MAC0352 - Redes de Computadores e Sistemas Distribuídos (2022)

EP02 - Criação de um servidor para jogo da velha

Protocolo

Estrutura dos pacotes

Os pacotes gerados na comunicação são bastante simples, possuindo um cabeçalho fixo e um payload (corpo) contendo informações sobre aquela requisição.

- O primeiro byte do cabeçalho indica o tipo do pacote.
- O segundo e terceiro bytes do cabeçalho indicam o tamanho restante do pacote e são utilizados para saber quantos bytes o payload possui.
- No projeto existe um total de vinte pacotes.
- Os únicos tipos de dados transmitidos nos pacotes são inteiros ou strings. Para os inteiros, existem dois tipos: com um ou dois bytes. As strings são escritas todas na mesma convenção: os primeiros dois bytes servem para indicar o tamanho da string e os bytes seguintes são o seu valor.

Projeto

Estrutura do projeto

- Os principais arquivos para o cliente e servidor são respectivamente *src/client.cpp* e *src/server.cpp*
- Os pacotes foram implementados nos arquivos *src/packages.cpp* e *src/packages.h.*
- Algumas funcionalidades do cliente e servidor foram extraídas e implementadas em arquivos separados (*client-helper.cpp* e *server-helper.cpp*).
- As funcionalidades referentes ao jogo da velha em si foram implementadas nos arquivos *game-board.cpp*
- A escrita de logs pelo servidor foi implementada nos arquivos *log.cpp*.
- Funções para uso geral foram implementadas em *util.cpp*.
- Os logs gerados são salvos no arquivo *ttt.log* e o registro de usuários é salvo em *users.db*.

Estrutura do servidor

O servidor permanece escutando em uma porta por conexões de clientes. Assim que uma nova conexão é feita, o servidor cria um processo filho para cuidar desse novo cliente.

- Cada processo filho possui três threads que executam ações diferentes:
  - **heartbeat_handler_thread**: responsável por enviar os heartbeats para o cliente a cada 10 segundos
  - **invitation_handler_thread**: responsável por verificar se houve algum convite para o cliente sob seu processo
  - **entrada_handler_thread**: responsável por receber pacotes do cliente, processá-los e tomar as ações necessárias referente a cada um dos diferentes tipos de pacote

Estrutura do cliente

O cliente possui duas threads para seu funcionamento:

- **ui_thread:** responsável por esperar comandos do usuário, processá-los e enviar pacotes para o servidor requisitando alguma resposta.
- **entrada_thread:** espera por pacotes do servidor e retorna suas respostas para o mesmo (caso dos heartbeats) ou exibe a resposta do servidor para o usuário.

Quando o servidor perde a conexão, o cliente tenta se conectar novamente a cada 1 segundo. Ele faz isso durante um período de 3 minutos e, caso não tenha sucesso, encerra o seu processo. Caso tenha sucesso, o cliente envia um pacote para o servidor identificando-se e tudo retorna ao normal.

Jogo

Comunicação cliente / servidor

Sejam C1 e C2 dois clientes e P1 e P2 os processos filhos do servidor que cuidam desses clientes respectivamente.

- Digamos que C1 convide C2 para jogar. Então, C1 envia um pacote para P1 com o nome de C2 e depois, P1 avisa P2 que C1 quer jogar com C2.
  - Para cada cliente, existe uma variável client_invitation a qual todos os processos filhos têm acesso e que indica se este cliente possui um convite para partida e quem fez esse convite. Assim, P1 atualiza essa variável em C2 indicando que C1 foi quem fez o convite.
  - O client_invitation também armazena a resposta do convite. Então, C2 envia para P2 a resposta e este atualiza a variável.
  - Depois de enviar o convite, P1 fica verificando se houve resposta a cada 1 segundo e assim que a resposta aparecer ele envia para C1.

