UAc
UNIVERSIDADE
DOS AÇORES

Sistemas Distribuídos

Distributed Tetris

Realizado por:
- Simão Nunes;
- Luís Viveiros;
- Mafalda Correia.

Docentes:
- Professor José Manuel Veiga Ribeiro Cascalho;
- Professor Francisco Cipriano da Cunha Martins.

Índice

Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular Sistemas Distribuídos, e teve como objetivo colocar em prática os vários conceitos lecionados durante as aulas. O projeto consiste numa aplicação baseada no jogo ***Tetris***. Contudo, para aplicar os conhecimentos relativos ao desenvolvimento de sistemas distribuídos, os alunos tiveram de criar uma versão **multi-jogador** do jogo original, onde vários jogadores podem interagir com o mesmo "tabuleiro" de jogo em simultâneo, tentando marcar mais pontos que os adversários enquanto houver espaço no tabuleiro. Foi necessário desenvolver ***middleware*** e um **protocolo** de comunicação para que o servidor do jogo conseguisse transmitir informações aos vários jogadores e para que os jogadores conseguissem interagir com o estado do jogo no servidor. Para isso, foi também necessário aplicar conhecimentos relativos à utilização de ***locks*** e ***threads*** para poder haver vários jogadores a jogar ao mesmo tempo, mas sem conflitos e/ou condições de corrida. Em algumas áreas, foi um projeto bastante complexo, e o grupo encontrou alguns obstáculos, mas no final, a ideia principal do projeto conseguiu ser concretizada.

Funcionamento do Jogo

O grupo tentou desenvolver o jogo mais parecido possível com a versão original, mas retirando algumas funcionalidades que seriam muito complicadas de implementar, ou que iriam consumir muito tempo e não eram pedidas nos requisitos do projeto. Como no jogo original, ao iniciar uma partida, o jogador recebe uma peça aleatoriamente selecionada da lista de peças. Essa peça é instanciada no topo do tabuleiro. Á medida que o tempo passa, a peça vai descendo até que assenta no limite inferior do tabuleiro ou em cima de outra peça já "trancada". O jogador consegue mover a peça para os lados e rodar tanto no sentido do relógio como no sentido inverso. Essas operações de movimento da peça não afetam a descida da mesma. Quando uma peça assenta e é bloqueada, o jogo verifica se alguma linha foi completada. Caso hajam linhas completadas, o jogador responsável pela peça que completou a linha recebe um número de pontos igual ao número de linhas completadas. De seguida, essas linhas são eliminadas e o tabuleiro adapta-se à sua falta, fazendo cair quaisquer outros blocos preenchidos até ocupar o espaço deixado pelas linhas eliminadas. O jogador recebe então outra peça e começa de novo. O jogo termina se alguma peça ultrapassa o limite superior do tabuleiro (acontece quando já existem muitas peça bloqueadas no tabuleiro) ou se todos os jogadores abandonarem a partida. Por fim, o jogo informa todos os jogadores de quem foi o vencedor (o jogador que tiver mais pontos no fim do jogo). O aspeto multi-jogador funciona no estilo *drop-in-drop-out*, em que os jogadores podem entrar e sair quando quiserem, e o jogo decorre mesmo só com um jogador.

Estrutura do Projeto

O projeto foi desenvolvido com duas ***source roots*** de modo a simular dois programas diferentes que correriam em máquinas diferentes. Temos programa **dt_server** e o **dt_ui**.

Dt_server:

É o servidor do jogo, que é composto por dois pacotes: o ***game*** (que contem a lógica do jogo/servidor) e o ***skeleton*** (que disponibiliza o *middleware* para realizar a comunicação com os jogadores).

Esta aplicação é composta pelas classes **Piece**, **Player**, **Match** e **Server**. As classes Piece e Player participam principalmente na classe Match, que contem a lógica principal do jogo. O Match por sua vez comunica com uma instância estática da classe Server, que contem métodos que transferem informações entre o *middleware* (*skeleton*) e o Match.

Dt_ui:

É a aplicação cliente, que o jogador usa para comunicar com o servidor e interagir com o jogo. O cliente é também composto por dois pacotes: o ***ui*** (que contem a lógica do cliente/jogador) e o ***stubs*** (que contem o *middleware* para que o cliente consiga interagir com o servidor e com o jogo).

Através do *stub*, uma instância da classe **Client** consegue comunicar com o servidor para enviar instruções do movimento das peças ou para receber atualizações do estado do jogo. Também teríamos uma classe **Gui**, onde estariam alojadas as regras para a atualização da interface gráfica, mas esse foi um dos aspetos do projeto que não conseguimos desenvolver.

