UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

Gabriel Machado da Roza

MODERNIZAÇÃO DA FERRAMENTA PARA CRIAÇÃO DE JOGOS EM ARQUITETURA CLIENTE-SERVIDOR NETGAMES.

FLORIANÓPOLIS
2023

GABRIEL MACHADO DA ROZA

MODERNIZAÇÃO DA FERRAMENTA PARA CRIAÇÃO DE JOGOS EM ARQUITETURA CLIENTE-SERVIDOR NETGAMES

Trabalho de Conclusão de Curso submetido à Universidade Federal de Santa Catarina, como requisito necessário para obtenção do grau de Bacharel em Sistemas de Informação

Florianópolis, 2023

UNIVERSIDADE FEDERAL DE SANTA CATARINA

GABRIEL MACHADO DA ROZA

Esta Monografia foi julgada adequada para a obtenção do título de Bacharel em Sistemas de Informação, sendo aprovada em sua forma final pela banca examinadora:

Orientador: Prof. Dr. Ricardo Pereira e Silva
Universidade Federal de Santa Catarina - UFSC

Prof. Mestre José Eduardo De Lucca
Avaliador
Universidade Federal de Santa Catarina - UFSC

Prof. Dr. Alex Sandro Roschildt Pinto
Avaliador
Universidade Federal de Santa Catarina - UFSC

Florianópolis, 2023

Resumo

O desenvolvimento de aplicações cliente servidor pode ser desafiador para iniciantes na programação. Assim, surgem ferramentas para auxiliar nesta tarefa. Este trabalho propõe uma versão modernizada para uma destas ferramentas, com foco no desenvolvimento de jogos multijogador sem tempo real. Utilizando-se do protocolo WebSocket, a ferramenta criada é futuramente extensível a novas plataformas. Foi possível aferir o uso da ferramenta criada através de sua utilização nas disciplinas de Engenharia de Software I e Análise e Projeto de Sistemas, respectivamente dos cursos de Ciências da Computação e Sistemas de Informação da Universidade Federal de Santa Catarina.

Palavras-chave: Python, WebSockets.

Abstract

Developing client-server applications can be a challenging assignment for novice developers. As such, tools are created to facilitate this kind of task. This thesis proposes a new version of one such a tool, which focuses on non-real time multiplayer games. The proposed tool seeks, through the use of WebSocket, to be extentedable to new platforms in the future, should the need arise. The tool's usage was assesed through it's application in the subjects of Software Engineering I and Systems Analysis and Design, respectively of the Computer Science and Information Systems courses at the Federal University of Santa Catarina.

Keywords:Python, WebSockets.

Lista de ilustrações

Lista de códigos

Lista de abreviaturas e siglas

$HTTP$	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
RFC	Request for Comments
$UTF-8$	Unicode Transformation Format 8 bit
API	Application programming interface
NRT	Non-real time
$J2ME$	Java Platform, Micro Edition
$JSON$	JavaScript Object Notation
$HTML5$	HyperText Markup Language versão 5

Sumário

1 Introdução e Objetivos

1.1 Introdução

Desenvolvida em 2007, a ferramenta Netgames auxilia no desenvolvimento de jogos multi-jogador sem suporte a tempo real, abstraindo conceitos de programas cliente-servidor através de uma API de alto nível, de forma que iniciantes na programação consigam criar seus jogos multi-jogador sem grandes dificuldades. Composta por dois artefatos, um servidor e outro cliente (integrado ao código dos jogos), a ferramenta é genérica pois não é atrelada a lógica de nenhum jogo em particular. Essas são características que contribuíram para seu contínuo uso acadêmico nos anos que se passaram desde sua criação. Há, entretanto, uma característica que nos tempos recentes dificulta sua aplicação: seus artefatos são fortemente atrelados a plataforma Java devido à escassez de documentação de seu mecanismo de comunicação, o framework DualRPC, o que inviabiliza o aproveitamento do componente servidor e a criação de um novo artefato cliente que faça interface com outras plataformas populares, como por exemplo Javascript ou Python. Dessa forma, este trabalho propõe a criação de uma nova ferramenta, capaz de reproduzir as virtudes do Netgames original em plataforma Python, enquanto que se atentando para que a tecnologia de comunicação utilizada, os WebSockets, não prejudique futuros desenvolvimentos.

1.2 Justificativa

Como mencionado anteriormente, o atual Netgames não comporta desenvolvimento além do Java. Dessa forma, este trabalho propõe uma ferramenta capaz de ser utilizada em Python, para que os benefícios de Netgames não se percam frente a uma mudança de linguagem.

1.3 Método de pesquisa

1.3.1 Natureza: Aplicada

Visando criar conhecimento a fim de minimizar um problema existente. Para este trabalho, o desenvolvimento de uma solução que auxilie o desenvolvimento de jogos multi-jogador sem suporte a tempo real, compatível com a linguagem Python.

1.3.2 Abordagem: Qualitativo

As soluções já utilizadas em ferramentas similares a esse já são uma fonte de dados para o problema que está sendo abordado, não necessitando de cálculos específicos para capturar as informações necessárias.

1.3.3 Objetivo: exploratório

Almejando a compreensão do problema através de trabalhos relacionados elucidados nesse trabalho usando estudo de soluções passadas para problemas similares ao que está sendo abordado.

1.3.4 Procedimentos técnicos: Bibliográficos, Estudo de Caso

O desenvolvimento de uma ferramenta que auxilie na elaboração de jogos multi-jogador sem suporte a tempo real já foi uma ideia previamente explorada em estudos passados. Busca-se replicar a mesma ideia para a linguagem Python.

1.4 Objetivo geral

Propor uma ferramenta baseada em Python para auxílio ao desenvolvimento de jogos multi-jogador sem tempo real.

1.5 Objetivos Específicos

1. Desenvolver um artefato servidor;
2. Desenvolver um artefato cliente que exponha uma API de alto nível, abstraindo detalhes da comunicação com o artefato servidor;
3. Integrar o artefato a uma aplicação existente, de forma a exemplificar o uso dos artefatos;
4. Documentar e disponibilizar os artefatos desenvolvidos.

1.6 Estrutura do documento

O capítulo 1 introduz o trabalho, apresentando sua justificativo e objetivos. No capítulo 1 são aprofundados conceitos relevantes ao trabalho. O capítulo 3 aborda trabalhos relacionados, seguido pelo capítulo 4 onde é apresentada a ferramenta desenvolvida. Finalmente, no capítulo 5 são apresentados os resultados obtidos e no 6 a conclusão.

2 Fundamentação Teórica

2.1 Python

Python é uma linguagem de programação de alto nível, largamente utilizada e capaz de suportar múltiplos paradigmas, incluindo orientação a objetos, programação imperativa e funcional (SRINATH, 2017), criada em 1991 por Guido van Rossum. A execução de um programa Python se dá através de interpretadores (dos quais existem diversas implementações), seu sistema de tipos é dinâmico e o gerenciamento de memória é automático. Tais características, aliadas a uma filosofia de design da linguagem que preza pela legibilidade do código (TULCHAK; MARCHUK, 2016), tornaram Python uma ferramenta adequada para introdução a programação, ultrapassando a linguagem Java (SRINATH, 2017).

```
1  def hello():
2      print('Hello World!')
3  hello()
```

Código 1: Exemplo de código Python

Uma das características mais prestigiadas do Python é a suíte de ferramentas incluída por padrão: bibliotecas de matemática, manipulação de arquivos e serviços criptográficos são alguns exemplos. Conforme novas versões da linguagem são desenvolvidas, certas sofisticações, que vão além de meras funções são adicionadas. Aqui serão abordados alguns destes recursos.

2.1.1 Type hints

O suporte a sinalizações de tipos aproxima os códigos Python, ainda que visualmente, de linguagens onde há um controle de tipos mais rígidos. Tomando por exemplo o Código 2, é definida uma função que recebe dois argumentos e retorna o resultado da aplicação do operador +. Por mais que o nome da função sugira que a intenção do código é concatenar caracteres, nada impede que seja realizada uma invocação onde os argumentos sejam dois números inteiros, e nesse caso o resultado da operação também será um número inteiro.

```
1  def concatenar(a, b):
2      return a + b
```

Código 2: Exemplo de função Python sem sinalizações de tipo

Com o advento das sinalizações de tipo, introduzidas na versão 3.5 da linguagem, é possível expressar de maneira mais clara as intenções do código, como no exemplo 3. É de suma importância destacar, entretanto, que essas sinalizações não tornam a linguagem fortemente tipada, mas apenas servem para facilitar entendimento. Dessa forma, ainda é perfeitamente possível invocar a função do exemplo abaixo com argumentos de tipos diferentes de *str*.

```
1  def concatenar(a: str, b: str) -> str:
2      return a + b
```

Código 3: Exemplo de função Python com sinalizações de tipo

2.1.2 Programação assíncrona com asyncio

Em se tratando de programação concorrente, a versão 3.5 da linguagem introduziu suporte a corrotinas através da sintaxe *async/await*. Corrotinas, segundo (FOUNDATION, 2015) são casos especiais de subrotinas que podem ter sua execução interrompida e continuada em diversos pontos, enquanto que subrotinas convencionais só possuem um ponto de entrada e um ponto de saída. Ao se orquestrar corrotinas de uma maneira centralizada, o programa como um todo se torna mais escalável pois corrotinas que estão presas a tarefas bloqueantes, como por exemplo um acesso a disco, podem ser suspensas em benefício de outras corrotinas que não estão bloqueadas. No Código 10, as corrotinas seriam as funções *hello* e *print_hello*, e a orquestração fica a cargo da biblioteca padrão *asyncio*.

```
1  import asyncio
2
3  async def hello():
4      return 'Hello World'
5
6  async def print_hello():
7      print(await hello())
8
9  if __name__ ==  '__main__':
10     asyncio.run(print_hello())
```

Código 4: Exemplo de código Python com sintaxe *async/await*

Antes do *async/await*, o suporte a corrotinas já existia através de uma sintaxe baseada em geradores (SELIVANOV, 2015), porém a introdução das palavras reservadas *async/await* proporcionou um modelo mental mais próximo da programação síncrona, enquanto que mantendo os benefícios de performance da programação assíncrona.

2.1.3 Bibliotecas gráficas para Python

2.1.3.1 *Tkinter*

Segundo a documentação oficial de *Tkinter*, disponível em: https://docs.python.org/3/library/tkinter.html (Foundation (2023)), o pacote *tkinter* é a interface Python padrão para as bibliotecas gráficas *Tcl/Tk*, adicionando uma quantidade considerável de lógica própria para tornar a experiência mais idiomática, em linha com os padrões da linguagem.

Ainda segundo Foundation (2023), *Tcl/Tk* (a biblioteca abstraída por *tkinter*) é composta por três módulos, resumidamente:

- *Tcl*: Uma linguagem dinâmica e interpretada, comumente embarcada em aplicações escritas na linguagem C;
- *Tk*: Um pacote *Tcl* que permite criar e manipular elementos de interface gráfica;
- *TTk*: Uma versão mais nova de *Tk* com componentes de aparência atualizada.

Internamente, *Tk* e *TTk* utilizam recursos do sistema operacional para criar elementos de interface. Portanto, segundo Foundation (2023) um fluxo de execução de *tkinter* em Python terá as seguintes etapas para que um elemento gráfico seja criado:

1. Em Python, é criada uma instância de elemento gráfico (como por exemplo um botão) a partir da API de *tkinter*;
2. *tkinter* cria um comando *Tcl/Tk* e o repassa para um executável nativo interno, que contém um interpretador *Tcl*;
3. O interpretador *Tcl* invoca o pacote *Tk*;
4. Finalmente, o pacote *Tk* chama recursos do sistema operacional para exibir o elemento que foi requisitado.

Um exemplo de janela criada com *tkinter* é incluído no pacote, podendo ser executado com o comando *python -m tkinter*, resultando na tela apresentada na figura 1.

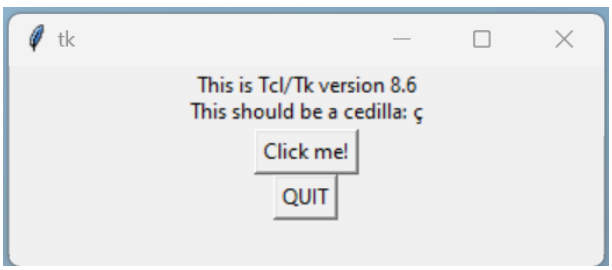

Figura 1 – Exemplo de janela *tkinter*, incluído no pacote.

A documentação provê ainda um exemplo de código para criação de uma janela, onde o código 5 resulta na tela apresentada na figura 2

```
1 from tkinter import *
2 from tkinter import ttk
3 root = Tk()
4 frm = ttk.Frame(root, padding=10)
5 frm.grid()
6 ttk.Label(frm, text="Hello World!").grid(column=0, row=0)
7 ttk.Button(frm, text="Quit", command=root.destroy).grid(column=1, row=0)
8 root.mainloop()
```

Código 5: Exemplo de código *tkinter*. Fonte: Foundation (2023)

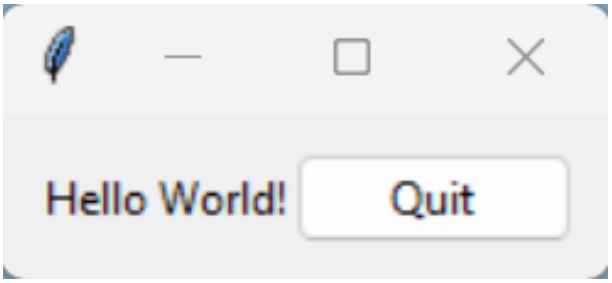

Figura 2 – Tela criada a partir do código 5.

