Jogo da velha distribuído EP2

Autores:
Daniel Oliveira Sanches Leal, 11321180
Thiago Guerrero Balera, 11275297

Professor:
Daniel Macedo Batista

Protocolo

Protocolo

- O protocolo definido foi baseado no protocolo MQTT.
- O protocolo possui um header fixo para todos os tipos de pacote e, ainda, pode possuir um payload variável.
- O header fixo é composto por 2 bytes:
  - Byte 1: Tipo de pacote
  - Byte 2: Sessão de usuário
- O payload pode ser composto por múltiplas informações. Para cada informação, o payload terá n+1 bytes adicionais:
  - Byte 1: Número de bytes da informação (n)
  - Bytes 2...n+1: Informação
- É importante notar que, como o protocolo define somente 1 byte para representar o tamanho de uma informação, o tamanho máximo que uma informação qualquer pode ter são 256 bytes.
- Pela mesma explicação, o número máximo de clientes conectados simultaneamente são 256 (256 sessões distintas, onde `0` não é uma sessão válida).

Protocolo

- Os tipos de pacote são: Ping, New, In, Pass, Out, Fail*, List, Call, Start, Play*, End, Over*, Halloffame e Disconnect*.
- Com exceção dos tipos marcados com *, todos estes pacotes possuem uma versão de resposta.
    - Em especial para o pacote Fail, ele por si só já é um pacote de resposta.
- Para cada pacote que possui uma resposta, é mandado a sua versão de resposta em casos bem sucedidos, Fail caso contrário.
- Mais informações sobre cada tipo de pacote e sua estrutura está definido no arquivo README.md.

Servidor e Cliente

Servidor e Cliente

- Através da orientação a objetos, muitos comportamentos puderam ser generalizados para funcionarem tanto para o Cliente quanto para o Servidor.
- O arquivo *config.hpp* possui algumas configurações gerais do sistema.
- O arquivo *packet.hpp* representa a estrutura do protocolo em termos de código.
- Os arquivos *tcp.hpp* e *udp.hpp* possuem a implementação genérica de um Cliente/Servidor TCP e UDP, respectivamente.

Servidor TCP

- O processo principal será responsável por ouvir por novas conexões.
- O servidor pode ser de dois modos:
  - Multi processos
  - Mono processo
- Como o próprio nome já diz, o modo mono processo executa todas as operações do servidor em um único processo.
  - Essa categoria é útil para o caso dos servidores P2P da aplicação, onde não faz sentido aceitar múltiplas conexões diferentes.
- Para o modo multi processos, cada conexão irá gerar um processo novo.
  - Essa categoria é útil para o caso dos servidores de uma arquitetura Cliente/Servidor, onde faz sentido aceitar múltiplos clientes em simultâneo.

Servidor TCP

- Independente do modo, cada processo terá 2 threads.
  - A thread principal será responsável por ouvir, processar e responder o cliente.
    - No modo mono processo ela também será responsável por ouvir novas conexões após a desconexão de um cliente.
  - A segunda será responsável apenas por mandar pacotes de PING para o cliente.
- Se após *TIMEOUT_IN_SECONDS* (definido no arquivo de configuração) o cliente não mandar nenhuma mensagem, a conexão é encerrada.
- É importante notar que o tempo de timeout deve ser maior que a frequência do heartbeat (envio do pacote de PING), ou seja, o cliente só deve ser considerado desconectado se tiver passado tempo suficiente para responder ao heartbeat.

Servidor UDP

- O processo principal será responsável por ouvir todas as mensagens.
- Após o recebimento de uma mensagem, um processo filho é criado.
  - Este processo será responsável por processar a mensagem recebida e enviar uma resposta.
  - Com isso, o processo pai fica livre para ouvir novas mensagens de forma quase instantânea.
- Para cada mensagem, é checado se o IP/Porta emissor já foi visto antes.
  - Caso não tenha sido, é adicionado em uma lista e uma thread de heartbeat será criada.

