Universidade Federal de Ouro Preto - UFOP
Nome: Ian Langkammer Batista, Leonardo Moreira Sepulveda
Disciplina: Redes de Computadores - CSI301
Matrícula: 20.1.8008 / 20.1.8006

Detalhes da Implementação

1) Storage.java

Classe responsável por criar e escrever os arquivos .txt referentes ao histórico do vencedor do jogo. Nada de muito complexo ocorreu nesse passo, a classe possui um *BufferedWriter* para manipulação do arquivo e um método chamado de *"storeMatch()"* responsável pela criação do arquivo com o nome igual a data no momento da gravação e o conteúdo passado como parâmetro.

2) ConnectionManager.java

Classe compartilhada por ambos os pacotes "*client*" e "*server*", essa classe foi criada a fim de abstrair os processos de comunicação entre cliente/servidor. Primordialmente, foram implementados os métodos "*read()"* que lê a mensagem do cliente/servidor e uma "*send()*" que envia mensagem pro cliente/servidor. É fulcral ressaltar que o seu construtor precisa receber um parâmetro "*connection"* do tipo *Socket*, pois sem ele não é possível construir os *streams*, e em seguida inicializa os atributos *Stream* da classe. Ao longo do desenvolvimento do projeto foram necessários implementar o método *"getIp()"* que retorna o IP do host da conexão e um método para retornar a instância da conexão passada no construtor.

Universidade Federal de Ouro Preto - UFOP
Nome: Ian Langkammer Batista, Leonardo Moreira Sepulveda
Disciplina: Redes de Computadores - CSI301
Matrícula: 20.1.8008 / 20.1.8006

3) ServerConnectionManager.java

Classe responsável pela comunicação do servidor com o cliente. Ela recebe uma conexão do tipo *Socket* no seu construtor e a instância do *Server* que ela está executando, além disso, para poder ler as mensagens do cliente e enviar as respostas ela herda a classe *ConnectionManager*. A classe também implementa a interface *Runnable*, permitindo ser executada em uma *thread* para que o *Server* consiga manipular várias conexões em várias *threads* simultaneamente. Ela também possui duas instâncias de *Game*, sendo uma da própria conexão e a outra da segunda conexão do servidor, intituladas de "*game*" e *"enemy"* respectivamente, essas instâncias são resolvidas na classe *Server*.

Em seu método "*run()"*, a execução fica presa em um laço de repetição que aguarda por alguma mensagem do seu cliente - que será denominado a partir de agora de *request*. Assim que o *request* é lido, uma *response* é gerada através da "*processRequest()"* que recebe o conteúdo do *request* do cliente no formato "user;action;param", esse método interpreta essa mensagem e chama a *"execRequest()"* para executar de fato o que foi solicitado pelo cliente e produzir uma *response*, que é enviada de volta para o cliente. Caso o *request* seja de "*disconnect"*, o laço de repetição é quebrado e essa *thread* é finalizada.

As *actions* interpretadas pelo servidor são auto-explicativas e as mais complexas serão explicadas a seguir:

- "*makeGuess"*: funciona no modo de jogo *singleplayer* e *multiplayer*, chama o método "*makeGuess()"* da instância de *Game* e verifica se acertou o número ou não, caso tenha acertado ele salva a partida chamando o método "*storeMatch()"* através da instância do *Server*. Se for modo *multiplayer*, ele também chama o método *"changeTurns()"* da própria classe, que seta o jogo para aguardar a resposta do seu adversário.
- "*finishGame"*: em caso de modo *singleplayer*, o método simplesmente reseta os dados da instância de *Game*. Caso seja *multiplayer*, ele reseta os dados da sua própria instância de *Game* e a do inimigo, para informá-lo que ele não deseja continuar, em seguida, troca os turnos.
- "*multiplayer"*: reseta o jogo e verifica se existe algum jogador para enfrentá-lo através do método *"multiplayer()"* da instância *Game*. Caso exista inimigo retorna "*ready*", caso contrário retorna "*wait"*. Se for "*ready*", troca os turnos.
- "*isMyTurn*": esse método foi criado para que o cliente verifique se é a sua vez de jogar ou se ele está aguardando a ação do adversário.
- "*didILost"*: esse método foi implementado para verificar antes de dar o próximo palpite se o meu adversário já acertou a palavra antes de mim, uma vez que ele tenha acertado, não é necessário continuar pois ele já venceu.

Universidade Federal de Ouro Preto - UFOP
Nome: Ian Langkammer Batista, Leonardo Moreira Sepulveda
Disciplina: Redes de Computadores - CSI301
Matrícula: 20.1.8008 / 20.1.8006

4) Game.java

Classe que representa o jogo Tiro em Mosca e suas funcionalidades. Os seus atributos são o nome de usuário, número a adivinhar, número total de rounds, histórico de palpites, *Boolean* que indica se a instância está disponível, modo de jogo, *Boolean* que identifica se é a vez dessa instância jogar, *Boolean* que identifica se a instância perdeu o jogo e o número de vitórias. Além disso, existem alguns métodos importantes como:

- *"generateRandomNumber()"* que gera um número aleatório sem caracteres repetidos de 100 a 999, utilizado no modo *singleplayer*.
- *"countShot()"* & *"countFly"* que contam os tiros e moscas que o jogador teve com o seu palpite, utilizado pelo método que retorna a *String* do histórico de palpites.
- "*makeGuess"* é um método um pouco mais complexo, no modo *singleplayer*, ele valida se o palpite é 0, ou seja, se o jogador desistiu, caso não seja, ele aumenta o número total de rounds em 1 e adiciona o palpite no histórico. Se o jogador acertar o palpite, o número de vitórias é acrescido em 1 e seta o status de *loser* da instância do inimigo como *true*. No fim da execução, retorna *true* se acertou ou palpite e *false* se não.

