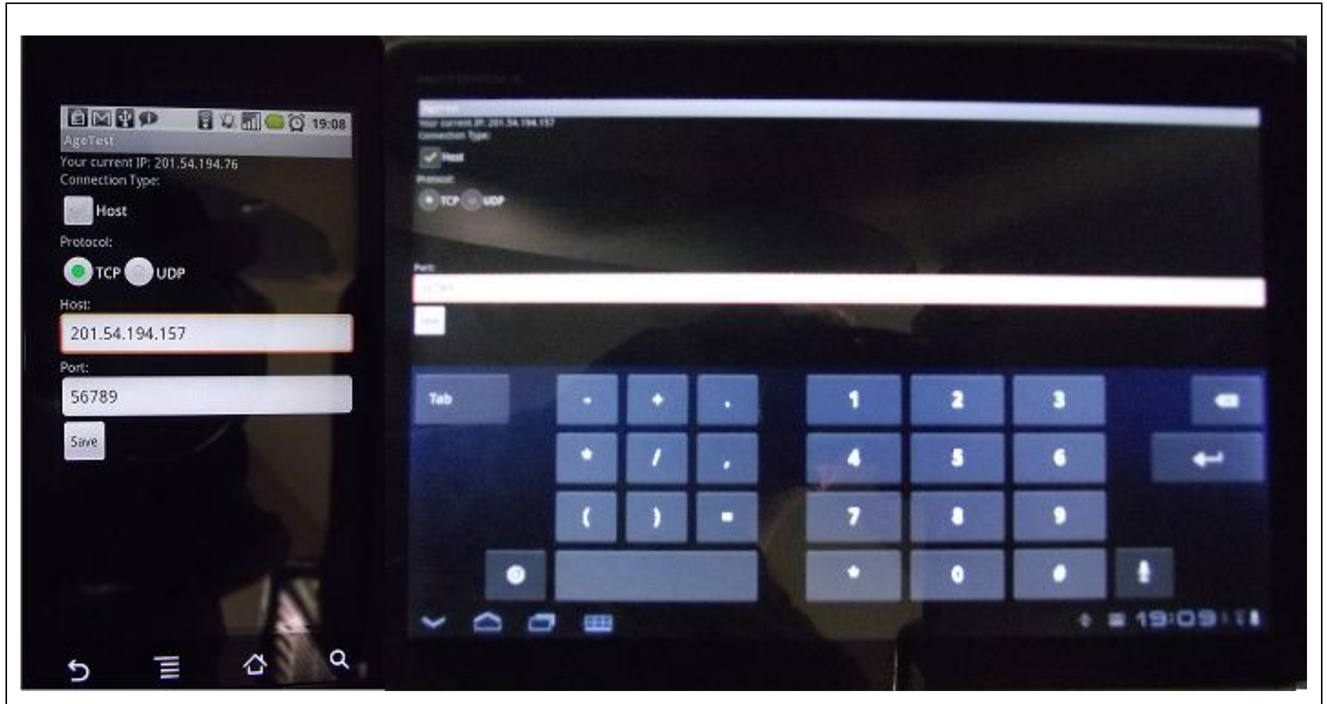


Figura 21 – Tela de configuração

Este protótipo também possui um aplicativo servidor desenvolvido apenas em Java. Ele pode ser configurado como TCP ou UDP e envia mensagens constantemente aos seus clientes. O objetivo deste aplicativo foi demonstrar que o framework pode ser utilizado também em aplicativos que não utilizam a plataforma Android e testar a capacidade do *framework* de enviar e receber um grande número de mensagens (Figuras 22).