2.1.3.2 *Pygame*

Pygame é, segundo sua documentação oficial disponível em https://www.pygame.org/wiki/about (Pygame (2023)), um conjunto de módulos Python para a criação de jogos, adicionando funcionalidades a uma biblioteca de mais baixo nível de abstração, a *SDL*. *Pygame* permite, além de jogos, a criação de programas multimídia.

Ainda segundo Pygame (2023), a ferramenta é compatível com diversos sistemas operacionais, gratuita e popularmente utilizada, tendo milhões de downloads. Conforme a documentação, funções internas são implementadas em *C* ou *Assembly*, linguagens de baixo nível que apresentam performance superior a soluções puramente Python.

Através de um exemplo disponibilizado na documentação e replicado no código 6, é possível compreender uma das principais características de *Pygame*, o *main loop*. No código, é possível observar que as funções de *Pygame* são chamadas pelo programa, e não o contrário (funções definidas pelo usuário sendo invocadas pela biblioteca).

```python
import sys, pygame
pygame.init()

size = width, height = 320, 240
speed = [2, 2]
black = 0, 0, 0

screen = pygame.display.set_mode(size)

ball = pygame.image.load("intro_ball.gif")
ballrect = ball.get_rect()

while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()

    ballrect = ballrect.move(speed)
    if ballrect.left < 0 or ballrect.right > width:
        speed[0] = -speed[0]
    if ballrect.top < 0 or ballrect.bottom > height:
        speed[1] = -speed[1]

    screen.fill(black)
    screen.blit(ball, ballrect)
    pygame.display.flip()
```

Código 6: Exemplo de *main loop* de *Pygame*. Fonte: Pygame (2023)

Figura 3 – Imagem resultante do código 6.

Na linha 13 do código 6 é criado um loop infinito, onde em cada iteração uma função de *pygame* é invocada para obter os eventos que ocorreram. Estes eventos, por sua vez, incluem: cliques, teclas pressionadas, movimentos de mouse e demais periféricos. Também podem ser definidos eventos customizados, permitindo o tratamento de eventos do domínio do programa.

2.2 WebSocket

WebSocket é um protocolo para comunicação bidirecional entre cliente e servidor (FETTE; MELNIKOV, 2011). Tradicionalmente, tal capacidade era emulada através de um uso descomedido do protocolo *HTTP*, onde o cliente repetidamente enviava requisições ao servidor para para verificar se houve mudanças no dado desejado. Esta abordagem, denominada *polling*, implica no uso de muitas conexões *TCP*, e na necessidade do cliente manter um controle lógico entre as diversas requisições e respostas.

O protocolo WebSocket teve sua padronização finalizada em 2011, na *RFC* 6455 (FETTE; MELNIKOV, 2011). Apesar de ser tradicionalmente associado a plataforma Web, nada impede seu uso em cenários onde o cliente não é um navegador. Em se tratando da filosofia por trás do design do protocolo, seus criadores buscaram desenvolver uma camada fina sobre o *TCP*, de forma que a maior sofisticação existe no processo de *handshake*, e a troca de mensagens em si é largamente delegada aos mecanismos existentes do *TCP*.

O processo de *handshake*, como descrito na *RFC* 6455 (FETTE; MELNIKOV, 2011), se inicia com uma requisição *HTTP* do tipo *Upgrade*, que indica ao servidor a intenção de se trocar para o protocolo WebSocket. Esse mecanismo explicita a capacidade de coêxistencia entre servidores WebSocket e *HTTP* convencional. Assim, para a miríade de aplicações Web que já existem, não é necessária a criação de configurações que fogem do padrão, por exemplo, de abrir a porta 80.

Uma vez realizado o *handshake*, a troca de mensagens em si se dá através de quadros sobre a conexão *TCP*. Como detalhado pela *RFC* 6455 (FETTE; MELNIKOV, 2011), deliberadamente existem poucos tipos possíveis de quadros, que inicialmente podem ser divididos em quadros de controle e quadros de dados. Os quadros de controle incluem o *Close*, o *Ping* e o *Pong*. O *Close* é utilizado para solicitar o encerramento da conexão, e pode incluir dados que explicam a razão do fim da conexão. A inexistência de um hipotético quadro *Open* se deve ao fato do handshake ocorrer, por exemplo, através do *HTTP*. Os quadros *Ping* e *Pong*, por sua vez, são utilizados para verificar a responsividade de uma das partes na conexão. Ambos podem ter conteúdo relacionado a aplicação, e o *Pong* necessariamente deve responder com os mesmos dados provenientes do *Ping*. Finalmente, os quadros de dados se dividem em quadros de texto e binário. Os quadros de texto contém *UTF-8* válido diretamente. Já os quadros binários têm conteúdo cuja interpretação fica

completamente a cargo da aplicação.

A vasta maioria dos detalhes de implementação do WebSocket são abstraídos por bibliotecas, de forma que os desenvolvedores acabam por não conhecer alguns dos conceitos previamente comentados, como por exemplo os quadros de controle. Particularmente relevante a este trabalho é a biblioteca Python *websockets*, que provê uma *API* simples para a criação de clientes e servidores WebSocket. No Código 7, é mostrado um exemplo de cliente. Percebe-se a ausência de referências a *handshake* e quadros, estes conceitos são abstraídos de forma que se expõe apenas métodos para conexão, envio e recebimento de mensagens.

```
1  import asyncio
2  import websockets
3
4  async def hello():
5      async with websockets.connect("ws://localhost:8765") as websocket:
6          await websocket.send("Hello world!")
7          await websocket.recv()
8
9  asyncio.run(hello())
```

Código 7: Exemplo mínimo de Cliente WebSocket através da biblioteca websockets

Fonte: Augustin (2021)

```
1  import asyncio
2  import websockets
3
4  async def echo(websocket, path):
5      async for message in websocket:
6          await websocket.send(message)
7
8  async def main():
9      async with websockets.serve(echo, "localhost", 8765):
10         await asyncio.Future()
11
12 asyncio.run(main())
```

Código 8: Exemplo mínimo de servidor de eco WebSocket através da biblioteca websockets

Fonte: Augustin (2021)

3 Trabalhos Relacionados

3.1 Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real (Netgames)

Também conhecido pelo nome de seus artefatos de software (Netgames), este trabalho foi desenvolvido por Brasil (2007) no contexto das disciplinas de Análise e Projeto Orientados a Objetos ministradas pelo professor Ricardo Pereira e Silva, onde o projeto final consiste no desenvolvimento de um jogo. Segundo Brasil (2007), identificou-se que a possibilidade de produzir jogos multi-jogador poderia ser um elemento de motivação para o desenvolvimento de bons projetos, porém as dificuldades inerentes ao desenvolvimento de software distribuído poderiam ser um impeditivo de implementação. Dessa forma surgiu este trabalho, com o objetivo de produzir artefatos de software capazes de abstrair algumas destas dificuldades.

Brasil (2007) aponta que o desenvolvimento teve entre suas premissas suportar a linguagem Java (utilizada nas disciplinas), e jogos sem tempo real, isto é, jogos onde o tempo para resposta a um evento não é definido, como por exemplo em uma partida de xadrez. A partir disto Brasil (2007) definiu a criação de dois artefatos: um servidor multi-jogos independente de lógica de jogo e um framework para acesso ao servidor, sendo este o componente a ser embarcado na base de código dos jogos para abstrair conceitos de rede referentes a comunicação com o servidor.

Para implementar a comunicação entre framework e servidor, Brasil (2007) considerou três tecnologias: *sockets*, *Java RMI* e *DualRPC*. Ainda segundo Brasil (2007), os dois primeiros se mostraram inadequados por motivos distintos: *sockets* são uma API de baixo nível, e *Java RMI* uma tecnologia verbosa, que requer uma série de passos para que se consiga realizar uma chamada remota. Dessa forma, Brasil (2007) optou pelo *DualRPC*, um framework Java para comunicação bidirecional com base em conexão TCP/IP contínua. *DualRPC* permite implementar a comunicação entre cliente e servidor através de poucas classes, e suporta chamadas síncronas e assíncronas. *DualRPC* teria ainda entre suas virtudes, segundo Brasil (2007), a possibilidade de conectar clientes de diferentes linguagens através de protocolos e formatos como *SOAP* e *JSON*. Dos pontos negativos, Brasil (2007) destaca a limitação no tamanho dos parâmetros trafegados (contornado pelo fato dos jogos geralmente necessitarem de pequenas mensagens, que representam uma mudança de estado) e o limite de 1500 (mil e quinhentas) conexões simultâneas.

A partir da escolha de *DualRPC* para a comunicação entre cliente e servidor, Brasil (2007) realizou o desenvolvimento dos componentes.

Em relação ao servidor desenvolvido por Brasil (2007), é possível destacar ser genérico o suficiente para suportar diversos jogos sem exigência de tempo real: múltiplas partidas de múltiplos jogos podem acontecer simultaneamente. Quanto ao framework de Brasil (2007), sua API utiliza o padrão de projeto *proxy* para representar a interface do servidor, abstraindo a utilização do *DualRPC*. Certos casos de uso da solução de Brasil (2007) ocorrem de forma síncrona (como a conexão), permitindo que em cenários de erro haja um um retorno imediato, no formato de exceções a serem tratadas na própria pilha de chamada. Outros casos de uso se dão de forma assíncrona, como o recebimento de uma jogada remota.

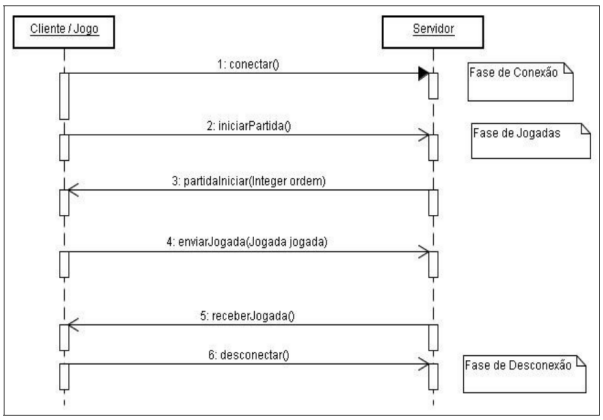

Figura 4 – Diagrama de seqüência do protocolo de Netgames. Fonte: Brasil (2007)

3.2 DOG - doing online games

Desenvolvido por Silva (2022) para suprir a impossibilidade de se utilizar Netgames com jogos desenvolvidos em Python, DOG é uma solução para criação de jogos como programas distribuídos, abstraindo elementos de rede para permitir o desenvolvimento tal qual em jogos centralizados. Disponibilizado e documentado em https://www.inf.ufsc.br/ ricardo.silva/dog/index.html (material utilizado de base para esta seção), DOG é composto por dois artefatos Python: um servidor e um framework a ser embarcado no código dos jogos (como ilustrado na figura 5).

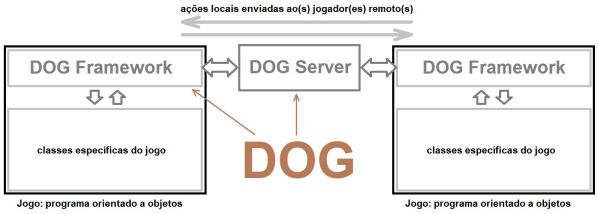

Figura 5 – Interação de artefatos de DOG. Fonte: Silva (2022)

O componente servidor difere do servidor de Netgames ao não ser disponibilizado diretamente, sua existência é abstraída pelo componente framework, que incorpora no caso de uso de conexão o endereço do servidor de forma transparente para o usuário.

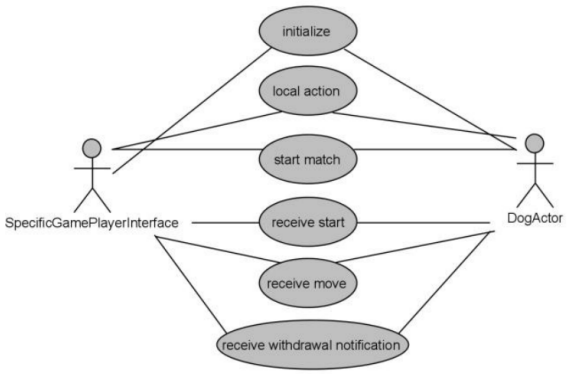

Figura 6 – Casos de uso de DOG. Fonte: Silva (2022)

Conforme a documentação de DOG Silva 2022, o artefato do framework é disponibilizado através de download direto, e deve ser incorporado a base de código dos jogos pela inclusão da pasta contento o código do framework. Uma vez realizada a importação, o que se encontra em termos de desenvolvimento é um conjunto de casos de uso muito mais enxuto do que o visto em Netgames, com etapas resumidas (conexão e solicitação de partidas incorporadas em um único caso de uso) ou até mesmo omitidas, como é o caso da desconexão, que fica à cargo do encerramento do programa como um todo.

Internamente a comunicação entre framework e servidor ocorre de forma síncrona através de chamadas HTTP. Como se trata de um código a ser embarcado em aplicações gráficas (os jogos), o bloqueio proveniente da espera por respostas do servidor acarretaria em atrasos de renderização. Para ações iniciadas localmente, como por exemplo a inicialização e o envio de jogadas, o bloqueio ocorre, e inclusive permite um tratamento de erro simplificado pois sobe exceções na pilha em caso de erro, porém para aguardar ações remotas essa estratégia é inviável pois bloquearia de forma indefinida. Isso é contornado através da criação de uma thread separada onde é realizado o polling para buscar eventos, conforme código 9.

