Implementando um jogo Bridge Defense
Trabalho Prático 2 - Redes de Computadores

Bruno Esteves Campoi {2020054250}[1]
Vinicius Silva Gomes {2021421869}[1]
Mirna Mendonça e Silva {2021421940}[1]

[1] Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG) – Belo Horizonte, MG – Brasil

{bec, vinicius.gomes, mirnamendonca}@dcc.ufmg.br

1. Introdução

Nesse TP foi implementado um Bridge Defense, uma simplificação dos jogos de Tower Defense. O jogo funciona de modo que existem 4 rios e 8 pontes, alguns canhões posicionados ao longo das pontes e o objetivo principal é afundar a maior quantidade de navios que estejam passando por cada rio, utilizando os canhões posicionados.

A parte interessante é que o jogo é executado automaticamente de maneira remota, isto é, diversas requisições são realizadas para o servidor para se obter a posição dos canhões, receber a posição dos navios a cada turno, atirar num navio, etc. Mais especificamente, cada servidor é responsável por um rio e, portanto, é preciso manter e sincronizar a comunicação com o servidor de cada rio para que os navios sejam atualizados corretamente, as requisições de tiro aconteçam nos servidores corretos, etc.

Sendo assim, a ideia é que seja desenvolvido um programa que executa uma partida completa quando iniciado e, ao final, retorne o *score* obtido naquele jogo. A política de tiros dos canhões é um dos detalhes principais da implementação, que permite que o jogo seja jogado sem interação do usuário. Esse e outros detalhes pertinentes do funcionamento do jogo serão discutidos ao longo das próximas seções.

A seção 2 discute como o problema foi modelado computacionalmente; na seção 3 os detalhes da implementação realizada são discutidos; na seção 4 os passos para executar o código são detalhados; e a seção 5 encerra o relatório.

2. Modelagem computacional

Para representar o jogo computacionalmente, a classe `BridgeDefense` foi definida com os principais atributos e métodos necessários para que ele ocorra. Dentre os atributos, destacam-se: `_currentTurn`, um inteiro que marca o turno atual do jogo; `_cannons`, um vetor que armazena as posições de cada canhão no jogo; e `_ships`, uma matriz que armazena os navios sobre cada ponte, em cada rio. Alguns atributos foram omitidos por não serem tão importantes para o funcionamento do jogo em si.

Os canhões foram armazenados em um vetor para que o algoritmo de tiro fosse facilitado. Como queremos destruir a maior quantidade de navios, queremos que os canhões sempre atirem o máximo possível. Portanto, parte da rotina do algoritmo de tiro é passar por cada canhão presente no jogo e verificar se ele pode atirar em alguém.

Sobre a matriz dos navios, essa estrutura possui em cada célula uma lista de dicionários, na qual cada dicionário representa um navio sob aquela ponte, naquele rio

específico. Ela foi definida dessa forma pois possibilitou a atualização rápida e prática do estado a cada turno, de modo que pouca computação a mais fosse necessária para armazenar o estado recebido na resposta. Além disso, esse formato permite o acesso direto aos navios quando se sabe o rio e a ponte que deseja acessar.

Quanto aos métodos, de modo geral, cada tarefa específica realizada pelo jogo é dividida em uma função separada. Os métodos são:

- **`_get_ip_address()`**: método auxiliar que recebe o *host* e a porta do servidor e retorna o endereço IP (v4 ou v6 dependendo do *host* informado, preferenciando o v6 para endereços) e a família de endereços para a criação do *socket*;
- **`_serverCommunication()`**: método auxiliar usado por todos os outros métodos para se comunicar com o servidor. Ele já abstrai toda a lógica de retransmissão e lida com a resposta de *gameover*;
- **`_authenticationRequest()`**: realiza a autenticação do programa nos 4 servidores para que o jogo possa começar;
- **`_cannonPlacementRequest()`**: requisita para o primeiro servidor a posição de cada um dos canhões no jogo e popula o atributo dos canhões;
- **`_turnStateRequest()`**: requisita o estado dos navios para o servidor de cada rio e popula a matriz dos navios com os dados obtidos;
- **`_shotMessage()`**: itera pelos canhões e atira com todos que podem atirar em algum navio, seguindo a política de tiros definida na subseção 3.4;
- **`_gameTerminationRequest()`**: manda a requisição de finalização de jogo para o primeiro servidor;
- **`playGame()`**: método público chamado pela `main` para começar o jogo.

Além dessas, `_authenticationRequest()` e `_turnStateRequest()` definem funções auxiliares internamente para que a paralelização seja possível. A subseção 3.3 define melhor como essa estratégia é implementada.

3. Detalhes da implementação

3.1. Como os turnos são reconstruídos

A cada turno o jogo manda requisições de turno paralelas para os servidores. Cada servidor, ou rio, responde com uma lista de navios que estão em uma ponte específica. O servidor responde com 8 respostas seguidas, onde cada resposta é respectiva a uma ponte.

O programa adiciona esses navios à matriz `_ships`, indexados nas devidas posições [rio][ponte]. Se a atualização for bem sucedida, ele incrementa o turno em uma unidade e retorna.

Esse estado atualizado será recebido pelo método de tiros, que irá analisar o estado atual, a vida corrente dos navios e realizar a melhor escolha de navios possível para alvejar. Quando o algoritmo de tiros realiza um tiro com sucesso, ele atualiza o atributo *hits* do navio em questão, para manter o estado atualizado e a consistência com relação aos próximos canhões que podem vir a atirar nesse mesmo navio.

3.2. O algoritmo de retransmissão

A retransmissão do algoritmo é feita quando acontece o *timeout* da requisição. Após uma série de testes em diversas dificuldades e observações de como o algoritmo reagia a cada uma delas, chegou-se no tempo de 0.4 segundos até o *timeout*.