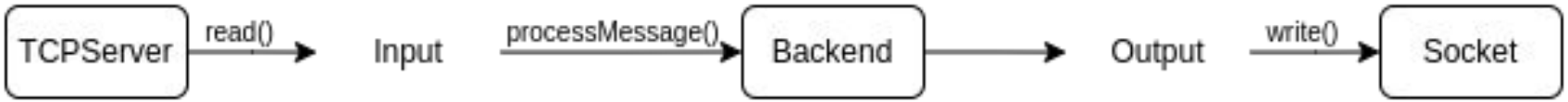
Servidor

Servidor

- O servidor deve ser iniciado contendo as portas para o servidor TCP e UDP, respectivamente.
- Uma instância da classe *Backend* será inicializada e passada para os servidores UDP e TCP, cada um estando em um processo dedicado.
- O processo pai é um processo gerenciador, ou seja, ele somente cria os dois processos citados no ponto anterior e espera.
  - Quando este processo morre, todos os outros também morrerão.
  - Para fins de execução, é sempre recomendado que somente este processo seja finalizado.
- Cada tipo de servidor terá um log dedicado.
- A estrutura da mensagem de log é:
  - [<Datetime> | <Server Type>:<Function>] <Message>

Backend

- É a classe responsável por gerenciar todo o processamento do servidor.
- Possui o seguinte fluxo:



FileHandler

- É a classe responsável por persistir os dados gerados pelo *Backend*.
- Cada informação presente em um arquivo está em uma linha diferente.
- Cada informação presente em um arquivo está na forma definida pelo payload do protocolo, ou seja, 1 byte de tamanho seguido do conteúdo.
- Os dados são persistidos de duas formas:
  - Usuário do sistema
  - Usuário conectado

FileHandler

- Os usuários do sistema são quaisquer usuários cadastrados pelo comando “New”.
  - Cada usuário possui um arquivo dentro de *TMP_FOLDER* chamado *<username>*.
  - O arquivo contém 2 linhas
    - Senha do usuário apontado por *<username>*.
    - Classificação do usuário.
- Os usuários conectados são quaisquer usuários que fizeram o login pelo comando “In”.
  - Cada usuário conectado possui uma sessão única entre 1 e 256 gerada aleatoriamente.
  - Cada usuário conectado possui um arquivo dentro de *ACTIVE_USERS_FOLDER* chamado <sessão>
  - O arquivo contém 4 linhas
    - Status do usuário (0 para livre, 1 para ocupado)
    - Nome do usuário
    - IP/Porta da conexão entre o servidor e o cliente
    - Porta em que o usuário está esperando por conexões P2P.

Cliente

Client

- O cliente deve ser iniciado com o IP e a porta do servidor, seguido do protocolo e a porta que o mesmo irá abrir para aceitar conexões p2p.
- O fluxo do cliente envolve:
  - O processo principal, lidando com os inputs do usuário e o devido envio destes.
  - 1 thread responsável por escutar respostas do servidor
  - 1 thread responsável por escutar por conexões de outros clientes
  - 1 thread responsável por escutar respostas do cliente ao qual ele se conectou quando usou o comando "call"

ClientHandler

- Responsável por lidar com os inputs do usuário e os outputs do servidor
- Responsável por unificar informações referentes ao usuário, partida e conexões

GameProcessor

- Responsável pelas ações tomadas ao receber mensagens do outro cliente e a respeito de uma partida em progresso com este cliente
- Responsável por fazer a comunicação com o servidor para informar sobre início e fim de uma partida

Match

- Classe responsável por:
  - Aplicar uma jogada no tabuleiro
  - Imprimir o tabuleiro
  - Checar se a partida encerrou e identificar empate ou derrota.
- O tabuleiro consiste em um estado em valor numérico
  - X possui um fator multiplicador 1 e O fator 2.
    - Ao jogar um X na posição 1 2, o estado passa a ter o valor somado em 1 * 3 ^ 1
    - Ao jogar O na posição 3 3, estado passa a ter o valor somado em 2 * 3 ^ 8
  - Sobre o valor desse estado são feitas operações de mod 3 subsequentes para descobrir se aquela posição está vazia ou se possui um X ou um O.

