UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COORDENAÇÃO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
CURSO DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

IVAN LUIZ PICOLI

ARQUITETURA CLIENTE-SERVIDOR EM JOGOS *MULTIPLAYER*

TRABALHO DE CONCLUSÃO DE CURSO

PONTA GROSSA

2011

IVAN LUIZ PICOLI

ARQUITETURA CLIENTE-SERVIDOR EM JOGOS *MULTIPLAYER*

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Diplomação, do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná – UTFPR, como requisito parcial para obtenção do título de Tecnólogo.

Orientador: Prof. Dr. André Koscianski

PONTA GROSSA

2011

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CAMPUS PONTA GROSSA
DIRETORIA DE GRADUAÇÃO
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA DE TRABALHO DE DIPLOMAÇÃO

TERMO DE APROVAÇÃO

ARQUITETURA CLIENTE-SERVIDOR EM JOGOS *MULTIPLAYER*

Ivan Luiz Picoli

Este Trabalho de Diplomação foi considerado adequado como cumprimento das exigências legais do currículo do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas e aprovado em sua forma final pela Coordenação de Análise e Desenvolvimento de Sistemas da Universidade Tecnológica Federal do Paraná – Campus de Ponta Grossa.

Prof.° Dr. ANDRÉ KOSCIANSKI
Orientador

Profª. Msc. HELYANE BRONOSKI BORGES
Responsável pelo Trabalho de Diplomação

Prof. Dr. ANDRÉ KOSCIANSKI
Coordenador do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Banca Examinadora:

Prof. Dr. LOURIVAL A. DE GÓIS

Prof. Dr. GLEIFER VAZ ALVES

Aos espíritos de luz, à família, à vida, ao conhecimento,
aos gênios da humanidade, ao Half
e aos três pássaros.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me concedido a oportunidade de adquirir tantos conhecimentos os quais consegui estudando nesta universidade, e aos seres de luz que me acompanham e me ajudam nesta jornada.

Agradeço aos meus pais, que sempre me apoiaram em meus estudos e me ajudaram em tudo que precisei, às vezes oferecendo até o que não podiam me conceder em dado momento, abdicando de seus próprios desejos e enfrentando dificuldades para que eu pudesse alcançar esse objetivo.

Agradeço ao meu irmão Tony por me apoiar e sempre seguir a mesma linha de raciocínio para tomar decisões importantes, e também por estar ao meu lado mesmo na distância, em todos os momentos que precisei. Obrigado irmão.

Não poderia deixar de agradecer os grandes amigos que fiz e pessoas especiais que conheci durante minha estadia nesta cidade, pessoas as quais me ajudaram nos momentos difíceis e sorriram comigo nos momentos de felicidade. Não poderia deixar de agradecer ao Pozzo, Jacson, Silvio, Bluetooth, Vinícius e Alexssandro pelas conversas, partidas de Age e momentos de descontração. À Emanoély, Larissa e Lays pela amizade e ajudas durante os estudos. E aos outros que me ajudaram no decorrer dos semestres e não me lembro.

Agradeço aos meus professores por ter transmitido seus conhecimentos e me preparar para o mercado de trabalho e para a carreira acadêmica. Agradecimentos especiais ao meu orientador Dr. André Koscianski, por ter me concedido muitas oportunidades durante o curso e por me ajudar no decorrer deste estudo. Também ao professor Dr. Góis por me incentivar e me ajudar no momento em que estava no Exército, momento esse que precisei de incentivo para continuar os estudos e concluir o curso do NPOR.

Agradeço a todas as outras pessoas que não me esqueceram em 2009, durante a carreira militar conjunta com a universidade. Agradecimentos especiais à Isabel, pessoa a qual foi uma fonte de incentivo e me ajudou nos momentos em que realmente precisei, nunca esquecerei e muito obrigado.

E agradeço a todos que de alguma forma, me ajudaram nesses anos.

“Vedi di non chiamare intelligenti solo
quelli che la pensano come te”
(Ugo Ojetti)

“Cuidado para não chamar de inteligentes
apenas aqueles que pensam como você”
(Ugo Ojetti)

RESUMO

PICOLI, Ivan Luiz. **Arquitetura Cliente-Servidor em Jogos *Multiplayer***. 2011. Trabalho de Conclusão de Curso de graduação – Curso de Tecnologia em Análise e Desenvolvimento de Sistemas – Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2011.

O desenvolvimento de jogos envolve diversas áreas do conhecimento. Um jogo que englobe a plataforma *multiplayer* possibilita que os jogadores interajam em um mesmo ambiente, vendo as ações executadas em computadores remotos controlados pelos outros jogadores e podendo estar localizados em lugares diferentes e distantes. Um jogo *multiplayer* é um sistema distribuído, e para que seus processos se comuniquem, é necessário o envio de mensagens através da rede. Desenvolver um padrão de mensagens que possibilite um maior desempenho do jogo é uma tarefa que exige uma análise prévia. Este trabalho propõe uma arquitetura de rede e um padrão de mensagens que minimiza o tráfego de rede em jogos *multiplayer*, e possibilita que os objetos dos jogadores remotos se mostrem sincronizados em todos os clientes conectados no mesmo ambiente distribuído, tendo suas coordenadas corretas e movimentos realizados em tempo real.

Palavras-chave: Linguagem de Programação. Python. Jogos em Rede. Redes de Computadores. Desenvolvimento de Jogos.

ABSTRACT

PICOLI, **Ivan Luiz. Client-Server Architecture in *Multiplayer* Games**. 2011. Final Paper – Systems Analysis and Development Technology, Federal Technological University of Paraná. Ponta Grossa, 2011.

Game developing involves several areas of knowledge. A *multiplayer* game allows users to interact in the same environment, seeing the actions executed on remote computers controlled by other players, who can be located in different and distant places. A *multiplayer* game is a distributed system and for their processes to communicate, it is required the dispatch of messages through the network. Developing a pattern of messages that allows a higher performance of the game is a task that requires a previous analysis. This work proposes a network architecture and a message pattern that decrease the network traffic in *multiplayer* games, and allow the remote objects of the players to show up synchronized on all clients connected to the same distributed environment, with their corrected coordinates and movements performed in real time.

Keywords: Programming Language. Python. Network Gaming. Computers Networks. Game Development.

LISTA DE FIGURAS

Figura 1 – Dinâmica de melhoria entre produto e processo na Engenharia de Software....19
Figura 2 – Semelhança de importação entre linguagens 21
Figura 3 – Caminho e transformação de uma mensagem transmitida através das camadas....24
Figura 4 – Modelo OSI e as sete camadas que o compõe....25
Figura 5 – Modelo TCP/IP comparado ao modelo OSI 27
Figura 6 – Faixas do endereço IP e suas respectivas classes....28
Figura 7 – Primitivas de *Sockets* utilizadas no protocolo TCP 30
Figura 8 – Estrutura de um frame que transita pela rede, contendo um pacote TCP 32
Figura 9 – Comunicação entre processos no modelo cliente-servidor....33
Figura 10 – Modelo cliente-servidor utilizando *thread* para múltiplos clientes....34
Figura 11 – Exemplo de aplicação que implementa duas bolas quicando....38
Figura 12 – Trecho de código python que controla a velocidade vertical da bola quicando....38
Figura 13 – Bee Invader....39
Figura 14 – Classe Bee, representando a abstração de uma abelha....41
Figura 15 – Criação de *thread* pelo método start_new_*thread*() 42
Figura 16 – Herança da classe *Thread*....42
Figura 17 – Criação e primitivas de *socket* passivo com protocolo TCP....43
Figura 18 – Console das aplicações de dois clientes e do servidor do *chat* 44
Figura 19 – Criação das *threads* do servidor ao receber conexões de novos clientes 45
Figura 20 – Envio e redirecionamento de mensagens entre os clientes do *chat* 46
Figura 21 – Padrão de envio das coordenadas dos objetos 49
Figura 22 – Saltos dos objetos remotos ao enviar coordenadas por tempo....49
Figura 23 – Padrão de envio das ações dos jogadores 51
Figura 24 – Estrutura e informações enviadas no padrão das ações do jogador....51
Figura 25 – Três jogadores interagindo no mesmo ambiente do jogo....53
Figura 26 – Diagrama de Casos de Uso das ações até o início do jogo 53
Figura 27 – Diagrama de Casos de Uso das ações durante o jogo 54
Figura 28 – Atividades do servidor mostradas no console....56
Figura 29 – Diagrama de Classes do servidor do jogo....57
Figura 30 – Lista de servidores ativos encontrados no console da aplicação cliente 59
Figura 31 – Diagrama de Atividades referente ao controle da lógica do jogo 60
Figura 32 – Serialização de mensagens em Python 61
Figura 33 – Diagrama de Classes do Cliente na execução das opções de menu 64
Figura 34 – Diagrama de Classes do cliente na execução do ambiente do jogo....64
Figura 35 – Diagrama de Classes da estrutura de controle do jogo....65

LISTA DE TABELAS

Tabela 1 – Quantidade de mensagens recebidas pelo servidor em um período de tempo 67

Tabela 2 – Quantidade de impressões dos objetos dos jogadores (FPS) vistos nos outros clientes 67

SUMÁRIO

1 INTRODUÇÃO 14
1.1 DELIMITAÇÃO DO TEMA 14
1.2 ABORDAGEM DO PROBLEMA 15
1.3 OBJETIVOS 15
1.3.1 Objetivo Geral 15
1.3.2 Objetivos Específicos 16
1.4 ESTRUTURA 16
1.5 INTRODUÇÃO A DESENVOLVIMENTO DE JOGOS 16
1.5.1 Tecnologias 16
1.5.2 Gêneros de Jogos 17
1.5.3 Jogos em Rede 18
2 EMBASAMENTO TEÓRICO 19
2.1 DESENVOLVIMENTO DE JOGOS 19
2.1.1 Engenharia de Software para Jogos 19
2.1.2 Bibliotecas e Motores 20
2.2 COMUNICAÇÃO EM REDE 22
2.2.1 Protocolos de Comunicação 22
2.2.2 Modelos de Referência 24
2.2.2.1 Modelo OSI 24
2.2.2.2 Modelo TCP/IP 26
2.2.3 Protocolo IP (Internet Protocol) 27
2.2.4 A Camada de Transporte 29
2.2.4.1 Soquetes (*Sockets*) 29
2.2.4.2 Protocolo UDP 30
2.2.4.3 Protocolo TCP 31
2.2.4.4 Pilha de Protocolos 31
2.3 ARQUITETURA CLIENTE-SERVIDOR 32
2.3.1 *Threads* 33
2.3.2 Problemas de Sincronia e Uso de Multi-*threads* 35
2.4 LINGUAGEM DE PROGRAMAÇÃO PYTHON 35
3 DESENVOLVIMENTO 36
3.1 LINGUAGEM PYTHON E BIBLIOTECA GRÁFICA PYGAME 37
3.1.1 Apresentação Do *Engine* 37
3.1.2 Exemplos De Implementação 37
3.1.2.1 Bolas quicando 37
3.1.2.2 Bee invader 39
3.2 PYTHON E A ORIENTAÇÃO A OBJETOS 40
3.2.1 Tecnologia Orientada A Objetos 40

3.2.2 Implementando Orientação A Objetos Em Python...................................... 40
3.3 *THREADS* E *SOCKETS* EM PYTHON 41
3.3.1 Implementação De *Threads*.. 41
3.3.2 Implementação De *Sockets* 42
3.3.3 O Uso Conjunto De *Threads* E *Sockets*.. 43
3.4 IMPLEMENTAÇÃO DE UM *CHAT* SIMPLES 44
3.4.1 Métodos E Arquitetura Utilizada... 44
3.4.2 Processo De Conexão De Novos Clientes 45
3.4.3 O Redirecionamento De Mensagens 45
3.4.4 Arquitetura Do *Chat* Em Jogos *Multiplayer* 46
3.5 UTILIZANDO A REDE NO DESENVOLVIMENTO DE JOGOS......................47
3.5.1 Problemas Encontrados Ao Utilizar A Rede Em Jogos............................... 47
3.5.2 Possíveis Soluções Aos Problemas Encontrados....................................... 47
3.5.2.1 Envio das coordenadas dos objetos 48
3.5.2.2 Envio das ações dos jogadores 50
3.6 PROJETO E DESENVOLVIMENTO DE UM JOGO *MULTIPLAYER* 52
3.6.1 Descrição Geral 52
3.6.2 Visão Geral 52
3.6.3 Tecnologias Utilizadas 54
3.6.4 O Servidor... 55
3.6.4.1 Envio de broadcast para servidor dinâmico 55
3.6.4.2 Conexões de novos clientes... 55
3.6.4.3 Redirecionamento de mensagens 56
3.6.4.4 Tratamento de conexão perdida 57
3.6.4.5 Estrutura da aplicação servidora... 57
3.6.5 O Cliente... 58
3.6.5.1 Busca e escolha de servidores 58
3.6.5.2 Controle do fps e separação da lógica 59
3.6.5.3 Estrutura interna das mensagens enviadas... 61
3.6.5.4 Mensagem refresh para controle de sincronia... 62
3.6.5.5 Estrutura da aplicação cliente... 63
3.7 ANÁLISE DOS RESULTADOS...65
3.7.1 Implementações Anteriores 65
3.7.2 Melhorias Alcançadas... 66
3.7.3 Pontos Críticos.. 68
4 CONCLUSÃO..68
4.1 ANÁLISE GERAL DO DESENVOLVIMENTO... 68
4.2 TRABALHOS FUTUROS 69
4.3 CONSIDERAÇÕES FINAIS 70
REFERÊNCIAS 71

APÊNDICE A – FLUXO DE DADOS DURANTE A CONEXÃO DE UM CLIENTE E O ENVIO DE MENSAGENS COM DOIS CLIENTES CONECTADOS..............74
APÊNDICE B – GRÁFICOS DE DESEMPENHO DOS PADRÕES DE ENVIO DE MENSAGENS 76

1 INTRODUÇÃO

A Tecnologia da Informação cresce à medida que novas tecnologias de hardware e software são disponibilizadas no mercado, possibilitando que empresas especializadas utilizem-nas para desenvolvimento. Com essa crescente inovação, é necessário manter-se atualizado junto a gama de tecnologias do momento.

O mercado de software é extremamente heterogêneo quanto às áreas de desenvolvimento, empresas do mundo todo competem para desenvolver inovações na informatização de hospitais, mercados, órgãos do governo, e uma infinidade de outras instituições. O desenvolvimento de videogames é voltado para os jogadores, que geralmente exigem gráficos de alto desempenho, ligado com uma boa jogabilidade e velocidade, proporcionando uma imagem em tempo real e a imersão do usuário (ROOLINGS, 2004).

Levando em consideração estes aspectos, o desenvolvimento de um jogo é uma tarefa complexa, que exige da empresa e seus programadores um grande planejamento e capacidade de trabalhar em equipe, envolvendo as mais variadas áreas, como história e enredo, gráficos, sons, jogabilidade, animação e a programação em si. Tudo exige minuciosa atenção, o projeto de um jogo tem alto custo, e terá um retorno frustrante caso o universo de jogadores o rejeitem assim que lançado, porém, se bem aceito com certeza será lembrado por muito tempo e o retorno será muito benéfico, aliás, quem não se lembra do famoso Mundo de Mário, ou das lutas e personagens de Street Fighter e Mortal Kombat, e relacionando com jogos do tipo RPG, o lendário Zelda e as aventuras em Chrono Trigger.

1.1 DELIMITAÇÃO DO TEMA

O presente documento propõe demonstrar a arquitetura envolvida no processo de desenvolvimento de jogos *multiplayer*, através de implementações na linguagem python dos conceitos apresentados. Será mostrado o fluxo de informações entre os computadores durante a comunicação entre os processos correntes do jogo, também serão abordadas soluções para eventuais problemas ocorridos na comunicação pela rede.

1.2 ABORDAGEM DO PROBLEMA

Ao programar jogos utilizando a rede como meio de comunicação entre processos, surgem diversos empecilhos que acabam por diminuir o desempenho do jogo em relação àquele onde não fora implementado tal recurso. Quando dois ou mais jogadores conectam-se e iniciam uma partida, é necessário que as imagens de todos os jogadores apareçam nos locais corretos. Ao sobrecarregar a rede com muita informação, os dados essenciais para o correto funcionamento do jogo podem não chegar na correta sincronia, ocasionando um jogo lento e com falhas.

A solução proposta visa o funcionamento de um jogo *multiplayer* utilizando uma arquitetura que possibilite o menor tráfego de informações possível pela rede, armazenando todas as informações nos clientes e enviando ao servidor apenas o necessário, com isso previne-se também o problema de sobrecarga do servidor.

Outro problema, porém secundário, é tratar o consumo de recursos de hardware durante o processamento do jogo. O controle da impressão das imagens e sua separação da lógica do jogo deverão possibilitar que a plataforma *multiplayer* funcione sem a preocupação com o consumo de hardware, o que poderia gerar mais um problema em relação às posições e sincronia dos jogadores.

Tais problemas foram alvos de estudo de um projeto de Iniciação Científica, o qual resultou neste Trabalho de Conclusão de Curso.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

Desenvolver e modelar uma arquitetura que possibilite o menor tráfego de informações pela rede durante o processamento de um jogo *multiplayer*, resultando na criação de uma aplicação distribuída, utilizando a linguagem de programação Python e a arquitetura cliente-servidor.

Neste ambiente, vários jogadores poderão conectar-se ao ambiente do jogo e interagir com os outros que se encontram conectados, onde todos possam visualizar as imagens e objetos controlados pelos jogadores remotos em suas reais posições.

1.3.2 Objetivos Específicos

- Descrever as tecnologias utilizadas para desenvolvimento da aplicação e relacioná-las com o projeto conforme necessário.
- Disponibilizar alguns exemplos relevantes de código Python implementando tecnologias como *sockets* e *threads*;
- Projetar e implementar uma aplicação distribuída, utilizando a arquitetura cliente-servidor, na qual o servidor funcionará como redirecionador de mensagens e controlador dos jogadores clientes conectados à ele;
- Apresentar os resultados obtidos através de testes realizados com a arquitetura do jogo proposta.
- Descrever a implementação de maneira a ajudar a construção de futuros projetos semelhantes;

1.4 ESTRUTURA

O presente documento é dividido em quatro partes. No capítulo 1 apresenta-se a introdução. No capítulo 2 são apresentadas todas as tecnologias, conceitos e fundamentos utilizados para desenvolvimento do jogo. Durante a abordagem dos conceitos científicos são citados os pontos do desenvolvimento onde se encaixam tais conceitos. O capítulo 3 é o desenvolvimento do projeto, demonstra desde os estudos iniciais da linguagem, mostrando as implementações realizadas para obter o conhecimento necessário para o desenvolvimento do jogo *multiplayer*, além de conter o projeto e o desenvolvimento da versão final do jogo e seus resultados. No capítulo 4 a conclusão do projeto, mostrando os trabalhos futuros e considerações finais.

1.5 INTRODUÇÃO A DESENVOLVIMENTO DE JOGOS

1.5.1 Tecnologias

Muitas empresas especializam-se em desenvolvimento de *frameworks* (CAPELLER, 2010), ajudando na velocidade e praticidade de desenvolvimento em outras empresas. Esse tipo de software, por trabalhar com partes já prontas de

código, facilitam e agilizam o desempenho do programador, são muito procurados e bem vistos no mercado. Na programação de jogos não é diferente, os frameworks utilizados possuem o nome *engine* (ou motor), e é a partir dele que todos os cálculos necessários são programados e executados.

Um jogo pode conter cálculos matemáticos extremamente complexos, principalmente na parte de renderização de imagens, utilizando matrizes bidimensionais e tridimensionais, por esse motivo consomem grande quantidade de processamento gráfico (BOURG, 2002; TREMBLAY, 2004). Sendo assim, os *engines* são muito utilizados pelas empresas produtoras de jogos, algumas até produzem o seu próprio, é o caso dos clássicos mais famosos como Crysis, que utiliza o Cry*Engine*.