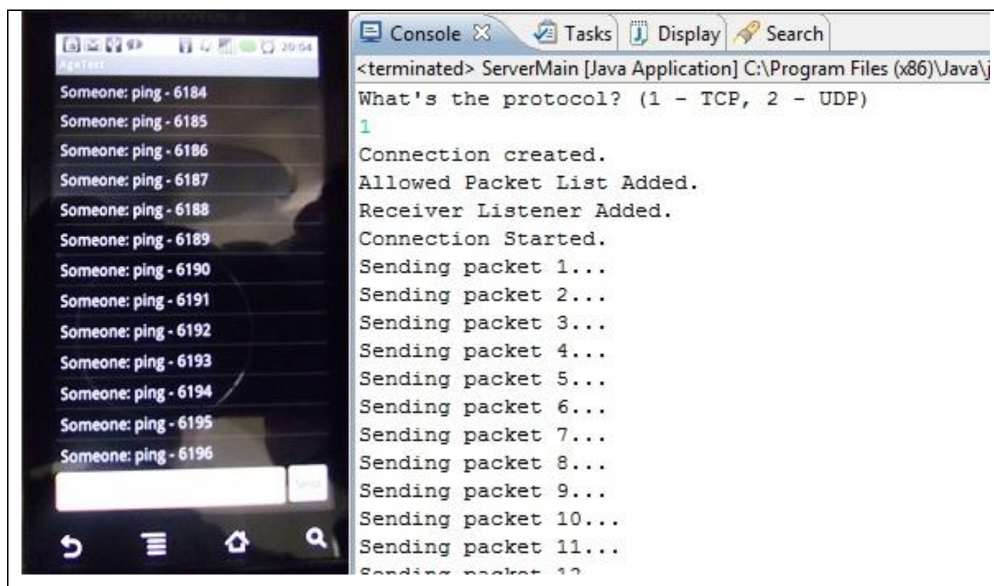


Figura 22 – A esquerda Motorola Milestone após receber o pacote 6196; a direita início da execução do aplicativo servidor em Java

3.3.3.2 Aplicativo de teste 2: protótipo de jogo *snake*

O aplicativo de protótipo de jogo *snake* tem o objetivo de testar o *framework* em um ambiente onde há troca de informações constantes entre os dispositivos (o que é muito comum na maioria dos jogos).

Visto que é um protótipo para testar o *framework*, o jogo possui apenas a implementação das cobras, ou seja, não há comida para as cobras aumentarem de tamanho e não há tratamento de colisão para determinar o ganhador do jogo. A Figura 23 apresenta a tela principal com o jogo iniciado. A segunda tela é a tela de configuração da conexão. Esta tela é uma cópia da tela do protótipo 1 e tem a mesma funcionalidade.

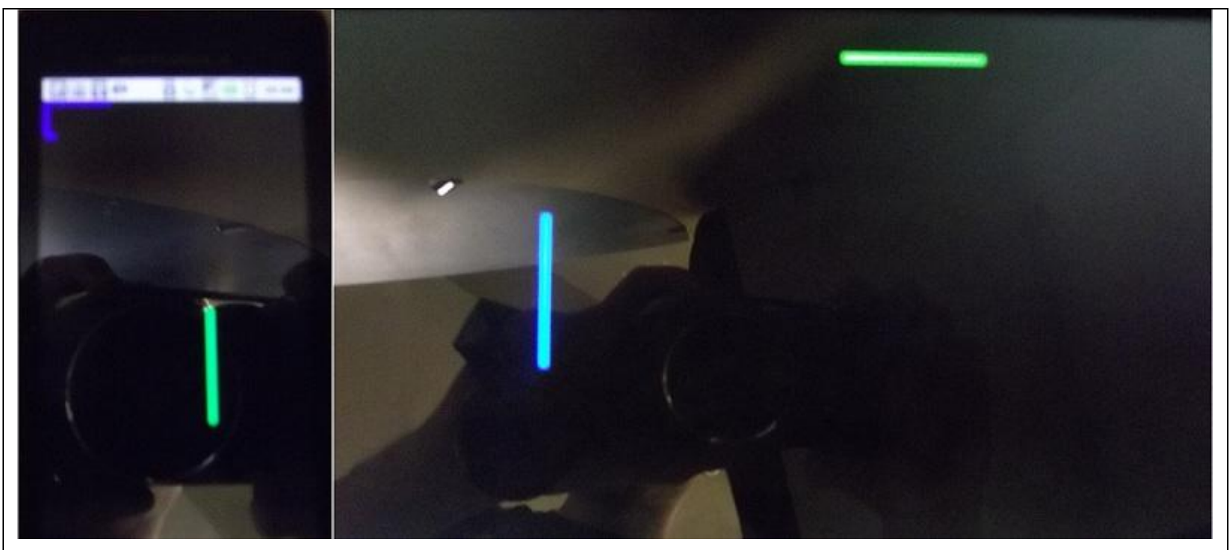


Figura 23 – Jogo snake iniciado no Motorola Milestone à esquerda e no Motorola XOOM à direita. A cobra verde é a cobra controlada pelo jogador e a azul é a do adversário.