Experimentos

Experimentos

- Para fins de experimentação, cada processo estará em um contêiner do Docker.
  - Contêiner 1: Servidor
  - Contêiner 2: Cliente 1
  - Contêiner 3: Cliente 2
- Os processos filhos gerados pelo servidor estarão todos no contêiner 1.
- Estes contêineres estarão rodando em uma rede virtual gerenciada pelo Docker.
- Para ocorrer a comunicação entre os contêineres, o Docker dispõe de um sistema de DNS.
  - Contêineres dentro de uma mesma rede podem se comunicar através do seu nome de contêiner.
  - Para isto, o cliente deve implementar uma busca DNS pelo nome do contêiner que deseja descobrir o IP.

Ambientes

- **Computacional**
  - AMD Ryzen 5600G
    - 3.9 GHz
    - 6 núcleos (12 threads)
  - Ubuntu 20.04.4 LTS
- **Rede**
  - Placa Mãe Gigabyte A520M DS3H
    - Chip Realtek ® GbE
    - LAN (1000 Mbit / 100 Mbit)

Medições

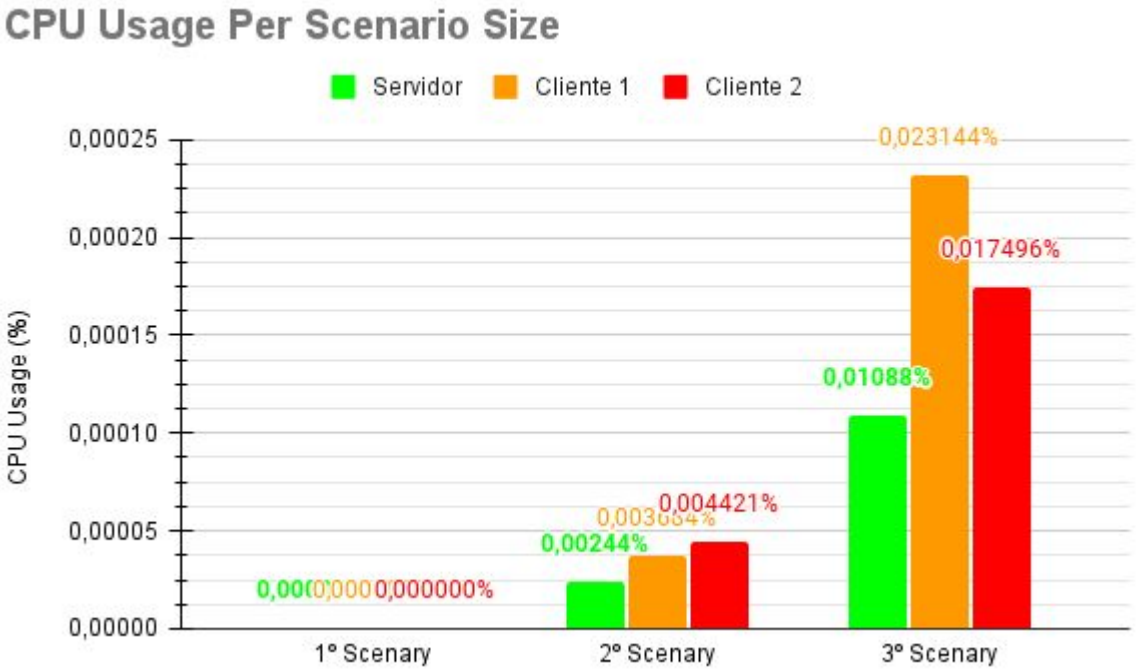
- As medições tanto de uso de CPU quanto de uso de Rede serão realizadas através do comando docker stats.
- Pela página da documentação, temos que podemos medir:
  - **CPU %:** the percentage of the host's CPU the container is using.
  - **NET I/O:** The amount of data the container has sent and received over its network interface.