Através da estratégia de polling implementada no código 9, DOG é capaz de implementar comportamento assíncrono no recebimento de jogadas, evitando bloquear a thread principal (e por consequência a interface gráfica) dos jogos que o incorporam. Supondo o status 3, onde se espera por um movimento remoto, uma vez por segundo será realizada uma chamada para o servidor (através de *self.proxy.match_status()*) até que se receba um movimento e o status seja alterado para aguardar a jogada local. Essa estratégia, enquanto que viável através da thread separada, pode vir a onerar um servidor

```python
from threading import Thread
import time


class PollingThread(Thread):
    def __init__(self, a_proxy, daemon_value):
        Thread.__init__(self, daemon=daemon_value)
        self.proxy = a_proxy

    def run(self):
        while True:
            status = self.proxy.get_status()
            if status == 2:  #   connected without match
                self.proxy.start_status()
            elif status == 3:  #   waiting remote move
                self.proxy.match_status()
            time.sleep(1)  # Sleep for 1 second
```

Código 9: Código de DOG responsável por polling. Fonte: Silva (2022)

no cenário de um número considerável de jogos em andamento.

3.3 Back-end de jogos educacionais

Desenvolvido por Arantes (2022), este trabalho apresenta um servidor capaz de orquestrar partidas de jogos multi-jogador sem tempo real, com um foco em jogos que possam ser utilizados em contexto educativo. A partir deste foco, o servidor desenvolvido por Arantes (2022) apresenta além dos conceitos esperados ao se tratar de jogos (como partidas e jogadas) também o conceito de usuários para alunos e professores.

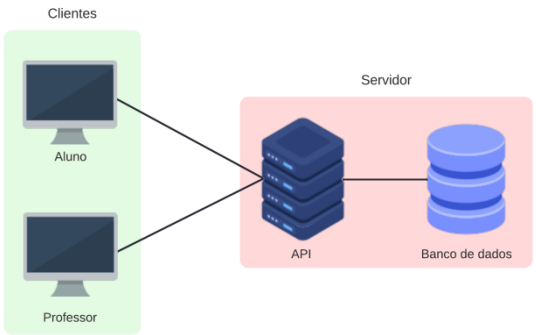

Figura 7 – Estrutura envisionada para uso do servidor. Fonte: Arantes (2022)

Conforme figura 7, Arantes (2022) desenvolveu neste trabalho o componente servidor (em vermelho). Implementado na linguagem *Typescritpt*, uma variação de *Javascript* com

suporte a sinalizações de tipo (semelhantes ao recurso de Python apresentado em 2.1.1), o servidor utiliza APIs HTTP para operações relacionadas a autenticação e gerenciamento de salas, enquanto que o decorrer de uma partida utiliza a biblioteca *Socket.IO*.

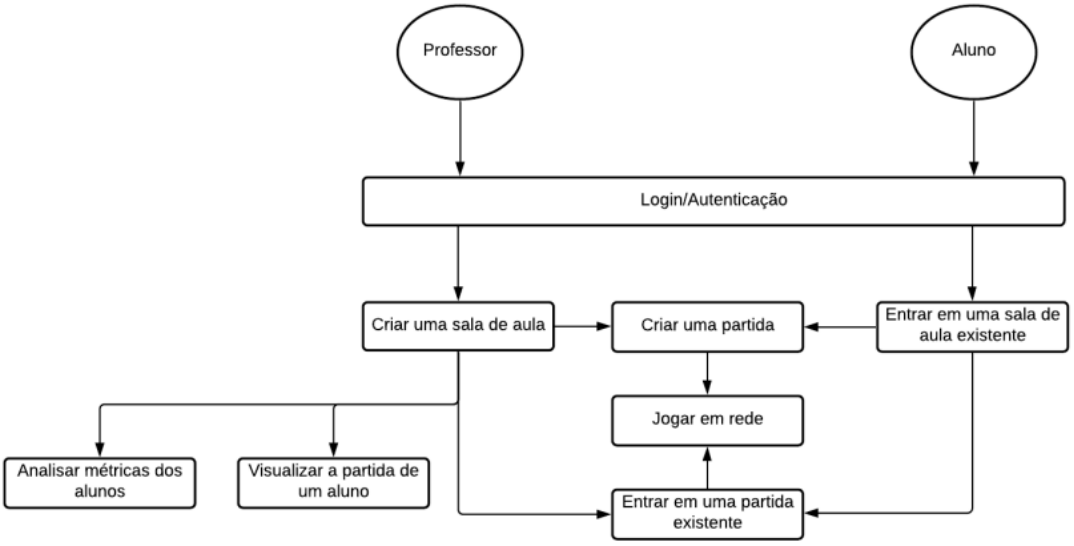

Figura 8 – Fluxo de uso do servidor. Fonte: Arantes (2022)

A figura 8 ilustra alguns casos de uso da solução desenvolvida por Arantes (2022), onde "Jogar em rede" utiliza a comunicação através de *Socket.IO*. *Socket.IO*, como explicado por Arantes (2022), abstrai comunicação bidirecional entre cliente e servidor através de duas estratégias internas: primariamente tenta utilizar WebSocket, e caso não consiga passa a utilizar *HTTP long polling*. *HTTP long polling* consiste em manter conexões HTTP convencionais até que o servidor possa responder, seguido imediatamente pela criação de uma conexão por parte do cliente.

3.4 Considerações

Nos anos que seguiram a publicação de Netgames, desenvolvido por Brasil (2007), a solução foi amplamente utilizado no contexto das disciplinas de Análise e Projeto de Sistemas da UFSC, com centenas de jogos criados ao longo de aproximadamente uma década de uso, até que a linguagem utilizada no curso passou a ser Python. Em que pese o fato de Brasil (2007) ter ponderado que *DualRPC* permitiria a criação de clientes em outras linguagens e dessa forma o reaproveitamento do servidor, com o tempo *DualRPC* se mostrou uma ferramenta inviável pela escassez de documentação, com a própria fonte para download citada no trabalho sendo inacessível em 2023, dessa forma fazendo-se necessário o desenvolvimento de novas soluções.

Em relação a DOG, de Silva (2022), se trata uma solução habilmente enxuta, capaz de atender a necessidade inicial, criação de jogos multi-jogador abstraindo conceitos de rede, com uma interface muito mais simplificada que Netgames. Enquanto que aparentemente ineficiente, a estratégia de realizar chamadas HTTP para comunicação com o servidor se torna versátil pois permite o tratamento síncrono de ações locais e, através de uma thread separada, o recebimento assíncrono de jogadas remotas. Pontos negativos desta estratégia se tornam menores ao considerar a pequena escala do uso da solução, que assim como Netgames é o contexto das disciplinas de Análise e Projeto de Sistemas. Entre pontos de melhoria é possível destacar a distribuição do software, onde a possibilidade de instalar através dos gerenciadores de pacote do ecossistema Python seria mais alinhado com o que é praticado em projetos atuais, e uma maior utilização de sinalizações de tipo no código poderiam facilitar a experiência de desenvolvimento ao integrar o framework.

Finalmente, assim como em Netgames e DOG, o servidor criado por Arantes (2022) é genérico em relação a jogos. Ao se iniciar uma partida, um dos parâmetros é o formato de dados que será usado em cada jogada. O servidor então cria uma tabela dinamicamente para armazenar as jogadas, com colunas mapeadas a partir do parâmetro de formato recebido. A principal diferença em relação aos trabalhos abordados anteriormente é o fato de não haver um framework que abstraia os elementos de programação distribuída: para se utilizar o servidor é necessário implementar as chamadas e conexões diretamente.

Em suma, os três trabalhos apresentados implementam servidores capazes de lidar com jogos multi-jogador sem tempo real. As principais diferenças se manifestam a partir do consumo deste servidor, e são destacadas na comparação abaixo (que inclui py-netgames, a solução proposta neste trabalho).

Nome	**Framework**	**Comunicação**
Netgames Brasil (2007)	Java	DualRPC (Síncrona e Assíncrona)
DOG Silva (2022)	Python	HTTP e HTTP polling (Síncrona e Assíncrona)
Arantes (2022)	N/A	HTTP e Socket.IO (Síncrona e Assíncrona)
py-netgames	Python	Websockets (Assíncrona)

Tabela 1 – Comparação de soluções

A existência de um framework para conexão com o servidor é essencial para a aplicação proposta neste trabalho, onde se faz necessário abstrair detalhes de programação distribuída. A possibilidade de comunicação síncrona permite um tratamento de erros mais natural na realização de ações locais (como enviar uma jogada), em que pese a necessidade de considerar o bloqueio de thread de interface gráfica. A assincronia para recebimento de jogadas é essencial e presente em todas as soluções. No que diz respeito a criação de novos clientes, somente Netgames, de Brasil (2007) é inviabilizado: enquanto que em tese DualRPC permitiria tal implementação, a escassez de documentação da ferramenta

passados tantos anos de seu uso inicial indica que Netgames é na prática limitado a plataforma que já foi implementado, o Java.

4 Solução Proposta

4.1 Visão geral

Para atender os objetivos descritos anteriormente, foi desenvolvida uma solução, denominada py-netgames em alusão ao Netgames, trabalho de Brasil (2007). Esta solução inicialmente é composta por três módulos e dois exemplos de aplicação, todos desenvolvidos em Python. Os módulos são:

- *py_netgames_client*: provê API de alto nível que abstrai o envio e recebimento de mensagens WebSocket com o componente *py_netgames_server*. De modo geral, este é o único módulo que precisa ser importado por um usuário que deseja usar py-netgames;
- *py_netgames_server*: responsável por conectar jogadores em partidas de um mesmo jogo. Pode ser executado remotamente ou provisionado de forma transparente pelo *py_netgames_client*;
- *py_netgames_model*: módulo importado internamente por *py_netgames_server* e *py_netgames_client*, define a estrutura das mensagens trafegadas via WebSocket (reservando campos genéricos para informações específicas de cada jogo).

4.1.1 Módulo *py_netgames_client*

Se trata do módulo voltado para o público, que precisa ser importado em jogos que desejam utilizar a solução. Dentro deste módulo são expostas duas APIs, uma baseada em composição e herança, pensada para uso com a biblioteca gráfica nativa do Python, o *tkinter*. Já a segunda API é específica para uso com a biblioteca gráfica *pygame* e funciona através do *event loop* da mesma. Dado que no contexto de uso da solução há foco em *tkinter*, aqui será aprofundado o funcionamento da API de *py_netgames_client* para *tkinter*, seguido por uma visão geral do funcionamento com *pygame*.

4.1.1.1 Casos de uso de *py_netgames_client*

Considerando a perspectiva de um usuário, os casos de usos são essencialmente os mesmos independente de qual biblioteca gráfica e API do *py_netgames_client* está sendo utilizada (*tkinter* ou *pygame*), mudando somente a implementação destes casos de uso (em termos de UML, modelagem de classes e diagramas de sequência). Abaixo o diagrama e o refinamento textual dos casos de uso de *py_netgames_client*.

Figura 9 – Diagrama de casos de uso de *py_ netgames_ client*

- Add Listener: Trata do registro do AtorLocal no framework, aqui indentificado por AtorPyNetgames, para que sejam recebidas atualizações remotas como o início de partidas e jogadas;
- Send Connect: Trata da conexão com o componente remoto (*py_ netgames_ server*). Diferentes execuções de um mesmo jogo precisam estar conectadas na instância de *py_ netgames_ server* para que seja possível a disputa de partidas. Dessa forma, é preciso que o usuário se atente ao endereço de servidor que será informado ao AtorPyNetgames. Como parte do desenvolvimento da solução, é disponibilizado uma instância de *py_ netgames_ server* em nuvem;
- Receive Connection: Trata da confirmação de sucesso de uma conexão solicitada no caso de uso Send Connect;
- Send Match: Trata da solicitação de uma partida para a quantidade de jogadores informada pelo AtorLocal. Uma vez que o *py_ netgames_ server* identifica que existem jogadores suficientes dada a quantidade solicitada, uma partida será iniciada (caso de uso Receive Match). A disntição entre diferentes jogos que se utilizam o framework se dá através um identificador gerado pelo AtorPyNetgames, e para que diferentes instâncias de um mesmo jogo conectem entre si é necessário que o usuário se atente ao identificador que está sendo usado;
- Receive Match: Trata do recebimento de uma partida, conforme solicitada no caso de uso Request Match. É recebido um objeto que contém um identificador da partida e um inteiro que indica a vez do jogador.

- Send Move: Trata do envio de um movimento para o componente remoto do framework. Importante destacar que o framework não realiza controle de "vez"dos jogadores, ficando isso a cargo da implementação do jogo com base na posição recebida no caso de uso Receive Match. Ao enviar o movimento, é necessário informar o id da partida, também recebido no caso de uso Receive Match;

- Receive Move: Trata do recebimento de uma jogada remota. O parâmetro recebido é um objeto que contém o identificador da partida, e os dados da jogada remota (conforme enviado pela outra instância do jogo);

- Send Disconnect: Trata da solicitação de desconexão com o componente remoto do framework. Caso não haja conexão em vigor, nada ocorre. Caso haja partida em andamento, os demais jogadores também serão desconectados;

- Receive Disconnect: Trata do recebimento de uma desconexão, seja ela solicitada (caso de uso Send Disconnect) ou recebida por desconexão de outros jogadores. O tratamento recomendado é voltar o jogo para um estado inicial;

- Receive Error: Trata do recebimento de eventuais erros que ocorram no framework. O tratamento recomendado é voltar o jogo para um estado inicial.