O jogo pode ser desenvolvido para dois jogadores. Cada jogador tem controle de uma cobra quando o jogo começar. Na versão final, o objetivo do jogo é fazer com que o adversário colida com a sua cobra, perdendo assim o jogo. Para tornar mais interessante são geradas comidas randomicamente durante o jogo. A cobra que conseguir pegar a comida tem seu corpo aumentado, tornando assim mais fácil do outro jogador colidir.

3.4 RESULTADOS E DISCUSSÃO

A implementação do *framework* permitiu a criação de dois aplicativos exemplificando

e validando seu funcionamento. Uma das dificuldades encontradas durante a realização deste trabalho foi a realização dos testes. Devido a uma limitação do plugin ADT não é possivel vincular projetos a projetos Android. O vínculo deve ser realizado gerando um arquivo .jar do projeto que pretende-se vincular e adicionando este ao projeto Android como biblioteca. Devido à esta restrição, toda vez que é realizada alguma alteração no projeto do *framework* os projetos das aplicações devem ter seus arquivos `.jar` do *framework* atualizados manualmente, ou altomaticamente atravéz da criação de um script para este fim.

Ainda na execução dos testes há uma certa dificuldade na infra-estrutura necessária para sua realização. A estrutura de rede não pode utilizar proxy, pois não foi implementado o suporte a proxy. É necessário que todos os dispositivos possam se conectar na rede utilizada. O dispositivo Motorola Milestone apresenta dificuldades em conectar em algumas redes *Wi-Fi Protected Access* (WPA).

Como a grande parte dos trabalhos correlatos são ferramentas pagas, é difícil realizar uma comparação aprofundada das funcionalidades de cada trabalho. O Quadro 19 mostra as funcionalidades básicas de um *framework* de redes e quais trabalhos possuem estas funcionalidades implementadas.

Trabalho	TCP	UDP		Encriptação	Grau de abstração da camada de rede
		Sem garantia de entrega	Com garantia de entrega		
GNE	N	S	S	N	Baixo
GNet	?	S	S	?	Alto
Photon	S	S	S	?	Médio
RakNet	?	S	S	S	Médio
framework	S	S	N	S	Baixo

Quadro 19 – Comparação das funcionalidades básicas de um framework de redes

4 CONCLUSÕES

Este trabalho apresentou um *framework* de rede multi-plataforma, pode-se notar que todos os objetivos foram alcançados pelo *framework*. A forma como foi desenvolvido, buscando não utilizar nenhuma biblioteca do Android no *framework*, apenas utilizando Java, fez com que o *framework* possa ser utilizado com servidores e clientes tanto em Java como na plataforma Android. O servidor pode ser centralizado em um computador ou dispositivo utilizando a conexão do tipo servidor e utilizando a conexão do tipo cliente nos outros computadores ou dispositivos ou descentralizado utilizando uma conexão do tipo servidor e várias do tipo cliente uma para cada computador que se deseja comunicar.

O *framework* também possui segurança permitindo ao desenvolvedor definir quais tipos de pacotes podem ser recebidos, utilizar encriptação ou não, e permite também que o desenvolvedor possa criar sua própria implementação para efetuar a encriptação.

A eficiência do *framework* pôde ser testada a partir dos dois protótipos de aplicativos que foram implementados, demonstrando que o *framework* funciona, é simples de ser utilizado, e foi capaz de ser utilizado tanto em um protótipo de jogo *multiplayer* quanto em um protótipo de um aplicativo em rede comum.

4.1 EXTENSÕES

Durante o desenvolvimento foram observadas alguma possibilidades de extensões deste trabalho.

A primeira possibilidade de extensão é incluir a comunicação via Bluetooth no *framework*. Isto pode ser realizado implementando as interfaces do caso de uso UC01. A principal vantagem de se utilizar a conexão Bluetooth é poder fazer a comunicação sem a necessidade de *access points*.

