INSTITUTO POLITÉCNICO DE BRAGANÇA
Escola Superior de Tecnologia e Gestão

JEagle: Módulo de comunicação flexível para o motor de jogos jMonkeyEngine

Marcelo Penteado Ferreira Filho

Dissertação apresentada à Escola Superior de Tecnologia e Gestão para obtenção do Grau de Mestre em Sistemas de Informação

Trabalho realizado sob a orientação de:

Professor Rui Pedro Lopes

Professor Richard Ribeiro

Bragança

Maio de 2018

Dedicatória

Dedico esse trabalho a meus pais, Marcelo e Jocemara, que sempre com muito apoio e dedicação, nunca mediram esforços para que eu completasse mais essa etapa da minha vida.

Agradecimentos

Agradeço aos meus orientadores Rui Pedro Lopes e Richard Ribeiro pela paciência e ajuda que me possibilitou concluir esse trabalho, agradeço também aos meus professores do Instituto Politécnico de Bragança e aos professores da Universidade Tecnológica Federal do Paraná que durante muito tempo me ensinaram e me deram a base de meu conhecimento necessário para que eu pudesse obter o título de mestre em sistemas da informação.

Resumo

O desenvolvimento de jogos *multiplayer* é normalmente uma das áreas em que os desenvolvedores de jogos geralmente encontram dificuldades, visto que desenvolver um jogo *singleplayer* é uma tarefa totalmente diferente do que desenvolver um jogo *multiplayer*, pois muitas das mecânicas que foram desenvolvidas para apenas um jogador devem ser capazes de funcionar para vários jogadores que estarão simultaneamente conectados. Os módulos de comunicação *multiplayer* têm de ser desenhados com cuidado, devido a restrições de conetividade, à latência de rede e outros fatores. É necessário planejar quais serão os protocolos trocados entre os integrantes do jogo, que ações deverão ser executadas, como interpretar cada ação recebida e outros, que devem ser levados em consideração para que um jogo funcione como esperado.

O objetivo desse trabalho é o desenvolvimento de um módulo de comunicação flexível em Java para o motor de jogos jMonkeyEngine, utilizando a arquitetura cliente/servidor e também P2P para que a comunicação entre os integrantes do jogo possa estabelecer sua comunicação e o desenvolvedor que posteriormente deseje desenvolver um jogo *multiplayer* tenha alternativas para suportar as necessidades de comunicação.

Palavras Chave: Motores de jogos, P2P, Cliente-servidor, JGroups, jMonkeyEngine.

Abstract

The development of multiplayer games is usually one of the areas where game developers often encounter difficulties, since developing a singleplayer game has a different set of requirements than a multiplayer game. The multiplayer communication modules involve carefully design, to cope with connectivity issues and with the inherent network latency. I tis necessary to plan what protocols will be used, what actions should be performed, how to interpret each action received, among others.

The main objective of this work is the development of a flexible Java communication module for the jMonkeyEngine game engine using the client/server architecture and also P2P architecture, so in this way, the communication between the members of the game will be established, and the developer, woudln’t need to worry about the development of the communication module.

Keywords: Game engines, P2P, Client-server, JGroups, jMonkeyEngine.

Índice Geral

Capítulo 1 Introdução 1

1.1. Enquadramento 1

1.2. Objetivos 2

1.3. Estrutura do documento 2

Capítulo 2 Desenvolvimento de jogos *Multiplayer* 3

2.1. Motor de jogo 3

2.2. Comunicação em jogos 5

2.3. Arquiteturas e tecnologias de rede em jogos 6

2.3.1. Topologias de rede 7

2.3.2. Protocolos 9

2.3.3. Comunicação em jogos 12

2.4. Configuração de processos 14

2.5. Como os jogos se comunicam atualmente 18

2.6. Latência 19

Capítulo 3 Tecnologias e ferramentas 21

3.1. *Sockets* 21

3.2. Java em jogos 22

3.3. jMonkeyEngine 23

3.3.1. LWJGL 25

3.3.2. SpiderMonkey Networking API 27

3.3.3. *Listeners* 27

3.4. JGroups 28

3.5. Serialização e deserialização de objetos em Java 29

Capítulo 4 Desenvolvimento 31

4.1. Introdução 31

4.2. Módulo de comunicação centralizado 32

4.3. Módulo de comunicação descentralizado 35

4.4. Planejamento da comunicação 39

4.4.1. Execução das ações e simulações 42

4.4.2. Enviando e recebendo mensagens 43

4.4.3. *Headers* presentes nos módulos de comunicação centralizado 46

4.4.4. *Headers* presentes nos módulos de comunicação descentralizado 52
4.5. Desenvolvimento do módulo de comunicação centralizado 53
4.5.1. Classes comuns aos protocolos TCP e UDP 54
4.5.2. Classes utilizadas no cliente do protocolo TCP 55
4.5.3. Classes utilizadas no servidor do protocolo TCP 57
4.5.4. Classes utilizadas no cliente do protocolo UDP 59
4.5.5. Classes utilizadas no servidor do protocolo UDP 61
4.6. Desenvolvimento do módulo de comunicação descentralizado 63
4.7. Injetando a *interface HandleReceiverControl* no código do JEagle 67
4.8. Executando o JEagle 68
4.8.1. O jogo Monkey Blaster 68
4.8.2. Adaptação do jogo Monkey Blaster para multijogador 70
4.8.3. Junção do jogo Monkey Blaster com o JEagle 73
4.8.4. Mapeamento das ações 75
4.9. Conclusão 80
Capítulo 5 Análise e discussão de resultados 81
5.1. Introdução 81
5.2. Dados técnicos dos testes 82
5.3. Testes efetuados 83
5.4. Análise de resultados 91
5.5. Resultados 96
Capítulo 6 Conclusões 99
6.1. Conclusão 99

Índice de Figuras

Figura 1: Arquitetura presente em um motor de jogo. (adaptado de [1]) 5
Figura 2: Topologia ponto a ponto. (adaptado de [3]) 8
Figura 3: Topologia multiponto. (adaptado de [3]) 8
Figura 4: Topologia estrela. (adaptado de [3]) 9
Figura 5: Arquitetura cliente/servidor. 15
Figura 6: Arquitetura Peer-To-Peer (P2P). 15
Figura 7: Módulo de conexão centralizado. 33
Figura 8: Exemplo de visualização de atraso de comunicação com os protocolos TCP e UDP. 34
Figura 9: *Peer* local e *peers* externos. 36
Figura 10: Posição diferentes dos jogadores no jogo Monkey Blaster 37
Figura 11: Ações de atirar e movimentar no jogo Monkey Blaster. 40
Figura 12: Ações morrer e nascer no jogo Monkey Blaster. 41
Figura 13: Passos para a execução de ações. 42
Figura 14: Classes *Datagram* e *Packet* do módulo de comunicação centralizado. 44
Figura 15: Classe *Packet* do módulo de comunicação descentralizado. 44
Figura 16: Classe *MensagemTexto*. 45
Figura 17: Classes comuns aos protocolos UDP e TCP 54
Figura 18: Classes pertencentes ao cliente com o protocolo TCP 56
Figura 19: Classes pertencentes ao servidor com o protocolo TCP. 58
Figura 20: Classes pertencentes ao cliente com o protocolo UDP 60
Figura 21: Classes pertencentes ao servidor com o protocolo UDP 62
Figura 22: Diagrama de classes do módulo de comunicação descentralizado do JEagle 65
Figura 23: Jogo Monkey Blaster. 69
Figura 24: Jogo Geometry Wars 69
Figura 25: Seção de entrada 70
Figura 26: Modularização da seção de entrada 72
Figura 27: Execução das ações através de mensagens 73
Figura 28: Monkey Blaster + JEagle 74
Figura 29: Arquitetura centralizada - configuração 1. 83
Figura 30: Arquitetura centralizada - configuração 2. 84
Figura 31: Arquitetura distribuída - configuração 1. 86
Figura 32: Arquitetura centralizada - configuração 3. 87
Figura 33: Arquitetura distribuída - configuração 2. 88
Figura 34: Arquitetura distribuída - configuração 3. 90
Figura 35: Gráfico referente ao modelo de testes 1 91
Figura 36: Gráfico referente ao modelo de testes 2 92
Figura 37: Gráfico referente ao modelo de testes 3 92
Figura 38: Gráfico referente ao modelo de testes 4 93
Figura 39: Gráfico referente ao modelo de testes 5 94
Figura 40: Gráfico referente ao modelo de testes 6 94

Figura 41: Média simples do tempo de resposta utilizando os protocolos TCP, UDP e TUNNEL do JGroups. 95
Figura 42: A classe *ClientInformations* 103
Figura 43: A classe abstrata *DataToSend* 105
Figura 44: A classe *Datagram* 106
Figura 45: A classe abstrata *Informations* 107
Figura 46: A classe *Packet*....108
Figura 47: A classe *Ping*....110
Figura 48: A classe *ServerInformations* 111
Figura 49: A classe *Client*....113
Figura 50: A *interface HandleReceiverControl*....116
Figura 51: A classe *Receiver*....117
Figura 52: A classe *ReceiverControl* 119
Figura 53: A classe *Client*....120
Figura 54: A classe *ClientChannel* 122
Figura 55: A *interface HandleReceiverControl*....124
Figura 56: A classe *ReceiverControl* 126
Figura 57: A classe *Server* 127
Figura 58: A classe *Client*....130
Figura 59: A *interface HandleReceiverControl*....133
Figura 60: A classe *PingPong*....135
Figura 61: A classe *Receiver*....137
Figura 62: A classe *ReceiverControl* 138
Figura 63: A classe *Client*....139
Figura 64: A *interface HandleReceiverControl*....141
Figura 65: A classe *PingPong*....142
Figura 66: A classe *ReceiverControl* 144
Figura 67: A classe *Server* 146
Figura 68: A classe *Channel* 149
Figura 69: A classe *Cluster*....152
Figura 70: A classe *Member* 153
Figura 71: A classe *Packet*....154
Figura 72: A classe *Ping* 155
Figura 73: A classe *ViewControl* 157
Figura 74: A classe *ReceiverControl* 159
Figura 75: A *interface HandleReceiverControl*....160

Índice de Tabelas

Tabela 1: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP no modelo de testes 1 84
Tabela 2: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP no modelo de testes 1 84
Tabela 3: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP na configuração de testes 2. 85
Tabela 4: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP na configuração 2. 85
Tabela 5: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups no modelo de testes 3 87
Tabela 6: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP no modelo de testes 4 88
Tabela 7: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP no modelo de testes 4 88
Tabela 8: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups jutamente com o protocolo TUNNEL no modelo de testes 5 89
Tabela 9: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups juntamente com o protocolo TUNNEL no modelo de testes 6 90

Capítulo 1 Introdução

Esse capítulo tem como função introduzir o tema do trabalho, assim como descrever as ideias gerais que irão gerar uma solução para o seu futuro desenvolvimento. Os objetivos do trabalho, sua importância e foco também serão descritos.

1.1. Enquadramento

A grande maioria dos motores de jogos que estão disponíveis oferecem suporte para o desenvolvimento de jogos *multiplayer*, utilizando a arquitetura de comunicação cliente/servidor, onde os clientes requisitam informações enviando ao servidor mensagens que, por sua vez, as processa e lhes dá resposta, contendo assim, os resultados do processamento.

As tecnologias utilizadas para a comunicação normalmente incidem sobre os protocolos TCP, UDP e também modelos de conexão como RPC. Porém, na maioria dos jogos, é possível perceber que o protocolo UDP é principalmente utilizado em situações de tempo real como, por exemplo, a posição dos personagens. Por outro lado, o protocolo TCP é utilizado para funções que necessitem de uma garantia de entrega, como por exemplo sistemas de autenticação.

Como na maioria dos casos apenas a arquitetura cliente/servidor é utilizada, será desenvolvido um novo método utilizando uma arquitetura alternativa de conexão.

1.2. Objetivos

O objetivo do trabalho é desenvolver um novo módulo de conexão para o motor de jogo jMonkeyEngine. Esse novo módulo, além de permitir usar a arquitetura cliente-servidor, também permite usar a arquitetura *Peer-to-Peer* (P2P), onde cada uma das máquinas que estão conectadas ao grupo possuem a capacidade de compartilhar informações, possibilitando assim uma comunicação direta sem a necessidade da utilização de um servidor centralizado. Neste caso, os jogadores conectam-se uns aos outros e compartilham a informação do jogo sem a necessidade da existência de um servidor, de modo que todos os dados são processados localmente pelo *peer* em questão e, então, transmitidos para os outros *peers*.

1.3. Estrutura do documento

A estrutura dos capítulos presentes nesse trabalho descrevem o desenvolvimento de uma solução para o problema da comunicação *multiplayer* utilizando a linguagem de programação Java e o motor de jogos jMonkeyEngine, sendo assim, o trabalho esta organizado da seguinte forma: no capítulo 2 será introduzido ao leitor o que é um motor de jogo, como funciona a comunicação em jogos e como as arquiteturas e tecnologias de rede são aplicadas em jogos; no capítulo 3 serão apresentadas as tecnologias e ferramentas já existentes para o desenvolvimento de um módulo de comunicação; no capítulo 4 será apresentado todos os passos necessários para o desenvolvimento do módulo de comunicação JEagle; no capítulo 5 serão feitos testes utilizando o módulo de comunicação JEagle e também serão apresentados os resultados desses testes; no capítulo 6 serão apresentadas as conclusões e direções para os trabalhos futuros.

Capítulo 2 Desenvolvimento de Jogos *Multiplayer*

Esse capítulo irá explicar o modelo dos motores de jogos e os principais elementos utilizados na construção de um jogo, englobando conceitos universais e genéricos, que normalmente estão presentes em todos os motores de jogos.

2.1. Motor de jogo

Um motor de jogos é um programa ou um conjunto de bibliotecas que contém ferramentas previamente desenvolvidas para facilitar o desenvolvimento de jogos ou aplicações gráficas pelos desenvolvedores. Portanto, um motor de jogos é uma camada intermediária entre a aplicação final e as ferramentas que o motor oferece, fornecendo dessa forma uma maior transparência e abstração para o desenvolvedor, ou seja, o desenvolvedor deve preocupar-se em escrever um código apenas uma vez, e ele funcionará em múltiplas plataformas.

O motor de jogo é um dos principais fatores que vai adicionar recursos sofisticados aos jogos, aumentando dessa forma seu realismo e proporcionando uma melhor jogabilidade. Como consequência disso, o *software* desenvolvido ganhará um maior apelo comercial e competitividade com os jogos de outras desenvolvedoras [1].

Além disso, um motor de jogo é o *software* responsável por estabelecer a integração entre as diferentes ferramentas e componentes para o desenvolvimento rápido de um jogo ou uma aplicação gráfica. Geralmente, os componentes que um motor de jogo contém são: motor gráfico, motor sonoro, interface com dispositivos de entrada e saída e recursos de rede [1]. O desenvolvedor irá poupar tempo, pois ele não terá a necessidade de desenvolvedor tudo do início, visto que o motor de jogo já contém uma grande quantidade de recursos pré-implementados. Estes componentes implementam funcionalidades de: comunicação, interação, processamento e visualização.

A comunicação é responsável por executar o jogo de forma distribuída com vários jogadores. Esse componente normalmente utiliza *sockets* para criar a comunicação entre os vários clientes espalhados pelo mundo, porém a sua implementação de baixo nível varia dependendo da plataforma escolhida. Esse componente também é responsável por enviar dados de entrada dos jogadores externos para o componente de processamento [1].

O componente de interação tem a finalidade de obter as entradas de dispositivos como teclado, mouse, *joystick*, etc. Sendo assim, esse componente tem apenas a finalidade de tratar os eventos de entrada gerados pelos jogadores [1].

O componente de processamento controla a lógica e todos os objetos pertencentes ao jogo. Para essa tarefa poder ser executada, é necessário que exista uma descrição do jogo assim como seus respetivos recursos que, em conjunto com os dados do componente de interação e o componente de comunicação, é possível desenhar o mundo do jogo e mostrá-lo no componente de visualização. Além disso, o componente de processamento contém vários controladores que são responsáveis por operações lógicas como, por exemplo, motores de física ou inteligência artificial, lembrando que esses motores lógicos são totalmente abstratos, podendo dessa forma, serem utilizados em diferentes jogos [1].

O componente de visualização é o último componente a ser executado, ele utiliza todos os outros componentes como parâmetros de entrada para renderizar o mundo do jogo e também executar sons de acordo com a perspetiva da entidade controlada pelo respetivo jogador. Esse componente normalmente utiliza um monitor para executar a saída do processamento da visualização de imagem. Da mesma forma, a saída do processamento da visualização do som é executada em caixas de som, fones de ouvido ou *headsets* [1].

Essa arquitetura de motor de jogo pode ser melhor observada na Figura 1.

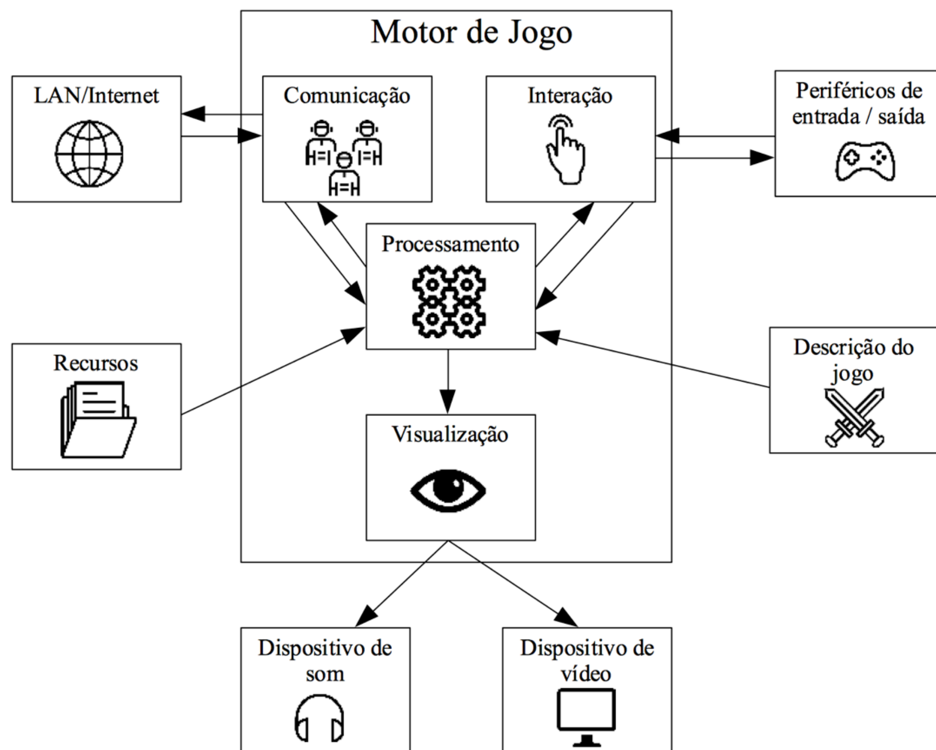

Figura 1: Arquitetura presente em um motor de jogo. (adaptado de [1])

Essa arquitetura de motor de jogo foi desenvolvida para ser genérica, de forma que é possível utilizá-la para criar os mais variados tipos de jogos. Ele funcionará independente da implementação, descrição ou recursos utilizados pelo jogo, além de poder ser executado em diferentes plataformas.

Portanto, um motor de jogo contém um conjunto de ferramentas que dispensam o desenvolvedor de as implementar, sendo que cada ferramenta será responsável por solucionar problemas comuns para o desenvolvimento de aplicações gráficas ou jogos [1].

2.2. Comunicação em jogos

Jogos, assim como qualquer outra aplicação, comunicam através de uma rede local ou da Internet onde, para a comunicação ser executada, são utilizados protocolos como o TCP e/ou o UDP [2].

Para os jogos funcionarem de maneira distribuída, arquiteturas de rede devem ser utilizadas, como por exemplo uma arquitetura cliente/servidor ou então *Peer-To-Peer* (P2P). Sendo assim, cada cliente mantém-se em comunicação constante com os outros clientes, ou com o servidor, lembrando que a forma de conexão depende da arquitetura de rede utilizada [2]. Um cliente que está funcionando como servidor também pode executar o jogo da mesma forma que os outros clientes. Porém, se isso ocorrer, ele não será chamado de servidor dedicado, visto que não executa estritamente apenas a aplicação do servidor.

Além das arquiteturas de rede utilizadas para estabelecer a comunicação, também é importante que os dados sejam trocados de forma rápida. Para tal, é necessário que apenas informações importantes sejam transmitidas, de forma que a rede não seja sobrecarregada com pacotes desnecessários [2].

2.3. Arquiteturas e tecnologias de rede em jogos

Existem três conceitos importantes na área de redes de computadores que estão diretamente relacionados com motores de jogos e seu componente de comunicação. O primeiro conceito é a topologia de rede escolhida para criar a conexão física, seguindo-se o protocolo que será utilizado para o envio e recebimento das informações e, por último, a configuração de processos comunicantes, que geralmente segue uma arquitetura cliente/servidor ou *Peer-To-Peer* (P2P).

Para que um jogo se comunique de forma rápida mesmo utilizando conexões lentas, necessário minimizar o número de pacotes trocados, normalmente recorrendo a protocolos não orientados à conexão. Esse protocolo envia datagramas para uma determinada porta em uma máquina, porém a ordem e a entrega dos pacotes não são garantidas.

Normalmente os jogos combinam esses dois protocolos de comunicação, ou seja, TCP para pacotes que precisam ter garantia de entrega e ordenação de pacotes, como em sistema de autenticação, e o UDP para troca de mensagens que não necessitam de uma entrega garantida, como por exemplo, um sistema de chat [1].

O último conceito diz respeito à configuração de processos comunicantes, que passa, normalmente, pela arquitetura cliente/servidor ou P2P. Na primeira, um dos processos funciona como servidor, que recebe as mensagens enviadas pelos clientes, as processa utilizando as lógicas do jogo, e distribui os resultados pelos clientes. É importante perceber que o mesmo processo pode funcionar com base no protocolo TCP ou no UDP [1]. Essa arquitetura pode diminuir o desempenho do jogo, pois os clientes necessitam a todo o momento esperar uma resposta do servidor, pois todo o processamento é centralizado [1]. Como alternativa a esse processo, pode-se adotar um modelo alternativo de arquitetura, onde não existe um processamento centralizado, de forma que cada máquina pode comunicar (P2P).

2.3.1. Topologias de Rede

As topologias de rede são definidas por arquiteturas físicas e lógicas, onde são constituídas de vários equipamentos, roteadores, modems, *hubs* e também protocolos, *proxy*, etc. [3]. As topologias de rede são divididas em duas partes, nomeadamente, a parte física da rede, que é composta pelo *hardware* e a parte lógica que é composta pelo *software*.

Logo, a topologia está diretamente relacionada com a forma que os nós de uma rede estão interligados, ou seja, ela é um canal, com o qual, uma rede está conectando as máquinas. No nível que estabelece as conexões fisicamente, existem várias topologias de redes para os vários fins com suas respetivas vantagens e desvantagens, porém as importantes para o desenvolvimento de jogos online são as ponto-a-ponto, multiponto e estrela.

Na topologia ponto a ponto (Figura 2) existe um nó A e um nó B que estão diretamente interligados, ou seja, um emissor e um recetor. Nesta forma de topologia não existe o compartilhamento do meio com vários usuários [3].

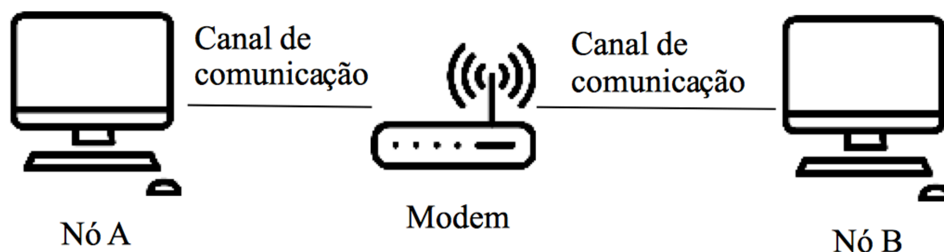

Figura 2: Topologia ponto a ponto. (adaptado de [3])

Na topologia multiponto (Figura 3), um nó envia dados para múltiplos nós pertencentes a rede, onde um mesmo meio pode ser utilizado e várias derivações são executadas ao longo deste [3].

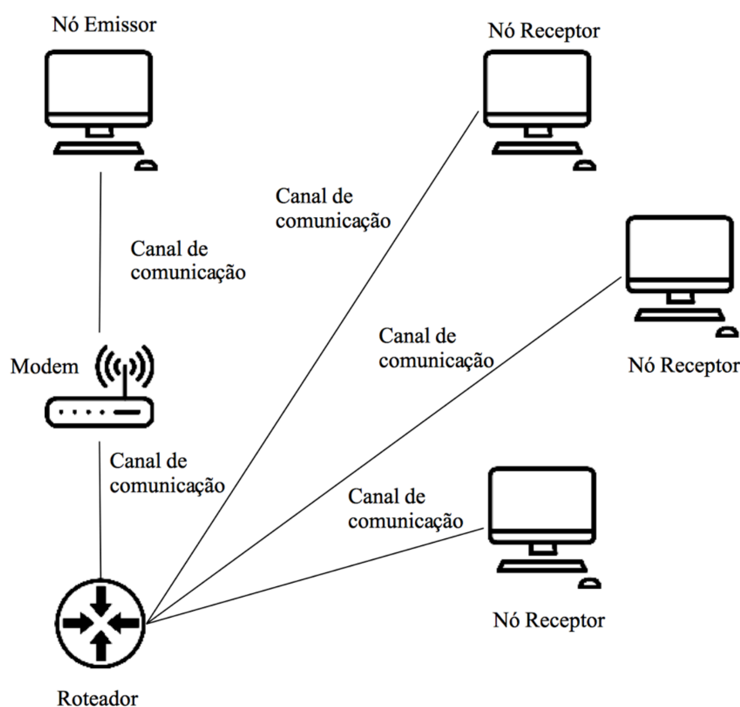

Figura 3: Topologia multiponto. (adaptado de [3])

Relativamente à configuração em estrela (Figura 4), é possível criar este tipo de topologia conectando os nós através de *hardwares* como por exemplo *switches* ou outros tipos de concentradores ou comutadores [3]. Uma rede possui uma topologia estrela quando uma

máquina se liga a outra apenas com a utilização de um equipamento centralizado, porém é importante observar, que não existe uma ligação direta [3].

Essa topologia é utilizada em redes locais (LAN's), sendo que nessa topologia, todos os dados estão centralizados em um nó, onde caso ocorra uma falha nesse nó, toda a rede ficará prejudicada [3].

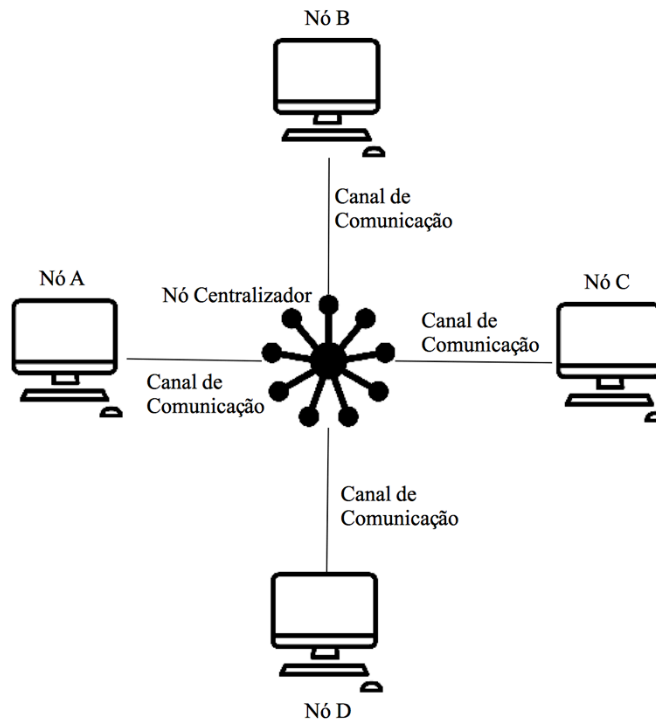

Figura 4: Topologia estrela. (adaptado de [3])

2.3.2. Protocolos

Os protocolos são um conjunto de regras bem definidas que garantem o envio e a entrega das informações entre máquinas distintas. Essas regras são criadas através do uso de algoritmos que foram desenvolvidos com o objetivo de estabelecer a comunicação entre duas ou mais máquinas em uma rede [4]. Existem vários protocolos disponíveis para os mais variados fins e eles podem ser ou não orientados a conexão, como é o caso do protocolos IP, TCP e UDP.

Os protocolos estabelecem formatos, mensagens enviadas e ordens entre os clientes presentes na rede e também as ações que serão tomadas, onde uma máquina deve compreender as informações que a outra esta enviando, portanto, os protocolos funcionam como um idioma entre os dispositivos, permitindo assim a sua comunicação.

Atualmente existem dois protocolos principais presentes na camada de transporte da Internet, um deles orientado à conexão, e o outro não-orientado à conexão [5]. UDP é um protocolo não orientado a conexões, completo e genérico, de forma que os desenvolvedores podem criar seus próprios protocolos baseando-se nele [5]. O protocolo orientado a conexões é o TCP, contendo muitas funcionalidades se comparado ao UDP, ou seja, ele estabelece conexões, acrescenta a garantia de entrega dos e ordenação dos pacotes, também contém um controle de congestionamento e de fluxo dos dados, porém essas funcionalidades extras causam uma perda de velocidade de transmissão [5].

O IP, também chamado de *Internet Protocol*, define métodos e formas de rotear pacotes em uma rede de computadores, sendo este um protocolo da camada de rede do modelo OSI. O IP também possui informações de endereçamento e algumas informações de controle que possibilitam então o correto roteamento dos pacotes, atendendo dessa forma perfeitamente aos padrões de comunicação WAN e LAN [6].

O protocolo IP possui as finalidades de entregar pacotes ou datagramas sem a necessidade de estabelecer uma conexão, além de também possibilitar a fragmentação e reconstrução dos mesmos no momento de envio e recebimento.

A forma com que esse protocolo endereça as máquinas está integrado com o processo de roteamento dos pacotes através das redes que estão conectadas, onde cada endereço IP possui uma estrutura bem definida e segue, portanto, um padrão. Com isso é possível subdividir esses endereços e usá-los para criar endereços de redes e consequentemente sub-redes. Logo, cada máquina que está conectada à rede, recebe um endereço único que possui dois números importantes, o primeiro deles é o número da rede e o seguinte é o número único que lhe é atribuído no momento da conexão, diferenciando-o assim, das outras máquinas que também estão conectadas [6].

Dados enviados utilizando esse protocolo são, geralmente, divididos em pequenas partes, denominados pacotes. Cada um desses pacotes contém o endereço do remetente e o endereço do destinatário [6]. Quando os pacotes são enviados, eles podem seguir rotas diferentes, além de também poderem chegar em uma ordem diferente do que foram enviados, sendo da responsabilidade dos protocolos reordenar os pacotes, como por exemplo, o *Transmission Control Protocol* (TCP).

O TCP é um dos principais protocolos de comunicação utilizado atualmente, uma das suas principais funções é enviar dados para um recetor e então receber uma confirmação de que os dados foram recebidos corretamente. Além disso, os pacotes são numerados para assegurar que possam ser reordenados corretamente, evidenciando a principal característica do protocolo, a confiabilidade.

O TCP também é considerado como um protocolo orientado à conexão, visto que antes que um emissor inicie o processo de envio de informações para um recetor, eles devem primeiro passar por um processo de sincronização, isto é, eles devem em primeiro momento enviar algumas informações preliminares um ao outro para que os parâmetros de transferência sejam estabelecidos e a conexão seja então criada [7].

Uma conexão TCP provem um serviço conhecido como *full-duplex*, ou seja, caso exista uma conexão utilizando o protocolo TCP entre um nó A e um nó B, logo os dados poderão fluir de A para B ao mesmo tempo que dados fluem de B para A, com isso percebe-se que as conexões do tipo TCP são sempre criadas ponto a ponto, ou seja, existe apenas um emissor e um recetor [7]. Em suma, esse protocolo é orientado à conexão entre um ponto A e B, providenciando confiabilidade de entrega, sequenciamento dos pacotes e controlo do fluxo da rede [8].

O *User Datagram Protocol* (UDP) tem como principal objetivo a agilidade na transmissão das informações, visto que ele apresenta uma estrutura e uma quantidade de funções menos robusta do que outros protocolos. É não-orientado à conexão, não faz verificação dos pacotes e também não possui funções para a ordenação de pacotes. Esse protocolo envia os dados para um determinado destino sem confirmação de que são recebidos. Embora muitos problemas possam acontecer com a utilização desse protocolo, pode-se garantir a agilidade da troca de informações entre um ponto A e B.

O UDP faz apenas as funções necessárias que um protocolo precisa executar. Além das suas funções de empacotamento, desempacotamento e algumas verificações básicas de erros, ele é basicamente o protocolo IP, de forma que, se o desenvolvedor escolher utilizar o protocolo UDP ao invés de outros protocolos, ele estará basicamente criando uma aplicação que se comunica diretamente com o protocolo IP [7].

Assim sendo, o UDP recebe as mensagens que estão presentes na camada da aplicação, adiciona os números da porta da origem e destino e repassa então os resultados a camada de rede [7]. O segmento resultante desse processo encapsula as informações dentro de um datagrama pertencente ao protocolo IP. Caso o segmento resultante seja entregue ao destinatário, então o protocolo UDP usará o número da porta de destino para entregar os dados ao processo correto [7]. É importante perceber que o UDP não possui uma representação do emissor e remetente antes de enviar o segmento de protocolo IP, fazendo dele então um protocolo não orientado à conexão [7].

Ou seja, o protocolo UDP é um protocolo considerado não confiável, contendo assim apenas os serviços essenciais de endereçamento e fragmentação, porém é muito utilizado por conter uma estrutura mais simples e com isso proporcionando uma menor sobrecarga da rede. Ele mostra-se ideal para a utilização em aplicações onde a velocidade é considerada mais útil do que a confiabilidade, ou seja, a correta entrega e envio dos pacotes não é priorizada.

Esse protocolo também é muito utilizado em jogos online, principalmente para comunicar a localização dos personagens. Apesar de eficiente, é muito comum observar personagens serem “teleportados” pelo cenário. Isto ocorre porque, caso algum pacote que contenha os dados de movimentação forem perdidos, apenas o próximo será recebido, fazendo com que só seja possível observar a próxima posição do personagem.

2.3.3. Comunicação em jogos

As características do TCP e do UDP influenciam a foram como a comunicação em jogos é concretizada. O protocolo TCP é baseado em conexões, possui uma garantia de entrega, ordenação de pacotes e um controle do fluxo da rede. Por outro lado, o UDP é um

protocolo mais simples, não orientado à conexão, não possui uma garantia de entrega e nem ordenação de pacotes.

Num primeiro momento o TCP mostra-se a escolha perfeita por possuir mais funções e também ser mais fácil de utilizar, enquanto que o UDP possui uma ausência de funções e mostra-se mais complicado. Apesar do TCP ser mais robusto, ele apresenta uma perda de performance significativa se comparado ao UDP, de forma que, para aplicações que necessitam do conceito de tempo real, a utilização do protocolo TCP deve ser evitada, como por exemplo em jogos *multiplayer*. O TCP pode ser utilizado em métodos de autenticação, mas não para a movimentação dos personagens.

O problema em utilizar TCP para jogos que necessitem de uma interação em tempo real, é que muitas partes do jogo, como por exemplo, a posição dos personagens, não levam em consideração o que aconteceu há um segundo, mas sim o que esta acontecendo agora.

Um exemplo simples do problema da utilização do protocolo TCP é que, quando um pacote é perdido, ele precisa ser reenviado. Nesse caso, deveria haver suspensão do tempo de jogo, até que o pacote fosse recebido novamente e sem erros. Só depois o jogo voltaria a continuar normalmente, o que não é possível de concretizar, visto que acabaria com a noção de tempo real. Para contornar esse problema é necessário utilizar um protocolo mais simples, mais leve e que não possua a necessidade de reenviar pacotes perdidos.

O que pode ser feito é a utilização simultânea do protocolo UDP e TCP, onde o protocolo TCP pode ser utilizado para carregar mapas, métodos de autenticação e temporizadores, e o UDP para a movimentação dos personagens, mensagens de texto e voz, além de também poder ser utilizado controlar entidades do mundo do jogo. Porém deve-se sempre ter em mente a quantidade de pacotes a trocar, para não sobrecarregar a rede e o tempo de resposta do servidor com o cliente seja o mínimo possível.

2.4. Configuração de processos

A configuração de processos segue, tipicamente, duas arquiteturas. Uma centralizada, designada por cliente-servidor, e outra distribuída, designada por P2P.

Na arquitetura cliente-servidor, o servidor inicia sua execução e entra em um estado de espera até que um cliente requisite uma conexão (Figura 5). Após o processo de conexão ser executado, o cliente começa a enviar requisições ao servidor. Depois que o servidor aceita essas requisições e as processa, ele envia uma resposta para o cliente, referente à sua requisição. Portanto, a máquina que contém a aplicação cliente é a que envia as requisições para o servidor e a máquina que contém a aplicação do servidor é a que recebe as requisições dos clientes, as processa e envia uma resposta para o cliente que a requisitou. Por outras palavras, o cliente é o processo requisitante e o servidor é o processo provedor das respostas. Na maioria das aplicações baseadas nessa arquitetura, os dados são processados no servidor e os resultados são retornados aos clientes, para que a performance dos clientes aumente [9].

Sistemas baseados em cliente/servidor tornaram-se muito populares, porque são utilizados não só em várias aplicações atuais como também são a arquitetura padrão em vários protocolos de comunicação, como por exemplo, protocolos de transferência de arquivos e o protocolo HTTP [9].

O armazenamento de informações também se torna muito mais eficaz, removendo dessa forma a necessidade de os clientes conterem grande quantidade de disco, visto que praticamente todos os dados são armazenados nos servidores, excluindo então, a necessidade do armazenamento local.

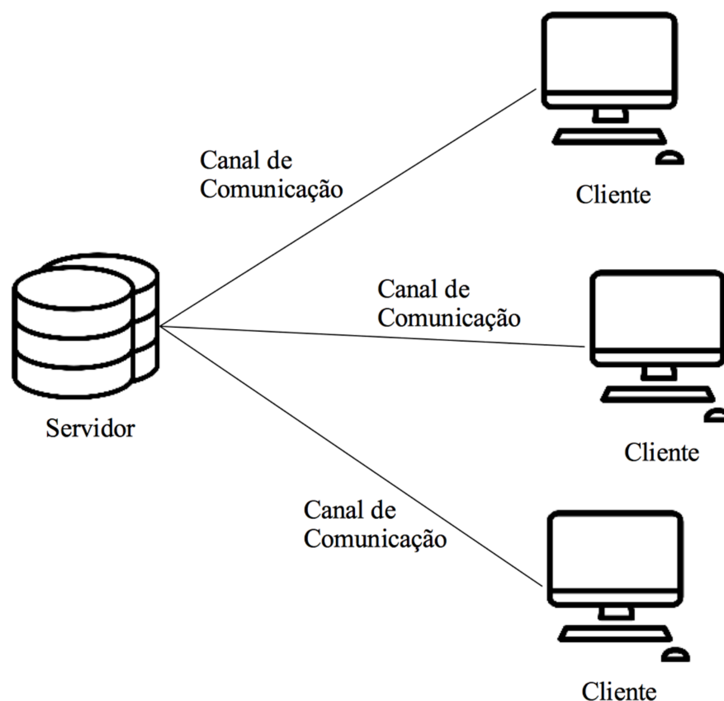

Figura 5: Arquitetura cliente/servidor.

A arquitetura P2P ganhou popularidade com a crescente utilização de *software* de compartilhamento de arquivos, onde é possível obter arquivos diretamente de outras máquinas sem a necessidade da existência de um servidor centralizado (Figura 6).

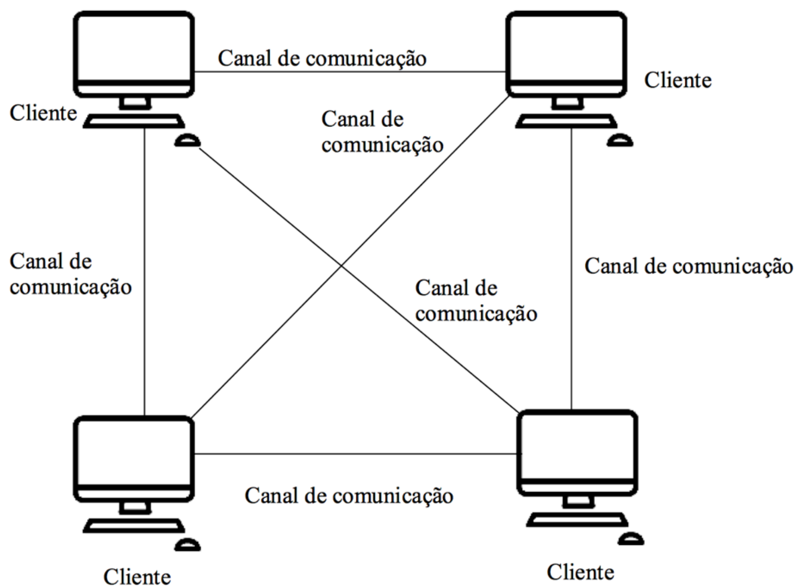

Figura 6: Arquitetura Peer-To-Peer (P2P).

Diferentemente da arquitetura cliente-servidor, onde todas as informações transitam obrigatoriamente por um servidor centralizado, a arquitetura P2P utiliza uma forma de troca de informação descentralizada [10].

A descentralização dos dados também contém uma grande vantagem, pois é possível utilizar os recursos ociosos em cada máquina para fornecer grandes possibilidades de processamento em conjunto, criando, portanto, uma forma de processamento distribuída [10].

A arquitetura P2P é considerada como um modelo não hierárquico, resultando em algumas dificuldades e características que é necessário resolver [10]. Alguns dos problemas encontrados são a descoberta dos nós, o roteamento das mensagens, o gerenciamento dos grupos de usuários, entre outros [10].

A resolução destes problemas pode passar por um sistema hierárquico baseado no conceito de *super peer*, concentrando a responsabilidade de descoberta da rede, dos grupos e dos serviços são centralizados. Esse processo tornou-se uma técnica consolidada na utilização de redes que possuem essa arquitetura, porém muitos desenvolvedores tem a opinião de que esse sistema não corresponde a uma arquitetura P2P pura, e sim a uma arquitetura híbrida, sendo que existe uma forma de centralização, nem que seja mínima [10].

Em sistemas P2P puros, a responsabilidade do roteamento de mensagens e de armazenamento de informação deve ser distribuída pela rede [10]. Nestes casos, a distribuição é assíncrona e pode ser auxiliada por um processo de como *rendezvous*, da responsabilidade de alguns *peers* específicos [10]. Estes armazenam rotas e mensagens que transitam pela rede, fornecendo os dados aos *peers* que os requisitarem [10]. Logo, percebe-se que é necessário conter pelo menos um *peer* estático que sirva como forma de entrada da rede, caso contrário, não existe uma forma dos *peers* se conectarem a rede sem a existência de pelo menos um *rendezvous* previamente conectado [10].

Para que a comunicação P2P ocorra é importante que as funcionalidades de descoberta de *peers*, serviços e recursos estejam presentes [10]. Existem duas formas de executar essas descobertas na rede, sendo que elas se adaptam a diferentes situações, são elas a descoberta direta e a descoberta indireta [10]. A descoberta direta é utilizada normalmente em redes

locais onde existe um suporte nativo para *broadcast*, sendo assim não é necessário um *super peer* ou um *rendezvous* para que a descoberta da rede seja executada. A descoberta indireta acontece para suportar ligações com máquinas que estão fora de um domínio de *broadcast*.