Esses cálculos são necessários para obter movimentos e renderizar as imagens mostradas no vídeo. Em relação aos gráficos, existem os jogos desenvolvidos em plataforma 2D ou duas dimensões, geralmente exigem menos processamento. Outro tipo de plataforma é a 3D ou três dimensões, exigem mais processamento gráfico, porém proporcionam uma maior realidade e imersão em sua jogabilidade (BOURG, 2002). São inúmeros os *engines* para as plataformas 2D e 3D, alguns serão citados no próximo capítulo. Em geral, a utilização de um motor para desenvolver um jogo é crucial para o sucesso de um bom produto final.

1.5.2 Gêneros de Jogos

Para os jogadores, o videogame é uma diversão temporária, para outros, um objetivo, no qual prevalece a vontade de finalizar a história fictícia na qual se submeteram ao iniciá-lo. Vendo por outro ângulo, do lado dos desenvolvedores, um jogo não é simplesmente uma diversão, muito menos um objetivo do gênero, e sim uma meta, na qual prevalece a vontade de finalizar o desenvolvimento e sentir a satisfação não somente no lucro da empresa, mas também de saber que seus usuários, ou seja, os jogadores valorizaram o software e criaram o objetivo de finalizar a história.

Sendo assim, o gênero do jogo é muito importante para decidir o público alvo, existem fanáticos pelos mais variados gêneros, segundo dados da internet os mais procurados são do gênero de ação e guerra, como Battlefield, FarCry e Crysis, muito jogados também, são os gêneros de corrida como Dirt e Need for Speed, ação

e terror como Resident Evil e outros gêneros que se enquadram outros clássicos, envolvendo pessoas do mundo todo em suas histórias.

O gênero considerado mais "viciante" pelos jogadores é o *Role-playing game* (RPG) o qual incluem, em sua maioria a arquitetura *multiplayer*. Trata-se de jogos onde o jogador cria seu personagem e entra em um mundo de fantasias com o objetivo de tornar-se mais poderoso em relação a sua experiência, armas, armaduras e magias, podendo assim enfrentar desafios mais difíceis, as chamadas *quests*, que são as missões realizadas pelo personagem.

Um jogo deste gênero associado com a plataforma *multiplayer*, que é alvo de estudo no contexto, tem como resultado o *Multi Massive Online Role-Playing Game* (MMORPG), que reúne jogadores de todo o mundo, relacionando-se no mundo de ficção através de seu personagem. Alguns clássicos famosos que implementam essa plataforma são o World of Warcraft e Perfect World, que além, utilizam gráficos em três dimensões, exigindo grande processamento gráfico e uma banda de Internet razoável, 1MB ou superior.

1.5.3 Jogos em Rede

Com o início da comunicação entre os computadores através da rede e o advento da Internet, os primeiros sistemas começaram a se interagir através de protocolos (TANENBAUM, 2004). Não foi diferente na área de programação de jogos, imaginou-se muito interessante que os jogos do momento, tão bem vistos pela comunidade de jogadores, pudessem ser jogados em uma plataforma *multiplayer* em ambientes separados e totalmente independentes entre si, comunicando-se apenas através da infraestrutura de rede.

Com a expansão da Internet pelo mundo, não demorou muito até que muitos jogos *multiplayer online* estivessem disponíveis, principalmente os MMORPG. Um jogo desse tipo deve implementar a arquitetura Cliente-Servidor para que os jogadores possam interagir à distância, essa arquitetura será melhor definida no decorrer do trabalho. Ao utilizar a Internet, os usuários de um sistema podem se localizar em qualquer lugar do planeta, portanto, um servidor para centralizar as informações dos computadores conectados é muito importante para obter sucesso nas informações enviadas aos clientes. Erros de programação na parte da comunicação entre os processos do sistema podem acarretar em erros de sincronia

graves e o jogo poderá ficar totalmente sem nexo. Por exemplo, em um computador o inimigo está na posição *x* e será morto por um disparo, porém no computador do inimigo ele se encontra na posição *y* e morre sem saber o motivo.

2 EMBASAMENTO TEÓRICO

2.1 DESENVOLVIMENTO DE JOGOS

2.1.1 Engenharia de Software para Jogos

Como todo software, para o bom desenvolvimento de um jogo é necessário a aplicação de Engenharia de Software em seu projeto. O fundamento dessa técnica é aplicar boas normas de desenvolvimento e de processos com o intuito de obter um bom resultado no produto final. Flynt (2005) demonstra esse processo detalhadamente, afirmando que melhorar os processos de desenvolvimento é a base para melhorar seus produtos, e buscando sempre o melhor produto, também haverá motivação para buscar essa melhoria nos processos. Sendo assim, a Engenharia de Software une essas duas atividades em um ciclo contínuo, como demonstra a Figura 1.

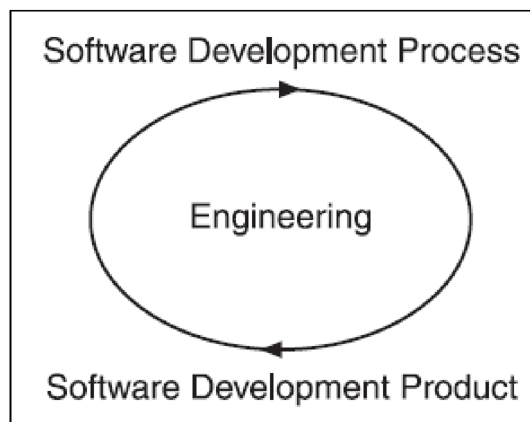


Figura 1 – Dinâmica de melhoria entre produto e processo na Engenharia de Software
Fonte: FLYNT (2004, P. 3)

O processo de desenvolvimento de um jogo envolve diversas áreas de estudo. Antes de iniciar o projeto do jogo em si, demonstrando como o mesmo será implementado mais adiante através de diagramas, é necessário ter em mãos a história, ou seja, o enredo do jogo. Ele inclui todos os requisitos, a partir daí, começarão a aparecer os sons, modelos de personagens, e também o projeto do desenvolvimento, contendo os esboços e diagramas finais das classes e objetos que

irão compor o jogo. É nessa parte que entrará a Engenharia de Software e todo o cuidado em realizar a análise do funcionamento interno do software.

O estudo dessa área no desenvolvimento de jogos é importante, porém é um conteúdo extenso e requer tempo e dedicação para estudá-lo mais a fundo, portanto teremos o foco nos diagramas da UML (*Unified Modeling Language*) (LARMAN, 2000) utilizados no decorrer do projeto. Diagramas são úteis para analisar o funcionamento do software e ajudam a encontrar erros mais facilmente. Os mais utilizados são:

- Diagrama de Casos de Uso: Descreve a funcionalidade proposta para um novo sistema, que será projetado.
- Diagrama de Classes: É a representação da estrutura e relações das classes que servem de modelos para os objetos.
- Diagrama de Sequência: Representa a sequência de processos, ou seja, a troca de mensagens entre os objetos de um sistema no decorrer do tempo.
- Outros também importantes, a título de conhecimento são: Diagrama de Colaboração, Diagrama de Estados, Diagrama de Atividades, Diagrama de Componentes e de Implantação.

Esses diagramas fazem parte da UML 2.0, detalhadamente demonstrados no livro de Larman (2000), e podem ser mais bem compreendidos ao estudar o paradigma da Orientação a Objetos, que será abordado no Capítulo 3.2.1.

2.1.2 Bibliotecas e Motores

Com o desenvolvimento da tecnologia, linguagens de programação vão ganhando cada vez mais recursos. Novas funcionalidades que facilitam o desenvolvimento de softwares são desenvolvidas, esses são pedaços de códigos, funções pré-compiladas com a linguagem que bastam ser importadas no seu código para serem utilizadas. Praticamente todas as linguagens de programação possuem essas funções, porém são representadas de formas diferentes, e há funções existentes em certas linguagens que não são encontradas em outras. Dessa forma, ao produzir um software é importante analisar seus requisitos e escolher uma boa linguagem para desenvolvê-lo. As funções são armazenadas em arquivos embutidos na linguagem e que podem ser importados por padrão quase da mesma forma,

como será mostrado na Figura 2. Essas são as chamadas bibliotecas, e apelidadas de motor (ou *engine*) quando desenvolvidas com o intuito de programar jogos.

```
13  import java.util.Timer;
14  public class NewJFrame extends javax.swing.JFrame {
15      Timer t;
```
Importação em java

```
1  import time
2
3  time.time()
```
Importacão em Python

Figura 2 – Semelhança de importação entre linguagens

Fonte: Autoria própria

Uma das linguagens mais utilizadas para produção de jogos é a linguagem C, e C++ que implementa a orientação a objetos. O motivo dessa escolha é pela velocidade que C proporciona e por ser uma linguagem de baixo nível, onde é possível controlar quase que totalmente o fluxo de informações durante a execução do código. Alguns *engines* que podem ser adicionados e utilizados são:

- Allegro: Utilizada no desenvolvimento de jogos em 2D, possui funções fáceis de aprender, pode ser encontrada juntamente com sua documentação em *alleg.sourceforge.net/*.
- SDL: Utilizada também no desenvolvimento de aplicativos de multimídia, possui funções mais complexas e requer mais tempo de estudo, contém funções para desenvolvimento de jogos em 3D, pode ser encontrada juntamente com sua documentação em *www.libsdl.org/*.
- Irrlicht: Utilizada na produção de jogos em 3D, é uma boa opção para iniciantes, por ser mais simples de entender, pode ser encontrada juntamente com sua documentação em *irrlicht.sourceforge.net/*. Exemplos de implementação desse *engine* são demonstrados por Koscianski (2011).

Jogos podem ser desenvolvidos nas mais diversas linguagens, porém deste ponto em diante a linguagem estudada será Python, uma vez que o exemplo de implementação de um jogo *multiplayer* fora implementado com tal *engine*, e será apresentado no decorrer do trabalho.

Alguns motores que podem ser utilizados com Python:

- Pygame: biblioteca multimídia de fácil manipulação. Permite manipular superfícies de desenho e criação de jogos 2D. Pode ser encontrada juntamente com sua documentação em *www.pygame.org/*. Será detalhada mais a frente.
- Panda 3D: biblioteca gráfica, desenvolvida pelos estúdios Disney, com ela é possível criar até mesmo jogos profissionais. Pode ser encontrada juntamente com sua documentação em *www.panda3d.org/*.
- PyOgre: Outra ferramenta utilizada na produção de jogos 3D. Pode ser encontrada juntamente com sua documentação em *www.ogre3d.org/*.

2.2 COMUNICAÇÃO EM REDE

Antes do advento da tecnologia de comunicação através da rede e pelo meio eletrônico, ou seja, em grande parte do século XX e nos séculos anteriores, a comunicação era feita através de cartas e mensagens enviadas por meio físico, demorando dias ou dependendo da distância, meses até alcançar o destino. Nos dias de hoje, com a crescente tecnologia nessa área, tornou-se possível não somente a comunicação em tempo real a longa distância, mas também a procura e aquisição de vídeos, músicas, livros e documentos em geral, incluindo filmes e aulas digitais através da internet. Com o passar do tempo acredita-se que será possível efetuar ações através da rede que hoje e no passado parecem práticas impossíveis de se realizar, como talvez a implementação de uma realidade virtual conectada à Internet, onde até o olfato seria utilizado como forma de interação (TANENBAUM, 2003).

Os principais conceitos em relação à comunicação através da rede serão demonstrados a seguir, sendo importantes para auxiliar a compreensão das implementações realizadas no decorrer do projeto.

2.2.1 Protocolos de Comunicação

Seria muito fácil dizer que um computador comunica-se com outro através de mensagens enviadas pela rede, porém a comunicação entre processos em máquinas distintas é muito mais complexa, envolvendo uma série de tarefas e transmissão de bytes para que a mensagem propriamente dita seja encaminhada ao seu destino. Essas tarefas são executadas entre as camadas existentes em uma

máquina, que serão detalhadas no próximo capítulo através dos modelos. O importante a saber é que as camadas mais baixas fornecem chamadas de procedimentos para as camadas superiores, e esses procedimentos são definidos em interfaces existentes entre as camadas (TANENBAUM, 2003).

Quando ocorre uma transmissão, diz-se que há uma transmissão lógica entre as camadas do mesmo nível da máquina cliente e da máquina servidora e a transmissão física é feita após a mensagem chegar à camada mais baixa da máquina cliente, esse conceito ficará mais claro ao entender o funcionamento de cada camada.

Para que a mensagem enviada a outro computador seja interpretada, é necessário um cabeçalho indicando a qual processo ela é destinada. Caso a mensagem seja muito grande, ela pode chegar fragmentada, sendo necessário um índice indicando a sequência correta para montá-la. Esses são exemplos de soluções de problemas encontrados durante a transmissão, muitos outros ocorrem, porém seria necessário um trabalho dedicado apenas para este assunto. O que define se a mensagem será fragmentada ou quem constrói os cabeçalhos contendo as informações de envio são os protocolos, quando um protocolo executa uma ação sobre a mensagem que se encontra em uma camada x, diz-se que este é um protocolo da camada x (TANENBAUM, 2003).

Protocolo é um padrão utilizado para a transmissão de dados, com o propósito de que a máquina receptora possa entender a sequência de bytes que recebeu e forme a mensagem correta. Se todos os computadores de uma rede utilizarem o mesmo protocolo, será possível todos se comunicarem, pois enviam e entendem da mesma maneira, como pessoas que falam a mesma língua. Geralmente as camadas e protocolos de baixo nível estão implementados no hardware. Os de nível mais alto, implementados no sistema operacional, as chamadas a camada mais alta são efetuadas pelo processo através de APIs (*Application Programming Interfaces*) do sistema operacional. Todo esse conjunto de camadas e protocolos é chamado de arquitetura de rede (COMER, 1998; TANENBAUM, 2003) e é possível entendê-lo melhor visualizando a Figura 3.

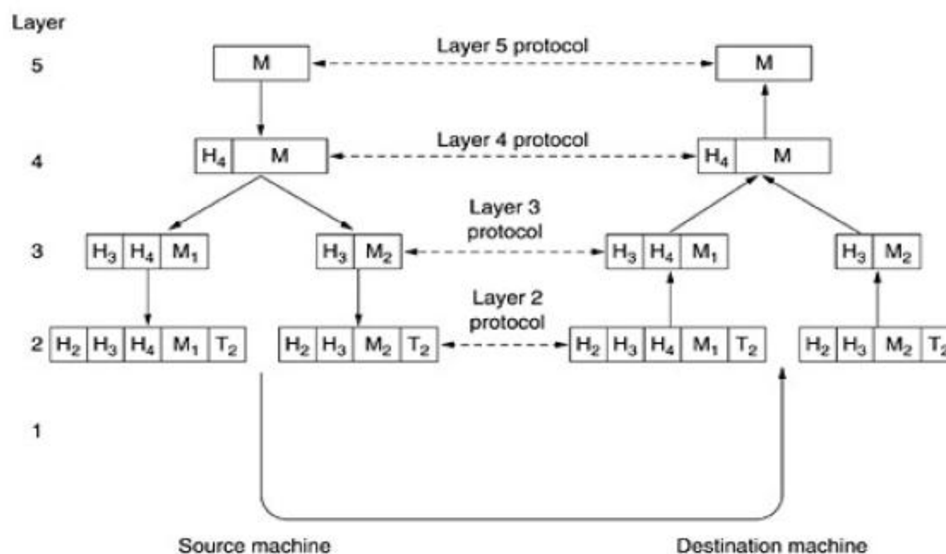


Figura 3 – Caminho e transformação de uma mensagem transmitida através das camadas
Fonte: Tanenbaum (2003, P. 32)

Na Figura 3, a mensagem M ganha um cabeçalho pelo protocolo da camada quatro, é dividida em dois pedaços pelo protocolo da camada três, recebe um rodapé pelo protocolo da camada dois, é transmitida pelo protocolo da camada um, e faz o caminho inverso ao atingir seu destino.

2.2.2 Modelos de Referência

2.2.2.1 Modelo OSI

O modelo OSI (*Open Systems Interconnection*) é a primeira tentativa de padronização dos protocolos de rede, ele contém sete camadas cuidadosamente analisadas a fim de separar o máximo possível cada nível de abstração, e o menor número de camadas possível para que a implementação não se torne muito complexa, porém, sem perder os níveis de abstração. Tanenbaum (2003) mostra os cinco critérios para escolha das camadas:

> 1. Uma camada deve ser criada onde houver necessidade de outro grau de abstração.
> 2. Cada camada deve executar uma função bem definida.
> 3. A função de cada camada deve ser escolhida tendo em vista a definição de protocolos padronizados internacionalmente.
> 4. Os limites de camadas devem ser escolhidos para minimizar o fluxo de informações pelas interfaces.

> 5. O número de camadas deve ser grande o bastante para que funções distintas não precisem ser desnecessariamente colocadas na mesma camada e pequeno o suficiente para que a arquitetura não se torne difícil de controlar.

As camadas que compõe o modelo OSI, da mais baixa para a mais alta são, respectivamente, camada física, de enlace de dados, de rede, de transporte, de sessão, de apresentação e de aplicação. As três camadas mais baixas são camadas encadeadas, por estarem presentes nos roteadores em que a mensagem passa antes de atingir seu destino, as quatro camadas mais altas são camadas fim a fim, por estarem implementadas apenas nas máquinas de origem e destino (TANENBAUM, 2003). A implementação das sete camadas do modelo são representadas pela Figura 4.

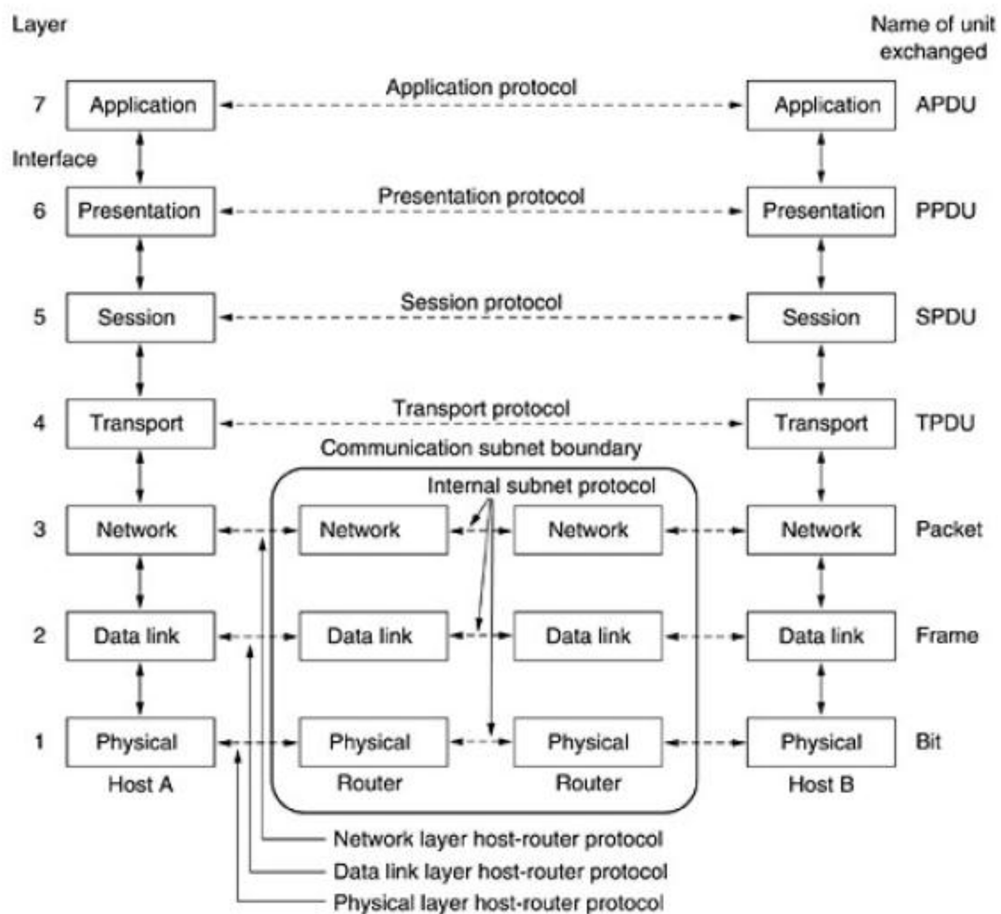


Figura 4 – Modelo OSI e as sete camadas que o compõe

Fonte: Tanenbaum (2003, P. 41)

Na Figura 4, as linhas pontilhadas representam as comunicações lógicas entre as camadas do Host A, Host B e os roteadores. Também se pode perceber

que apenas as três camadas mais baixas estão representadas nos roteadores, formando o encadeamento. As linhas sem o pontilhado entre as camadas são as interfaces de comunicação, citadas anteriormente. Dentre as sete camadas, é válido citar a função da camada de transporte, por ser a responsável por fragmentar e desfragmentar os dados e garantir que cheguem à outra extremidade da forma correta, sendo ela o elo principal de ligação entre duas máquinas, justamente por ser a primeira camada fim a fim.