4.1.1.2 *py_netgames_client* com *tkinter*

A modelagem de classes da API de *py_ netgames_ client* para *tkinter*, na perspectiva de um usuário, se dá através do diagrama UML abaixo:

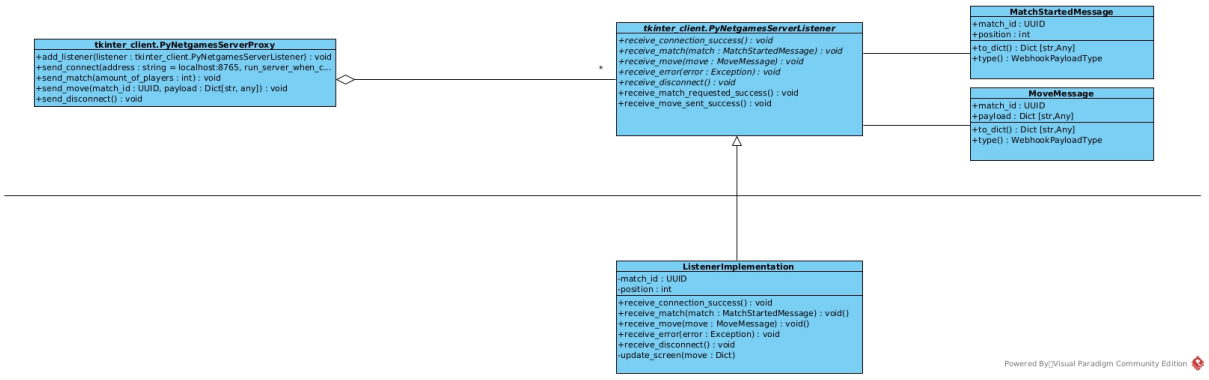

Figura 10 – Diagrama de classes de *py_ netgames_ client* para *tkinter*

- PyNetgamesServerListener: Classe abstrata que deve ser implementada e registrada a uma instância de PyNetgamesServerProxy. Responsável pelo recebimento de partidas, jogadas e confirmações de ações;

- ListenerImplementation: não faz parte do framework e representa apenas a implementação de PyNetgamesServerListener que se faz necessária em jogos que utilizem *py_ netgames_ client* com *tkinter*;
- PyNetgamesServerProxy: Utilizada para iniciar e finalizar conexões, solicitar partidas e enviar jogadas. Representa o componente remoto do framework;
- MatchStartedMessage: estrutura da mensagem recebida por instâncias de PyNetgamesServerListener no início de uma partida;
- MoveMessage: estrutura da mensagem recebida por instâncias de PyNetgamesServerListener no ato de uma jogada remota.

4.1.1.3 *py_netgames_client* com *pygame*

A modelagem de classes da API de *py_ netgames_ client* para *pygame* , na perspectiva de um usuário, se dá através do diagrama UML abaixo:

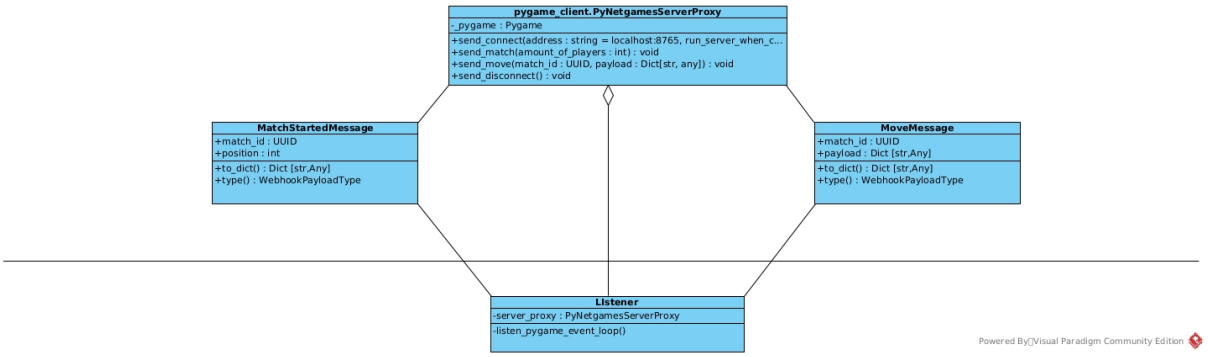

Figura 11 – Diagrama de classes de *py_ netgames_ client* para *pygame*

Em comparação ao *py_ netgames_ client* com *tkinter*, há uma notável ausência de métodos para receber ações remotas, como partidas e jogadas. Como, então, ocorre a implementação destes casos de uso? Em se tratando de *pygame*, o tráfego de informações ocorre através de um *event loop*, nada mais que um loop infinito (até que a aplicação seja interrompida) onde para cada iteração se obtém de *pygame* os eventos que aconteceram desde a última iteração. Estes eventos incluem, por exemplo, cliques de mouse e teclas acionadas no teclado, são identificados por tipo (um inteiro, geralmente definido em constante para facilitar identificação) e possuem um conteúdo, chamado de mensagem. Ciente deste mecanismo, a implementação de *py_ netgames_ client* para *pygame* define constantes de eventos próprios, atrelados aos casos de usos de recebimento de informações (como partidas e jogadas), e propaga eventos cuja qual mensagens são as de *py_ netgames_ client*, dessa forma atendendo aos casos de usos do componente e conformando com as práticas da biblioteca gráfica *pygame*. O trecho de código abaixo elucida uma possível implementação

do método *listen_pygame_event_loop* para exemplificar o funcionamento do *event loop* de *pygame* com *py_netgames_client*.

```
import pygame
from py_netgames_client.pygame_client.PyNetgamesServerProxy import
CONNECTED, CONNECTION_ERROR, DISCONNECTED, MATCH_REQUESTED, MATCH_STARTED

..............

def listen_pygame_event_loop(self):
    while self._is_running:
        events = pygame.event.get()
        for event in events:
            if event.type == CONNECTED:
                // Handle connection
            elif event.type == DISCONNECTED:
               // Handle disconnection
            elif event.type == CONNECTION_ERROR:
               // Handle connection error
            elif event.type == MATCH_REQUESTED:
                // Handle match request confirmation
            elif event.type == MATCH_STARTED:
                match_started_mesage = event.message
                // Handle match start
            elif event.type == pygame.MOUSEBUTTONUP:
                // Handle UI click

        pygame.display.update()
```

Código 10: Recebimento de eventos de *py_netgames_client* com *pygame*

No exemplo acima em cada iteração se obtém de *pygame* uma lista de eventos, que uma vez filtrada pelos tipos expostos por *py_netgames_client*, permite o tratamento das mensagens da solução. Demais eventos de interface gráfica, como um clique, também podem ser tratados, através de tipos expostos pelo próprio *pygame*.

4.1.1.4 Funcionamento interno de *py_netgames_client*

Para evitar a duplicidade de código entre as APIs de *pygame* e *tkinter*, é utilizada uma estrutura de herança, com uma superclasse comum à ambas as APIs. Esta superclasse, denominada *BaseWebsocketProxy*, abstrai elementos de mais baixo nível com a finalidade de facilitar o uso de *py_netgames_client*. Destas abstrações, é possível destacar:

4.1.1.4.1 Validações e mensagens de erro de fácil entendimento

Se tratando de uma aplicação destinada ao uso acadêmico, é importante que erros no uso da biblioteca sejam apresentadas de forma a facilitar a compreensão do usuário. Para tal, foram empregado os módulos nativos *warnings* e *logger*. Tomando por exemplo o cenário de uma requisição de partida antes da realização da conexão com o servidor, conforme código abaixo:

```
1  from py_netgames_client.tkinter_client.PyNetgamesServerProxy
2  import PyNetgamesServerProxy
3
4  if __name__ == '__main__':
5      pyNetgamesServerProxy = PyNetgamesServerProxy()
6      pyNetgamesServerProxy.send_match(amount_of_players=2)
```

Código 11: Requisição de partida sem conexão prévia

Da perspectiva de *py_ netgames_ client*, não há como seguir em frente com uma requisição de partida sem que haja uma conexão ativa. Para indicar corretamente o local de origem do erro, é utilizado o método *warnings.warn* com o parâmetro *stacklevel*. Isso resultará em uma mensagem de erro indicando diretamente o local em que a chamada para solicitar foi realizada sem a conexão ativa.

```
1  warnings
2      .warn(
3          f"Call to send_move when there is no active connection",
4          stacklevel=2
5      )
```

Código 12: Utilização interna de *warnings.warn*

```
1      __main__.py:5: UserWarning:
2      Call to send_match when there is no active connection
3       pyNetgamesServerProxy.send_match(amount_of_players=2)
```

Código 13: Mensagem de erro com referência ao código do usuário

Tal funcionamento permite que a biblioteca indique diretamente que se trata de um erro na utilização, e para o usuário fica evidente onde tal erro aconteceu. Isso difere de mensagens convencionais, onde o erro apresentado indicaria inicialmente uma linha de código interna da biblioteca.

4.1.1.4.2 Abstração do *asyncio* proveniente dos Websockets

A biblioteca utilizada para implementar a comunicação com o *py_ netgames_ server*, chamada *websockets* expõe APIs que utilizam a biblioteca de corrotinas disponibilizada pelo Python, o *asyncio*. De uma forma simplificada, corrotinas permitem que a execução de um código ocioso (aguardando algum retorno I/O, por exemplo de rede ou disco) seja suspensa a favor de outro trecho que já pode continuar, permitindo maior eficiência no uso do processador. Em termos práticos, isso se manifesta através da palavra reservada *async* na assinatura de uma função, conforme código abaixo:

```
1     async def example():
2       pass
```

Código 14: Assinatura *async*

A biblioteca *websockets* utiliza amplamente esse recurso, seja para iniciar uma conexão, desconexão ou enviar ou receber mensagens. Nesse último caso, isso é feito através de *async for*, onde para cada nova mensagem recebida ocorrerá uma iteração do loop:

```
1   async for message in websocket:
2     //handle message
```

Código 15: Loop *async for*

Ocorre, entretanto, que uma função *async* só pode ser chamada por outras funções *async*. E, como *py_ netgames_ client* se trata de um código que será importado para uso em outras bases de código (os jogos em si), impor o uso do *async* é inviável pois a interação do *asyncio* com bibliotecas gráficas como o *tkinter* é complexa. Dessa forma, para evitar que a assinatura dos métodos de *py_ netgames_ client* fossem *async*, *BaseWebsocketProxy* possui referência para um objeto *AbstractEventLoop*. No método construtor, esse objeto é inicializado conforme código abaixo (extraído trechos relevantes):

```
1     def __init__(self) -> None:
2         super().__init__()
3         self._loop = asyncio.new_event_loop()
4         def start_background_loop(loop: asyncio.AbstractEventLoop):
5             asyncio.set_event_loop(loop)
6             loop.run_forever()
7         Thread(target=start_background_loop, args=(self._loop,)).start()
```

Código 16: Inicialização de *Thread* e *AbstractEventLoop*

Este código cria e mantém uma referência para um loop de eventos em outra *Thread*, permitindo a invocação de métodos *async* da biblioteca *websockets* sem que haja

a propagação do *async* pela pilha de chamadas (o que afetaria os jogos que utilizam *py_netgames_client*). Abaixo o trecho de código relevante que realiza o envio de jogadas, e o mecanismo de abstração do *async*.

```
1  class BaseWebsocketProxy(ABC):
2      _loop: AbstractEventLoop
3      _websocket: Optional[WebSocketClientProtocol]
4
5      def send_move(self, match_id: UUID, payload: Dict[str, any]) -> None:
6          move_message = MoveMessage(match_id, payload).to_payload().to_json()
7          self._send(move_message, self._move_sent_success)
8
9      def _send(self, message: str) -> None:
10         async def async_send():
11             await self._websocket.send(message)
12
13         self._run(target=async_send)
14
15     def _run(self, target) -> Future:
16         return asyncio.run_coroutine_threadsafe(target(), self._loop)
```

Código 17: Envio de mensagem

Se define uma função interna que possui a assinatura correta (com *async*), e então delega-se a execução desta função interna para a referência do loop criada previamente. Essa estratégia permite o uso da biblioteca *websockets* sem a propagação do modificador *async*.

4.1.1.4.3 Provisionamento de *py_netgames_server*

Se tratando do módulo executado localmente nos jogos, *py_netgames_client* necessita de uma instância de *py_netgames_server* para o funcionamento correto, a partir do caso de uso Send Connect. Tal necessidade demandaria, no simples cenário de um teste local, a execução prévia do módulo servidor. Entretanto, para abstrair essa necessidade e facilitar o uso da solução, *py_netgames_client* possui *py_netgames_server* como dependência , e programaticamente provisiona uma instância local quando necessário.

```
1  def send_connect(
2      self,
3      address: str = "ws://localhost:8765",
4      run_server_when_connection_refused: bool = True
5  ) -> None:
```

Código 18: Declaração do método de conexão

No código acima, a declaração do método de conexão, é possível observar dois parâmetros com valores padrão, *address* e *run_server_when_connection_refused*. A combinação desses argumentos permite que um usuário invoque o método *send_connect* sem valores e obtenha um conexão com sucesso, que será feita a uma instância local do servidor provisionada a partir da inicial falha de conexão. Isso faz com que uma segunda instância do mesmo jogo, executada com os mesmos parâmetros de conexão, conecte-se com sucesso no servidor criado previamente, e seja possível a realização de uma partida local.