Além dessas questões, também existem outras que devem ser analisadas, no processo de descoberta dos serviços [10]. Um grande problema para esse processo é a complicação que os sistemas de *firewall* podem trazer. Além dos *firewalls* estarem presentes na maioria das organizações, eles podem bloquear os protocolos e também as portas utilizadas por aplicações que estabelecem uma comunicação P2P [10]. Outro problema comum que também deve ser mencionado é gerado pelos serviços de NAT, impedindo processos externos de estabelecer uma conexão com os processos internos [10]. Para contornar esses problemas é necessário que protocolos e portas padrões sejam utilizadas, como por exemplo a porta HTTP que geralmente é a 80, para os problemas dos serviços de NAT é necessário que os *peers* internos acessem o *rendezvous* da rede em períodos de tempo, verificando assim se existe alguma mensagem na cache que lhe pertença [10].

Assim como em outras arquiteturas, essa também possui seus respetivos protocolos de rede e enlace de dados. Os protocolos básicos da arquitetura P2P são o protocolo de *peer discovery* que providencia a descoberta de serviços que são fornecidos pelos *peers* na rede P2P [10]. O protocolo *peer* resolver é o responsável pelo envio e processamento das requisições [10]. O protocolo *rendezvous* é quem trata da respetiva propagação das mensagens em uma rede P2P [10]. O protocolo de informações dos *peers* possibilita que informações dos *peers* sejam obtidas da rede [10]. O protocolo *pipe binding* ou também conhecido como protocolo de tunelamento possibilita a criação de um canal de comunicação entre nós distintos na rede P2P [10]. O protocolo *endpoint routing* é quem cria um conjunto de mensagens necessárias para o roteamento entre um *peer* de origem e outro de destino [10].

2.5. Como os jogos se comunicam atualmente

Basicamente, a maioria dos jogos baseiam-se nos protocolos de comunicação TCP e UDP, ou seja, esses protocolos são utilizados visando melhorar a performance de conexão, de modo a tentar diminuir a latência e outros problemas que estão presentes entre o servidor e os clientes. Na maior parte dos jogos, o protocolo UDP é utilizado para eventos que necessitam de interação rápida por parte dos jogadores, como por exemplo, a posição no mapa. Por outro lado, propriedades importantes que necessitam de garantia de entrega, como por exemplo, lista de jogadores, devem utilizar o protocolo TCP.

Em jogos do estilo RTS ou *Real Time Strategy Games*, geralmente utiliza-se o protocolo TCP, pois é necessária a garantia de entrega das posições e jogadas dos usuários. Já em jogos do estilo MMORPG ou *Massive Multiplayer Online Role-Playing Game* e jogos de tiro ou *Shooter*, geralmente utiliza-se o protocolo UDP, porque o jogo deve acontecer em tempo real, não sendo possível uma verificação se um pacote foi entregue ou não. Nesses jogos é comum observar personagens serem "teletransportados" de uma posição A para B instantaneamente, essa é uma forma de prova que o jogo em questão utiliza o protocolo UDP, pois como um pacote foi perdido e outro seguido foi recebido, a última posição do jogador que será levada em consideração é a última recebida, produzindo então esse efeito de "teletransporte".

Alguns jogos também podem utilizar o protocolo de comunicação HTTP, apesar de ser simples, ele é muito utilizado por jogos que são executados em navegadores e geralmente eles podem funcionar através do modelo de conexão Cliente/Servidor ou P2P, esse modelo dependerá exclusivamente do jogo que será desenvolvido em alguma linguagem web, como por exemplo HTML5. Na maioria das vezes, esses jogos utilizam o protocolo HTTP onde apenas a troca de texto é utilizada, lembrando que em jogos mais avançados, até mesmo a troca de objetos é possível.

Os modelos mais comuns de comunicação entre os jogos atuais são Cliente/Servidor e P2P, geralmente o modelo Cliente/Servidor é utilizado por jogos maiores que necessitam de servidores dedicados, como por exemplo em jogos de MMORPG ou MMO ou *Massive Multiplayer Online Games*, pois uma grande quantidade de jogadores estará conectada ao

jogo, o que necessita de máquinas melhores e também uma melhor velocidade de conexão. Em jogos que estabelecem uma conexão com menos jogadores, não é necessário que um servidor dedicado seja utilizado, eliminando assim a utilização do modelo Cliente/Servidor, para esses casos utiliza-se o modelo P2P, onde a conexão é criada apenas entre os jogadores em questão, geralmente esse modelo é utilizado por jogos de cooperação, onde um jogador cria uma sala de jogo, e outros jogadores se conectam a ela.

2.6. Latência

Latência é o atraso de tempo que ocorre desde o início de um estímulo até à verificação da sua consequência. Num contexto de rede, a latência representa o atraso entre o envio da mensagem e a receção da resposta correspondente (*ping*), ou o atraso entre o envio do pacote e o recebimento do mesmo por um outro nó.

Para enviar um pacote de um ponto para outro é necessário que ocorra a serialização e deserialização do mesmo, esse processo transforma os dados em *bytes* para que possam transitar pela rede. Como esse processo demanda tempo de processamento e varia de máquina para máquina, ele não é calculado junto ao tempo de atraso, portanto, o cálculo da latência é iniciado no momento em que o pacote é enviado.

Resumidamente, a latência é o somatório dos atrasos que ocorrem em uma rede de computadores e os equipamentos que são utilizados nela. Tendo como base a camada da aplicação, a latência é o tempo de resposta resultante da diferença do tempo de entrega dos pacotes entre diferentes nós [11]. Os principais fatores que aumentam o tempo de atraso em redes são o atraso de propagação, a velocidade de transmissão e também o tempo necessário para que os equipamentos processem os pacotes, sendo que o atraso de propagação é o tempo necessário para que os dados sejam propagados no meio físico, caraterística tal, onde o administrador da rede não possui influência; a velocidade de transmissão é gerenciada pelo administrador da rede, sendo que a velocidade depende da distância e dos equipamentos utilizados, ou seja, em redes locais tem-se uma alta velocidade a baixo custo, já em redes distantes, como WAN's, a velocidade depende dos

equipamentos, algo que normalmente demanda de um alto custo; o último fator influenciador é o tempo necessário de processamento dos equipamentos, ou seja, os equipamentos que serão utilizados, como roteadores, *switches*, servidores de acesso remoto, *firewalls*, etc [11].

Capítulo 3 Tecnologias e ferramentas

Nessa capítulo será descrito o motor de jogos jMonkeyEngine, conceitos importantes, *toolkits* e também as ferramentas utilizadas no projeto.

3.1. *Sockets*

Atualmente com a padronização dos protocolos de comunicação utilizados, os dispositivos conseguem comunicar de forma simples e eficaz, sem a necessidade de existir uma conversão dos dados transmitidos entre um ponto A e B. Com essa padronização surgiram algumas tecnologias capazes de estabelecer essa comunicação, uma das mais comuns são os *sockets*, sendo que um *socket* é utilizado na maioria das vezes para modelar uma arquitetura de comunicação de cliente-servidor, permitindo assim, a troca de mensagens entre diferentes máquinas.

Esse método de troca de mensagens, conhecido como *socket*, é uma abstração dos endereços para os processos que se comunicam, sendo que cada máquina contém um identificador único composto pelo endereço da máquina e a porta, para mapear o processo. Após conhecer-se o endereço IP mais o valor da porta do emissor e do recetor, pode-se então iniciar o processo de envio e recebimento de dados.

Sockets são uma interface normalmente utilizada para a programação de comunicação através de redes de computadores na camada de transporte do modelo OSI, sendo que a comunicação utilizando *sockets* executa operações muito semelhantes as executadas em

operações de entrada e saída, de forma que, a manipulação de *sockets* é tratada como uma manipulação de arquivos [12].

As operações utilizadas em funções de entrada e saída de manipulações de arquivos também são aplicadas nas operações executadas por *sockets*, além disso a comunicação de *sockets* é independente da linguagem de programação utilizada para implementá-la, de forma que um *socket* programado utilizando a linguagem de programação Java pode comunicar-se perfeitamente com um *socket* programado utilizando a linguagem de programação C++ [12].

Ou seja, um *socket* cria um ponto de comunicação que pode ser nomeado e também endereçado em uma rede de computadores [13]. *Sockets* muitas vezes são implementados utilizando uma API de desenvolvimento que cria links de comunicação entre processos locais e remotos [13].

Os aplicativos que são desenvolvidos utilizando *sockets* podem residir no mesmo sistema ou em sistemas diferentes, podendo também estar em redes iguais ou até mesmo em redes diferentes, sendo que *sockets* podem ser úteis tanto para aplicações que utilização ou não comunicação através da rede, dessa forma, os *sockets* lhe possibilitam trocar informações através de processos que estão presentes na mesma máquina ou então através da rede, sendo assim, o acesso a informações distribuídas torna-se muito mais simples [13].

3.2. Java em jogos

Desde alguns anos atrás, a linguagem de programação Java tornou-se uma plataforma gráfica realmente capaz de executar jogos e aplicações gráficas profissionais, onde os desenvolvedores criaram ferramentas para o desenvolvimento de jogos que atualmente produzem resultados impressionantes.

Java foi desenvolvido desde o princípio com o objetivo de o desenvolvedor codificar apenas uma vez, e então possuir um executável para vários sistemas operacionais diferentes.

Uma outra informação importante sobre Java, é que o popular sistema operacional para celulares chamado Android, é baseado nessa linguagem de programação. Isso significa que os jogos que foram desenvolvidos em Java podem ser facilmente convertidos para o Android, obtendo assim uma maior quantidade de plataformas para qual o jogo estará disponível.

Uma da outra razão pela qual os jogos Java no passado não obtiveram sucesso em comparação com outras linguagens de programação foi porque historicamente houve uma falta de poderosos mecanismos de desenvolvimento de jogos disponíveis em Java, fazendo com que motores de jogos famosos como Unity e Unreal Engine dominassem o mercado. Nos últimos anos, essa situação mudou, porque motores de jogos como o jMonkeyEngine já estão em um nível relativamente maduro de desenvolvimento.

Este motor de jogos permite criar jogos 3D, permitindo desenvolver jogos para diferentes sistemas operacionais e com um desempenho competitivo mesmo com linguagens de programação de mais baixo nível, como por exemplo C ou C++.

3.3. jMonkeyEngine

Desenvolvido em Java, esse motor de jogos foi projetado para o desenvolvimento de jogos sem a necessidade de conhecer OpenGL. Totalmente desenvolvido em código aberto e de alto desempenho, ele fornece todas as ferramentas necessárias para o desenvolvimento de jogos e aplicações gráficas [14]. A renderização acontece através de LWJGL e OpenGL [14]. Esse motor de jogos é utilizado por várias empresas, como por exemplo a NCsoft, Three Rings, Jadestone, etc. [14].

Em 2003, Mark Powell criou a primeira versão do jMonkeyEngine (JME) em algumas de suas pesquisas sobre como executar a renderização de gráficos tridimensionais em OpenGL utilizando Java [14]. Rapidamente a primeira versão do JME tornou-se um motor de jogos que possuía várias ferramentas primitivas [14]. Ele baseou-se no livro Game Engine Design que utilizava a linguagem de programação C++ para criar uma arquitetura de grafo que seria utilizada em cenas. Seu projeto foi tão bem aceito pela comunidade de

desenvolvedores que rapidamente foi incorporado ao repositório Java da Sun [14]. Desde então seu motor de jogos vem crescendo cada vez mais, tornando-se uma plataforma estável para o desenvolvimento de jogos [14].

O JME baseia-se em LWJGL para assim ter acesso a uma quantidade maior de ferramentas, como por exemplo: OpenGL, OpenAL, FMOD, etc. LWJGL é uma API livre e multiplataforma, assim como a linguagem JAVA [14].

O jMonkeyEngine contém uma abrangente comunidade ativa que aumenta cada dia mais, isso implica em uma constante atualização e evolução da API [14].

Esse motor de jogos foi desenvolvido especialmente para o desenvolvimento de aplicações que necessitem de renderização em três dimensões, pois ela faz um uso intensivo da tecnologia de *shaders* e OpenGL 4, porém nada impede que aplicações em duas dimensões também sejam desenvolvidas [15]. Esse motor de jogos é um projeto centrado no desenvolvimento baseado na comunidade, pois ele é um projeto de código aberto sobre a licença BSD [15], sendo assim, ele mante-se sendo desenvolvido e atualizado constantemente, logo, por esse motivo, empresas e instituições educacionais começaram a utilizar esse motor de jogos em seus projetos. O JME também vem acompanhado de um avançado SDK que facilita o desenvolvimento das aplicações [15].

Esse motor de jogos teve o processo de desenvolvimento do seu projeto estagnado desde 2008, porém, apesar da comunidade ter continuado a atualizar o projeto, ele parecia não estar caminhando para nenhuma direção, sendo que no inicio do lançamento da sua versão 3.0, ele não passava de um experimento [15]. No início de 2009, o lançamento da versão 3.0 do motor de jogos agitou a comunidade, sendo que a maioria dos integrantes concordou que essa versão poderia ser a sucessora da versão 2.0 do jMonkeyEngine, logo a comunidade formalizou o seu lançamento, desde esse período, a versão 3.0 desse motor de jogos vem sendo atualizada [15].

Apesar de existir uma grande quantidade de motor de jogos disponíveis para o desenvolvimento de jogos utilizando a linguagem de programação Java, o jMonkeyEngine apresenta vários benefícios, como por exemplo, o fato de ser totalmente código aberto, estar sendo desenvolvido a uma quantidade de tempo relativamente alta, conter uma comunidade

inteira de desenvolvedores que estão engajados em corrigir *bugs* e desenvolver novas funções e também pelo fato de ser um motor de jogos e um framework totalmente flexível, podendo então dessa forma, ser utilizado através de IDE complexas ou até mesmo com simples editores de texto. Esse motor de jogos é desenvolvido totalmente em Java, não sendo necessário, portanto, aprender uma nova linguagem de programação, e por esse motivo, o motivo de ser desenvolvido em Java, é possível portar o projeto para vários sistemas operacionais, como por exemplo Linux ou macOS, sendo que ele também faz o uso de *shaders* e é totalmente baseado em OpenGL, logo, mais uma vez de código aberto, isso dá a possibilidade ao desenvolvedor de alterar o que ele desejar no motor de jogos, tornando o JME totalmente customizável.

3.3.1. LWJGL

Lightweight Java Game Library ou apenas LWJGL é uma eficiente, mas também leve biblioteca, para a linguagem de programação Java com o foco em desenvolvimento de aplicações em três dimensões [16]. Através dessa biblioteca é possível obter acesso a outras bibliotecas de código aberto, como por exemplo, OpenGL e OpenAL, através do LWJGL já é possível obter acesso a biblioteca OpenGL com suas principais extensões na versão 2.1 e também OpenAL em sua versão 1.0 [16]. Infelizmente, o LWJGL não é uma biblioteca que faz o desenvolvimento de aplicações em três dimensões tornar-se simples, pois o LWJGL fornece funções que os programadores Java podem não estar acostumados a utilizar [16]. Além do gerenciamento gráfico, o LWJGL também suporta o gerenciamento de dispositivos de entrada, como por exemplo, mouse, teclado e *joysticks* [16]. Portanto o LWJGL representa uma alternativa que não está fechada apenas para uma plataforma, ou seja, por ser baseado em OpenGL e Java, o LWJGL pode ser executado em praticamente todos os sistemas operacionais [16]. O LWJGL também possui suas próprias bibliotecas para exibição ao usuário, ou seja, elas são independentes de AWT ou Swing [16].

Comparada a outras bibliotecas, o LWJGL apresenta vários benefícios que transformam a linguagem de programação Java em uma plataforma capaz de desenvolver jogos ou aplicações gráficas em três dimensões, como:

- Velocidade de execução - O LWJGL basea-se em OpenGL, e na linguagem de programação C, suas funções são executadas de forma eficiente, porém em Java, as coisas não funcionam exatamente como em C, portanto alguns métodos, por serem classificados como pesados, foram removidos dessa biblioteca, como por exemplo, o método *glColor3v* [16].
- Portabilidade - Essa biblioteca foi projetada para ser executada em dispositivos que não contém muitos recursos, como por exemplo, celulares ou computadores que não possuem um hardware tão potente, porém atualmente, ainda não existem celulares potentes que podem executar jogos 3D baseados nessa biblioteca, apesar disso, essa biblioteca foi projetada para que no futuro, por exemplo, novos celulares, possam executar essa biblioteca sem problemas [16].
- Simplicidade - Essa biblioteca foi desenvolvida de forma que, tanto para iniciantes, quanto para desenvolvedores que já possuem experiência em desenvolvimentos de aplicações gráficas, o desenvolvimento torne-se simples [16]. Além disso, um modelo foi desenvolvido para que todos os dispositivos consigam executar a aplicação da mesma forma, sejam eles um celular ou computador [16].
- Minimalismo - Com a utilização do LWJGL, por fator da biblioteca ser pequena, ela torna-se portável, pois uma minimilização da biblioteca faz com que as funções sejam melhor compreendidas e melhor utilizadas, além de também diminuir os erros presentes no código [16].
- Segurança - LWJGL é desenvolvido inteiramente em Java, sendo assim, essa biblioteca funciona através de uma linguagem de mais alto nível, não possuindo dessa forma ponteiros ou *buffers* [16]. Tamanho de *arrays* são automaticamente limitados, de forma a promover a segurança, ou seja, os tamanhos desses *arrays* são automaticamente escolhidos, para que os valores se encaixem perfeitamente nos objetos utilizados, evitando assim o problema de vazamento de memória [16].
- Robustez - Diferentemente de outras linguagens de programação, o Java consegue tratar e também evitar exceções de forma elegante e nativa, função esta que foi herdada pelo LWJGL, promovendo então uma melhor e mais eficaz forma de exibir mensagens de erro para os usuários.

Por padrão, o LWJGL já vem implementado baseando-se nas bibliotecas desenvolvidas em C, são elas a OpenGL e também a OpenAL, mas nada impede que os desenvolvedores utilizem outras bibliotecas escritas em Java [16].

3.3.2. SpiderMonkey Networking API

SpiderMonkey Networking API é utilizada quando um jogo desenvolvido com a jMonkeyEngine necessita que vários jogadores tenham a possibilidade de estarem simultaneamente conectados ao mundo do jogo [17]. Essa API utiliza a arquitetura de conexão cliente-servidor [17], de forma que os clientes estarão sempre conectados a um servidor e uma cópia do mundo do jogo será replicada para cada um dos clientes conectados.

Cada cliente pertencente ao mundo do jogo irá enviar primeiramente ao servidor as informações do seu respetivo cliente, após o servidor receber essas informações do seu último estado, ele as replica para os outros clientes, mantendo dessa forma uma sincronização, para que todos os clientes compartilhem o mesmo mundo do jogo [17]. Após o processo de sincronização, cada cliente exibirá um respectivo estado do jogo de acordo com a perspetiva do jogador [17].

Essa API contém um conjunto de classes e interfaces que estão presentes no pacote *com.jme3.network* [17], removendo dessa forma a necessidade dos desenvolvedores implementarem do zero uma biblioteca específica para a comunicação *multiplayer*, sendo então possível criar um servidor e clientes, gerenciar as mensagens recebidas e enviadas sem maiores complicações.

3.3.3. *Listeners*

Os *Listeners* são utilizados para gerenciar eventos e funções específicas dos clientes e de algumas funções do servidor, sendo possível utilizar vários *listeners* para notificar a ocorrência de certos eventos. Um exemplo de *listener* é o *MessageListeners* que é utilizado pelo cliente e pelo servidor para receber notificações quando novas mensagens são recebidas, pelo que é possível utilizar esse recurso de *listeners* de forma customizada, para

receber notificações apenas sobre certos tipos de mensagens [17]. O *ClientStateListeners* é utilizado para informar o cliente das alterações ocorridas em seu estado de conexão como, por exemplo, quando ele é expulso do servidor [17]. *ConnectionListeners* tem como função avisar ao servidor quando um cliente se conecta ou se desconecta. O *listener ErrorListeners* avisa ao cliente sobre os possíveis problemas que podem ocorrer na rede como, por exemplo, caso se desligue, pelo que o *listener* dispara um evento avisando sobre o problema e que ação deve ser tomada em relação a isso [17].

3.4. JGroups

JGroups é um *toolkit* que implementa a técnica de comunicação em grupo. Esse *toolkit* baseia-se em IP *multicast*, sendo que vários protocolos de comunicação podem ser utilizados como, por exemplo, o TCP e o UDP. A configuração do tipo de protocolo é efetuado diretamente no ficheiro de configuração XML, dispensando a alteração do código [18]. Esses protocolos de comunicação são estendidos para serem utilizados para a criação de grupos de forma confirmada, ou seja, as mensagens devem ser enviadas para todos os participantes do grupo de forma que exista a retransmissão de mensagens, caso necessário [18]. A ordenação dos pacotes também é suportada, mas se o protocolo TCP for utilizado essa propriedade já é automaticamente executada. Se for usado o UDP, o JGroups se encarregará de garantir que os pacotes enviados na ordem A, B sejam entregues na ordem correta e não em B, A [18]. A propriedade de atomicidade também esta presente nesse *toolkit*, ou seja, o JGroups encarrega-se de que caso uma mensagem seja enviada e se um membro do grupo receber a mensagem, então todos os participantes do grupo devem recebê-la, se isso não acontecer, então nenhum participante do grupo deve receber a mensagem [18].

O JGroups também já possui implementado funções de notificação dos membros, ou seja, é possível conhecer todos os membros que estão conectados ao grupo e quando um membro do grupo se conecta ou se desconecta [18]. Esse *toolkit* suporta modos de comunicação *One-To-One* (*unicast*) e *One-To-Many* (*multicast*) [18].

Além desses recursos, o recurso mais poderoso do JGroups é a sua pilha de protocolos flexível, que dá a possibilidade aos desenvolvedores de a adaptar para corresponder exatamente às necessidades de seu aplicativo e também às características pretendidas para a rede [18]. Esta possibilidade de combinar protocolos permite flexibilizar a aplicação de forma a se adaptar aos requisitos da aplicação, nomeadamente, em termos de *unicast*, *multicast*, gestão de nós, identificação de falhas, etc. [18].

3.5. Serialização e deserialização de objetos em Java

Serializar um objeto significa transformar o seu código em uma sequência de *bytes* ordenada para que possa, posteriormente, ser recuperado pela operação simétrica [19].

A serialização e deserialização de objetos é utilizada para persistir objetos [19], ou seja, salvar os objetos em um armazenamento externo, para uma possível utilização posterior, e para enviar objetos através de uma rede de computadores [19].

Para que esse processo seja executado pode-se implementar métodos manuais para a conversão de objetos para uma sequência de *bytes* e também para o processo inverso, porém a incorreta implementação desses algoritmos pode afetar a performance da máquina, sendo que para tal objetivo pode-se utilizar a classe *SerializationUtils* [20] já implementada no projeto *Apache Commons* ou então as classes nativas do Java.

Para utilizar as capacidades de serialização nativas do Java basta que os desenvolvedores implementem a Interface *java.io.Serializable* nos objetos que posteriormente serão serializados e/ou deserializados [21]. Isso torna o processo de desenvolvimento muito mais ágil, visto que essa Interface não contém métodos, logo não é necessário implementar mais nenhuma linha de código [21].

No contexto de jogos, a serialização de objetos permite enviar e receber estruturas de dados complexas entre os participantes num jogo *multiplayer*.

Capítulo 4 Desenvolvimento

Nesse capítulo será explicado o desenvolvimento do trabalho, assim como os pontos mais importantes da implementação que fizeram com que o trabalho cumprisse com seu objetivo. Detalhes mais aprofundados sobre a implementação e seus diagramas podem ser visualizados nos anexos.

4.1. Introdução

Um jogo *multiplayer* depende de funcionalidade de comunicação, que pode assumir uma arquitetura centralizada ou distribuída. Um módulo de conexão centralizado geralmente utiliza a arquitetura de conexão cliente-servidor, onde o servidor é o nó que processa todas as requisições enviadas pelos clientes, gera as respostas e então as envia para os clientes que ele julgar necessário. Dessa forma cada cliente conterá uma cópia do mundo do jogo do servidor.

Quando um cliente executa alguma ação local como, por exemplo, movimentar um personagem, uma requisição é enviada para o servidor. Este irá verificar a ação excutada, processá-la, gerar os resultados e enviar uma resposta contendo atributos do mundo do jogo atualizados para todos os clientes que o servidor achar necessário. Como último passo, os clientes recebem essa resposta e atualizam o estado do jogo.

Seguindo uma arquitetura descentralizada (tipicamente P2P), cada *peer* pertencente ao grupo irá estabelecer uma conexão direta com os outros *peers*, removendo a necessidade de

existir um servidor dedicado. Cada membro do grupo irá controlar e gerenciar o seu próprio mundo do jogo.

No momento em que um cliente executa uma ação, ela será processada localmente, podendo dessa forma utilizar qualquer parâmetro que lhe for conveniente. Após o processamento local das informações pelo *peer* em questão, as informações desse processamento são enviadas para os outros clientes, atualizando o seu estado. Com isso, cada *peer* terá um mundo do jogo individual que será parecido com o mundo de jogo dos outros *peers*.

Nos tópicos seguintes será apresentado a descrição da implementação do módulo de conexão denominado JEagle, desenhado para implementar ambas as arquiteturas de conexão cliente-servidor e P2P. Para os testes desse módulo foi utilizado um jogo já implementado chamado de Monkey Blaster desenvolvido por Daniel Gallenberger, e adaptado que que fosse possível testar o módulo de conexão JEagle.

O foco dessa descrição será apresentar uma forma de criar a comunicação e sincronização dos nós presentes na rede utilizando o módulo de conexão JEagle.

4.2. Módulo de comunicação centralizado

É necessário compreender alguns conceitos iniciais antes de iniciar o desenvolvimento desse módulo de conexão, ou seja, em módulos de conexão centralizados existe um nó central, conhecido como servidor, que será responsável por intermediar a ligação de todos os outros nós conectados, que são conhecidos como clientes. O servidor será o responsável por receber todas as requisições dos clientes, processá-las e então enviar respostas para quem ele achar necessário. Dessa forma todos os dados estarão centralizados, logo o mundo do jogo estará no servidor e será replicado para todos os clientes que estiverem conectados, sendo assim todos os clientes irão compartilhar o mesmo mundo de jogo.

Quando um jogador quiser executar alguma ação no mundo do jogo, ele deve primeiramente enviar uma requisição para o servidor, o servidor irá processar essa ação,

executá-la no mundo do jogo que esta presente no servidor e então enviar as atualizações ou parte das atualizações do mundo do jogo para todos os clientes que o servidor achar necessário (Figura 7).

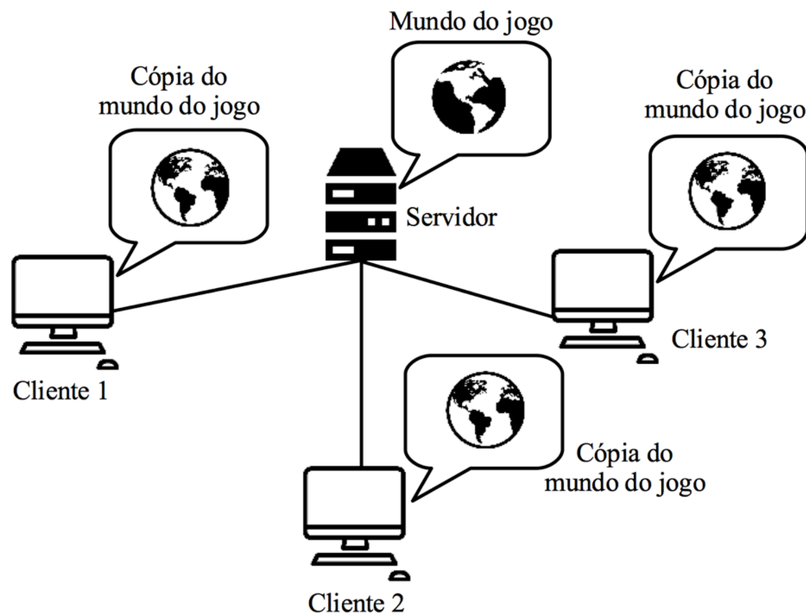

Figura 7: Módulo de conexão centralizado.

Alguns problemas podem acontecer nesse processo de replicação do mundo do jogo ou parte do mundo do jogo para os clientes que estão conectados ao servidor como, por exemplo, latência elevada. Geralmente o que se observa nesse problema é um atraso do tempo de resposta desde a ação do jogador até que a ação do personagem seja executada. Como por exemplo, se a latência da rede estiver muito alta e o personagem local tentar movimentar-se, o que ele irá visualizar é o seu personagem movimentando-se não no momento em que ele pressionar uma tecla do teclado, mas sim algum tempo depois, mostrando assim uma forma de *lag* (Figura 8).

Protocolo TCP

Protocolo UDP

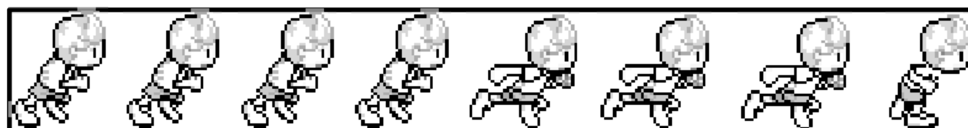

Figura 8: Exemplo de visualização de atraso de comunicação com os protocolos TCP e UDP.

O módulo de comunicação centralizado também apresenta algumas vantagens em relação aos outros módulos, pois como todo o processamento é centralizado, pode-se dizer que ele apresenta uma garantia em relação a trapaças, pois caso o servidor verifique que existe alguma informação alterada ou incorreta em algum pacote, ele será automaticamente descartado e o emissor do pacote será automaticamente identificado. Além disso, como o processamento é centralizado em um nó especifico, os jogadores não precisam conter uma supermáquina para executar os clientes, pois basicamente sua responsabilidade será apenas de renderizar os gráficos, executar o som do jogo e enviar os pacotes ao servidor com a entrada dos jogadores, nada mais.

Compreendidos os conceitos elementares sobre o módulo de conexão centralizado, de que todos os processamentos das mensagens enviadas na rede serão processados pelo servidor, e que posteriormente os clientes que o servidor achar necessário receberão uma cópia exatamente igual do mundo do jogo presente no servidor, já é possível iniciar os passos iniciais sobre o desenvolvimento desse módulo de comunicação.

É importante executar um levantamento das funcionalidades básicas que esse módulo de comunicação deverá apresentar nas próximas etapas do desenvolvimento. O processo utilizado para desenvolver essas funcionalidades que estarão presentes no JEagle será através da utilização de *sockets* e do envio e recebimento de pacotes para o protocolo de

comunicação TCP, além de datagramas para o protocolo de comunicação UDP. As seguintes funcionalidades por defeito já estão presentes no protocolo TCP:

- Executar a fragmentação de mensagens consideradas grandes;
- Executar a ordenação dos pacotes;
- É criado um canal de comunicação;
- Possível reconhecer quando um cliente se conecta ou se desconecta;
- A devida ordenação dos pacotes;
- Controle do trafego existente na rede.

Portanto as funções descritas acima já não precisam mais ser implementadas utilizando o protocolo TCP, pois já estarão presentes por omissão. Em contrapartida, algumas funcionalidades não estarão presentes quando for ser utilizado o protocolo UDP.

Os protocolos terão diferentes funcionalidades já pré-implementadas, porém as seguintes funcionalidades básicas deverão estar presentes em ambos os protocolos, são elas:

- Enviar e receber mensagens;
- Detetar falhas em clientes, desconectando dessa forma, os que foram a baixo;
- Conter uma lista de clientes conectados ao servidor;
- Compressão de mensagens.

4.3. Módulo de comunicação descentralizado

Antes de começar o desenvolvimento desse módulo de comunicação é importante compreender alguns conceitos importantes no desenvolvimento de módulos de conexão descentralizados, ou seja, sem um nó central para processar os pedidos e retornar as respostas. É, portanto, necessário que cada *peer* contenha um mundo do jogo único e que cada mudança importante no seu mundo de jogo executada pelo jogador local seja enviada para todos os outros *peers* conectados ao grupo. Por esse motivo o jogador irá observar dois tipos diferentes de atores no jogo, um ator principal que será controlado pelo jogador local e outros atores que irão simular as ações importantes que são executadas pelos outros *peers* (Figura 9).

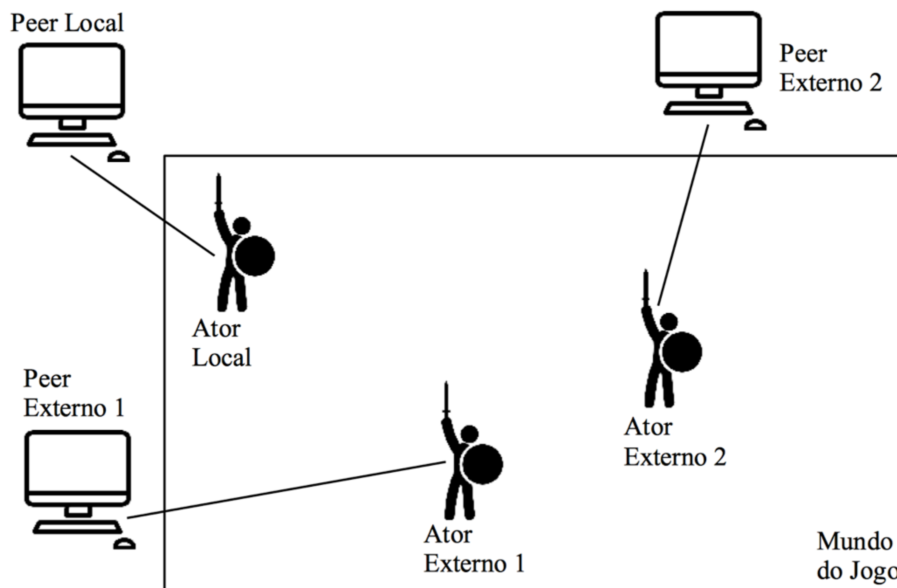

Figura 9: *Peer* local e *peers* externos.

As ações do jogador local são executadas instantaneamente, de forma que o mundo do jogo é atualizado no momento em que o jogador executar uma ação, então após esse processo, o *peer* local enviará uma mensagem aos outros *peers*, informando-os de que uma ação foi executada para que eles possam simular a ação do *peer* remoto em seu próprio mundo do jogo. É importante observar que quando um *peer* recebe uma mensagem através da rede, o *peer* emissor pode já ter terminado de executar essa ação, e provavelmente já estará executando outra ação. A latência da rede somada com essas ações recebidas em tempos diferentes faz com que o mundo do jogo em cada *peer* seja semelhante, mas não idêntico.

Para diminuir essa diferença que o mundo do jogo de cada *peer* apresenta, o *peer* local após executar uma ação não deve executá-la localmente e então enviar uma mensagem para os outros *peers* simularem sua ação, ao invés disso, o *peer* local deve reconhecer a ação que deverá ser executada e então enviar um pacote em *broadcast* para todo o grupo, inclusive para ele mesmo. Dessa forma todos os *peers* do grupo, inclusive o *peer* local, irão receber a mensagem para executar a ação praticamente no mesmo tempo, lembrando que o tempo de recebimento do pacote esta diretamente relacionado com a latência da rede. Com isso é possível diminuir relativamente a diferença entre o mundo de jogo de cada *peer*. Para que o mundo de jogo de cada *peer* fosse idêntico seria necessário que não existisse a latência

presente na rede, dessa forma, a ação executada no *peer* local e a simulação executada nos outros *peers* presentes no grupo seriam executadas exatamente no mesmo tempo, não apresentando então diferenças.

Com o aumento da latência, a representação do mundo do jogo em cada *peer* torna-se diferente, como por exemplo, quando um *peer* se movimenta e então envia uma mensagem para os outros *peers* presentes no grupo avisando que a simulação de movimento deve ser executada e a latência da rede mostra-se muito alta, geralmente o que os jogadores dos *peers* irão visualizar é um "teletransporte" de uma posição A para B na simulação de movimentação de um *peer* externo. Esse problema acontece porque quando a mensagem chega ao *peer* receptor, o *peer* emissor já esta movimentando-se em outra posição, e então a outra posição é enviada novamente, substituindo dessa forma, a posição antiga que esta sendo mostrada, como essa substituição acontece quase que de forma instantânea, o jogador acaba visualizando esses problemas (Figura 10).

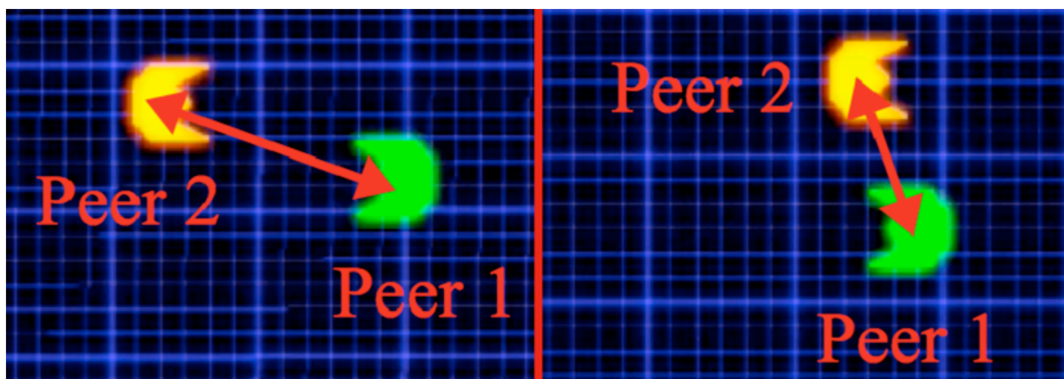

Figura 10: Posição diferentes dos jogadores no jogo Monkey Blaster

Assim, dois *peers* pertences ao mesmo grupo no jogo Monkey Blaster, apresentam seu mundo individual do jogo, em que cada representação observa que a distância e a posição dos jogadores não são as mesmas, isso ocorre devido aos problemas de sincronização e de latência da rede descritos acima. Existem vários algoritmos que utilizam técnicas, como por exemplo interpolação e predição de posições que podem minimizar esse problema, porém eles não foram implementados na adaptação desse jogo, pois o foco do trabalho é o desenvolvimento do módulo de comunicação e não o jogo em si.

Agora que foi percebido que cada *peer* irá conter o seu próprio mundo do jogo, que não existe uma centralização do grupo, que a latência da rede afeta a replicação das ações em cada *peer*, que as ações de cada *peer* externo serão apenas uma simulação das ações reais, e também que a posição de cada jogador não será idêntica em cada *peer*, mas sim parecida, já é possível iniciar com os passos iniciais do desenvolvimento do módulo de conexão P2P. O primeiro passo será encontrar um *middleware* responsável por intermediar a comunicação P2P.

Para o desenvolvimento de comunicação em grupo de forma descentralizada é necessário utilizar um *middleware*, pois já existem várias funcionalidades implementadas, poupando dessa forma, tempo de desenvolvimento para tentar solucionar problemas que já foram resolvidos. Métodos complexos como por exemplo funcionalidades para descoberta de rotas e *peers* na rede já foram desenvolvidos, portanto não precisamos mais nos preocupar em implementar tais funcionalidades.

Um *middleware* é uma camada no desenvolvimento de software capaz de prover funcionalidades em segundo plano entre diferentes *software*, *frameworks*, módulos, etc., dessa forma o *middleware* escolhido para intermediar a comunicação descentralizada do JEagle é o JGroups que já foi descrito nos tópicos anteriores. Através do JGroups já não é mais necessário preocupar-se com o desenvolvimento das seguintes funcionalidades:

- Enviar mensagens utilizando os protocolos de transporte TCP e UDP;
- Executar a fragmentação de mensagens grandes;
- Desenvolver protocolos de descoberta de membros para juntar-se a um grupo;
- Retransmitir mensagens perdidas;
- Deteção de falhas em membros, excluindo dessa forma membros que foram a baixo;
- Ordenação de pacotes;
- Lista de membros que estão presentes no grupo;
- Compartilhamento de informações através da rede;
- Controle de tráfego;
- Compressão de mensagens.

Com todas essas funcionalidades já implementadas, o próximo passo é iniciar a análise das ações executadas pelos jogadores e também as ações executadas pelos *peers* que deverão

ser enviadas e recebidas pelos membros do grupo. Essa análise irá estabelecer a forma como a comunicação entre os *peers* será feita durante a execução do jogo em rede no módulo de conexão descentralizado.

4.4. Planejamento da comunicação

Um dos passos mais importantes para o desenvolvimento de módulos de comunicação é definir as ações que serão executadas pelos *peers* ou pelos clientes presentes no grupo ou no servidor. Essas ações serão divididas em duas partes. A primeira parte são as ações relevantes apenas aos *peers* ou aos clientes e, devido a isso, não estarão visíveis aos jogadores como, por exemplo, verificação de quando um *peer* ou cliente se conecta ou se desconecta do grupo, atualização da lista de membros ou clientes presentes no grupo ou no servidor, que ação tomar quando um pacote, datagrama ou mensagem for recebida, etc. A segunda parte são as ações executadas pelos jogadores que deverão posteriormente ser implementadas pelo desenvolvedor do jogo. Essa segunda parte deve ser suficientemente simples para que o desenvolvedor não se preocupe com questões relativas ao desenvolvimento do módulo de comunicação. Ações como enviar e receber ataques, movimentação dos jogadores, sala de chat, entre outros, são alguns exemplos. Portanto essas ações são responsáveis por possibilitar ao *peer* local alterar o mundo do jogo e fazer com que o jogador visualize as simulações de tais ações.

As ações relativas aos *peers* ou clientes devem ser implementadas em uma classe relacionada com a receção das mensagens e devem ser sobrescritas em uma interface, para que o desenvolvedor do jogo possa adicionar códigos relativos as ações executadas pelos jogadores.

A seguir é possível visualizar as ações básicas que devem ser sobrescritas na *interface*:

- Informar que a conexão com o grupo foi aceita;
- Informar que a conexão com o grupo foi rejeitada;
- Informar a latência do grupo;
- Adicionar o criador do grupo para o módulo de comunicação descentralizado;

- Alterar o criador do grupo para o módulo de comunicação descentralizado;
- Informar quando um novo membro ou cliente conecta-se ao grupo ou servidor;
- Informar quando um membro ou cliente desconecta-se do grupo ou servidor;
- Método para tratar as mensagens, pacotes ou datagramas relativos aos jogadores, que deve ser implementado pelo desenvolvedor do jogo e não pelos desenvolvedores do módulo de comunicação.

As ações importantes do jogo Monkey Blaster que devem ser implementadas pelo desenvolvedor do jogo são as ações de atirar, movimentar, morrer e nascer (Figura 11, Figura 12).

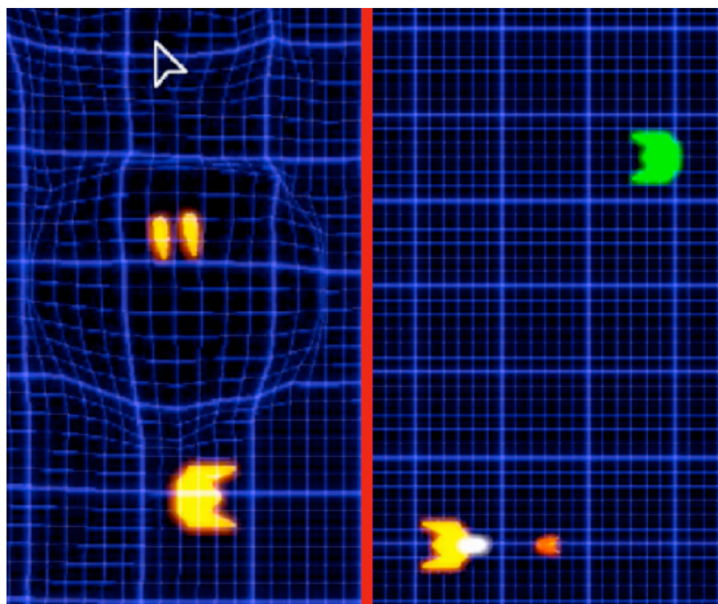

Figura 11: Ações de atirar e movimentar no jogo Monkey Blaster.