Comunicação cliente / servidor

- Para iniciar o jogo um cliente utiliza a função “get_free_port” para encontrar um porta livre e iniciar uma conexão com seu rival.
- Assim que C1 receber a resposta positiva junto da porta, ele pede faz a conexão com C2 por meio dessa porta e o jogo começa.
- Durante o jogo a thread “entrada_match_thread” fica recebendo e processando os pacote de conexão do jogo. A thread “ui_match_thread” fica responsável por interagir com o usuário e espera os comandos do jogador.
- A thread “latency_match_thread” é esponsável por checar o delay entre os clientes. Ela envia uma requisição de ping para o outro jogador e calcula o tempo de delay durante a resposta.
- Assim que o jogo acaba, cada cliente é responsável por enviar sua pontuação para o servidor.

Experimentos

Ambiente de testes

- Computador Acer Predator Helios 300
- 32 GB RAM
- 2 GHZ
- 8 núcleos
- Ubuntu 18.04 LTS
- Chip Realtek PCIe GBE
- Internet de 100 MB/s

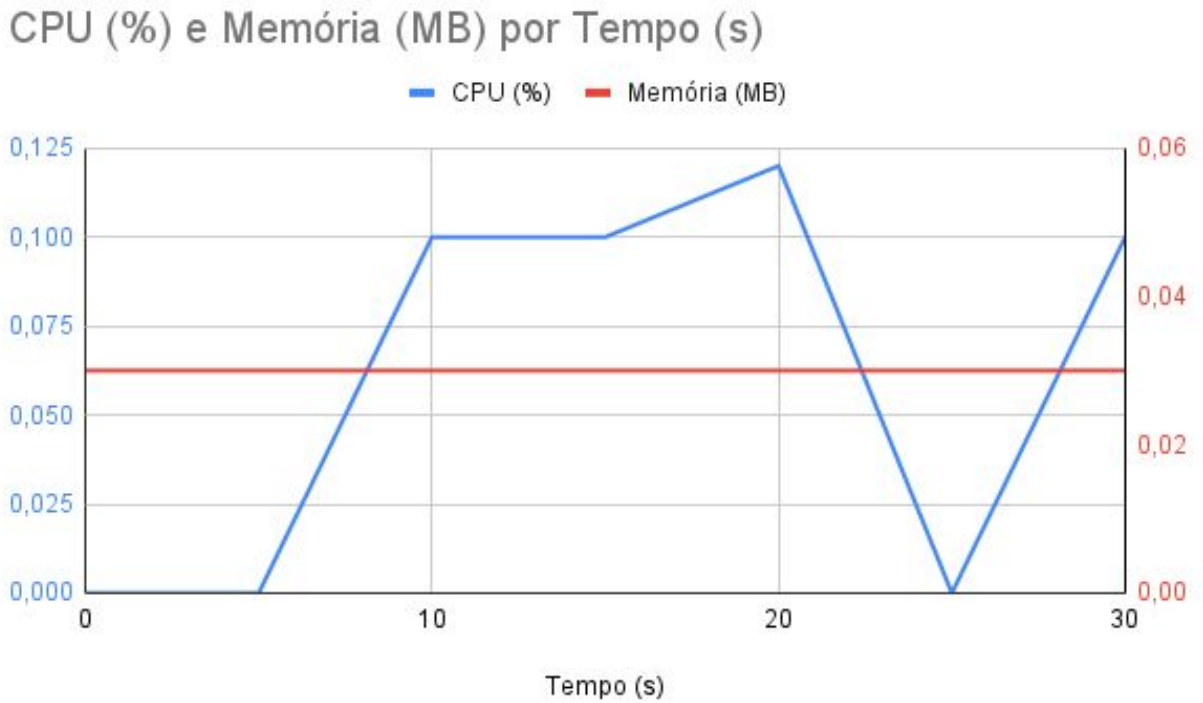
Experimentos

- O servidor e os clientes dos testes foram executados dentro de contêineres do Docker.
- As portas dos contêineres foram vinculadas às portas do host.
- Com o servidor e clientes dentro de contêineres do Docker, pode-se obter as métricas de CPU (CPUPerc) e Rede (Net I/O) de forma mais simples através do comando docker stats.