2.2.2.2 Modelo TCP/IP

Este é o modelo utilizado na Internet, a maior rede de computadores existente, possui esse nome por causa de seus principais protocolos, o TCP e o IP, que serão detalhados adiante. O TCP/IP foi criado a partir do modelo OSI, pelo motivo de que o modelo usado até então, tornou-se incompatível com as novas tecnologias de hardware, e começaram a surgir muitos problemas em relação à comunicação. Era necessário um modelo que proporcionasse uma segurança a ponto de que a comunicação continuasse ativa mesmo que alguns dispositivos de hardware parassem de funcionar (COMER, 1998; TANENBAUM, 2003).

O TCP/IP possui quatro camadas, as camadas de apresentação e de sessão desapareceram do modelo, por serem consideradas pouco úteis. A camada de rede, no TCP/IP, foi modificada para camada internet que agora possibilita que pacotes sejam injetados em qualquer rede, podendo seguir várias rotas diferentes, o que resolve o problema de que alguns dispositivos de hardware parem de funcionar, essa característica é possível por não existir uma conexão estabelecida entre as redes. Essa camada possui um modelo de pacote padrão e seu protocolo é o IP (Internet Protocol), o roteamento de pacotes é muito importante nessa camada, pois sua responsabilidade é entregar pacotes IP onde eles são necessários. A camada de transporte é idêntica a do modelo OSI, sendo uma camada fim a fim e com a responsabilidade de que os hosts de origem e destino mantenham a comunicação, seus principais protocolos são o TCP e o UDP (COMER, 1998; TANENBAUM, 2003).

A camada de aplicação é a camada mais alta e permaneceu no modelo, a camada mais baixa é a camada *host*/rede e por não ser muito especificada no modelo diz-se que sua função é conectar o *host* a rede para que possa enviar

pacotes IP. A Figura 5 mostra as camadas do modelo TCP/IP comparadas com a do modelo OSI.

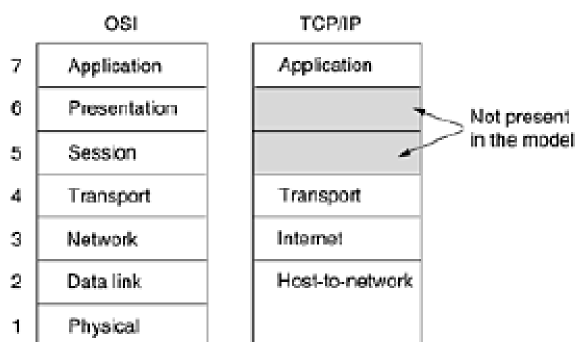


Figura 5 – Modelo TCP/IP comparado ao modelo OSI
Fonte: Tanenbaum (2003, P. 46)

Praticamente todas as redes utilizadas hoje em dia são construídas sobre o modelo TCP/IP, portanto, teremos o foco de estudo nos protocolos utilizados por esse modelo. O jogo *multiplayer* construído utiliza esses protocolos em seu funcionamento, e no decorrer da codificação serão demonstradas as ligações com o funcionamento dos protocolos a seguir.

2.2.3 Protocolo IP (*Internet Protocol*)

O IP é um protocolo da camada de Internet, que recebe pacotes da camada de transporte e os transforma em datagramas IP, esses datagramas são formados por um cabeçalho contendo informações de endereçamento e controle, e pelo corpo da mensagem. Dois dos campos do cabeçalho são reservados para armazenar os endereços IP de origem e destino do datagrama, esses endereços são formados por quatro pares de oito bits, e cada computador em uma rede possui um endereço IP, que é único na rede. Quando um computador deseja se comunicar com outro, utiliza o endereço correspondente ao computador destino e o insere no datagrama.

Cada byte do endereço IP é representado por um número de 0 a 255, e cada número é separado por um ponto, formando quatro octetos, como mostra o exemplo (1):

[1] **200.32.100.65**

Então, os endereços podem variar de 0.0.0.0 à 255.255.255.255, sendo que alguns endereços são reservados, como veremos adiante. As faixas de endereços são separadas em cinco classes, como podemos observar na Figura 6. Cada endereço é dividido em duas partes, onde dos 32 bits, uma quantidade representa o número da rede, e a outra o número do host, essa quantidade de bits é determinada pela máscara de sub-rede, que também contém 32 bits. A quantidade de bits 1 na máscara determina os bits da rede, e os bits 0 determinam os do host, por exemplo, uma máscara de sub-rede 255.0.0.0 diz que o primeiro octeto do endereço IP é o número da rede e o restante é o número do host, vejamos a representação em binário em (1):

[1] **255.0.0.0 = 11111111 00000000 00000000 00000000**

Logo, se o endereço IP 10.1.1.1 possuir essa máscara, diz-se que sua rede é 10.0.0.0 e que os três últimos octetos são o número do host. Repare que uma rede é representada por um número no octeto onde a máscara possui bits 1 e completada com 0 no restante, então, o primeiro endereço de uma rede é um dos endereços reservados. Outro endereço reservado é o último, por exemplo, 10.255.255.255, este é o endereço de broadcast, que significa todos os hosts da rede. Se um datagrama for enviado a esse endereço, será enviado a todos os computadores pertencentes àquela rede (COMER, 1998).

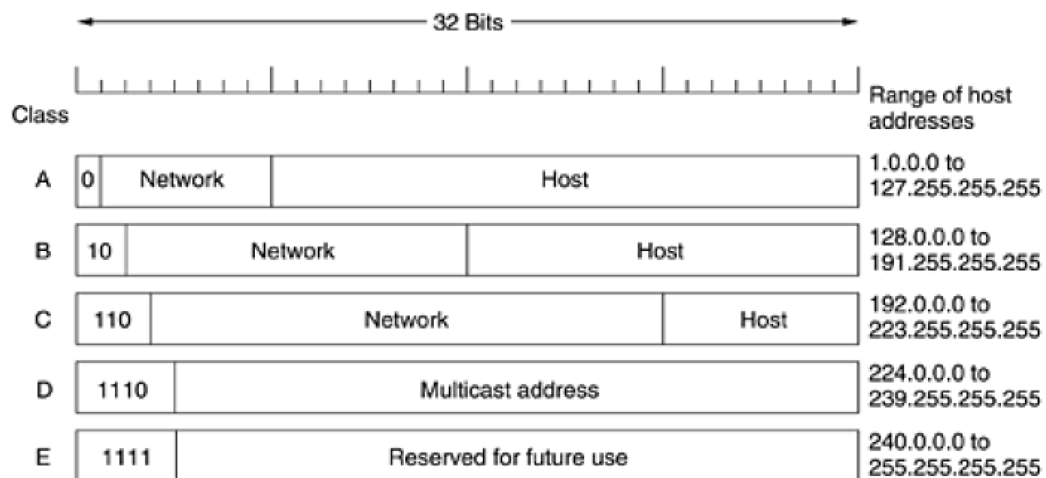


Figura 6 – Faixas do endereço IP e suas respectivas classes
Fonte: Tanenbaum (2003, P. 465)

As classes A, B e C são as utilizadas para as redes da Internet, e respectivamente possuem 8, 16 e 24 bits reservados para o número da rede, conforme a definição da máscara de sub-rede.

2.2.4 A Camada de Transporte

A função dessa camada é fornecer recursos de rede para os usuários, ou seja, para a camada de aplicação. Os usuários são os processos do sistema operacional e os recursos disponíveis são representados pelas primitivas do serviço de transporte, que serão descritas com detalhes ao falarmos dos *sockets*. A camada de transporte deixa transparente ao usuário as falhas que ocorrem na camada de rede, e implementam dois principais protocolos, o TCP (Transmission Control Protocol), orientado a conexão, e o UDP (User Datagram Protocol), que não utiliza conexões. Outra característica dessa camada é proporcionar que datagramas IP sejam entregues em aplicações distintas, processadas paralelamente no sistema operacional, através do conhecimento da porta de destino (COMER, 1998; STALLINGS, 2004; TANENBAUM, 2003).

2.2.4.1 Soquetes (*Sockets*)

Os *sockets* contêm um conjunto de primitivas utilizadas no serviço de transporte, essas primitivas são utilizadas para obter uma conexão e transferir dados entre as camadas de transporte dos hosts de origem e destino (TANENBAUM, 2003). A Figura 7 mostra as primitivas e seus significados.

Praticamente todas as aplicações que precisam se comunicar com outro computador utilizam os *sockets* da camada de transporte, toda a Internet funciona dessa maneira, portanto, o jogo *multiplayer* implementado neste estudo utilizará *sockets* para comunicação entre o cliente e o servidor, onde as primitivas serão chamadas através de métodos no código python, deixando mais claro a teoria e a utilização destas primitivas.

Primitiva	Significado
SOCKET	Criar um novo socket, que será utilizado para comunicação
BIND	Anexa um endereço local a um socket, no servidor, a porta
LISTEN	Anuncia a disposição para aceitar conexões
ACCEPT	Bloqueia o responsável pela chamada até uma tentativa de conexão ser recebida
CONNECT	Tenta estabelecer uma conexão ativamente
SEND	Envia dados através de uma conexão ativa
RECEIVE	Recebe dados de uma conexão ativa
CLOSE	Encerra a conexão

Figura 7 – Primitivas de *Sockets* utilizadas no protocolo TCP
Fonte: Adaptado de Tanenbaum (2003)

A Figura 7 mostra as primitivas utilizadas no protocolo TCP, também existem as primitivas utilizadas pelo protocolo UDP, são as mesmas, porém algumas delas não aparecem, por ser um protocolo que não é orientado a conexões (COMER, 1998).

Para que ocorra uma conexão, é necessário que o cliente e o servidor criem um *socket* e utilizem a diretiva BIND para endereçamento, porém, o servidor será passivo e utilizará as primitivas LISTEN e ACCEPT para aceitar conexões dos clientes. Os clientes por sua vez serão ativos e utilizarão a primitiva CONNECT para se conectarem ao servidor que espera por conexões. Após a conexão estabelecida, o cliente e o servidor poderão utilizar as primitivas SEND e RECEIVE para enviar e receber dados, sendo que, para utilizar SEND o servidor deverá estar bloqueado com RECEIVE.

Os dois principais tipos de *socket* são o DATAGRAM_*SOCKET*, que utiliza o protocolo UDP para transporte e o STREAM_*SOCKET*, que utiliza o protocolo TCP para transporte (TANENBAUM, 2003).

2.2.4.2 Protocolo UDP

O protocolo UDP é utilizado para transporte de dados sem conexão, portanto não oferece a garantia de entrega e nem a entrega de dados na sequência correta. Ao utilizar o UDP, o responsável por esse tipo de controle é a aplicação do usuário, porém ao utilizar esse protocolo, há um ganho de velocidade na transmissão de dados (COMER, 1998; STALLINGS, 2004).

Tanto o UDP quanto o TCP necessitam de uma porta de destino e uma porta de origem para enviar dados, essas portas são representadas por uma sequência de 16 bits, ou seja, um número inteiro que varia de 0 a 65535. A porta de origem é utilizada para que o processo receptor envie uma resposta ao processo emissor.

Utilizando UDP as primitivas LISTEN, ACCEPT, CONNECT e CLOSE não aparecem, pois são primitivas utilizadas na obtenção e fechamento de conexões (COMER, 1998).

Para utilizar esse protocolo em uma linguagem de programação, deve-se procurar a forma de declaração do DATAGRAM_*SOCKET*, como será visto ao estudarmos a linguagem Python.

2.2.4.3 Protocolo TCP

O TCP utiliza conexão para envio de dados, por essa razão é considerado confiável e tem garantia de entrega do pacote, também garante a sequência correta dos dados que chegam ao receptor. Essa garantia de entrega é possível pelo fato de que o receptor envia uma confirmação, chamada de ACK, ao emissor, sendo que este só envia o próximo pacote ao receber a confirmação, caso isso não aconteça, o pacote é retransmitido. Essa técnica pode ser considerada segura para garantir a sequência e a chegada de informações, porém, deixa a transferência mais lenta, devido ao maior fluxo de dados pela rede (COMER, 1998).

As primitivas para utilização deste protocolo estão descritas na Figura 7. Ao utilizar esse protocolo em uma linguagem de programação, deve-se procurar pela declaração do STREAM_*SOCKET*, sendo assim, utilizando o TCP podemos dizer que estamos efetuando uma transmissão de dados *stream* (COMER, 1998).

2.2.4.4 Pilha de Protocolos

Para finalizar a questão do funcionamento das camadas e protocolos de comunicação na rede, é necessário entender a estrutura da pilha de protocolos. Sendo que cada camada possui seus protocolos, e as camadas estão situadas uma acima da outra na hierarquia dos modelos, o conjunto de protocolos por onde as

mensagens passam, recebem modificações e são adicionados cabeçalhos, é chamado de pilha de protocolos (TANENBAUM, 2003).

A mensagem ao sair da camada de aplicação, é tratada pelo protocolo da camada de transporte, geralmente TCP ou UDP, recebe um cabeçalho desses protocolos contendo a porta de destino e a porta de origem, e recebe o nome de pacote ou datagrama, respectivamente. O pacote ou datagrama já pode ser tratado pelo protocolo IP, da camada de Internet, e recebe mais um cabeçalho, contendo os endereços IP de origem e destino, e agora é chamado de datagrama IP. Os dados estão prontos para serem enviados pelo meio físico da rede, onde serão tratados pela camada de host/rede e podem ser quebrados em pedaços chamados quadros ou frames, recebendo mais um cabeçalho pelo protocolo desta camada (COMER, 1998). A mensagem que transita pela rede, após passar pela pilha de protocolos, onde no final é chamada de quadro ou frame pode ter sua estrutura visualizada na Figura 8.

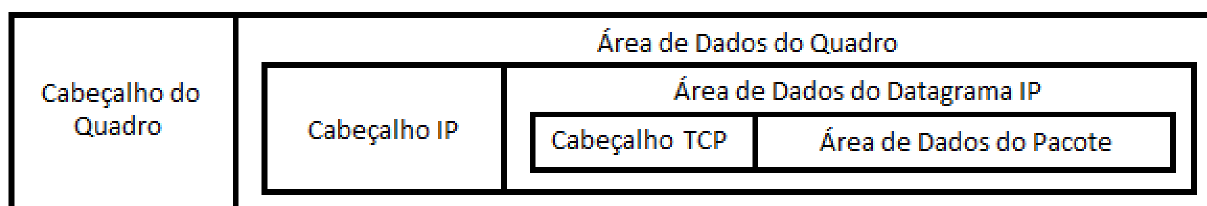


Figura 8 – Estrutura de um frame que transita pela rede, contendo um pacote TCP
Fonte: Adaptado de Comer (1998)

Analisando a Figura 8, percebe-se a quantidade de informações adicionadas para possibilitar o transporte de uma mensagem, apenas a área de dados do pacote será utilizada pela aplicação. Essas informações que não fazem parte do corpo da mensagem são chamadas de *overhead*.

2.3 ARQUITETURA CLIENTE-SERVIDOR

Essa arquitetura possibilita que dois processos executados em máquinas distintas enviem mensagens um para o outro, e para isso, um processo deve ser o servidor, que estará escutando em uma determinada porta e o outro processo, o cliente, o qual enviará uma mensagem ao servidor através de algum protocolo

(TANENBAUM, 1995). Para que o cliente envie com sucesso uma mensagem ao servidor, este deve necessariamente estar escutando em alguma porta. A Figura 9 mostra a arquitetura cliente-servidor e as primitivas de *socket* utilizadas com o protocolo TCP durante a comunicação entre os processos remotos.

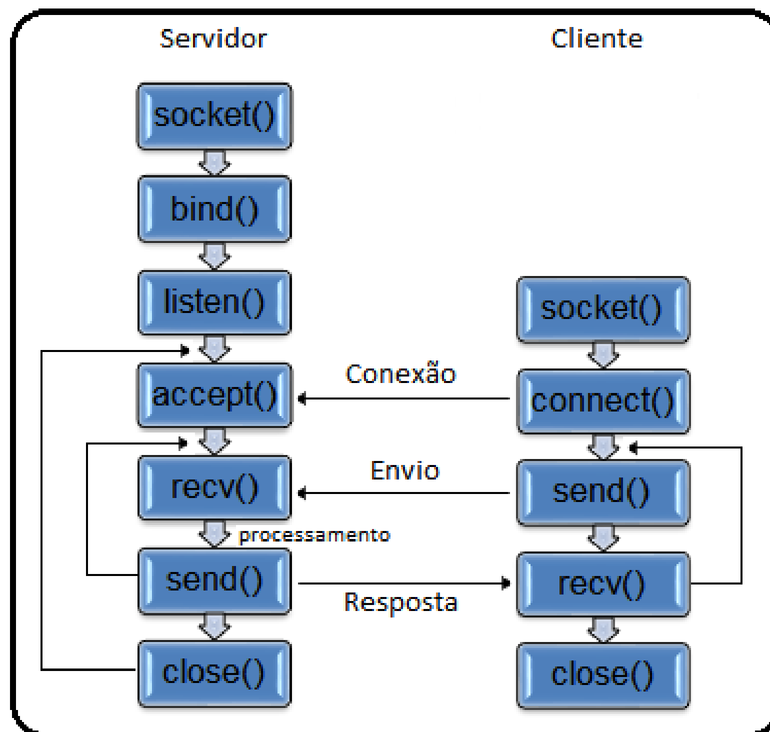


Figura 9 – Comunicação entre processos no modelo cliente-servidor
Fonte: Adaptado de Góis (2011)

Utilizando essa arquitetura, possibilita-se que processos remotos respondam requisições de outros processos (SHAY, 1996; TANENBAUM, 1995), como pode ser visualizado na Figura 9. Depois de criado o *socket* do servidor, o mesmo utiliza LISTEN() e ACCEPT(), e o cliente utiliza CONNECT(). Após, os processos trocam mensagens por SEND e RECV() até a conexão ser encerrada com CLOSE(). Repare que o servidor volta em ACCEPT() para tratar novas conexões.

2.3.1 *Threads*

Para que o servidor possa receber e processar uma mensagem, utiliza-se a primitiva RECV(), porém, esta é uma primitiva bloqueante, ou seja, o processo fica

bloqueado até que uma mensagem chegue pela porta que está escutando, dessa forma é possível que apenas um cliente seja atendido. Para que outro possa interagir com o servidor, este deverá esperar o outro terminar sua conversação.

Utilizando *threads* (SILBERS*CHATZ*, 2004) é possível resolver este problema, pois os processamentos das primitivas bloqueantes ficam separados cada um em uma *thread* distinta, e o processamento principal fica encarregado apenas de receber novas conexões e criar uma nova *thread* para tratá-la. A Figura 10 mostra o modelo cliente-servidor utilizando *threads* para tratar vários clientes ao mesmo tempo.