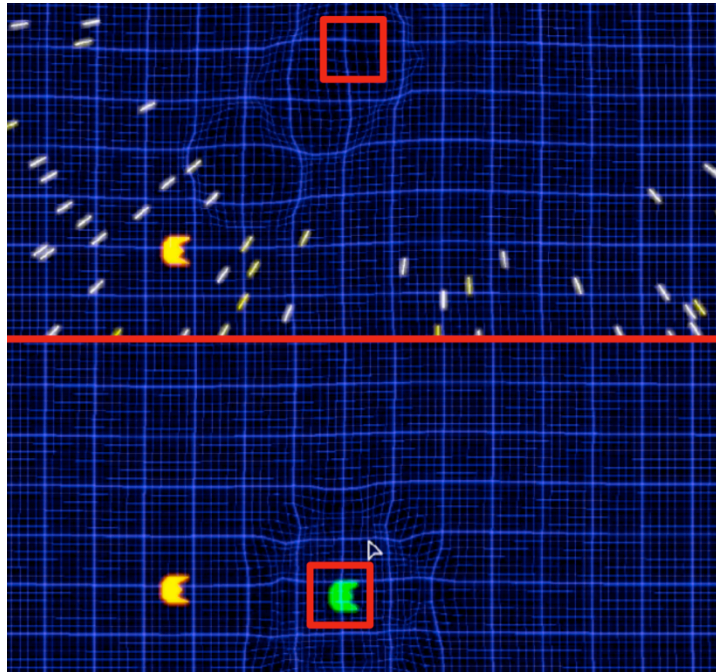

Figura 12: Ações morrer e nascer no jogo Monkey Blaster.

Cada ação mostrada acima será transmitida pelo jogador para todos os *peers* ou clientes presentes no grupo ou conectados ao servidor, portanto é importante que exista uma noção por parte do desenvolvedor do jogo sobre a quantidade de dados que serão enviados pela rede, pois quanto maior for a mensagem que um *peer* ou cliente deseja enviar para os outros *peers* ou clientes, maior será o tempo necessário para que essa mensagem seja enviada. Além do tempo físico necessário para que uma mensagem saia de um ponto A e chegue à um ponto B, o tempo de processamento da CPU, o tempo para serializar e deserializar um pacote também deve ser levado em consideração, de forma que isso possa vir a alterar a representação do mundo do jogo de *peer* no módulo de comunicação descentralizado e também possa gerar o problema de *lag* no módulo de comunicação centralizado. Assim, deve-se tentar minimizar o tempo de envio das mensagens de um *peer* ou cliente para os outros *peers* ou para outros clientes, lembrando que no módulo de comunicação descentralizado, o servidor também possui a responsabilidade de enviar os dados de forma rápida para os clientes, caso contrário problemas na execução do jogo nos clientes também acontecerá.

4.4.1. Execução das ações e simulações

Depois que as ações dos *peers*, dos clientes e dos jogadores foram mapeadas, é preciso reproduzir essas ações sem nenhuma forma de entrada local dos jogadores externos, ou seja, os jogadores conectados devem ser capazes de enviar suas ações ao jogador local sem a necessidade de estarem presentes fisicamente com o jogador local. Como por exemplo, a um primeiro contato, as ações que os personagens executam estão diretamente ligadas com as entradas do usuário, fazendo com que não seja possível utilizar o código em diferentes situações com diferentes jogadores. Porém, a partir do momento que se abstrai o conceito de jogo em rede, é possível perceber que removendo o código da lógica de entrada, utilizando segmentação de código e considerando cada personagem do jogo como um ator, é possível utilizar o mesmo código para diferentes atores, bastando apenas considerá-los como uma referência de ator nos parâmetros dos métodos em questão. Além desse processo remover alguns problemas futuros, ele também segue os conceitos de engenharia de *software* para evitar a replicação de código (Figura 13).

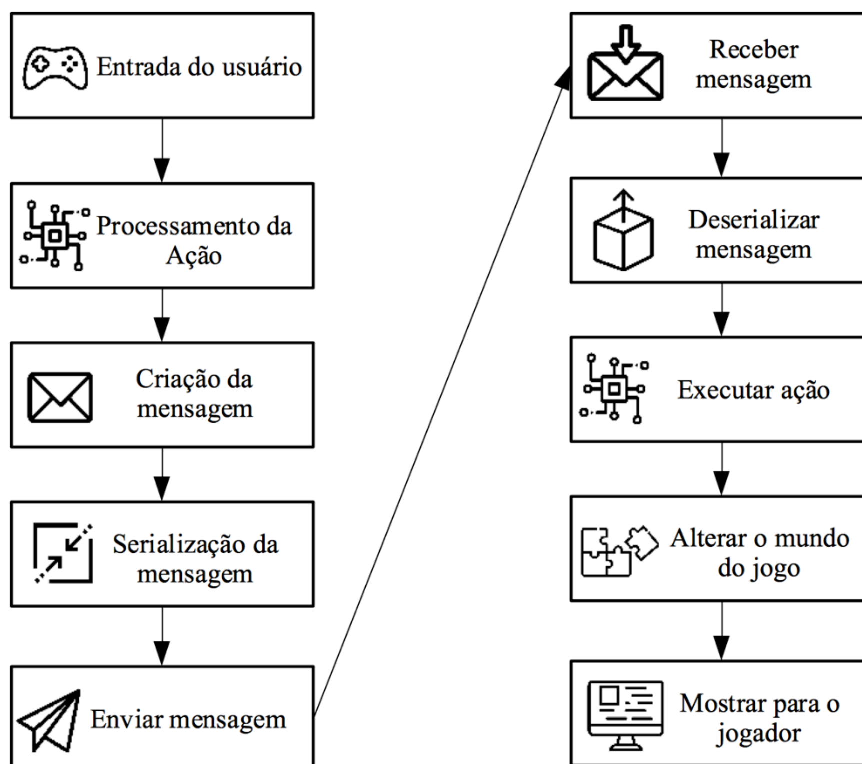

Figura 13: Passos para a execução de ações.

Para iniciar esse processo de replicação de ações é necessário que o usuário execute uma entrada através de um dispositivo. As ações executadas através do dispositivo de entrada serão processadas e uma mensagem deverá ser criada com as informações dessa ação. Posteriormente essa mensagem deve ser serializada e então enviada para todos os *peers* do grupo ou para os clientes conectados ao servidor. A partir do momento em que os *peers* ou os clientes receberem essa mensagem, ela deverá ser deserializada e transformada novamente em um objeto, executando, de seguida, a ação e atualizando o estado do mundo do jogo.

As classes responsáveis por tratar as ações do jogador contêm apenas informações necessárias para descrever o jogador, além de dos métodos responsáveis por manipular os seus atributos, como as informações de sua posição, quantidade de vida, etc. Baseando-se na possibilidade de abstrair a classe do jogador como uma referência para um ator, um ator é uma superclasse de um jogador, os códigos de multijogador podem executar as alterações nos atributos dos jogadores da mesma forma que os dispositivos de entrada dos jogadores fazem, resultantes das mensagens recebidas.

Ao final da compreensão de como as ações e simulações serão executadas nos *peers* presentes do grupo e nos clientes conectados ao servidor, é necessário começar a planejar como os dados serão enviados de um *peer* para outro *peer* ou de um cliente para outro cliente, ou seja, como a estrutura, os atributos e métodos do objeto "mensagem" serão desenvolvidos. Essas informações serão tratadas no próximo tópico.

4.4.2. Enviando e recebendo mensagens

Para que a comunicação entre *peers* e entre os clientes seja estabelecida, é necessário que uma convenção que controle a troca de mensagens seja criada, ou seja, antes que qualquer dado seja trocado, é preciso desenvolver um protocolo de comunicação que defina uma quantidade de regras que sejam capazes de descrever como um objeto "mensagem" será criado e posteriormente lido e compreendido por todos os integrantes do grupo. Portanto

para que o sistema de troca e recebimento de mensagens no módulo de comunicação centralizado e descentralizado funcione adequadamente, as seguintes classes foram criadas:

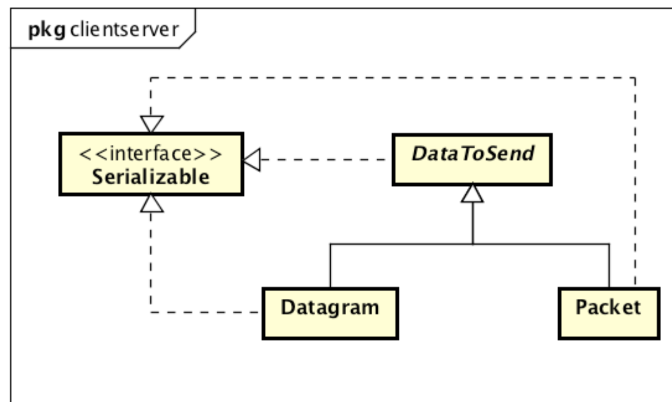

Figura 14: Classes *Datagram* e *Packet* do módulo de comunicação centralizado.

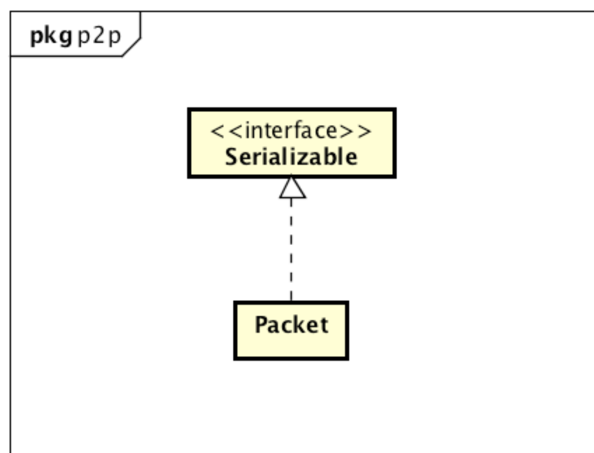

Figura 15: Classe *Packet* do módulo de comunicação descentralizado.

Os atributos necessários para que uma mensagem seja compreendida entre os *peers* e pelos clientes são basicamente quatro, o primeiro e um dos mais importantes é o atributo *header*. Esse atributo será responsável por informar o *peer* ou o cliente recetor qual o tipo de ação que terá de executar quando a mensagem chegar. O atributo *content* armazena o objeto que será trocado entre os *peers* ou os clientes, portanto é muito importante que esse atributo seja do tipo *Object*, para que qualquer objeto possa ser enviado e também recebido. O atributo

srcMember no módulo de comunicação descentralizado e o atributo *uuidSrc* no módulo de comunicação centralizado armazenam as informações do *peer* ou do cliente que enviou a mensagem como, por exemplo, seu endereço e identificador único no grupo ou no servidor, da mesma forma o atributo *dstMember* no módulo de comunicação descentralizado e o atributo *uuidDst* no módulo de comunicação centralizado armazena as informações do *peer* e do cliente destino.

Como por exemplo, no desenvolvimento de um sistema simples de *chat* que troque mensagens de texto, podem-se utilizar as classes descritas acima para enviar e receber as mensagens. Para enviar as mensagens de texto, o primeiro atributo da classe *Packet* que devemos adicionar um valor é o *header*, com uma *String* como "MENSAGEM_TEXTO", para que os *peers* ou os clientes que receberem essa mensagem já estejam cientes que o objeto recebido contém dentro do atributo *content* um objeto que contém a mensagem de texto. No atributo *content* deve-se adicionar um objeto que contenha a devida mensagem de texto, esse objeto que conterá a mensagem de texto pode ser instanciado desde a seguinte classe presente na Figura 16.

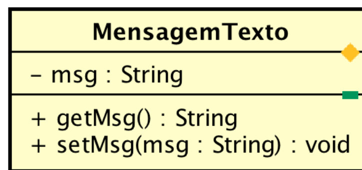

Figura 16: Classe *MensagemTexto*.

Portanto, é necessário criar uma classe que contenha a mensagem de texto, armazenada no atributo *msg*. É possível enviar outros tipos de dados, como por exemplo imagem, vídeo ou áudio.

Depois de instanciar a classe *MensagemTexto*, adiciona-se um objeto do tipo *String* no atributo *msg* para então adicionar a classe *MensagemTexto* no atributo *content* do objeto do tipo *Packet* que será enviado para os *peers* ou então no objeto do tipo *Packet* ou *Datagram* que será enviado para os clientes. Agora que o atributo *content* já esta preenchido, o próximo passo é preencher o atributo *srcMember* com as informações do *peer* ou o atributo *uuidSrc* com as informações do cliente que está enviando a mensagem, e, da mesma forma,

preencher o atributo *dstMember* com as informações do *peer* ou então o atributo *uuidDst* com as informações do cliente para o qual a mensagem deverá ser enviada. Também se pode deixar o atributo com as informações para que *peer* ou cliente a mensagem deve ser enviada, ou *NULL* para que a mensagem seja enviada para todos os *peers* presentes no grupo ou clientes conectados ao servidor. Como último passo, deve-se então serializar o objeto do tipo *Packet* que será enviado para os *peers* ou então o objeto do tipo *Packet* ou *Datagram* que será enviado para os clientes, para que possa finalmente ser enviado. Com todos os passos completos, o objeto *Packet* do módulo de comunicação descentralizado ou então o objeto do tipo *Packet* ou *Datagram* do módulo de comunicação centralizado já esta pronto para ser enviado para um *peer* ou cliente especifico ou então em *broadcast* para todos os *peers* presentes no grupo ou clientes conectados ao servidor.

Esse exemplo é essencial para compreender como o processo de enviar e receber mensagens funciona, pois deve-se seguir a mesma lógica para que os *peers* ou clientes possam compreender e ler as mensagens que serão trocadas no grupo ou no servidor.

Tomando como base o exemplo acima, mas abstraindo a aplicação não mais para uma aplicação de *chat*, mas sim para o controle dos *peers* presentes no grupo ou os clientes conectados ao servidor, é o momento de criar *headers* específicos que serão capazes de obter informações do grupo ou do servidor, como por exemplo, informações sobre quando um membro ou cliente se junta ou se desconecta do grupo, ou quando é necessário atualizar a lista de membros presentes no grupo ou a lista de clientes conectados ao servidor.

Também é importante perceber que as mensagens funcionam em duas vias, ou seja, sempre quando uma mensagem enviada estiver requisitando algo dos *peers* ou clientes, eles devem enviar uma outra mensagem de resposta, contendo dessa forma os dados requisitados.

4.4.3. *Headers* presentes nos módulos de comunicação centralizado

Os *headers* das mensagens que serão enviados pelo servidor e então processados pelos clientes utilizando o protocolo TCP são:

- *CONNECTION_SUCESS_RESPONSE* - Essa mensagem recebida pelo servidor representa uma resposta de mensagem bem-sucedida, ou seja, avisando ao cliente

que sua conexão foi aceita com sucesso. Essa mensagem também tem em seu atributo *content* um objeto que contém o identificador único do cliente. Após o recebimento dessa mensagem serão enviadas duas requisições ao servidor, a primeira é pedindo que o servidor envie uma lista dos clientes conectados e também um pedido para correção da contagem do *ping*;

- *CONNECTION_REFUSED_RESPONSE* - O cliente recebe essa mensagem quando o servidor é alcançável e é possível estabelecer uma conexão, porém o servidor já possui o número máximo de clientes conectados, fazendo com que não seja mais possível aceitar nenhuma conexão, portanto o cliente em questão irá receber essa mensagem;
- *GET_CLIENTS_LIST_RESPONSE* - Nessa mensagem o cliente receberá uma lista dos clientes conectados no servidor e através dessa lista ele também verificará se algum cliente conectou-se ou desconectou-se do servidor, mas essa verificação só será executada se o cliente já conter uma lista desatualizada de clientes;
- *GET_PING_RESPONSE* - Essa mensagem recebida do servidor contém um objeto do tipo *Ping* com o tempo decorrido desde o envio da requisição do *ping* até a chegada da sua resposta pelo servidor;
- *CLIENT_ACTION_JOIN* - Quando o cliente recebe essa mensagem do servidor significa que um novo cliente conectou-se, então o que o cliente faz é pedir novamente uma lista atualizada de clientes, para posteriormente verificar qual cliente conectou-se;
- *CLIENT_ACTION_LEFT* - Segue a mesma ideia do *header* a cima, porém agora o servidor avisa que um cliente desconectou-se do servidor;
- *GET_PING_RESPONSE_FIX* - É a resposta do servidor para a correção da contagem de *ping*, pois quando envia-se uma requisição para o servidor de contagem de *ping* pela primeira vez, a resposta que chegar do servidor conterá um valor muito alto não condizente com a real situação da rede, portanto executa-se esse pequeno ajuste no processo de iniciação do cliente, para que nas contagens de *ping* futuras esse problema não esteja mais presente;
- *SERVER_INFORMATIONS_UPDATE* - A mensagem que conter esse *header* terá em seu atributo *content* um objeto do tipo *ServerInformations* que possui as

informações relativas ao servidor, dessa forma quando o cliente receber essa mensagem será possível atualizar as informações do servidor no cliente, como por exemplo seu identificador único na rede, ou *uuid*, e também o nome do servidor.

Os *headers* das mensagens que serão enviados pelos clientes e processados pelo servidor utilizando o protocolo TCP são apenas três, pois o protocolo TCP é orientado a conexões e não existe a necessidade de reimplementar esses métodos, logo os seguintes *headers* são:

- *GET_CLIENTS_LIST* - Quando o servidor recebe essa requisição de um cliente específico, ele deve enviar uma mensagem com o *header* "GET_CLIENTS_LISTS_RESPONSE" para o cliente que requisitou essa mensagem. A mensagem deve conter em seu atributo *content* uma lista com as informações dos clientes que estiverem conectados ao servidor;
- *GET_PING* - Essa mensagem é apenas a requisição da contagem de *ping* enviada por um cliente específico, o servidor deve apenas receber essa mensagem e envia-la novamente para o cliente requisitante;
- *GET_PING_FIX* - Assim como o *header* explicado acima, o servidor deve re-enviar essa mensagem para o cliente requisitante, a única diferença aqui, é que essa mensagem diz respeito a correção da verificação de *ping*, que em um primeiro momento não faz a contagem correta da latência.

Os *headers* das mensagens que serão enviados pelo servidor e então processados pelos clientes utilizando o protocolo UDP são:

- *CONNECTION_ACCEPT* - O cliente que receber uma mensagem que contenha esse *header* teve sua conexão aceita, o servidor envia essa mensagem para os clientes quando ele conter espaço suficiente para clientes conectarem-se. Após o cliente receber essa mensagem, ele envia a seguinte mensagem para o servidor "CONNECTION_ACCEPT_END_STEP", para que o próximo e último passo da conexão seja executada;
- *PING_FIX_RESPONSE* - Essa mensagem é uma resposta do servidor para a requisição da correção da contagem de *ping* de um determinado cliente. Esse passo é executado para corrigir a primeira contagem de *ping* entre o cliente e o servidor, que normalmente não condiz com a real situação da latência que esta presente na rede, portanto é necessário corrigi-la;

- *GET_PING_RESPONSE* - A mensagem que contém esse *header* é uma resposta a requisição de *ping* de um determinado cliente, ou seja, o cliente envia uma mensagem para o servidor com um objeto do tipo *ping* em seu atributo *content*, nesse objeto *ping* existe um atributo que armazena o momento em que o objeto foi enviado para o servidor. Assim que o servidor receber essa mensagem, ele envia automaticamente uma resposta para o cliente com o mesmo conteúdo da mensagem que foi posteriormente recebida pelo servidor, de forma a preservar o objeto *ping* que foi recebido. Assim que o cliente receber essa mensagem, ele adiciona ao objeto *ping* recebido o tempo em que a mensagem foi recebida, e através do tempo que a mensagem foi enviada e recebida é possível calcular a latência da rede;
- *CONNECTION_ACCEPT_END_STEP_RESPONSE* - Quando o cliente recebe uma mensagem que contém esse *header* do servidor significa que seu processo de conexão foi finalizado, sendo que após o cliente receber essa mensagem ele atualiza as suas informações do servidor, como por exemplo o *uuid* do servidor, o nome do servidor e a lista de clientes que estão conectados ao servidor. Além disso o cliente também altera seu *status* para conectado e envia uma mensagem para o servidor com o *header* "PING_FIX", para que a primeira contagem da latência da rede, que geralmente não condiz com a realidade, seja executada, de forma que as outras contagens do *ping* sejam então executadas de forma correta;
- *CONNECTION_REFUSED* - Quando o cliente recebe essa mensagem do servidor significa que sua conexão foi recusada, esse problema ocorre porque não existe mais lugares livres para estabelecer conexões com o servidor;
- *DISCONNECTION_ACCEPT* - Essa mensagem recebida do servidor significa que o cliente teve sua desconexão aceita, e pode então ser finalizado;
- *CLIENT_JOIN* - Quando o cliente recebe uma mensagem do servidor que contenha esse *header*, significa que um novo cliente conectou-se ao servidor, portanto o cliente deve imediatamente requisitar ao servidor uma nova lista atualizada dos clientes que estão conectados com o seguinte *header* "CLIENTS_LIST_REQUEST";
- *CLIENT_LEFT* - Essa mensagem significa que um cliente desconectou-se do servidor, no momento em que o cliente receber essa mensagem ele deve enviar uma

outra mensagem ao servidor requisitando uma lista dos clientes que estão conectados atualizada com o seguinte *header* “CLIENTS_LIST_REQUEST”;

- *CLIENTS_LIST_REQUEST_RESPONSE* - Essa mensagem enviada pelo servidor contém em seu atributo *content* uma lista dos clientes conectados. Sendo que depois do momento em que a lista é recebida, o cliente verifica se ocorreu alguma alteração na lista, em caso afirmativo, ele checa se algum cliente conectou-se ou desconectou-se do servidor, informando então posteriormente ao usuário essa alteração na lista.

Diferentemente do protocolo TCP, os *headers* das mensagens que serão enviados pelos clientes e processados pelo servidor utilizando o protocolo UDP serão sete ao todo:

- *CONNECTION_ATTEMPT* - Quando o servidor recebe uma mensagem que contenha esse *header* significa que um cliente esta tentando estabelecer uma conexão. Para que a conexão seja aceita é necessário que exista espaço suficiente no servidor para que o cliente conecte-se, em caso afirmativo o servidor irá enviar uma mensagem de resposta para o cliente com o seguinte *header* “CONNECTION_ACCEPT”, caso contrário o servidor irá enviar uma mensagem para o cliente com o seguinte *header* “CONNECTION_REFUSED”;
- *PING_FIX* - Essa é uma requisição para que o servidor envie novamente a mensagem de *ping* para o cliente emissor, porém as mensagens que contém esse *header* são executadas geralmente no inicio do processo de conexão do cliente, pois a primeira contagem da latência não condiz com a real latência da rede, de forma que é necessário executar esse pequeno ajuste;
- *GET_PING* - É apenas uma requisição da contagem de latência do cliente até o servidor. Assim que o servidor recebe essa mensagem de *ping* ele deve enviar novamente essa mensagem para o cliente para que seja possível calcular o tempo de resposta para que uma mensagem seja enviada de um ponto A até um ponto B, e que essa mesma mensagem volte de B para A;
- *CONNECTION_ACCEPT_END_STEP* - Quando o servidor recebe essa requisição significa que o cliente teve sua conexão bem estabelecida com o servidor, e que agora é necessário executar o último passo para o termino desse processo, o último passo é então enviar uma resposta para o cliente com o seguinte *header* “CONNECTION_ACCEPT_END_STEP_RESPONSE”, que conterá então as

informações do servidor, como *uuid*, nome e também lista dos clientes que estão conectados ao servidor, além disso também será enviado uma outra mensagem para o servidor com o *header* "CLIENT_JOIN", avisando então que esse cliente que requisitou o último passo da sua conexão, conectou-se ao servidor;

- *DISCONNECTION_ATTEMPT* - Essa mensagem significa que o cliente teve desconexão aceita pelo servidor, e então uma mensagem com o *header* "CLIENT_LEFT" será enviada em *broadcast* para todos os clientes que estiverem conectados ao servidor, além disso o cliente que requisitou sua desconexão será removido da lista de clientes conectados;
- *CLIENTS_LIST_REQUEST* - A mensagem que contém esse *header* significa uma requisição de um cliente específico da lista de clientes conectados no servidor, assim que o servidor recebe essa requisição, ele envia uma lista dos clientes que estão conectados para o cliente requisitante através de uma mensagem que contém o seguinte *header* "CLIENTS_LIST_REQUEST_RESPONSE";
- *SERVER_REACHABLE* - É uma mensagem para verificar se o servidor é alcançável, ou seja, para verificar se um servidor esta online em um determinado endereço, em caso afirmativo, uma mensagem com o seguinte *header* "SERVER_REACHABLE_RESPONSE" será enviada ao cliente requisitante, afirmando que o servidor existe e que esta aceitando conexões.

Diferente do módulo de comunicação descentralizado, o módulo de comunicação centralizado necessita de um número maior de *headers*, pois nenhum *middleware* é utilizado, de forma que todo o controle dos clientes e como o servidor os gerência deve ser planejado e implementado pelo desenvolvedor do módulo de comunicação, lembrando que eles têm de ser transparentes, para que os jogadores ou o desenvolvedor do jogo não percebam de forma direta que esse controle esta sendo executado.

Agora que os todas as mensagens para criar uma comunicação básica entre os clientes utilizando os protocolos de comunicação TCP e UDP já estão prontas, é possível iniciar o desenvolvimento do módulo de comunicação centralizado, ou seja, que utiliza a arquitetura de cliente/servidor no JEagle.

4.4.4. *Headers* presentes no módulo de comunicação descentralizado

Os *headers* que serão utilizados para controlar as ações executadas pelos *peers* são:

- *REQUEST_MEMBERS_LIST* - A mensagem que contém esse *header* não contém um valor em *content*, pois essa mensagem apenas requisita a lista de membros presentes no grupo para o criador do grupo;
- *REQUEST_MEMBERS_LIST_RESPONSE* - Esse *header* é incluído nas mensagens que são enviadas como resposta as mensagens que contém o *header* "REQUEST_MEMBERS_LIST", o atributo *content* dessa mensagem contem a lista atualizada dos membros que estão presentes no grupo;
- *REQUEST_PING* - É um *header* identificador das mensagens enviadas pelos *peers* para o criador do grupo. Essa mensagem contém um objeto do tipo *Ping* para contar a quantidade de tempo necessária para a mensagem sair do emissor e chegar até o criador do grupo;
- *REQUEST_PING_RESPONSE* - É o *header* da mensagem de resposta do "REQUEST_PING" que será enviada pelo criador do grupo para os *peers* requisitantes;
- *REQUEST_PING_FIX* - Esse *header* funciona da mesma forma que o "REQUEST_PING" porém ele é utilizado assim que o membro conecta-se ao grupo para fazer uma primeira contagem da latência do grupo, normalmente essa primeira checagem de *ping* retorna um valor alto que não condiz com a real situação da rede, portanto corrige-se o *ping* no momento da conexão;
- *REQUEST_PING_FIX_RESPONSE* - É o *header* da mensagem de resposta do "REQUEST_PING_FIX" que será enviada pelo criador do grupo para os *peers* requisitantes.

Não é necessário existir uma grande quantidade de mensagens que serão trocadas entre os integrantes do grupo para estabelecer uma comunicação básica, pois o *middleware* utilizado para criar a comunicação descentralizada, chamado de JGroups, já contém praticamente todos os métodos implementados.

A partir do momento que já foi estabelecido as mensagens para estabelecer uma comunicação básica entre os *peers* pertencentes ao grupo, já pode-se dar início ao desenvolvimento das classes responsáveis por criar a comunicação P2P do módulo JEagle.

4.5. Desenvolvimento do módulo de comunicação centralizado

O desenvolvimento do módulo de comunicação centralizado utilizando a arquitetura cliente/servidor será descrito nesse tópico. O projeto de cliente/servidor foi dividido em dois pacotes, o primeiro para as classes relacionadas ao protocolo de comunicação TCP e o segundo pacote para as classes que tem relação com o protocolo UDP. Em relação ao protocolo TCP foram desenvolvidas 4 classes para a execução do cliente e mais 5 classes para a execução do servidor, já em relação do protocolo UDP foram desenvolvidas 5 classes para a execução do cliente e também do servidor, sendo que naturalmente foram desenvolvidos mais métodos para o UDP do que para o TCP, pois o TCP já contém vários métodos pré-desenvolvidos. Como por exemplo, pelo fato do protocolo UDP não ser orientado a conexões, foi necessário emular alguns métodos, de forma que o controle de quando um cliente esta conectado ou não ao servidor teve de ser implementado utilizando um esquema chamado de *ping pong*.

O esquema *ping pong* utilizado no protocolo UDP é caracterizado pela interação entre dois nós ou entidades conectadas, sendo que um nó A irá enviar uma mensagem de *ping* para um outro nó B. No momento em que o nó B receber a mensagem, ele irá enviar uma mensagem de *ping* para o nó A, de modo que quando o nó A receber essa mensagem, significa que o nó B é alcançável, portanto, é possível utilizar esse esquema para verificar se um cliente esta conectado ou não a um servidor.

Através dos diagramas que serão apresentados a seguir, será possível desenvolver todo o sistema de comunicação centralizado para os protocolos TCP e UDP, de forma que cada classe exercerá uma função importante para o funcionamento do mesmo.

4.5.1. Classes comuns aos protocolos TCP e UDP

Na Figura 17 é possível ver o diagrama de classe que representa as classes relativas aos dois protocolos TCP e UDP.

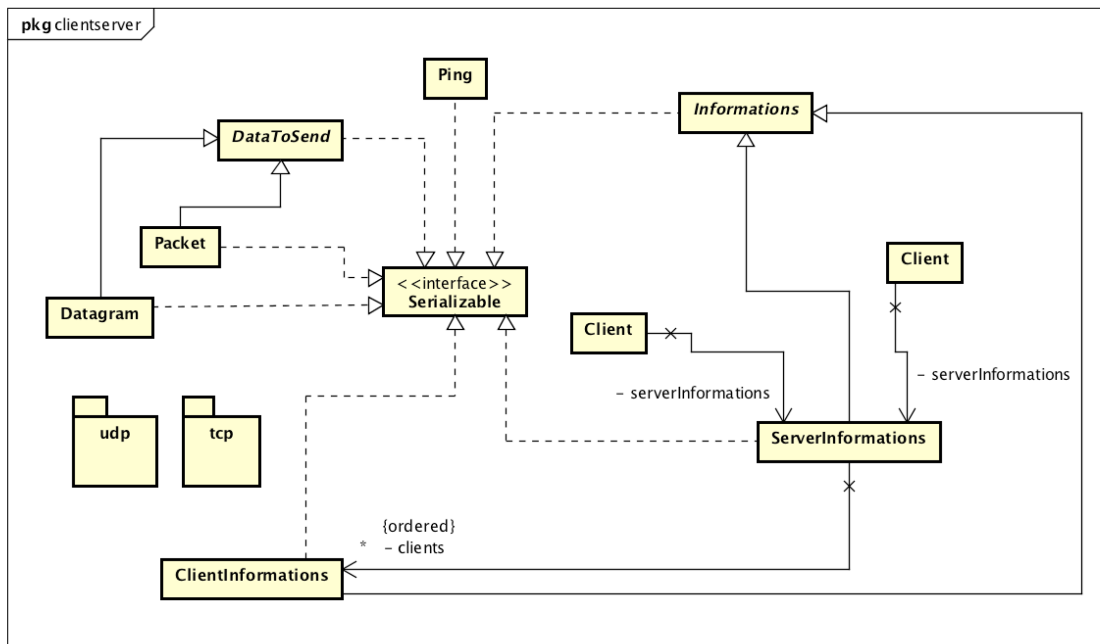

Figura 17: Classes comuns aos protocolos UDP e TCP

A classe ClientInformations

Essa classe será uma classe filha da classe *Informations*, sua funcionalidade é armazenar as informações de um cliente específico, como por exemplo seu identificador único, endereço e porta.

A classe abstrata *DataToSend*

Essa classe é a superclasse das classes que irão posteriormente representar os pacotes e datagramas que serão enviados. Nessa classe serão armazenadas informações cruciais para a troca de dados entre os clientes conectados.

A classe *Datagram*

A classe *Datagram* será uma classe filha da superclasse *DataToSend*. Ela representará um datagrama que será utilizado no protocolo UDP.

A classe abstrata *Informations*

Essa classe será herdada pelas outras classes que armazenaram as informações das entidades presentes no grupo, como por exemplo a classe *ServerInformations* e *ClientInformations*.

A classe *Packet*

Essa classe é a representação de um pacote que será trocado entre os clientes conectados ao servidor utilizando o protocolo de comunicação TCP.

A classe *Ping*

A classe *Ping* representa a contagem do tempo de resposta entre o cliente e o servidor em milissegundos, segundos, minutos, horas e dias. Porém o padrão aqui utilizado para a contagem de tempo será em milissegundos.

A classe *ServerInformations*

Com as classes comuns a ambos os protocolos já prontas, é possível planejar o desenvolvimento das classes utilizadas pelo cliente e servidor dos protocolos TCP e UDP. Nos tópicos a seguir serão descritas essas classes.

4.5.2. Classes utilizadas no cliente do protocolo TCP

No diagrama de classes presente na Figura 18 é possível observar as classes que fazem parte do projeto do cliente que utiliza o protocolo TCP.

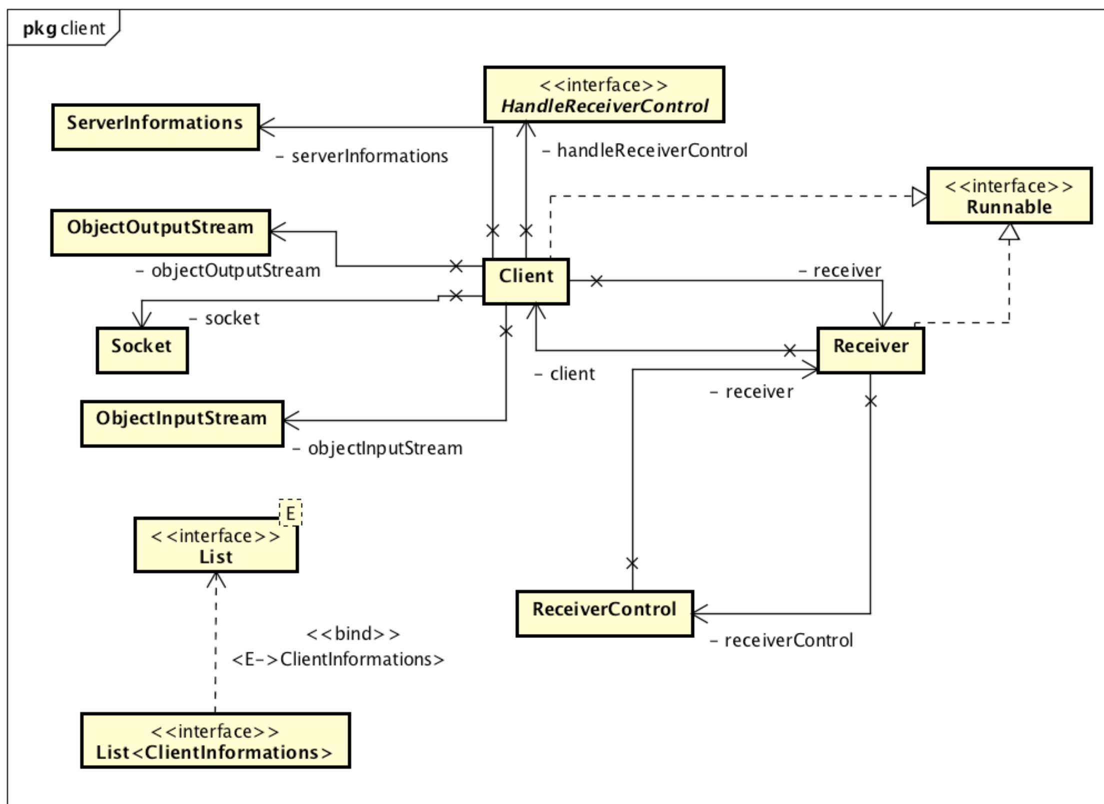

Figura 18: Classes pertencentes ao cliente com o protocolo TCP

A classe *Client*

A classe *Client* será a representação propriamente dita de um determinado cliente que esta conectado ao servidor.

A *interface HandleReceiverControl*

Essa *interface* conterá métodos importantes que posteriormente deverão ser implementados pelo desenvolvedor externo, para que códigos customizados possam ser executados, por exemplo, quando um novo cliente se conecta ao servidor, o cliente local receberá uma mensagem informando que um novo cliente se conectou, nesse momento o desenvolvedor externo poderá executar o código que quiser. A ideia é utilizar essa *interface* para que os desenvolvedores externos possam adicionar seus códigos customizados quando uma ação ocorrer, além disso o desenvolvedor também poderá trocar suas próprias mensagens com os *headers* que desejar.

A classe *Receiver*

A classe *Receiver* receberá os pacotes do servidor e então os repassará para a classe *ReceiverControl*.

A classe *ReceiverControl*

Essa classe é uma das mais importantes no projeto do cliente, pois será aqui que os pacotes serão processados de acordo com o conteúdo do seu *header*, de forma que ações possam ser executadas.

4.5.3. Classes utilizadas no servidor do protocolo TCP

No diagrama de classes presente na Figura 19, é possível visualizar as classes e suas respetivas relações, que estão presentes no projeto do servidor que utiliza o protocolo TCP.

Figura 19: Classes pertencentes ao servidor com o protocolo TCP

A classe *Client*

Essa é a representação de um cliente que esta conectado com o servidor.

A classe *ClientChannel*

Essa classe representa o canal de comunicação existente entre o cliente o servidor.

A *interface HandleReceiverControl*

Essa *interface* conterá métodos importantes que posteriormente deverão ser implementados pelo desenvolvedor externo, para que seus códigos customizados possam ser executados, por exemplo, quando um novo cliente se conecta ao servidor, nesse momento o desenvolvedor externo poderá executar o código que quiser. A ideia é utilizar essa *interface*

para que os desenvolvedores externos possam adicionar seus códigos customizados quando uma ação ocorrer, além disso o desenvolvedor também poderá trocar suas próprias mensagens com os *headers* que desejar.

A classe *ReceiverControl*

Essa classe receberá os pacotes dos clientes e então os processará de acordo com o valor presente em seu *header*.

A classe *Server*

A principal classe do servidor, essa classe será a representação do servidor, contendo assim todas as informações mais importantes, como por exemplo, a lista de clientes conectados e também os métodos responsáveis por enviar mensagens para os clientes.

4.5.4. Classes utilizadas no cliente do protocolo UDP

No diagrama de classe presente na Figura 20 é possível visualizar as classes e suas respetivas relações, que são utilizadas no projeto do cliente com o protocolo UDP. Também é importante ressaltar que apesar do protocolo UDP não ser orientado a conexões, o conceito de conexão foi implementado através do esquema de *ping pong*.

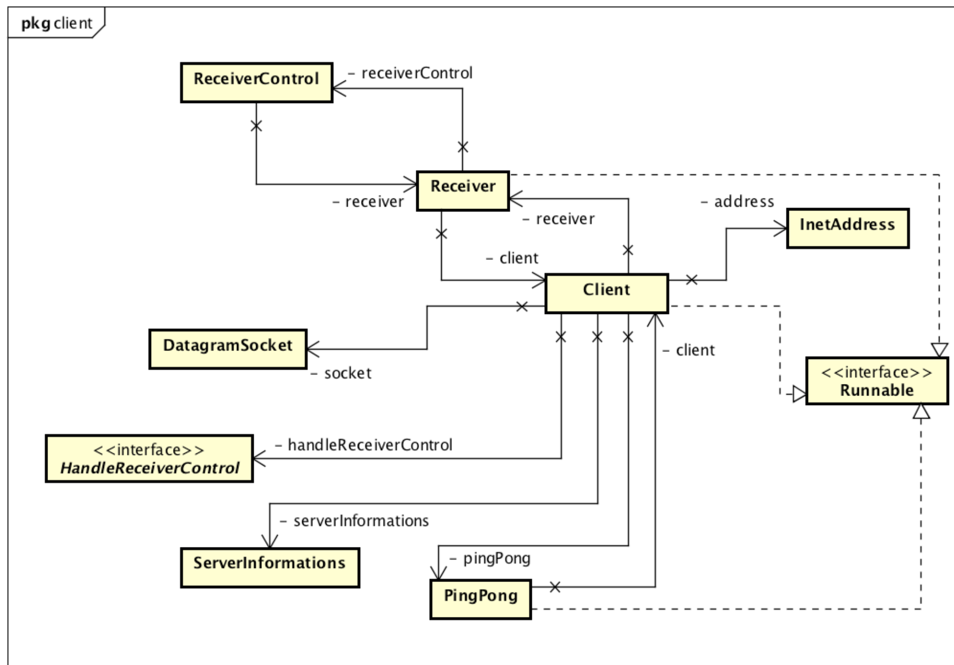

Figura 20: Classes pertencentes ao cliente com o protocolo UDP

A classe *Client*

Essa classe é a representação do cliente que futuramente irá conectar-se ao servidor.

A *interface HandleReceiverControl*

Essa *interface* conterá métodos importantes que posteriormente deverão ser implementados pelo desenvolvedor externo, para que códigos customizados possam ser executados, por exemplo, quando um novo cliente conecta-se ao servidor, o cliente local receberá uma mensagem informando que um novo cliente conectou-se, nesse momento o desenvolvedor externo poderá executar o código que quiser. A ideia é utilizar essa *interface* para que os desenvolvedores externos possam adicionar seus códigos customizados quando uma ação ocorrer, além disso o desenvolvedor também poderá trocar suas próprias mensagens com os *headers* que desejar.

A classe *PingPong*

Essa classe é a representação do esquema *ping pong* do lado do cliente que futuramente será utilizado para simular uma conexão entre o cliente e o servidor, visto que o protocolo

UDP não é orientado a conexões. A ideia para que isso funcione, será esperar por respostas de tempos em tempos do servidor, onde essas respostas conterão um datagrama com *header* igual a “PONG”. No momento em que o cliente receber essa mensagem, ele saberá que o servidor ainda é alcançável. Da mesma forma, o cliente deverá enviar mensagens de tempo em tempo para o servidor, com um *header* igual a “PONG”, avisando dessa forma que o cliente local ainda esta conectado.

A classe *Receiver*

Essa classe irá receber os datagramas do servidor para posteriormente os repassar para a classe *ReceiverControl*, onde eles serão posteriormente processados de acordo com o valor presente no seu atributo *header*.

A classe *ReceiverControl*

Uma das classes mais importantes do projeto, pois quando os datagramas forem recebidos pela classe *Receiver*, eles serão repassados para a classe *ReceiverControl*, onde então em seguida serão processados de acordo com o valor do seu atributo *header*.

4.5.5. Classes utilizadas no servidor do protocolo UDP

No diagrama de classes presente na Figura 21 é possível observar as classes e seus respetivos relacionamentos que fazem parte do projeto do servidor que utiliza o protocolo UDP.

