Juan Lessa

nº mec: 91359

Universidade de Aveiro

Distributed Black Jack

Introdução

Esse projeto da cadeira de computação distribuída tem como objetido desenvolver um jogo p2p de black jack 100% distribuído a cumprir os seguintes requisitos:

- player.py consegue jogar um jogo solitário.
- 2 player.py conseguem jogar um jogo.
- 3 player.py conseguem jogar um jogo
- "bad_player" implementado com as opções:
  - Tirar uma carta a mais.
  - Mentir sobre ter ganho e/ou o valor das suas cartas.
  - Mentir sobre o valor de hash obtido do deck
- Restantes jogadores são capazes de detectar que houve trapaça no jogo.

Protocolo p2p:

Toda a comunicação entre jogadores foi feita implementando o protocolo pear to pear seguindo o padrão de mensagens abaixo:

- **"H":** Abreviação de Hit, significa que o jogador pediu uma carta ao deck.
- **"S":** Abreviação de Stand, significa que o jogador deu a vez.
- **"W":** Abreviação de Win, significa que o jogador declarou ter vencido a partida.
- **"D":** Abreviação de Defeat, significa que o jogador declarou ter excedido 21 pontos.
- **"U":** Abreviação de Unavailable, significa que não foi possível estabelecer conexão com o deck.
- **"O":** Abreviação de Ok, significa que a operação ocorreu corretamente, ou que não há diferença entre o hash fornecido pelo deck e o hashcalculado pelo jogador.
- **"C":** Abreviação de Cheating, significa que há diferença entre o hash fornecido pelo deck e o hash calculado pelo jogador, ou seja, houve trapaça.

O contato entre jogadores utiliza sockets TCP, para cada conexão com outro jogador é criado um socket para enviar e/ou receber uma mensagem, em seguida é feito socket.close() para evitar que existam conexões abertas durante toda a execução do programa e assim prevenir possíveis erros decorrentes de sockets que não foram fechados.

Nomes importantes

Para melhor compreensão do funcionamento do código deixo aqui o nome de algumas variáveis e listas pertinentes bem como sua respectiva utilidade.

- need_connect: lista com as portas dos jogadores cuja conexão ainda não foi testada.
- players_connection: lista com o nome e a porta de conexão para cada player.
- current_score: variável inteira que armazena a própria pontuação.
- players_move: lista que armazena todos os movimentos da partida e o jogador que o efetuou.
- playing: lista que armazena todos os jogadores que ainda não perderam.

Início:

O jogo inicia com os jogadores informando sua porta e as portas dos demais jogadores via argumento no terminal (-s ou –self para informar a própria porta e -p ou --players para informar as portas dos outros jogadores), caso não seja passado argumento -p o jogo inicia em modo single player, que terá seu funcionamento desenvolvido futuramente neste relatório.

Assumindo que foram passadas portas no argumento -p, o jogo inicia por perguntar ao player qual o nome para se referir a ele durante a partida, o nome passado é armazenado na lista player_connection junto com a porta passada no argumento -s. Em seguida inicia um teste de conexão com os demais jogadores. Primeiro o jogador tenta estabelecer conexão com cada um dos demais jogadores, caso bem sucedida o jogador envia aos demais seu nome e sua porta para conexões futuras, em seguida recebe como resposta o nome do jogador que foi conectado. Caso a conexão não seja bem sucedida o programa avança e assume que este jogador tentará estabelecer contato posteriormente.

Ao fim de tentar se conectar com todos os jogadores, caso hajam conexões mal sucedidas o programa entra em modo de espera, aguardando que os players que faltam se conectem a ele. Para isso é usado um selector em modo EVENT_READ e uma socket ligado (bind) ao (‘localhost’, self_port), quando todos os jogadores tiverem estabelecido conexão com os demais o jogo se inicia.

Decorrer do jogo

Para não tornar esta explicação demasiado confusa todas as funções criadas serão explicada posteriormente na seção métodos.