Também pode ser incluído no framework a comunicação por VoIP, permitindo assim que seja utilizado para transmissão de som.

Por fim, mais uma possibilidade de extensão é unir este trabalho com trabalhos de *framework* gráfico, *framework* de inteligência artificial, *framework* de física, *framework* de persistência de dados, entre outros com o objetivo de formar uma *engine* completa para jogos.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDROID DEVELOPERS. **What is Android?** [S.l.], 2011a. Disponível em: <http://developer.android.com/guide/basics/what-is-android.html>. Acesso em: 04 ago. 2011.

______. **Intents and intent filters** [S.l.], 2011b. Disponível em: <http://developer.android.com/guide/topics/intents/intents-filters.html>. Acesso em: 04 ago. 2011.

ARMITAGE, Grenville; CLAYPOOL, Mark; BRANCH, Philip. **Networking and online games**: understanding and engineering multiplayer internet games. West Sussex: John Wiley & Sons, 2006.

DELOURA, Mark A. **Game programming gems**. Rockland: Charles River Media, 2000.

EXIT GAMES. **Photon**. [S.l.], 2011. Disponível em: <http://www.exitgames.com/Photon>. Acesso em: 20 set. 2011.

FIEDLER, Glenn. **Gafferongames.com**. [S.l.], 2008. Disponível em: <http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/>. Acesso em: 29 jul. 2011.

GAMATRA. **GNet**. Shanghai, 2009. Disponível em: <http://www.gamantra.com/products.html>. Acesso em: 20 set. 2011.

JENKINS SOFTWARE. **RakNet**. [S.l.], [2011?]. Disponível em: <http://www.jenkinssoftware.com/>. Acesso em: 20 set. 2011.

MULHOLLAND, Andrew; HAKALA, Teijo. **Programming multiplayer games**. Boulevard Plano: Wordware Publishing, 2004.

SMED, Jouni; HAKONEN, Harry. **Algorithms and networking for computer games**. West Sussex: John Wiley & Sons, 2006.

WINNEBECK, Jason. **Gillius's programming**. Rochester, 2011. Disponível em: <http://www.gillius.org/gne/>. Acesso em: 29 jul. 2011.

ANEXO A – Detalhamento dos casos de uso

4.1.1.1 Trafegar informações

Este caso de uso descreve como o desenvolvedor interage com o framework para o envio e recebimento de informações.

Detalhes do caso de uso são descritos no Quadro 1.

UC01 – Trafegar informações	
Descrição	O sistema deve permitir o tráfego de informações através de uma interface padrão. A interface padrão deve ser extensível, permitindo a implementação do tráfego de informações por diferentes protocolos. Além se extensível a interface também deve ser simples, ou seja, possuir uma boa abstração das informações técnicas, permitindo assim que o desenvolvedor se preocupe com as informações que serão transmitidas e não como as informações serão transmitidas. Essa interface também deve permitir o tráfego de informações como servidor ou cliente.
Pré-Condição	O desenvolvedor deve informar os dados da conexão proveniêntes do UC-02.
Cenário Principal	1. o desenvolvedor cria uma conexão informando as configurações; 2. o sistema cria a conexão de acordo com a configurações informadas; 3. o desenvolvedor informa quais os pacotes são permitidos para a conexão; 4. o desenvolvedor associa à conexão um ou mais *listeners* para receber os pacotes; 5. o desenvolvedor inicia a conexão; 6. o sistema estabelece a conexão; 7. o desenvolvedor envia um pacote para um receptor; 8. o sistema transmite o pacote; 9. o sistema recebe um pacote e dispara os *listeners* associados à conexão; 10. o desenvolvedor trata o pacote recebido nos *listeners*; 11. o desenvolvedor fecha a conexão; 12. o sistema notifica o receptor que a conexão será fechada e então finaliza a conexão.
Cenário Alternativo 1	1. se no passo 1 o desenvolvedor informa nas configurações que a conexão é do tipo servidor, no passo 6 o desenvolvedor poderá enviar o pacote para um ou vários receptores e no passo 11 o sistema irá notificar todos os receptores.
Exceção	1. se as configurações estiverem incorretas ao executar o passo 4 do cenário principal o sistema deve retornar um erro.
Pós-Condição	1. o sistema deve transmitir e/ou receber as informações de outro servidor; 2. outro servidor deve transmitir as informações para o sistema e/ou receber as informações do sistema.