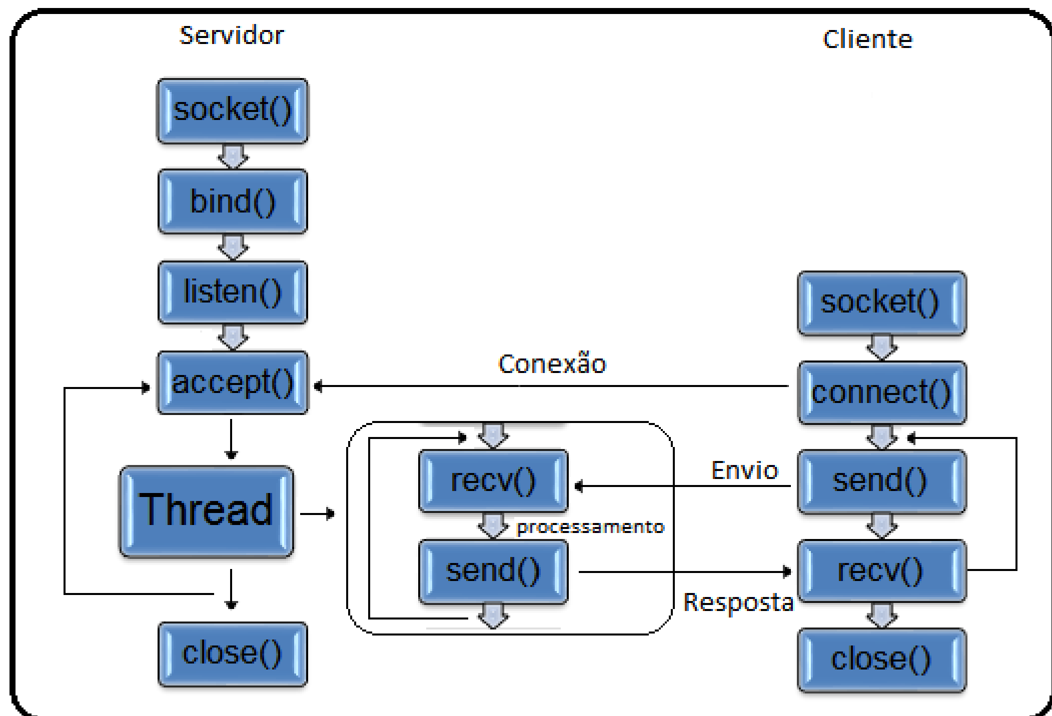


Figura 10 – Modelo cliente-servidor utilizando *thread* para múltiplos clientes
Fonte: Adaptado de Góis (2011)

Analisando a Figura 10, pode-se entender que uma *thread* é uma parte de um processo que pode ser processada paralelamente à parte principal. Um processo pode possuir muitas *threads* paralelas, e com isso, a *thread* fica bloqueada em RECV() aguardando a mensagem do cliente, enquanto o processo principal retorna à ACCEPT() aguardando novas conexões e obtendo processamento paralelo. É possível tal recurso, pois o processamento das *threads* é escalonado pelo sistema operacional (SHAY, 1996).

2.3.2 Problemas de Sincronia e Uso de *Multi-threads*

É totalmente inviável a construção de um sistema distribuído sem o uso de *threads*, pois enquanto escuta uma porta, o processamento deve continuar e controlar o restante dos clientes conectados. O uso de multi-*threads* deve ser rigorosamente analisado antes da implementação, uma vez que são processadas paralelamente e não se sabe o momento em que o sistema operacional irá escalonar seu processamento, também não se sabe qual das *threads* irá acessar um método primeiro (SHAY, 1996; TANENBAUM, 1995). Se duas *threads* utilizam o mesmo método de uma classe, e as duas acessarem ao mesmo tempo, erros graves de sincronia podem ocorrer, então, é aconselhável tratar essa sincronia ao programar sistemas distribuídos, uma das formas é impossibilitar que uma *thread* acesse o método se outra estiver utilizando. Algumas linguagens de programação como Java já vem com esse tipo de funcionalidade, onde utiliza-se a palavra reservada *synchronized* antes da declaração do método.

Um jogo *multiplayer* deve utilizar múltiplas *threads* para controlar os clientes, e tratamentos para evitar erros de sincronia devem ser implementados. Caso ocorra um erro desse gênero durante a execução do jogo, é possível até que o servidor trave e o jogo necessite ser reiniciado.

2.4 LINGUAGEM DE PROGRAMAÇÃO PYTHON

Python (BORGES, 2010) é uma linguagem muito poderosa, de fácil aprendizagem pela simplicidade de seu uso. É possível utilizá-la para grandes projetos ou apenas em partes de um projeto (MENEZES, 2010). Para o desenvolvimento do jogo *multiplayer* foi escolhida a linguagem Python, juntamente com o *engine* Pygame (PYGAME.ORG, 2010) para auxiliar na programação do jogo. É considerada uma linguagem de alto nível por ser clara e objetiva na hora da programação, uma de suas simplicidades é a alocação dinâmica de variáveis, ou seja, não é necessário declarar as variáveis, pois ao utilizá-las, o espaço em memória é alocado em tempo de execução. Essa característica pode parecer muito prática, porém deixa mais trabalhoso encontrar erros que envolvam variáveis locais e globais (TANENBAUM, 1995).

Outra característica da linguagem é que por possuir declaração dinâmica, é possível armazenar qualquer tipo de dado nas variáveis, pois o Python aloca dinamicamente o tamanho necessário de memória para a variável. Porém, como no problema anterior de alocação dinâmica, alocar dinamicamente o espaço necessário também gera um problema ao encontrar erros de tipos incompatíveis de dados.

Python é uma linguagem interpretada, ao contrário de outras linguagens que são compiladas, como a linguagem C. Entende-se por linguagem interpretada aquela que necessite de um software que interprete seu código, que são as máquinas virtuais. A linguagem Java utiliza uma máquina virtual para seu funcionamento, como o Python necessita da máquina virtual Python para interpretar seu código. Apesar disso, existem *plug-ins* que compilam o código Python para que obtenha a extensão EXE, e possa ser executado no Windows sem a máquina virtual.

Por fim, é interessante ter o conhecimento da forma de identação da linguagem, pois Python interpreta o início de um bloco de código pelo símbolo de dois pontos (“:”), e todo o código no interior deste bloco deve estar na mesma identação. Python interpreta o fim do bloco assim que encontrar um recuo de identação. Por não utilizar ponto e vírgula no fim dos comandos e utilizar a identação como forma de início e fim de blocos, é necessária grande atenção no momento da programação.

3 DESENVOLVIMENTO

Utilizando as tecnologias vistas anteriormente, desenvolveu-se uma aplicação distribuída utilizando uma *engine* gráfica na linguagem de programação Python, visando estabelecer uma comunicação viável entre os processos remotos e de modo que possua o menor tráfego de rede possível, onde possibilitou-se uma maior realidade e sincronia no jogo *multiplayer*. A seguir serão abordados os estudos e a sequência de implementações desenvolvidas até que se alcançasse o software final. Os exemplos de código estão na linguagem Python versão 3.2, pelo fato da aplicação ser construída em tal linguagem, possibilitando a exemplificação de sua sintaxe e uso na programação de sistemas.

3.1 LINGUAGEM PYTHON E BIBLIOTECA GRÁFICA PYGAME

3.1.1 Apresentação do *Engine*

Pygame é uma biblioteca gráfica para desenvolvimento de jogos 2D utilizando a linguagem de programação Python. Pygame não é incluído com o pacote básico do Python, é necessário efetuar o download da versão do Pygame desejada e instalar em um computador que possua o Python instalado.

Esse *engine* é de fácil aprendizado, por possuir funções simples que controlam praticamente todos os dispositivos de entrada e saída que um computador possui, como som, mouse, teclado e vídeo. O controle do vídeo é a parte mais importante de um jogo, e o Pygame possui funções prontas para desenhar formas geométricas, movimentar objetos na tela e gerenciar cores.

3.1.2 Exemplos de Implementação

Antes de projetar um jogo *multiplayer* para ser desenvolvido em Python e Pygame, é necessário estudar a linguagem e o motor, principalmente procurar conhecer funções importantes que ajudem no desenvolvimento.

Alguns exemplos de pequenos aplicativos foram desenvolvidos para aprendizagem, e serão apresentados a seguir.

3.1.2.1 Bolas quicando

Um exemplo simples do uso do Pygame que envolve cálculos de física, é a implementação de duas bolas quicando na tela e se colidem quando chegam perto uma da outra, invertendo a direção em que se tocaram. Além dos cálculos efetuados, é feito o controle do teclado para mudar a direção de uma das bolas e também modificar a força com que incide no canto inferior da tela, fazendo com que a bola quique mais alto ou mais baixo. A Figura 11 mostra a imagem da aplicação.

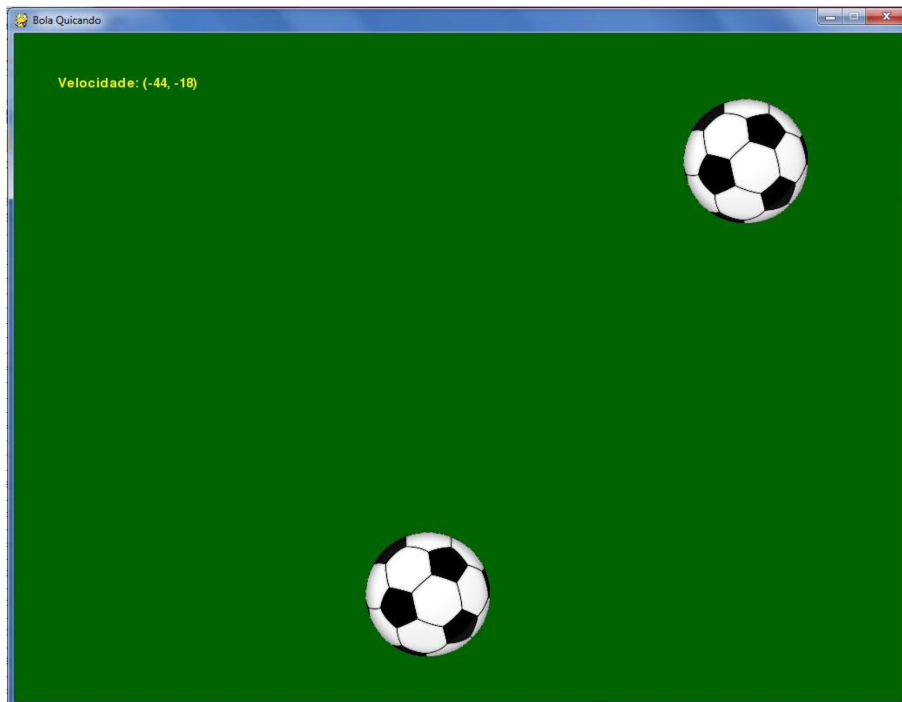


Figura 11 – Exemplo de aplicação que implementa duas bolas quicando
Fonte: Autoria Própria

Para o controle da gravidade é necessário que a velocidade diminua conforme a bola sobe, e fique negativa para que a bola desça. Assim que toca o chão, a velocidade perde o sinal negativo e a velocidade se inverte para que a bola suba novamente, como pode ser visualizado no trecho de código da Figura 12.

```
# aumenta a velocidade vertical das bolas (positivo desce e negativo sobe)

    speed_v = (speed_v[0]+2, speed_v[1]+2)

# faz as bolas quicarem (inverte a velocidade e adiciona atrito
# quando chega no chão)

    if x[0] >= size[1]-ball_rect[0][2] :
        speed_v = (speed_v[0] - (speed_v[0]*2)+2, speed_v[1]) # bola 1

    if x[1] >= size[1]-ball_rect[1][2] :
        speed_v = (speed_v[0], speed_v[1] - (speed_v[1]*2)+2) # bola 2
```

Figura 12 – Trecho de código python que controla a velocidade vertical da bola quicando
Fonte: Autoria Própria

Na linha 3, *speed_v* é a velocidade atual da bola, na linha 8, *ball_rect* são as coordenadas da bola, e *x* é a posição que indica que a bola encostou no solo.

3.1.2.2 Bee Invader

Outro exemplo implementado foi o Bee Invader, onde o jogador controla uma nave que possui uma arma que atira verticalmente, e para atirar é necessário esperar a barra de energia se encher. É possível mover-se lateralmente com a nave na tela, e disparar a qualquer momento desde que haja energia, os disparos servem para destruir as abelhas que surgem aleatoriamente na tela. Uma contagem de abelhas perdidas e abelhas destruídas são apresentadas no canto superior da tela. A Figura 13 mostra a imagem do exemplo.

Esse exemplo implementa colisão do disparo com as abelhas, onde no local surge uma Figura de uma explosão e contabiliza as abelhas destruídas, além, utiliza sons no momento da explosão.

Figura 13 – Bee Invader

Fonte: Autoria Própria

3.2 PYTHON E A ORIENTAÇÃO A OBJETOS

3.2.1 Tecnologia Orientada a Objetos

A orientação a objetos é uma tecnologia que possibilita várias características durante a programação, como reuso de código, e facilidade de manutenção caso ocorra algum erro.

Dentre os conceitos de orientação a objetos, se destacam as classes, que são estruturas de dados pré-definidos que contêm atributos e métodos, os atributos são variáveis da classe e os métodos são as funções que tal classe pode executar. Uma classe pode ser instanciada, criando um objeto da classe, muitos objetos podem ser instanciados a partir da mesma classe, e todos possuirão os atributos e métodos predefinidos. Os objetos podem se comunicar com outros objetos, basta o mesmo conhecer ou instanciar um objeto de outra classe (BOOCH, 2000; RUMBAUGH, 1994).

Além do conceito de classe, outros conceitos são muito úteis na orientação a objetos, como herança, que possibilita o reuso de código, e classes abstratas, que possibilitam que as subclasses que a herdem, reescrevam os métodos abstratos com implementações diferentes em cada subclasse (RUMBAUGH, 1994).

Com a orientação a objetos é possível abstrair problemas do mundo real através dos diagramas da UML. No desenvolvimento de um jogo, é possível representá-lo através desses diagramas antes da programação. Aplicar orientação a objetos em seu desenvolvimento facilita muito no que se diz abstração, pois os objetos serão realmente os objetos que se vê na tela, além de outros tipos de objetos abstratos, como uma classe que controla a rede ou o teclado.

3.2.2 Implementando Orientação a Objetos em Python

Python implementa todos os conceitos de orientação a objetos, portanto, o uso da linguagem para programação de jogos é uma boa escolha. Um conceito que nem todas as linguagens orientadas a objetos implementam e Python implementa é a herança múltipla (MENEZES, 2010), muito útil em alguns casos no interior dos jogos.

Um exemplo de abstração utilizando a orientação a objetos em Python está descrito na classe da Figura 14, onde representamos uma abelha que aparece na tela de um jogo. A classe pode ser instanciada e criar um objeto da classe Bee.

```
1  class Bee(Image):
2      def __init__(self, image, x, y, speed):
3          Image.__init__(self, image, x, y)
4          self._speed = speed
5
6      def move(self, direction):
7          if(direction == "RIGHT"):
8              self._position = (self._position[0]+self._speed, \
9              self._position[1], self._position[2], self._position[3])
10         elif(direction == "LEFT"):
11             self._position = (self._position[0]-self._speed, \
12             self._position[1], self._position[2], self._position[3])
13         elif(direction == "UP"):
14             self._position = (self._position[0], self._position[1]-self._speed, \
15             self._position[2], self._position[3])
16         elif(direction == "DOWN"):
17             self._position = (self._position[0], self._position[1]+self._speed, \
18             self._position[2], self._position[3])
19
20     def increaseSpeed(self, value):
21         if self._speed < 20 :
22             self._speed += value
23
24     def decreaseSpeed(self, value):
25         if self._speed > 1 :
26             self._speed -= value
```

Figura 14 – Classe Bee, representando a abstração de uma abelha
Fonte: Autoria Própria

O método *__init__* representa o construtor, e contém os atributos da classe, no caso da classe *Bee* temos o atributo *speed*. Essa classe herda a classe *Image* e chama o construtor da superclasse ao ser instanciada. Os métodos da classe, que farão com que a abelha se mova e mude sua velocidade na tela são: *move()*, *increaseSpeed()* e *decreaseSpeed()*.

3.3 *THREADS* E *SOCKETS* EM PYTHON

3.3.1 Implementação de *Threads*

Em Python, existem várias formas de se utilizar *threads*, todas acabam por obter o mesmo resultado, o processamento paralelo. A única diferença é a forma de se escrever o código, onde é possível herdar a classe *Thread* e sobrescrever o método *run()*, ou utilizar diretamente o método *start_new_thread()* da biblioteca *thread*, onde é possível escolher qualquer método de uma classe e transformá-lo em

uma *thread* (BORGES, 2010). As Figuras 15 e 16 demonstram a diferença entre as duas formas descrita.

```
1   thread.start_new_thread(self.waitMessage, tuple([]))
2
3   def waitMessage(self):
4       while True:
5           position = self._connection.recv(4096)
6           self._position[0] = position
7           self._connection.send("ok")
8           ok = self._connection.recv(4096)
9           self.sendPosition()
```

Figura 15 – Criação de *thread* pelo método start_new_*thread*()

Fonte: Autoria Própria

```
1   class Connection(threading.Thread):
2       def __init__(self, host):
3           self._host = host
4           threading.Thread.__init__(self)
5
6       def run(self):
7           while True:
8               connection, client = self._tcp.accept()
9               self.add_client(connection, client)
```

Figura 16 – Herança da classe *Thread*

Fonte: Autoria Própria

Ao utilizar a herança da classe *Thread* em uma classe para transformá-la em uma *thread*, é necessário chamar o construtor da superclasse e sobrescrever o método *run()*, como podemos observar na Figura 16. Essa foi a forma escolhida para implementar as *threads* do jogo *multiplayer*, por deixar o código mais organizado e ter uma melhor abstração por utilizar as classes da orientação a objetos.

3.3.2 Implementação de *Sockets*

Um *socket* pode ser executado em dois estados distintos, ativo e passivo. Quando passivo fica "escutando" a porta escolhida e esperando por novas conexões, geralmente utilizado na implementação de servidores que controlam as

conexões de seus clientes, e quando ativo, envia requerimentos de uma nova conexão, no caso, para um servidor que possui um *socket* passivo.

Os *sockets* em Python são criados como mostra a Figura 17, diferenciando-se ao utilizar protocolos diferentes, como o TCP e UDP, respectivamente representados pelas constantes SOCK_STREAM e SOCK_DGRAM. Para utilizar tal recurso é necessário importar a biblioteca *socket*.

```
1  import socket
2  addr = ("10.1.1.1",3454)
3  _tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4  _tcp.bind(addr)
5  _tcp.listen(1)
6  connection, client = self._tcp.accept()
```

Figura 17 – Criação e primitivas de *socket* passivo com protocolo TCP
Fonte: Autoria Própria

3.3.3 O Uso Conjunto de *Threads* e *Sockets*

Para que um servidor de um sistema distribuído seja funcional, ou seja, possa aceitar conexões de vários clientes paralelamente, é crucial o uso de multi-*threads* (EICHELT, 2011) juntamente com os *sockets*, pois a cada cliente conectado, é necessário que uma nova conexão seja criada, essas conexões são estabelecidas nas *threads*, as quais controlam seus respectivos clientes, enviando e recebendo as mensagens do mesmo. Ao receber uma mensagem, é efetuado o devido processamento para que seja enviada a resposta correta.

O controle das *threads* deve ser feito organizadamente, pois se as referências às *threads* se perdem, é possível que os clientes se desconectem ou ocorram falhas de conexão, sendo necessário reiniciar a aplicação. Outro problema devido a essa referência consiste no momento de se utilizar muitas instâncias paralelas ao mesmo tempo, pois se não forem corretamente identificadas na sua criação, o programador poderá ter problemas para encontrar a *thread* correta. O ideal é inserir todas as *threads* dentro de um vetor ou separá-las por tipo em vetores diferentes, e não esquecer de renomear as instâncias, se necessário.

A respeito das conexões estabelecidas com os clientes, as *threads* deverão sempre estar à espera de novas mensagens. Ao recepcionar a mensagem,

o *socket* retorna à primitiva RECEIVE(). Paralelamente às *threads* que controlam os clientes, o servidor deverá estar sempre aguardando por novas conexões, e ao estabelecer uma conexão e criar uma nova *thread*, o *socket* retorna à primitiva ACCEPT(). A implementação de um *chat* é um exemplo simples da utilização de *threads* e *sockets* para controle de múltiplos clientes pelo servidor.

3.4 IMPLEMENTAÇÃO DE UM *CHAT* SIMPLES

3.4.1 Métodos e Arquitetura Utilizada

O *chat* desenvolvido é composto por uma aplicação servidora executada em um computador da rede, e uma aplicação cliente que pode possuir várias instâncias executando em computadores da mesma rede que a aplicação servidora. Ao conectar vários clientes, é criado um ambiente de uma sala de conversa entre os usuários. O ambiente de programação distribuída é transparente aos usuários.

Para executar o cliente, é necessário que o servidor já esteja executando e digitar o IP e a porta que o mesmo esteja aguardando por conexões. Ao estabelecer a conexão, torna-se possível o envio de mensagens no console, que serão exibidas no console do servidor e de todos os outros clientes conectados.