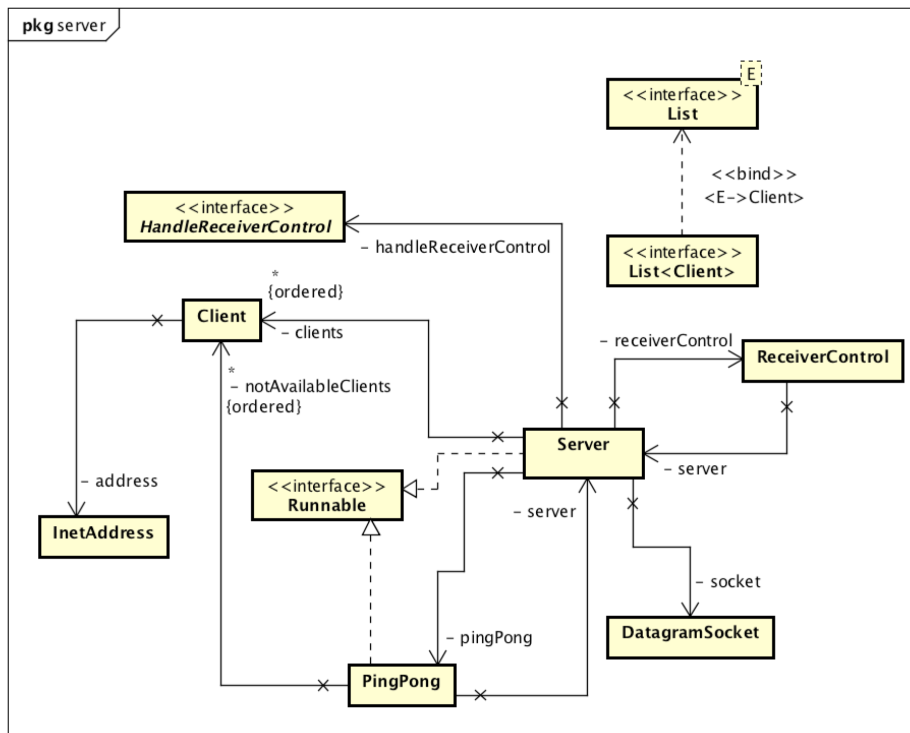

Figura 21: Classes pertencentes ao servidor com o protocolo UDP

A classe *Client*

Essa classe será a representação de um cliente que esta conectado com o servidor.

A *interface HandleReceiverControl*

Essa *interface* conterá métodos importantes que posteriormente deverão ser implementados pelo desenvolvedor externo, para que códigos customizados possam ser executados, por exemplo, quando um novo cliente se conecta ao servidor, nesse momento o desenvolvedor externo poderá executar o código que quiser. A ideia é utilizar essa *interface* para que os desenvolvedores externos possam adicionar seus códigos customizados quando uma ação ocorrer, além disso o desenvolvedor também poderá trocar suas próprias mensagens com os *headers* que desejar.

A classe *PingPong*

Essa classe é a representação do esquema *ping pong* do lado do servidor que futuramente será utilizado para simular uma conexão entre o cliente e o servidor, visto que o protocolo UDP não é orientado a conexões. A ideia para que isso funcione, será esperar por respostas de tempos em tempos do cliente, onde essas respostas conterão um datagrama com *header* igual a "PONG". No momento em que o servidor receber essa mensagem, ele saberá que o cliente ainda esta conectado. Da mesma forma, o servidor deverá enviar mensagens de tempo em tempo para os clientes conectados, com um *header* igual a "PONG", avisando dessa forma que o servidor local ainda esta sendo executado.

A classe *ReceiverControl*

Essa classe será a responsável por receber os datagramas dos clientes e então posteriormente processa-los de acordo com o valor presente em seu *header*.

A classe *Server*

Essa será a principal classe do servidor, pois será a representação propriamente dita do servidor, contendo dessa forma todos os métodos e atributos mais importantes.

Uma explicação detalhada das classes do módulo de comunicação centralizado pode ser visualizada nos anexos.

4.6. Desenvolvimento do módulo de comunicação descentralizado

Nesse tópico será descrito como o código foi implementado, para que o módulo de comunicação descentralizado utilizando a arquitetura P2P com o *middleware* JGroups funcionasse. No total foram necessárias 8 classes para que a comunicação fosse estabelecida, essas classes, uma por uma, serão descritas a seguir, porém ainda é necessário conhecer alguns conceitos, que por sua vez também são utilizados pelo JGroups, para que

seja possível continuar. Portanto, antes de continuar é necessário compreender o que é um *channel* um *cluster* e também uma *view*.

Um *channel* representa o ponto de comunicação entre um grupo, ou seja, quando um membro se conecta ou se desconecta de um grupo ele utiliza seu *channel* para criar esse canal de comunicação. De uma forma simplificada, pode-se pensar em um *channel* como sendo um *socket*. As mensagens enviadas por um *channel*, de acordo com seus parâmetros, são recebidas por todos os integrantes de um grupo ou por um membro específico do grupo. Cada *channel* é único, sendo seu endereço único, ou seja, o endereço do *channel* será seu identificador.

Um *cluster* é o termo utilizado pelo JEagle para abstrair a noção dos grupos, a partir desse meio é possível ter acesso ao grupo, ao criador do grupo e aos seus integrantes que por sua vez serão abstraídos do conceito de *views* utilizado pelo JGroups.

Uma *view* é uma representação de um conjunto dos endereços dos membros presentes em um grupo, de forma que um grupo conterá apenas uma *view*. As *views* contém uma lista de endereços de membros, um identificador único e também o endereço do membro criador do grupo. A lista de membros presente na *view* sempre estará ordenada, de forma que o primeiro integrante da lista é o criador do respetivo grupo e o último membro é o último a ter se conectado. Dessa forma, sempre que um membro do grupo for desconectado ou quando um novo membro se juntar ao grupo, todos os membros obterão uma lista atualizada e sempre ordenada através da *view*. Através dessa lista de membros, o JEagle cria uma lista própria de membros do tipo lista para *Member*.

Agora que esses conceitos já foram compreendidos pode-se criar um diagrama de classes das classes que irão fazer parte do módulo de comunicação P2P do JEagle. Na Figura 22 pode-se ver esse diagrama, e a partir dele o módulo irá desenvolver-se.

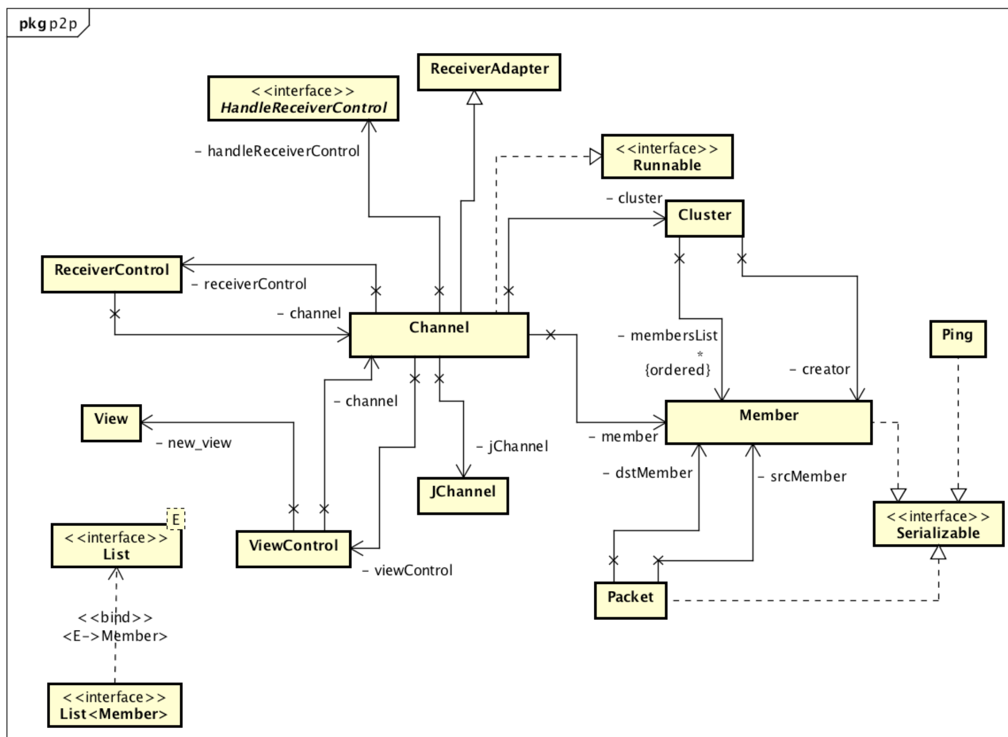

Figura 22: Diagrama de classes do módulo de comunicação descentralizado do JEagle

Cada classe nesse modelo terá uma função específica no funcionamento do projeto, seja ela para controlar o canal de comunicação ou então até mesmo para contar a quantidade de tempo necessária que um pacote demora para sair de A, chegar até B, e então voltar novamente para A. A seguir será detalhado o funcionamento de cada classe.

A classe *Channel*

Será a representação de um conceito herdado do *middleware* JGroups, ou seja, a representação de um canal de comunicação do membro local com os outros membros do grupo, onde um canal é a representação final da comunicação com um grupo, assim como um *socket*.

A classe *Cluster*

Representará o conceito de grupos herdado do *middleware* JGroups, essa classe conterá as informações relativas ao grupo que um membro específico esta conectado.

A classe *Member*

Essa classe é a representação de um membro local, armazenando assim as suas informações, como por exemplo seu endereço, que será o identificador único do membro presente no grupo.

A classe *Packet*

É uma das principais classes do módulo de comunicação descentralizado do JEagle, essa será a abstração de mensagem responsável por armazenar os dados que serão trocados entre os integrantes do grupo.

A classe *Ping*

A classe *Ping* representa a contagem do tempo de resposta entre o cliente e o servidor em milissegundos, segundos, minutos, horas e dias. Porém o padrão aqui utilizado para a contagem de tempo será em milissegundos.

A classe *ViewControl*

Essa é a classe de controle das *views* que foram herdadas do *middleware* JGroups, sendo que uma *view* é a representação local de um membro do grupo, sendo que apenas uma *view* será inicializada em um canal, ou seja, a representação do membro local é única. As *views* também podem o endereço do seu criador, seu endereço local e também uma lista do endereço dos membros conectados ao grupo.

A classe *ReceiverControl*

Essa classe receberá as mensagens contidas nos objetos do tipo *Packet* e então as processará de acordo com o valor do seu *header*.

A *interface HandleReceiverControl*

Através dessa *interface*, que posteriormente deve ser implementada em uma classe qualquer pelo desenvolvedor externo, contém métodos que serão automaticamente executados, caso

implementados, quando uma ação envolvendo os membros do grupo for executada, como por exemplo quando um membro conecta-se ou desconecta-se do grupo. A ideia é utilizar essa *interface* para que os desenvolvedores externos possam adicionar seus códigos customizados quando uma ação ocorrer, além disso o desenvolvedor também poderá trocar suas próprias mensagens com os *headers* que desejar.

Uma explicação detalhada das classes do módulo de comunicação descentralizado pode ser visualizada nos anexos.

4.7. Injetando a *interface HandleReceiverControl* no código do JEagle

Nesse ponto do desenvolvimento do JEagle os módulos de comunicação estão prontos a ser usados. O desenvolvedor recorre à *interface HandleReceiverControl* para possibilitar a sua utilização.

Como já foi abordado nos tópicos anteriores é necessário encontrar uma forma do desenvolvedor adicionar seu código dentro do código do JEagle, como por exemplo, criar os seus próprios *headers* ou executar uma ação customizada quando um membro ou cliente se conecta ou se desconecta do grupo ou servidor. Para que isso seja possível, o que deve ser feito é inicializar um dos atributos presente tanto no módulo de conexão centralizado quanto no módulo de conexão descentralizado chamado de *handleReceiverControl* com a referência para alguma classe qualquer que implementa a *interface HandleReceiverControl*.

Para que essa *interface* seja utilizada pelos clientes, membros ou servidores, é necessário que uma referência para um objeto do tipo da classe que implementa essa *interface* seja adicionada, ou seja, caso o atributo *handleReceiverControl* presente em ambos os módulos de comunicação, seja ele centralizado ou descentralizado, venha a conter o valor *NULL*, nenhum código customizado dos desenvolvedores externos será executado, porém, caso exista lá uma referência armazenada, então esses métodos serão automaticamente executados quando uma determinada ação ocorrer, como por exemplo, quando uma

conexão ou desconexão de um cliente ocorrer, o método *actionConnectionSucess(...)* será executado.

4.8. Executando o JEagle

Após o termino do desenvolvimento de ambos os módulos de comunicação, ou seja, o módulo de comunicação centralizado e descentralizado, é necessário executar o JEagle em conjunto com algum jogo desenvolvido utilizando o motor de jogos jMonkeyEngine. Para esse fim, foi escolhido o jogo Monkey Blaster desenvolvido originalmente para ser executado em modo *offline*, porém serão executadas adaptações nesse jogo para o tornar *multiplayer* com a devida utilização do JEagle.

4.8.1. O jogo Monkey Blaster

Monkey Blaster foi desenvolvido por Daniel Gallenberger e é uma adaptação do jogo Geometry Wars que foi criado pela Bizarre Creations. Assim como no jogo original, o objetivo dessa adaptação é sobreviver o maior tempo possível e também acumular a maior quantidade de pontos possíveis destruindo todos os outros jogadores que estiverem presentes no mundo do jogo. O jogo é executado em um mundo retangular onde o jogador controla uma nave que pode movimentar-se e atirar em qualquer direção. Conforme o jogador progride no jogo, ele acumula mais vidas e também o multiplicador e sua quantidade de pontos aumenta. De forma similar, a quantidade de inimigos que são gerados no mundo do jogo aumenta de acordo com a progressão do jogador. Quando um inimigo colide com a sua nave, a nave explode e o jogador perde uma vida, e o multiplicador de pontos é retornado ao valor original que é 1. O jogo encerra assim que o jogador não conter mais vidas (Figura 23).

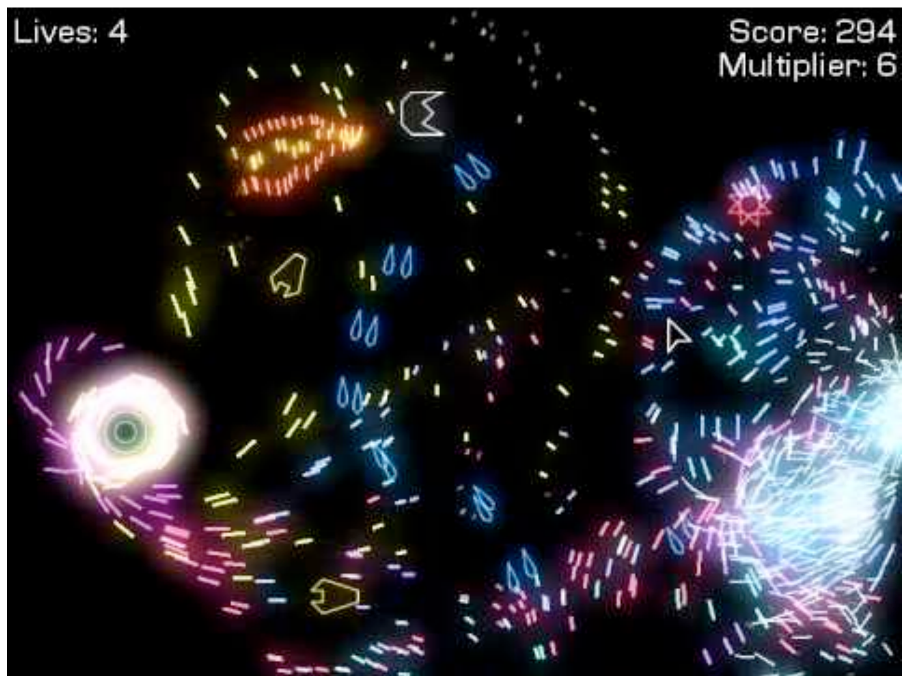

Figura 23: Jogo Monkey Blaster.

Na Figura 24 pode-se ver como é o jogo original e como a adaptação descrita acima é similar.

Figura 24: Jogo Geometry Wars

A adaptação do jogo foi desenvolvida para funcionar sem a intervenção de multijogadores, de forma que para ser possível testar o JEagle é necessário executar alterações em seu código fonte para que multijogadores se conectem ao mesmo mundo do jogo.

4.8.2. Adaptação do jogo Monkey Blaster para multijogador

Antes de começar a perceber como fazer a adaptação do jogo Monkey Blaster para multijogador é necessário compreender como as ações do jogo são executadas pelo jogador local. Assim como nos outros jogos *offline*, as ações básicas do jogo, como por exemplo movimentação e ataque do personagem dão-se início através de um dispositivo de entrada, como por exemplo teclado, mouse ou *joystick*. Logo as ações do personagem estão diretamente ligadas as ações de entrada executadas pelo jogador (Figura 25).

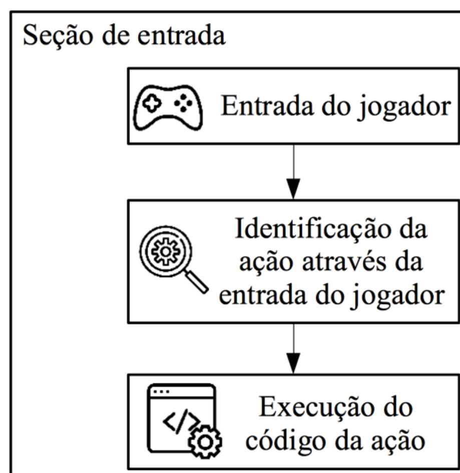

Figura 25: Seção de entrada

Ou seja, baseando-se nessa imagem pode-se rapidamente perceber que os métodos de ação do personagem são executados diretamente na seção de entrada do motor de jogo. A um primeiro momento parece não ser possível executar ações em jogadores externos, pois eles não têm acesso a seção de entrada do mundo de jogo local, apenas o jogador local tem acesso a essa seção. Para contornar esse problema deve-se utilizar a modularização de código para o dividir em partes distintas, identificar onde os códigos das ações são executados e reescrever os códigos em métodos e seções diferentes.

O primeiro passo da alteração é não mais utilizar apenas um objeto do tipo jogador, mas sim uma lista para jogadores, ou seja, uma lista que armazena vários objetos do tipo jogador, e cada objeto do tipo jogador que estiver presente nessa lista representará um membro do grupo que esta conectado, dessa forma é possível executar ações em objetos do tipo jogador diferentes, por exemplo, movimentar o jogador 2 e não o jogador 1.

Assim que existe uma lista para jogadores, é necessário criar um método para a sua instanciação, sendo que o método deve conter informações especificas de cada jogador, como por exemplo sua cor e um identificador universal que será utilizado para reconhecimento entre os jogadores, além disso, esse método deve ser capaz de retornar um objeto do tipo jogador, logo é possível instanciar quantos jogadores quiser no mundo do jogo, simulando assim a sua conexão. Além desse método de instanciação dos jogadores, também é necessário ter um método que destrói o objeto de um jogador, ou seja, um método capaz de simular a sua desconexão. Após ter sido executada essas mudanças no jogo é chegado o momento de encontrar os trechos no código referentes as ações dos jogadores.

O segundo passo é encontrar os códigos das ações, onde o primeiro código encontrado é sobre a movimentação do jogador, que esta presente na seção de entrada e funciona da seguinte maneira, quando o jogador local pressiona uma das teclas *W, A, S, D* o objeto do personagem local é transladado em *n* unidades para cima, esquerda, baixo ou direita respetivamente no momento em que o jogador esta pressionando as teclas, assim que ele solta as teclas a ação de movimentação encerra-se. Aqui a mudança a ser feita é remover o código que translada o personagem e reescrevê-lo em um método específico para o movimento contendo em seu parâmetro uma variável que referência um objeto do tipo jogador, de forma que através desse parâmetro é possível movimentar jogadores diferentes. O segundo código encontrado é sobre a ação de ataque do personagem que é executada sempre que o jogador pressiona o botão esquerdo do *mouse*. Deve-se executar o mesmo processo exemplificado acima, ou seja, criar um método para a ação de ataque do personagem com os parâmetros sendo uma referência para o jogador que executou o ataque e também as posições de onde o ataque deve ser instanciado, assim como a sua direção. Para simular o nascimento de um personagem especifico basta utilizar o método de

instanciação dos personagens, porém o método para simular a morte de um personagem não pode utilizar o mesmo método de destruir um personagem, pois quando um personagem morre, um efeito de explosão deve ser executado, algo que o método para destruir o objeto do personagem não faz, portanto o que deve ser feito, é reutilizar o método para destruir o personagem e então executar o efeito de explosão. Resumidamente, o que se deve alterar no código do Monkey Blaster é a remoção dos códigos de ação da seção de entrada e sua devida modularização em métodos separados (Figura 26).

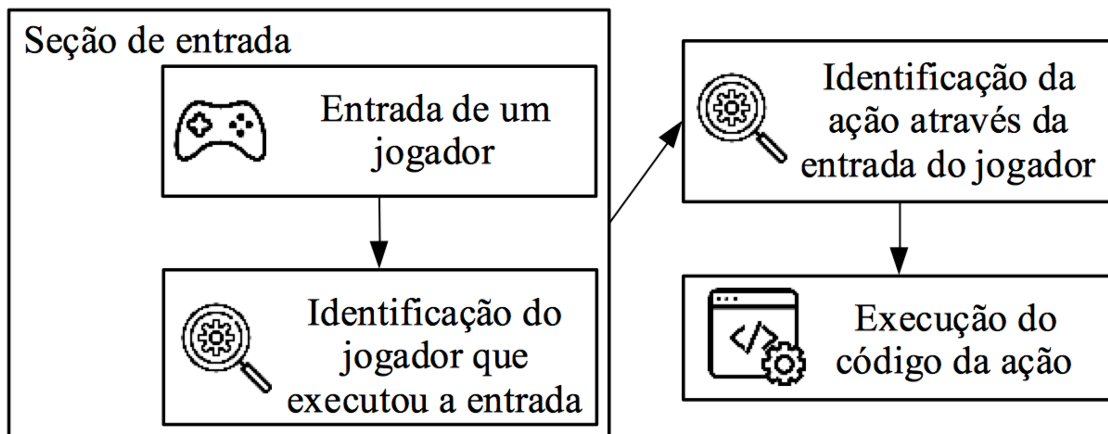

Figura 26: Modularização da seção de entrada

O que acontece no processo é basicamente a modularização dos métodos, onde agora a seção de entrada será responsável apenas por receber a entrada do jogador e identificar qual jogador executou a devida entrada, em seguida é necessário executar outro método para descobrir qual será a ação que o devido jogador deverá executar baseando-se em sua entrada. Após descoberto qual é a ação a ser executada basta apenas executa-la e o jogador em questão irá alterar o mundo do jogo.

Seguindo esses passos o jogo Monkey Blaster passa a ser executado em modo multijogador, porém apenas na mesma máquina. Para fazer com que máquinas distintas compartilhem o mesmo mundo do jogo é necessário utilizar um módulo de comunicação, que nesse caso será o JEagle.

4.8.3. Junção do jogo Monkey Blaster com o JEagle

Para que o jogo Monkey Blaster funcione em rede e compartilhe o mesmo mundo do jogo com vários jogadores é necessário que um módulo de comunicação seja usado.

Baseando-se nos tópicos anteriores, depois que a seção de entrada é segmentada e os códigos de ação dos jogadores não são mais executados na seção de entrada e sim em métodos e seções diferentes, o que é necessário fazer agora é alterar a execução das ações dos personagens para serem executadas quando receberem uma mensagem através da rede informando que uma ação foi executada, e não mais através das entradas dos usuários (Figura 27).

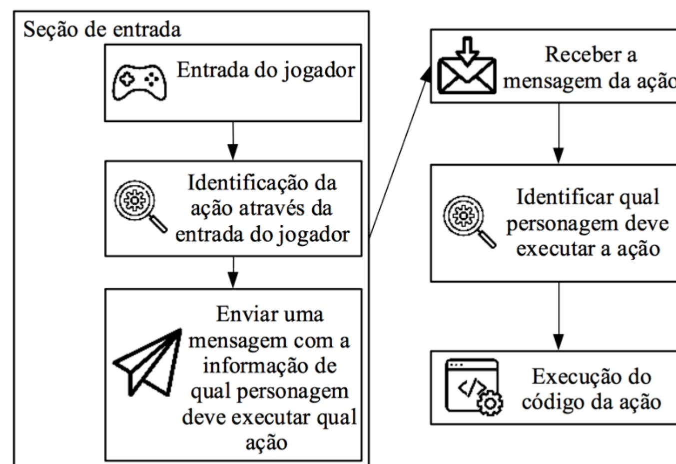

Figura 27: Execução das ações através de mensagens

Resumidamente, o que deve ser feito é iniciar a execução das ações dos jogadores através de mensagens e não mais através das entradas do usuário.

Todo esse processo acontece devido à troca de mensagens, ou seja, quando um jogador pressiona uma tecla do teclado ou inicia qualquer outra forma de entrada, essa entrada é processada, então a ação que o jogador deverá executar é definida e, posteriormente, é criada uma mensagem com a informação dessa ação e também qual é o jogador que deverá executa-la, sendo que o jogador é definido através de seu identificador único. Após o objeto da mensagem ser criado, ele é enviado por *broadcast* para todos os integrantes ou então para um integrante específico. A partir desse momento os integrantes irão receber a

mensagem e posteriormente o atributo *header* dessa mensagem será processado até que uma *String* equivalente seja encontrada e a ação que foi enviada seja executada, atualizando os atributos do jogador em questão.

Seguindo esses passos o resultado obtido é um jogo multijogador com as possibilidades de ser executado utilizando a arquitetura cliente/servidor com o protocolo de comunicação TCP, UDP e também a arquitetura P2P. Na Figura 23 é possível visualizar a adaptação do jogo finalizada utilizando a arquitetura de conexão P2P com dois membros conectados ao grupo.

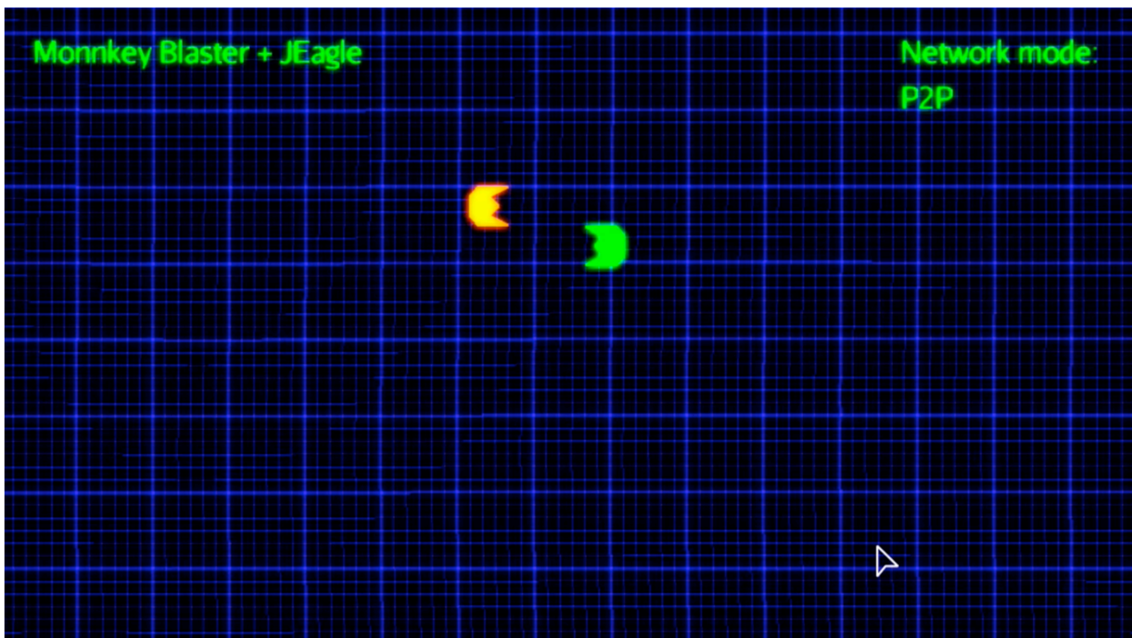

Figura 28: Monkey Blaster + JEagle

Depois que se percebe como adaptar um jogo de apenas um jogador para multijogadores é necessário estabelecer uma forma de comunicação que seja comum a todos os integrantes do grupo ou aos clientes conectados ao servidor. A forma de comunicação que precisa ser desenvolvida é a criação de um protocolo que mapeie as principais ações executadas pelos jogadores. As principais ações que deverão ser mapeadas são as ações de criar personagem, renascer, morrer, atirar e movimentar os personagens que estão presentes no mundo do jogo. Todas essas ações serão implementadas de forma similar independente do módulo de comunicação utilizado, sendo que quando um jogador executar uma ação, essa ação deve

ser replicada para todos os outros integrantes do grupo ou para os clientes conectados ao servidor, para que os mesmos possam simular as ações como se elas tivessem sendo executadas localmente.

4.8.4. Mapeamento das ações

Para ambos os módulos de conexão, seja o módulo de conexão centralizado ou descentralizado, as ações que serão descritas a baixo funcionarão da mesma forma. As principais ações que deverão ser mapeadas pelos desenvolvedores externos, criando assim mensagens com *headers* customizados em ambos os módulos de conexão são as ações de criar personagem, renascer, morrer, atirar e movimentar, essas ações serão melhor descritas nos subtópicos a seguir.

A Ação de criar personagem

Será executada quando um novo jogador se juntar ao grupo ou então se conectar ao servidor: *actionClientJoin(ClientInformations clientInformations)* para o módulo de conexão centralizado e o método *memberJoin(Member member)* para o módulo de conexão descentralizado. Dessa forma os jogadores locais saberão quando um novo jogador se juntar ao mundo do jogo, sendo assim, o que deve ser feito é requisitar as informações do jogador que se conectou para o adicionar à lista de jogadores conectados e também para instanciar seu personagem no mundo do jogo de cada jogador conectado. Dessa forma todos os jogadores terão uma cópia do personagem.

O jogador local irá requisitar as informações do novo jogador conectado enviando-lhe uma mensagem com um *header* de requisição, que pode ser por exemplo o seguinte "REQUEST_PLAYER_INFORMATIONS". Quando o novo jogador receber essa requisição ele irá enviar uma mensagem de resposta para o jogador que requisitou essas informações, com as informações do novo jogador em seu atributo *content* e o texto do seu *header* poderá ser o seguinte "REQUEST_PLAYER_INFORMATIONS_RESPONSE". Quando o jogador que requisitou a mensagem receber uma mensagem de resposta, ele deverá executar o método *createPlayer(...)* com as informações do novo jogador em seu parâmetro.

Seguindo os passos acima será possível instanciar um novo personagem em todos os jogadores, fazendo com que a partir daqui cada ação feita em cada personagem possa ser simulada, como se ela estive acontecendo localmente nas máquinas de cada integrante do jogo.

A Ação de renascer

Será a re-execução da ação de criar um personagem, onde e quando um personagem deve renascer. O que deve ser feito é recriar o personagem local e enviar uma mensagem em *broadcast* para todos os outros jogadores presentes no grupo ou conectados ao servidor de que o jogador em questão deve ser recriado. A mensagem enviada deve conter um *header* específico, como por exemplo "RESPAWN_PLAYER" e, no atributo *content* da mensagem, deve estar armazenado um objeto que contenha as informações do personagem que deve ser recriado. Quando os outros jogadores receberem essa mensagem, eles saberão que um personagem deve ser recriado pelo *header* da mensagem, e então recriarão o personagem através de suas informações que estiverem armazenados no atributo *content*. Dessa forma o personagem será recriado em todos os jogadores que estiverem conectados ao servidor ou então que fizerem parte do grupo.

A Ação de morrer

É uma das mais simples ações que devem ser simuladas nas máquinas dos jogadores, assim como em um jogo *offline*, deve-se seguir a mesma ideia de quando um projétil colidir com o personagem do jogador local, ele deve ser destruído. Para que isso ocorra entre vários jogadores devemos abstrair essa ideia onde os projeteis pertencerão não mais apenas ao jogador local, mas cada projétil terá um dono, que será um dos jogadores que estão presentes no grupo ou conectados ao servidor.

Seguindo a ideia descrita acima, o que deve ser feito agora é verificar se existe uma colisão do personagem do jogador local com algum projétil pertencente ao mundo do jogo, essa verificação será executada no método *handleCollisions()*, em que quando a colisão existir, o jogador local enviará uma mensagem para os outros jogadores presentes no grupo ou aos jogadores conectados ao servidor com um *header* informando que o personagem deve ser

destruído como, por exemplo, o *header* “KILL_PLAYER”. O atributo *content* dessa mensagem deve conter as informações do personagem que deve morrer. Quando os outros jogadores receberem essa mensagem, eles devem primeiramente aceder o objeto que está armazenado no atributo *content* para descobrir que personagem deve morrer e então fazer com que o personagem seja destruído.

A ação de atirar

Antes de compreender como a ação de ataque, ou seja, a ação de atirar funciona, é importante conhecer o método *onAnalog(String name, float value, float tpf)* utilizado pela jMonkeyEngine. Esse método será automaticamente chamado pelo motor de jogo quando uma entrada do usuário de forma análoga for recebida, ou seja, quando o jogador executar uma entrada de forma contínua. O parâmetro *name* assim como no método *onAction(...)* será utilizado para reconhecer uma entrada do usuário registrada, ou seja, como por exemplo quando o jogador pressiona o botão direito do *mouse*, essa entrada será registrada como “botão_direito_mouse” pelo desenvolvedor e, então o motor de jogo reconhecerá essa entrada quando o jogador pressionar o botão direito do mouse. O parâmetro *value* representa o valor do eixo, que varia de 0 até 1. O parâmetro *tpf* representa a quantidade de *frames* por segundo no momento em que a entrada ocorreu, pode-se utilizar esse parâmetro para executar ações que envolvem movimento de forma fluída.

Agora que o método *onAnalog(...)* já foi percebido, o que deve ser feito é sempre que o jogador pressionar o botão direito do mouse enviar uma mensagem para os membros do grupo ou ao clientes conectados ao servidor informando que o jogador atual executou um ataque. Para que esse processo funcione deve-se dentro do método *onAnalog(...)* executar a condição, se o jogador pressionar o botão direito do mouse, então deve-se enviar uma mensagem para os outros jogadores com um *header* específico informando o tipo de ação que deve ser simulado em cada máquina. Por exemplo, um *header* que pode ser utilizado é o “CREATE_BULLET”, de forma, que assim que os outros jogadores recebam uma mensagem que contenha esse *header*, eles irão iniciar a simulação local do ataque do personagem especifico que esta contido no atributo *content* da mensagem.

Ao seguir esses passos, todos os jogadores presentes no grupo ou conectados ao servidor serão capazes de visualizar a ação de ataque de forma simultânea.

A ação de movimentar

O primeiro passo é compreender como o método *onAction(String name, boolean isPressed, float tpf)* presente na classe *MonkeyBlasterMain* funciona. Esse método é chamado automaticamente pelo motor de jogo jMonkeyEngine quando uma entrada do usuário registrada é recebida, como por exemplo quando uma tecla do teclado é pressionada. O parâmetro *name* é o nome da entrada do usuário que foi registrada, como por exemplo "tecla_a" quando a tecla *a* é pressionada. O parâmetro booleano *isPressed* contém um valor verdadeiro caso uma tecla tenha sido pressionada, e falso quando a tecla que foi pressionada é liberada. O parâmetro *tpf* é o valor do *time per frame* atual, ou seja, a quantidade de *frames* que são gerados no momento.

Através do método *onAction(...)* será possível receber as entradas do usuário, como por exemplo *W, A, S, D* para quando ele quiser iniciar a movimentação do seu personagem, reconhecer através da tecla pressionada para qual direção o jogador deseja movimentar seu personagem e então distribuir essa informação entre os membros conectados ao grupo, ou aos clientes conectados ao servidor. Através do mesmo método também é possível avisar aos integrantes do grupo ou aos clientes conectados ao servidor quando uma tecla anteriormente pressionada foi liberada, para que desse modo, o movimento do personagem seja então encerrado.

Nesse ponto já é possível perceber que pacotes ou datagramas serão enviados aos membros do grupo ou aos clientes conectados ao servidor para que a simulação da movimentação seja simulada em suas máquinas. Porém é importante perceber que não será viável enviar essas mensagens a todo momento que o jogador estiver pressionando uma tecla, pois dessa maneira, serão enviados uma grande quantidade de mensagens pela rede, fazendo com que um grande tráfego seja gerado. Para contornar esse problema é necessário que as mensagens de movimentação sejam enviadas apenas quando o movimento é iniciado e quando ele é parado, ou seja, apenas duas mensagens serão enviadas pela rede e não milhares. Como apenas duas mensagens deverão ser enviadas, o parâmetro *isPressed* do

método *onAction(...)* será de grande ajuda, pois assim será possível saber o momento de quando uma determinada tecla foi pressionada e o momento de quando uma tecla foi liberada.

O que deve ser feito agora é dentro do método *onAction(...)* criar algumas condições, onde nessas condições serão verificadas se uma determinada tecla, relacionada a movimentação, foi pressionada ou liberada, e nessas condições enviar as mensagens aos outros membros do grupo ou clientes conectados ao servidor, informando a ação que deverá ser executada.

Por exemplo, quando o jogador pressiona uma tecla para movimentar o personagem para cima, automaticamente o método *onAction(...)* será chamado, e então a seguinte condição deverá ser satisfeita, ou seja, se o parâmetro *name* for igual a tecla para cima e o parâmetro *isPressed* for igual a verdadeiro, então significa que a tecla para cima foi pressionada. Com isso devemos enviar uma mensagem para todos os membros integrantes do grupo ou para todos os clientes conectados ao servidor informando que a tecla para cima foi pressionada, e a simulação de movimento do personagem em questão deve ser iniciada.

Essa mensagem deve conter obrigatoriamente um *header* que informe o tipo de ação que deverá ser executada, o *header* aqui utilizado foi "PLAYER_MOVE_UP_START" e o *content* da mensagem deve ser as informações relativas ao personagem em questão. Então quando os outros jogadores receberem essa mensagem, eles saberão que devem iniciar a simulação de movimento relativa ao personagem que encontra-se no atributo *content* da mensagem. Deve seguir-se a mesma lógica para parar de movimentar os personagens, apenas deve-se utilizar outro *header* na mensagem, como por exemplo "PLAYER_MOVE_UP_STOP", e então quando os outros jogadores receberem essa mensagem, eles saberão que é o momento de parar a simulação do movimento do personagem em questão.

Com todas as ações mapeadas, o jogo já esta pronto para ser testado em conjunto com o JEagle, seja ele com o módulo de comunicação centralizado ou com o módulo de comunicação descentralizado. Com o jogo e os módulos de comunicação já finalizados, o que deve ser feito agora é testar o jogo juntamente com o JEagle para ter-se uma noção de como os módulos de comunicação serão executados e quanto tempo eles demorarão para

trocar informações entre os integrantes do grupo ou entre os clientes conectados ao servidor.

4.9. Conclusão

Todos os passos aqui descritos foram desenvolvidos através de um estudo sobre como possibilitar a comunicação de jogos entre diferentes máquinas, de tal forma que o mundo do jogo de todos os integrantes seja o mesmo e também compartilhado, independente do módulo de comunicação utilizado. Com isso, esse capítulo foi organizado como uma forma introdutória para o desenvolvimento de jogos *multiplayer* utilizando a linguagem de programação Java, possuindo também uma linguagem mais simplificada para que os iniciantes na área de sistemas distribuídos possam compreender os passos básicos para desenvolver seu próprio módulo de comunicação. A contribuição aqui apresentada, foi sanar um dos problemas muito comum que sempre esta presente no desenvolvimento de jogos, que é o problema do desenvolvimento do módulo de comunicação.

Capítulo 5 Análise e discussão de resultados

Esse capítulo será destinado para a explicação de como os testes com o JEagle foram executados e posteriormente serão apresentados os resultados e suas devidas discussões.

5.1. Introdução

Para a execução dos testes envolvendo o jogo MonkeyBlaster junto com o JEagle foram utilizados três cenários de teste para cada módulo de comunicação (centralizado e distribuído). No primeiro, os cenários de teste foram estruturados como:

- Primeiro cenário: 3 máquinas, uma dedicada ao servidor e 2 para a execução dos clientes.
- Segundo cenário: 2 máquinas, sendo a primeira a executar o servidor e também um cliente, a segunda máquina é dedicada para um cliente.
- Terceiro cenário contém 3 máquinas, uma a executar o servidor e 2 a executar os clientes. O servidor encontra-se localizado numa rede distinta, separada por um roteador.

Relativamente aos testes da arquitetura distribuída, também foram definidos 3 cenários:

- Primeiro cenário inclui a utilização de 3 máquinas, cada uma a executar um membro pertencente ao grupo.

- Segundo cenário usa, também, 3 máquinas, cada uma das máquinas estará executando um membro presente no grupo, sendo que os membros estarão presentes em redes diferentes e necessitaram de um túnel para possibilitar sua comunicação.
- O terceiro cenário de testes inclui 4 máquinas, sendo 3 máquinas pertencentes a mesma rede e executando 1 membro em cada máquina, a quarta máquina pertence a uma rede externa e é dedicada exclusivamente para a execução do *GrossipRouter*, para ultrapassar limitações de NAT.

5.2. Dados técnicos dos testes

Nos testes executados com o módulo de comunicação centralizado foram utilizados objetos do tipo *Packet* para o protocolo TCP e *Datagram* para o protocolo UDP, já no módulo de comunicação descentralizado foi utilizado o objeto do tipo *Packet* do pacote pertencente a arquitetura P2P, sendo que os testes se basearam no tempo de reposta que uma mensagem demora para sair do emissor, chegar ao destino e regressar novamente para o emissor original.

Os objetos que foram trocados entre os integrantes do grupo e o servidor formam devidamente serializados utilizando o método *serialize(...)* disponibilizado pelo *Apache Commons* e, posteriormente, adicionados a um vetor de *bytes* para que fosse então possível contar a quantidade de *bytes* presente em cada objeto que foi enviado pela rede.

Para o módulo de comunicação centralizado, um objeto do tipo *Packet* e *Datagram* vazio, ou seja, sem conter um valor no atributo *header*, *content* e nos identificadores únicos do cliente emissor e do recetor, aqui chamados de *uuids*, contém um total de 188 *bytes*. Quando esses objetos não se encontram vazios, eles contêm um tamanho que varia entre os 360 e os 380 *bytes*.

Para o módulo de comunicação descentralizado, um objeto do tipo *Packet* vazio, ou seja, sem conter objetos armazenados no atributo *header*, *content* e nos membros emissores e recetores contém um total de 154 *bytes*. Quando esse objeto não esta vazio, ele contém um total que varia de 310 a 330 *bytes*.

Portanto, os objetos enviados pela rede de um ponto A até um ponto B que foram aqui testados variam de 310 a 380 *bytes* e foram testados conforme as respetivas descrições dos tópicos seguintes.

5.3. Testes efetuados

De acordo com o definido anteriormente, cada arquitetura foi testada seguindo 3 cenários distintos. O teste ao modelo centralizado iniciou com a configuração de 1 servidor e 2 clientes (Figura 29).

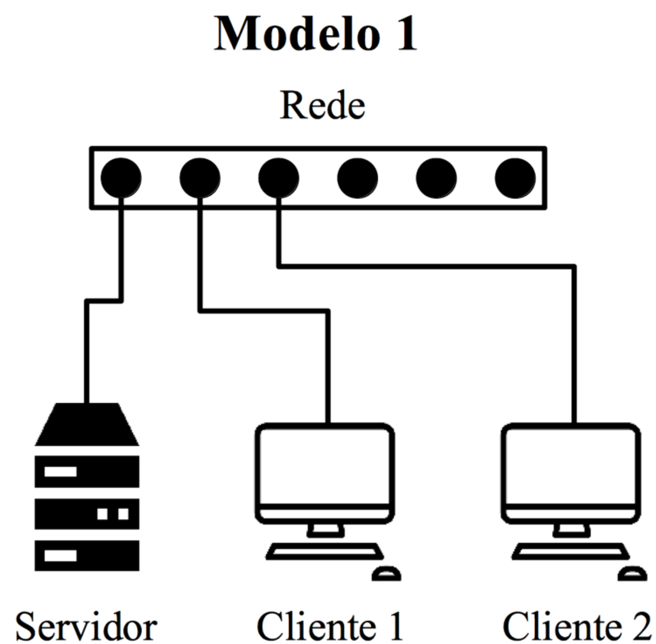

Figura 29: Arquitetura centralizada - configuração 1.

