CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS
Sistemas Distribuídos (2016/2) - DECOM

Implementação de um Jogo Multijogador online (MMOG) usando o Phaser: Exclusão Mútua

Felipe Duarte, Juan Lopes, Luís Nascimento
Professora: Anolan Milanés

INTRODUÇÃO

No segundo trabalho prático da disciplina de Sistemas Distribuídos, ministrada pela professora Anolan Milanés, foi proposto o desenvolvimento de um jogo multiplayer que utilize recursos compartilhados para realizar o tratamento de condições de corrida.

Assim sendo, optou-se por construir uma aplicação cliente-servidor, utilizando as seguintes ferramentas: Node-js, como plataforma de desenvolvimento presente no módulo servidor; Phaser, como principal framework responsável por renderizar e construir a dinâmica do jogo.

Todo o código-fonte do trabalho pode ser encontrado no Github (https://github.com/lcnascimento/agar-mmo).

DESCRIÇÃO DO JOGO

O jogo construído é relativamente simples do ponto de vista de dificuldade e dinâmica tendo em vista o escopo do trabalho, cujo objetivo é a melhor compreensão do problema da exclusão mútua.

Assim, nesse jogo, cada jogador será apresentado como uma esfera no mundo criado. Além das esferas, são renderizadas moedas aleatoriamente no mapa do jogo a cada 5 segundos. Todo jogador inicia o jogo com uma pontuação nula e, à cada colisão com uma moeda, sua pontuação será acrescida em uma unidade. Foram estabelecidos limites para a movimentação de cada jogador, que correspondem às fronteiras do mapa, porém, não há restrições para jogadores ocupando a mesma posição.

A princípio, o jogo não estabelece um fim bem definido, de modo que a pontuação de cada jogador pode acumular infinitamente. Espera-se continuar desenvolvendo a dinâmica do jogo em um futuro breve, estabelecendo um limite para o vencedor, criando uma funcionalidade de lobby, entre outras.

O nome do jogo, Agar, vem da tentativa de reprodução das funcionalidades disponíveis no jogo agar.io.

PROCESSO DE INSTALAÇÃO

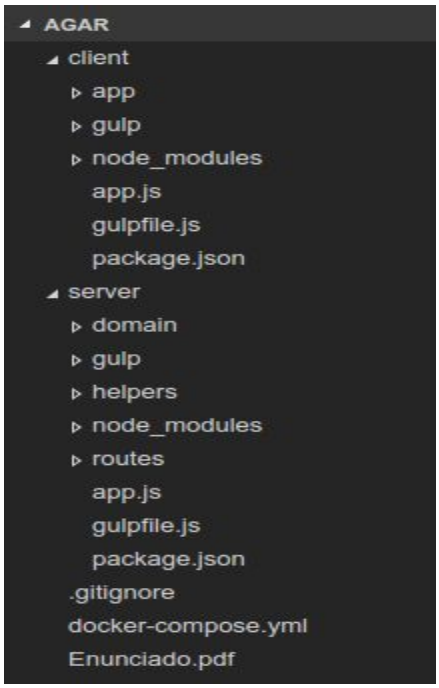

Figura 01 - Estrutura de pastas da aplicação

Conforme pode-se perceber pela Figura 01, o sistema foi dividido em 2 aplicações menores: cliente e servidor. São pré-requisitos para execução da aplicação:

1. Nodejs
2. NPM
3. Docker (não obrigatório)

O processo de instalação e execução do aplicativo em ambiente Linux será descrito à seguir:

1. Instalação dos pacotes de dependência
   1.1. Acessar a pasta client e executar o comando “npm install”
   1.2. Acessar a pasta client/app e executar o comando “npm install”
   1.3. Acessar a pasta server e executar o comando “npm install”
2. Caso exista docker e docker-compose devidamente instalados no sistema
   2.1. Acessar a pasta raiz da aplicação e fazer “docker-compose up -d”
3. Caso contrário
   3.1. Acessar a pasta server e executar o comando “node app.js”
   3.2. Acessar a pasta client e executar o comando “sudo node app.js” (necessário sudo pois a aplicação cliente utiliza a porta 80 do sistema)

Observação: Dependendo da distribuição utilizada, o pacote node pode ser instalado como ‘nodejs’. Neste caso, basta substituir ‘node’ por ‘nodejs’ no passo-à-passo anterior.

Com a aplicação rodando, pode-se acessar a aplicação localmente utilizando o endereço "http://localhost" em um browser, ou acessar em uma máquina com o cliente e servidor executando utilizando o IP da respectiva máquina.

PROJETO DO SISTEMA

Conforme citado anteriormente, construiu-se um sistema cliente-servidor. Para isso, cada usuário será tratado como um cliente que se conecta à um servidor único da aplicação. A comunicação entre esses processos acontece por meio de troca de mensagens.

Para esse fim, optou-se por utilizar um módulo da plataforma node-js chamado de 'socket.io'. Com ele, ao se conectar no servidor, o cliente terá consigo um objeto 'socket', que lhe dá opção de enviar e receber mensagens do servidor. O servidor, por sua vez, ao receber mensagens, pode encaminhar novas mensagens para o requerente, para os demais processos conectados ao servidor (com exceção do requerente), ou para todos os processos conectados ao servidor no momento (incluindo o requerente).