A cada cliente, o servidor gera uma nova *thread* e uma nova conexão de *socket* do tipo STREAM, que utiliza o protocolo TCP. A Figura 18 mostra o console de envio de mensagens das aplicações clientes e da aplicação servidora.

Cliente 1

C:\Python32\python.exe

```
Digite seu nome: Cliente 1
Conectado ao Servidor!
Cliente 2 conectou-se...
Olá a todos
Cliente 2 diz: Olá, como vai?
Bem conectado!
Cliente 2 diz: Eu também =)
Cliente 2 diz: Vou sair!
Adeus!
```

Servidor

C:\Python32\python.exe

```
Bem vindo ao Servidor!!!
Cliente 1 conectou-se...
Cliente 2 conectou-se...
Cliente 1 diz: Olá a todos
Cliente 2 diz: Olá, como vai?
Cliente 1 diz: Bem conectado!
Cliente 2 diz: Eu também =)
Cliente 2 diz: Vou sair!
Cliente 1 diz: Adeus!
```

Cliente 2

C:\Python32\python.exe

```
Digite seu nome: Cliente 2
Conectado ao Servidor!
Cliente 1 diz: Olá a todos
Olá, como vai?
Cliente 1 diz: Bem conectado!
Eu também =)
Vou sair!
Cliente 1 diz: Adeus!
```

Figura 18 – Console das aplicações de dois clientes e do servidor do *chat*

Fonte: Autoria Própria

3.4.2 Processo de Conexão de Novos Clientes

O servidor do *chat* utiliza *sockets* do tipo STREAM e aguarda por conexões de clientes. A cada cliente que requisita uma conexão, é criada uma nova *thread* para controle do mesmo, a Figura 19 mostra um diagrama de sequência que simplifica o a criação das *threads* do servidor a partir das requisições de conexão por parte dos cliente.

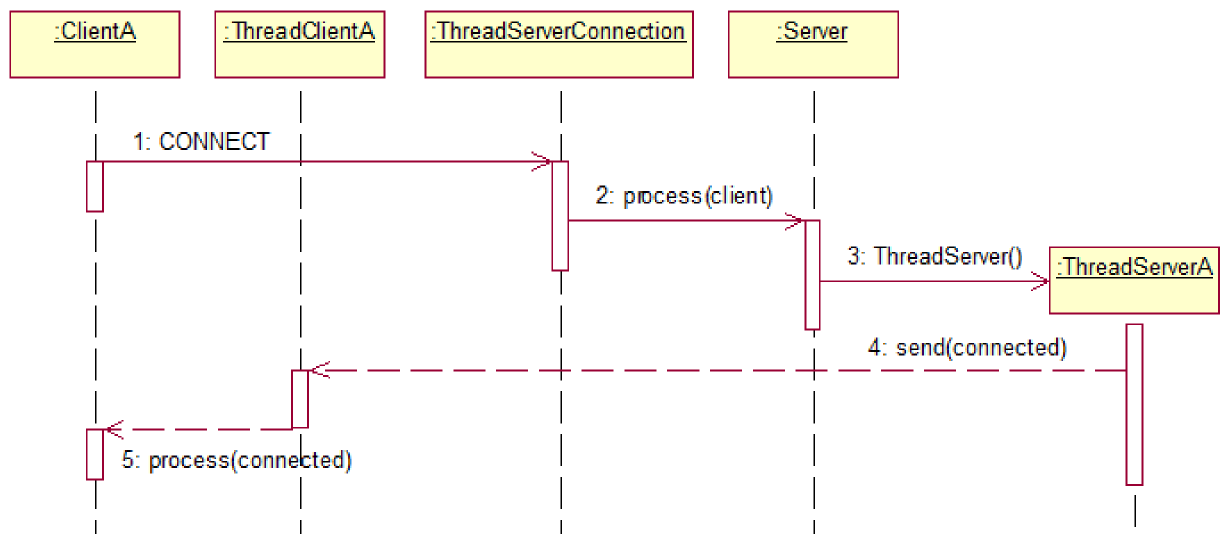


Figura 19 – Criação das *threads* do servidor ao receber conexões de novos clientes
Fonte: Autoria Própria

O objeto representado por *ThreadClientA* representa a *thread* que controla as mensagens recebidas pelo cliente A, já o objeto representado por *ThreadServerConnection* representa a *thread* presente no servidor que aguarda por novas conexões de clientes. Ao receber uma requisição de um novo cliente, o servidor cria a nova *thread* para controlá-lo, representada por *ThreadServerA*, na Figura 18.

3.4.3 O Redirecionamento de Mensagens

O cliente é formado por um processo principal, o qual habilita o console para envio de mensagens, porém, há uma *thread* que aguarda por mensagens que venham do servidor, mensagens essas que vieram de outros clientes.

A sequência de mensagens enviadas e recebidas descritas no diagrama da Figura 20 mostra que o servidor serve como redirecionador de mensagens, pois ao receber uma mensagem de um cliente, redireciona para os outros.

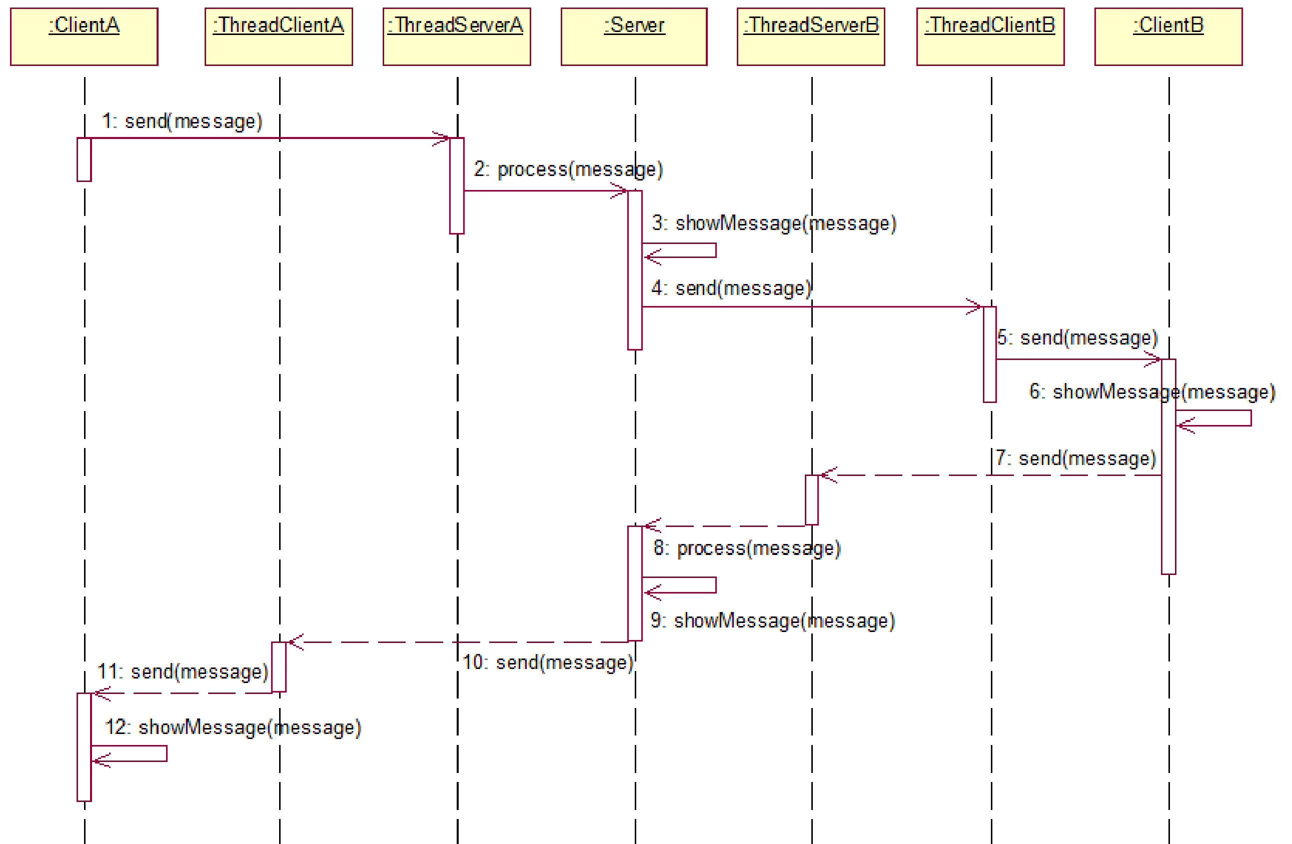


Figura 20 – Envio e redirecionamento de mensagens entre os clientes do *chat*

Fonte: Autoria Própria

Além do redirecionamento, o diagrama descreve o processo de envio de uma mensagem do cliente A para o ambiente do *chat*, e após, o envio de uma mensagem pelo cliente B. Nos passos 3; 6; 9; e 12, a mensagem é impressa no console da aplicação onde ela se encontra.

3.4.4 Arquitetura do *Chat* em Jogos *Multiplayer*

A arquitetura e tecnologias utilizadas para desenvolvimento do *chat* será utilizada para o desenvolvimento do jogo *multiplayer*. Porém, o *chat* envia mensagens digitadas pelos usuários, e no jogo, será abordado o desenvolvimento

de um padrão de mensagens, a fim de melhorar o desempenho e diminuir o tráfego de rede, além da utilização da *engine* Pygame para controlar a parte gráfica do jogo.

3.5 UTILIZANDO A REDE NO DESENVOLVIMENTO DE JOGOS

3.5.1 Problemas Encontrados ao Utilizar a Rede em Jogos

No projeto de um jogo *multiplayer*, deve-se atentar na escolha das informações que serão enviadas pela rede. Uma análise irregular pode acarretar em uma série de problemas de desempenho, como a lentidão, posições erradas dos jogadores e objetos, e até o travamento do jogo. Um dos principais problemas é possibilitar que todos os jogadores conectados vejam a posição correta e o deslocamento em tempo real dos outros jogadores (PICOLI, 2011).

A respeito das informações enviadas entre os processos remotos do jogo, devem-se selecionar apenas as mais relevantes para envio, pois o mínimo possível deve trafegar, a fim de diminuir o tráfego e sobrecarga do servidor. Em jogos, as informações úteis são aquelas que as outras aplicações clientes não têm condições de saber ou deduzir sem que receba a informação pela rede.

Além de projetar um padrão de dados selecionados para envio, outro fator que influencia o desempenho do jogo são erros de rede, onde pacotes podem se perder ou demorar a chegar ao destino, ocasionando a lentidão do jogo e fazendo com que as imagens não se posicionem nas coordenadas corretas. Perdas de pacotes são inevitáveis, porém, o tratamento em caso de falhas é a solução para o problema.

3.5.2 Possíveis Soluções aos Problemas Encontrados

O padrão de mensagens desenvolvido para diminuir o tráfego de informações entre os processos remotos foi um aprimoramento de outro padrão testado anteriormente. O padrão de envio de coordenadas dos objetos foi o pioneiro, porém esse padrão não obteve êxito, sendo substituído pelo padrão de envio das ações dos jogadores. A seguir são demonstrados os dois padrões propostos.

3.5.2.1 Envio das coordenadas dos objetos

Uma solução encontrada para fazer com que objetos remotos se posicionassem corretamente, foi enviar as coordenadas dos objetos em uma mensagem (PICOLI, 2011), e os clientes possuíam em cada objeto, o endereço IP ao qual pertenciam. Quando uma nova atualização de coordenada chegava pela rede, atualizava-se a imagem de determinado usuário pelo IP do pacote, utilizando as novas coordenadas.

A imprevisibilidade das ações do jogador, como movimentos repentinos e mudanças de direções podem se tornar um problema para que seus movimentos e posições sejam visualizados corretamente pelos outros jogadores. Para obter sensação de tempo real enviando as coordenadas, seria necessário enviá-las em um espaço muito curto de tempo, preferencialmente a cada impressão de imagem na tela, ou seja, se o FPS (*frames* por segundo) for 30, essa seria a quantidade de mensagens enviadas, o que ocasionaria um tráfego enorme na rede. Outra questão seria se essas mensagens não chegassem exatamente separadas por 1/30 segundos, isso ocasionaria uma movimentação falha do objeto, onde poderia aparecer acelerando e freando, quando na verdade, estava se movendo em MRU (Movimento Retilíneo Uniforme) no computador de origem. Aumentando o número de jogadores conectados, a sobrecarga no servidor seria maior ainda.

Esse grande número de jogadores conectados enviando uma quantidade excessiva de mensagens para serem redirecionadas pelo servidor, acarreta em erros de sincronia e falhas graves no servidor, não obtendo êxito ao utilizar o jogo com vários jogadores ao mesmo tempo. A Figura 21 representa um jogo utilizando o padrão de envio das coordenadas.

O jogador A, ao se movimentar para a direita, alterou sua coordenada, enviando-a para o servidor, que redireciona para os outros jogadores. A aplicação do jogador B, ao receber a nova coordenada, atualiza a posição do objeto do jogador A.

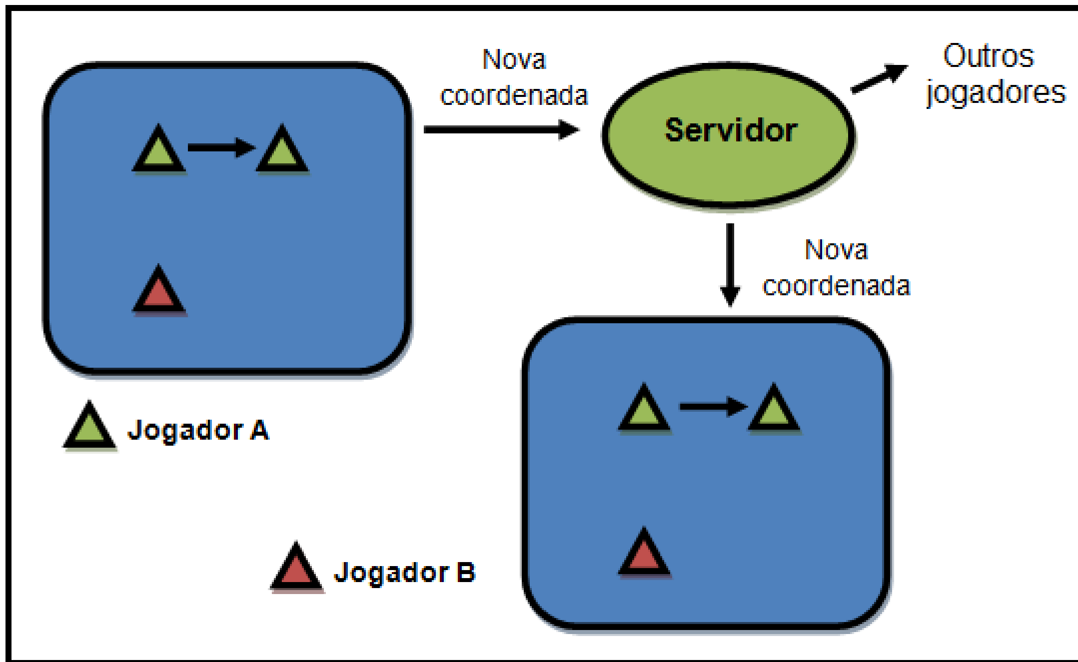


Figura 21 – Padrão de envio das coordenadas dos objetos
Fonte: Autoria Própria

Para resolver o problema dos envios das coordenadas, pode-se enviá-las dentro de um período de tempo pré-determinado, porém isso gera outro problema. As imagens remotas saltam na tela de um local para o outro, sem traçar a rota correta, isso porque faltaram as coordenadas da rota entre o ponto antigo e o ponto novo. Esses saltos são representados na Figura 22.

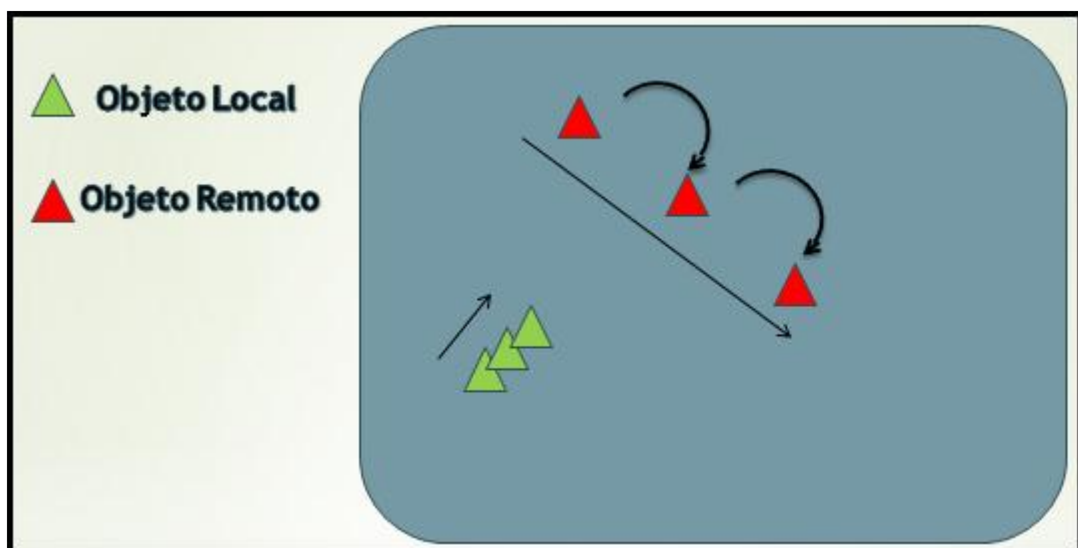


Figura 22 – Saltos dos objetos remotos ao enviar coordenadas por tempo
Fonte: Autoria Própria

Na Figura 22, o triângulo verde, representando o objeto do jogador local, movimenta-se normalmente, já o triângulo vermelho, representando outro jogador, movimenta-se por saltos. Isso se deve pela aplicação apenas alterar a posição do objeto quando recebe as coordenadas do servidor. Como não são todas as coordenadas do trajeto que são enviadas, a posição é atualizada apenas nas coordenadas que são recebidas. Esse não é um padrão eficaz, pois se perde quase toda a sensação de tempo real do jogo, além de tornar-se um jogo muito imprevisível.

3.5.2.2 Envio das ações dos jogadores

Devido aos problemas encontrados no padrão de envio das coordenadas, desenvolveu-se um novo padrão. Este visa diminuir o tráfego de rede ocasionado anteriormente, fazendo com que o desempenho do jogo melhore, e consequentemente, melhorando a sincronia entre os jogadores e a sensação de tempo real.

Esse padrão consiste em enviar as ações dos jogadores pela rede, ou seja, informar o servidor de todas as interações do jogador com o teclado durante a execução do jogo. Dessa forma, a quantidade de mensagens diminuiu drasticamente, devido às teclas que são pressionadas em uma quantidade bem menor do que os objetos mudam de coordenadas. As mensagens enviadas mostram a tecla que foi pressionada ou solta, ou se houve uma combinação de teclas.

Os clientes armazenam informações antigas, que ocorreram no decorrer do jogo, e apenas com as ações que chegam pela rede, conseguem deduzir as posições e movimentos exatos dos outros jogadores. A Figura 23 representa o jogo utilizando o padrão proposto.

O padrão mostrou-se eficiente, porém devem-se escolher as informações corretas sobre as ações a se enviar, a Figura 24 mostra como a mensagem para envio é estruturada. Pode-se dizer que esta estrutura é um protocolo desenvolvido para funcionamento do jogo, e esse método lembra uma chamada de procedimento remoto (RÉGIS, 2010), o qual a mensagem enviada fará com que os métodos remotos necessários sejam invocados para o correto funcionamento da aplicação distribuída.