4.1.2 Módulo *py_netgames_model*

Se trata do módulo que define a estrutura das mensagens trafegadas, além das classes responsáveis por serialização e desserialização.

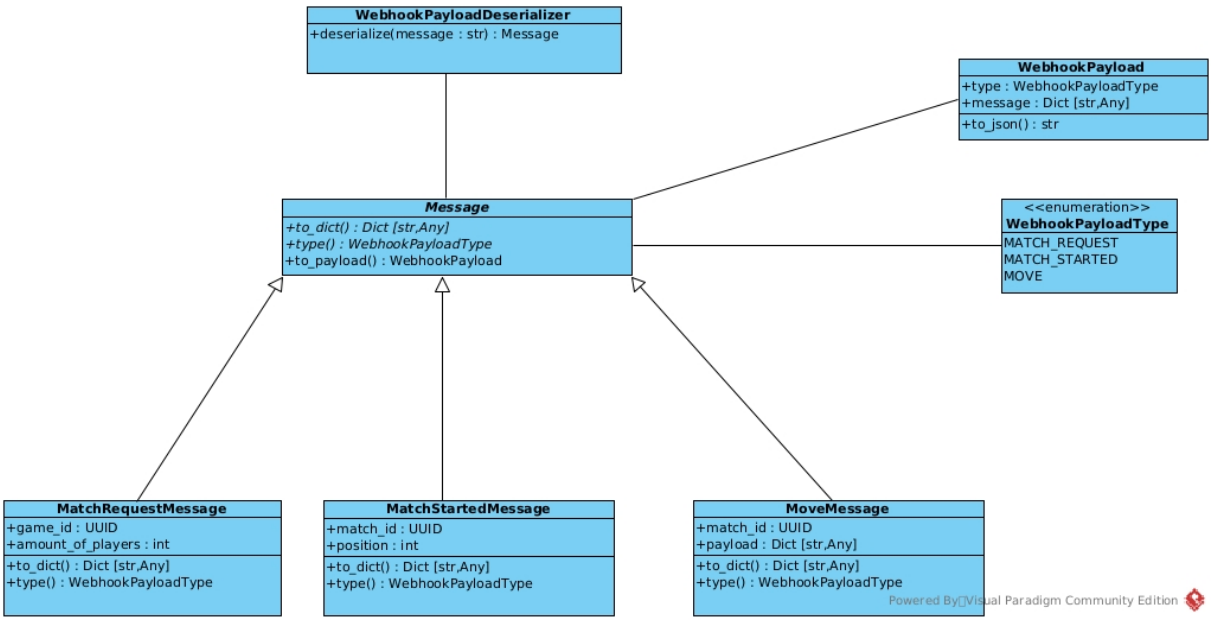

Figura 12 – Diagrama de classes de *py_netgames_model*

O módulo *py_netgames_model* é uma dependência tanto de *py_netgames_client* quanto *py_netgames_server*, dessa forma centralizando a lógica do tratamento de mensagens. Inicialmente, existem três classes concretas associadas aos casos de uso onde há o tráfego de mensagens: *MatchRequestMessage*, *MatchStartedMessage* e *MoveMessage*. Todas herdam da classe abstrata *Message*, e identificam qual seu tipo através do método *type*. Objetos da classe *Message*, são convertidos para a classe *WebhookPayload* mediante invocação do método *to_payload*, que por sua vez é convertida em uma string no formato *JSON* para finalmente trafegar no WebSocket.

4.1.2.0.1 Serialização

Para auxiliar na conversão de objetos Python em *JSON*, é utilizada a biblioteca *dataclasses_json*. Esta biblioteca expõe o *decorator* *@dataclasses_json*, que ao ser adicionado na declaração da classe, acrescenta uma implementação de método estático para conversão do objeto em *JSON*.

```
1  @dataclass_json
2  @dataclass
3  class WebhookPayload:
4      type: WebhookPayloadType
5      message: Dict[str, Any]
6
7      def to_json(self) -> str:
8          return WebhookPayload.to_json(self)
```

Código 19: Utilização de *@dataclasses_json*

Na linha 8 é possível observar uso do método gerado pela biblioteca, que ao receber uma instância própria retorna a conversão, assim contemplando a necessidade de serialização.

4.1.2.0.2 Desserialização

```
1  class WebhookPayloadDeserializer:
2      __deserialization_table: Dict[
3          WebhookPayloadType,
4          Callable[[Dict[str, Any]], Message]
5      ]
6
7      def __init__(self):
8          self.__deserialization_table = {
9              WebhookPayloadType.MATCH_REQUEST: MatchRequestMessage.from_dict,
10             WebhookPayloadType.MATCH_STARTED: MatchStartedMessage.from_dict,
11             WebhookPayloadType.MOVE: MoveMessage.from_dict
12         }
13
14     def deserialize(self, message) -> Message:
15         webhook_payload = WebhookPayload.from_json(message)
16         return self.__deserialization_table[
17             webhook_payload.type
18         ](webhook_payload.message)
```

Código 20: Lógica de desserialização

Para deserialização, o processo é semelhante. A classe *WebhookPayloadDeserializer* mantém uma tabela que associa o tipo de mensagem com o método que cria a instância do objeto correto a partir de um *dict*, a implementação deste método também proveniente da biblioteca *dataclasses_json*. Em um primeiro momento a mensagem recebida possui apenas um tipo e um dicionário de atributos, somente sendo possível a conversão para a implementação correta de *Message* através do atributo *type*.

4.1.3 Módulo *py_netgames_server*

Se trata do módulo responsável por coordenar o encaminhamento de mensagens entre instâncias que estiverem conectadas.

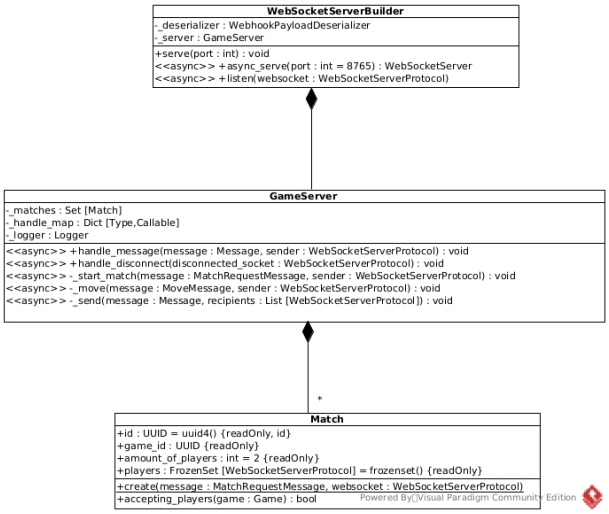

Figura 13 – Modelagem de *py_netgames_server*

Internamente é composto por três classes: *Match*, *GameServer* e *WebSocketServerBuilder*. *Match* representa uma partida, e contém entre seus atributos o identificador do jogo e os jogadores da partida, aqui representados diretamente pela conexão Websocket. Instâncias de *Match* são então agrupadas na classe *GameServer*, que contém a lógica do servidor para rotear as solicitações de partida e jogadas para as conexões corretas. De forma semelhante ao que ocorre em *py_netgames_client*, pode ser necessário abstrair o modificador *async* da invocação, neste caso, da inicialização do servidor. Para tal, *WebSocketServerBuilder* expõe dois métodos: *serve* e *async_serve*, que podem ser utilizados respectivamente quando o contexto da chamada é síncrono ou assíncrono. Tal opção é relevante a partir das formas que *py_netgames_server* pode ser utilizado ou provisionado, detalhadas abaixo.

4.1.3.0.1 Utilização programática

Como *py_netgames_server* é um módulo que pode ser importado de forma independente, uma das formas de executar é a partir de código. Assim faz o módulo

py_netgames_client, que tem *py_netgames_server* entre suas dependências, e o instancia a partir da classe *WebSocketServerBuilder*.

```
1           try:
2               await attempt_connection(address)
3           except ConnectionRefusedError as connection_refused_error:
4               if run_server_when_connection_refused:
5                   try:
6                       await WebSocketServerBuilder().async_serve()
7                       await attempt_connection("ws://localhost:8765")
8                   except Exception as e:
9                       return self._error(e)
10              else:
11                  return self._error(connection_refused_error)
12          except Exception as e:
13              return self._error(e)
```

Código 21: Utilização de *py_netgames_server* em *py_netgames_client*

A partir de uma falha de conexão em um endereço remoto, é provisionada uma instância do servidor. No 21, isso pode ser observado na linha 6, quando o método com modificador *async* da classe *WebSocketServerBuilder* é chamado. A opção pela versão assíncrona se dá pois a invocação ocorre dentro de uma função também *async*.

4.1.3.0.2 Utilização via pacote Python

Ao ser distribuído para ser utilizado como dependência em outros pacotes, o pacote de *py_netgames_server* também pode ser utilizado diretamente a partir de um ambiente com o pacote instalado. A sequência de comandos, partindo da instalação, é mostrada no código 22.

```
1           pip install py-netgames-server
2           python -m py-netgames-server
```

Código 22: Instalação e execução de *py_netgames_server*

No segundo comando, o parâmetro *-m* é do próprio Python e indica que será executado o arquivo *__main__.py* definido na raiz do pacote. No caso de *py_netgames_server* este arquivo é definido conforme o código 23.

```
1  def setup_logger(log_level):
2      logging.basicConfig(
3          format='%(name)s %(asctime)s %(levelname)-8s %(message)s',
4          datefmt='%Y-%m-%d %H:%M:%S'
5      )
6      logging.getLogger().setLevel(log_level)
7
8
9  def setup_arg_parser() -> argparse.ArgumentParser:
10     parser = argparse.ArgumentParser()
11     parser.add_argument(
12         '-d', '--debug',
13         help="Debug websocket connections",
14         action="store_const", dest="log_level", const=logging.DEBUG,
15         default=logging.ERROR
16     )
17     parser.add_argument(
18         '--host',
19         help="Server host",
20         dest="host", default="0.0.0.0"
21     )
22     parser.add_argument(
23         '-p', '--port',
24         help="Server port",
25         dest="port", default=8765
26     )
27     return parser
28
29
30 async def health_check(path, request_headers):
31     if path == "/health":
32         return http.HTTPStatus.OK, [], b"OK\n"
33
34
35 if __name__ == '__main__':
36     args = setup_arg_parser().parse_args()
37     setup_logger(args.log_level)
38     WebSocketServerBuilder().serve(args.host, args.port, health_check)
```

Código 23: __main__.py de *py_netgames_server*

Este arquivo, de suma importância para a execução do módulo de forma não programática, não é modelado na figura 13 por se tratar do ponto de entrada da aplicação e depender da condicional definida na linha 35 do código 23. Além do nome do arquivo e da condicional, para que a opção *-m* funcione o arquivo deve estar na raiz do pacote Python. Considerando a estrutura de arquivos conforme código 24, o arquivo *__main__.py* precisa estar no mesmo nível do *__init__.py*.

```
1    |-fly.toml
2    |-docker-compose.yml
3    |-README.md
4    |-Dockerfile
5    |-py_netgames_server
6    | |-websocket_server_builder.py
7    | |-__init__.py
8    | |-match.py
9    | |-__main__.py
10   | |-game_server.py
11   |-Pipfile.lock
12   |-pyproject.toml
13   |-LICENSE.txt
14   |-Pipfile
```

Código 24: Estrutura de arquivos de *py_netgames_server*

Também é possível observar no código 23, a definição de formato de *log* e a utilização da biblioteca nativa *argparse*. Esta biblioteca permite definir argumentos para parametrizar a execução a partir do terminal, e através de valores padrão estes argumentos podem ser opcionais. No caso de *py_netgames_server*, dois argumentos são definidos:

- *-d* ou *–debug*: Indica o nível dos logs para a execução. Caso não seja informado, por padrão são logados apenas erros;
- *–host*: Indica o endereço em que a aplicação irá responder, e por padrão utiliza o endereço local 0.0.0.0;
- *-p* ou *–port*: Indica a porta em que a aplicação irá responder a conexões, e por padrão utiliza a porta 8765.

4.1.3.0.3 Utilização via contêiner *Docker*

Como pode ser visto no código 24, na estrutura de arquivos de *py_netgames_server* existem dois arquivos relacionados a *Docker*: *Dockerfile* e *docker-compose.yml*. De uma forma simplificada, *Docker* é uma plataforma para a separação de aplicações da infraestrutura onde está sendo executada. No caso de *py_netgames_server*, isso permite que através da execução do *Dockerfile* (definido conforme código 25), seja possível obter uma instância de *py_netgames_server* independente do sistema operacional que esteja executando, desde que *Docker* esteja instalado e configurado.

A sequência apresentada pelo *Dockerfile* é simples: a partir de uma imagem pública que contém Python instalado (linha 1) é realizada a instalação e execução de *py_netgames_server* conforme parâmetros apresentados na seção 4.1.3.0.2. Para facilitar

```
FROM python:3.8-alpine

RUN pip3 install py-netgames-server

EXPOSE 8080

CMD [
    "python3",
     "-m",
     "py_netgames_server",
     "--debug",
     "--host",
     "0.0.0.0",
     "-p",
     "8080"
]
```

Código 25: *Dockerfile* de *py_netgames_server*

a execução do *Dockerfile*, é incluído também o arquivo *docker-compose.yml*, cuja qual definição abaixo referencia o próprio *Dockerfile* através do parâmetro *build*.

```
version: "3.7"
services:
  py-netgames-server:
    build: .
    ports:
      - "80:8080"
```

Código 26: *docker-compose.yml* de *py_netgames_server*

Por fim, a execução pode ser realizada com o comando *docker-compose up* (a partir da mesma pasta que contém *docker-compose.yml*), e resulta na instância de server disponibilizada.

```
$ docker-compose up
Creating network "server_default" with the default driver
Creating server_py-netgames-server_1 ... done
Attaching to server_py-netgames-server_1
py-netgames-server_1  | websockets.server 2023-10-03 09:26:47
    INFO server listening on 0.0.0.0:8080
```

Código 27: Execução de *py_netgames_server* com *Docker*

4.1.4 Utilização via endereço disponibilizado

Para facilitar o uso da solução além de testes locais (caso contemplado pelo provisionamento automático de *py_netgames_server* a partir de *py_netgames_client*), é disponibilizado um endereço onde há uma instância de *py_netgames_server* em execução. Isso permite partidas entre duas instâncias de um mesmo jogo através da internet, desde que ambas estejam conectadas no mesmo endereço.