Foram executados 10 testes em cada cliente, sendo registados os tempos de reposta em milissegundos obtidos para cada máquina utilizando o protocolo TCP (Tabela 1).

Tabela 1: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP no modelo de testes 1

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
C1	45ms	46ms	46ms	44ms	47ms	48ms	47ms	46ms	46ms	46ms
C2	46ms	49ms	45ms	46ms	46ms	46ms	48ms	45ms	46ms	45ms

Os resultados da Tabela 2 são relativos aos testes executados com o protocolo UDP, sendo que *C1* significa cliente 1, C2 cliente 2, e *T1, T2, ..., T10* os testes executados, que no total foram 10.

Tabela 2: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP no modelo de testes 1

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
C1	2ms	1ms	2ms	1ms	2ms	1ms	1ms	1ms	1ms	1ms
C2	1ms	1ms	2ms	1ms	1ms	2ms	1ms	1ms	1ms	2ms

De seguida, associou-se o servidor ao cliente 1, repetindo-se os mesmos testes (Figura 30).

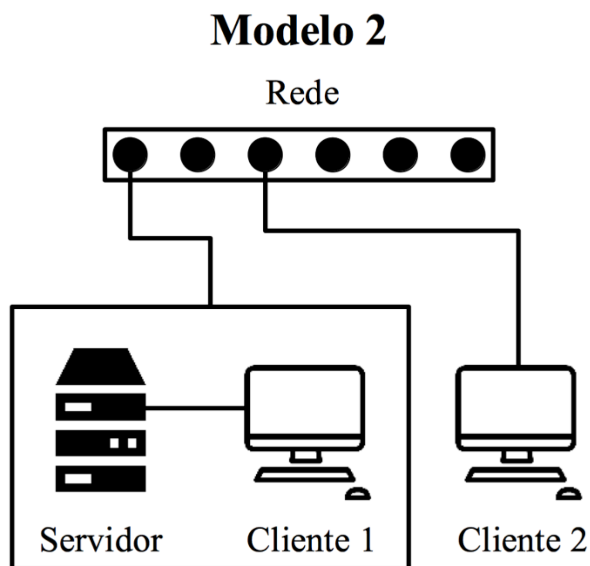

Figura 30: Arquitetura centralizada - configuração 2.

Nessa configuração, para o módulo de comunicação centralizado utilizando ambos os protocolos TCP e UDP, uma máquina foi selecionada para executar o servidor e também um cliente, a outra máquina foi dedicada exclusivamente para a execução do cliente.

Foram executados 10 testes em cada máquina que executava os clientes, e os testes de tempo de reposta em milissegundos obtidos para cada máquina utilizando o protocolo TCP (Tabela 3). De lembrar que o cliente 1 foi executado na mesma máquina que o servidor.

Tabela 3: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP na configuração de testes 2.

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
C1	43ms	42ms	45ms	43ms	42ms	45ms	43ms	43ms	42ms	45ms
C2	46ms	48ms	46ms	47ms	47ms	49ms	46ms	46ms	46ms	49ms

Os resultados da Tabela 4 são relativos aos testes executados com o protocolo UDP, sendo que *C1* significa cliente 1, *C2* cliente 2, e *T1, T2, ..., T10* os testes executados, que no total foram 10, lembrando que o cliente 1 foi executado na mesma máquina que o servidor.

Tabela 4: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP na configuração 2.

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
C1	1ms	1ms	2ms	1ms	1ms	1ms	2ms	1ms	1ms	2ms
C2	2ms	3ms	4ms	2ms	2ms	2ms	2ms	3ms	3ms	2ms

Na Figura 31 é possível visualizar como o modelo de testes 3 foi elaborado:

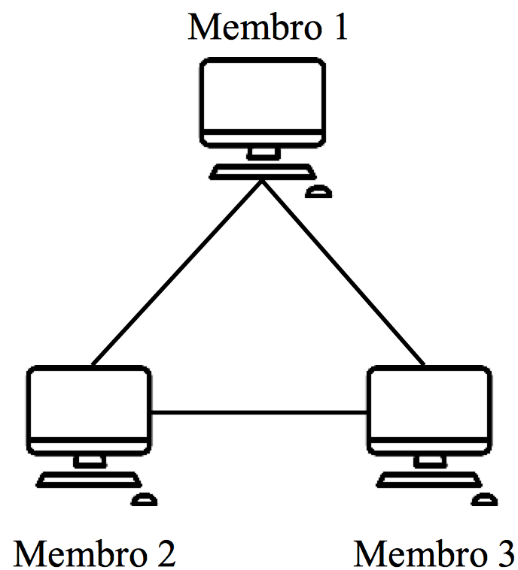

Figura 31: Arquitetura distribuída - configuração 1.

O modelo de testes 3 foi projetado para ser utilizado com a arquitetura distribuída, pois ele representa um modelo P2P, onde a comunicação ocorre de forma direta entre os membros do grupo.

Como o módulo de comunicação descentralizado utiliza o JGroups como *middleware*, o protocolo padrão utilizado para a comunicação entre os membros presentes no grupo é o UDP com certas customizações que providenciam funcionalidades extras, sendo algumas delas semelhantes as que estão presentes em outros protocolos, como por exemplo o protocolo TCP, fazendo dessa forma, com que a velocidade de comunicação entre os membros que estão presentes no grupo seja prejudicada.

Na Tabela 5 serão apresentados os resultados dos testes, sendo que *M1* significa membro 1, *M2* membro 2, *M3* membro 3, e *T1, T2, ..., T10* os testes executados, que no total foram 10, lembrando que o membro 1 era o criador do grupo.

Tabela 5: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups no modelo de testes 3

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
M1	1ms	2ms	2ms	2ms	1ms	1ms	1ms	1ms	1ms	1ms
M2	164ms	183ms	185ms	181ms	183ms	191ms	188ms	172ms	195ms	182ms
M3	190ms	169ms	163ms	198ms	183ms	180ms	184ms	186ms	192ms	196ms

Na Figura 32 é possível visualizar como o modelo de testes 4 foi elaborado:

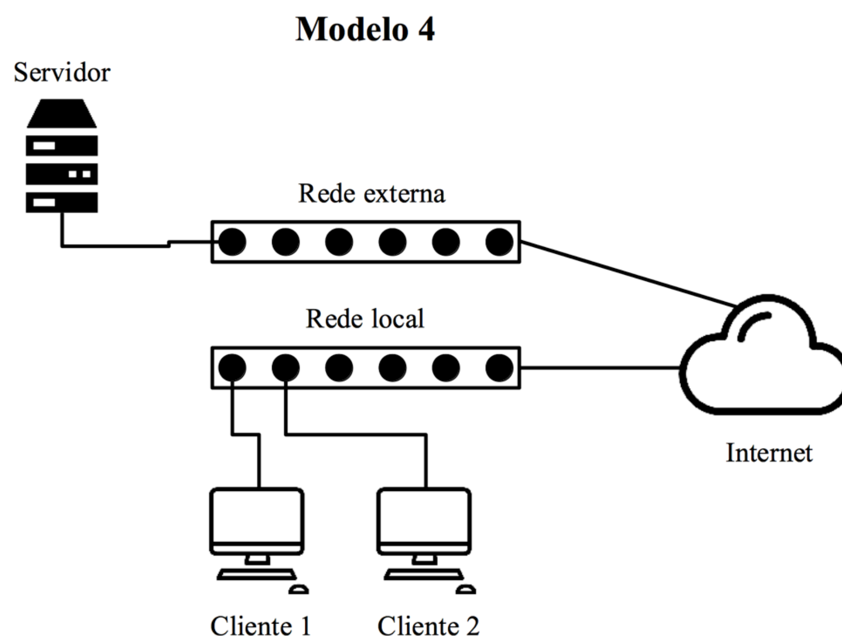

Figura 32: Arquitetura centralizada - configuração 3.

No modelo de testes 4 foi utilizado o módulo de comunicação centralizado com ambos os protocolos TCP e UDP. Nesses testes foram utilizadas uma rede local com duas máquinas executando os clientes e uma rede externa, onde nessa rede externa existia uma máquina dedicada exclusivamente para a execução do servidor.

O resultado da latência das duas máquinas utilizando o protocolo TCP podem ser observados, na Tabela 6, sendo que *C1* significa cliente 1, *C2* cliente 2, e *T1, T2, ..., T10* os testes executados, que no total foram 10.

Tabela 6: Resultados do tempo de resposta em milissegundos utilizando o protocolo TCP no modelo de testes 4

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
C1	56ms	50ms	52ms	52ms	51ms	48ms	50ms	48ms	48ms	48ms
C2	51ms	49ms	51ms	49ms	52ms	50ms	48ms	48ms	53ms	50ms

Os resultados da Tabela 7 são relativos aos testes executados com o protocolo UDP, sendo que *C1* significa cliente 1, *C2* cliente 2, e *T1, T2, ..., T10* os testes executados, que no total foram 10.

Tabela 7: Resultados do tempo de resposta em milissegundos utilizando o protocolo UDP no modelo de testes 4

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
C1	4ms	5ms	3ms	12ms	4ms	4ms	4ms	7ms	5ms	4ms
C2	11ms	7ms	3ms	4ms	4ms	3ms	3ms	4ms	4ms	3ms

Na Figura 33 é possível visualizar como o modelo de testes 5 foi elaborado:

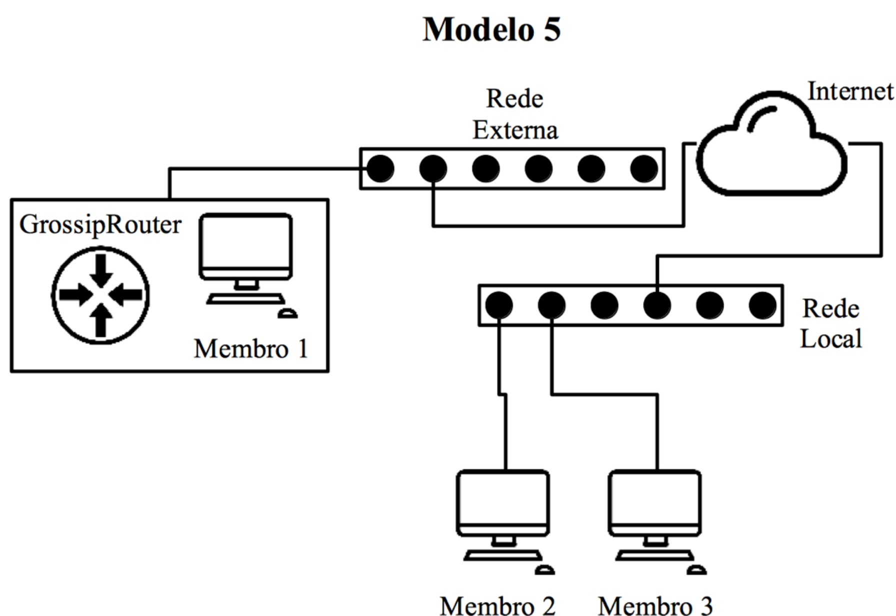

Figura 33: Arquitetura distribuída - configuração 2.

O penúltimo modelo de testes utilizou uma técnica especial do JGroups, a utilização de um túnel de comunicação. Segundo as definições do JGroups um túnel é um protocolo que encapsula outros protocolos, dessa maneira, o desenvolvedor pode enviar e receber dados através de um *firewall* utilizando de uma solução chamada de *GrossipRouter*, que deve estar sendo executado fora do *firewall*. Com a utilização dessa solução os outros integrantes do grupo, que também estiverem sendo protegidos pelo *firewall*, terão a possibilidade de acessar as informações trocadas pelo grupo.

Para que a comunicação funcione, o canal dentro do *firewall* deve utilizar o protocolo TUNNEL, para que posteriormente os dados presentes no protocolo TUNNEL sejam repassados para o protocolo TCP.

Lembrando que o *GrossipRouter* é executado em uma máquina com IP fixo, para que os outros membros do grupo possam conectar-se através desse túnel, ou seja, caso ocorra algum problema com o túnel, os membros ficarão impossibilitados de juntar-se ao grupo.

Para esse modelo de testes foi utilizado o módulo de comunicação descentralizado com o protocolo TUNNEL do JGroups, onde o criador do grupo estava sendo executado na mesma máquina que o *GrossipRouter*.

Foram executados 10 testes nas três máquinas, sendo que os resultados desses testes podem ser visualizados na tabela Tabela 8, onde *M1* significa membro 1, *M2* membro 2, *M3* membro 3, e *T1, T2, ..., T10* os testes executados, que no total foram 10, lembrando que o membro 1 era o criador do grupo.

Tabela 8: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups jutamente com o protocolo TUNNEL no modelo de testes 5

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
M1	2ms	1ms	3ms	3ms	2ms	2ms	2ms	2ms	2ms	1ms
M2	5ms	7ms	5ms	8ms	6ms	10ms	6ms	12ms	4ms	5ms
M3	11ms	5ms	5ms	5ms	5ms	6ms	13ms	5ms	7ms	4ms

Na Figura 34 é possível visualizar como o modelo de testes 6 foi elaborado:

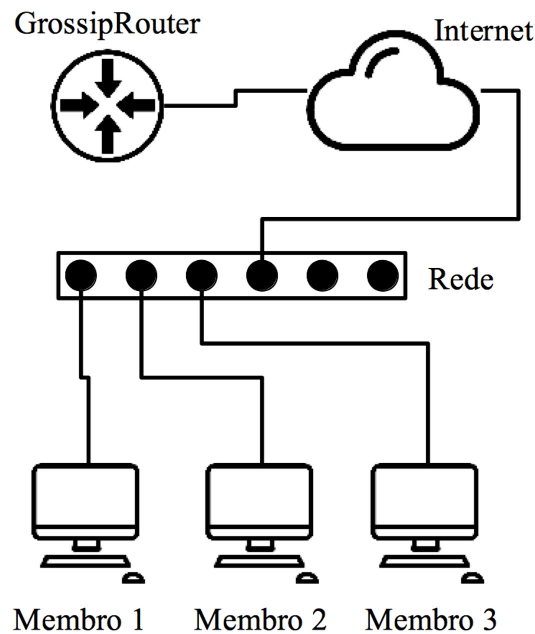

Figura 34: Arquitetura distribuída - configuração 3.

Assim como no modelo de testes 5, o *GrossipRouter* será novamente utilizado para que *peers* em diferentes redes possam conectar-se ao grupo, porém a grande diferença aqui, será que o *GrossipRouter* estará presente em uma rede diferente da rede dos membros do grupo. O protocolo TUNNEL foi novamente utilizado, sendo que os dados são repassados para o protocolo TCP, e então entregues aos membros.

Foram executados 10 testes nas três máquinas, sendo que os resultados desses testes podem ser visualizados na Tabela 9, onde *M1* significa membro 1, *M2* membro 2, *M3* membro 3, e *T1, T2, ..., T10* os testes executados, que no total foram 10, lembrando que o membro 1 era o criador do grupo.

Tabela 9: Resultados do tempo de resposta em milissegundos utilizando o *middleware* JGroups juntamente com o protocolo TUNNEL no modelo de testes 6

	T1	**T2**	**T3**	**T4**	**T5**	**T6**	**T7**	**T8**	**T9**	**T10**
M1	2ms	1ms	3ms	2ms	2ms	2ms	2ms	1ms	1ms	1ms
M2	9ms	6ms	7ms	6ms	8ms	10ms	8ms	10ms	7ms	10ms
M3	7ms	9ms	5ms	6ms	11ms	9ms	12ms	9ms	8ms	8ms

5.4. Análise de resultados

Baseando-se nos testes executados nos cenários anteriores, foram geradas 6 figuras, para uma representação visual do tempo de resposta necessário em cada modelo de testes utilizando protocolos como TCP, UDP e TUNNEL do JGroups, lembrando que o tempo de resposta foi medido em milissegundos e é exibido na vertical, enquanto que horizontalmente *T1, T2, ..., T10* representa os testes que foram executados.

Na Figura 35 é apresentado um gráfico referente ao modelo de testes 1, onde a linha na cor vermelha representa o protocolo TCP e a linha na cor azul turquesa representa o protocolo UDP para o cliente 1, de forma similar a linha na cor rosa representa o protocolo TCP e a linha na cor azul verde representa o protocolo UDP para o cliente 2.

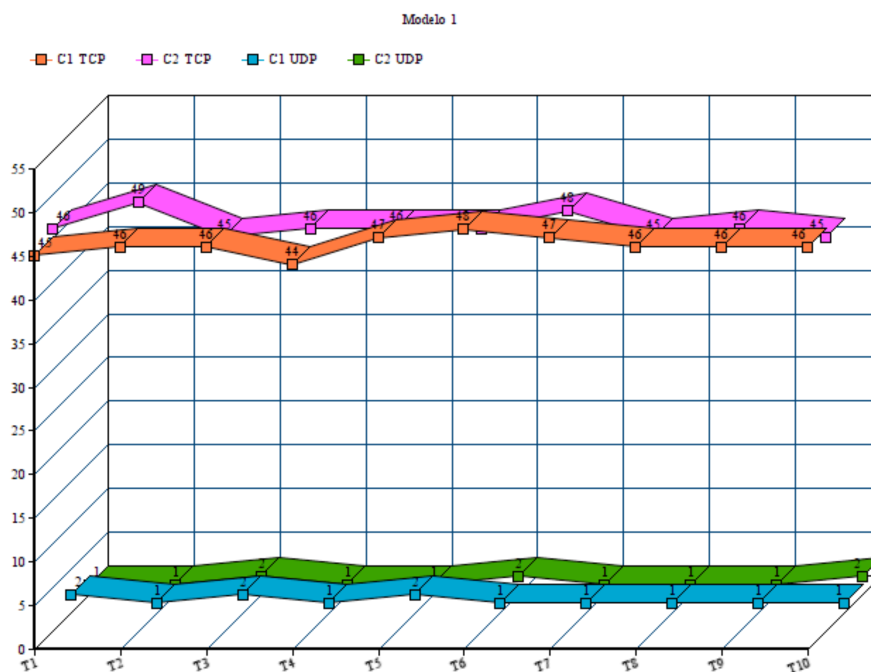

Figura 35: Gráfico referente ao modelo de testes 1

Na Figura 36 é apresentado um gráfico referente ao modelo de testes 1, onde a linha na cor vermelha representa o protocolo TCP e a linha na cor azul turquesa representa o protocolo UDP para o cliente 1, de forma similar a linha na cor rosa representa o protocolo TCP e a linha na cor azul verde representa o protocolo UDP para o cliente 2.

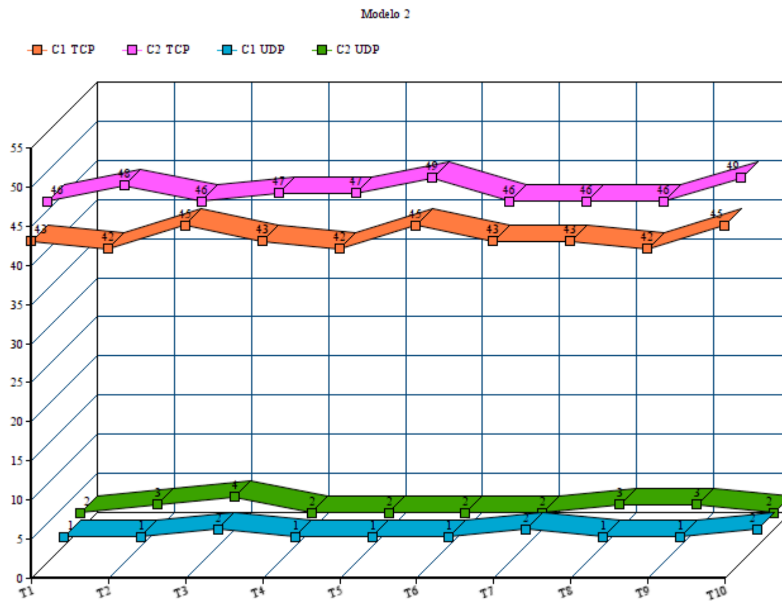

Figura 36: Gráfico referente ao modelo de testes 2

Na Figura 37 é apresentado um gráfico referente ao modelo de testes 3, onde a linha na cor vermelha representa o membro 1, a linha na cor rosa representa o membro 2 e a linha na cor azul turquesa representa o membro 3, todos os membros estavam utilizando o JGroups para estabelecer a comunicação, lembrando que o membro 1 era o criador do grupo.

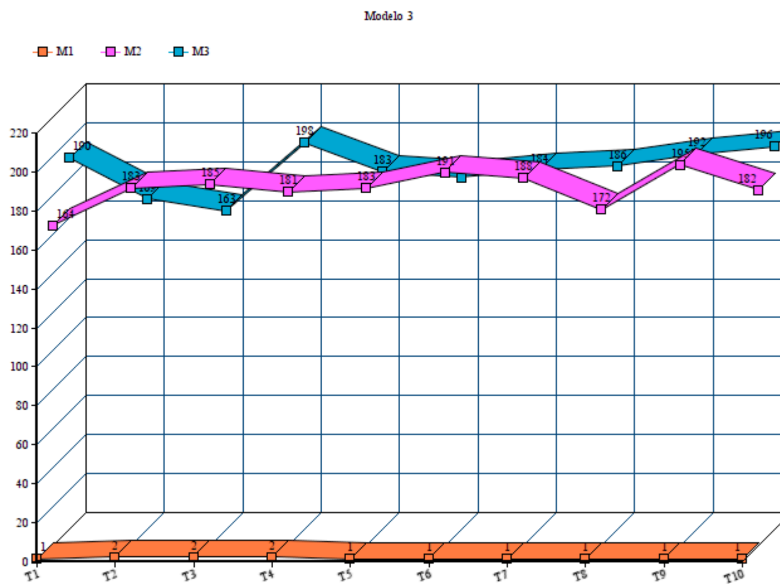

Figura 37: Gráfico referente ao modelo de testes 3

Na Figura 38 é apresentado um gráfico referente ao modelo de testes 4, onde a linha na cor vermelha representa o protocolo TCP e a linha na cor azul turquesa representa o protocolo UDP para o cliente 1, de forma similar a linha na cor rosa representa o protocolo TCP e a linha na cor azul verde representa o protocolo UDP para o cliente 2.

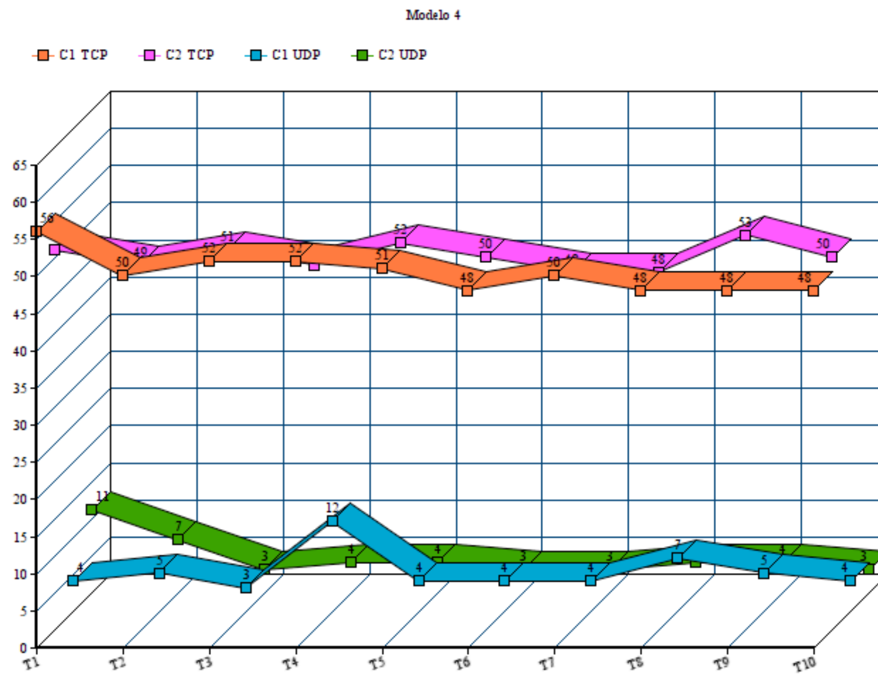

Figura 38: Gráfico referente ao modelo de testes 4

Na Figura 39 é apresentado um gráfico referente ao modelo de testes 5, onde a linha na cor vermelha representa o membro 1, a linha na cor rosa representa o membro 2 e a linha na cor azul turquesa representa o membro 3, todos os membros estavam utilizando o JGroups para estabelecer a comunicação, lembrando que o membro 1 era o criador do grupo.

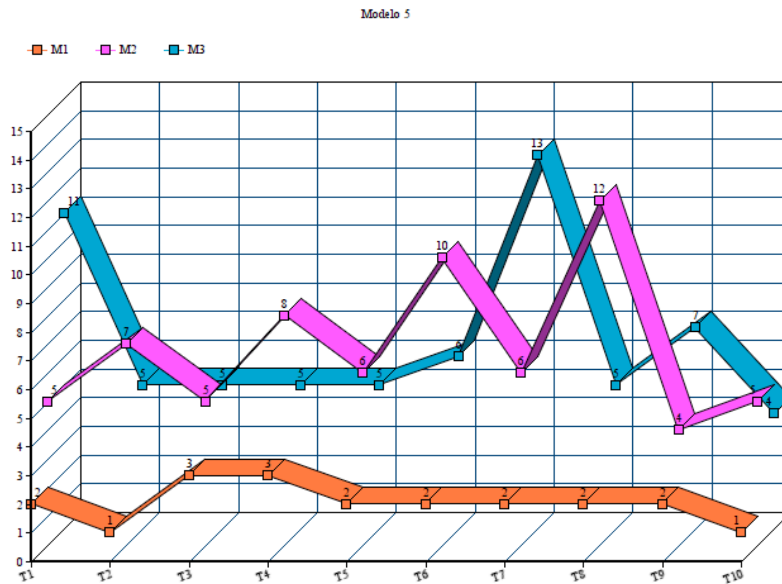

Figura 39: Gráfico referente ao modelo de testes 5

Na Figura 40 é apresentado um gráfico referente ao modelo de testes 6, onde a linha na cor vermelha representa o membro 1, a linha na cor rosa representa o membro 2 e a linha na cor azul turquesa representa o membro 3, todos os membros estavam utilizando o JGroups para estabelecer a comunicação, lembrando que o membro 1 era o criador do grupo.

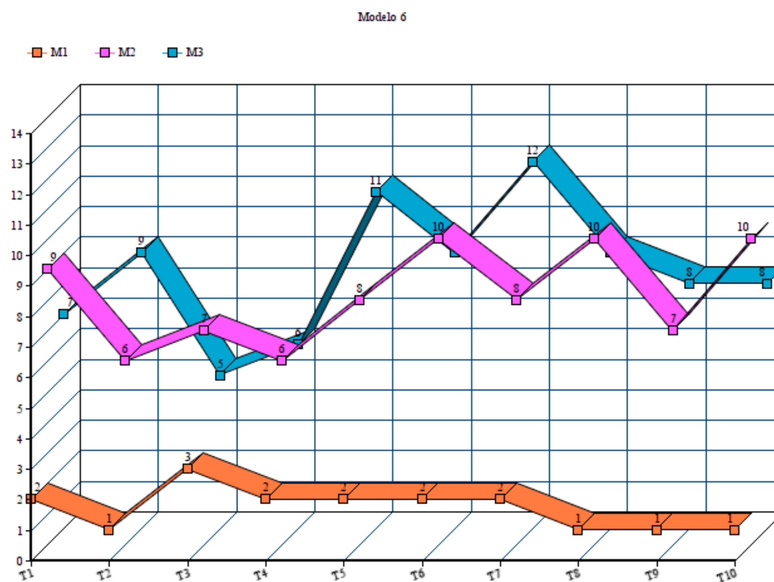

Figura 40: Gráfico referente ao modelo de testes 6

A partir dos gráficos anteriores é possível gerar um novo gráfico com a média simples de cada protocolo utilizado em cada modelo de testes (Figura 41).

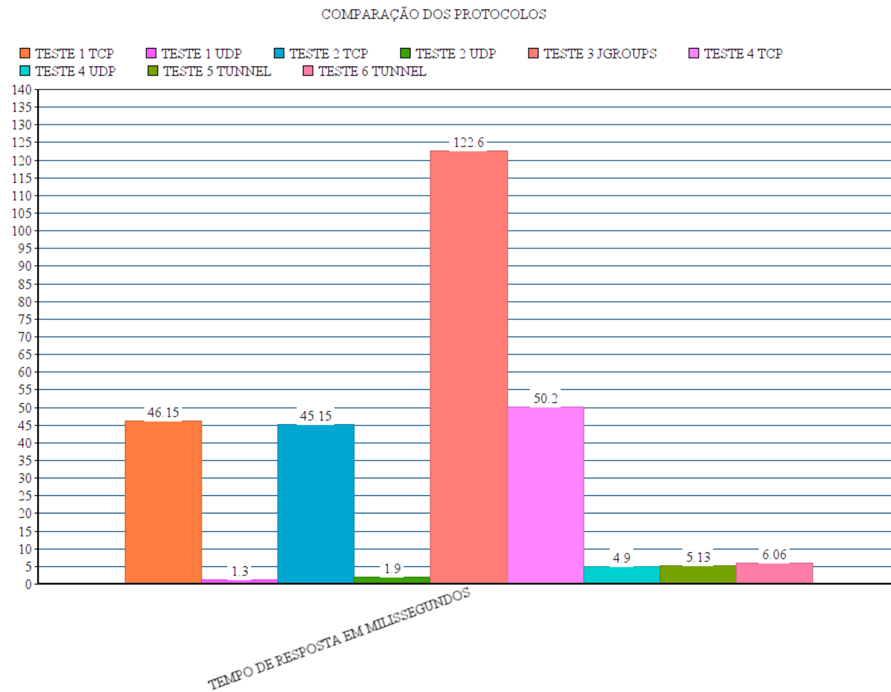

Figura 41: Média simples do tempo de resposta utilizando os protocolos TCP, UDP e TUNNEL do JGroups.

Com os resultados das médias dos tempos de reposta de cada modelo de testes, pode-se perceber que, como já era esperado, o *middleware* JGroups foi quem obteve os piores resultados e o protocolo UDP os melhores.

O JGroups utiliza como protoloco padrão uma adaptação do UDP, com funções que o tornam orientado a conexões e confiável como o TCP, porém como essas funções são implementadas em nível de *software*, o desempenho é afetado significativamente, tornando-o mais pesado até mesmo que o protocolo TCP, ultrapassando os 100ms de tempo de resposta em redes próximas, algo inviável para a execução de servidores de jogos dedicados em redes de longa distância, pois dessa forma, a latência será muito alta.

O protocolo TUNNEL utilizado pelo JGroups mostrou-se eficaz, mantendo uma média de latência muito próxima ao UDP, porém na documentação do JGroups é possível observar

que o protocolo TUNNEL repassa os dados para o protocolo TCP, para que em seguida sejam entregues aos membros conectados ao grupo.

Segundo a documentação do JGroups "*O TUNNEL estabelece uma conexão TCP com o processo GossipRouter (fora do firewall) que aceita mensagens de membros e as repassa para outros membros.*" [22].

Portanto o JGroups deve utilizar algum processo através do protocolo TUNNEL que faz com que o protocolo TCP torne-se mais rápido, ou então, o real protocolo utilizado é o UDP, pois visualizando o gráfico na figura 44 é possível observar que seu tempo de resposta é muito próximo ao UDP. É necessário executar mais testes para perceber o que acontece neste caso.

Com o protocolo TCP obteve-se os resultados esperados, ele é lento, porém muito superior ao protocolo padrão do JGroups.

O protocolo UDP mostrou-se o mais eficaz, muito rápido, porém sem a confiabilidade presentes nos outros protocolos, possibilitando assim que erros sejam executados durante longas seções de jogos.

5.5. Resultados

Como resultados finais, um módulo de comunicação para o motor de jogos jMonkeyEngine foi criado, com a possibilidade da utilização tanto da arquitetura cliente/servidor como da P2P.

Nos testes foi possível perceber na prática a grande diferença de tempo com os diferentes protocolos de comunicação utilizados e que, apesar do protocolo UDP não ser confiável, ele é muito rápido, com diferença de quase menos 50ms quando comparado com o protocolo TCP.

Baseando-se nisso, percebe-se que deve-se utilizar o protocolo UDP em conjunto com o protocolo TCP, ou seja, fazer com que as ações dos jogadores que não necessitarem de garantia de entrega, como por exemplo, movimentação no cenário, podem ser enviadas

utilizando o protocolo UDP. Ações que necessitem garantia de entrega, como por exemplo, mensagens de texto ou operações de *login*, devem ser executadas através da utilização do protocolo TCP.

Na adaptação do jogo para a sua execução em conjunto com o JEagle, foi possível perceber que existem vários problemas que devem ser corrigidos no jogo, como por exemplo a correção da posição dos jogadores no mundo de jogo de cada jogador, pois nos testes executados com o JEagle, principalmente com a utilização do protocolo UDP, muitas das vezes as posições dos jogadores não se mostravam iguais. Felizmente esse problema pode ser minimizado com técnicas de interpolação, para a predição e correção de posições dos jogadores. Como já foi dito em capítulos anteriores, esse problema não foi corrigido porque o foco do trabalho era o desenvolvimento dos módulos de comunicação, e não do jogo em si.

Com os resultados obtidos através da utilização do JGroups ficou evidente que deveria ter sido utilizado outro *middleware* para o desenvolvimento do módulo de comunicação descentralizado, de forma que a latência não fosse tão alta, ou pelo menos fosse parecida com os resultados do protocolo TCP. No entanto, o protocolo TUNNEL, que depois repassa os dados para o protocolo TCP, por algum motivo mostrou-se rápido ao nível do protocolo UDP, mas nos testes normais o JGroups teve uma latência superior a 100ms.

Apesar do *middleware* JGroups ser relativamente lento quando utilizado com redes externas e de longa distância, ele mostra-se funcional e confiável para redes locais, podendo então ser utilizado para jogos em LAN.

Capítulo 6 Conclusões

Este capítulo irá sintetizar as conclusões geradas a partir da análise dos resultados obtidos após a implementação e execução bem-sucedida do JEagle.

6.1. Conclusão

Desenvolver módulos de comunicação para que os jogos possam compartilhar um mundo de jogo entre vários jogadores através da Internet é uma tarefa excitante e desafiadora, pois ela requer uma visão diferente sobre como as coisas funcionam, ou seja, o jogo deve funcionar de forma simultânea entre vários jogadores que estarão dispersos em diferentes localizações e em diferentes máquinas. O desenvolvedor também precisa aprender conceitos que, a um primeiro momento, se mostram um pouco avançados, como por exemplo, os protocolos que serão planejados e como eles serão processados, o que deve ser feito para as ações dos jogadores serem replicadas na rede e o motivo pelo qual os jogadores visualizam apenas simulações replicadas e não mais uma ação verdadeira, ou seja, a abordagem de desenvolvimento de jogos é totalmente diferente para jogos multijogador.

A implementação do JEagle foi executada de forma simples, organizada e ágil, de forma que todos os componentes do projeto estão separados por pacotes e cada objeto possui sua função e responsabilidade bem definida. A linguagem de programação Java possibilitou com que o desenvolvimento desse módulo de conexão fosse rápido e flexível, pois muitas classes, objetos e métodos já estão previamente prontos, nos restando então apenas a

implementação final do módulo, portanto, essa linguagem de programação mostrou-se essencial para o desenvolvimento desse trabalho.

Através dos resultados obtidos foi possível perceber que o módulo de comunicação centralizado se mostrou muito mais eficiente que o módulo de comunicação descentralizado, pois a diferença de latência presente entre esses módulos de comunicação foi gigantesca. Portanto o que deve ser feito nos trabalhos futuros é a troca do *middleware* utilizado para estabelecer a comunicação P2P.

Como trabalhos futuros deve ser procurado outro *middleware* P2P que seja mais rápido que o JGroups, de forma que seja possível executar um jogo *multiplayer* utilizando essa arquitetura de comunicação. Além disso, também deve ser implementado uma solução que identifique os níveis de latência de rede, ou seja, quando a latência for muito alta, a arquitetura de comunicação deve ser trocada de cliente/servidor para P2P e vice-versa, de forma que a latência da rede se estabilize, ou então o criador do grupo, ou a máquina que esta executando o servidor deve ser alterada para outro integrante do jogo, porém o mundo do jogo presente em todos os integrantes não deve perder-se durante esse processo.

Também deve ser implementado um módulo de comunicação centralizado que mescle o protocolo TCP com o UDP, de forma que cada protocolo seja utilizado dependendo das ações que o jogador for executar. Espera-se que esse processo aumente de forma significativa a velocidade de comunicação entre os integrantes do jogo. Resultado, tal, que será verificado nos trabalhos futuros.

O software desenvolvido com este trabalho permite dar a possibilidade de escolha ao desenvolvedor, de forma a que possa utilizar o JEagle para o desenvolvimento de um jogo *multiplayer* ou até mesmo continuar o desenvolvimento desse módulo de comunicação adicionando melhorias e atualizações, numa filosofia de código aberto.

Bibliografia

[1] J. R. Bittencourt e F. S. Osório, “Motores para Criação de Jogos Digitais: Gráficos, Áudio, Interação, Rede, Inteligência Artificial e Física,” Unisinos, 2006.

[2] F. É. Gallão, “DESENVOLVIMENTO DE JOGO DISTRIBUÍDO EM REDE UTILIZANDO UNREAL DEVELOPMENT KIT”, Monografia, Universidade de São Francisco, Itatiba, 2011.

[3] Carlos Guerber. (2009, Ago.). *REDES DE COMPUTADORES* [Online]. Disponível: http://www.mfa.unc.br/info/carlosrafael/redes1/aula2A.pdf

[4] Renan O. Rios, *Protocolos e Serviços de Redes*. Espirito Santo: MEC, 2012.

[5] A. S. Tanenbaum e D. J. Wetherall, *Redes de computadores*. São Paulo: Pearson Prentice Hall, 2011.

[6] Carlos Guerber. (2007, Out.). *PROTOCOLO IP* [Online]. Disponível: http://www.mfa.unc.br/info/carlosrafael/rco/ip.pdf

[7] Kurose, J. F. e Ross, K. W. *Redes de Computadores e a Internet: Uma nova abordagem*. São Paulo: Addison Wesley, 2003.

[8] Z-World Inc. (2001). *An Introduction to TCP/IP* [Online]. Disponível: https://www.jameco.com/Jameco/Products/ProdDS/320733%20(TCP%20IP)%20Intro.pdf

[9] H. S. Oluwatosin, “Client-Server Model” IOSR J. Comput. Eng., vol. 16, no. 1, pp. 57–71, 2014.

[10] T. A. Rizzetti, C. Trois, J. Carlos, D. Lima, and I. Augustin, “Ambiente Colaborativo P2P para o projeto PDSCE.”

[11] J. Martins e H. Santana, “Qualidade de Serviço (QoS) em Redes IP Princípios Básicos, Parâmetros e Mecanismos,” Cursos Telecom e Telemática, 2006.

[12] *Socket Programming* [Online]. Disponível: http://www.buyya.com/java/Chapter13.pdf

[13] International Business Machines. (2006, Fev.). *System i: Programming Socket programming* [Online]. Disponível: https://www.ibm.com/support/knowledgecenter/ssw_i5_54/rzab6/rzab6.pdf

[14] C. P. Souza, “Desenvolvimento 3D em Java - modelo de visualização, sons e billboards Desenvolvimento 3D em Java,” no. January 2011, 2014.

[15] D. Johnston, “3D Game Engines as a New Reality,” Proc. 4th Annu. C. Conf. …, vol. 1, no. 4, pp. 36–41, 2004.

[16] D. Türpitz, “LWJGL - Eine Ausarbeitung von Daniel Türpitz im Rahmen der Lehrveranstaltung Graphisch-Interaktive Systeme,” Ausarbeitung, pp. 1–29, 2008.

[17] (2016, Mar.). *SpiderMonkey: Multi-Player Networking* [Online]. Disponível: https://wiki.jmonkeyengine.org/jme3/advanced/networking.html

[18] *JGroups - A Toolkit for Reliable Messaging* [Online]. Disponível: http://jgroups.org/overview.html

[19] Ken Slonneger. (2007). *Serialization* [Online]. Disponível: http://homepage.divms.uiowa.edu/~slonnegr/wpj/Serialization.pdf

[20] *Class SerializationUtils* [Online]. Disponível: https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/SerializationUtils.html

[21] H. W. Site, T. Web, U. Richter, and R. Nayak, “Impact of Object Serialization and Local Enterprise JavaBeansTM on Application Server Performance,” pp. 1–22, 2003.

[22] *jboss.org. Chapter 5. Advanced Concepts* [Online]. Disponível: http://www.jgroups.org/manual/html/user-advanced.html

Anexos

A seguir serão explicadas, de forma detalhada, cada classe que esta presente nos módulos de comunicação.

Módulo de comunicação centralizado

Classes comuns aos protocolos TCP e UDP

A classe *ClientInformations*

Essa classe é a responsável por armazenar as informações relativas a um cliente específico, ela irá herdar a classe *Informations*.

Na Figura 42 é possível ver um diagrama de classes que representa essa classe.

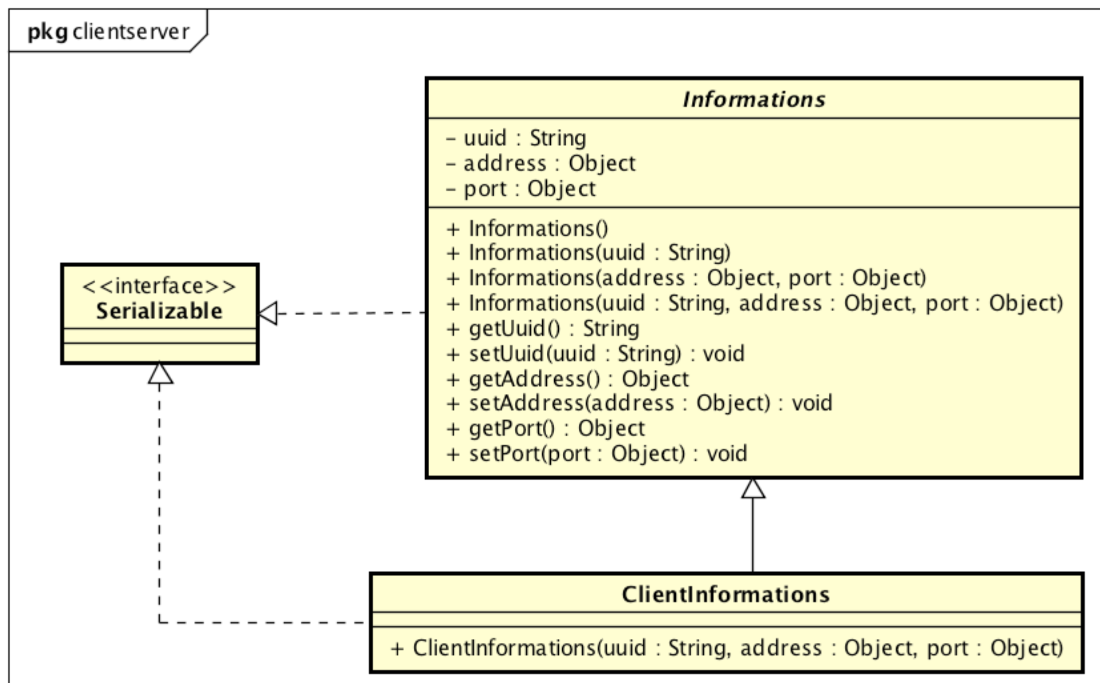

Figura 42: A classe *ClientInformations*

Os seguintes atributos estarão presentes nessa classe:

- *private String uuid* - Esse atributo irá armazenar um identificador único na rede, tornando o cliente diferente dos outros conectados ao servidor.