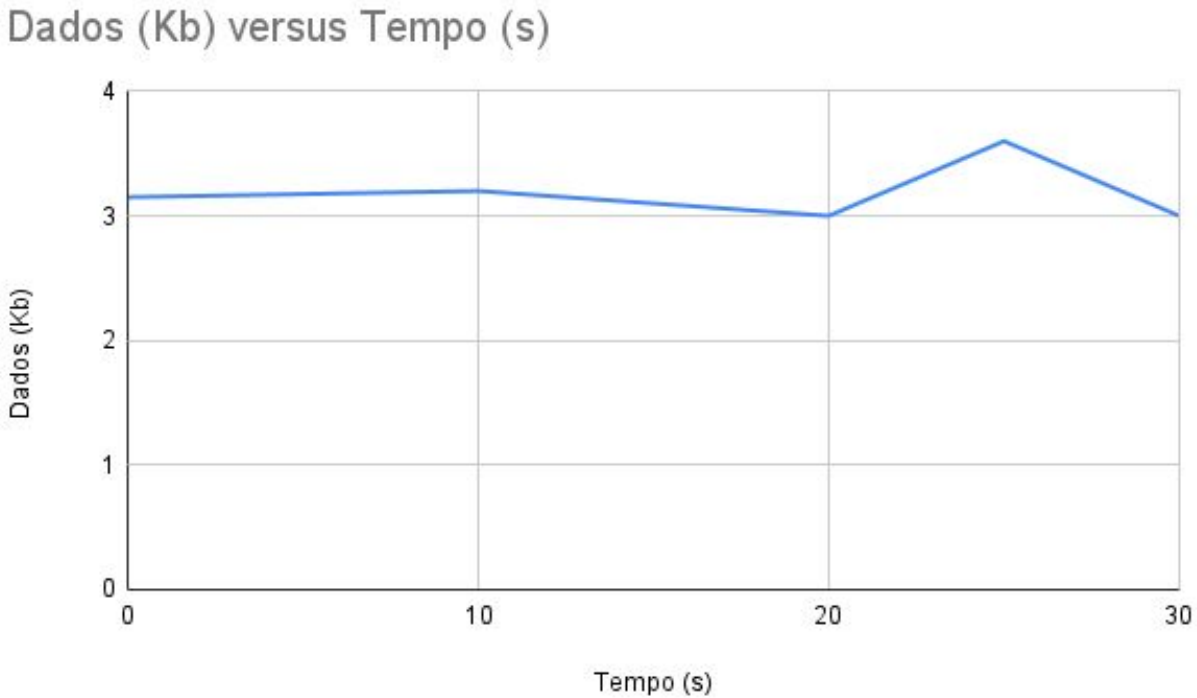
Cenários de teste

- Servidor rodando sem nenhum cliente.
- Servidor rodando com dois clientes conectados sem enviar comandos.
- Servidor rodando com dois clientes conectados enviando comandos.

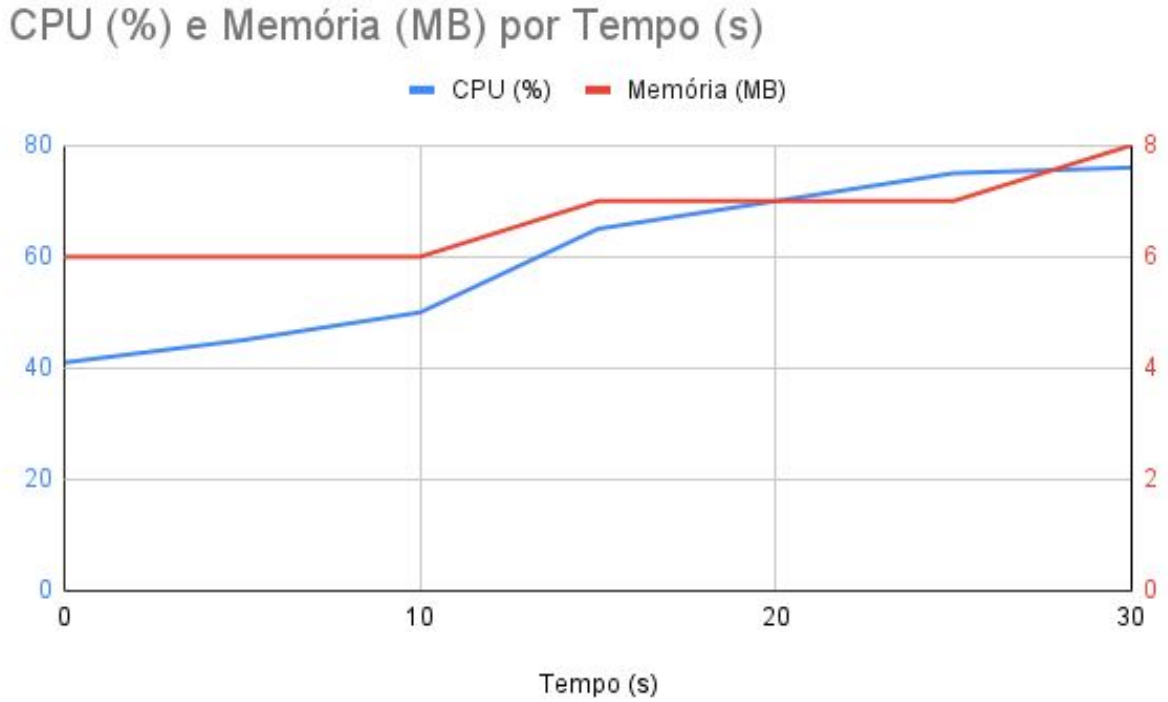
Resultados do servidor sem clientes conectados