À seguir são descritas as mensagens que são trocadas entre processos cliente/servidor:

1. ***connection*:** Mensagem enviada sempre que um processo cliente se conecta ao servidor. Aqui o servidor registra uma nova instância de jogador em sua lista local e notifica os demais processos de que existe um novo jogador no jogo através da mensagem ***player_registered***.
2. ***disconnect*:** Enviada sempre que um processo cliente se desconecta do servidor. Com ela, o servidor se vê responsável por notificar os demais processos de que um jogador foi desconectado. Estes, por sua vez, ao receber tal notificação, destroem a esfera correspondente ao jogador em tela.
3. ***initial_state*:** mensagem enviada ao cliente que está se conectando ao servidor. Nela são enviadas a lista de jogadores e moedas presentes no momento.
4. ***player_moved*:** Mensagem enviada ao servidor sempre que o usuário realizar alguma ação de mover seu avatar em tela através de cursores. Ao receber tal mensagem, o servidor notifica os demais processos que determinado jogador se movimentou. Desse modo os clientes podem atualizar a posição do mesmo em tela.
5. ***coin_eaten*:** Mensagem enviada sempre que um jogador colidir com uma moeda. Nesse instante, o servidor exclui a instância de moeda que foi "comida" e atualiza a pontuação do requerente. Os demais processos são devidamente notificados.
6. ***player_scored*:** Mensagem enviada aos clientes sempre que a pontuação de um dado jogador é modificada.

PROBLEMA DA EXCLUSÃO MÚTUA

Percebe-se pela dinâmica do jogo que os jogadores estão em constante disputa por recursos compartilhados, no caso as moedas. Isso remete ao problema da exclusão mútua pois, em um dado instante de tempo, dois ou mais processos podem enviar ao servidor a mensagem de ‘coin_eaten’, requerendo o ponto correspondente por comer a determinada moeda. Nessa situação o servidor deve ser capaz de identificar que somente um dos processos conseguiu de fato “comer” a moeda, notificando aos processos conectados somente o vencedor da disputa com pontuação atualizada.

Para tratar tal problema, utiliza-se outro módulo do nodejs conhecido como ‘locks’. Locks é uma implementação destinada unicamente para garantir que somente um processo tenha acesso a um recurso compartilhado, criando ao redor deste uma seção crítica.

```
var mutex = locks.createMutex();

function generate(){
    idCount++;

    var coin = {
        ID : idCount,
        position : position.getRandomPosition(),
        remove: function ( callback ) {
            var coin = this;
            mutex.lock( function() {
                var index = coins.indexOf( coin );

                if ( index > -1 ) {
                    coins.splice( index, 1 );

                    if ( typeof callback === 'function' )
                        callback();
                }

                mutex.unlock();
            });
        }
    };

    coins.push( coin );

    return coin;
}
```

Figura 02 - Exclusão Mútua

O módulo Locks fornece duas operações básicas: lock e unlock. A primeira requer ao mutex acesso à seção crítica, recebendo como parâmetro uma função que será executada assim que o acesso for concedido. Ao final da execução da função de remoção de moeda, a seção crítica é liberada (utilizando a operação unlock) para que um outro processo possa ter acesso à mesma.

Com isso, duas propriedades de sistemas distribuídos são satisfeita: Safety e Liveness. Com Liveness, garante-se que sempre que um processo deseje entrar na seção crítica, ou seja, sempre que algum jogador tentar comer uma moeda, o mesmo conseguirá. Com Safety, é garantido que no máximo um processo estará presente na seção crítica por vez, ou seja, no máximo um jogador irá comer uma moeda. Isso assegura que o algoritmo de exclusão mútua está de fato correto.

A implementação do algoritmo de exclusão mútua utilizado está centralizado no servidor para cada um dos recursos (moedas) compartilhados. Optou-se por essa abordagem pela característica da própria aplicação cliente-servidor. Concentrar o controle dos recursos no servidor favorece a simplicidade.

CONCLUSÃO

Ao desenvolver este trabalho percebe-se que a criação de jogos multiplayer é repleta de desafios no que se diz respeito à compartilhamento de recursos. Um problema interessante, para o qual adotou-se uma solução simplificada, se encontra no controle de posição dos jogadores. Deve-se concentrar o controle de posição dos jogadores? No cliente? No servidor? Se concentrarmos o controle no servidor, assumindo que node é uma plataforma de processo único, não haveria problema de concorrência, porém, o tempo de resposta para cada ação do usuário pode ser muito grande. Pensando em sistemas comerciais, tal abordagem pode não ser a melhor solução. Por outro lado, caso concentremos o controle no cliente, cada movimentação do usuário deverá realizar um multicast para o grupo, notificando os demais processos de que tal usuário se moveu. Além disso, a posição no espaço do jogo deveria ser encarada também como uma seção crítica, de modo a evitar que dois jogadores ocupem o mesmo espaço em um dado instante.

A fim de simplificar o projeto optou-se por manter o controle no cliente e permitir que dois jogadores ocupem o mesmo espaço em um dado instante de tempo. Após uma breve pesquisa, identificamos ainda que em geral, desenvolvedores de jogos costumam tratar este problema de uma maneira híbrida, tanto no cliente quanto no servidor. O avatar do usuário se movimenta livremente conforme ações do mesmo, porém sua posição será corrigida de forma mais suave pelo servidor.

Pode-se perceber que o mundo dos jogos é repleto de desafios e, através deste trabalho foi possível uma melhor compreensão de alguns problemas em sistemas distribuídos.

REFERÊNCIAS

1. https://nodejs.org/en/
2. http://phaser.io/
3. https://www.docker.com/
4. http://socket.io/
5. https://www.npmjs.com/package/locks
6. Material didático da disciplina Sistemas Distribuídos (slides)