Medições

- Para cada cenário, o comando será executado em um loop infinito.
  - Este loop será encerrado após a coleta de algumas observações ou após toda a execução de um cenário (último caso).
- A média das observações de uso da CPU sera o valor final para aquele cenário.
- Como as observações do uso de rede se acumulam, o valor final para aquele cenário será o valor da última observação.
- Para buscar relevância estatística, cada cenário será simulado 10 vezes.
- Será feito um intervalo de confiança com nível de confiança de 95%.

Resultados

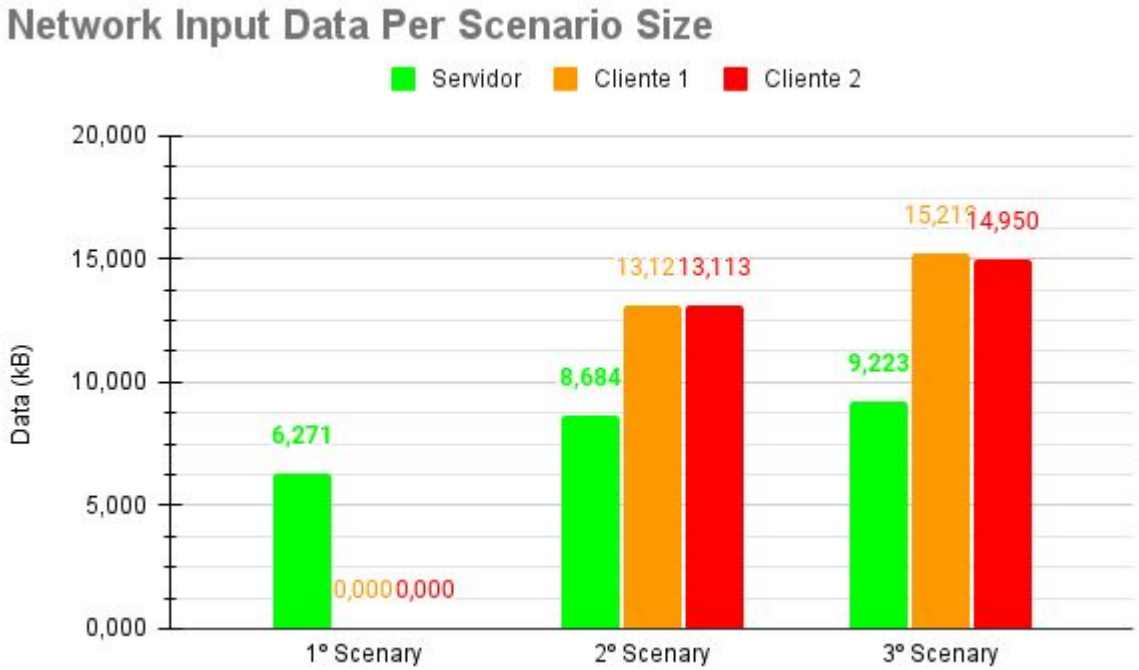
Uso de CPU



Intervalos de confiança:
[0; 0] [0; 0] [0; 0]
[0%; 0,007%] [0%; 0,027%] [0%; 0,0314%]
[0%; 0,048%] [0%; 0,077%] [0%; 0,051%]