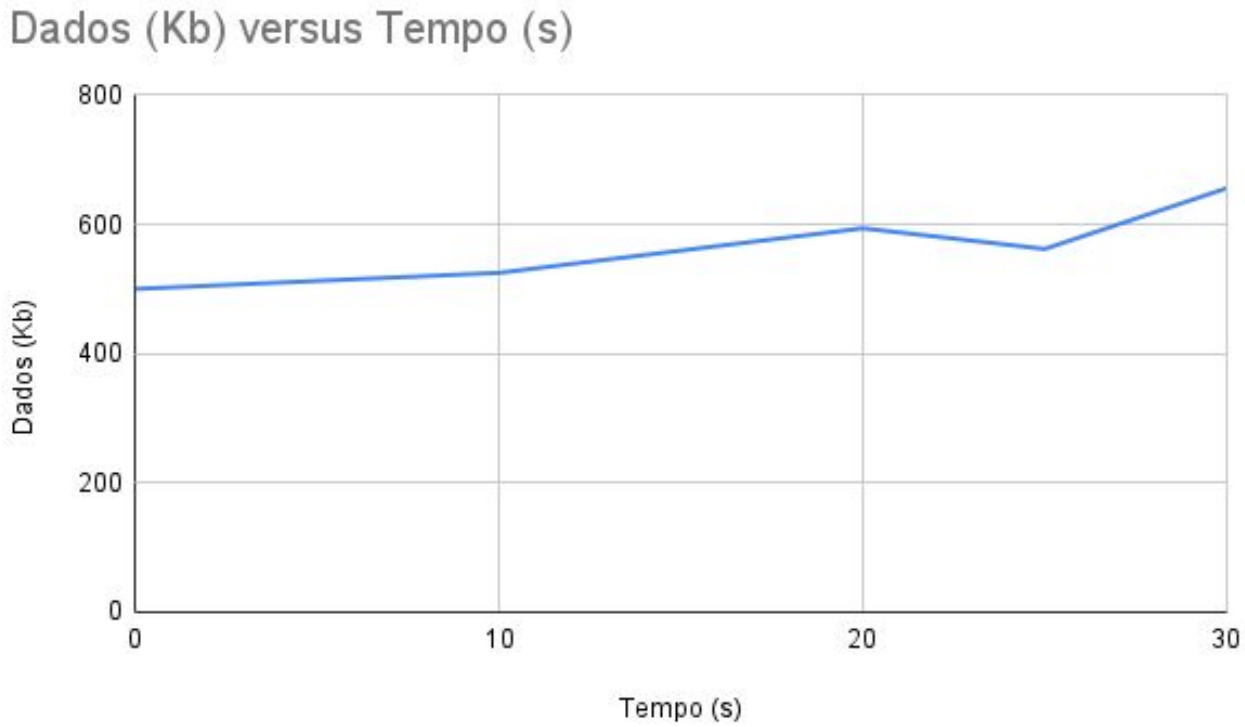
Resultados do servidor sem clientes conectados



