Sistemas Distribuídos: Relatório do Trabalho Prático

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Grupo 27: Afonso Sousa (a74196), Filipe Marques (a57812) e Rui Rodrigues (a74572)

10 de Junho de 2017

Conteúdo

Sumário - O projeto numa página **4**

Classes Java **5**

BancoContasJogadores 5
ComparatorConta 5
Conta 5
GestorQueues 5
LinkCliente 6
ParserCliente 6
ParserServinte 6
Partida 6
Queue 7
Servidor 7
ThreadServinte 7

Armazenamento e estruturas de dados **8**

Crítica e observações **8**

Resultado final 8
Observações 8

Resumo

Para este trabalho foi pedido que desenvolvêssemos *"uma aplicação distribuída para match-making num jogo online por equipas, semelhante ao Overwatch"*.

Neste trabalho desenvolvemos um programa que dá resposta a todos os pedidos do enunciado. Daremos atenção especial às situações em que são usadas as estratégias lecionadas nas aulas práticas e de que forma apareceu a necessidade de as usar.
Apesar de não termos experiência com o jogo Overwatch, nós baseámos o nosso programa num outro jogo já nosso conhecido e do mesmo género *MOBA*, o LeagueOfLegends. Este também tem o mesmo sistema de procura de partidas, diferindo apenas na forma como se joga, um é primeira pessoa e o outro em terceira pessoa (que não é o objetivo do trabalho). Vamos também usar a terminologia deste jogo por não conhecermos a do Overwatch, porque, provavelmente, serão muito semelhantes.
Segundo permissão do professor vamos exceder o limite de páginas permitido pelo enunciado, excluindo a capa e o índice da contagem.

Este relatório apresentará assim um sumário muito resumido com toda a informação relevante sobre o trabalho feito.

Sumário - O projeto numa página

Começámos por criar as classes e estruturas que nos pareceram relevantes para guardar os dados dos jogadores (Conta e BancoContasJogadores). De seguida fomos criar duas classes para correr em threads, uma LinkCliente para os clientes ligarem ao Servidor, e outra ThreadServinte para atender apenas um cliente.
De seguida, e já com estas classes prototipadas, escrevemos uma primeira tentativa de implementar filas de espera (*queues*) para os jogadores, orientadas segundo o nível (*rank*) de cada jogador. Como implementação para os ranks dos jogadores optámos por colocar também um sistema de pontuação, em que o jogador acumula ou perde pontos conforme os resultados obtidos nos jogos (entre -20 e 20, não existem empates). Quando o jogador atinge um dos limites (superior: 100 pontos, inferior: -100 pontos) é alterado o rank.
O próximo passo foi colocar comunicação entre o lado do cliente e o servidor com um socket entre eles. Naturalmente definimos algumas mensagens a serem trocadas entre os dois lados. Com recurso a um ciclo while() que lê uma mensagem do socket e um switch() que interpreta o recebido, metemos o lado do cliente e do servidor a comunicar.
As primeiras implementações tratavam de coisas simples como leitura de valores (rank e pontos), pedidos de login, criação de contas, mas conforme coisas mais complicadas iam sendo colocadas, tornava-se cada vez mais difícil conseguir gerir a comunicação na classe ThreadServinte.
O principal problema era encontrar uma forma para lidar com mensagens inesperadas por parte do sistema: a ThreadServinte tinha como objetivo cimeiro receber pedidos vindos do cliente e enviar respostas ao mesmo, mas por vezes era necessário receber coisas de outras threads no servidor.
Enquanto estavam a ser implementadas as queues, a ThreadServinte ficava bloqueada num read() para saber quando é que o jogo iniciava e alertar o jogador. O problema é que isto tornava impossível ler mensagens do cliente entretanto. Como o enunciado não menciona nenhum requisito sobre tal (e como coisas como a consulta do rank e de pontos foram acrescentadas por nós sem ser pedido) pensámos em remover essas funcionalidades extra e deixar a ThreadServinte a bloquear na leitura quando o jogador quisesse jogar.
Mais tarde, ao implementar o mecanismo de escolha dos heróis(*champion select*) com a classe Partida, tornou-se evidente que a estratégia até agora tomada não seria boa. Desta vez, era imperativo lidar com as mensagens assíncronas vindas da Partida porque era preciso fazer coisas como atualizar os terminais dos jogadores quando um membro da equipa escolhia um herói (*champion*). Se a ThreadServinte estivesse à escuta de mensagens vindas do cliente, haveria apenas uma maneira de receber também as mensagens da Partida: colocar a Partida a escrever no *BufferedReader* que a ThreadServinte está a ler.
Como tal é impossível, bem como ler de dois ao mesmo tempo, aqui fomos obrigados a mudar de caminho e recorrer a mais uma thread só para mensagens assíncronas, ficando as síncronas ao cargo da ThreadServinte. Assim surgiu o ParserServidor.
Esta classe é iniciada assim que um login é confirmado, e termina juntamente com a ThreadServinte quando é pedido o logout.
Pela mesma situação de ser necessário ler de dois locais ao mesmo tempo (stdin e socket), o lado do cliente também ficou com mais uma classe análoga, a ParserCliente. Esta classe fica encarregue de imprimir mensagens recebidas para o terminal, enquanto que a LinkCliente passa apenas a passar comandos do stdin para o socket.