Para assegurar que o jogo continue, o método que realiza as requisições mantém um *loop* enquanto uma resposta válida não é obtida. Portanto, o jogo não tem um limite de tentativas de requisições, ele sempre retransmite os pacotes a cada 0.4 segundos, caso uma resposta não seja obtida. Esse fluxo continua até que o jogo seja finalizado.

3.3. Comunicação em paralelo com os servidores

As requisições de novos turnos acontecem para cada rio, ou seja, cada servidor. Além disso, o servidor retorna 8 respostas consecutivas com os navios presentes em cada ponte, naquele rio. Sendo assim, é notável que, para os servidores de dificuldade mais alta, um código sequencial sofreria muito para lidar com a perda de pacotes de todas essas requisições e respostas. Portanto, para mitigar esse problema, as requisições foram paralelizadas.

Na implementação realizada pelo grupo, a requisição dos navios de cada servidor será separada e executada por uma *thread* diferente. Como a matriz de estados pode ser facilmente indexada por [rio][ponte], não há condição de corrida e cada uma das *threads* pode manipular livremente as posições que precisam alterar.

Assim, tanto a requisição de novos turnos quanto a autenticação antes do jogo começar, que também deve acontecer em cada servidor, são executadas de forma paralela, para evitar os problemas ocasionados pela perda de pacote. Ambos os métodos definem uma função interna que será a função executada pela *thread* e o método exterior é responsável por criar, executar e aguardar o retorno de cada uma das *threads*.

Por causa do *design* do método de transmissão, quando o jogo é finalizado e uma requisição de *gameover* é recebida, cada *thread* recebe essa requisição e também exibe o *score*. Então, mais uma requisição será feita até que todas as *threads* percebam que o jogo já terminou (recebe uma falha que o jogo não existe) e o programa é finalizado.

3.4. O algoritmo dos tiros

Uma vez por rodada, o algoritmo que cuida de mandar as mensagens de tiro é invocado. Seu funcionamento é: de acordo com a posição de cada canhão no mapa, os navio alcançáveis por ele são selecionados. Dentre esses navios, é identificado aquele que está mais próximo de ser afundado, ou seja, com menos pontos de vida. Esse navio será o alvo do tiro. Uma requisição de tiro é feita para cada canhão com navios alcançáveis, a cada nova rodada. O pseudocódigo é demonstrado no algoritmo 1.

Localmente, a quantidade de tiros tomada por um navio só é atualizada caso a requisição de tiro tenha obtido sucesso. Dessa forma, outros canhões poderão tomar a melhor decisão de tiro dentro da mesma rodada e em rodadas subsequentes.

Portanto, a política do algoritmo segue a ideia de um algoritmo guloso, tomando a melhor escolha local possível, ou seja, alvejar o navio com o menor número de pontos de vida no momento, esperando que isso resulte na maior quantidade de navios destruídos. Essa política é relativamente simples e intuitiva, mas demonstrou bons resultados nos testes realizados pelo grupo.

4. Como executar o código

O trabalho foi desenvolvido utilizando a linguagem de programação Python e alguns de seus módulos *built-in* (*socket*, *json*, *threading*, etc). Para tanto, é recomendado que a

Algoritmo 1

Entrada: $cannon_1 \dots cannon_n$ ▷ Lista de canhões
Entrada: map ▷ mapa da posição dos navios
  $weakestShip \leftarrow NULL$
  for $cannon_1$ to $cannon_n$ **do**
    $x \leftarrow cannon_n[1]$
    $y \leftarrow cannon_n[0]$
    for $ship$ in ($map[x][y]$ and $map[x+1][y]$) **do**
      $ship_{life} \leftarrow ship_{life} - ship_{hits}$
      $weakestShip \leftarrow \min(ship_{life})$ ▷ Guarda o navio com menor vida
    end for
    $shotRequest(weakestShip)$ ▷ Atira no navio com menor vida
  end for

instalação da linguagem seja feita através da documentação oficial[1].

Além disso, a versão da linguagem que foi utilizada durante o desenvolvimento do trabalho foi a 3.10.12. Portanto, é recomendado que a versão utilizada ao longo dos testes seja igual ou superior a essa. Uma vez que o ambiente esteja configurado, o comando

```
python3 client.py <host> <port> <GAS>
```

deve ser utilizado para executar as funcionalidades desenvolvidas e realizar a comunicação entre o cliente e o servidor do jogo.

Para a avaliação, o GAS utilizado foi:

```
# GAS
2021421869 :44:87407
   f792f59b7dde2bf51a0ae7216cf8c246a7169b52ac336bbf166938d91a1
   +2020054250 :44:50527
   ec32fc4c6fd5493533c67ce42f5fcad7bb59723976ff54acc6ae84385b8
   +2021421940 :44:
   a70a80b0528f580bb6c0a94ae37e3d8efdfb7adb9f939f3af675e9ea69694db4+
   f16d50fda86436470ba832a3f63525650dbd1fe021e867069f35ef4073d1b637
```

5. Conclusão

A experiência adquirida durante este trabalho foi grande, proporcionando a oportunidade de enfrentar alguns desafios existentes na comunicação em rede, como perda de pacotes, bem como otimização da comunicação por meio da paralelização e gerenciamento eficiente de múltiplos servidores simultaneamente.

A implementação do Bridge Defense possibilitou a aplicação prática dos conceitos aprendidos em sala e também destacou a importância da política de tiros dos canhões na automação do jogo, permitindo uma experiência sem a necessidade de interação do usuário. Este projeto foi fundamental para o aprimoramento de nossas habilidades e conhecimentos em redes, preparando-nos para desafios futuros.

[1]Disponível em: https://www.python.org/downloads/.