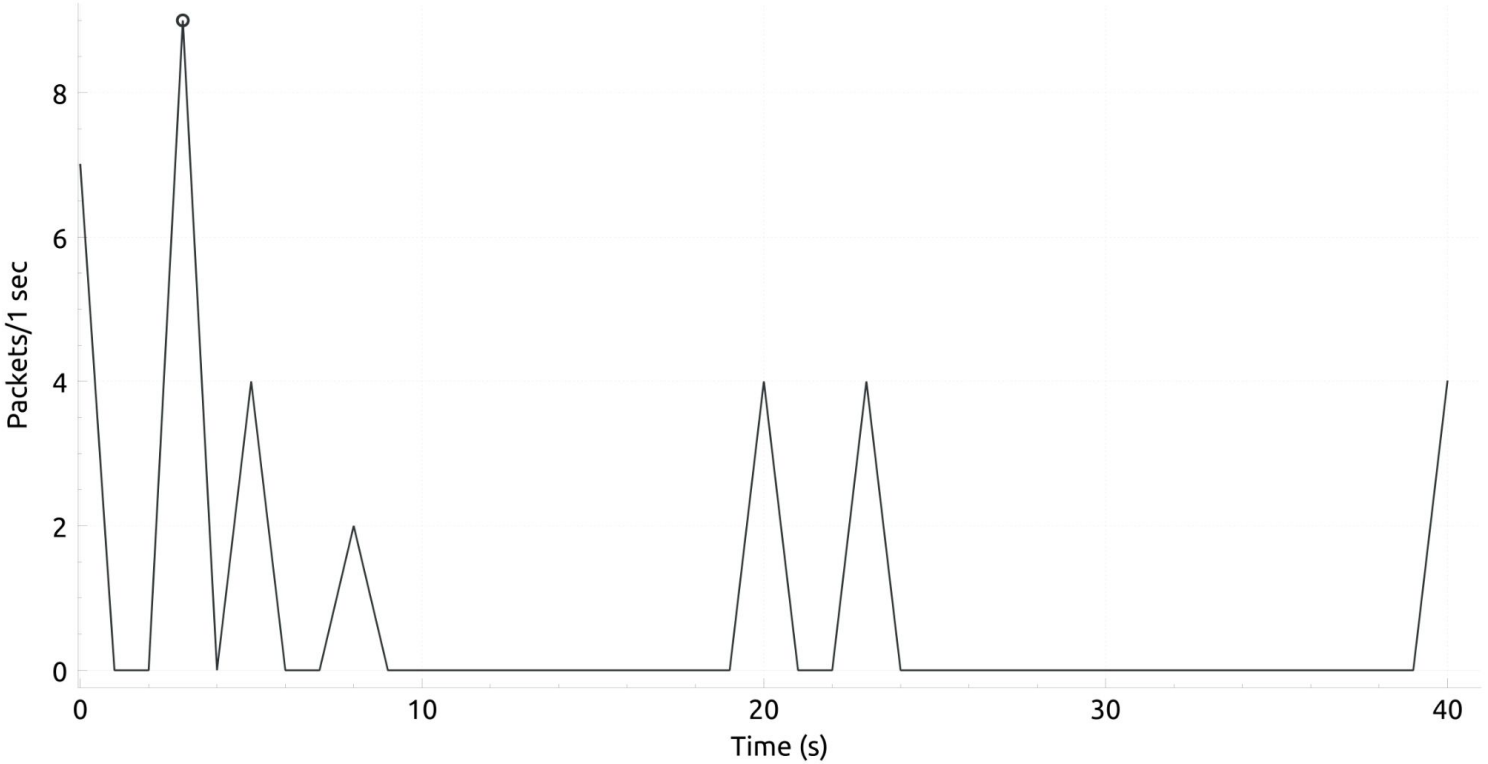
Resultados do servidor com dois clientes conectados sem jogar



