Escola de Artes, Ciências e Humanidades da Universidade de São Paulo

RELATÓRIO DO EP1 DE REDES DE COMPUTADORES

Turma 94

Júlia Du Bois Araújo Silva, 14584360

Oliver Koji Toyoki Kuramae, 14601326

São Paulo, SP

2024

Sumário

1. Proposta

Desenvolver uma aplicação que permite que duas pessoas joguem Connect4 por meio do terminal. Os protocolos iniciais propostos são mostrar o tabuleiro, iniciar uma partida, comunicar as jogadas entre os programas, enviar avisos entre os programas, realizar novas partidas no mesmo jogo e terminar o jogo.

2. Especificações

2.1 Arquitetura

O programa final é um híbrido entre Cliente-Servidor e Peer-to-Peer, possuindo características de ambos.

A principal característica do Cliente-Servidor está no fato que um dos jogadores lida com toda a lógica interna do jogo, enquanto o outro jogador atua como majoritariamente recebe e lida com as mensagens enviadas pelo primeiro jogador.

A característica Peer-to-Peer está no fato de que o programa que lida com a lógica interna do jogo (que é chamado de servidor neste trabalho) é um jogador assim como o programa cliente.

2.2 Organização do código

O arquivo `gameLibrary.py` é um arquivo Python com as lógicas de execução do jogo Connect4, em que se define as variáveis necessárias para executar o jogo como o tabuleiro e as variáveis dos jogadores e quem começa o jogo. Nesse arquivo pode se encontrar as funções de checagem de vitória, empate, input e visibilidade do tabuleiro.

Para mostrar o tabuleiro no terminal é uma iteração em uma matriz de 6 linhas e 7 colunas, estrutura utilizada para representar o tabuleiro. A biblioteca `termcolor` foi utilizada para printar as peças dos jogadores com cores.

A função de `input` filtra as respostas aceitas pelo jogo, no caso qual comunica se está adicionando uma peça, permitindo apenas números entre 0 e o tamanho máximo do tabuleiro (7 colunas).

Para a checagem da condição de vitória temos quatro funções responsáveis por verificar se um jogador possui quatro peças conectadas (vertical, horizontal, diagonal / e

diagonal \) que são verificadas toda vez que uma nova peça é adicionada por um jogador. Para as verificações na vertical e horizontal tem-se que a partir da última peça adicionada percorre-se a matriz na vertical e horizontal respectivamente. Enquanto para as diagonais percorre-se diagonalmente a matriz até que não se encontre mais peças pertencentes ao mesmo jogador. Essas verificações sempre respeitando o tamanho do tabuleiro e se as peças do mesmo jogador estão adjacentes.

O arquivo `gameServer.py` é um arquivo Python com a lógica do servidor. Por meio da função `serverProgram()`, este programa lida com toda a lógica de execução das funções do jogo, juntamente com o recebimento de inputs, a atualização de placares e o final de jogos.

Inicialmente, a função abre a uma porta selecionada pelo usuário, conectando-a a um socket e abrindo a possibilidade de conexão de um cliente. Após a conexão com o cliente, ele inicia o loop de jogos. Dentro desse loop, ele inicia um novo jogo, determinando de forma aleatória qual jogador deve começar por meio da função `gameStarter()` da `gameLibrary`. Em seguida, o jogo é continuado normalmente, alternando as jogadas do cliente e do servidor, até o tabuleiro estar completamente cheio ou houver uma vitória. Então, atualiza o placar de acordo com o resultado e pergunta aos dois jogadores se desejam continuar jogando. Caso os dois jogadores decidam continuar, o loop de jogos continua. Caso contrário, o jogo é encerrado e a conexão entre cliente e servidor é rompida.

O arquivo `gameClient.py` é um arquivo Python com a lógica do cliente. Esse arquivo possui duas funções, `clientProgram()` e `handleCommand()`. A primeira define a lógica de execução do cliente, recebendo as respostas do servidor, enquanto a segunda lida com as mensagens enviadas pelo servidor da maneira especificada posteriormente no subcapítulo 2.4.

A função `clientProgram()` inicia recebendo do usuário o host e a porta à qual deseja se conectar. Após a conexão, essa função entra em um loop para recebimento de protocolos até que seja sinalizado pelo servidor o final do loop de jogos.

2.3 Protocolo de mensagens

O protocolo de mensagens utilizado para as mensagens foi o TCP, como todos as comunicações entre cliente e servidor são essenciais, as informações trocadas são pequenas em tamanho e não pode haver alteração nos dados (considerando que qualquer mudança alteraria a mensagem, impedindo o funcionamento correto da função `handleCommand()` do

arquivo `gameClient`, mencionada anteriormente no subcapítulo 2.2), qualquer mudança ou falta de recebimento das mensagens obstruiria o funcionamento do jogo para os dois jogadores.