Módulos/Pacotes

ZMQ (pyzmq)

O ZMQ foi um dos pacotes mais importantes para o desenvolvimento deste projeto. Escolhemos trabalhar com o **ZMQ** em vez do pacote **socket** porque o ZMQ é mais fácil de configurar e traz algumas funcionalidades de mais alto nível, eliminando algumas preocupações que teríamos de ter durante a programação dos aspetos multi-jogador do projeto.

```
context = zmq.Context()

print("REPREQ: Connecting to game server")
self.conn_reqrep = context.socket(zmq.REQ)
self.conn_reqrep.connect("tcp://" + self.host + ":" + str(self.port_reqrep))

print("PUBSUB: Connecting to game server")
self.conn_pubsub = context.socket(zmq.SUB)
self.conn_pubsub.connect("tcp://" + self.host + ":" + str(self.port_pubsub))
```

As comunicações feitas através do ZMQ funcionam já de maneira assíncrona, o que reduziu a necessidade da utilização de threads. Para a troca de mensagens entre os clientes e o servidor, criamos dois canais: um ***Request-Reply*** e um ***Publisher-Subscriber***. O *request-reply* é usado para comunicações em que o cliente envia uma instrução para o server e o server retorna uma resposta. É o tipo de comunicação mais comum no projeto. O *publisher-subscriber* é usado em conjunto com threads para que o server consiga informar periodicamente os clientes acerca de alterações no seu estado (enviar o tabuleiro com a passagem do tempo, enviar pontuações, etc...). Esses dois padrões de comunicação foram bastante úteis e o ZMQ facilitou bastante o desenvolvimento do projeto.

Threading

Com o módulo **Threading** usamos os componentes **Thread**, **Timer** e **Lock**. As *threads* (por fezes em conjunto com *timers*) foram usadas para permitir operações paralelas. Devido às vantagens do **ZMQ** (comunicação assíncrona), não foram necessárias muitas *threads*. Existe uma *thread* no cliente que server para pedir atualizações do tabuleiro ao server, e uma *thread* com *timer* nos objetos **Match** para executar a passagem do tempo.

Time

O pacote **time** foi usado em algumas partes do projeto para que se pudessem executar algumas instruções de forma repetida em intervalos de tempo.

Pynput.keyboard

Para facilitar o *input* de instruções do jogador, o programa cliente utiliza o componente **keyboard** do pacote **Pynput.** Esse módulo permite criar um *listener* que escuta eventos do teclado. Assim, o jogador pode executar instruções de maneira mais ágil sem termos de recorrer ao método **input** do Python.

```
# Inicia o listener do teclado
def send_command(self):
    # while True:
    with Listener(on_press=self.on_press) as listener:  # Create an instance of Listener
        listener.join()  # Join the listener thread to the main thread to keep waiting for keys

# Corre quando o listener do teclado deteta uma tecla a ser pressionada
# Determina o que cada tecla faz
def on_press(self, key):
    if self.in_game:
        # print("Key pressed: {0}".format(key))
        if key.char == "a":
            self.print_board(self.format_board(self.server.move_left(self.name)))
        elif key.char == "d":
            self.print_board(self.format_board(self.server.move_right(self.name)))
        elif key.char == "e":
            self.print_board(self.format_board(self.server.rotate_right(self.name)))
        elif key.char == "q":
            self.print_board(self.format_board(self.server.rotate_left(self.name)))
        elif key.char == "x":
            self.server.disconnect(self.name)
            self.in_game = False
```

Tkinter

O pacote **Tkinter** foi explorado e seria usado para desenvolver a interface gráfica do jogo. É possível encontrar no projeto alguns ficheiros relativos a essa parte, mas o grupo inicialmente teve algumas dificuldades em ligar a lógica do jogo com a interface gráfica. Acabamos por investir mais tempo nas partes mais importantes do projeto, e não concluímos o *UI.*

Classes

Player

A classe **Player** representa os jogadores. Cada jogador tem um nome, o seu *score* e a uma referência para a peça que lhe pertence de momento. Quando um cliente tenta validar um nome de jogador e é bem sucedido, o servidor cria um novo jogador e associa esse jogador ao *match* (passando o novo jogador para a lista de jogadores da partida). Á medida que a partida decorre, são atribuídos pontos aos jogadores que completam linhas. Cada vez que um jogador bloqueia uma peça, é lhe atribuída uma nova peça. No final da partida, os jogadores são removidos da lista de jogadores do *match*. Quando um jogador decide sair da partida, é também removido da lista de jogadores da partida.

```python
class Player:
    def __init__(self, name):
        self.name = name
        self.score = 0
        self.active_piece = None

    def add_score(self, points):
        self.score += points

    def get_score(self):
        return self.score

    def get_name(self):
        return self.name

    def set_active_piece(self, piece):
        self.active_piece = piece

    def get_active_piece(self):
        return self.active_piece
```

Piece

A classe **Piece** contém todas a formas de rotação de uma peça, assim como as instruções e métodos de teste que permitem mover e rodar uma peça. Existem cinco métodos *check*: três para verificar os limites laterais e o limite inferior do tabuleiro antes de efetuar movimentos. Existem dois para verificar se existem limites do tabuleiro ou peças que possam impedir rotações. Existem também os métodos para mover a peça para os lados e para baixo, assim como o método para rodar a peça.