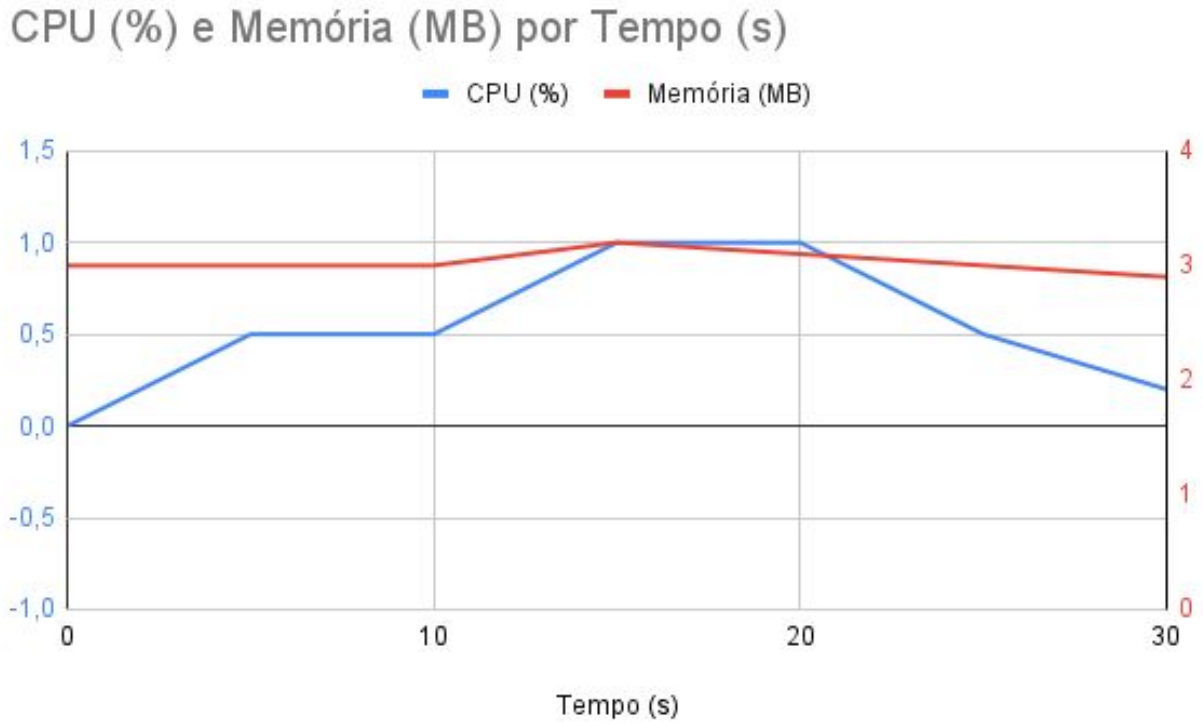
Resultados do servidor com dois clientes conectados sem jogar



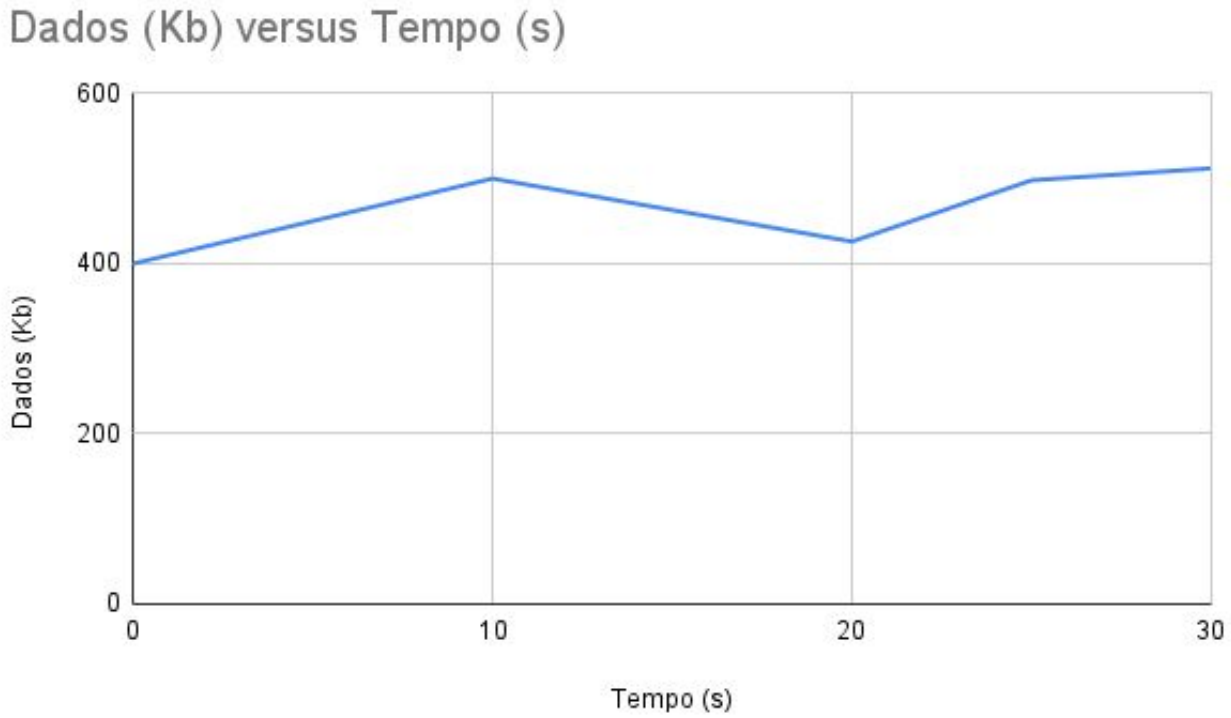
Resultados do servidor com dois clientes conectados sem jogar (gráfico de I/O do Wireshark)