- *private Object address* - Será do tipo *Object*, pois assim, esse atributo poderá armazenar uma representação textual de endereços do tipo *String* para o protocolo TCP, ele também poderá armazenar objetos do tipo *InetAddress* para o protocolo UDP.
- *private Object port* - Assim como no atributo anterior, o atributo *port*, para o protocolo TCP, poderá ser utilizado uma representação textual do tipo *String* da porta que um determinado cliente utiliza, da mesma forma, ele poderá ser utilizado como um objeto do tipo *InetAddress* para o mesmo fim, porém para o protocolo UDP.

Além dos métodos *set()* e *get()* que estão presentes através de herança, o seguinte método também faz parte dessa classe:

- *public ClientInformations(String uuid, Object address, Object port)* - O único método aqui presente, e também construtor da classe, instância a superclasse *Informations*, para que seja então possível armazenar as informações de um cliente específico.

A classe abstrata *DataToSend*

Essa classe precisa ser estendida por classes filhas, e é ela quem armazena as informações que serão enviadas nos pacotes ou datagramas, como *headers* e *contents*.

Na Figura 43 é possível ver um diagrama de classes que representa essa classe.

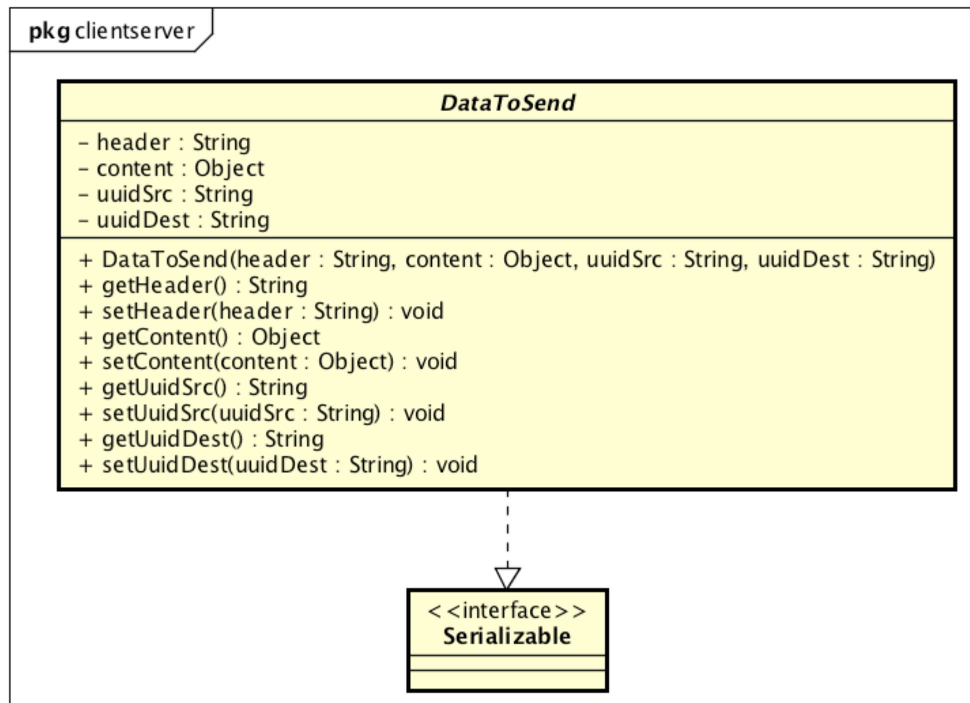

Figura 43: A classe abstrata *DataToSend*

Os atributos que estão presentes nessa classe são:

- *private String header* - Armazena o cabeçalho da mensagem, sendo que através desse cabeçalho, o cliente ou servidor pode compreender a mensagem recebida e então posteriormente processá-la.
- *private Object content* - Contém o objeto que será enviado na mensagem, através desse atributo será possível trocar dados entre os integrantes que estiverem conectados ao servidor.
- *private String uuidSrc* - Esse atributo tem o identificador único do cliente que envia a mensagem, para a sua posterior identificação.
- *private String uuidDest* - Da mesma forma que o atributo anterior, o *uuidDest* é quem armazena o identificador único do cliente que deverá receber a mensagem.

Além dos métodos de *set()* e *get()*, essa classe apresente o seguinte método construtor:

- *public DataToSend(String header, Object content, String uuidSrc, String uuidDest)* - Esse método é quem inicia todos os atributos da classe, lembrando que essa classe não pode ser instanciada, mas as classes filhas também irão conter esses atributos por herança, de forma que poderão iniciar seus atributos através desse método.

A classe *Datagram*

Será a responsável por abstrair uma mensagem para a arquitetura cliente/servidor utilizando o protocolo UDP. Ela irá estender a classe abstrata *DataToSend* e, portanto, conterá todos os atributos e métodos da superclasse.

Na Figura 44 é possível ver um diagrama de classes que representa essa classe.

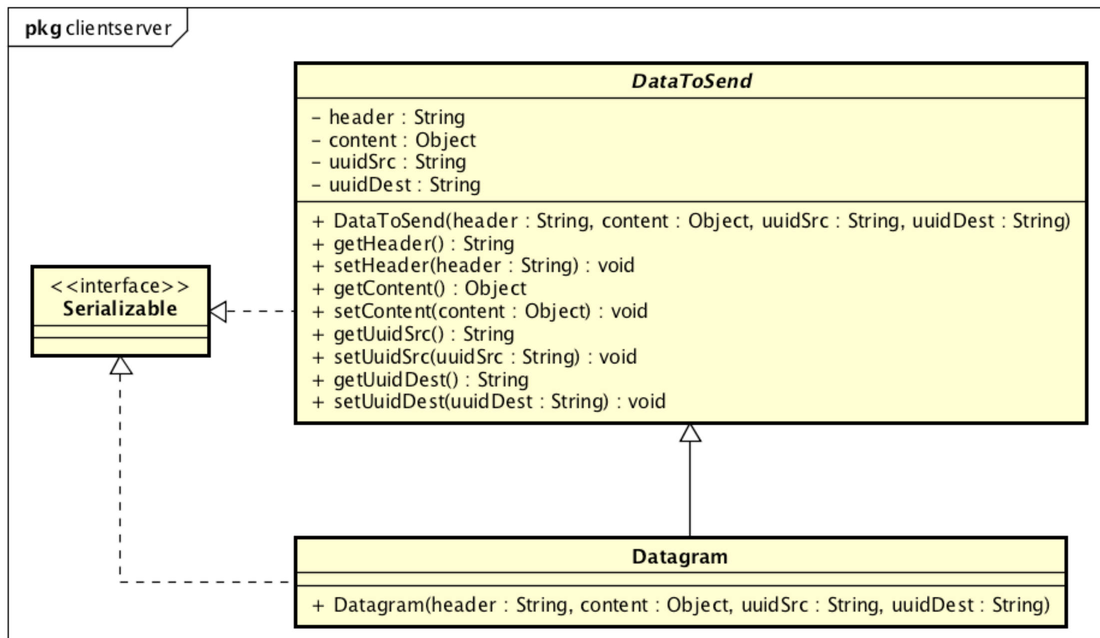

Figura 44: A classe *Datagram*

Essa classe conterá então os seguintes atributos:

- *private String header* - Irá representar o tipo de mensagem que estará sendo enviado para os integrantes do servidor.
- *private Object content* - armazenará o objeto que será trocado entre os integrantes do grupo.
- *private String uuidSrc* - Contém o *uuid* do cliente emissor.
- *private String uuidDest* - Da mesma forma como o atributo *uuidSrc*, esse atributo irá armazenar o *uuid* do cliente que receberá a mensagem.

Além dos métodos presentes na classe abstrata *DataToSend*, essa classe conterá o seguinte método:

- *public Datagram(String header, Object content, String uuidSrc, String uuidDest)* - É o único método que estará presente na classe, e, que por sua vez, terá a

finalidade de inicializar os atributos da superclasse, que através dos conceitos de herança, também serão pertencentes a classe *Datagram*.

A classe abstrata *Informations*

Conterá informações relativas aos clientes e ao servidor, ou seja, ela será a superclasse da classe *ClientInformations* e também da classe *ServerInformations*.

Na Figura 45 é possível ver um diagrama de classes que representa essa classe.

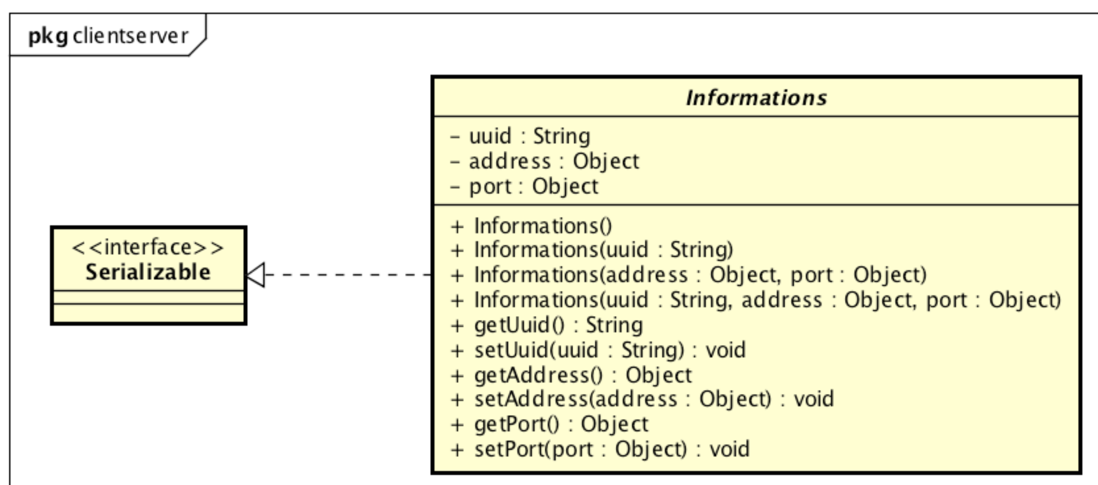

Figura 45: A classe abstrata *Informations*

Sendo assim, os seguintes atributos estarão presentes:

- *private String uuid* - Esse atributo armazena um identificador único textual para o servidor ou para um cliente.
- *private Object address* - Armazena um endereço que pode ser representado de forma textual do tipo *String* ou então o tipo *InetAddress*.
- *private Object port* - Armazenará a porta de um cliente ou do servidor utilizando uma representação textual do tipo *String* ou então do tipo *InetAddress*.

Os três métodos presentes nessa classe, além dos métodos de *set()* e *get()*, serão construtores que conterão diferentes parâmetros. Os métodos construtores são:

- *public Informations(String uuid)* - Instanciará a classe *Informations* inicializando o atributo *uuid* do tipo *String*.

- *public Informations(Object address, Object port)* - Instanciará a classe *Informations* inicializando os atributos *address* e *port* com os valores obtidos através dos parâmetros desse método construtor.
- *public Informations(String uuid, Object address, Object port)* - Também instanciará a classe *Informations*, porém agora inicializando os atributos *uuid*, *address* e *port* que estão presentes nessa classe.

A classe *Packet*

É a classe que representa um pacote enviado dos clientes para o servidor e também do servidor para os clientes quando o protocolo de comunicação TCP for utilizado.

Na Figura 46 é possível ver um diagrama de classes que representa essa classe.

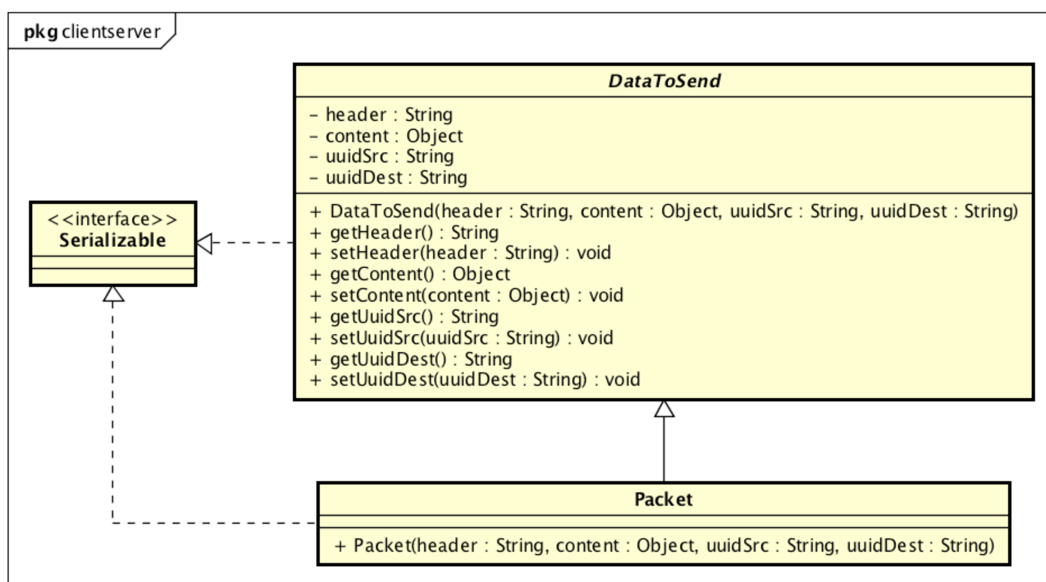

Figura 46: A classe *Packet*

Essa classe é uma classe filha da classe abstrata *DataToSend* e que, devido aos conceitos de herança, herdará os seguintes atributos:

- *private String header* - É o responsável por informar aos clientes ou ao servidor qual ação tomar quando o pacote for recebido.
- *private Object content* - Contém o objeto da mensagem que será enviado dos clientes para o servidor, ou então do servidor para os clientes.

- *private String uuidSrc* - Armazena o identificador único do cliente que enviou a mensagem.
- *private String uuidDest* - Contém o identificador único do cliente para o qual a mensagem esta sendo enviada.

Os métodos que estão presentes nessa classe além dos métodos de *set()* e *get()*, serão comuns a classe *DataToSend*, pois a classe *Packet* é uma de suas classes filhas, portanto, o único método diferente será descrito a seguir:

- *public Packet(String header, Object content, String uuidSrc, String uuidDest)* - Como a classe *DataToSend* é uma classe abstrata e não pode ser instanciada, a classe *Packet* irá ser instanciada através desse método construtor, e assim sendo, inicializará seus atributos através de seus parâmetros que estão presentes na superclasse *DataToSend* que foram herdados através de herança.

A classe *Ping*

Tem por finalidade calcular o tempo necessário, que pode ser em milissegundos, segundos, minutos, horas ou dias, para que um pacote ou datagrama seja enviado de um ponto A, chegue a um ponto B, e então volte novamente para o ponto A. O cálculo executado para calcular o *ping* é:

$$ping = timePacketReceive\text{-}timePacketSend$$

Na Figura 47 é possível ver um diagrama de classes que representa essa classe.

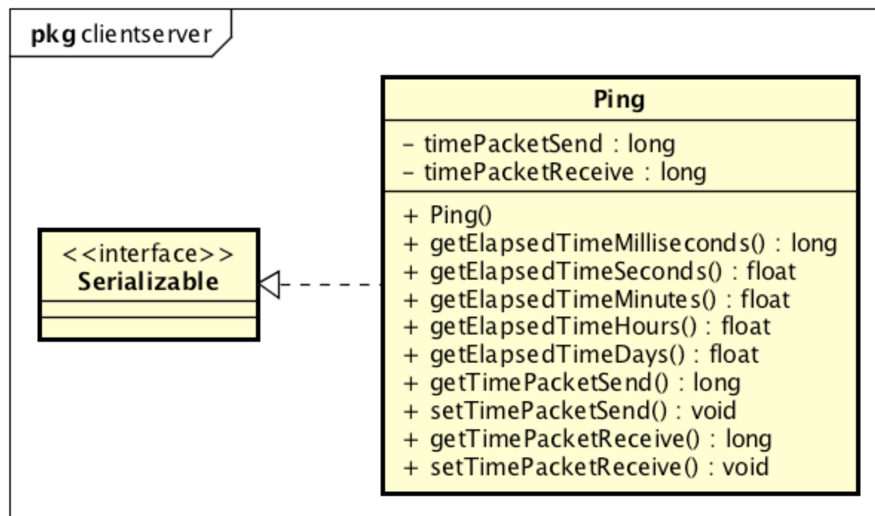

Figura 47: A classe *Ping*

Os seguintes atributos estarão presentes nessa classe:

- *private long timePacketSend* - Armazena o momento em que o pacote ou datagrama foi enviado do cliente emissor para o servidor.
- *private long timePacketReceive* - Contém o momento em que o pacote ou datagrama voltou para o cliente que anteriormente tinha requisitado a contagem de *ping* para o servidor.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Ping()* - É um construtor vazio, existente apenas para a devida inicialização da classe.
- *public long getElapsedTimeMilliseconds()* - Calcula o *ping* em milissegundos.
- *public float getElapsedTimeSeconds()* - Calcula o *ping* em segundos.
- *public float getElapsedTimeMinutes()* - Calcula o *ping* em minutos.
- *public float getElapsedTimeHours()* - Calcula o *ping* em horas.
- *public float getElapsedTimeDays()* - Calcula o *ping* em dias.

A classe *ServerInformations*

É uma das classes filhas da superclasse *Informations*, portanto ela irá herdar todos os atributos e métodos que estão presentes em sua classe pai.

Na Figura 48 é possível ver um diagrama de classes que representa essa classe.

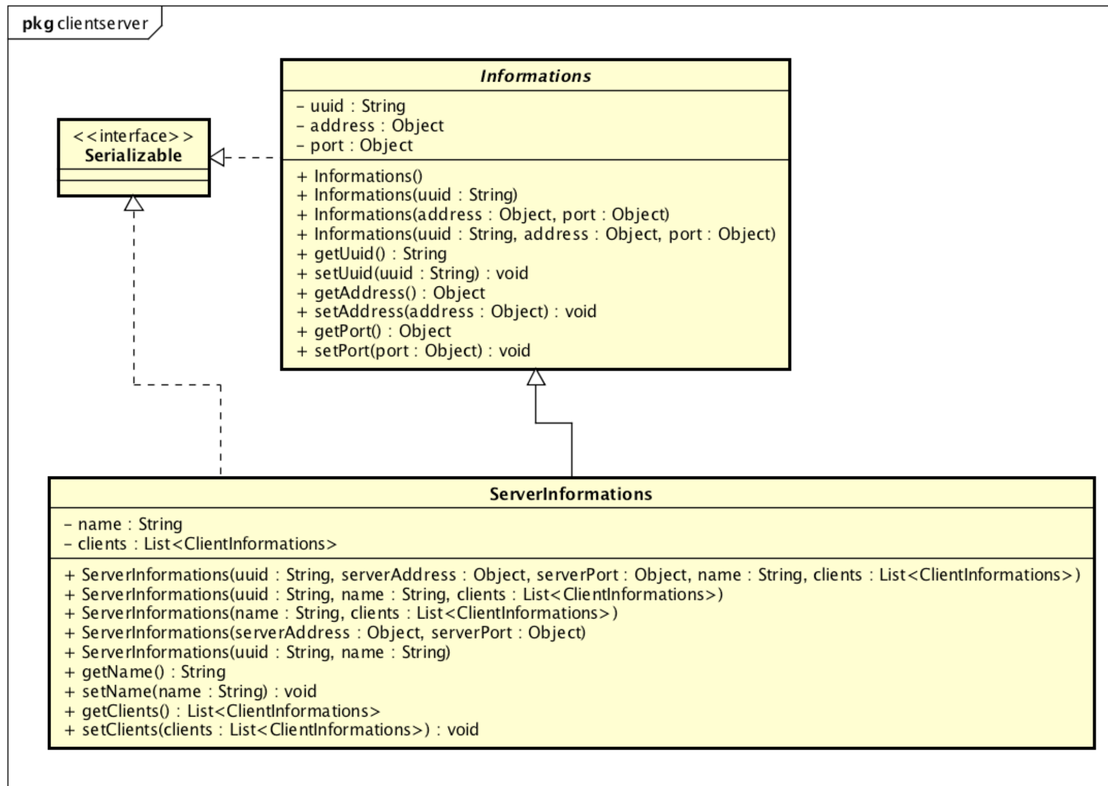

Figura 48: A classe *ServerInformations*

Além dos atributos herdados de sua classe pai, essa classe também contém dois outros atributos, são eles:

- *private String name* - Armazena o nome do servidor.
- *private List<ClientInformations> clients* - Armazena a lista de clientes conectados ao servidor.

Esses dois atributos são inicializados assim que a conexão do cliente com o servidor termina, onde o servidor envia uma mensagem para o cliente que conectou-se com essas informações atualizadas.

A seguir será possível visualizar que os métodos apresentados são métodos construtores, porém com parâmetros diferentes, portanto cada um deles é utilizado para inicializar essa classe de forma diferenciada. Dessa forma, além dos métodos de *set()*, *get()* e dos métodos que foram herdados da sua superclasse *Informations*, os seguintes métodos também estão presentes nessa classe:

- *public ServerInformations(String uuid, Object serverAddress, Object serverPort, String name, List<ClientInformations> clients)* - Além de instanciar a classe *ServerInformations* e inicializar os seus atributos *name* e *clients*, esse método também irá inicializar os atributos *uuid*, *address* e *port* que foram herdados da superclasse *Informations*.
- *public ServerInformations(String uuid, String name, List<ClientInformations> clients)* - Irá instanciar a classe *ServerInformations* e inicializar os atributos *name* e *clients* que pertencem a essa classe, além de também inicializar o atributo *uuid* que esta pertence a superclasse *Informations*.
- *public ServerInformations(String name, List<ClientInformations > clients)* - Não inicializa os atributos presentes na superclasse *Informations*, mas inicializa os atributos presentes na classe *ServerInformations*, ou seja, os atributos *name* e *clients*.
- *public ServerInformations(Object serverAddress, Object serverPort)* - Tem como finalidade inicializar dois atributos presentes na superclasse *Informations*, ou seja, ele irá inicializar os atributos *address* e *port*.
- *public ServerInformations(String uuid, String name)* - Esse método tem como finalidade inicializar o atributo *uuid* que esta presente na superclasse *Informations* e o atributo *name* que esta presente na classe *ServerInformations*.

Classe utilizadas no cliente do protocolo TCP

A classe *Client*

Representa o cliente que posteriormente irá conectar-se ao servidor, abstraindo o conceito de cliente, ele será o canal de comunicação entre a máquina local e o servidor.

Na Figura 49 é possível ver um diagrama de classes que representa essa classe.

Figura 49: A classe *Client*

Os seguintes atributos estão presentes nessa classe:

- *private Socket socket* - Ponto final entre a comunicação de duas máquinas, ou seja, a comunicação entre o cliente local e o servidor.
- *private ObjectInputStream objectInputStream* - Armazena um objeto deserializado que foi previamente escrito através de um *ObjectOutputStream* pelo servidor.
- *private ObjectOutputStream objectOutputStream* - Armazena um objeto serializado que será posteriormente enviado a um *ObjectInputStream* que esta presente no servidor.
- *private ServerInformations serverInformations* - Armazena as informações do servidor, como o identificador único, endereço, porta, nome do servidor e também a lista de clientes conectados.
- *private boolean running* - Mantém a *thread* principal sendo executada.
- *private Receiver receiver* - Mantém uma referência para a classe que receberá os pacotes recebidos pelo servidor para então processa-los.
- *private Thread threadReceiver* - É a *thread* que executará a classe *Receiver*, sendo executado de forma simultânea com o cliente, essa classe será capaz de receber todos os pacotes que forem enviados do servidor.
- *private String uuid* - Contém uma representação textual do identificador único do cliente.
- *private boolean connected* - O atributo booleano *connected* altera seu valor lógico de falso para verdadeiro quando o cliente tem sua conexão aceita com o servidor.
- *private HandleReceiverControl handleReceiverControl* - Armazena a *interface* responsável por injetar o código customizado do desenvolvedor externo ao código do JEagle.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Client(String serverAddress, int serverPort)* - É o construtor da classe, que por sua vez inicializará o atributo *serverInformations* com o endereço e porta do servidor.
- *public void run()* - É o responsável por iniciar a *thread* principal que manterá o cliente sendo executado enquanto o atributo booleano *running* conter um valor lógico verdadeiro.

- *public void connect()* - Estabiliza a conexão do cliente com o servidor e além de inicializar seus atributos também inicializa a *thread* da classe *Receiver*.
- *public void disconnect()* - Fecha o *socket* que estabeleceu a conexão entre o cliente e o servidor, esse método também executa o método *stop()* e fecha os atributos *objectInputStream* e *objectOutputStream*.
- *public void stop()* - Finaliza as *threads* que tiverem relação com a classe *client* e que também estiverem sendo executadas, além disso, esse método também altera os valores lógicos das variáveis booleanas *running* e *connected* de verdadeiro para falso.
- *public void send(Packet packet)* - Tem a finalidade de enviar um pacote de um cliente para o servidor.
- *public void getPing()* - Envia um pacote para o servidor com o seguinte *header* "GET_PING", para mais tarde receber o mesmo pacote novamente como resposta do servidor, para que seja possível contar a latência presente na rede.

A *interface HandleReceiverControl*

É a responsável por injetar os códigos do desenvolvedor no código fonte do JEagle, de forma que métodos e ações customizados possam ser executados. Essa *interface* precisa ser posteriormente implementada em uma outra classe qualquer, que será escolhida pelo desenvolvedor externo. O desenvolvedor externo pode utilizar qualquer *header* que desejar no método *receive*, contanto que não seja nenhuma das seguintes palavras reservadas abaixo:

- CONNECTION_SUCESS_RESPONSE
- CONNECTION_REFUSED_RESPONSE
- GET_CLIENTS_LIST_RESPONSE
- GET_PING_RESPONSE
- CLIENT_ACTION_JOIN
- CLIENT_ACTION_LEFT
- GET_PING_RESPONSE_FIX
- SERVER_INFORMATIONS_UPDATE
- GET_CLIENTS_LIST
- GET_PING

- GET_PING_FIX

Na Figura 50 é possível ver um diagrama de classes que representa essa *interface*.

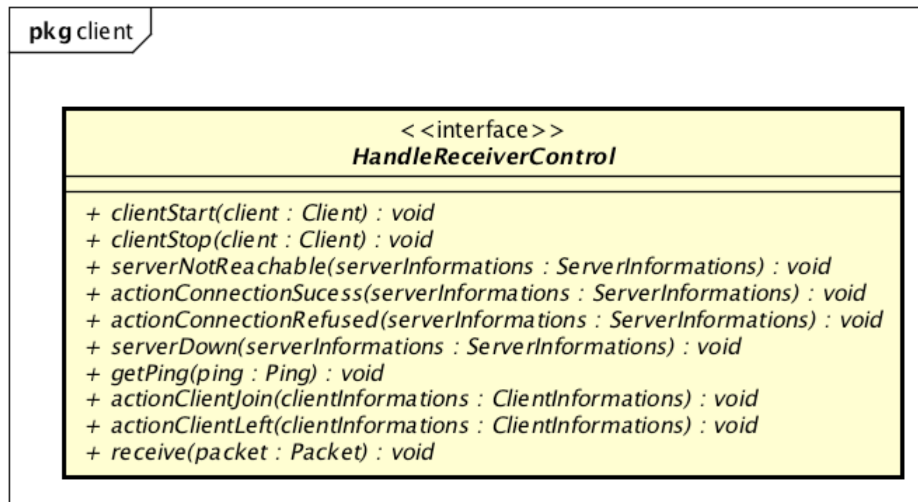

Figura 50: A *interface HandleReceiverControl*

Por ser uma *interface*, não existem atributos, porém os seguintes métodos que estão presentes nessa *interface* e que deverão posteriormente serem implementados pelos desenvolvedores externos são os seguintes:

- *public void clientStart(Client client)* - Esse método tem como função executar algum código desenvolvido pelos desenvolvedores externos quando o cliente inicia.
- *public void clientStop(Client client)* - Funciona da mesma forma que o primeiro método, porém ele é executado quando o cliente for finalizado.
- *public void serverNotReachable(ServerInformations serverInformations)* - Executa os códigos do desenvolvedor externo quando o servidor não for alcançável.
- *public void actionConnectionSucess(ServerInformations serverInformations)* - É executado quando o cliente ter sua conexão aceita com o servidor.
- *public void actionConnectionRefused(ServerInformations serverInformations)* - Esse método será executado quando a conexão do cliente for recusada.
- *public void serverDown(ServerInformations serverInformations)* - Será executado quando o servidor vir a baixo.

- *public void getPing(Ping ping)* - É executado quando a resposta de *ping* do servidor for recebida pelo cliente.
- *public void actionClientJoin(ClientInformations clientInformations)* - É executado quando um cliente conectar-se ao servidor.
- *public void actionClientLeft(ClientInformations clientInformations)* - É executado quando um cliente desconectar-se do servidor.
- *public void receive(Packet packet)* - É executado quando um pacote for recebido pelo cliente.

Lembrando que cada método deve ser implementado pelo desenvolvedor externo, para que seus códigos customizados funcionem.

A classe *Receiver*

É a responsável por receber os pacotes enviados pelo servidor e então deserializá-los, após o processo de deserialização ter ocorrido, o objeto retornado será transformado em um objeto do tipo *Packet* para que então possa ser processado pela classe *ReceiverControl*.

Na Figura 51 é possível ver um diagrama de classes que representa essa classe.

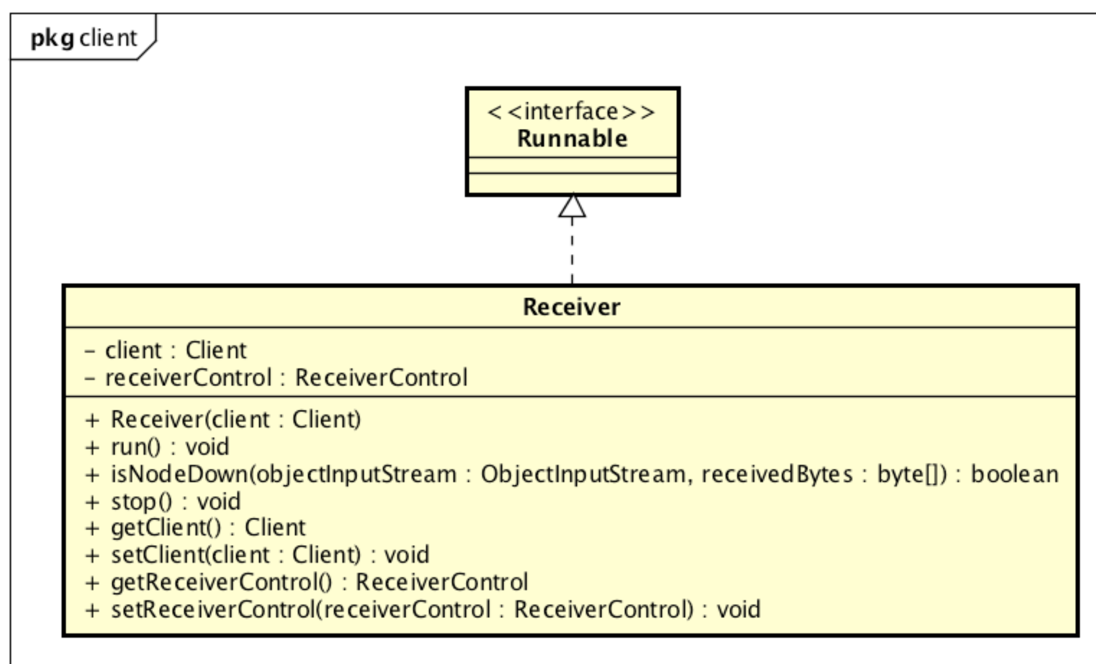

Figura 51: A classe *Receiver*

Os seguintes atributos estão presentes nessa classe:

- *private Client client* - Armazena uma referência para a classe principal do projeto, de forma que se for necessário, é possível acessar todos os atributos, classes e métodos pertencentes ao cliente.
- *private ReceiverControl receiverControl* - É uma referência para a classe responsável por processar os pacotes que são recebidos na classe *Receiver*.

Além dos métodos de *set()* e *get()*, os outros métodos que estão presentes nessa classe são:

- *public Receiver(Client client)* - É o construtor da classe e através do seu parâmetro inicializa o atributo *client*, além disso o atributo *receiverControl* também é instanciado.
- *public void run()* - É o método executado pela sua *thread*, que receberá as mensagens enviadas pelo servidor enquanto o atributo booleano *running* conter um valor lógico verdadeiro, e então repassará essa mensagem para o método *receive()* pertencente ao atributo *receiverControl*, que contém uma instância da classe *ReceiverControl*.
- *public void stop()* - É o responsável por finalizar a *thread* da classe *Receiver* e também alterar o valor lógico da variável booleana *running* de verdadeiro para falso.

A classe *ReceiverControl*

Tem por finalidade receber os pacotes enviados pelo servidor através da classe *Receiver*, e então, posteriormente processá-los.

Na Figura 52 é possível ver um diagrama de classes que representa essa classe.

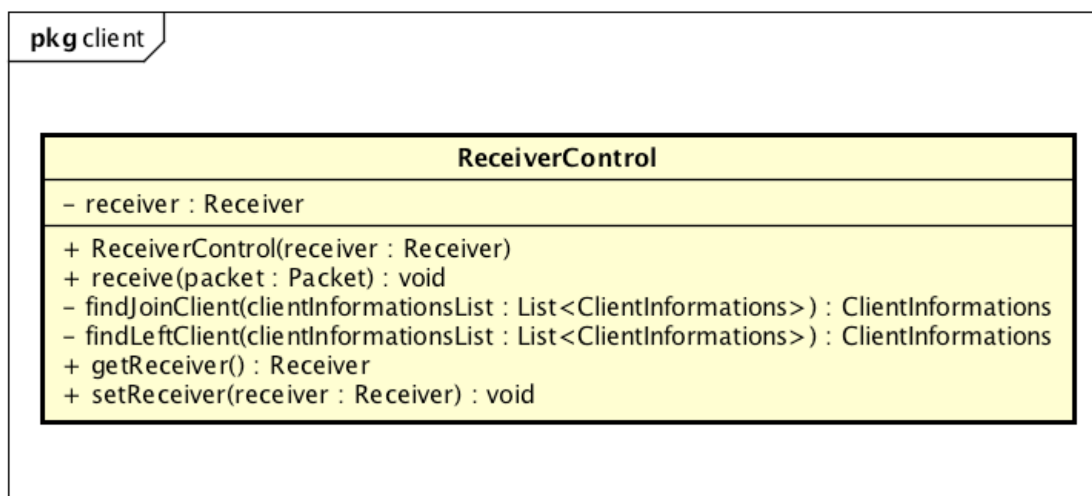

Figura 52: A classe *ReceiverControl*

O seguinte atributo esta presente nessa classe:

- *private Receiver receiver* - Contém uma referência para o objeto *receiver* que esta presente na classe *Client*, ou seja, através desse atributo é possível acessar a classe *Receiver*, e por consequência, também é possível acessar a classe *Client*.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ReceiverControl(Receiver receiver)* - É o construtor da classe, sendo que a referência do objeto *receiver* é passado por parâmetro, dessa forma é possível inicializar o atributo *receiver* que esta presente na classe *ReceiverControl*.
- *public void receive(Packet packet)* - Irá receber um pacote da classe *Receiver* e então irá processá-lo de acordo com a *String* que estiver contida no atributo *header* da classe *Packet*.
- *private ClientInformations findJoinClient(List <ClientInformations> clientInformationsList)* - Irá encontrar o cliente que conectou-se ao servidor, comparando a lista *clientInformationsList* recebida por parâmetro com a lista que esta presente na classe *ServerInformations* do cliente. O cliente que não for encontrado na lista *clientInformationsList* e na lista que esta presente na classe *ServerInformations* do cliente simultaneamente, será o cliente que conectou-se ao servidor.
- *private ClientInformations findLeftClient(List <ClientInformations> clientInformationsList)* - Esse método também irá comparar as duas listas,

porém agora com uma pequena diferença, o cliente que não estiver presente na lista *clientInformationsList* recebida por parâmetro, mas estiver presente na lista da classe *ServerInformations* do cliente, é o cliente que acabou de desconectar-se do servidor.

Classe utilizadas no servidor do protocolo TCP

A classe *Client*

É a abstração de cada cliente que esta conectado ao servidor, sendo que cada cliente deve ter o seu próprio canal de comunicação, que funciona como uma forma de canal de duas vias, ou seja, ele será o responsável por enviar e receber dados.

Na Figura 53 é possível ver um diagrama de classes que representa essa classe.

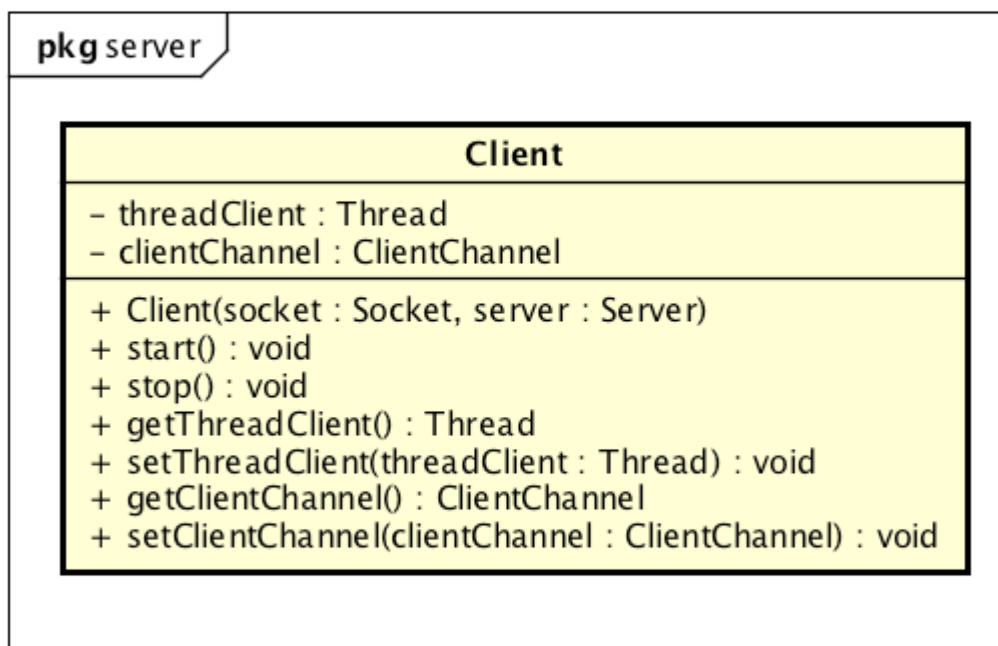

Figura 53: A classe *Client*

De acordo com essa abstração, os seguintes atributos estarão presentes nessa classe:

- *private Thread threadClient* - É a *thread* responsável por manter o canal de comunicação sempre ativo, sendo que cada cliente conectado ao servidor terá a sua própria *thread*.

- *private ClientChannel clientChannel* - É a abstração propriamente dita do canal de comunicação do cliente, onde cada cliente poderá ser gerenciado singularmente pelo servidor.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Client(Socket socket, Server server)* - Método construtor da classe, que através dos seus parâmetros irá instanciar o canal do cliente e também sua *thread*.
- *public void start()* - É quem inicia a execução do canal do cliente através de sua *thread*.
- *public void stop()* - Tem a função de parar a *thread* do canal de comunicação do cliente, fazendo com que o cliente em questão seja desconectado do servidor.

A classe *ClientChannel*

É abstração propriamente dita do canal de comunicação do cliente com o servidor, sendo que o nó representado por essa classe é o nó presente no servidor, ou seja, através desse canal será possível enviar dados para o cliente, e da mesma forma receber dados do cliente.

Na Figura 54 é possível ver um diagrama de classes que representa essa classe.

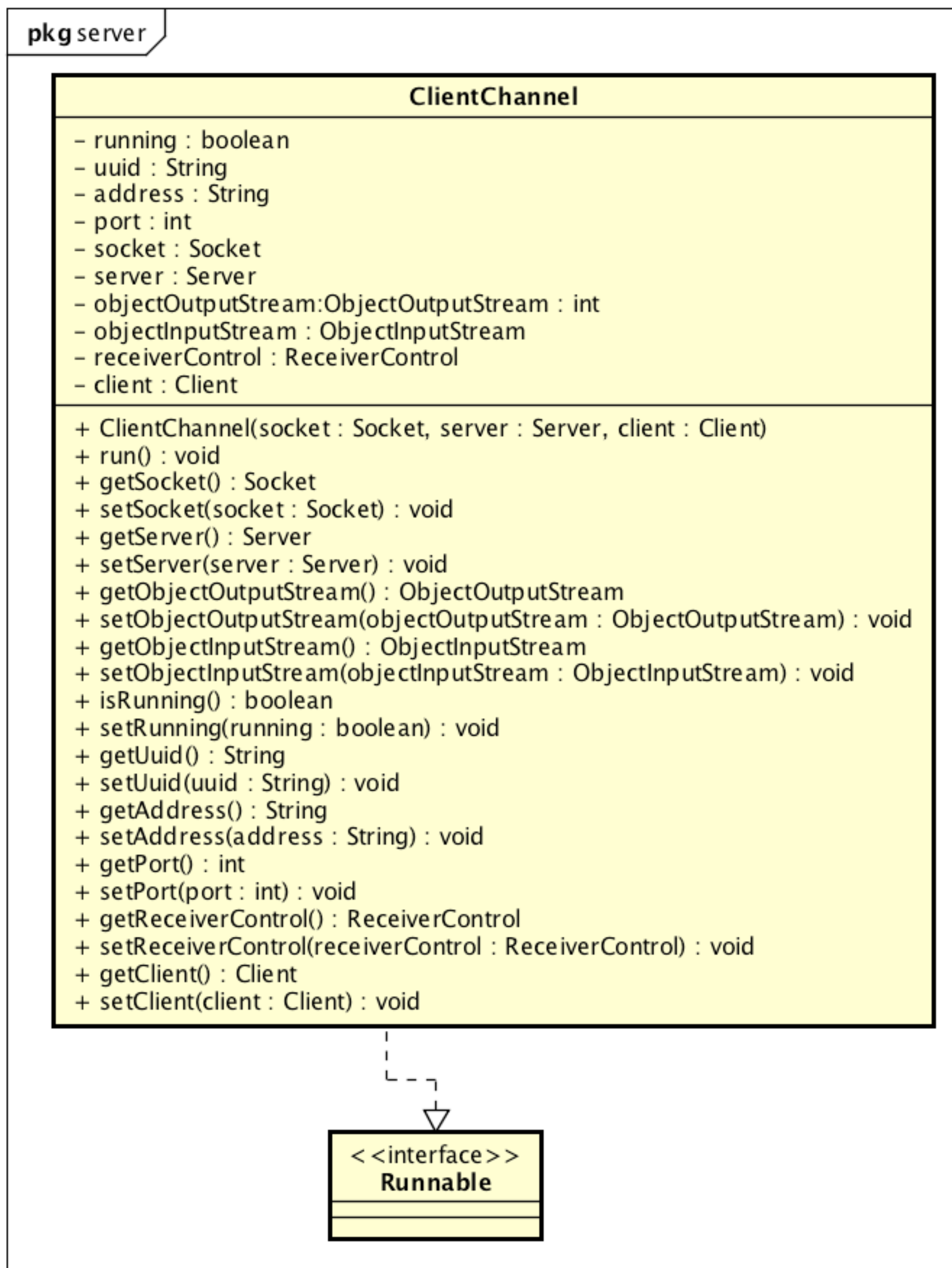

Figura 54: A classe *ClientChannel*

Com base nessa abstração, os seguintes atributos estão presentes nessa classe:

- *private Socket socket* - Esse atributo é inicializado através do parâmetro presente no método construtor, e ele representa a comunicação entre duas máquinas.