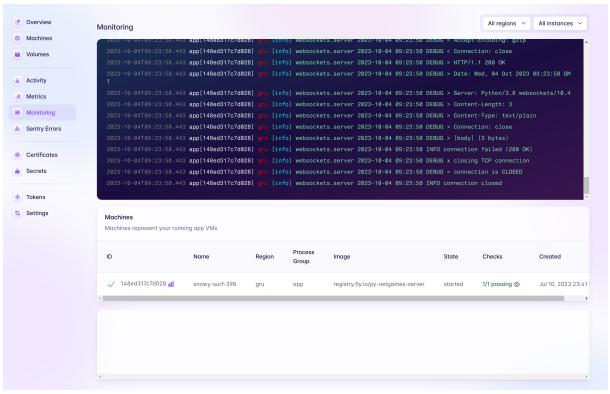

Figura 14 – Logs de execução de *py_netgames_server* na nuvem.

Através do *Dockerfile* apresentado anteriormente e da plataforma *fly.io*, foi possível disponibilizar uma instância sem grandes alterações de código. Como visto no código 24, há um arquivo *fly.toml* na raiz de *py_netgames_server*. Este arquivo define as configurações da plataforma, e tem o conteúdo abaixo.

```
1  app = "py-netgames-server"
2  kill_signal = "SIGTERM"
3
4  [[services]]
5    internal_port = 8080
6    protocol = "tcp"
7
8    [[services.http_checks]]
9      path = "/health"
10
11   [[services.ports]]
12     handlers = ["tls", "http"]
13     port = 443
```

Código 28: Configuração de *py_netgames_server* para uso na plataforma *fly.io*

Neste arquivo existem três referências para parâmetros que existem em código: *kill_signal*, *services.internal_port* e *services.http_checks.path*. O mais simples, *services.internal_port* se refere a porta configurada no *Dockerfile*. Já os demais parâmetros necessitaram pequenas alterações de código.

```
async def health_check(path, request_headers):
    if path == "/health":
        return http.HTTPStatus.OK, [], b"OK\n"

if __name__ == '__main__':
    args = setup_arg_parser().parse_args()
    setup_logger(args.log_level)
    WebSocketServerBuilder().serve(args.host, args.port, health_check)
```

Código 29: Alterações de *py_netgames_server* para uso na plataforma *fly.io*

O parâmetro *services.http_checks.path*, necessário para que a plataforma verifique se o contêiner da aplicação está rodando, foi contemplado com a configuração de uma API que responde com o código HTTP 200. Isso significa que na ausência de uma resposta, a plataforma assume que o contêiner está com problemas toma providências como o provisionamento de um novo.

```
def serve(self, host, port, health_check):
    stop = asyncio.get_event_loop().create_future()
    asyncio.get_event_loop().add_signal_handler(
        signal.SIGTERM, stop.set_result, None
    )
    asyncio.get_event_loop().run_until_complete(
        websockets.serve(
            self.listen, host, port, process_request=health_check
        )
    )
    asyncio.get_event_loop().run_until_complete(stop)
```

Código 30: Alterações de classe *WebSocketServerBuilder* para uso na plataforma *fly.io*

Já o parâmetro *kill_signal* indica qual sinal a plataforma pode enviar para o processo da aplicação para que aconteça o seu desligamento normal. Como foi configurado o *SIGTERM*, é criado um *signal_handler* (linha 3 do código 30) para "ouvir"a ocorrência do *SIGTERM*. Quando acontecer o envio do sinal, este será utilizado para encerrar o loop de eventos em que é executado o servidor *Websocket*. Este processo de sinalização para encerramento é criado na linha 2 e posteriormente configurado na linha 11 do código 30.

4.1.5 Exemplos de aplicação

Para exemplificar o uso da solução, foram desenvolvidos duas implementações de jogo da velha, uma para cada biblioteca gráfica inicialmente contemplada: *tkinter* e *pygame*.

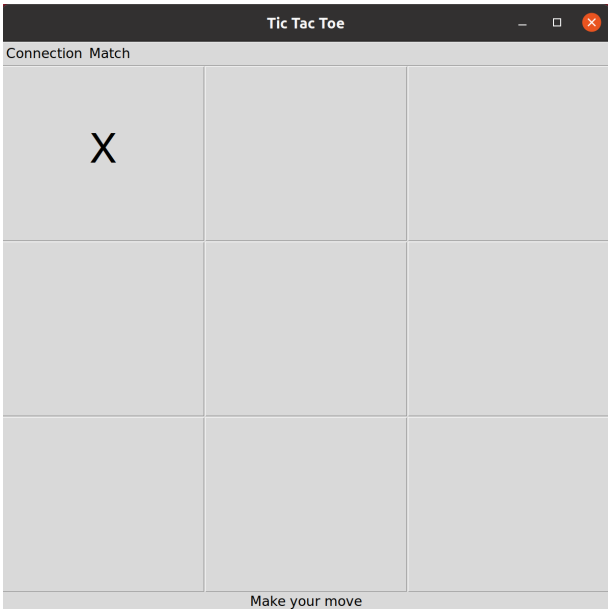

Figura 15 – Exemplo implementado em *tkinter*

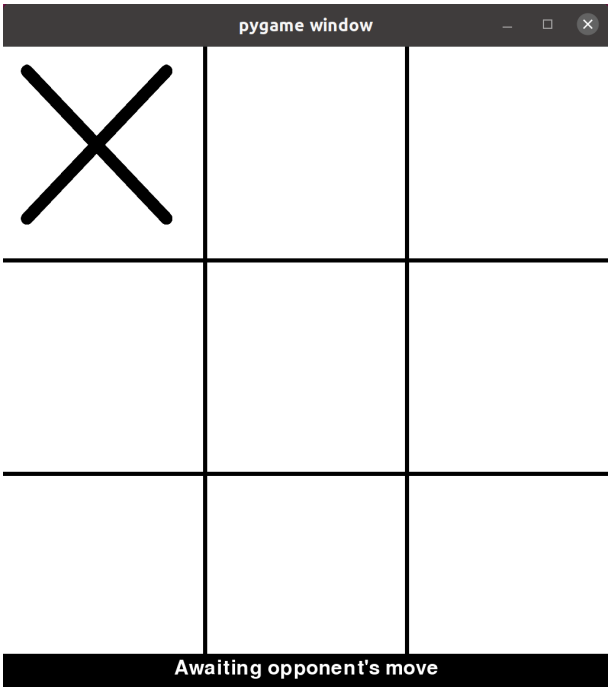

Figura 16 – Exemplo implementado em *pygame*

As implementações são interoperáveis entre si, isto é, é possível disputar partidas onde cada um dos jogadores utiliza uma implementação gráfica distinta (assumindo que o identificador de jogo de *py_netgames_client* seja o mesmo).

Todos os códigos apresentados neste capítulo estão disponibilizados em:

https://codigos.ufsc.br/gabriel.roza/py-netgames.

5 Resultados obtidos

5.1 Disponibilização e uso

Para viabilizar o uso da solução por parte de discentes, foi disponibilizada uma documentação de py-netgames, abordando detalhes de implementação, casos de uso e demais modelagens. Esta documentação permitiu o uso da solução através dos semestres 2023/1 e 2023/2 por parte de alunos das disciplinas de Engenharia de Software I e Análise e Projeto de Sistemas, respectivamente dos cursos de Ciências da Computação e Sistemas de Informação da Universidade Federal de Santa Catarina.

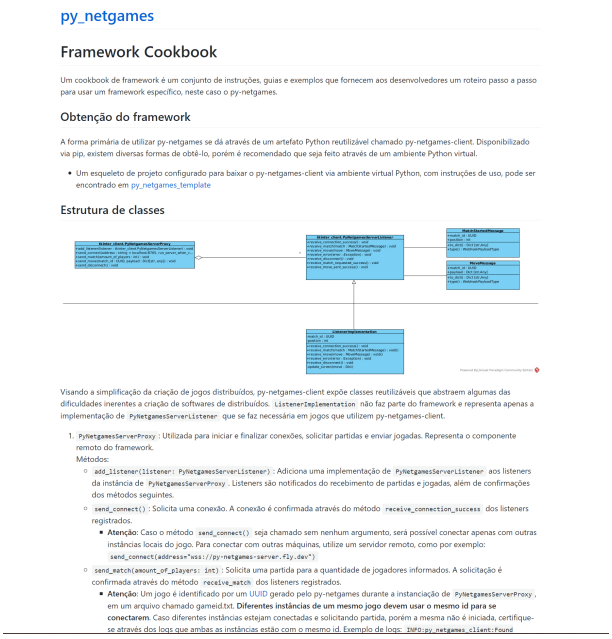

Figura 17 – Página de documentação

As disciplinas mencionadas possuem como entrega final a elaboração de um jogo multi-jogador. Além de py-netgames, os alunos tinham a opção de utilizar DOG, de Silva (2022), para implementar os componentes de rede. De um total de 31 jogos aprovados nos semestres 2023-1 e 2023-2, 13 utilizaram py-netgames.

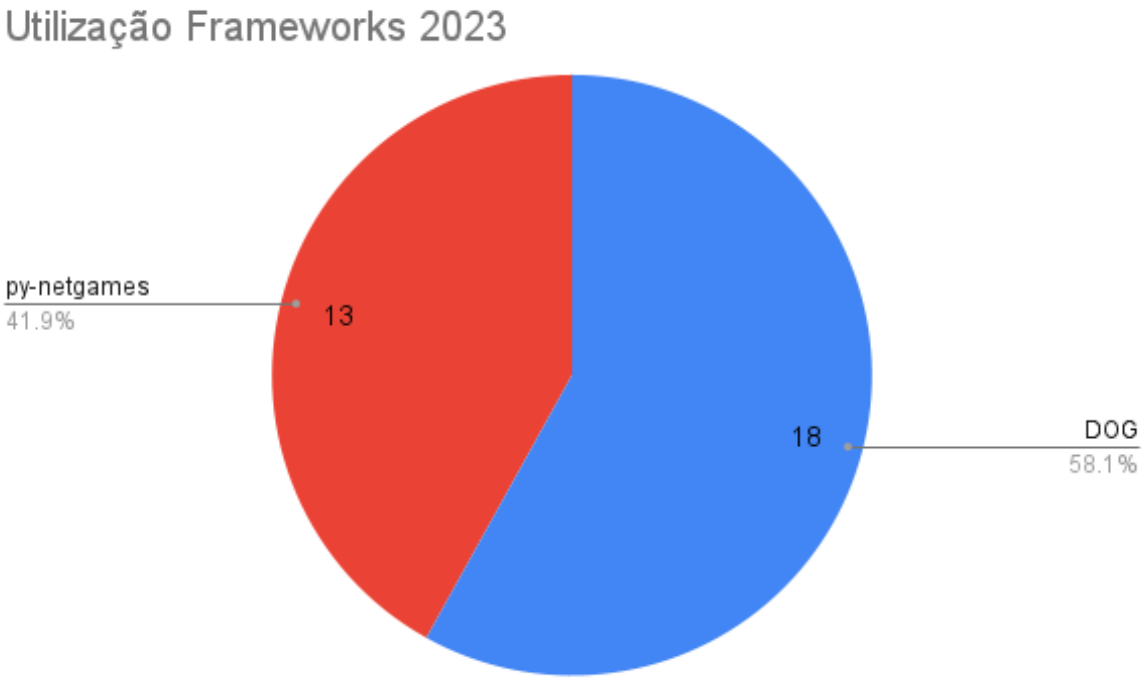

Figura 18 – Frameworks utilizados nos semestres de 2023

Os 13 jogos que utilizaram py-netgames são: Blackjack, Canastra, Connect 4, Go, Hex takeover, PyAdventure, Truco, Virtual Tabletop, Campo Minado, Yoté, Santorini, Mau Mau e Damas. O funcionamento de um destes, Connect 4, pode ser observado abaixo:

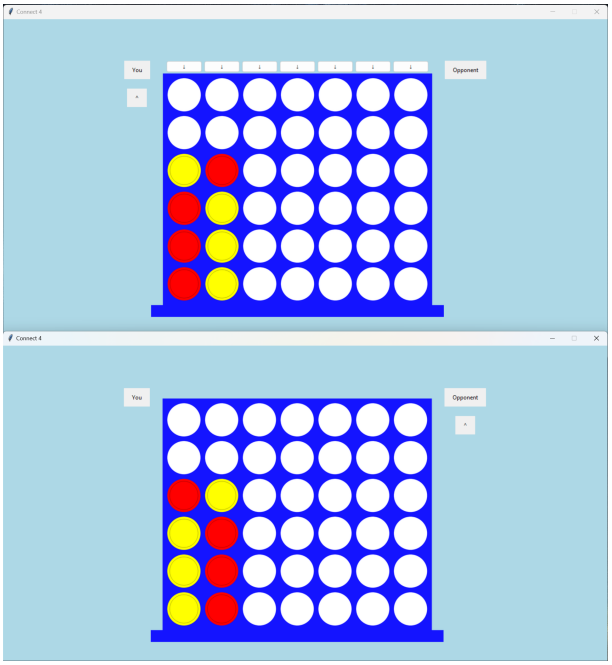

Figura 19 – Partida de Connect 4 em andamento, utilizando py-netgames.