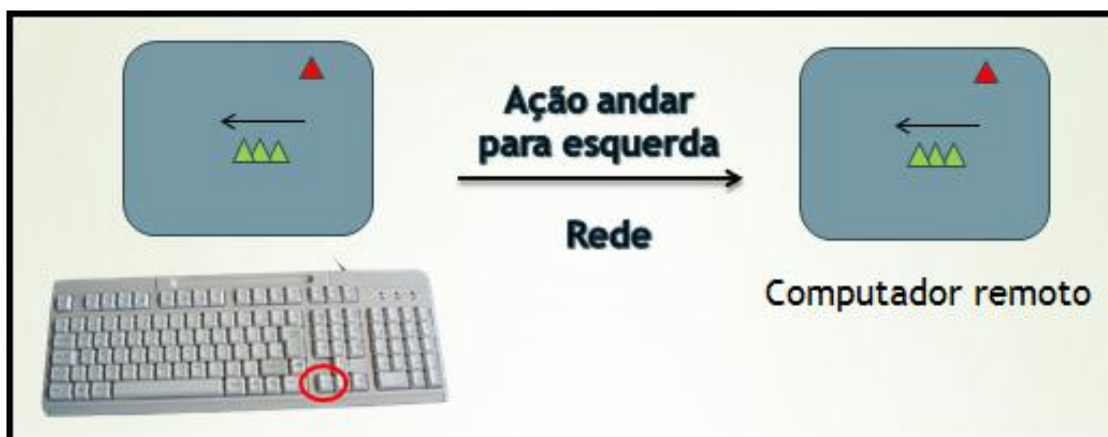


Figura 23 – Padrão de envio das ações dos jogadores

Fonte: Autoria Própria

Estrutura da Mensagem:

Message Type	Message

Valores:

Message Type =		
	KEY_DIRECTION	- UP, DOWN, LEFT, RIGHT, UP L, UP R, DOWN L, DOWN R
	KEY_UP	- (tecla que foi solta)
	KEY_SPEED	- (número inteiro contendo a velocidade em pixels/s)
	REFRESH	- (coordenadas, velocidade, direção)

Figura 24 – Estrutura e informações enviadas no padrão das ações do jogador

Fonte: Picoli (2011)

Na Figura 24, a mensagem de tipo KEY_DIRECTION transmite os valores das teclas de direção ou da combinação de teclas. Os tipos de teclas e combinações estão separados por vírgula na Figura, contendo oito direções distintas. Esse tipo determina a direção em que o cliente deve deslocar as imagens. O tipo KEY_UP, representa que uma tecla foi solta, e se for uma das setas de direção, o cliente deve parar de se mover. KEY_SPEED muda a velocidade de deslocamento das imagens, e REFRESH é uma mensagem especial que envia as coordenadas para possíveis erros de sincronia.

3.6 PROJETO E DESENVOLVIMENTO DE UM JOGO *MULTIPLAYER*

3.6.1 Descrição Geral

O jogo desenvolvido é uma aplicação distribuída, onde os jogadores iniciam o jogo em computadores distintos localizados em uma mesma rede. Ao iniciar, existe um menu onde é possível escolher um nome para o jogador e buscar os servidores disponíveis. Vários servidores podem estar presentes, porém, é necessário escolher um deles para jogar. Dessa forma, é possível que vários ambientes do jogo estejam ativos na rede, controlados por servidores distintos.

Ao escolher um servidor ativo, inicia-se o ambiente de interação com os outros jogadores conectados, onde o jogador é representado por um objeto. No exemplo, representado por uma abelha e possuindo seu nome descrito logo acima da imagem. O jogador tem as opções de movimentar-se em oito direções utilizando as teclas de direção, e diminuir ou aumentar sua velocidade de deslocamento utilizando, respectivamente, as teclas F1 e F2.

Caso haja apenas um jogador conectado, a imagem do mesmo se representará sozinha no ambiente, e conforme outros jogadores se conectam, vão surgindo em suas posições corretas na tela, inclusive com seus respectivos nomes acima da imagem. Os deslocamentos dos jogadores podem ser visualizados por todos os outros que se encontram no mesmo ambiente do servidor em questão.

3.6.2 Visão Geral

No exemplo, cada jogador é representado por uma imagem com seu nome, e controla uma aplicação diferente. As três aplicações foram executadas no mesmo computador para demonstração, porém é possível serem executadas em máquinas distintas, o que caracteriza um jogo *multiplayer*.

A Figura 25 demonstra o exemplo de um ambiente com três jogadores conectados a um mesmo servidor.

Figura 25 – Três jogadores interagindo no mesmo ambiente do jogo
Fonte: Autoria Própria

Os diagramas de casos de uso representados nas Figuras 26 e 27 demonstram as ações que serão ou podem ser executadas até o início do jogo, e durante a execução do jogo, simplificando a descrição vista.

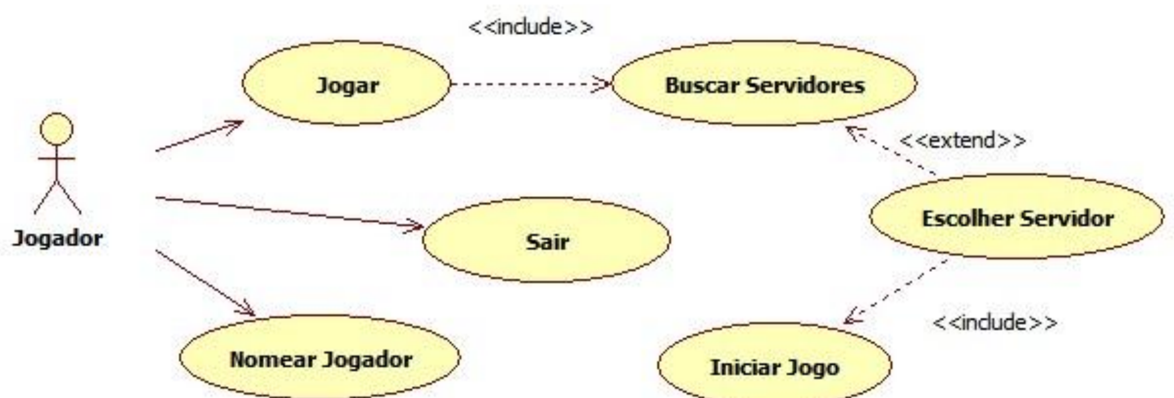


Figura 26 – Diagrama de Casos de Uso das ações até o início do jogo
Fonte: Autoria Própria

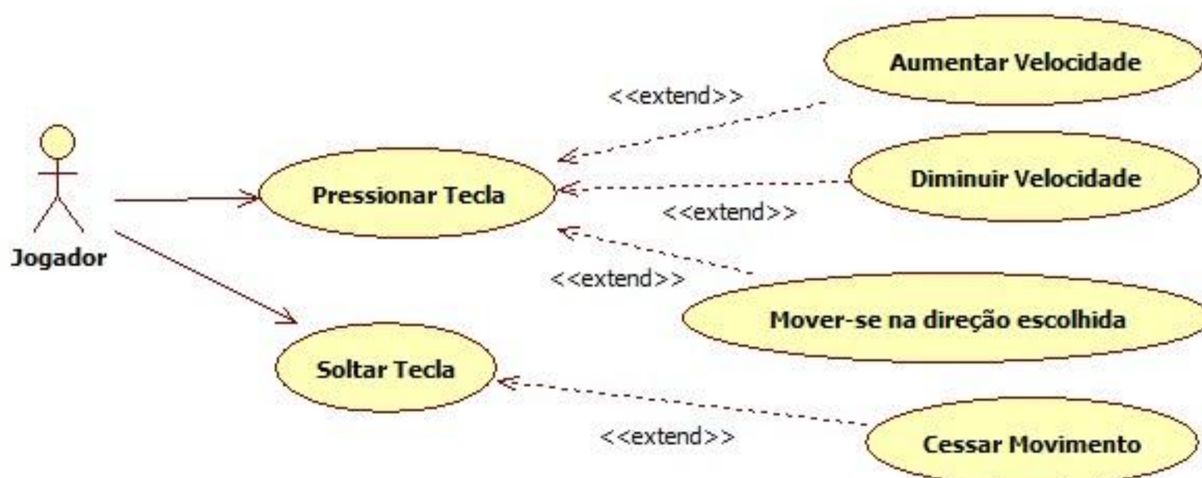


Figura 27 – Diagrama de Casos de Uso das ações durante o jogo
Fonte: Autoria Própria

3.6.3 Tecnologias Utilizadas

Várias características descritas são as mesmas tecnologias utilizadas para desenvolvimento do *chat*. A arquitetura cliente-servidor é uma delas, onde uma aplicação controla os clientes conectados e redireciona as mensagens que chegam para os outros jogadores, essa aplicação é o servidor do jogo. E o cliente é a aplicação que se conecta ao servidor e envia mensagens a serem redirecionadas.

Outra tecnologia utilizada é o uso de *threads* para controle dos clientes e para recepção de mensagens. A cada cliente conectado, o servidor cria uma nova *thread* para controle do mesmo, e no cliente, há uma *thread* que recepciona as mensagens que chegam do servidor. Existem outras *threads* que executam nos clientes, porém, veremos o uso das mesmas ao descrever o desenvolvimento do cliente mais a frente.

A teoria de redes de computadores é essencial em sistemas distribuídos, e é muito usada no desenvolvimento do jogo. Exemplos são a utilização de endereços IP e das portas da camada de transporte. Para a conexão entre os processos são utilizados *sockets* e o protocolo TCP, a partir dessa conexão são enviadas as mensagens durante a execução do jogo. O protocolo UDP é utilizado nos *sockets* para obter o servidor dinâmico, onde o cliente encontra automaticamente os servidores ativos na rede.

Além do uso da rede, é importante ressaltar as técnicas utilizadas para a programação. O jogo foi desenvolvido utilizando a linguagem de programação Python e orientação a objetos em sua totalidade, todos os problemas encontrados foram abstraídos em classes que se comunicam entre si. O *engine* utilizado para programação da parte gráfica foi o Pygame, que também foi utilizado nos exemplos de implementação de pequenos jogos apresentados no capítulo 3.1.2.

O padrão proposto de troca de mensagens utilizado foi o que se mostrou mais eficaz. O padrão de envio das ações dos jogadores foi o escolhido, aumentando o desempenho e a sincronia do jogo.

3.6.4 O Servidor

3.6.4.1 Envio de broadcast para servidor dinâmico

O jogo possui um sistema para os clientes encontrarem automaticamente os servidores ativos, para tal, foi projetado o envio de uma mensagem a partir do servidor, destinada ao endereço de broadcast das redes. Assim que o servidor é iniciado, essa mensagem é enviada a cada período de tempo em todas as interfaces de rede encontradas no computador onde o servidor está localizado. A partir desta mensagem, os clientes podem encontrar o endereço da aplicação servidora. O jogo desenvolvido envia a mensagem no endereço de broadcast a cada segundo, porém, esse tempo pode ser modificado.

O protocolo UDP, por não ser orientado a conexões, foi escolhido para envio de tal mensagem, pois como é uma mensagem destinada a todos os computadores da rede, não é possível estabelecer uma conexão. A forma como o cliente encontra os servidores ativos será descrita mais detalhadamente ao descrever o desenvolvimento da aplicação cliente.

3.6.4.2 Conexões de novos clientes

Os passos executados quando novos clientes solicitam uma conexão com servidor podem ser visualizados no diagrama de sequência da Figura 19, no capítulo em que foi descrito o desenvolvimento do *chat* (Capítulo 3.4.2).

Da mesma forma, conexões de novos clientes disparam no servidor a criação de uma nova *thread* para controle do mesmo. O objeto *thread* é armazenado

em um vetor no servidor, o qual será utilizado para envio de mensagens redirecionadas. Toda atividade e mensagens que passam pelo servidor são impressas no console para controle e auditoria do tráfego de rede e do processamento do servidor. A Figura 28 mostra as mensagens no console, contendo atividades de conexão e ações dos jogadores.

```
C:\Python32\python.exe
Servidor Iniciado na porta  5050
Broadcast Iniciado na porta  4565
('192.168.0.152', 52202)  conectou-se...
('192.168.0.152', 52202)  passou pelo servidor:  #KEYDOWN_DIRECTION#RIGHT
('192.168.0.152', 52202)  passou pelo servidor:  #REFRESH#22$0$53$54#2#2#RIGHT#I
VAN
('192.168.0.152', 52202)  passou pelo servidor:  #KEYDOWN_DIRECTION#DOWN RIGHT
('192.168.0.152', 52202)  passou pelo servidor:  #REFRESH#148$8$53$54#2#2#DOWN R
IGHT#IVAN
('192.168.0.152', 52202)  passou pelo servidor:  #REFRESH#274$134$53$54#2#2#DOWN
 RIGHT#IVAN
('192.168.0.152', 52202)  passou pelo servidor:  #KEYUP_DIRECTION#KEYUP
('192.168.0.152', 52202)  passou pelo servidor:  #KEYUP_DIRECTION#KEYUP
('192.168.0.152', 52202)  passou pelo servidor:  #REFRESH#316$176$53$54#0#2#-#IV
AN
('192.168.0.152', 52202)  passou pelo servidor:  #REFRESH#316$176$53$54#0#2#-#IV
AN
```

Figura 28 – Atividades do servidor mostradas no console
Fonte: Autoria Própria

3.6.4.3 Redirecionamento de mensagens

Na Figura 28, são representadas algumas mensagens que passaram pelo servidor, essas são as mensagens redirecionadas. No exemplo, as mensagens têm como emissor a máquina com endereço IP 192.168.0.152 na porta 52202, e foram redirecionadas para os outros clientes que se encontravam conectados, se houvesse algum. Como no *chat*, o servidor redireciona todas as mensagens que recepciona, elas chegam pela *thread* que controla o cliente emissor e são enviadas pelas *threads* que controlam os outros clientes, essas *threads* contêm os *sockets* e conexões que formam o canal de comunicação entre os processos remotos.

3.6.4.4 Tratamento de conexão perdida

Caso um cliente feche o jogo, ou simplesmente se desconecte, a conexão de *socket* é perdida e a *thread* que o controlava no servidor deve desaparecer. Então, caso alguma falha de conexão ocorra, a conexão é considerada encerrada e o servidor providencia para que o cliente seja devidamente desconectado, descartando sua *thread* e informando os outros clientes. Nesse caso, a imagem do jogador desconectado simplesmente desaparece para os outros que continuam ativos e o jogo prossegue normalmente.

3.6.4.5 Estrutura da aplicação servidora

O diagrama de classes da Figura 29 contém a modelagem do servidor do jogo *multiplayer*, onde as *threads* herdam a classe *Thread* e sobrescrevem o método *run()*, que no caso do Python é importada com o pacote *threading* (PYTHON.ORG, 2011).

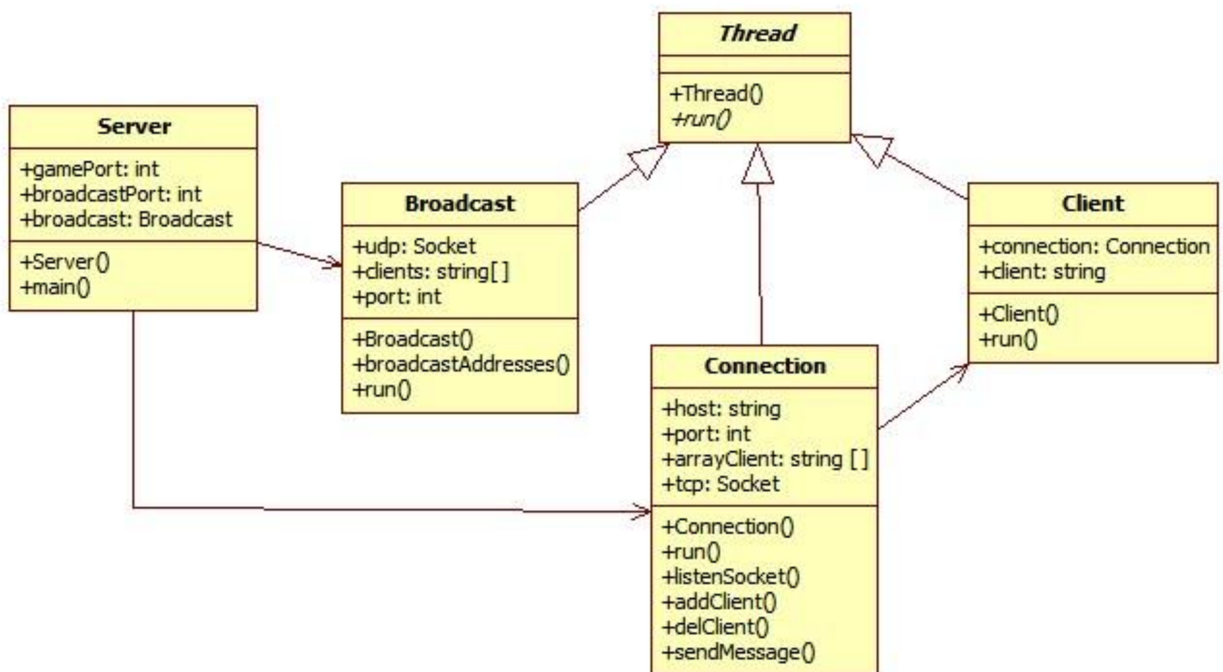


Figura 29 – Diagrama de Classes do servidor do jogo

Fonte: Autoria Própria

Ao iniciar o servidor, o método *main()* da classe *Server* contêm um laço, o qual é executado até o fim do processo, juntamente, é instanciada e iniciada a *thread Broadcast*, a qual controla o servidor dinâmico. A classe *Connection* também é instanciada com o início do servidor, e é ela que recepciona novos clientes e instancia uma nova *thread Client* para controle deste novo cliente. Os atributos e métodos das classes são utilizados no decorrer das etapas do servidor, para controle e gerenciamento dos clientes e conexões.

3.6.5 O Cliente

3.6.5.1 Busca e escolha de servidores

A partir do sistema de broadcast do servidor, é possível que o cliente recepcione a mensagem enviada e identifique o endereço e porta que se encontra tal servidor, e não apenas um, mas vários servidores ativos. Ao buscar servidores, a aplicação aguarda um tempo e armazena todos os endereços distintos que recebeu nesse período, e após, disponibiliza uma lista de servidores ao jogador, o qual escolhe um para conectar-se.

Como o protocolo utilizado para envio da mensagem no endereço de broadcast foi o UDP, e não é necessária uma conexão ativa, o *socket* da aplicação cliente recepciona qualquer mensagem que chegue pela porta criada. Dessa forma é possível encontrar vários servidores, o único detalhe é que a mensagem recebida deve ser analisada, para evitar que outros softwares na rede enviem mensagens na mesma porta e se passem por um servidor falso. A Figura 30 mostra a lista para escolha de servidores ativos.

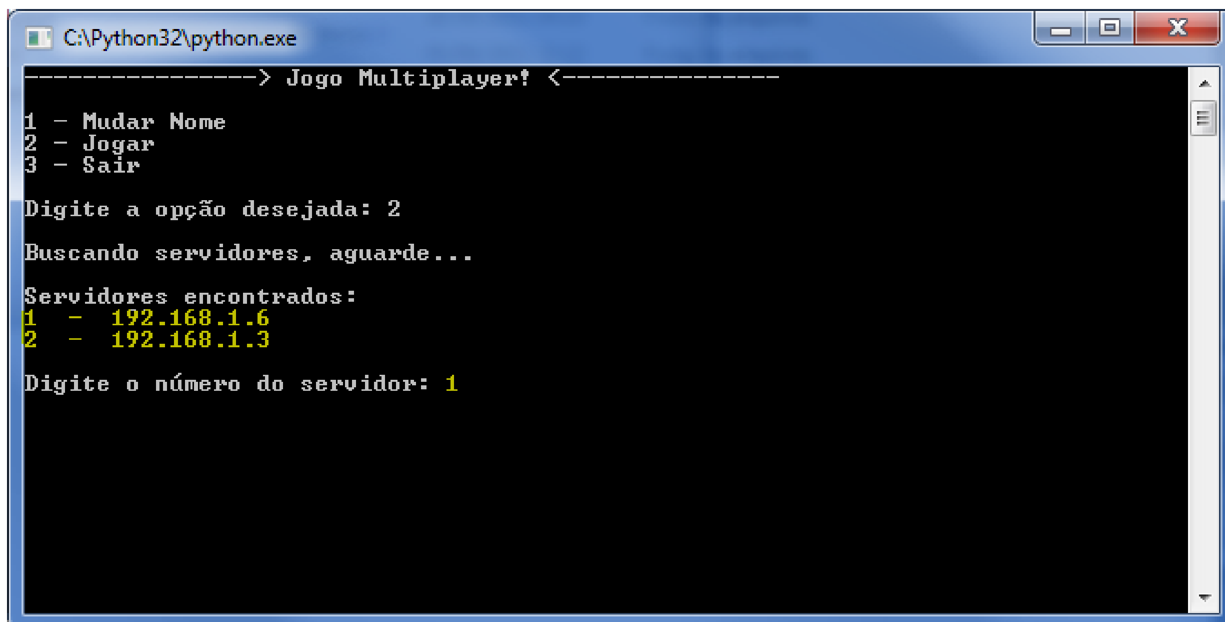


Figura 30 – Lista de servidores ativos encontrados no console da aplicação cliente
Fonte: Autoria Própria

3.6.5.2 Controle do FPS e separação da lógica

O consumo de hardware durante a execução de um jogo é muito importante para avaliar seu desempenho; se o consumo é muito alto, o jogo pode apresentar falhas e lentidão. Em jogos *multiplayer* esse problema acarreta falhas de sincronia entre os jogadores e posições erradas dos objetos locais e remotos.