Classes Java

O código está dividido em duas componentes: o package Cliente, que guarda toda a lógica usada para interação com o jogador, e o package Matchmaking, onde fica tudo o resto. Isto inclui o Servidor, GestorQueues, Conta e ThreadServinte.
Vamos explicar qual o propósito de cada classe definida, e como interage com o resto do programa. tratando-se isto de um resumo do trabalho feito, estão no código diversos comentários em locais oportunos a explicar os propósitos de instruções, bem como a estratégia implementada.

BancoContasJogadores

O banco de contas vai ser a classe encarregue de guardar e, até certo ponto, gerir o acesso a estas. Esta classe possui um *"HashMap<String,Conta> contas"* onde são colocadas todas as contas dos jogadores, com o username a servir de chave.
Quando um cliente quiser fazer login, a sua ThreadServinte terá que invocar *"banco.login(arg1,arg2)"*, sendo esta a única forma da thread conseguir obter uma conta do banco e "iniciar serviço". Existe também outro método que retorna uma conta quando dado um username, usado para buscar uma conta arbitrária, e usado, por exemplo, quando um jogador é colocado numa Queue.
Esta classe também disponibiliza a criação de uma nova Conta.

ComparatorConta

Esta classe é um simples comparator, definido para ser usado na ordenação de jogadores segundo a sua habilidade (rank e pontos). É usada pela classe GestorQueues para equilibrar as equipas.

Conta

A classe guarda toda a informação importante relativa a um jogador como *rank, pontos, username, password*. As informações que podem ser acedidas por outros jogadores são protegidas com locks() (ex: rank). Guarda também alguma informação para controlo de status como *inQueue*, *idPartida* e *sessao*, e outras coisas auxiliares como locks e arrays partilhados.
Estes arrays partilhados serão substituídos com sets() quando a Conta do jogador estiver numa Partida. Aí cada equipa terá um array partilhado para escolher os champions de forma concorrente. Disponibiliza métodos para coisas como login, registar resultados de uma partida, preparar a conta para um jogo (quando está prestes a começar a escolha de champions), mais gets e sets necessários.

GestorQueues

Como forma de ter o Servidor sempre disponível para receber ligações, a GestorQueues é uma classe que vai correr numa thread nova, e com acesso a estruturas guardadas no Servidor (mensagens e banco de contas). Esta classe está encarregue de ler os usernames de quem quer jogar da *"ArrayBlockingQueue mensagens"*, e colocar as suas contas de jogador nas Queue correspondentes ao seu rank.
Jogadores com rank 0 serão registados apenas na queue de rank 0, enquanto todos os outros ranks de 1 a 9 serão registados na queue do rank indicado e na queue do rank inferior. Como exemplo, um jogador com rank 3 será colocado nas queues de rank 3 e 2.