2.4 Protocolos de mensagens da aplicação

Essa aplicação possui 7 protocolos de mensagem principais, cuja semântica, gramática e funcionamento são descritos a seguir. Esses protocolos são utilizados na função `handleCommand()` de `gameClient`. Além desses protocolos, são descritas na seção 2.4.8 as mensagens enviadas do cliente para o servidor.

2.4.1 SHOWBOARD

SHOWBOARD segue a gramática padrão e seu funcionamento corresponde à função `showBoard()` da `gameLibrary`, que imprime o tabuleiro para o jogador. Essa mensagem é enviada pelo servidor ao cliente todas as vezes em que o servir chama a função de `gameLibrary` para que ambos os jogadores vejam o estado do tabuleiro. A sintaxe dessa mensagem é simplesmente "`SHOWBOARD`".

2.4.2 INITMATCH

INITMATCH notifica o cliente que uma nova partida. Como tanto o cliente quanto o servidor mantém um tabuleiro, é importante que esses sejam iguais para que ambos os jogadores saibam o estado atual da partida. Portanto, INITMATCH atualiza as informações para um novo jogo, limpando o tabuleiro anterior e registrando o novo placar de acordo com as informações enviadas pelo servidor. A sintaxe dessa mensagem é:

```
INITMATCH nomeDoServidor nomeDoCliente charDoServidor charDoCliente
        numeroPartida pontosDoServidor pontosDoCliente
```

Após o recebimento, a mensagem é separada por meio da função command.split, utilizando o espaço como discriminante para a alocação das informações e atualização dos valores em `gameClient`.

2.4.3 ASKPLAY

ASKPLAY é a mensagem utilizada para pedir ao cliente uma jogada. Após receber essa mensagem, o programa `gameClient` pede ao jogador a coluna na qual ele deseja posicionar sua peça, atualiza o próprio tabuleiro e repassa a coluna escolhida ao servidor. Ao receber essa resposta, o servidor atualiza o próprio tabuleiro. A sintaxe da mensagem ASKPLAY é simplesmente "`ASKPLAY`" e a sintaxe da resposta é simplesmente "`númeroDaColuna`".

2.4.4 REMOTEPLAY

REMOTEPLAY é a mensagem utilizada para notificar o cliente de uma jogada do servidor. A sintaxe dessa mensagem é:

```
REMOTEPLAY coluna
```

Após o recebimento, a mensagem é separada por meio da função command.split, utilizando o espaço como discriminante para a alocação das informações e atualização do tabuleiro de `gameClient`.

2.4.5 NOTICE

NOTICE é a mensagem utilizada para notificar o cliente que algo deve ser impresso na tela. A sintaxe dessa mensagem é:

```
NOTICE:texto para impressão
```

Após o recebimento, a mensagem é separada por meio da função command.split, utilizando "`:`" como discriminante, e imprimindo o restante da mensagem.

2.4.6 REPLAY

REPLAY é a mensagem utilizada para perguntar ao cliente se ele deseja jogar uma nova partida. Após o recebimento da mensagem, o programa entra em um loop até o cliente responder "`y`" ou "`n`", e envia essa resposta para o servidor. O servidor então avalia se o jogo deve continuar ou não, dadas as respostas de ambos os jogadores. A sintaxe da mensagem

REPLAY é simplesmente “REPLAY” e a sintaxe da resposta é “y” se for positiva ou “n” se for negativa.

2.4.7 QUIT

QUIT é a mensagem utilizada para notificar ao cliente que o jogo deve ser terminado. Ao receber a mensagem, o cliente encerra o loop de recebimento de instruções e fecha a socket utilizada para conexão. A sintaxe dessa instrução é simplesmente “QUIT”.

2.4.8 Outras mensagens

Outras mensagens trocadas, que não são interpretadas em handleCommand, são mensagens de seleção de personagem e de nome de jogador. Essas mensagens são trocadas logo antes do início do loop de jogos (ou seja, antes dos jogo começarem), e sua sintaxe é apenas o valor com o qual as variáveis devem ser preenchidas (ou seja, “character” no caso dos personagens e “nome” no caso dos nomes de jogador). Além dessas, existem também as respostas do cliente a certas mensagens do servidor, explicadas previamente.

2.5 Como rodar o programa

São necessárias as bibliotecas termcolor, time e socket para que o programa rode corretamente. Garanta que o programa gameLibrary.py está na mesma pasta que gameServer.py e gameClient.py.