Match

Um objeto **Match** é responsável por gerir uma partida. Contém o estado do tabuleiro, a lista de jogadores e informação relativa a todas as peças jogáveis. Um *match* é instanciado quando o primeiro jogador se autentica no servidor, e a partida é logo iniciada. O objeto *match* trata de iniciar os *timers*, atribuir peças aos jogadores, processar a passagem do tempo, fazer alterações ao tabuleiro e decidir quem vence a partida com base nos pontos marcados. Quando a partida termina, o *match* é dado como terminado e é eliminado. Quando o servidor recebe comunicações através do seu *skeleton*, o servidor comunica principalmente com o *match*.

Server

A classe **Server** trata de transmitir algumas informações entre o *match* e o cliente. Alguns dados recebidos pelo *skeleton* são processados na classe Server e reencaminhados para o *match*, enquanto outras informações são interpretadas e respondidas logo pelo *skeleton*.

Server_skeleton

O *skeleton* é responsável por receber e enviar informações de e para o *stub* do cliente. Para perceber que tipo de informações é que está a receber, foram definidos certos comandos no ficheiro **__init__.py** do pacote **skeletons**. Quando o *stub* do

cliente envia alguma mensagem, envia juntamente um desses comandos em forma de *string*. Quando o *skeleton* recebe a mensagem, vê qual é o comando e reencaminha para o método (da classe **Server**) que lida com esse tipo de instruções.

Client

O cliente é a interface entre o jogador e o *match.* O jogador envia instruções através do cliente para o servidor e o servidor executa essas instruções. O jogador consegue principalmente autenticar-se para jogar, enviar pedidos de movimentação e rotação da sua peça e pedir para se desconectar da partida.

Client_stub

O *stub* do cliente é o ponto de comunicação com o exterior no que toca ao cliente. As instruções inseridas pelo jogador são comunicadas através do *stub* para o *skeleton* do server. A maioria das mensagens enviadas pelo *stub* são compostas pelo conteúdo da mensagem juntamente com um código ou comando que é interpretado pelo *skeleton* e reencaminhado para o server.

Gui

A classe **Gui** seria responsável por armazenar a lógica da atualização da interface gráfica, mas essa parte do projeto não pode ser concluída.

Middleware e Comunicação Cliente-Servidor

As comunicações são principalmente iniciadas do lado do cliente através de *inputs* do teclado. Quando é preciso efetuar um pedido ao servidor, o cliente interage com o seu *stub* para enviar uma mensagem do tipo **request-reply** ao *skeleton* do servidor. As mensagens enviadas pelo *stub* são sempre acompanhadas por um comando em *string* que permite ao *skeleton* saber o que fazer com a mensagem. O *skeleton* pode responder logo ou pode ter de pedir informações ao

server antes de enviar a resposta. Existem também um canal do tipo **publisher-subscriber**, que é principalmente usado para que o server possa transmitir certas informações ao cliente, sem que este tenha que pedir essas informações. O padrão *publisher-subscriber* foi útil para que o servidor pudesse informar o cliente das atualizações feitas ao tabuleiro devido ao passar do tempo, algo que está apenas do lado do servidor.

Obstáculos e Dificuldades

Inicialmente, grupo teve alguma dificuldade em perceber como poderiam usar o ZMQ e aproveitar as suas vantagens. Contudo, após algumas sessões de esclarecimento com os professores, foi possível desenvolver essa parte do projeto sem problemas, e o projeto beneficiou imenso das funcionalidades do ZMQ.

Algo que ficou por resolver foram os problemas com a interface gráfiac do jogo. O grupo ficou "bloqueado" por algum tempo a tentar usar o Tkinter ou o Pygame para elaborar o GUI, mas foram encontradas várias dúvidas em relação ao funcionamento das *threads* no ambiente gráfico. No final, o grupo decidiu focar o seu trabalho nos conceitos mais fundamentais do projeto, como o desenvolvimento do protocolo de comunicação, funcionalidades multi-jogador e utilização de *threads* e *sockets*.

Conclusão

Em conclusão, o grupo ficou satisfeito com o projeto desenvolvido, embora tivesse sido muito bom implementar a interface gráfica. Mesmo assim, foi possível trabalhar com os conceitos principais lecionados ao longo da unidade curricular, como a utilização de *sockets* e *threads*, os diferentes padrões de comunicação e os métodos usados para desenvolver aplicações cliente-servidor.