Quando uma queue fica cheia (10 jogadores), é iniciado um jogo novo. Primeiro, os jogadores são retirados das outras filas de espera em que possam estar, sendo depois o *ArrayList<Conta> listaJogadores* colocado numa Partida nova.
Nenhum pedido para jogar é perdido por ter uma queue cheia, porque as queues são verificadas de cada vez que um jogador é inserido, e enquanto a thread está a criar a partida nenhum pedido é lido em simultâneo. Os pedidos que chegarem entretanto são colocados no fim da *ArrayBlockingQueue mensagens*, cuja capacidade é para 30 pedidos simultâneos em espera. A *ArrayBlockingQueue* é uma estrutura que implementa uma fila de espera semelhante ao BoundedBuffer lecionado nas aulas: tem uma capacidade máxima de pedidos simultâneos, segue uma política FIFO, e é bloqueante tanto quando uma leitura não encontra nada, como quando uma escrita não tem espaço na queue.
Uma possibilidade considerada para implementar as filas de espera foi uma coisa análoga à Barreira das aulas práticas, mas como aqui as ThreadServintes não acedem diretamente aos objetos, não fazia sentido colocar uma barreira de acesso a objetos. Em vez disso, quando chegam os 10 jogadores, a própria barreira(GestorQueues) vai tomar conta de lançar uma Partida.

LinkCliente

Esta classe tem como único objetivo ler do terminal os comandos do seu jogador, e encaminhá-los para a ThreadServinte via socket. O envio de comandos é feito argumento a argumento, tornando-se assim muito simples à ThreadServinte saber qual o comando invocado (por ser sequencial) e quantas mensagens deve ler para invocar o pedido do cliente.
Vai ser esta classe a funcionar como main() dos clientes, a abrir o socket e a lançar um ParserCliente para lidar com as respostas. O buffer de chegada (*InputStream*) nunca será lido por esta classe. Cada jogador terá o seu próprio LinkCliente.

ParserCliente

Serve como interpretador para todas as mensagens enviadas pelo servidor. Esta classe terá uma thread própria e estará sempre a ler do socket. A leitura é bloqueante, e quando chega uma mensagem, é interpretada para saber o que deve ser impresso no terminal, por vezes lendo mais argumentos para construir a mensagem a ser impressa (ex: imprimir os champions escolhidos pelos jogadores).

ParserServinte

Recebe, processa e encaminha mensagens assíncronas de threads no servidor. Quando uma nova mensagem chega à conta associada em *ArrayBlockingQueue toCliente*, a ParserServinte dá seguimento à resposta necessária.
Exemplos de mensagens assíncronas: início do jogo(*start*), atualizar as escolhas(*updateEquipa*), início da fase de escolhas (*escolherAgora*).

Partida

Esta classe contém toda a lógica necessária à simulação de um jogo. É executada numa thread nova, sendo lançada uma Partida em cada vez que o GestorQueues enche uma Queue.

Para além de anotar nas Contas dos jogadores que estão numa partida, esta vai colocar o alarme de 30 segundos, e enquanto não tocar vai estar em ciclo infinito a fazer broadcast de mensagens. O primeiro broadcast segue para todas as 10 ParserServintes, avisando que o jogo começou. Daí em diante, cada vez que um jogador escolhe um champion, é feito um broadcast para a sua equipa alertando para atualizar as escolhas (atualiza o terminal do jogador com as escolhas novas).
Ao toque do alarme, a Partida irá ver o resultado das escolhas e analisar se todos escolheram um champion. Quem não o tenha feito será penalizado com uma derrota, e caso todos tenham escolhido é feito um broadcast com as escolhas gerais (usernames de toda a gente e suas escolhas), e é gerado um resultado aleatório do jogo. No caso em que a partida é cancelada por alguém não ter escolhido (timeout), todos os jogadores terão que pedir para jogar novamente antes de serem postos em queue.