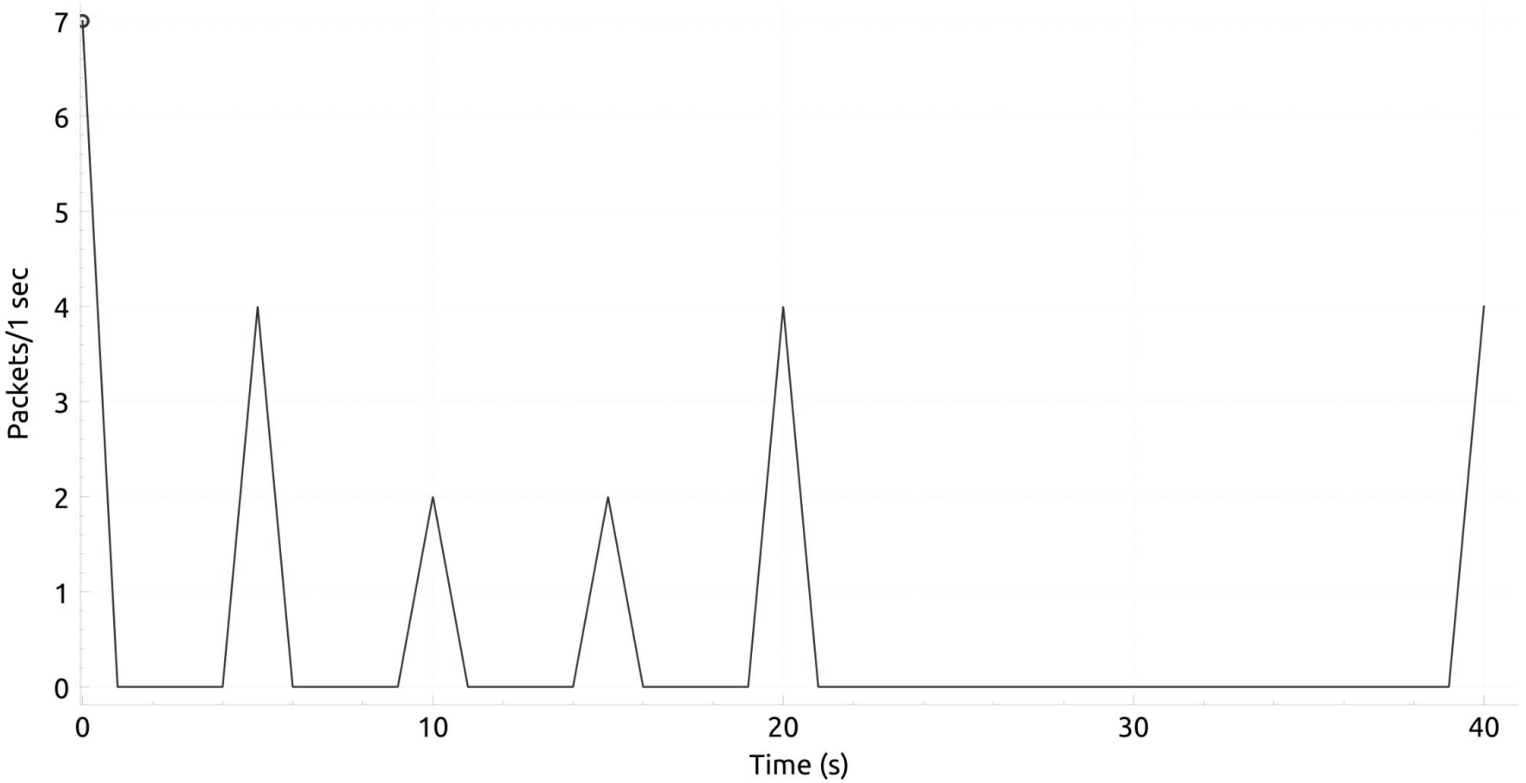
Resultados do cliente conectado ao servidor sem jogar