Universidade Federal de Ouro Preto - UFOP
Nome: Ian Langkammer Batista, Leonardo Moreira Sepulveda
Disciplina: Redes de Computadores - CSI301
Matrícula: 20.1.8008 / 20.1.8006

5) Server.java

Classe referente a execução do servidor que recebe as conexões de clientes. Possui como atributos: porta, número máximo de conexões, número de conexões, uma instância de *Storage*, *Boolean* referente ao status do servidor, uma instância de *ServerSocket*, uma instância de *Socket*, uma instância de *ServerConnectionManager* e duas instâncias de *Game*. No seu construtor inicializamos esses atributos e essa classe possui uma *"main"* que instância e roda a própria *Server*. Caso ocorra algum erro durante a execução ou CTRL + C seja informado no terminal, o servidor irá entrar em processo de *"shutdown()"* para sair do ar.

Primeiramente na execução dessa *Server*, é instanciado o atributo *ServerSocket* e o seu atributo *running* para *true*. Logo em seguida a execução entra em *loop* enquanto *running* for *true*, aguardando por conexões de clientes. No momento em que o cliente se conecta, a *ServerConectionManager* é instanciada, verifica se o número de conexões máximas foi atingido e retorna se o servidor está disponível ou não para novas conexões. Caso o servidor esteja disponível, o método *"resolveInstance()*" retorna qual instância de *Game* está disponível e seta ela na *ServerConnectionManager*, além disso, seta a outra instância de *Game* como sendo o adversário. Por fim a *thread* da conexão é inicializada e o número de conexões é acrescentado em 1.

Quando algum jogador se desconecta o número de conexões é decrescido em 1. Assim também, ao vencer um jogo, a instância *Storage* é utilizada para salvar o histórico dessa partida.

6) ClientConnectionManager.java

Classe que herda *ConnectionManager*, é responsável pela comunicação do cliente com o servidor, possui um atributo com o nome do usuário e um método *"sendRequest()",* que envia alguma requisição para o servidor e aguarda por uma resposta, ao obter a resposta retorna ela no fim da execução do método. Vale ressaltar que o seu construtor precisa receber uma conexão *Socket*.

7) Interface.java

Classe exclusiva para formatação das informações em tela e leitura de *inputs* que são essenciais durante execução como a opção de menu, o palpite, número que o adversário deve acertar, etc… Possui uma instância de *Scanner* para ler os dados do terminal, que é inicializada em seu construtor.

Universidade Federal de Ouro Preto - UFOP
Nome: Ian Langkammer Batista, Leonardo Moreira Sepulveda
Disciplina: Redes de Computadores - CSI301
Matrícula: 20.1.8008 / 20.1.8006

8) Client.java

Classe que se conecta com o servidor e consome as suas funcionalidades. Possui um atributo porta padrão e constante, uma instância de *Interface* para auxiliar nas funcionalidades, formatações e leituras, e uma instância de *ClientConnectionManager*, referente a conexão. Em seu método *"main"*, instancia a própria classe *Client* e inicializa a sua execução.

Em sua inicialização, é solicitado o IP do servidor, que por padrão vem "localhost", caso ele não encontre o IP uma *Exception* será gerada. Caso o servidor seja encontrado, se lê o nome do jogador e instancia a *ClientConnectionManager*, em seguida verifica se o servidor está disponível. Caso ele não esteja disponível exibe no terminal uma mensagem de servidor indisponível e finaliza a execução.

No entanto, caso o servidor esteja disponível, printa uma mensagem de bem vindo e printa o menu, em seguida fica aguardando até que algumas das opções sejam informadas.

- 1. Jogar sozinho: envia uma requisição informando que esse modo de jogo foi escolhido e recebe o número a ser acertado gerado pelo servidor. Se mantém printando o histórico de palpites, solicitando um novo palpite e mandando um *request* de validação do servidor acerca do palpite. Ao acertar o número ou informar 0, é printado uma mensagem de *feedback* que varia se o jogador acertou ou desistiu.
- 2. Dois jogadores: se mantém preso em um loop (*"awaitEnemy()"*) mandando *requests* para o servidor a fim de verificar se algum adversário foi encontrado, ao encontrar um adversário, solicita o número que o seu adversário deve acertar e aguarda por sua vez. Antes do palpite nesse modo de jogo, é verificado se o meu adversário acertou no último palpite. Caso não tenha acertado, printa o histórico de ambos os jogadores e solicita um novo palpite, em seguida manda uma *request* para o servidor validar o palpite. Se o palpite estiver correto ou o adversário tiver acertado o último palpite, um *feedback* é printado com relação ao jogo e aguarda a resposta dos jogadores acerca de continuar com a jogatina. O jogador vencedor começa informando se deseja continuar ou não, caso ele não desejar, avisa o servidor que ele saiu do *multiplayer* e volta para o menu. Caso ele deseje continuar, aguarda a resposta do outro jogador, se ele decidir continuar uma nova partida é iniciada com o jogador vencedor inicializando o palpite, caso contrário informa que o adversário não continuou no jogo e volta para o menu.
- 3. Contra o computador não foi implementado.
- 4. Sair printa uma mensagem de despedida, encerra a conexão e finaliza a execução.