Para que as imagens apareçam e movimentem-se, é necessário que sejam impressas em um curto período de tempo, tempo esse que seja imperceptível para os olhos humanos. Cada imagem é chamada de quadro ou *frame*. A quantidade de frames que são impressos por segundo é chamada de FPS do jogo.

Avaliando o desempenho do jogo desenvolvido, observou-se que quanto mais alto o FPS, maior o consumo de hardware, ocasionando perda de desempenho do jogo. Chegou-se a conclusão que o controle do FPS deveria ser feito. O jogo em sua versão final possui um FPS com uma média de 30, o necessário para simular um movimento uniforme e sem saltos, porém, é possível liberar o FPS e obter o máximo possível dependendo do poder de processamento do computador utilizado.

O controle do FPS mostrou-se eficaz no que se diz consumo de hardware, possibilitando várias instâncias do jogo executando no mesmo

computador, facilitando os testes. Na Figura 25 (Capítulo 3.6.2), no canto superior direito das aplicações, é apresentando o FPS no momento de execução.

É possível controlar o FPS limitando a execução da lógica e da impressão da imagem em um período de tempo, porém, ao limitar o FPS dessa forma, obrigatoriamente, a lógica do jogo, como a verificação do teclado e movimento dos objetos na memória também serão limitados. É interessante que essa lógica seja controlada a parte, a fim de obter maior desempenho, sendo assim, o uso de *threads* auxilia no desenvolvimento.

Ao iniciar o jogo, uma nova *thread* é criada, a qual habilita uma variável de tempo em tempo para que seja executada a lógica. O FPS é controlado por outra *thread*, disponibilizada pelo *engine* Pygame. A Figura 31 representa o diagrama de atividades referente ao controle da execução da lógica do jogo.

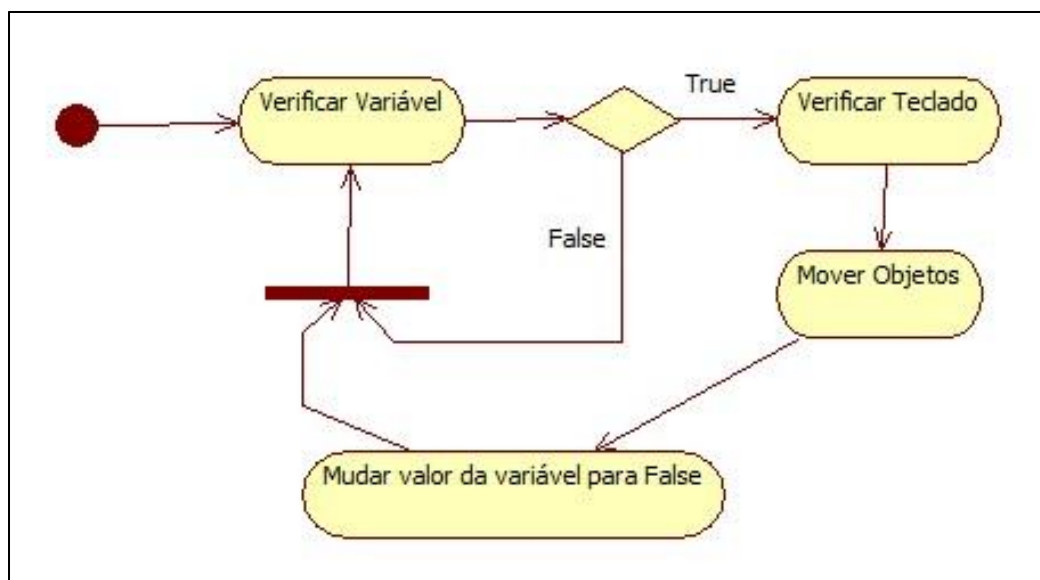


Figura 31 – Diagrama de Atividades referente ao controle da lógica do jogo
Fonte: Autoria Própria

A variável de controle é alterada para o valor *True* pela *thread* de controle. Na *thread*, a cada período de tempo, verifica-se se o valor da variável é *False*, se sim, é alterada para *True*. Na execução principal, como descrito no diagrama da Figura 31, existe o laço que verifica o valor, e executa as atividades conforme descritas caso a variável esteja habilitada.

Dessa forma é possível controlar quantas vezes por segundo os passos são executados, tendo um maior controle do consumo de hardware a partir da separação do FPS e da lógica do jogo.

3.6.5.3 Estrutura interna das mensagens enviadas

Como visto nos capítulos anteriores, o jogo utiliza o padrão de envio das ações dos jogadores para seu funcionamento, e neste capítulo veremos como as mensagens são estruturadas internamente. Antes, é importante entender como é feito o controle do teclado, pois as ações dos jogadores são teclas acionadas no teclado.

Uma classe chamada *Keyboard* contêm os métodos de controle do teclado, essa classe pode ser visualizada no diagrama da Figura 34 (Capítulo 3.6.5.5). Durante a execução da lógica, um dos passos é verificar a situação do teclado, essa verificação é feita invocando o método *keyControl()* do objeto controlador do teclado. O Pygame possui um buffer que armazena as teclas que foram pressionadas e soltas, essa lista é analisada e verifica-se a situação, determinando um estado para o teclado, o método *executeState()* é invocado para que seja enviada a mensagem correta ao servidor, mensagem essa que contêm a ação do jogador. Para envio, é invocado o método *sendMsg()* da classe *Connection*.

Mensagens enviadas pela conexão de *socket* devem ser serializadas, ou seja, em formato binário. Para isso, utiliza-se no momento do envio, uma função do Python para serializar textos, e da mesma forma, ao receber uma mensagem, é utilizada uma função para obter o inverso e transformar a mensagem serializada em um texto. As funções descritas podem ser visualizadas na Figura 32, onde o método *encode()* na linha 1 serializa a mensagem para ser enviada, e o método *decode()* na linha 3 faz o papel contrário ao receber uma mensagem.

```
1  connection.send(str.encode(msg))
2
3  msg = bytes.decode(self._tcp.recv(1024))
```

Figura 32 – Serialização de mensagens em Python
Fonte: Autoria Própria

A mensagem contendo a ação do jogador deve possuir uma única string, a fim de poder ser serializada para envio. Sendo assim, criou-se um padrão para separar os dados dentro da mensagem. Podemos visualizar na Figura 24 (Capítulo 3.5.5.2) os tipos de ações que podemos obter, informação essa que deve seguir com a mensagem. Também deve seguir a informação propriamente dita, no caso, a tecla, ou combinação de teclas pressionadas ou soltas.

As informações são separadas por um caracter especial. Foi utilizado o caracter “#”, então, assim que a mensagem chega ao destino, as informações são separadas utilizando como chave este caracter. Cabe ressaltar que os dois processos que se comunicam devem conhecem a ordem de envio das informações. No exemplo (1) é mostrado um exemplo de mensagem de ação ao pressionar a tecla para direita:

[1] **#KEY_DIRECTION#RIGHT**

3.6.5.4 Mensagem REFRESH para controle de sincronia

Mesmo utilizando o padrão proposto de troca de mensagens que envia as ações dos jogadores, é possível que haja erros de sincronia entre os jogadores, causados por diversos motivos, dentre eles, problemas da rede física, lentidão por sobrecarga da rede e baixo desempenho de hardware dos outros clientes (PICOLI, 2011). Para resolver essas situações, periodicamente os clientes enviam uma mensagem especial, para atualização, e esta contém as coordenadas corretas do jogador. Isso possibilita aos outros clientes, uma conferência e possível correção da posição.

Essa mensagem especial é do tipo REFRESH, e os dados da mensagem são as coordenadas exatas, a velocidade e direção do movimento e o nome do jogador. Como nas mensagens descritas no capítulo anterior, as informações são separadas pelo caracter “#”, porém, as coordenadas são uma informação dividida em mais quatro dados. Dentro da coordenada, os valores são divididos pelo caracter especial “$”, e devidamente separados ao serem entregues nos outros clientes. No exemplo (1) é mostrado um exemplo de uma mensagem do tipo REFRESH:

[1] **#REFRESH#20$10$345$140#10#RIGHT#JOÃO**

Respectivamente, na mensagem do exemplo (1) temos o tipo, as coordenadas do objeto remoto, a velocidade em pixels por segundo, a direção do movimento e o nome do jogador.

Testes foram feitos durante a execução do jogo e verificou-se que erros de sincronia ocorrem, em uma quantidade bem menor do que no padrão de envio das coordenadas. Essa mensagem do tipo REFRESH demonstrou-se útil para pequenas correções de posições, o que em longo prazo faria com que o objeto se afastasse muito da posição correta. Então, corrigindo esses pequenos erros de tempo em tempo, o objeto sempre estará localizado em uma coordenada muito próxima ou correta em relação à coordenada real do objeto remoto.

3.6.5.5 Estrutura da aplicação cliente

Os diagramas de classes das Figuras 33 e 34 contêm a modelagem do cliente do jogo *multiplayer*, onde as *threads* herdam a classe *Thread* e sobrescrevem o método *run()*, que no caso do Python é importada com o pacote *threading* (PYTHON.ORG, 2011). Na Figura 33, são representadas as classes utilizadas durante o menu do jogo, na escolha do servidor e início do jogo. Na Figura 34 são representadas as classes utilizadas durante a execução do ambiente do jogo.

Os atributos e métodos que as classes do diagrama da Figura 33 possuem, são utilizados durante a execução das opções de menu e execução do jogo. Cabe ressaltar que os atributos e métodos existentes na classe *Game* serão apresentados neste diagrama apenas, e nos diagramas a seguir, essa classe possuirá apenas seu nome, porém, os atributos e métodos serão os mesmos.

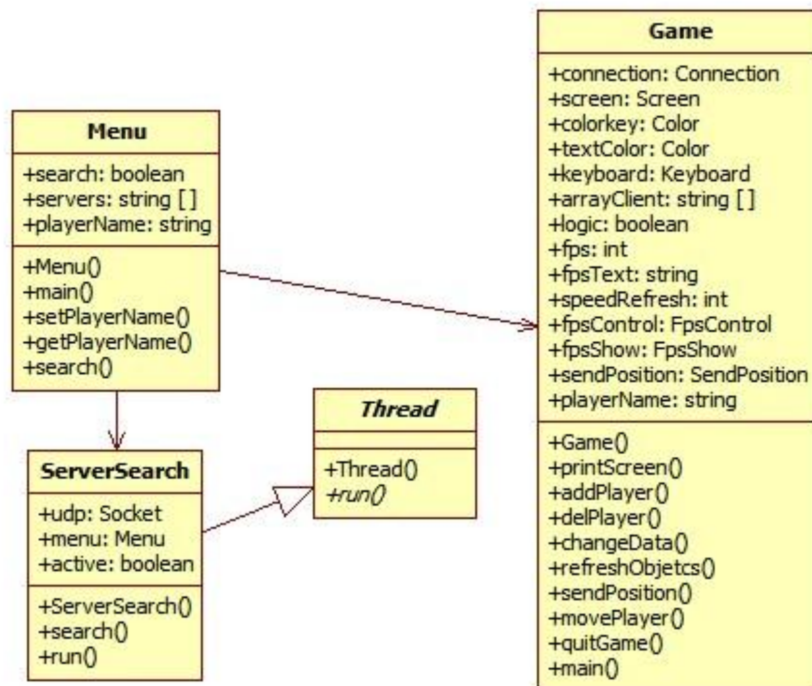


Figura 33 – Diagrama de Classes do Cliente na execução das opções de menu

Fonte: Autoria Própria

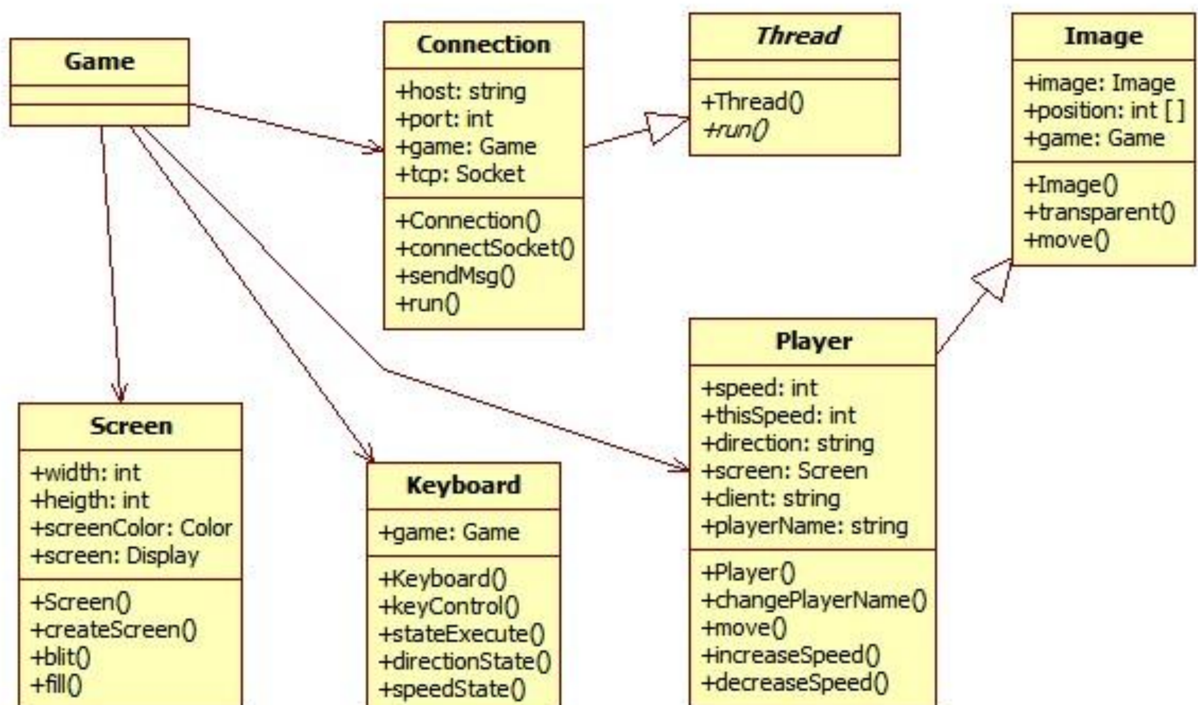


Figura 34 – Diagrama de Classes do cliente na execução do ambiente do jogo

Fonte: Autoria Própria

A Figura 35 contém o diagrama de classes que representa a estrutura das classes de controle do jogo, como o envio da mensagem especial REFRESH representado pela classe *SendPositionControl*, a separação da lógica e do FPS do jogo representada pela classe *FPSControl*, além de uma *thread* utilizada para mostrar o valor do FPS na tela a cada um segundo, representada pela classe *FPSShow*.

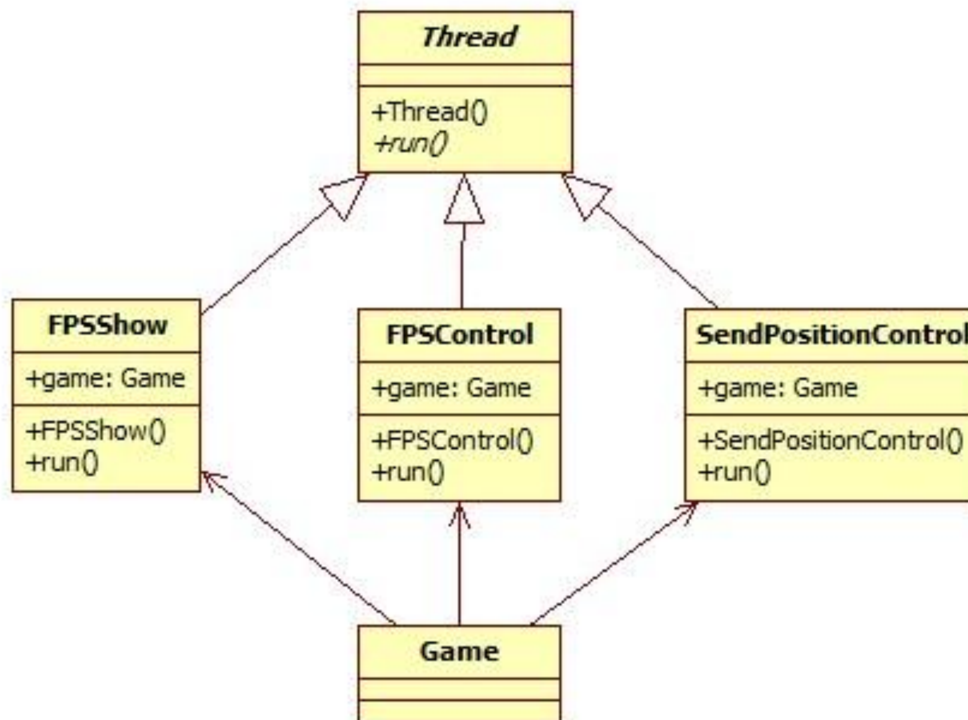


Figura 35 – Diagrama de Classes da estrutura de controle do jogo
Fonte: Autoria Própria

3.7 ANÁLISE DOS RESULTADOS

3.7.1 Implementações Anteriores

O jogo foi desenvolvido com os dois padrões de mensagens propostos, o envio das coordenadas e o envio das ações dos jogadores. Ambos foram implementados, porém, o escolhido foi o envio das ações, por mostrar-se mais eficaz em relação ao envio das coordenadas. A única diferença entre as duas aplicações é o padrão de mensagens, sendo que em todas as outras partes utilizou-se as

mesmas tecnologias, tecnologias essas que foram demonstradas no decorrer do trabalho. Nos próximos capítulos serão mostradas a análise dos resultados obtidos e os pontos críticos ao desenvolver o jogo com os dois padrões.

3.7.2 Melhorias Alcançadas

Testes foram feitos em três computadores de uma rede de 100 Mb, utilizando cabo de par trançado, interligados por um *switch*. O servidor foi executado na máquina com maior poder de processamento e os clientes foram executados em todas as máquinas, inclusive a que continha o servidor, para testes com mais de três clientes foram abertas diversas instâncias do jogo nas máquinas da rede.

Ao utilizar o padrão de envio das coordenadas dos objetos obteve-se os seguintes resultados na tela dos outros clientes conectados:

- As imagens dos outros jogadores não se movimentavam de forma regular, “pulando” na tela, pelo fato da atualização ocorrer de acordo com as coordenadas exatas que chegavam.
- A cada novo cliente conectado, o jogo ficava mais lento, aumentando o tempo para atualização dos objetos, ou seja, os “pulos” ficavam maiores, fazendo com que o jogo perdesse cada vez mais a ação.
- Às vezes o jogo parava de funcionar pelo motivo do travamento do servidor, devido a enorme quantidade de mensagens que chegavam e não conseguia processar.

Ao utilizar o padrão de envio das ações os problemas apresentados anteriormente foram sanados, e o número de clientes conectados – até o limite usado no teste anterior – não influenciava mais no desempenho do jogo. Todas as imagens eram movimentadas sem saltos e com sincronia em todos os clientes (PICOLI, 2011).

A quantidade de mensagens que passavam pelo servidor diminuiu drasticamente ao utilizar o novo padrão, como podemos observar na Tabela 1.

Tabela 1 – Quantidade de mensagens recebidas pelo servidor em um período de tempo

Período de Tempo	Clientes Conectados	N° de mensagens (coordenadas)	Nº de mensagens (ações)
5 segundos	03	Aprox. 600	Aprox. 45
5 segundos	09	Aprox. 1800	Aprox. 135

Fonte: Picoli (2011)

Os dados da Tabela 1 foram extraídos durante a ação intensa dos clientes, movimentando-se constantemente, mudando de direção em média duas vezes por segundo e alterando a velocidade algumas vezes.