Quadro 1 – Caso de uso UC01

4.1.1.2 Configurar informações da conexão

Este caso de uso descreve como o desenvolvedor configura as informações da conexão.

UC02 – Configurar informações da conexão	
Descrição	O sistema deve possuir uma forma única para configurar qualquer tipo de comunicação que estenda as interfaces do caso de uso UC01, deve ser possível informar as configurações do host do servidor, da porta, do protocolo da conexão (TCP, UDP ou outros) e se é do tipo cliente ou servidor. Além desses dados deve ser possível incluir outras configurações através de propriedades.
Pré-Condição	Não há.
Cenário Principal	1. o desenvolvedor cria uma instância de configuração; 2. o desenvolvedor informa a configuração do tipo de conexão como cliente; 3. o desenvolvedor informa a configuração do tipo de protocolo como TCP; 4. o desenvolvedor informa o host do servidor; 5. o desenvolvedor informa a porta do servidor; 6. o desenvolvedor cria uma conexão com a instância da configuração; 7. o sistema retorna a conexão mais apropriada para as configurações informadas.
Exceção	1. se no passo 7 o sistema não encontrar a conexão mais apropriada deverá retornar um erro.
Pós-Condição	Não há.

Quadro 2 – Caso de uso UC02

4.1.1.3 Trafegar informações pelo protocolo TCP/IP

Este caso de uso descreve como será realizada a implementação do protocolo TCP/IP utilizando as interfaces padrões fornecidas pelo UC-01.

UC03 – Trafegar informação pelo protocolo TCP/IP	
Descrição	O protocolo TCP/IP é um protocolo que pode ser utilizado em jogos em rede simples ou até jogos mais complexos desde que estes não exijam um tráfego muito elevado de informações. O sistema deverá possuir uma implementação das interfaces do caso de uso UC-01 para o protocolo TCP/IP.
Pré-Condição	Não há.
Cenário Principal	1. o desenvolvedor cria uma configuração conforme UC-02 para o protocolo TCP/IP; 2. o desenvolvedor associa um ou mais *listeners* de pacote a conexão; 3. o desenvolvedor informa quais os pacotes permitidos para a conexão; 4. o desenvolvedor inicia a conexão; 5. o sistema cria uma conexão do tipo TCP/IP; 6. o desenvolvedor envia um pacote; 7. o sistema envia o pacote utilizando o protocolo TCP/IP, com garantia de entrega e ordem; 8. o pacote é enviado com sucesso; 9. o desenvolvedor finaliza a conexão.
Exceção	1. se no passo 1 o sistema não encontrar a conexão mais apropriada deverá retornar um erro; 2. se no passo 7 ocorrer algum problema no envio do pacote, deverá retornar um erro; 3. se no passo 9 ocorrer algum problema ao finalizar a conexão deverá retornar erro.
Pós-Condição	Não há.

Quadro 3 – Caso de uso UC03

4.1.1.4 Trafegar informações pelo protocolo UDP/IP

Este caso de uso descreve como será realizada a implementação do protocolo UDP/IP

utilizando as interfaces padrões fornecidas pelo UC-01.

UC04 – Trafegar informação pelo protocolo UDP/IP	
Descrição	O protocolo UDP/IP é um protocolo que pode ser utilizado em jogos em rede mais complexos que exijam um tráfego muito elevado de informações. O sistema deverá possuir uma implementação das interfaces do caso de uso UC-01 para o protocolo UDP/IP.
Pré-Condição	Não há.
Cenário Principal	1. o desenvolvedor cria uma configuração conforme UC-02 para o protocolo UDP/IP; 2. o desenvolvedor associa um ou mais *listeners* de pacote a conexão; 3. o desenvolvedor informa quais os pacotes permitidos para a conexão; 4. o desenvolvedor inicia a conexão; 5. o sistema cria uma conexão do tipo UDP/IP; 6. o desenvolvedor envia um pacote; 7. o sistema envia o pacote utilizando o protocolo UDP/IP, sem garantia de entrega e ordem; 8. o pacote é enviado; 9. o desenvolvedor finaliza a conexão.
Exceção	1. se no passo 1 o sistema não encontrar a conexão mais apropriada deverá retornar um erro; 2. se no passo 7 ocorrer algum problema no envio do pacote, deverá retornar um erro; 3. se no passo 9 ocorrer algum problema ao finalizar a conexão deverá retornar erro.
Pós-Condição	Não há.