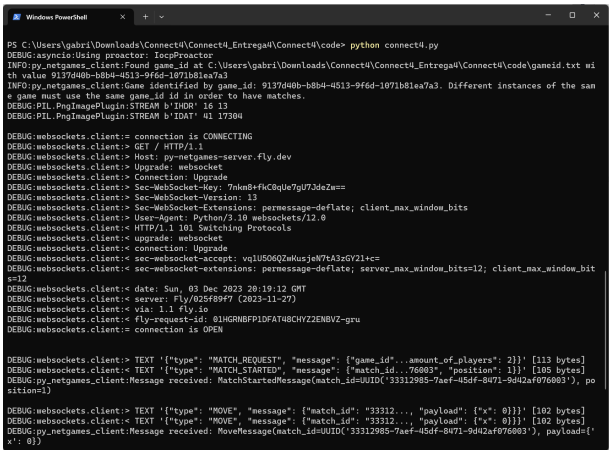

Figura 20 – Logs de py-netgames durante execução de Connect 4.

Durante o processo de desenvolvimento foi disponibilizado o e-mail do presente autor para auxiliar em eventuais dúvidas. Apesar disso os desenvolvimentos ocorreram sem necessidade de auxílio, evidenciando a suficiência da documentação disponibilizada.

Assim como demonstrado pela aprovação dos trabalhos e o continuado uso de py-netgames por mais de um semestre, os jogos funcionaram corretamente de forma distribuída, estabelecendo a ferramenta como uma opção para facilitar a implementação de jogos distribuídos no contexto das disciplinas.

6 Conclusão

O principal objetivo deste trabalho foi criar uma solução de software que replicasse as virtudes de Netgames, uma solução de software criada em 2007 por Brasil (2007) para facilitar o desenvolvimento de jogos multi-jogador sem exigência de tempo real. Enquanto tecnologicamente viável, Ao longo dos anos Netgames foi largamente utilizado em disciplinas dos cursos de Sistemas de Informação e Ciências da Computação da Universidade Federal de Santa Catarina, auxiliando desenvolvedores iniciantes a criar jogos distribuídos sem a necessidade de aprofundado conhecimento de conceitos como redes de computadores. O uso de Netgames viria a ser interrompido devido à sua incompatibilidade com a linguagem Python, que passou a ser utilizada nos cursos mencionados.

Suceder uma ferramenta que obteve tão extensivo uso ao longo dos anos não seria uma tarefa trivial, e para tal diversos conceitos foram abordados ao longo deste trabalho. Através da fundamentação foram apresentadas temas como a linguagem Python, algumas de suas características e bibliotecas, e WebSockets, protocolo que viria a ser essencial na solução desenvolvida.

Nos trabalhos relacionados foi possível aprofundar o entendimento de Netgames, e compreender a inviabilidade de seu reaproveitamento: DualRPC, o framework utilizado internamente, teoricamente suportaria uma nova plataforma, porém nos dias de hoje não há documentação suficiente que permita uma nova implementação. A partir desta impossibilidade, Silva (2022) desenvolve DOG com a mesma premissa de atender as disciplinas em que Netgames era utilizado. Observando trabalhos externos à Universidade Federal de Santa Catarina, foi encontrado um trabalho com foco em jogos educacionais, desenvolvido por Arantes (2022), em que um servidor genérico para jogos sem tempo real também foi criado. Comparando as soluções, o trabalho com foco em jogos educacionais é inviável por não possuir um componente capaz de abstrair os detalhes de programação distribuída. DOG, por sua vez, já é utilizado nas disciplinas, sendo possível identificar apenas alguns pontos em que a solução aqui desenvolvida poderia tratar de forma diferente.

A partir disto, é possível retomar os objetivos definidos na introdução e observar como foram atingidos.

6.0.1 Objetivos

6.0.1.1 Desenvolver um artefato servidor

Foi criado um artefato servidor, em Python, genérico ao suportar jogos multi-jogador sem tempo real. Este servidor utiliza o protocolo WebSockets para que seja

possível futuramente seu reaproveitamento em novas plataformas, caso se faça necessário. O servidor desenvolvido pode ser utilizado de diversas formas, desde programaticamente até via uma instância já provisionada na internet.

6.0.1.2 Desenvolver um artefato cliente

Desenvolver um artefato cliente que exponha uma API de alto nível, abstraindo detalhes da comunicação com o artefato servidor era o principal objetivo em termos de programação, ao considerar que este é o componente que é utilizado diretamente por estudantes. Aqui uma série de desafios foram encontrados, dos quais é possível destacar: elementos que facilitem o uso por parte de desenvolvedores iniciantes e a abstração da assincronia do protocolo WebSocket. Entre os elementos para facilitar o uso, a implementação traz uma gama de recursos como *logs*, *warnings* e sinalizações de tipo que buscam facilitar a incorporação do artefato em um jogo. Em relação a assincronia, foi necessário aprofundar o conhecimento em *asyncio* para chegar em uma solução que permita o uso do componente com bibliotecas gráficos que precedem o *asyncio*. Com estas implementações realizados, se tornou viável a publicação do artefato, disponibilizado em repositório Python público.

6.0.1.3 Integrar o artefato a uma aplicação existente

Integrar o artefato a uma aplicação existente, de forma a exemplificar o uso dos artefatos era o objetivo que permitiria aferir o funcionamento da solução, e para tal foi realizada a incorporação da solução em jogos desenvolvidos com as duas bibliotecas gráficas suportadas pelo artefato cliente, *Tkinter* e *Pygame*.

6.0.1.4 Documentar e disponibilizar os artefatos desenvolvidos

Considerando que se trata de uma solução para uso acadêmico, é importante que haja documentação capaz de suportar seu uso por parte de discentes. Para esta finalidade, foi disponibilizada de forma online, em https://gabrielroza.github.io/py_netgames/, uma série de modelagens, exemplos, e descrições capazes de auxiliar a utilização da solução sem a necessidade de intervenção do autor. A documentação criada também foi disponibilizada no repositório da instituição, em https://codigos.ufsc.br/gabriel.roza/py-netgames.

6.0.2 Utilização

A ferramenta criada, intitulada py-netgames, teve seu uso aferido nas mesmas disciplinas em que o Netgames original foi utilizado. Com diversos jogos criados utilizando py-netgames, a ferramenta de fato se tornou uma opção para facilitar o desenvolvimento de jogos distribuídos. Contrastando o objetivo geral de propor uma ferramenta baseada

em Python para auxílio ao desenvolvimento de jogos multi-jogador sem tempo real e a utilização apresentada, é possível concluir que o objetivo foi atingido.

Referências

ARANTES, M. S. Back-end de jogos educacionais. Universidade Estadual Paulista (Unesp), 2022. Citado 5 vezes nas páginas 9, 30, 31, 32 e 57.

AUGUSTIN, A. *websockets*. 2021. Disponível em: <https://websockets.readthedocs.io/en/stable/>. Citado na página 25.

BRASIL, L. d. S. Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real. 2007. Disponível em: <https://repositorio.ufsc.br/handle/123456789/183984>. Citado 7 vezes nas páginas 9, 27, 28, 31, 32, 35 e 57.

FETTE, I.; MELNIKOV, A. *The websocket protocol*. RFC 6455, December, 2011. Disponível em: <https://datatracker.ietf.org/doc/html/rfc6455>. Citado na página 24.

FOUNDATION, P. S. Coroutines. *Python Glossary*, 2015. Disponível em: <https://docs.python.org/3.5/glossary.html#term-coroutine>. Citado na página 20.

FOUNDATION, P. S. tkinter — python interface to tcl/tk. 2023. Disponível em: <https://docs.python.org/3/library/tkinter.html>. Citado 3 vezes nas páginas 11, 21 e 22.

PYGAME. *Pygame*. 2023. Disponível em: <https://www.pygame.org/wiki/about>. Citado 3 vezes nas páginas 11, 22 e 23.

SELIVANOV, Y. Pep 492–coroutines with async and await syntax. *Index of Python Enhancement Proposals*, 2015. Disponível em: <https://www.python.org/dev/peps/pep-0492/>. Citado na página 20.

SILVA, R. P. *DOG doing online games*. 2022. Disponível em: <https://www.inf.ufsc.br/~ricardo.silva/dog/index.html>. Citado 8 vezes nas páginas 9, 11, 28, 29, 30, 32, 53 e 57.

SRINATH, K. Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, v. 4, n. 12, p. 354–357, 2017. Disponível em: <https://www.irjet.net/archives/V4/i12/IRJET-V4I1266.pdf>. Citado na página 19.

TULCHAK, L.; MARCHUK, A. History of python. 2016. Disponível em: <http://ir.lib.vntu.edu.ua/bitstream/handle/123456789/10471/461.pdf?sequence>. Citado na página 19.

Apêndices

APÊNDICE A – Códigos

Todos os códigos referentes a implementação, incluindo documentação, podem ser encontrados em: https://codigos.ufsc.br/gabriel.roza/py-netgames.

APÊNDICE B – Artigo SBC

Modernização da ferramenta para criação de jogos em arquitetura cliente-servidor Netgames.

Gabriel Machado da Roza, Ricardo Pereira e Silva

[1]Departamento de Informatica e Estatıstica – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 5040 – 88040-900
Florianopolis - SC - Brazil

Abstract. *Developing client-server applications can be a challenging assignment for novice developers. As such, tools are created to facilitate this kind of task. This thesis proposes a new version of one such a tool, which focuses on non-real time multiplayer games. The proposed tool seeks, through the use of WebSocket, to be extentedable to new platforms in the future, should the need arise. The tool's usage was assesed through it's application in the subjects of Software Engineering I and Systems Analysis and Design, respectively of the Computer Science and Information Systems courses at the Federal University of Santa Catarina.*

Resumo. *O desenvolvimento de aplicações cliente servidor pode ser desafiador para iniciantes na programação. Assim, surgem ferramentas para auxiliar nesta tarefa. Este trabalho propõe uma versão modernizada para uma destas ferramentas, com foco no desenvolvimento de jogos multijogador sem tempo real. Utilizando-se do protocolo WebSocket, a ferramenta criada é futuramente extensível a novas plataformas. Foi possível aferir o uso da ferramenta criada através de sua utilização nas disciplinas de Engenharia de Software I e Análise e Projeto de Sistemas, respectivamente dos cursos de Ciências da Computação e Sistemas de Informação da Universidade Federal de Santa Catarina.*

1. Introdução

Desenvolvida por em 2007, a ferramenta Netgames auxilia no desenvolvimento de jogos multi-jogador sem suporte a tempo real, abstraindo conceitos de programas cliente-servidor através de uma API de alto nível, de forma que iniciantes na programação consigam criar seus jogos multi-jogador sem grandes dificuldades [Brasil 2007]. Composta por dois artefatos, um servidor e outro cliente (integrado ao código dos jogos), a ferramenta é genérica pois não é atrelada a lógica de nenhum jogo em particular [Brasil 2007]. Essas são características que contribuíram para seu contínuo uso acadêmico nos anos que se passaram desde sua criação. Há, entretanto, uma característica que nos tempos recentes dificulta sua aplicação: seus artefatos são fortemente atrelados a plataforma Java devido à escassez de documentação de seu mecanismo de comunicação, o framework DualRPC, o que inviabiliza o aproveitamento do componente servidor e a criação de um novo artefato cliente que faça interface com outras plataformas populares, como por exemplo Javascript ou Python. Dessa forma, este trabalho propõe a criação de uma nova ferramenta, capaz de reproduzir as virtudes do Netgames original em plataforma Python, enquanto que se atentando para que a tecnologia de comunicação utilizada, os WebSockets, não prejudique futuros desenvolvimentos.

2. Python

Python é uma linguagem de programação de alto nível, largamente utilizada e capaz de suportar múltiplos paradigmas, incluindo orientação a objetos, programação imperativa e funcional [Srinath 2017], criada em 1991 por Guido van Rossum. A execução de um programa Python se dá através de interpretadores (dos quais existem diversas implementações), seu sistema de tipos é dinâmico e o gerenciamento de memória é automático. Tais características, aliadas a uma filosofia de design da linguagem que preza pela legibilidade do código [Tulchak and Marchuk 2016], tornaram Python uma ferramenta adequada para introdução a programação, ultrapassando a linguagem Java [Srinath 2017].