A Tabela 2 mostra a quantidade de impressões por segundo (FPS) dos objetos de outros jogadores, visto em uma máquina cliente, essas impressões por segundo determinam o tamanho dos “pulos” dos objetos ao utilizar o padrão do envio das coordenadas, o outro padrão proposto possui um número suficiente de impressões para simular tempo real, devido que os próprios clientes realizam o movimento dos objetos, e não esperam por mensagens de coordenadas para movimentá-los. A imagem em tempo real significa que a quantidade de impressões é a mesma do FPS do jogo, se o FPS for 30, o objeto remoto terá 30 impressões.

Tabela 2 – Quantidade de impressões dos objetos dos jogadores (FPS) vistos nos outros clientes

Período de Tempo	Clientes Conectados	Quant. de impressões (coordenadas)	Quant. De impressões (ações)
1 segundo	02	Aprox. 08	Tempo Real*
1 segundo	03	Aprox. 05	Tempo Real*
1 segundo	04	Aprox. 02	Tempo Real*

* A imagem em tempo real significa que a quantidade de impressões é a mesma do FPS do jogo, se o FPS for 30, o objeto remoto terá 30 impressões.

Fonte: Picoli (2011)

Observando a Tabela 2, conclui-se que utilizando o padrão de envio das coordenadas, a cada cliente conectado o desempenho do jogo cai sensivelmente, sendo inviável o uso do jogo com muitos clientes ativos ao mesmo tempo. Já o

padrão de envio das ações do jogador, resolve o problema de desempenho, por não haver tantas mensagens a serem processadas.

3.7.3 Pontos Críticos

Um dos principais pontos onde se deve analisar cuidadosamente é o tempo de envio da mensagem especial REFRESH, pois como são enviadas as coordenadas do objeto, acaba-se por utilizando parte da implementação anterior, porém em uma escala bem menor. O tempo de envio deve ser analisado de acordo com o desempenho da rede física, pois se muitos pacotes se perderem, a falta de sincronia será maior, necessitando de mais mensagens REFRESH, já se a rede for de excelente desempenho, a mensagem será muito pouco necessária.

Vários outros valores devem ser determinados para execução do jogo, onde todos, se mal regulados, diminuem o desempenho e a jogabilidade. Alguns cuidados que se deve ter são: quanto aos números das portas de recepção de mensagens; a quantidade de frames por segundo da aplicação cliente (FPS); o número de vezes que a lógica é executada por segundo; textos de mensagens devem ser formatados corretamente para que os processos compreendam os dados.

4 CONCLUSÃO

Este trabalho de conclusão de curso é a continuação de um projeto de Iniciação Científica. O projeto foi publicado nos anais do XVI SICITE (XVI Seminário de Iniciação Científica e Tecnológica da UTFPR), realizado nos dias 28 a 30 de setembro de 2011, resultando em um artigo de titulação Padrões de Troca de Mensagens na Arquitetura Cliente-Servidor em Jogos *Multiplayer*.

A seguir, as considerações finais, encerramento do presente trabalho e serão mostrados possíveis trabalhos futuros que possam desencadear a partir da arquitetura proposta.

4.1 ANÁLISE GERAL DO DESENVOLVIMENTO

Os dois padrões de envio de mensagens fazem com que seja possível a interação entre os jogadores pela rede, porém, o padrão proposto do envio das

ações dos jogadores demonstrou-se extremamente mais eficiente em relação ao desempenho e sincronia. Possibilitou a observação dos movimentos e posições exatas dos jogadores em todas as estações que possuíam clientes ativos. Em casos da falta de sincronia, a mensagem de atualização REFRESH demonstrou-se eficaz, sincronizando o ambiente distribuído rapidamente. Além disso, esse padrão possibilitou a conexão de um maior número de clientes consecutivos sem perder desempenho.

Uma diferença negativa, porém irrelevante para computadores com bom desempenho de hardware, foi que ao forçar o cliente a deduzir os movimentos dos outros jogadores utilizando apenas suas ações, percebeu-se um pequeno aumento de uso de hardware em relação ao padrão das coordenadas. Com o controle e separação do FPS da lógica do jogo, o consumo de hardware foi controlado, e esse pequeno aumento do processamento em relação ao outro padrão se mostrou mais imperceptível.

Todo o estudo resultou no desenvolvimento da aplicação distribuída proposta, porém, um jogo completo deve possuir outros recursos além de uma arquitetura *multiplayer*. Essas características não foram abordadas neste trabalho, deixando futuros projetos que venham a aprimorar o jogo desenvolvido.

4.2 TRABALHOS FUTUROS

A arquitetura demonstrada neste trabalho deixa uma plataforma *multiplayer* pronta para o desenvolvimento de jogos mais aprimorados. Um jogo interativo deve possuir sons, cenário, melhores gráficos, colisões entres os objetos e se possível, o uso de banco de dados para armazenar e salvar os dados de uma partida, possibilitando que os jogadores retomem o jogo na mesma situação em que foi salvo.

A inteligência artificial é uma boa prática para dinamizar os videogames. Uma importante característica para desenvolvimento de personagens controlados pelo computador é o estudo da programação de personagens autônomos (POZZO, 2011), onde é implementada uma lógica para que o personagem escolha a melhor rota a se seguir.

Quanto ao cenário, uma boa alternativa é utilizar um ambiente isométrico (GRACHINSKI, 2009), onde aplicando cálculos sobre as coordenadas dos objetos,

obtêm-se um efeito que simula profundidade. Testes com *engine*s 3D também podem ser aplicados utilizando a arquitetura *multiplayer* proposta.

As condições das redes nem sempre estão da forma mais adequada para uso de sistemas distribuídos, esse é um dos motivos para uso da mensagem REFRESH, porém, o tempo de envio é fixo, deixando futuros trabalhos de análise da rede utilizada para calcular o tempo de envio dinamicamente, resultando em um maior desempenho do jogo.

Outro estudo, o qual resultaria em dados relevantes para desempenho do servidor, seria efetuar testes com a inversão da lógica de broadcast para localização do endereço do servidor. Neste caso, quem enviaria o *broadcast*, seria o cliente, e o servidor apenas responderia a requisição.

4.3 CONSIDERAÇÕES FINAIS

Ao finalizar o desenvolvimento, concluiu-se que os objetivos fixados foram alcançados com êxito, demonstrando todas as tecnologias, arquiteturas e padrões utilizados, e resultando na aplicação distribuída proposta. No decorrer do trabalho, a linguagem Python foi abordada constantemente, e pedaços de códigos foram disponibilizados, a fim de demonstrar sua sintaxe fácil e incentivar novos estudantes e pesquisadores a utilizá-la em seus estudos.

REFERÊNCIAS

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML, Guia do Usuário.** Rio de Janeiro: Campus, 2000.

BORGES, Luiz Eduardo. **Python para Desenvolvedores.** 2ª ed. Rio de Janeiro: Edição do Autor, 2010.

BOURG, David M. **Physics for Game Developers.** 1. ed. Gravestein, Highway North, Sebastopol: O'Reilly & Associates, 2002.

CAPELLER, Paulo E. B.; ANDRADE, Vinícius C. **Uso do Processo Dirigido a Responsabilidades no Desenvolvimento da Arquitetura e Modelagem do Framework de Preço de Venda**. 2010, 162f. Trabalho de Conclusão de Curso – Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2010.

COMER, Douglas E. **Interligação em Rede com TCP/IP:** Princípios, protocolos e arquiteturas. vol. I. Rio de Janeiro: Campus, 1998.

EICHELT, Samir J; PADOIN, Edson Luiz. **Programação multi-*thread* em arquitetura MultiCore.** Disponível em <http://www.ctec.unicruz.edu.br/revista/artigos/50.pdf>. Acesso em 10-10-2011.

FLYNT, John P; SALEM, Omar. **Software *Engine*ering for Game Developers.** Boston: Thonson Course Technology, 2005.

GÓIS, L. A. **Servidor Mono e Multi Clientes:** Sistemas Distribuídos. 29-03-2011. Nota de Aula. Apresentação de slides.

GRACHINSKI, Leonardo José. **Criação de plataforma para desenvolvimento de um jogo isométrico.** Ponta Grossa, 2009. 104 f.: Trabalho de Conclusão de Curso (Graduação) – UTFPR. Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas. Ponta Grossa, 2009.

KOSCIANSKI, André. **Game Programming with Irrlicht.** s.l. : Create Space, 2011.

LARMAN, Craig. **Utilizando UML e Padrões:** Uma introdução à análise e ao projeto orientados a objetos. Porto Alegre: Bookman, 2000.

MENEZES, Nilo Ney Coutinho. **Introdução a Programação com Python.** São Paulo: Novatec Editora, 2010.

PICOLI, Ivan Luiz; KOSCIANSKI, André. Padrões de Troca de Mensagens no Modelo Cliente-Servidor em Jogos *Multiplayer*. In: XVI SICITE – **Seminário de Iniciação Científica e Tecnológica da UTFPR**. Ponta Grossa, 2011.

POZZO, Luiz Gustavo. **Desenvolvimento de Jogo com Personagens Autônomos.** 2011. 52 f. Trabalho de Conclusão de Curso – Tecnologia em Análise e Desenvolvimento de Sistemas, Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2011.

PYGAME.ORG. **Pygame Documentation**. Disponível em <www.pygame.org/docs/>. Acesso em 14-08-2011.

PYTHON.ORG. **Python v3.2.2 Documentation**. Disponível em <http://docs.python.org/py3k/>. Acesso em 14-08-2011.

RÉGIS, L. O. C., RIBEIRO, J. M., POZZO, L. G., SILVA, S. C. R., KOSCIANSKI, A. Implementação de uma arquitetura esb em um jogo educacional. In: I CICPG – **Congresso de Iniciação Científica e Pós-Graduação - Sul Brasil**. Florianópolis, 2010.

ROOLINGS, Andrew; MORRIS, Dave. **Game Architecture and Design:** A New Edition. Indianapolis: New Riders Publishing, 2004.

RUMBAUGH, James; BLAHA, Michael; PREMERLANI, William; EDDY, Frederick; LORENSEN, William. **Modelagem e Projetos Baseados em Objetos**. Rio de Janeiro: Editora Campus, 1994.

SHAY, William A. **Sistemas Operacionais**. São Paulo: MAKRON Books, 1996.

SILBERS*CHATZ*, A. GALVIN, P. B., GAGNE, G.. **Fundamentos de sistemas operacionais**. Rio de Janeiro: LTC, 2004.

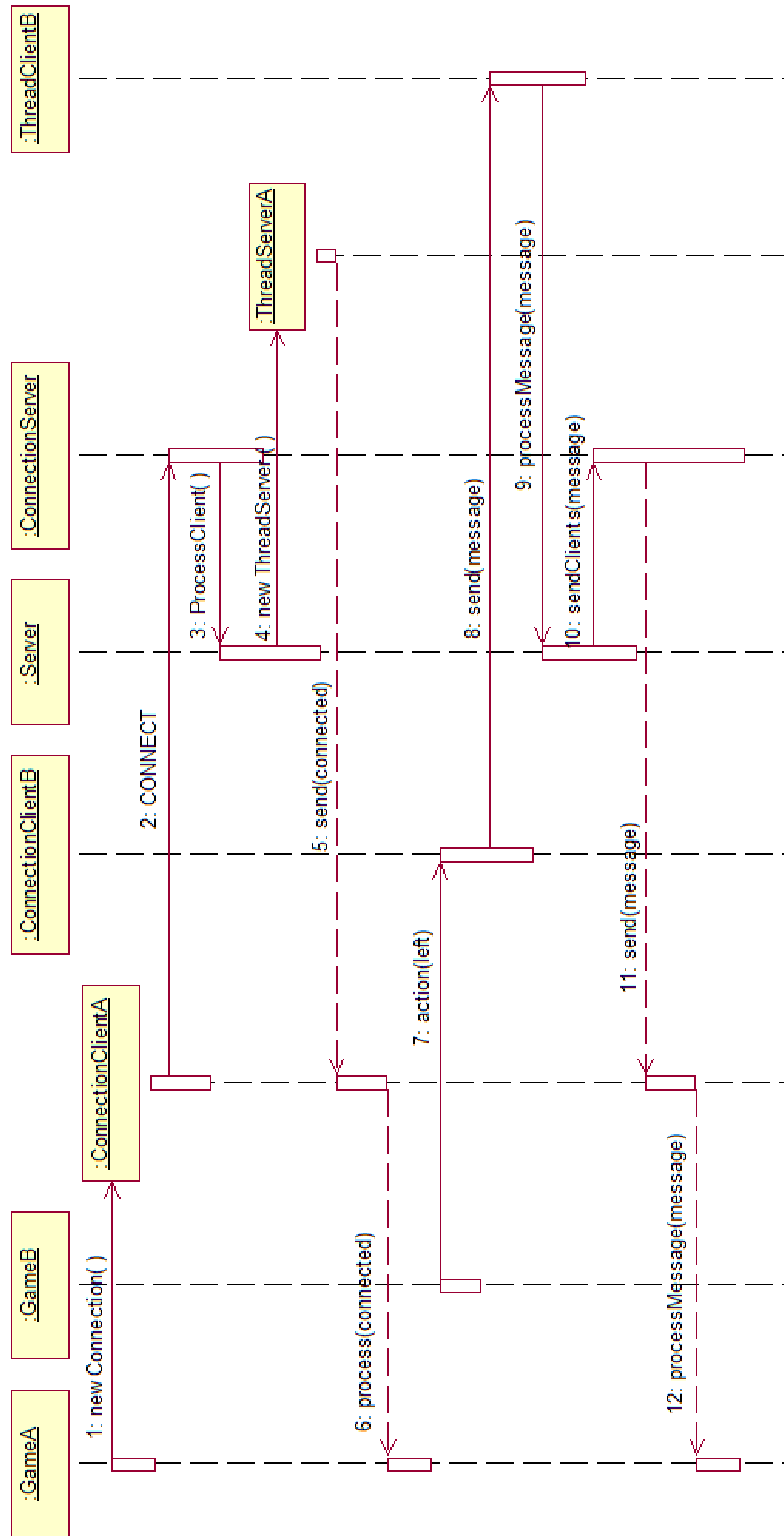
STALLINGS, William. **Data and Computer Communications.** 7. ed. Upper Saddle River, New Jersey: Pearson Education, 2004.

TANENBAUM, Andrew S. **Redes de Computadores.** 4. ed. Rio de Janeiro: Elsevier, 2003.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos.** Rio de Janeiro: LTC – Livros Técnicos e Científicos,1995.

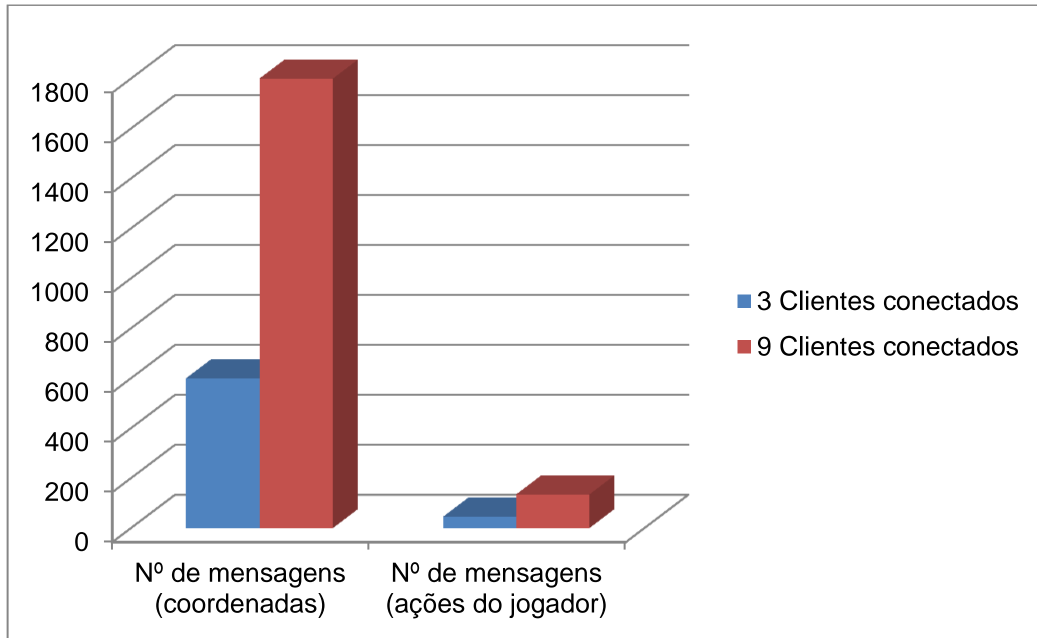
TREMBLAY, Christopher. **Mathematics for Game Developers.** Boston: Thonson Course Technology, 2004.

APÊNDICE A – FLUXO DE DADOS DURANTE A CONEXÃO DE UM CLIENTE E O ENVIO DE MENSAGENS COM DOIS CLIENTES CONECTADOS

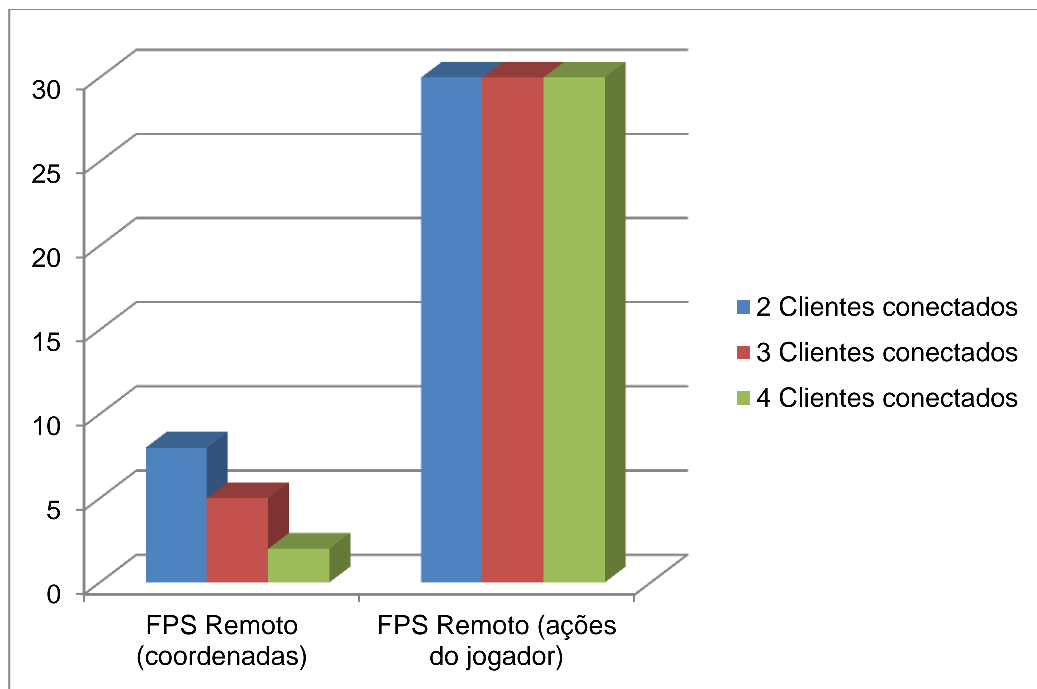

:GameA
:GameB
:ConnectionClientA
:ConnectionClientB
:Server
:ConnectionServer
:ThreadServerA
:ThreadClientB
1: new Connection( )
2: CONNECT
3: ProcessClient( )
4: new ThreadServer( )
5: send(connected)
6: process(connected)
7: action(left)
8: send(message)
9: processMessage(message)
10: sendClients(message)
11: send(message)
12: processMessage(message)

APÊNDICE B – GRÁFICOS DE DESEMPENHO DOS PADRÕES DE ENVIO DE MENSAGENS

Quantidade de mensagens processadas pelo servidor nos padrões de envio das coordenadas dos objetos e envio das ações dos jogadores em um período de 05 (cinco) segundos



FPS dos objetos remotos obtidos na aplicação cliente tendo o FPS local com o valor 30 (trinta)



Obs.: Os dados dos gráficos estão disponíveis nas tabelas do capítulo 3.7.2.