- *private Server server* - é a referência para a classe principal do servidor, através dela é possível acessar todos os elementos do servidor.
- *private ObjectOutputStream objectOutputStream* - É o responsável por enviar um pacote serializado para o cliente.
- *private ObjectInputStream objectInputStream* - É o responsável por receber os pacotes serializados do cliente.
- *private boolean running* - Armazena o valor lógico verdadeiro ou falso, para que seja possível a *thread* principal da classe *ClientChannel* manter-se executando.
- *private String uuid* - Armazena a representação textual do identificador único do cliente.
- *private String address* - Armazena o endereço do cliente em uma representação textual.
- *private int port* - Armazena a porta com a qual o cliente esta conectado.
- *private ReceiverControl receiverControl* - Mantém uma referência para a classe responsável por processar os pacotes recebidos do cliente.
- *private Client client* - Mantém uma referência para a classe que representa o cliente conectado com o servidor.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ClientChannel(Socket socket, Server server, Client client)* - É o construtor da classe, responsável por inicializar o atributo *socket* por parâmetro, com o *socket* aceito pela classe principal do servidor. Esse método também manterá uma referência para a classe *Server*, podendo assim, acessar os principais métodos e atributos dessa classe.
- *public void run()* - Método principal da *thread* do canal do cliente, esse método receberá as mensagens do servidor e então as irá passar para a classe *ReceiverControl* para que possam ser posteriormente analisadas e processadas de acordo com o conteúdo do seu *header*. Além disso, esse método verifica quando um cliente desconecta-se do servidor através da exceção *java.io.EOFException*, onde caso uma desconexão seja reconhecida, o cliente desconectado será removido da lista de clientes conectados.

A *interface HandleReceiverControl*

É a responsável por injetar o código customizado dos desenvolvedores externos diretamente no código do JEagle através de métodos já definidos. Essa *interface* precisa ser posteriormente implementada em uma outra classe qualquer, que será escolhida pelo desenvolvedor externo. O desenvolvedor externo pode utilizar qualquer *header* que desejar no método *receive*, contanto que não seja nenhuma das seguintes palavras reservadas abaixo:

- CONNECTION_SUCESS_RESPONSE
- CONNECTION_REFUSED_RESPONSE
- GET_CLIENTS_LIST_RESPONSE
- GET_PING_RESPONSE
- CLIENT_ACTION_JOIN
- CLIENT_ACTION_LEFT
- GET_PING_RESPONSE_FIX
- SERVER_INFORMATIONS_UPDATE
- GET_CLIENTS_LIST
- GET_PING
- GET_PING_FIX

Na Figura 55 é possível ver um diagrama de classes que representa essa *interface*.

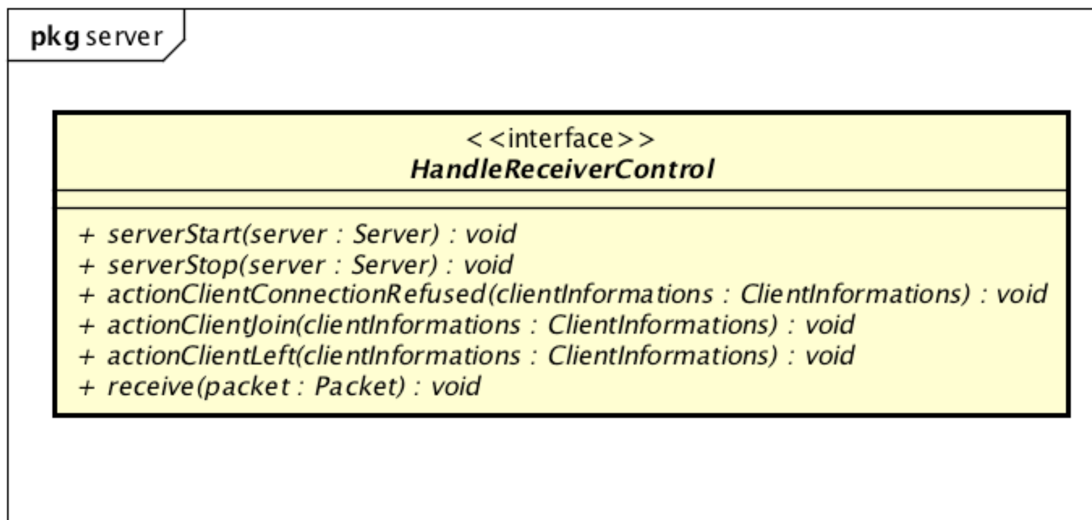

Figura 55: A *interface HandleReceiverControl*

Por ser uma *interface* ela não conterá atributos, porém os seguintes métodos, estarão presentes:

- *public void serverStart(Server server)* - Esse método executará os métodos customizados do desenvolvedor externo assim que o servidor é iniciado.
- *public void serverStop(Server server)* - Executará os códigos do desenvolvedor externo no momento em que o servidor for encerrado.
- *public void actionClientConnectionRefused(ClientInformations clientInformations)* - Irá executar os códigos customizados no momento em que cliente ter sua conexão recusada.
- *public void actionClientJoin(ClientInformations clientInformations)* - Esse método será executado quando um cliente conectar-se ao servidor.
- *public void actionClientLeft(ClientInformations clientInformations)* - Executará os códigos customizados do desenvolvedor externo no momento em que um cliente desconectar-se do servidor.
- *public void receive(Packet packet)* - É utilizado para receber os pacotes customizados criados pelo desenvolvedor externo.

A classe *ReceiverControl*

Recebe os pacotes da classe *ClientChannel*, que previamente foram recebidos do cliente pelo servidor, e então os processa baseando-se em seus *headers*.

Na Figura 56 é possível ver um diagrama de classes que representa essa classe.

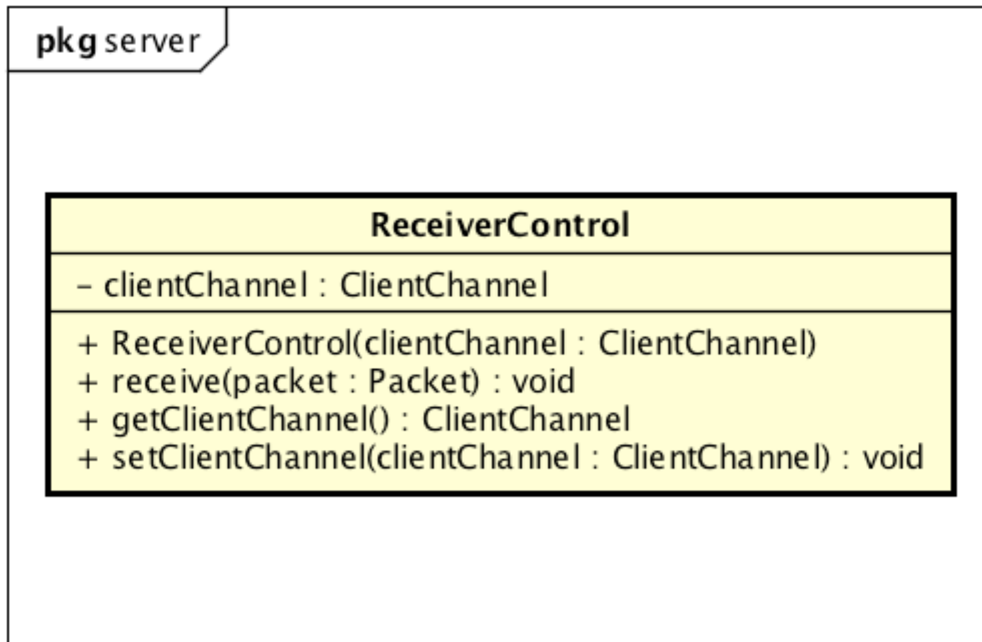

Figura 56: A classe *ReceiverControl*

O seguinte atributo esta presente nessa classe:

- *private ClientChannel clientChannel* - Esse único atributo esta aqui presente para que seja possível acessar todos os atributos do canal do cliente, além disso, acessando o *clientChannel* também é possível acessar a classe principal do servidor, ou seja, a classe *Server*.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ReceiverControl(ClientChannel clientChannel)* - É o construtor da classe, responsável por instanciar a classe *ReceiverControl* e também inicializar o atributo *clientChannel* com a referência passada por parâmetro.
- *public void receive(Packet packet)* - É o método que receberá o pacote que foi recebido na classe *Receiver* e então posteriormente irá processá-lo de acordo com o atributo *header* presente no objeto do pacote.

A classe *Server*

É a classe que representa o servidor, pois é nela que todas as operações do servidor serão executadas.

Na Figura 57 é possível ver um diagrama de classes que representa essa classe.

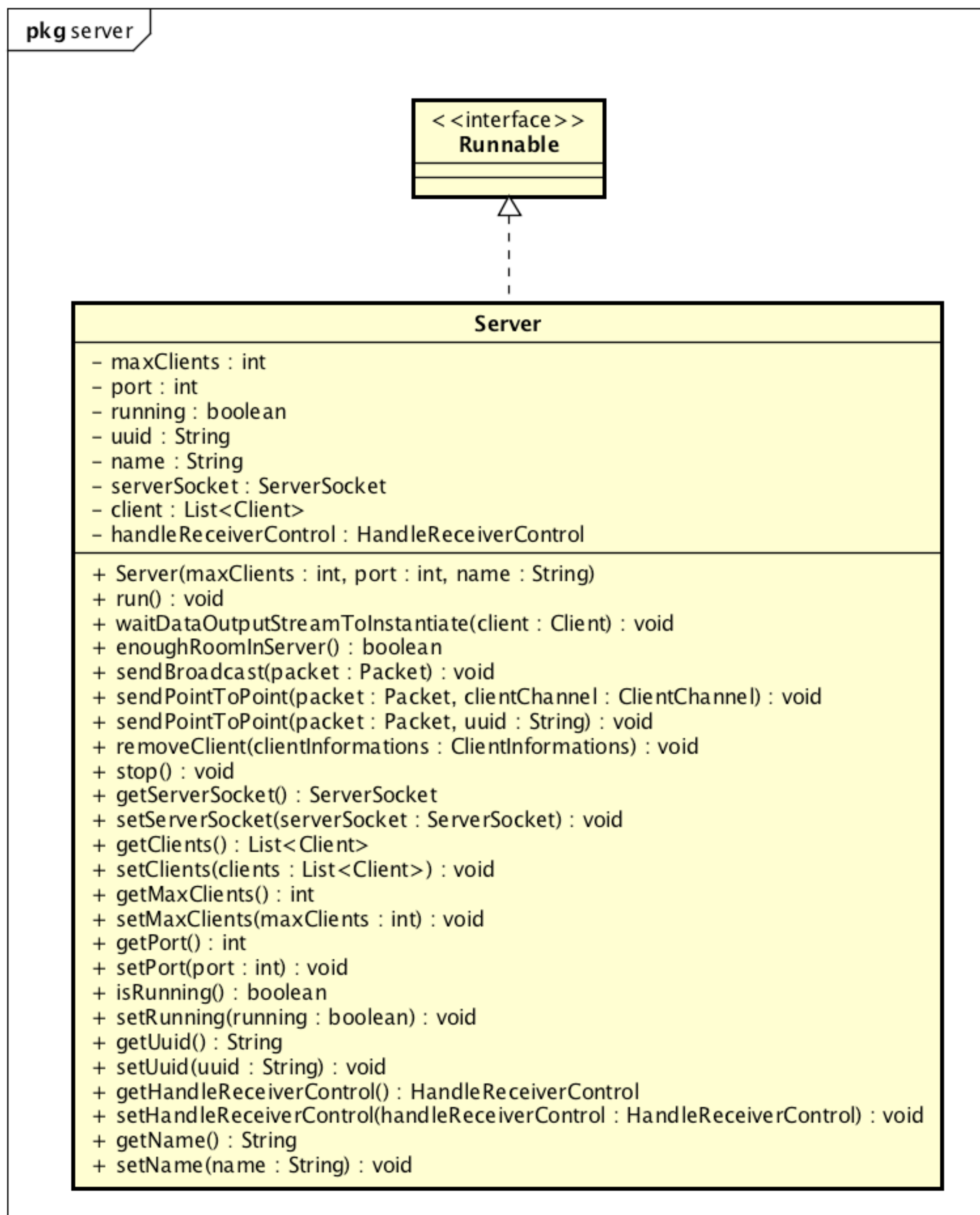

Figura 57: A classe *Server*

Os seguintes atributos estão presentes nessa classe:

- *private ServerSocket serverSocket* - É um atributo que depois de instanciado irá esperar por requisições através da Internet e posteriormente executar operações

baseadas nessas requisições, sendo que possíveis resultados poderão ser enviados ao cliente, caso seja necessário.

- *private List <Client > clients* - Irá conter a lista de todos os clientes conectados ao servidor.
- *private int maxClients* - Armazena a informação de quantos clientes conectados simultaneamente são necessários.
- *private int port* - Armazena a porta utilizada pelo servidor para aceitar as conexões externas.
- *private boolean running* - Mantém a *thread* principal do servidor sendo executada, e consequentemente recebendo conexões.
- *private String uuid* - É quem contém a representação textual do identificador único do servidor.
- *private String name* - Armazena o nome do servidor.
- *private HandleReceiverControl handleReceiverControl* - Irá armazenar a implementação da *interface HandleReceiverControl* para que seja possível executar um código customizado do desenvolvedor externo.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes na classe *Server*:

- *public Server(int maxClients, int port, String name)* - é o construtor da classe *Server*, sendo que nesse construtor, a quantidade máxima de clientes, a porta e o nome do servidor serão inicializados.
- *public void run()* - É onde as operações da *thread* principal dessa classe são executados, ou seja, nessa classe o atributo *serverSocket* será inicializado e enquanto o atributo booleano *running* conter um valor verdadeiro, os clientes terão sua conexão aceita pelo servidor caso a quantidade de clientes presentes na lista de clientes conectados seja menor que o valor da variável *maxClients*. Caso o servidor contenha espaço suficiente para aceitar novas conexões, então o servidor enviará um pacote para todos os clientes conectados informando que um novo cliente conectou-se ao servidor, caso contrário um pacote será enviado ao cliente que tentou estabelecer uma conexão, informando-o que sua conexão foi rejeitada.

- *public void waitDataOutputStreamToInstantiate(Client client)* - Faz com que as operações do servidor esperem até que o atributo *ObjectOutputStream* do cliente já esteja inicializado.
- *public boolean enoughRoomInServer()* - Verifica se existe espaço suficiente no servidor, verificando se a quantidade de clientes presentes na lista de clientes conectados é menor que o valor contido na variável *maxClients*, em caso afirmativo um valor verdadeiro será envaido, caso contrário um valor de falso será retornado.
- *public void sendBroadcast(Packet packet)* - Enviará um pacote para todos os clientes conectados ao servidor.
- *public void sendPointToPoint(Packet packet, ClientChannel clientChannel)* - Enviará um pacote para um canal de conexão especifico.
- *public void sendPointToPoint(Packet packet, String uuid)* - Enviará um pacote para um cliente que contém o identificador único igual ao parâmetro *uuid* do tipo *String*.
- *public void removeClient(ClientInformations clientInformations)* - Irá remover um cliente da lista de clientes conectados que contém as mesmas informações presentes no parâmetro *clientInformations*.
- *public void stop()* - É quem irá parar a *thread* principal do servidor, fazendo de forma, com que o servidor seja parado.

Classes utilizadas no cliente do protocolo UDP

A classe *Client*

Na Figura 58 é possível ver um diagrama de classes que representa essa classe.

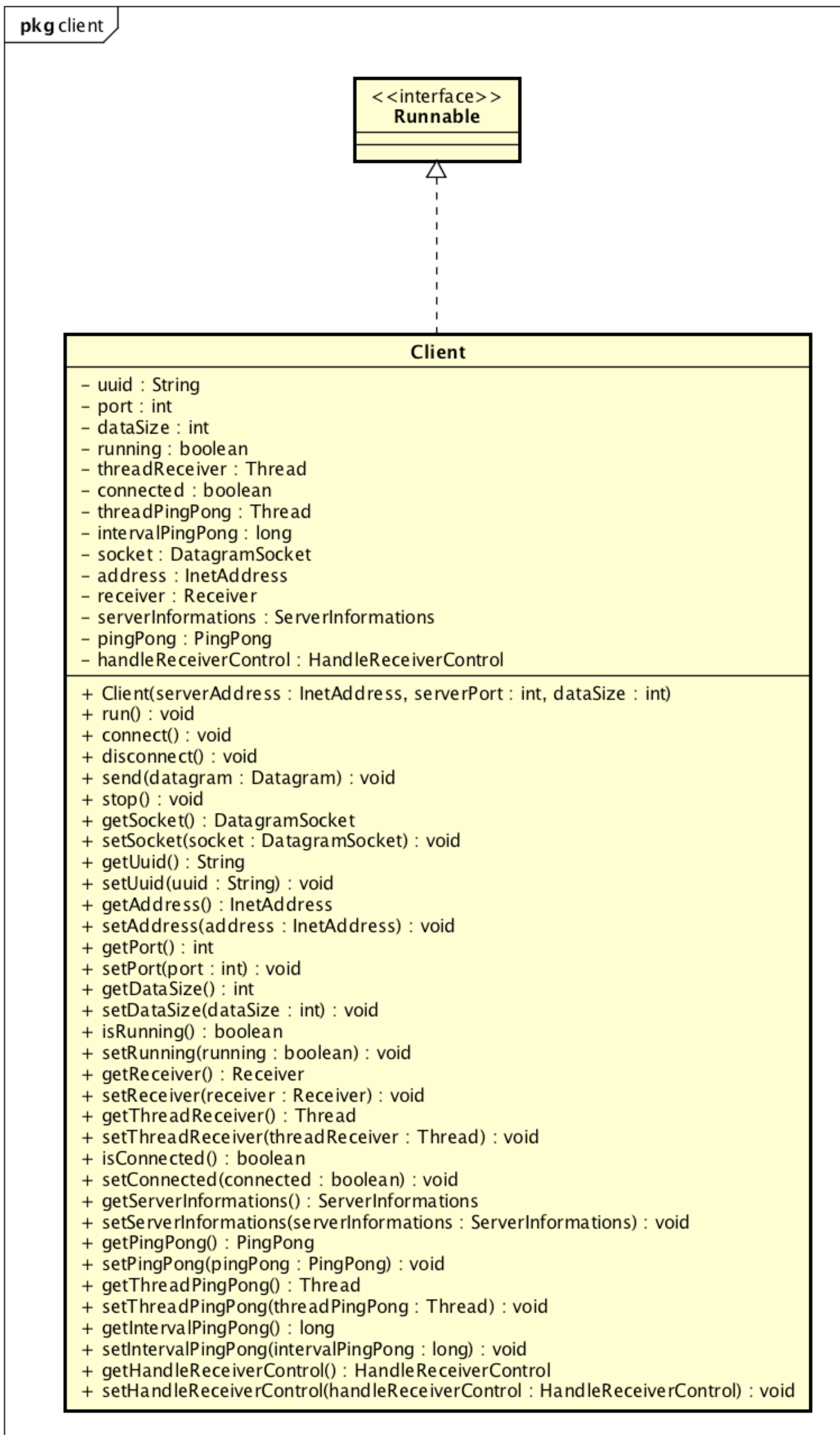

Figura 58: A classe *Client*

Será a classe que representará o cliente conectado ao servidor, logo essa classe conterá os seguintes atributos:

- *private DatagramSocket socket* - Será o responsável por enviar e receber datagramas.
- *private String uuid* - Irá armazenar o identificador único do cliente em uma representação textual, lembrando que o *uuid* do cliente é recebido do servidor assim que sua conexão for aceita.
- *private InetAddress address* - É quem armazena o endereço do cliente que esta conectado ao servidor.
- *private int port* - Irá armazenar a sua porta, responsável por receber os datagramas do servidor.
- *private int dataSize* - É quem define o tamanho do vetor de *bytes* que será enviado e recebido, ou seja, o tamanho do datagrama será definido assim que a classe do cliente for instânciada.
- *private boolean running* - Manterá a *thread* principal do cliente sendo executada, e consequentemente manterá o cliente sendo executado.
- *private Receiver receiver* - É a referência do objeto que posteriormente irá escolher qual operações executar baseando-se no *header* dos datagramas que foram recebidos do servidor.
- *private Thread threadReceiver* - irá armazenar a *thread* principal da classe *Receiver*.
- *private boolean connected* - Irá informar se o cliente esta conectado ao servidor ou não, através de um valor verdadeiro ou falso.
- *private ServerInformations serverInformations* - Armazenará as informações do servidor que o cliente esta conectado, como por exemplo, endereço, porta e lista de clientes conectados.
- *private PingPong pingPong* - Irá enviar e receber pacotes em um período de tempo determinado para verificar se o servidor ainda é alcançável.
- *private Thread threadPingPong* - É quem armazenará a *thread* principal do atributo *pingPong*, fazendo com que ele seja executado indefinidamente.
- *private long intervalPingPong* - Irá armazenar o tempo em milissegundos de quando o atributo *pingPong* deve verificar se o servidor ainda é alcançável.

- *private HandleReceiverControl handleReceiverControl* - É o responsável por injetar o código dos desenvolvedores externos no código do JEagle.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes na classe *Client*:

- *public Client(InetAddress serverAddress, int serverPort, int dataSize)* - É o construtor da classe *Client*. Através desse construtor, será possível instanciar o atributo *serverInformations* com o endereço do servidor e também a sua porta, para que seja possível estabelecer uma conexão mais tarde, além disso, o parâmetro *dataSize* irá inicializar o atributo *dataSize* ditando o tamanho máximo que um datagrama poderá ter.
- *public void run()* - É quem irá instanciar o atributo *socket*, a classe *Receiver* e também a classe *PingPong*, além de também manter o cliente sendo executado enquanto o atributo booleano *running* conter um valor verdadeiro.
- *public void connect()* - Irá enviar um datagrama para o servidor pedindo para poder conectar-se através do *header* "CONNECTION_ATTEMPT", é necessário enviar esse datagrama, pois não existe o conceito de conexão com a utilização do protocolo UDP.
- *public void disconnect()* - Irá alterar o valor booleano do atributo *connected* para falso.
- *public void send(Datagram datagram)* - Envia um datagrama para o servidor.
- *public void stop()* - Irá alterar o valor booleano do atributo *running* para falso, e também irá parar a *thread* principal do cliente, fazendo com que ele seja desconectado do servidor.

A *interface HandleReceiverControl*

É quem possibilitará a injeção de códigos dos desenvolvedores externos no código do JEagle. O desenvolvedor externo pode utilizar qualquer *header* que desejar no método *receive*, contanto que não seja nenhuma das seguintes palavras reservadas abaixo:

- CONNECTION_ACCEPT
- PING_FIX_RESPONSE

- GET_PING_RESPONSE
- CONNECTION_ACCEPT_END_STEP_RESPONSE
- CONNECTION_REFUSED
- DISCONNECTION_ACCEPT
- CLIENT_JOIN
- CLIENTS_LIST_REQUEST
- CLIENT_LEFT
- CLIENTS_LIST_REQUEST
- CLIENTS_LIST_REQUEST_RESPONSE
- CONNECTION_ATTEMPT
- PING_FIX
- GET_PING
- CONNECTION_ACCEPT_END_STEP
- DISCONNECTION_ATTEMPT
- CLIENTS_LIST_REQUEST
- SERVER_REACHABLE

Na Figura 59 é possível ver um diagrama de classes que representa essa *interface*.

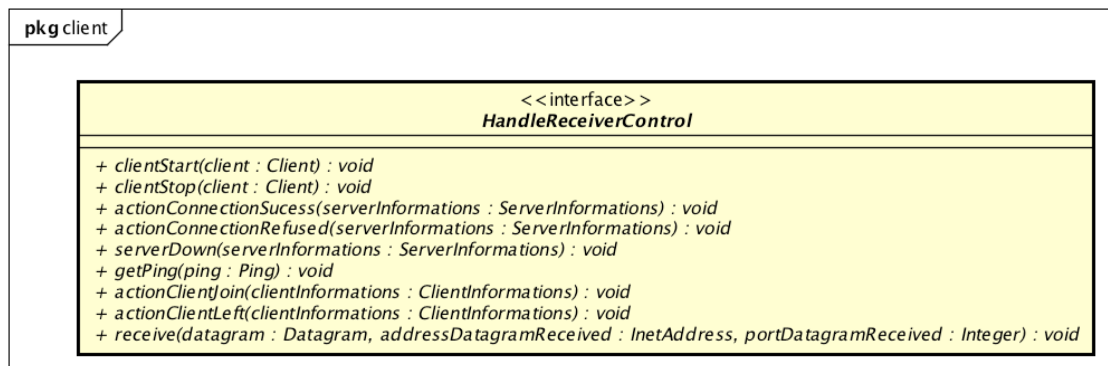

Figura 59: A *interface HandleReceiverControl*

Por ser uma *interface* ela não conterá atributos, mas os seguintes métodos deverão ser implementados posteriormente pelos desenvolvedores externos:

- *public void clientStart(Client client)* - Será executado no momento em que o cliente for iniciado.

- *public void clientStop(Client client)* - Assim como o método anterior, será executado quando o cliente for encerrado.
- *public void actionConnectionSucess(ServerInformations serverInformations)* - Será executado quando a conexão do cliente for aceita pelo servidor.
- *public void actionConnectionRefused(ServerInformations serverInformations)* - Será executado quando a conexão do cliente for rejeitada pelo servidor, a rejeição irá ocorrer quando não existir mais espaço suficiente no servidor para aceitar novas conexões.
- *public void serverDown(ServerInformations serverInformations)* - Será executado quando o servidor vir a baixo.
- *public void getPing(Ping ping)* - Será executado quando uma resposta de ping for recebida do servidor.
- *public void actionClientJoin(ClientInformations clientInformations)* - Esse método será executado no momento em que um cliente conectar-se ao servidor.
- *public void actionClientLeft(ClientInformations clientInformations)* - Será executado no momento em que o cliente desconectar-se do servidor.
- *public void receive(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - Será executado quando um datagrama customizado for recebido do servidor.

A classe *PingPong*

É uma abstração do esquema *ping pong* do lado do cliente, ou seja, sua função será enviar datagramas contendo os *headers* "PING" e "PONG", dependendo da requisição do servidor em intervalos de tempo definidos.

Na Figura 60 é possível ver um diagrama de classes que representa essa classe.

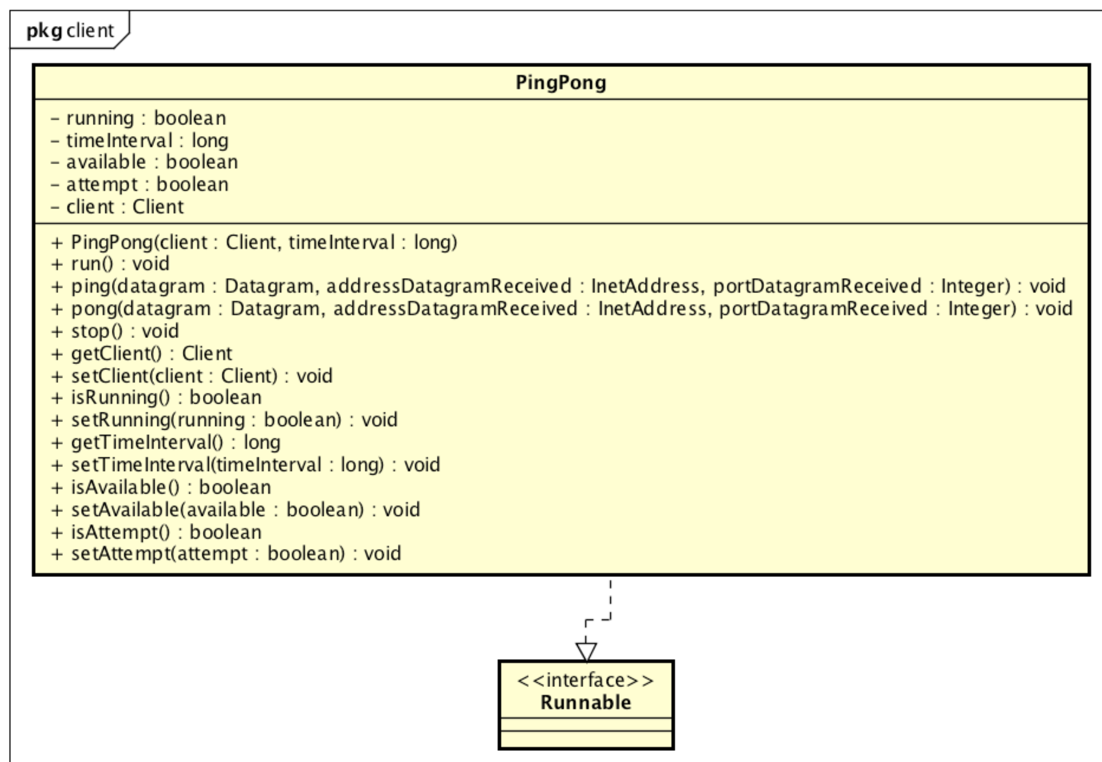

Figura 60: A classe *PingPong*

Os seguintes atributos estão presentes nessa classe:

- *private Client client* - É quem manter armazenada uma referencia para a classe principal *Client*, ou seja, através dela é possível acessar todos os atributos e métodos presentes no projeto do cliente.
- *private boolean running* - Dirá a *thread* principal da classe *PingPong* se ela deve continuar sendo executada ou não.
- *private long timeInterval* - Armazenará o tempo necessário que a classe *PingPong* deve esperar para analisar se uma resposta de *pong* do servidor foi recebida.
- *private boolean available* - Informa se o servidor é alcançável ou não.
- *private boolean attempt* - Irá dizer a classe *PingPong* que a primeira tentativa de receber um *pong* do servidor não foi executada com sucesso, portanto é necessário alterar seu valor para verdadeiro e tentar novamente receber uma resposta de *pong* do servidor, caso a resposta não seja recebida, o servidor não é alcançável.

Além dos métodos de *set()* e *get()*, os seguintes métodos estão presentes nessa classe:

- *public PingPong(Client client, long timeInterval)* - É o construtor da classe, sendo que ele irá inicializar a referência do cliente e também a quantidade de tempo que ela deverá esperar para checar se uma resposta do servidor foi recebida.
- *public void run()* - É o método principal da *thread* da classe *PingPong*, sendo que esse método será o responsável por continuadamente checar se uma resposta de pong do servidor foi recebida, caso uma resposta seja recebida o atributo *available* receberá um valor verdadeiro, caso contrário o atributo *attempt* receberá o valor verdadeiro e o servidor terá mais uma chance para enviar uma resposta de *pong*, se ainda sim o servidor não enviar nenhuma resposta, então o cliente será parado, pois o servidor não é mais alcançável.
- *public void ping(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - É quem verificará se um datagrama de *ping* foi recebido do servidor, ou seja, uma requisição do servidor, perguntando ao cliente se ele ainda esta conectado, em caso afirmativo, o cliente em questão deve enviar uma mensagem de *pong*.
- *public void pong(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - É quem verificará se o servidor enviou uma resposta de *pong*, em caso afirmativo o atributo booleano *available* receberá um valor verdadeiro, fazendo com que o cliente compreenda que o servidor é alcançável.
- *public void stop()* - Possui a finalidade de parar a *thread* principal da classe *PingPong* fazendo com que o cliente não continue mais verificando se o servidor ainda é alcançável.

A classe *Receiver*

É a responsável por receber os datagramas que foram enviados pelo servidor, e posteriormente repassa-los para a classe *ReceiverControl*.

Na Figura 61 é possível ver um diagrama de classes que representa essa classe.

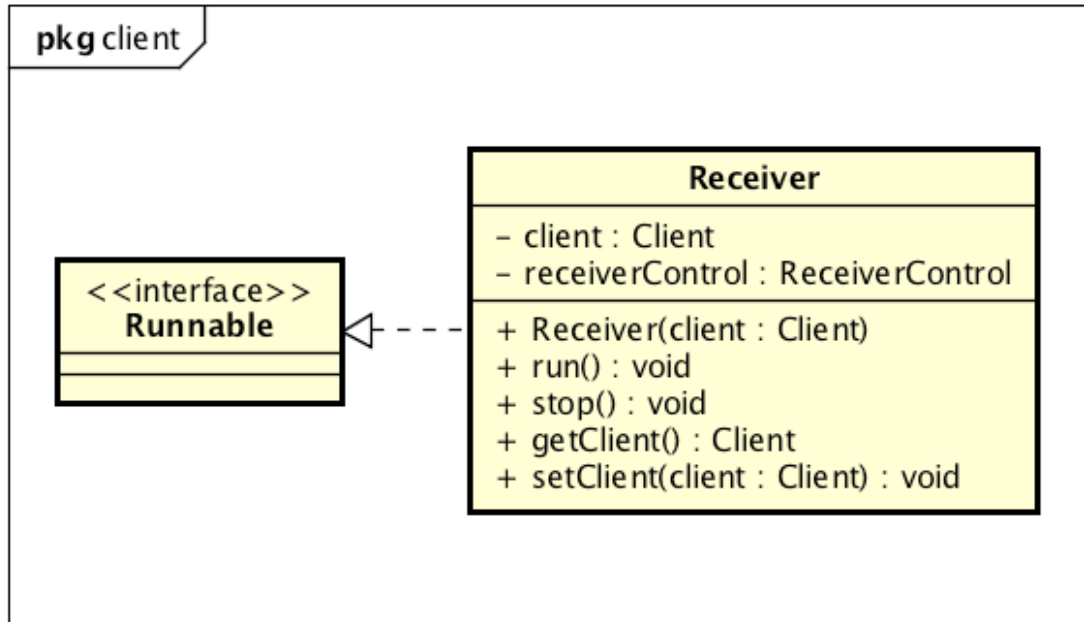

Figura 61: A classe *Receiver*

Os seguintes atributos estão presentes nessa classe:

- *private Client client* - É quem manterá uma referência para a classe principal do projeto do cliente, sendo possível então, dessa forma, acessar todos os atributos e métodos principais do cliente.
- *private ReceiverControl receiverControl* - Conterá uma referência de uma classe do tipo *ReceiverControl*, para que posteriormente os datagramas recebidos sejam processados.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes na classe *Receiver*:

- *public Receiver(Client client)* - É o construtor da classe, sendo que nesse construtor a referência para a classe principal do projeto do cliente é adicionada, além de também instanciar um novo objeto do tipo *ReceiverControl*.
- *public void run()* - É o método principal da *thread* da classe *ReceiverControl*, sendo que nesse método os datagramas serão recebidos, deserializados e checados se alguma mensagem de *pong* ou *ping* foi recebida, além dos datagramas também serem repassados para a classe *ReceiverControl*, onde mais tardes serão processados.

- *public void stop()* - Possui a finalidade de parar a *thread* principal da classe *Receiver*.

A classe *ReceiverControl*

Será a responsável por receber os datagramas do servidor e então processá-los de acordo com o conteúdo dos seus *headers*.

Na Figura 62 é possível ver um diagrama de classes que representa essa classe.

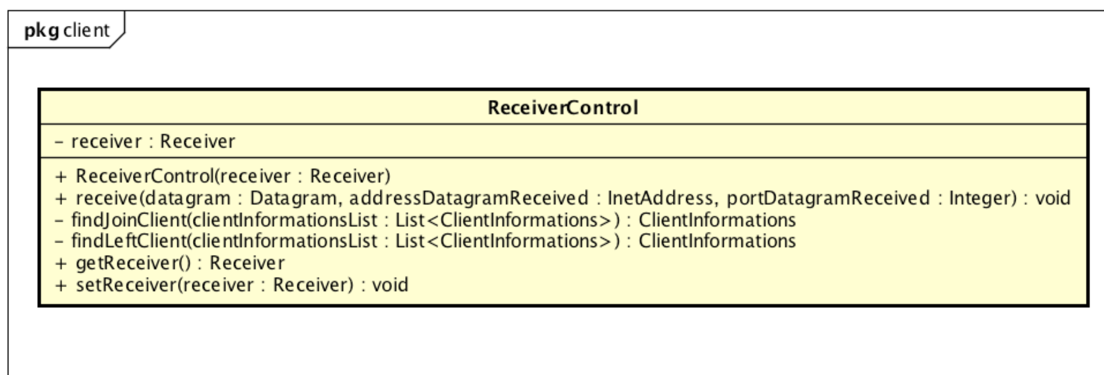

Figura 62: A classe *ReceiverControl*

O seguinte atributo esta presente nessa classe:

- *private Receiver receiver* - É uma referência para a classe *Receiver* pertencente a classe *Client*, sendo que através dela é possível acessar os principais atributos e métodos da classe *Client*.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes na classe *ReceiverControl*:

- *public ReceiverControl(Receiver receiver)* - É o construtor da classe, que irá adicionar a referência recebida pelo parâmetro do método para o atributo *receiver*.
- *public void receive(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - É quem irá receber os datagramas do servidor e então processa-los de acordo com seu *header*.

- *private ClientInformations findJoinClient(List <ClientInformations> clientInformationsList)* - É quem verificará se algum cliente conectou-se ao servidor, verificando se algum cliente pertencente a lista de clientes conectados recebida por parâmetro não esta presente na lista de clientes conectados no cliente, se algum cliente for diferente, então ele é o novo cliente conectado ao servidor.
- *private ClientInformations findLeftClient(List <ClientInformations> clientInformationsList)* - Funciona de maneira similar ao método acima, porém com a diferença que agora procura-se pelo cliente que desconectou-se do servidor, ou seja, se algum cliente pertencente a lista de clientes conectados do clientes não estiver presente na lista de clientes passada por parâmetro, então significa que esse cliente desconectou-se do servidor.

Classes utilizadas no servidor do protocolo UDP

A classe *Client*

Na Figura 63 é possível ver um diagrama de classes que representa essa classe.

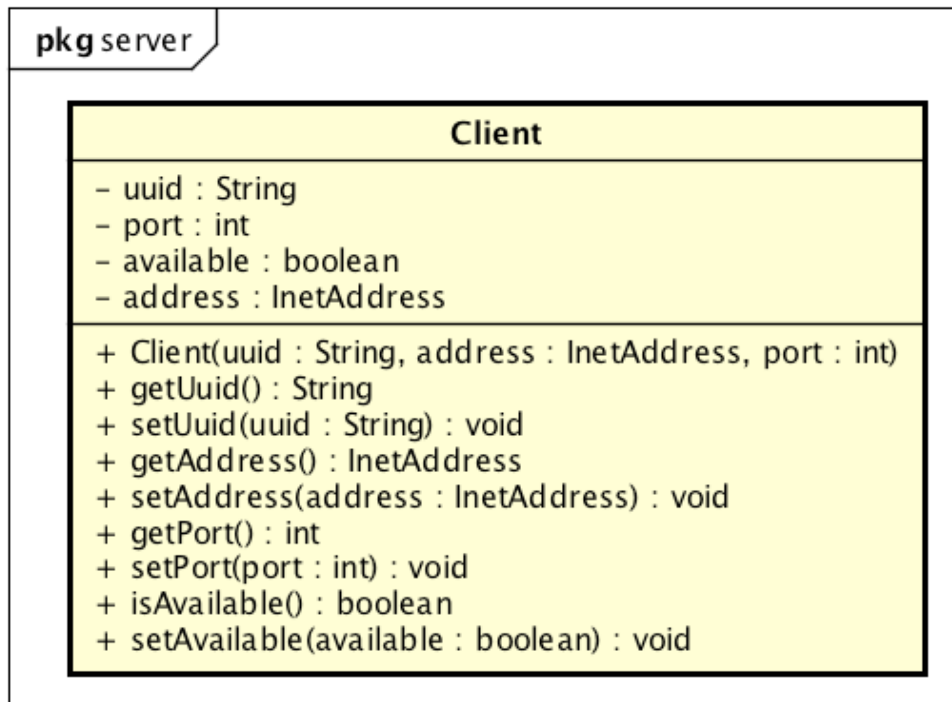

Figura 63: A classe *Client*

É a responsável por representar um determinado cliente que esta conectado ao servidor, portanto os seguintes atributos estão presentes nessa classe:

- *private String uuid* - É a representação textual do identificador único do cliente.
- *private InetAddress address* - Armazena a representação de um endereço que utiliza o protocolo IP.
- *private int port* - É quem armazena a porta do cliente utilizada para a comunicação com o servidor.
- *private boolean available* - É a variável de controle da classe *PingPong*, para saber se um determinado cliente ainda esta conectado ao servidor.

Além dos métodos de *set()* e *get()*, o seguinte método também esta presente nessa classe:

- *public Client(String uuid, InetAddress address, int port)* - É o construtor da classe, que irá inicializar o identificador único do cliente, seu endereço e também sua porta.

A *interface HandleReceiverControl*

Terá a finalidade de injetar os códigos customizados dos desenvolvedores externos no código do JEagle. O desenvolvedor externo pode utilizar qualquer header que desejar no método receive, contanto que não seja nenhuma das seguintes palavras reservadas abaixo:

- CONNECTION_ACCEPT
- PING_FIX_RESPONSE
- GET_PING_RESPONSE
- CONNECTION_ACCEPT_END_STEP_RESPONSE
- CONNECTION_REFUSED
- DISCONNECTION_ACCEPT
- CLIENT_JOIN
- CLIENTS_LIST_REQUEST
- CLIENT_LEFT
- CLIENTS_LIST_REQUEST

- CLIENTS_LIST_REQUEST_RESPONSE
- CONNECTION_ATTEMPT
- PING_FIX
- GET_PING
- CONNECTION_ACCEPT_END_STEP
- DISCONNECTION_ATTEMPT
- CLIENTS_LIST_REQUEST
- SERVER_REACHABLE

Na Figura 64 é possível ver um diagrama de classes que representa essa *interface*.

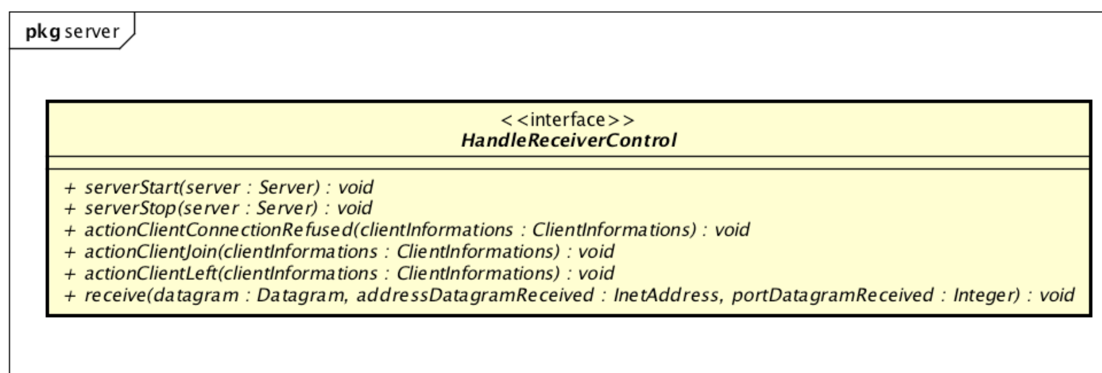

Figura 64: A *interface HandleReceiverControl*

Como a *HandleReceiverControl* é uma *interface* ela não conterá atributos, porém os seguintes métodos deverão posteriormente serem implementados em uma classe qualquer escolhida pelo desenvolvedor externo:

- *public void serverStart(Server server)* - Será executado assim que o servidor for iniciado.
- *public void serverStop(Server server)* - Assim como o primeiro método, esse será executado assim que o servidor for encerrado.
- *public void actionClientConnectionRefused(ClientInformations clientInformations)* - Esse método será executado quando a tentativa de conexão de um determinado cliente com o servidor for rejeitada, a rejeição ocorre quando não existir mais espaço suficiente para novos clientes conectarem-se ao servidor.
- *public void actionClientJoin(ClientInformations clientInformations)* - Será executado no momento em que um novo cliente conectar-se ao servidor.

- *public void actionClientLeft(ClientInformations clientInformations)* - Assim como o método anterior, será executado quando um cliente desconectar-se do servidor, lembrando que as informações do cliente podem ser acessadas através do parâmetro *clientInformations*.
- *public void receive(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - Esse método será executado no momento em que o cliente receber uma mensagem do servidor.