Queue

Implementa uma fila de espera na GestorQueues. Cada Queue terá um rank associado, e disponibiliza os métodos necessários para inserção, remoção, etc. Guarda um *ArrayList<Conta> listaJogadores*. A gestão dos ranks é feita no GestorQueues.

Servidor

Classe que armazena os objetos principais do programa e que recebe conexões dos clientes. No arranque, é aberto um socket, um BancoContasJogadores e um GestorQueues (este numa thread nova) que lê mensagens do *ArrayBlockingQueue mensagens*, também variável de instância do Servidor que no máximo aguenta com 30 pedidos em espera antes de bloquear escritas.
Estado o servidor inicializado, o Servidor vai ficar em ciclo à espera de ligações. Quando um LinkCliente se liga à socket, o Servidor lança uma ThreadServinte numa nova thread.

ThreadServinte

Responsável por receber todas as mensagens escritas no socket oriundas do jogador, esta classe vai interpretar o pedido e ler os restantes argumentos necessários para o comando. De seguida invoca o pedido do jogador.
Esta classe executa numa thread lançada pelo Servidor, e só disponibiliza todas as funcionalidades depois do jogador ter feito login com sucesso(o método login() retorna a Conta desse username), ou criado uma Conta nova. Quando o jogador entra numa Conta, lança uma ParserServinte numa nova thread e começa a responder a coisas como logout, consultar o rank ou os pontos de um jogador, procurar um jogo (entrar nas Queues), escolher um champion, etc. Termina a execução quando o jogador fizer logout.
De notar é a forma como a escolha dos champions é feita: quando um conjunto de 10 jogadores são colocados numa Partida, é colocado um ArrayList partilhado onde as ThreadServintes vão aceder concorrentemente: quem chegar primeiro fica com o lock e consegue pedir uma escolha. Essa escolha será depois propagada com uma atualização nos terminais dos jogadores da sua equipa.

Armazenamento e estruturas de dados

Como no enunciado não é pedida persistência, todos os dados estão em memória. No centro de todo o sistema está a classe Servidor, onde são colocadas as estruturas de dados partilhadas entre threads. O banco de contas, o gestor das queues, a queue para mensagens e o socket estão lá guardados.
Entre todas as classes, importa referir que o lado do cliente não tem acesso direto a nada que esteja no servidor, nem são mantidas cópias nessas classes. Mesmo quando é para atualizar as escolhas de champions, nada que vem do servidor fica guardado: ou é impresso diretamente, ou gera uma impressão com uma mensagem pré-definida (ex: *"Perdeu a partida!"*).
É também de salientar que quando um cliente pede para consultar dados de outro jogador, a ThreadServinte terá que fazer um pedido ao BancoContasJogadores, recebendo apenas a informação pedida e não a conta toda.

Crítica e observações

Resultado final

O resultado final parece-nos muito bom, cumprindo com todos os objetivos enunciados e disponibilizando funcionalidades extra. Para além de simular o matchmaking, permite que uma pessoa participe com um terminal, podendo escrever pedidos, e ver as respostas aos mesmos. Ainda assim, há espaço para melhorar, e mais funcionalidades que se podiam implementar, mesmo que fora do âmbito da disciplina.
No início, os professores foram-nos sugerindo implementações com múltiplas threads a gerir a comunicação, mas nós não achámos necessário tal coisa. Conforme o programa ia ficando mais complexo, chegamos à conclusão de que essa seria mesmo a forma certa de fazer as coisas. Agora temos 4 threads dedicadas a ler/escrever no socket, tal como sugerido pelos professores.

Observações

Devido à restrição do número de páginas permitidas, não nos foi possível colocar imagens pertinentes do código, mas demos o nosso melhor para expor tudo o que nos pareceu importante.

Um agradecimento aos professores pela prontidão em responder às nossas dúvidas e por permitirem exceder o limite superior de páginas permitidas neste relatório.