O decorrer do jogo ocorre dentro de um ciclo while True, visto que, não sabemos quantas rodadas terá o jogo, cada rodada acontece de forma ordenada, da menor para a maior porta, para isso faz – se uso da variável current_player que varia entre 0 e o número de jogadores -1 e da lista player_connection ordenada de forma crescente pelas portas de cada jogador. Caso o jogador identifique que está na sua vez de jogar (player_connection[current_player][1] == self_port) é feito uma verificação de initial_round, uma variável booleana que armazena True se for o round inicial e False caso contrário. Se initial_round == True são pedidas duas cartas ao deck através do método get_card(), em seguida é chamada a função interact_with_user1, que retorna um dos quatro valores “H”, “S”, “W”, “D”, esse valor será adicionado a lista players_moves para posteriormente sabermos a ordem das cartas jogadas.

Caso interact_with_user1 retorne:

- “H” (Hit), é pedida uma carta ao deck essa carta é armazenada na variável last_card, e só será adicionada a lista current_cards na próxima rodada. Em seguida é chamada a função inform_players() para enviar aos demais jogadores a mensagem “H”.

- “S” (Stand), é feito last_card = “” para, na próxima rodada, nenhuma carta ser adicionada a current_cards, em seguida é chamada a função inform_players() para enviar aos demais jogadores a mensagem “S”.

- “W” (Win), é chamada a função inform_players() para enviar aos demais jogadores a mensagem “W”, em seguida é feito break do ciclo while True.

- “D” (Defeat), , é chamada a função inform_players() para enviar aos demais jogadores a mensagem “D”, em seguida o jogador é removido da lista playing.

Caso o programa identifique que não é a sua vez de jogar, ele fica aguardando que o jogador da vez informe sua jogada, que será uma dentre as seguintes, e também será armazenada na lista players_moves.

- “H” significa que o jogador pediu uma carta ao deck.

- “S” significa que o jogador passou a vez.

- “W” significa que o jogador diz ter vencido a partida, nesse caso é feito break do ciclo while.

- “D” significa que o jogador diz ter perdido (excedeu 21 pontos), nesse caso o jogador é removido da lista playing.

- “U” significa que o jogador não conseguiu estabelecer conexão com o deck para pedir uma carta, nesse caso o jogo não pode proceder e todos os jogadores fazem close de seus selectors e sockets e dão a partida por encerrada,

Esse processo segue em loop até que um jogador declare ter vencido a partida ou reste apenas um jogador (len(playing) == 1), em seguida começa o processo de armazenar as cartas no redis.

Colocar as cartas no Redis

O processo de armazenar as cartas no redis também é feito de forma crescente conforme as portas dos jogadores. Caso o programa identifique que está na sua vez ele inicia um cliente redis e armazena todas as suas cartas numa lista com a chave igual a self_port. Caso contrário o programa aguarda receber um Ok do jogador da vez.

Obter as cartas dos demais jogadores

Nessa etapa o programa busca no redis a lista de cartas de cada jogador, e armazena a porta do jogador, a sua pontuação e a lista das suas cartas na lista players_score. No final é encerrada a conexão com o redis e passamos para a próxima etapa.

Determinar vencedor

Cada jogador é capaz de verificar quem ganhou o jogo, para tal consultamos a pontuação de cada jogador, que está armazenada na lista players_score, caso o jogador possua 21 pontos ele é

automaticamente o vencedor, caso o jogador exceda 21 pontos ele automaticamente perde, caso jogador possua menos de 21 pontos e mais pontos que o jogador anterior ele é eleito ganhador. Ao fim de, no pior caso, verificar todos os jogadores temos o vencedor.

Hash md5

Os dois jogadores de maiores portas ficam encarregados de pedir e verificar o hash das cartas junto ao deck, bem como informar aos jogadores se houve ou não trapaça. Para pedir o hash é usado o método get_hash(), e para verificá – lo precisamos da lista de todas as cartas distribuídas pelo deck ordenadas cronologicamente, para isso usamos a lista players_moves que contém todas as jogadas e o respectivo jogador que a efetuou.