Primeiramente, rode o programa gameServer.py. Coloque a porta que deseja utilizar e o nome do jogador que está com o servidor. Em seguida, rode o programa gameClient.py. Insira o IP do servidor em “Host” e a porta em “Port”. Os dois programas devem se conectar se utilizados corretamente. Caso a mesma máquina esteja sendo utilizada, o IP interno pode ser utilizado. Caso contrário, utilize o IPv4 da máquina usada como servidor.

Alternativamente, dois arquivos executáveis gameServer.exe e gameClient.exe estão disponíveis dentro da pasta dist do .zip entregue, que podem ser utilizados para executar o programa caso necessário. Após o início da execução, devem ser seguidos os mesmos passos do parágrafo anterior.

3. Testes e resultados

3.1 Diferentes sistemas e múltiplas máquinas

Esse programa foi testado em uma máquina (utilizando tanto o IP local quanto o IPv4) e em duas máquinas na mesma rede. Esses testes foram realizados nos sistemas operacionais Linux (na distribuição Ubuntu 24.04) e Windows 11 (versão 23H2), tanto com as duas máquinas no mesmo sistema quanto em sistemas diferentes, com o servidor e o cliente atuando em ambas os sistemas. Todos esses testes obtiveram sucesso e não foram encontrados erros. Apesar disso, é importante destacar que o programa funciona apenas se ambas as máquinas estiverem na mesma rede de internet, pois o programa não possui recursos para procurar o roteador e buscar o servidor em redes externas.

3.2 Lag

Os testes de lag estão relatados a seguir:

Lag (ms)	Lado	Resultado
0	servidor	sucesso
500	servidor	sucesso
1000	servidor	sucesso
1500	servidor	sucesso
2000	servidor	sucesso
2500	servidor	sucesso
3000	servidor	sucesso
3500	servidor	falha
0	cliente	sucesso
500	cliente	sucesso
1000	cliente	sucesso
1500	cliente	sucesso
2000	cliente	sucesso
2500	cliente	sucesso

3000	cliente	sucesso
3500	cliente	falha

Esses testes de lag foram realizados utilizando máquinas virtuais por meio do docker. Os comandos utilizados para esse teste podem ser encontrados no anexo 1.

3.3 Desconexões

Além disso, também foi testado o que ocorreria se um dos jogadores encerrasse a execução do programa subitamente. No caso do servidor encerrar subitamente, o cliente fica eternamente à espera de uma resposta. Isso pode ser consertado por meio de alguma sinalização para o cliente também encerrar a conexão ou com a implementação de um Dead Man's Trigger, que não foi implementada pela percepção do problema ter ocorrido próximo à data de entrega do trabalho. Caso o cliente encerre subitamente, o servidor recebe o erro "BrokenPipeError". Isso pode ser evitado com o uso de uma Exception que detecte tal erro.

4. Fontes

Socket Programming in Python: Client, Server, and Peer-to-Peer libraries - DEV Community

Socket Programming HOWTO — Python 3.13.0 documentation

Python Socket Programming - Server, Client Example | DigitalOcean

Socket Programming in Python (Guide)

Simulate high latency network using Docker containers and "tc" commands | by Kazushi Kitaya | Medium

How do I simulate a low bandwidth, high latency environment? - Stack Overflow

networking:netem [Wiki]

tc-netem(8) - Linux manual page

NetEm - Network Emulator at Linux.org

How to Use a Dockerfile to Build a Docker Image | Linode Docs

Why am I getting an RTNETLINK Operation Not Permitted when using Pipework with Docker containers? - Stack Overflow

5. Anexo

5.1 Comandos para teste de lag

5.1.1 Instalação do docker

```
sudo apt install docker.io
sudo apt install docker-compose
sudo apt install docker-compose-v2 docker-buildx
```

5.1.2 Build

```
sudo docker build . -t gserver -f docker/server/Dockerfile
sudo docker build . -t gclient -f docker/client/Dockerfile
```

5.1.3 Teste de lag do lado do servidor

Esse teste foi realizado utilizando dois terminais.

	Terminal 1 (servidor)	Terminal 2 (cliente)
1	`sudo docker run --rm -it --cap-add=NET_ADMIN --name serverLag gserver`	
2		`sudo docker exec serverLag tc qdisc add dev eth0 root netem delay 1500ms`
3		`sudo docker run --rm -it gclient`
4		`Host: 172.17.0.2`

5.1.4 Teste de lag do lado do cliente

Esse teste foi realizado utilizando dois terminais.

	Terminal 1 (servidor)	Terminal 2 (cliente)
1		`sudo docker run --rm -it --cap-add=NET_ADMIN --name clienteLag gclient`
2	`sudo docker exec clienteLag tc qdisc add dev eth0 root netem delay 1500ms`	
3	`sudo docker run --rm -it gserver`	
4		`Host: 172.17.0.3`