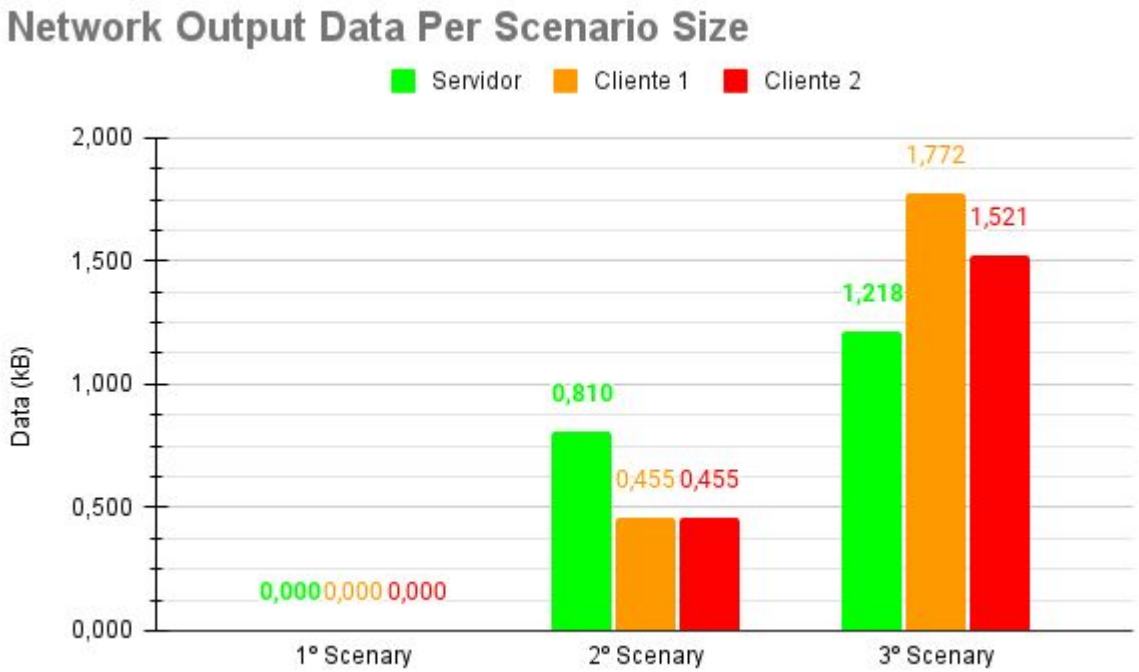
Uso de Rede: Recebimento de pacotes

Intervalos de confiança:
[5,522 kB; 7,02 kB] [0; 0] [0; 0]
[6,850 kB; 11,596 kB] [7,464 kB; 18,79 kB] [7,655 kB; 18,571 kB]
[7,073 kB; 10,295 kB] [14, 459 kB; 15,961 kB] [14,462 kB; 15,438 kB]



Uso de Rede: Envio de pacotes

Intervalos de confiança:
[0; 0] [0; 0] [0; 0]
[0,6218 kB; 0,998 kB] [0,371 kB; 0,539 kB] [0,371 kB; 0,539 kB]
[0,796 kB; 1,639 kB] [1,143 kB; 2,4 kB] [1,146 kB; 1,896 kB]



Resultados

- O gráfico de uso de CPU mostra que quando os clientes estão em partida, seu uso de CPU cresce bastante.
  - Entretanto, graças à arquitetura P2P, é possível notar que esse aumento não reflete para o servidor.
  - Idealmente, o uso de CPU do cenário 2 deveria ser ligeiramente menor do cenário 3.
  - Porém, é possível notar que isso não é bem verdade no gráfico.
  - O cenário 3 teve menos medições do que os demais cenários, pois o mesmo possui um início e fim bem definidos. Ou seja, durante todas as suas medições estavam acontecendo algum processamento, o que não ocorre no cenário 1 e 2.
  - Esse comportamento ligeiramente diferente deve ter influenciado nesse aumento de uso.
- Observações similares podem ser feitas em relação ao uso de rede.
  - Novamente graças à arquitetura P2P.

Resultados

- Mesmo com zero clientes conectados, ainda houve um pequeno recebimento de mensagens pelo servidor.
  - Isso se deve a algumas trocas de mensagem que acontecem visando o funcionamento da rede virtual do Docker.
- Alguns intervalos de confiança de uso de rede ficaram grandes.
  - Como os testes precisaram ser feitos manualmente, isso restringiu bastante o número de experimentos que pode ser feito buscando significância estatística.