Quadro 4 – Caso de uso UC04

4.1.1.5 Fornecer pacotes padrões para o tráfego de informações

Este caso de uso descreve os dois pacotes padrões fornecidos pelo sistema para o uso

do desenvolvedor.

UC05 – Fornecer pacotes padrões para o tráfego de informações	
Descrição	O sistema deverá fornecer dois pacotes padrões para o envio de informação, são estes o pacote de envio de string e o pacote de envio de objeto. O objetivo desses pacotes é fornecer uma comunicação simples sem a necessidade da criação de um pacote próprio e servir como referência caso haja necessidade da criação de um novo pacote.
Pré-Condição	1. o desenvolvedor criou uma configuração conforme UC-02; 2. o desenvolvedor cadastrou o pacote na lista de pacotes permitidos; 3. o desenvolvedor iniciou a conexão com sucesso.
Cenário Principal	1. o desenvolvedor cria um pacote do tipo string ou objeto informando a string ou objeto no construtor; 2. o desenvolvedor envia o pacote; 3. o sistema transforma a informação em bytes e envia até o receptor; 4. o sistema receptor faz a leitura dos bytes transmitidos; 5. o sistema receptor identifica o pacote; 6. o sistema cria uma instância do pacote identificado e envia os bytes com a informação para o pacote transformar os bytes em informação novamente; 7. o sistema trata o pacote através dos *listeners* de pacote.
Exceção	1. se no passo 2 ocorrer algum problema no envio do pacote, deverá retornar um erro.
Pós-Condição	Não há.

Quadro 5 – Caso de uso UC05

4.1.1.6 Fornecer segurança para as informações trafegadas

Este caso de uso descreve como será realizada a segurança das informações que serão trafegadas pelo *framework*. O desenvolvedor poderá utilizar dois artifícios para garantir a segurança, sendo eles o filtro dos pacotes que podem ser transmitidos e a encriptação por um

algoritmo próprio ou através do algoritmo de encriptação *Advanced Encryption Standard* (AES) 128.

UC06 – Fornecer segurança para as informações trafegadas	
Descrição	O sistema deverá garantir a segurança das informações trafegadas de forma a não permitir a comunicação por pacotes que não são permitidos pela aplicação e permitir que os dados transferidos sejam encriptados pelo algoritmo AES128 ou qualquer outro implementado pelo desenvolvedor.
Pré-Condição	1. o desenvolvedor criou uma configuração conforme UC-02; 2. o desenvolvedor iniciou a conexão com sucesso.
Cenário Principal	1. o desenvolvedor cadastra o pacote do tipo string na lista de pacotes permitidos; 2. o desenvolvedor cria um pacote do tipo string e envia o pacote utilizando a encriptação AES128; 3. o sistema transforma a informação em bytes, encripta os bytes e envia até o receptor; 4. o sistema receptor faz a leitura dos bytes transmitidos; 5. o sistema receptor identifica o tipo de encriptação utilizado; 6. o sistema receptor decripta os byte; 7. o sistema receptor identifica o pacote; 8. o sistema cria uma instância do pacote identificado e envia os bytes com a informação para o pacote transformar os bytes em informação novamente; 9. o sistema trata o pacote através dos *listeners* de pacote.
Exceção	1. se no passo 2 ocorrer algum problema no envio do pacote, deverá retornar um erro; 2. se no passo 5 o receptor não encontrar o tipo de encriptação utilizado, deverá retornar erro; 3. se no passo 7 o receptor não encontrar o pacote na lista de pacotes permitidos deverá retornar erro.
Pós-Condição	Não há.

Quadro 6 – Caso de uso UC06