Após obtermos a lista ordenada de todas as cartas distribuídas no jogo (played_cards) calculamos o seu hash, comparamos com o hash fornecido pelo deck e enviamos "O" para os demais jogadores caso os hash coincidam e "C" caso os hash sejam diferentes.

Caso o jogador não tenha sido escolhido para calcular o hash das cartas ele fica a espera de receber uma mensagem de cada um dos jogadores que foram escolhidos para essa tarefa.

Validação da partida

Se o jogador tenha sido escolhido, na etapa anterior, para verificar o hash junto ao deck, a validação da partida é feita comparando o hash calculado com o hash fornecido pelo deck. Caso contrário a avaliação é feita comparando as duas mensagens dos jogadores que verificaram o hash, se ambas são "O" de Ok, significa que não houve trapaça.

Após informar ao jogador se houve trapaça ou não é feito close dos sockets e/ou selectors que foram usados e o jogo encerra.

Bad player

O codigo do bad_player.py funciona da mesma forma que o player comum, explicado até o momento, porém com algumas opções a mais.

Toda vez que o bad_player tira uma carta junto ao deck lhe é oferecida a oportunidade de pedir outra carta e eliminar essa.

Caso o bad_player seja escolhido para verificar o hash das cartas com o deck e tenha havido trapaça no jogo, isso é, o hash calculado é diferente do hash fornecido pelo deck, é perguntado se o bad_player deseja mentir sobre o hash, se optar por fazer isso é enviado para os demais players uma mensagem "O" para dizer que o hash está correto.

Single player

O modo player solitário é chamado caso na execução do jogo não tenha sido passado o argumento -p, esse modo inicia uma partida sem tentar se conectar a nenhuma jogador, a partida é iniciada, são pedidas duas cartas ao deck, o jogador pode pedir uma nova carta, passar a vez, dizer que venceu ou perdeu, neste modo não são feitas verificações de vencedor, se o player escolhe a opção (W)in é assumido que ele venceu bem como se ele escolhe (D)efeat é assumido que ele perdeu. após o player vencer ou perder é feita a verificação se houve ou não trapaça, o player pede o hash ao deck e verifica se coincide com o hash das suas cartas. Atento para o fato que em modo single player o bad_player atua como um player normal, isso é, não são lhe oferecidas as opções de pedir uma carta a mais e mentir sobre o hash.

Métodos

- redis_connec()t: conecta com servidor redis no localhost retorna essa conexão.
- acept(): função chamada pelo selector na fase de testar a conexão com os players. aceita conexão, recebe o nome do player que se conectou e envia o próprio nome.
- get_card(): função que estabelece conexão com o deck e pede uma carta, esse método retorna a carta obtida ou “U” caso não tenha conseguido se conectar com o deck.
- get_hash() função que pede ao deck o hash das cartas distribuídas até o momento.
- inform_players(): função que recebe como argumentos uma mensagem e uma lista de nomes e portas (players_connection), estabelece conexão com cada porta e envia a mensagem.
- receive_move(): função chamada pelo selector durante o jogo para receber uma mensagem de outro player.

Conclusão

Desenvolver um jogo de cartas completamente distribuído foi uma tarefa divertida, um pouco trabalhosa e se mostrou muito útil para pôr em prova os conhecimentos adquiridos na cadeira, lidar com as conexões entre os players tolerando possíveis falhas e desenvolver um protocolo p2p que cumprisse todas as necessidades de comunicação entre os players foi um processo trabalhoso, entretanto julgo ter desenvolvido a tarefa de forma suficientemente satisfatória. em meus testes o player.py e o bad_player.py se comportaram bem em modo single player e em modo multiplayer (fiz testes com até 5 jogadores), o jogo consegue tratar bem possíveis falhas de conexões com o deck, bem como informar tal falha aos demais jogadores.