A classe *PingPong*

É a representação do esquema *ping pong* do lado do servidor, ou seja, que verifica de tempos em tempos se os clientes ainda estão conectados ao servidor, enviando datagramas de *ping* e esperando por respostas de *pong*.

Na Figura 65 é possível ver um diagrama de classes que representa essa classe.

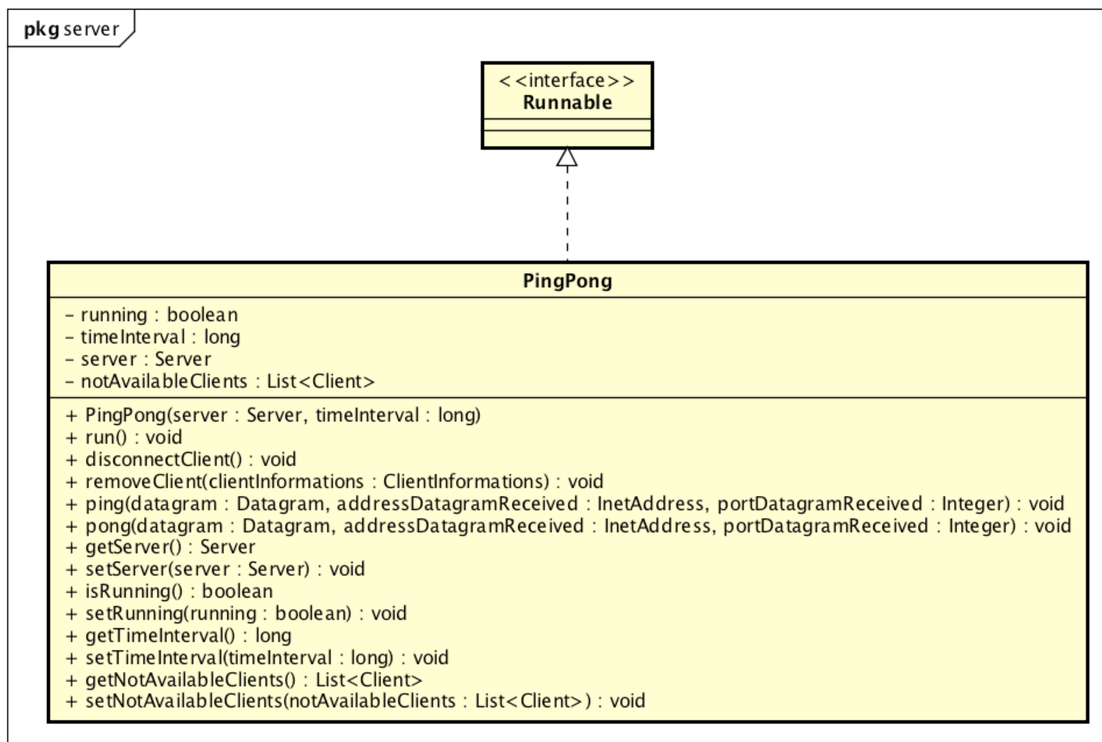

Figura 65: A classe *PingPong*

Os seguintes atributos estão presentes na classe *PingPong*:

- *private Server server* - Irá armazenar uma referência para a principal classe do projeto do servidor, sendo que então será possível acessar os seus principais atributos e métodos.
- *private boolean running* - Fará com que a *thread* principal da classe *PingPong* continue sendo executada enquanto esse atributo conter um valor verdadeiro.
- *private long timeInterval* - Armazena a quantidade de tempo necessária em milissegundos para que a verificação se os clientes ainda estão conectados ao servidor aconteça.
- *private List<Client> notAvailableClients* - Armazenar os clientes que não estão mais conectados ao servidor e posteriormente devem ser removidos da lista de clientes conectados, fazendo com que, dessa forma, eles sejam desconectados.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public PingPong(Server server, long timeInterval)* - É o método construtor da classe, sendo que a referência para a classe *server* será adicionada e o intervalo de tempo para que cada verificação se os clientes ainda estão conectados aconteça também será definido.
- *public void run()* - É principal método da *thread* principal da classe *PingPong*, nesse método será enviado um datagrama de *ping* para os clientes, e então um tempo será esperado, para verificar se os clientes enviaram uma resposta de *pong*, essa verificação acontecerá no próximo método.
- *public void disconnectClient()* - Irá verificar se uma resposta de *pong* foi recebida do cliente, em caso afirmativo o cliente mantem-se conectado, caso contrário, o cliente será adicionado a lista *notAvailableClients* para que uma segunda tentativa seja executada, ou seja, o servidor enviará novamente um datagrama de *ping* para esperar uma resposta de *pong*, se mesmo assim o servidor ainda não receber uma resposta de *pong*, então o cliente desconectou-se.
- *public void removeClient(ClientInformations clientInformations)* - Removerá um cliente específico da lista de clientes conectados presente na classe *Server*, e então irá enviar um datagrama em *broadcast* para todos os clientes conectados, os avisando que o cliente em questão desconectou-se do servidor.

- *public void ping(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - Verificará se os clientes enviaram um datagrama de *ping*, e então o servidor deve enviar um datagrama de *pong*, avisando, dessa forma, que o servidor ainda é alcançável.
- *public void pong(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - Verifica se um cliente enviou um datagrama de *pong*, ou seja, uma resposta, de que ele ainda esta conectado ao servidor. Quando essa resposta chegar, o servidor deve alterar o valor do atributo booleano *available* do respectivo cliente para verdadeiro, para que assim seja possível saber que o cliente ainda esta conectado ao servidor.

A classe *ReceiverControl*

É a responsável por receber os datagramas dos clientes e então processá-los de acordo com o conteúdo presente em seu atributo *header*.

Na Figura 66 é possível ver um diagrama de classes que representa essa classe.

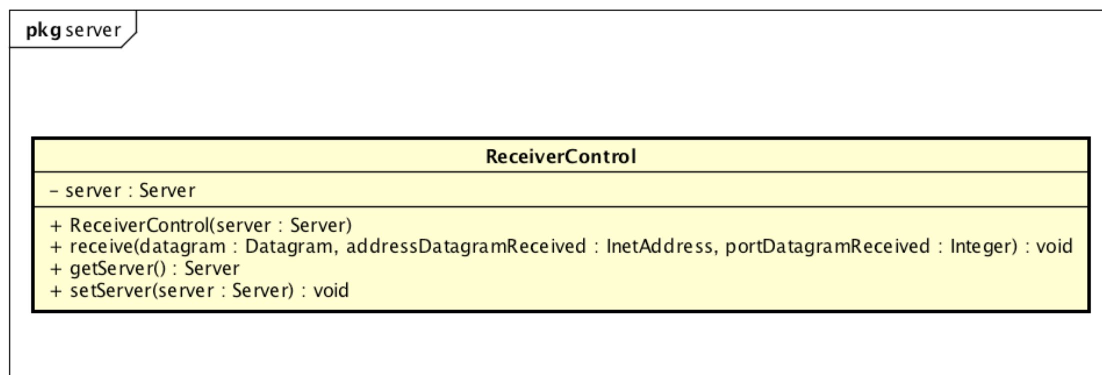

Figura 66: A classe *ReceiverControl*

O seguinte atributo esta presenta nessa classe:

- *private Server server* - Uma referência para a classe principal do servidor, ou seja, ela esta presente nessa classe para que seja possível acessar os principais atributos e métodos do projeto do servidor.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ReceiverControl(Server server)* - É o construtor da classe, e ele irá adicionar uma referência para a classe *Server*, para que seja então possível, acessar os principais atributos e métodos do projeto do servidor.
- *public void receive(Datagram datagram, InetAddress addressDatagramReceived, Integer portDatagramReceived)* - É o responsável por receber os datagramas dos clientes e então processá-los de acordo com o valor do seu atributo *header*.

A classe *Server*

Na Figura 67 é possível ver um diagrama de classes que representa essa classe.

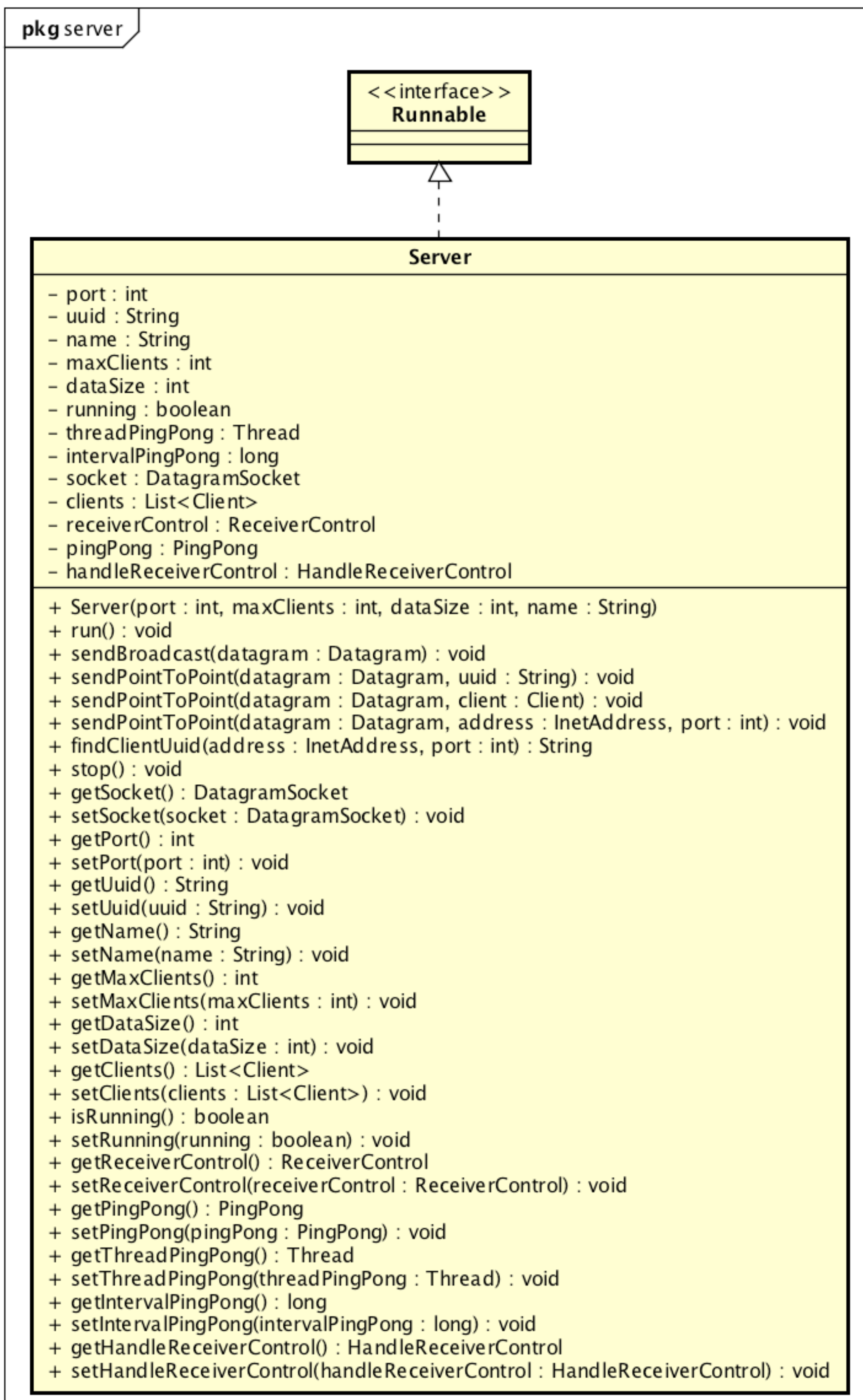

Figura 67: A classe *Server*

É a principal classe do servidor, pois será ela quem irá abstrair a ideia do servidor, ou seja, de aceitar a conexão dos clientes e receber os datagramas, portanto, essa classe contém os seguintes atributos:

- *private DatagramSocket socket* - Será o responsável por enviar e receber datagramas dos clientes.
- *private int port* - Armazenará a porta utilizada pelo servidor para estabelecer a conexão com os clientes.
- *private String uuid* - Será a representação textual do identificador único do servidor.
- *private String name* - Armazena o nome do servidor.
- *private int maxClients* - Armazena a quantidade máxima de clientes que podem conectar-se simultaneamente ao servidor.
- *private int dataSize* - Irá armazenar o tamanho máximo do vetor de *bytes* que um datagrama pode ter.
- *private List<Client> clients* - Armazena uma lista dos clientes que estão atualmente conectados ao servidor.
- *private boolean running* - Mantém a *thread* principal da classe *Server* sendo executada, enquanto ele conter um valor verdadeiro.
- *private ReceiverControl receiverControl* - É o responsável por armazenar uma referência para a classe *ReceiverControl*, onde posteriormente os datagramas recebidos pelos clientes serão processados de acordo com o valor do seu *header*.
- *private PingPong pingPong* - Manterá uma referência para a classe *PingPong* e posteriormente verificará de tempos em tempos, se os clientes ainda estão conectados ao servidor, enviando uma mensagem de *ping* e esperando por uma resposta de *pong*.
- *private Thread threadPingPong* - É uma referência para a *thread* principal que será utilizada pela classe *PingPong*.
- *private long intervalPingPong* - Armazenará o intervalo de tempo em que o esquema de *ping pong* será executado pela classe *PingPong*.
- *private HandleReceiverControl handleReceiverControl* - Será utilizado para injetar os códigos customizados dos desenvolvedores externos no código do servidor presente no JEagle.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Server(int port, int maxClients, int dataSize, String name)* - É o construtor da classe, e nele serão inicializados os atributos relativos a classe *Server*.
- *public void run()* - É o principal método da *thread* principal da classe *Server*, e nesse método os datagramas serão recebidos dos clientes e então eles passaram por uma verificação do esquema *ping pong* e posteriormente os datagramas serão repassados para a classe *ReceiverControl* onde serão processados de acordo com o atributo *header* presente nos datagramas.
- *public void sendBroadcast(Datagram datagram)* - Enviará um datagrama para todos os clientes conectados ao servidor.
- *public void sendPointToPoint(Datagram datagram, String uuid)* - Enviará uma mensagem para um cliente específico que conter o identificador único igual ao parâmetro *uuid*.
- *public void sendPointToPoint(Datagram datagram, Client client)* - Enviará uma mensagem para um cliente específico que for igual ao parâmetro *client*.
- *public void sendPointToPoint(Datagram datagram, InetAddress address, int port)* - Enviará uma mensagem para um cliente que conter seu endereço e porta igual aos parâmetros *address* e *port* respectivamente.
- *public String findClientUuid(InetAddress address, int port)* - Procura o identificador único de um cliente na lista de clientes conectados que conter o mesmo endereço e porta que esta respectivamente presente nos parâmetros *address* e *port*.
- *public void stop()* - Será o responsável por parar a execução da *thread* principal do servidor, e consequentemente parar a execução do servidor.

Módulo de comunicação descentralizado

Classes utilizadas pelo módulo de comunicação descentralizado

A classe *Channel*

É responsável por criar o canal de comunicação. Nessa classe herda-se os conceitos que foram desenvolvidos pelo JGroups e então é aplicada algumas modificações para a eventual adaptação ao JEagle.

Na Figura 68 é possível ver um diagrama de classes que representa essa classe.

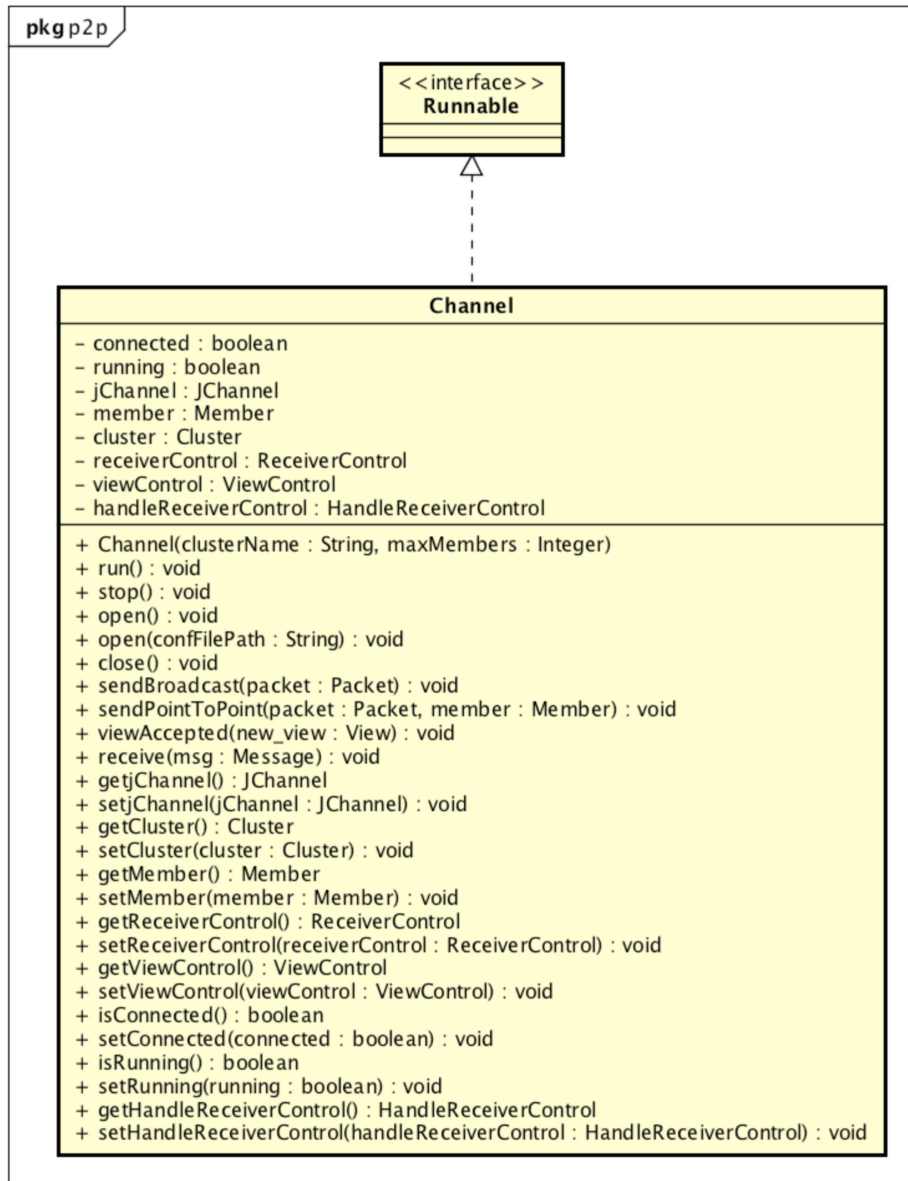

Figura 68: A classe *Channel*

Essa classe conterá os seguintes atributos:

- *private JChannel jChannel* - Esse atributo é o conceito herdado pelo JGroups, capaz de criar o canal de comunicação e então juntar-se a um grupo que esteja presente na rede.
- *private Member member* - É quem conterá as informações do membro local, ou seja, seu endereço único no grupo e também uma variável booleana informando se o membro local é o criador do grupo ou não.
- *private Cluster cluster* - É quem armazena o objeto *Cluster*, uma forma de abstração do conceito de grupos do JGroups, através dele é possível obter as informações do grupo, como por exemplo a lista de membros e quem é o criador do grupo.
- *private ReceiverControl receiverControl* - É o responsável por administrar as mensagens recebidas pelo canal local.
- *private ViewControl viewControl* - É quem gerencia as *views* do JGroups e as converte para *Members* do JEagle.
- *private boolean connected* - É apenas uma variável booleana para ter-se conhecimento se o canal esta conectado ou não ao grupo.
- *private boolean running* - É para ter conhecimento se a *thread* principal do módulo esta sendo executada, quando o JEagle estiver funcionando com *threads*.
- *private HandleReceiverControl handleReceiverControl* - É quem armazena a implementação da *interface HandleReceiverControl*, e através dela é possível adicionar métodos customizados ao módulo de comunicação.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Channel(String clusterName, Integer maxMembers)* - É o construtor da classe, esse construtor instância um novo *Cluster* com o seu respectivo nome e também a quantidade máxima de membros permitidos nesse *Cluster*, um *Cluster* nada mais é do que um grupo.
- *public void run()* - É quem inicia a *thread* principal do canal.
- *public void stop()* - É quem interrompe a execução da *thread* do canal.
- *public void open()* - É quem inicia a conexão do canal utilizando o arquivo de configuração padrão do JGroups “udp.xml”, ou seja, o método utilizado para iniciar a conexão com o grupo.

- *public void open(String confFilePath)* - É quem inicia a conexão do canal utilizando um arquivo de configuração customizado no endereço *confFilePath*, assim como o método anterior, ele é utilizado para conectar-se ao grupo.
- *public void close()* - É quem fecha o canal de comunicação com o grupo, ou seja, o método utilizado para desconectar-se.
- *public void sendBroadcast(Packet packet)* - É utilizado para enviar uma mensagem para todos os integrantes do grupo.
- *public void sendPointToPoint(Packet packet, Member member)* - É utilizado para enviar uma mensagem para um membro específico do grupo.
- *public void viewAccepted(View new_view)* -É um método herdado do JGroups quando acontece alguma forma de alteração sobre algum membro presente no grupo, ou seja, quando um membro conecta-se ou desconecta-se do grupo esse método é automaticamente executado, sendo assim, esse método atualiza a lista de membros conectados no grupo, além de também atualizar o criado do membro, caso ele tenha sido alterado.
- *public void receive(Message msg)* - É outro método herdado do JGroups. Quando um canal envia uma mensagem através da classe *JChannel* do JGroups, os membros do grupo que receberem essa mensagem irão automaticamente executar o método *receive()*, e então essa mensagem será repassada do método *receive()* do JGroups para o método *receive()* do JEagle.

A classe *Cluster*

É a abstração do conceito de grupos do JGroups para o JEagle, essa classe conterá as informações do grupo, e as deixará acessíveis para o membro local, sendo que ela conterá o nome do grupo, o número máximo de membros que poderão se conectar, a lista dos membros que estão conectados e também quem é o criador do grupo.

Na Figura 69 é possível ver um diagrama de classes que representa essa classe.

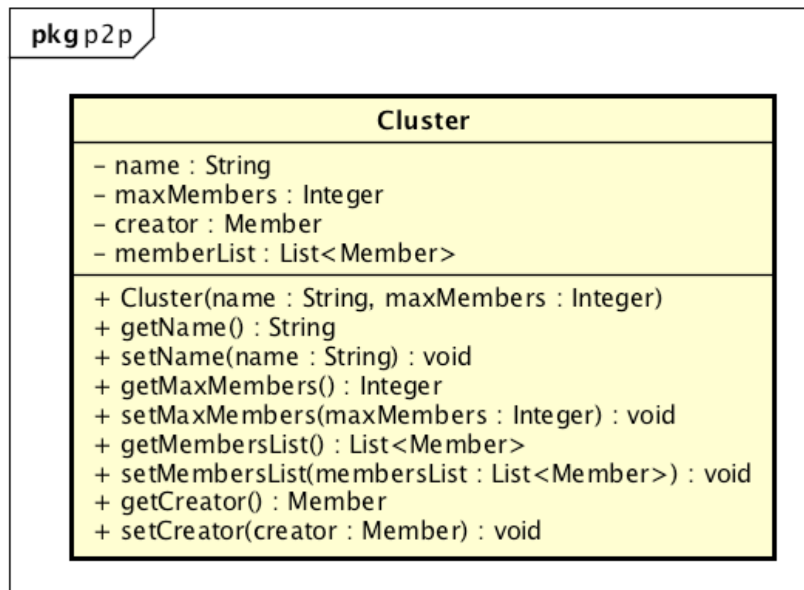

Figura 69: A classe *Cluster*

Os seguintes atributos estão presentes nessa classe:

- *private String name* - É uma variável do tipo *String* que armazena o nome do grupo.
- *private Integer maxMembers* - É uma variável do tipo inteiro que armazena a quantidade máxima de membros mutuamente conectados suportado pelo grupo.
- *private List<Member> membersList* - É uma lista que armazena os membros que estão atualmente conectados ao grupo.
- *private Member creator* - Armazena o atual membro criador do grupo.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Cluster(String name, Integer maxMembers)* - Método construtor da classe, adicionará um nome ao *cluster* e a quantidade máxima de membros que o grupo suportará.

A classe *Member*

É uma abstração das views que presentes no JGroups, ou seja, essa classe armazena as informações sobre um membro específico, essas informações são o endereço do membro no grupo e se ele é o criador do grupo ou não.

Na Figura 70 é possível ver um diagrama de classes que representa essa classe.

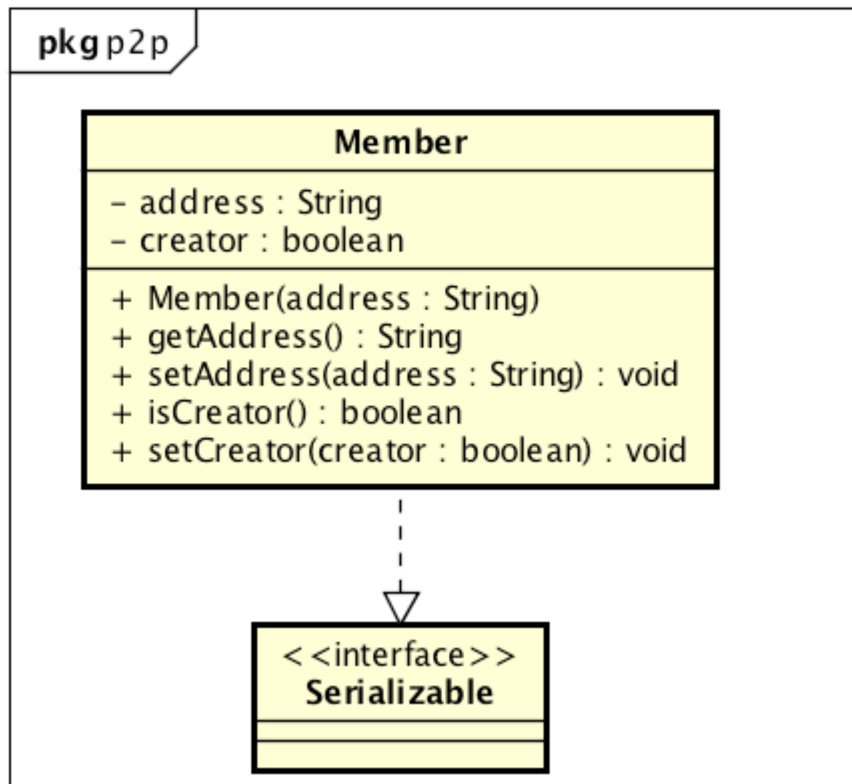

Figura 70: A classe *Member*

Essa classe contém os seguintes atributos:

- *private String address* - É uma adaptação da classe *Address* que esta presente no JGroups, ou seja, primeiramente obtemos a representação em *String* da classe *Address* e então a salvamos no atributo *address* do tipo *String* que esta presente na classe *Member*.
- *private boolean creator* - esse atributo é responsável por informar se o membro é o criador do grupo ou não.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Member(String address)* - Método construtor da classe, que apenas adicionará um endereço único ao membro.

A classe *Packet*

Já foi descrita nos tópicos passados, mas simplificadamente, ela é responsável por conter um *header* e também um *content*, de forma a estabelecer uma comunicação entre os *peers* que estão presentes no grupo.

Na Figura 71 é possível ver um diagrama de classes que representa essa classe.

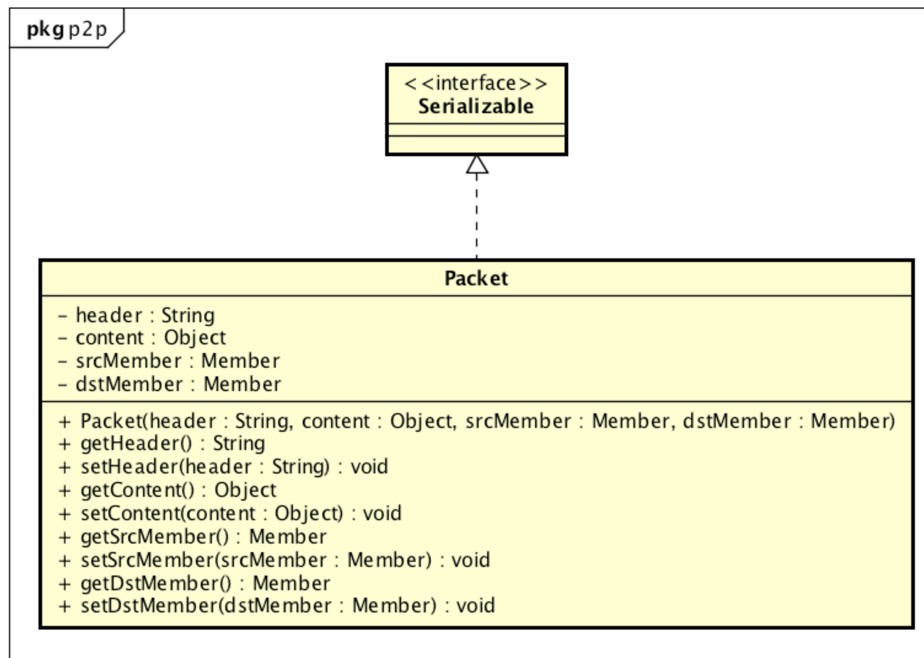

Figura 71: A classe *Packet*

Os seguintes atributos estão presentes nessa classe:

- *private String header* - Informa aos *peers* presentes no grupo qual ação eles devem executar quando receberem uma mensagem.
- *private Object content* - Variável do tipo *Object* que armazena qualquer tipo de objeto, essa variável existe para que seja possível existir a troca de qualquer tipo de informação entre os membros presentes no grupo.
- *private Member srcMember* - Variável do tipo *Member* que armazena informações sobre o membro emissor.
- *private Member dstMember* - Variável do tipo *Member* que armazena informações sobre o membro que deverá receber a mensagem.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public Packet(String header, Object content, Member srcMember, Member dstMember)* - Construtor da classe, que apenas adicionará as informações necessárias para que a mensagem seja enviada.

A classe *Ping*

É uma classe adicionada para calcular a quantidade de tempo necessária que uma mensagem demora para sair de um ponto A, chegar em B e então voltar novamente para o ponto A. Através dela é possível calcular a latência da rede.

Na Figura 72 é possível ver um diagrama de classes que representa essa classe.

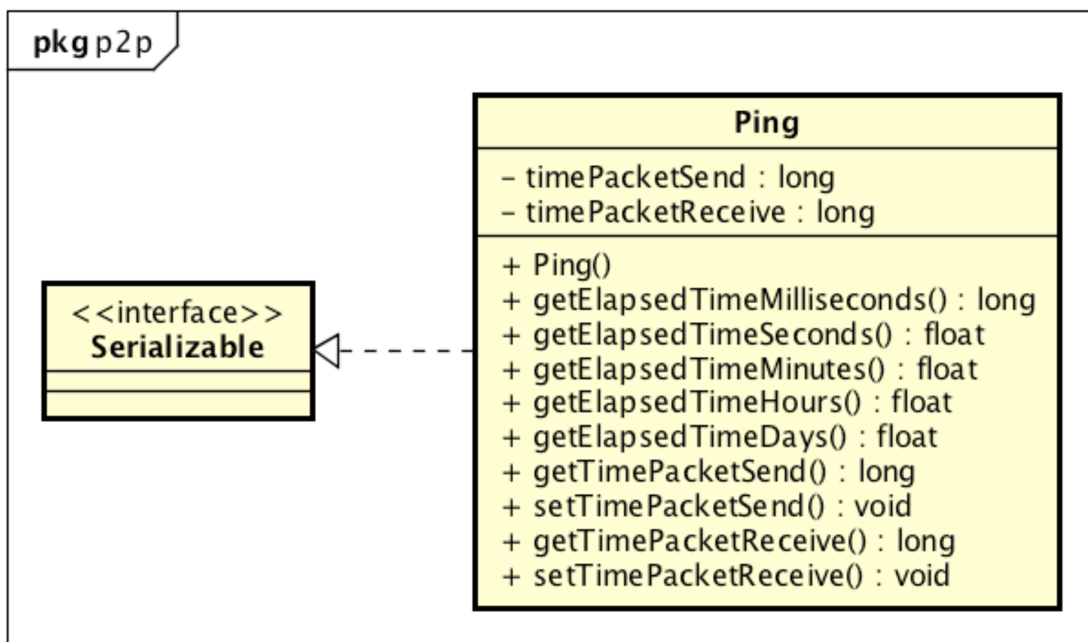

Figura 72: A classe *Ping*

Os seguintes atributos estão presentes nessa classe:

- *private long timePacketSend* - Variável do tipo *long* que deve receber um valor no momento em que o pacote for enviado.

- *private long timePacketReceive* - Assim como o atributo anterior, essa é uma variável do tipo *long*, porém agora seu valor deve ser adicionado no momento em que a mensagem for recebida.

Portanto contendo um tempo de quando a mensagem foi e enviada e quando a mensagem foi recebida, a operação que deve ser executada é:

$$ping = timePacketReceive\text{-}timePacketSend$$

Portanto através da fórmula acima é possível calcular o tempo necessário que uma mensagem demora para sair de A, chegar a B e então voltar a A novamente. Portanto essa classe contem apenas essa função, calcular a latência.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public long getElapsedTimeMilliseconds()* - Calcula a latência da rede em milissegundos.
- *public float getElapsedTimeSeconds()* - Calcula a latência da rede em segundos.
- *public float getElapsedTimeMinutes()* - Calcula a latência da rede em minutos.
- *public float getElapsedTimeHours()* - Calcula a latência da rede em horas.
- *public float getElapsedTimeDays()* - Calcula a latência da rede em dias.

A classe *ViewControl*

É a responsável por analisar as mudanças que acontecem nas *views* do JGroups e então executar as atualizações necessárias na lista de membros e no atual criador do grupo na classe *Cluster*.

Na Figura 73 é possível ver um diagrama de classes que representa essa classe.

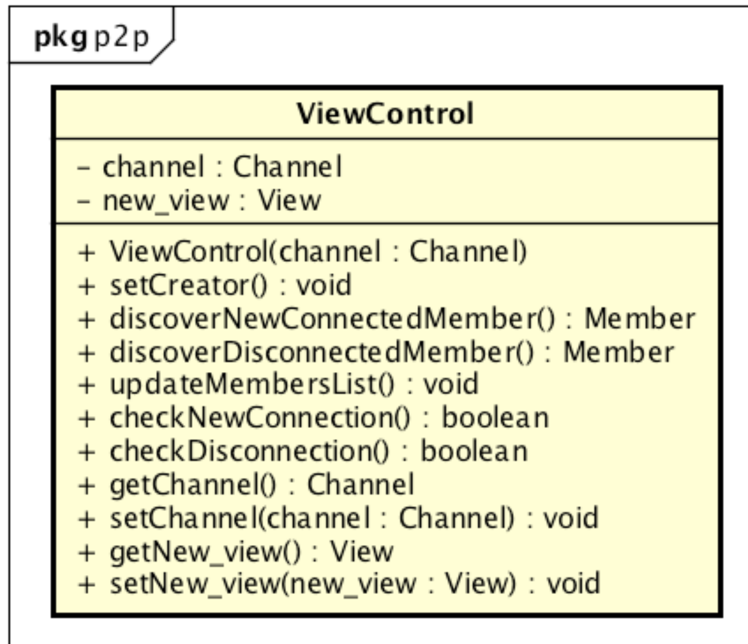

Figura 73: A classe *ViewControl*

Essa classe apresenta apenas dois atributos, eles são os seguintes:

- *private Channel channel* - Armazena uma referência para o atual canal, dessa forma pode-se acessar as informações do *cluster* e também do membro.
- *private View new_view* - Armazena uma cópia da *view* do JGroups, essa *view* é uma lista de *Address* dos membros atualmente conectados ao grupo.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ViewControl(Channel channel)* - É o construtor da classe, a função desse método é referenciar o canal do membro atual.
- *public void setCreator()* - Esse método adiciona um novo criador do grupo ao *cluster* do canal, caso esse criador ainda não exista. Caso o criador do grupo desconecte-se, então é necessário atualizar o criador do grupo com o próximo membro presente na *view* do JGroups, lembrando que a *view* é uma lista ordenada, e o primeiro membro da lista sempre será o criador do grupo.
- *public Member discoverNewConnectedMember()* - É responsável por descobrir o novo membro presente no grupo, para encontrar esse membro é necessário percorrer a *view* do JGroups e também percorrer a lista de membros presente no *cluster*, e verificar se cada membro da *view* esta presente na lista do *cluster*,

aquele membro que não estiver presente é o novo membro conectado ao grupo, então agora basta apenas retorná-lo.

- *public Member discoverDisconnectedMember()* - É o método responsável por encontrar o membro que desconectou-se. Para procurar esse membro basta percorrer a lista de membros do *cluster* e então comparar todos os membros dessa lista com os membros da *view* do JGroups, o membro da lista de membros do *cluster* que não estiver presente na *view* é o membro que desconectou-se do grupo.
- *public void updateMembersList()* - Converte a *view* do JGroups para a uma lista de membros compatível com a lista do *cluster* e então atualiza a lista do *cluster*.
- *public boolean checkNewConnection()* - Verifica se a *view* é maior que a lista do *cluster*, em caso afirmativo um novo membro conectou-se ao grupo.
- *public boolean checkDisconnection()* - Checa se a *view* é menor que a lista do *cluster*, em caso afirmativo um novo membro desconectou-se ao grupo.

A classe *ReceiverControl*

É a classe coração do módulo de comunicação, pois é aqui que as mensagens são recebidas, compreendidas e então os métodos necessários para o funcionamento de todo o controle dos membros do grupo e posteriormente do jogo que será desenvolvido serão executados. De uma maneira geral, essa classe apresenta um método principal que será executado sempre que uma nova mensagem for recebida pelo JGroups, e então essa mensagem será repassada para essa classe ReceiverControl do JEagle.

Na Figura 74 é possível ver um diagrama de classes que representa essa classe.

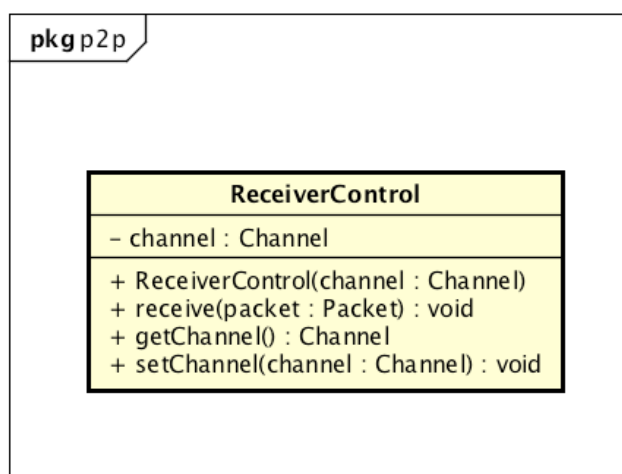

Figura 74: A classe *ReceiverControl*

Essa classe apresenta o seguinte atributo:

- *private Channel channel* - Único atributo presente nessa classe é uma referência para o canal atual, sendo que através dessa referência é possível ter acesso a todos os elementos do JEagle, como a lista de membros, o membro atual, o canal atual, etc.

Além dos métodos de *set()* e *get()*, os seguintes métodos também estão presentes nessa classe:

- *public ReceiverControl(Channel channel)* - É o construtor da classe, ele é responsável apenas por adicionar a referência do canal do grupo ao atributo *channel*.
- *public void receive(Packet packet)* - É um dos métodos principais do JEagle, pois é ele quem recebe a mensagem do tipo *Packet* e então baseando-se no *header* do *packet* é possível processá-lo e executar suas devidas ações.

A *interface HandleReceiverControl*

É a responsável por dar a possibilidade ao desenvolvedor que posteriormente irá utilizar o JEagle de adicionar seu código dentro das ações do módulo de comunicação, como por exemplo, quando um novo membro conecta-se ao grupo ou se o desenvolvedor quiser adicionar seus próprios *headers* a classe *ReceiverControl*, através dessa *interface* tudo isso torna-se possível. O desenvolvedor só precisa estar atento as palavras

reservadas utilizadas nos *headers* do JEagle, ou seja, *headers* reservados pelo JEagle que o desenvolvedor não pode utilizar em seu código. Essa palavras reservadas são:

- REQUEST_MEMBERS_LIST
- REQUEST_MEMBERS_LIST_RESPONSE
- REQUEST_PING
- REQUEST_PING_RESPONSE
- REQUEST_PING_FIX
- REQUEST_PING_FIX_RESPONSE

Fora essas palavras reservadas, o desenvolvedor pode utilizar qualquer outra *String* para os *headers* das suas mensagens customizadas.

Na Figura 75 é possível ver um diagrama de classes que representa essa classe.

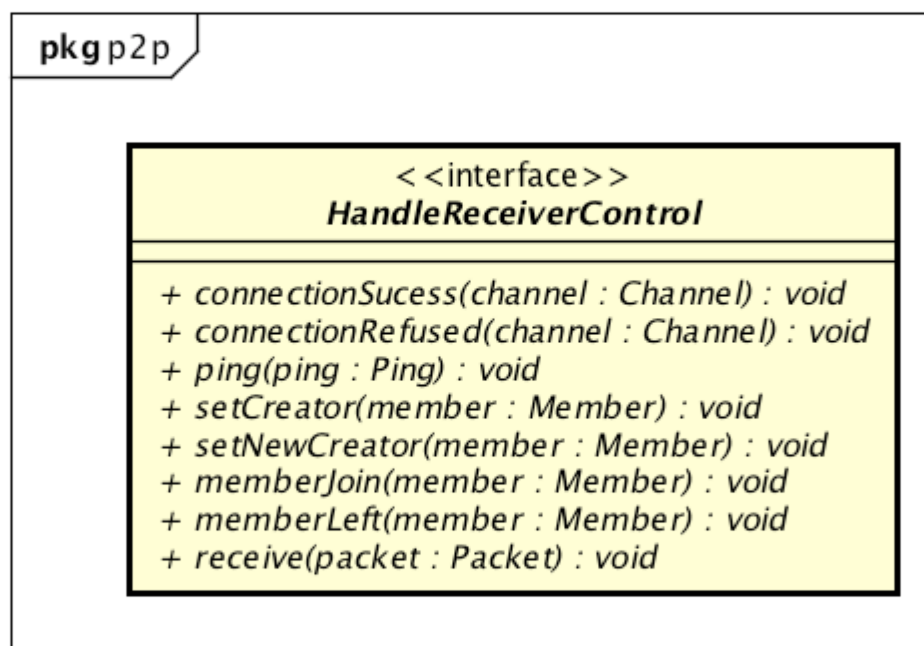

Figura 75: A *interface HandleReceiverControl*

Essa *interface* apresenta apenas os seguintes métodos que depois devem ser implementados em uma classe qualquer pelos desenvolvedores externos:

- *public void connectionSucess(Channel channel)* - Executará os códigos customizados do desenvolvedor quando o membro local conectar-se ao grupo.

- *public void connectionRefused(Channel channel)* - É onde o desenvolvedor pode executar qualquer código quando a conexão com o grupo for recusada, a condição para um membro ter sua conexão recusada é o grupo já conter o número máximo de membros conectados.
- *public void ping(Ping ping)* - É executado quando a requisição do comando de *ping* for retornado, aqui o desenvolvedor também poderá executar qualquer código.
- *public void setCreator(Member member)* - Qualquer código poderá ser executado quando o criador do grupo for adicionado pela primeira vez.
- *public void setNewCreator(Member member)* - Será executado quando o criador do grupo for atualizado, novamente, o desenvolvedor poderá executar qualquer código.
- *public void memberJoin(Member member)* - Esse método é executado quando um novo membro juntar-se ao grupo.
- *public void memberLeft(Member member)* - Esse método é executado quando um membro desconecta-se ao grupo.
- *public void receive(Packet packet)* - É um dos métodos mais importantes, pois é aqui que o desenvolvedor pode receber pacotes com *headers* customizados.