Uma das características mais prestigiadas do Python é a suíte de ferramentas incluída por padrão: bibliotecas de matemática, manipulação de arquivos e serviços criptográficos são alguns exemplos. Conforme novas versões da linguagem são desenvolvidas, certas sofisticações, que vão além de meras funções são adicionadas

2.1. Programação assíncrona com asyncio

Em se tratando de programação concorrente, a versão 3.5 da linguagem introduziu suporte a corrotinas através da sintaxe *async/await*. Corrotinas são casos especiais de subrotinas que podem ter sua execução interrompida e continuada em diversos pontos, enquanto que subrotinas convencionais só possuem um ponto de entrada e um ponto de saída. Ao se orquestrar corrotinas de uma maneira centralizada, o programa como um todo se torna mais escalável pois corrotinas que estão presas a tarefas bloqueantes, como por exemplo um acesso a disco, podem ser suspensas em benefício de outras corrotinas que não estão bloqueadas [Foundation 2015]. No Código 1, as corrotinas seriam as funções *hello* e *print_hello*, e a orquestração fica a cargo da biblioteca padrão *asyncio*.

```
1
2   import asyncio
3
4   async def hello():
5       return 'Hello World'
6
7   async def print_hello():
8       print(await hello())
9
10  if __name__ ==  '__main__':
11      asyncio.run(print_hello())
12
```

Código 1: Exemplo de código Python com sintaxe *async/await*

2.2. Bibliotecas gráficas para Python

2.2.1. Tkinter

O pacote *tkinter* é a interface Python padrão para as bibliotecas gráficas *Tcl/Tk/TTk*, adicionando uma quantidade considerável de lógica própria para tornar a experiência mais idiomática, em linha com os padrões da linguagem [Foundation 2023].

Internamente, *Tk* e *TTk* utilizam recursos do sistema operacional para criar elementos de interface [Foundation 2023]. Portanto, segundo um fluxo de execução de *tkinter* em Python terá as seguintes etapas para que um elemento gráfico seja criado [Foundation 2023]:

1. Em Python, é criada uma instância de elemento gráfico (como por exemplo um botão) a partir da API de *tkinter*;
2. *tkinter* cria um comando *Tcl/Tk* e o repassa para um executável nativo interno, que contém um interpretador *Tcl*;
3. O interpretador *Tcl* invoca o pacote *Tk*;
4. Finalmente, o pacote *Tk* chama recursos do sistema operacional para exibir o elemento que foi requisitado.

2.2.2. Pygame

Pygame é um conjunto de módulos Python para a criação de jogos, adicionando funcionalidades a uma biblioteca de mais baixo nível de abstração, a *SDL* [Pygame 2023]. *Pygame* permite, além de jogos, a criação de programas multimídia [Pygame 2023],.

A ferramenta é compatível com diversos sistemas operacionais, gratuita e popularmente utilizada, tendo milhões de downloads [Pygame 2023]. Conforme a documentação, funções internas são implementadas em *C* ou *Assembly*, linguagens de baixo nível que apresentam performance superior a soluções puramente Python [Pygame 2023].

Uma das principais características de *Pygame* é o *main loop* [Pygame 2023], onde a partir de um loop infinito, em cada iteração uma função de *pygame* é invocada para obter os eventos que ocorreram. Estes eventos, por sua vez, incluem: cliques, teclas pressionadas, movimentos de mouse e demais periféricos. Também podem ser definidos eventos customizados, permitindo o tratamento de eventos do domínio do programa.

3. WebSockets

WebSockets é um protocolo para comunicação bidirecional entre cliente e servidor [Fette and Melnikov 2011]. Tradicionalmente, tal capacidade era emulada através de um uso descomedido do protocolo *HTTP*, onde o cliente repetidamente enviava requisições ao servidor para para verificar se houve mudanças no dado desejado. Esta abordagem, denominada *polling*, implica no uso de muitas conexões *TCP*, e na necessidade do cliente manter um controle lógico entre as diversas requisições e respostas.

O protocolo WebSocket teve sua padronização finalizada em 2011, na *RFC* 6455 [Fette and Melnikov 2011]. Apesar de ser tradicionalmente associado a plataforma Web, nada impede seu uso em cenários onde o cliente não é um navegador. Em se tratando da filosofia por trás do design do protocolo, seus criadores buscaram desenvolver uma camada fina sobre o *TCP*, de forma que a maior sofisticação existe no processo de *handshake*, e a troca de mensagens em si é largamente delegada aos mecanismos existentes do *TCP* [Fette and Melnikov 2011].

O processo de *handshake*, como descrito na *RFC* 6455, se inicia com uma requisição *HTTP* do tipo *Upgrade*, que indica ao servidor a intenção de se trocar para o protocolo WebSocket [Fette and Melnikov 2011]. Esse mecanismo explicita a capacidade

de coêxistencia entre servidores WebSocket e *HTTP* convencional. Uma vez realizado o *handshake*, a troca de mensagens em si se dá através de quadros sobre a conexão *TCP* [Fette and Melnikov 2011].

4. Netgames

Intitulado ”Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real”, porém conhecido por Netgames, é um trabalho que foi desenvolvido no contexto das disciplinas de Análise e Projeto Orientados a Objetos da Universidade Federal de Santa Catarina [Brasil 2007]. Nessas disciplinas, identificou-se que a possibilidade de produzir jogos multi-jogador poderia ser um elemento de motivação para o desenvolvimento de bons projetos, porém as dificuldades inerentes ao desenvolvimento de software distribuído poderiam ser um impeditivo de implementação [Brasil 2007]. Dessa forma surgiu Netgames, com o objetivo de produzir artefatos de software capazes de abstrair algumas destas dificuldades [Brasil 2007].

Netgames é composto dois artefatos Java: um servidor multi-jogos independente de lógica de jogo e um framework para acesso ao servidor, sendo este o componente a ser embarcado na base de código dos jogos para abstrair conceitos de rede referentes a comunicação com o servidor [Brasil 2007]. Para a comunicação entre cliente e servidor, Netgames utiliza um framework chamado *DualRPC* [Brasil 2007].

5. DOG - doing online games

Desenvolvido para suprir a impossibilidade de se utilizar Netgames com jogos desenvolvidos em Python, DOG é uma solução para criação de jogos como programas distribuídos, abstraindo elementos de rede para permitir o desenvolvimento tal qual em jogos centralizados [Silva 2022]. DOG é composto por dois artefatos Python: um servidor e um framework a ser embarcado no código dos jogos [Silva 2022].

O componente servidor difere do servidor de Netgames ao não ser disponibilizado diretamente, sua existência é abstraída pelo componente framework, que incorpora no caso de uso de conexão o endereço do servidor de forma transparente para o usuário [Silva 2022].

Internamente a comunicação entre framework e servidor ocorre de forma síncrona através de chamadas HTTP [Silva 2022]. Para que a consulta por novas jogadas não bloqueiea execução do programa, DOG utiliza uma Thread separada para realizar consultas no servidor [Silva 2022].

6. Back-end de jogos educacionais

Este trabalho apresenta um servidor capaz de orquestrar partidas de jogos multi-jogador sem tempo real, com um foco em jogos que possam ser utilizados em contexto educativo [Arantes 2022]. partir deste foco, o servidor apresenta além dos conceitos esperados ao se tratar de jogos (como partidas e jogadas) também o conceito de usuários para alunos e professores, porém não disponibiliza um framework para abstrair a conexão com o servidor [Arantes 2022].

7. Solução Proposta

Foi desenvolvida uma solução, denominada py-netgames em alusão ao Netgames original. A solução inicialmente é composta por três módulos e dois exemplos de aplicação, todos desenvolvidos em Python.

- *py_netgames_client*: provê API de alto nível que abstrai o envio e recebimento de mensagens WebSocket com o componente *py_netgames_server*. De modo geral, este é o único módulo que precisa ser importado por um usuário que deseja usar py-netgames;
- *py_netgames_server*: responsável por conectar jogadores em partidas de um mesmo jogo. Pode ser executado remotamente ou provisionado de forma transparente pelo *py_netgames_client*;
- *py_netgames_model*: módulo importado internamente por *py_netgames_server* e *py_netgames_client*, define a estrutura das mensagens trafegadas via WebSocket (reservando campos genéricos para informações específicas de cada jogo).

7.1. Módulo *py_netgames_client*

Se trata do módulo voltado para o público, que precisa ser importado em jogos que desejam utilizar a solução. Dentro deste módulo são expostas duas APIs, uma baseada em composição e herança, pensada para uso com a biblioteca gráfica nativa do Python, o *tkinter*. Já a segunda API é específica para uso com a biblioteca gráfica *pygame* e funciona através do *event loop* da mesma.

A biblioteca utilizada para implementar a comunicação com o *py_netgames_server*, chamada *websockets* expõe APIs que utilizam a biblioteca de corrotinas disponibilizada pelo Python, o *asyncio*. Para que o *asyncio* não se propague para o código do jogo, onde as bibliotecas gráficas não são compatíveis, *py_netgames_client* abstrai o *asyncio* através de um *event loop* interno.

7.2. Módulo *py_netgames_model*

Se trata do módulo interno que define a estrutura das mensagens trafegadas, além das classes responsáveis por serialização e desserialização. É uma dependência tanto de *py_netgames_client* quanto *py_netgames_server*.

7.3. Módulo *py_netgames_server*

Se trata do módulo responsável por coordenar o encaminhamento de mensagens entre instâncias que estiverem conectadas. A instanciação deste módulo é abstraída por *py_netgames_client*, porém *py_netgames_server* também pode ser utilizado diretamente como pacote Python.

7.4. Exemplos de aplicação

Para exemplificar o uso da solução, foram desenvolvidos duas implementações de jogo da velha, uma para cada biblioteca gráfica inicialmente contemplada: *tkinter* e *pygame*. As implementações são interoperáveis entre si, isto é, é possível disputar partidas onde cada um dos jogadores utiliza uma implementação gráfica distinta.

8. Resultados obtidos

Para viabilizar o uso da solução por parte de discentes, foi disponibilizada uma documentação de py-netgames, abordando detalhes de implementação, casos de uso e demais modelagens. Esta documentação permitiu o uso da solução através dos semestres 2023/1 e 2023/2 por parte de alunos das disciplinas de Engenharia de Software I e Análise e Projeto de Sistemas, respectivamente dos cursos de Ciências da Computação e Sistemas de Informação da Universidade Federal de Santa Catarina.

As disciplinas mencionadas possuem como entrega final a elaboração de um jogo multi-jogador. Além de py-netgames, os alunos tinham a opção de utilizar DOG para implementar os componentes de rede. De um total de 31 jogos aprovados nos semestres 2023-1 e 2023-2, 13 utilizaram py-netgames.

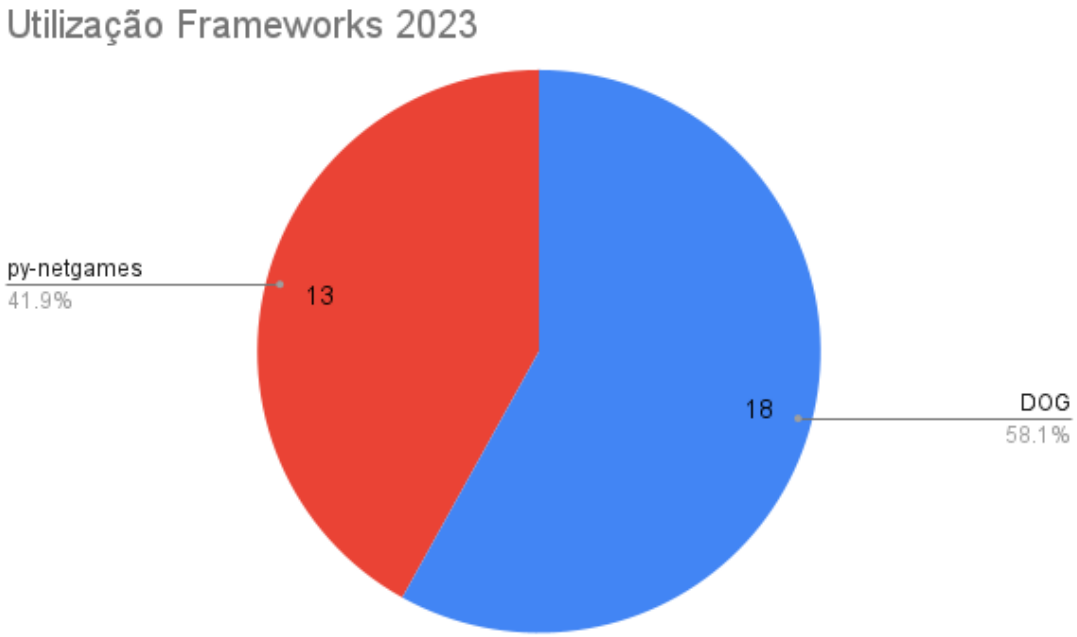

Figura 1. Frameworks utilizados nos semestres de 2023

Os 13 jogos que utilizaram py-netgames são: Blackjack, Canastra, Connect 4, Go, Hex takeover, PyAdventure, Truco, Virtual Tabletop, Campo Minado, Yoté, Santorini, Mau Mau e Damas. Durante o processo de desenvolvimento foi disponibilizado o e-mail do presente autor para auxiliar em eventuais dúvidas. Apesar disso os desenvolvimentos ocorreram sem necessidade de auxílio, evidenciando a suficiência da documentação disponibilizada.

Assim como demonstrado pela aprovação dos trabalhos e o continuado uso de py-netgames por mais de um semestre, os jogos funcionaram corretamente de forma distribuída, estabelecendo a ferramenta como uma opção para facilitar a implementação de jogos distribuídos no contexto das disciplinas.

Referências

Arantes, M. S. (2022). Back-end de jogos educacionais.

Brasil, L. d. S. (2007). Suporte ao desenvolvimento de jogos multi-jogador, sem exigência de tempo real.

Fette, I. and Melnikov, A. (2011). The websocket protocol.

Foundation, P. S. (2015). Coroutines. *Python Glossary*.

Foundation, P. S. (2023). tkinter — python interface to tcl/tk.

Pygame (2023). Pygame.

Silva, R. P. (2022). Dog doing online games.

Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357.

Tulchak, L. and Marchuk, A. (2016). History of python.