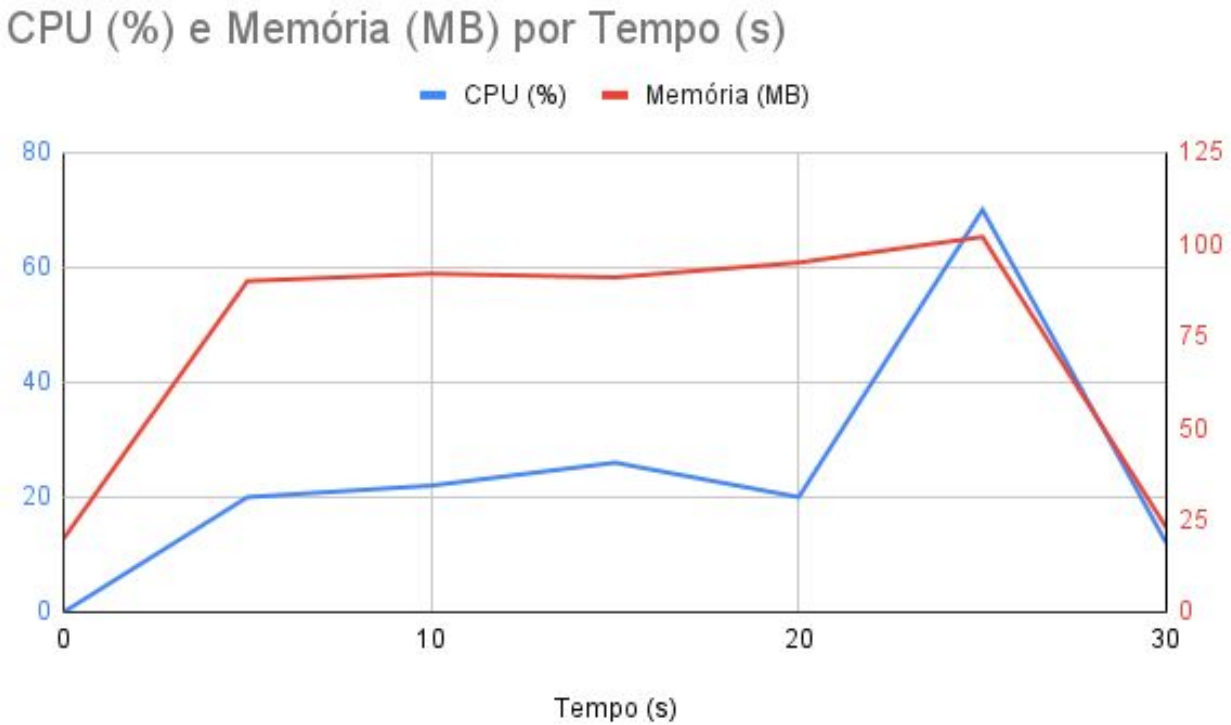
Resultados do cliente conectado ao servidor sem jogar



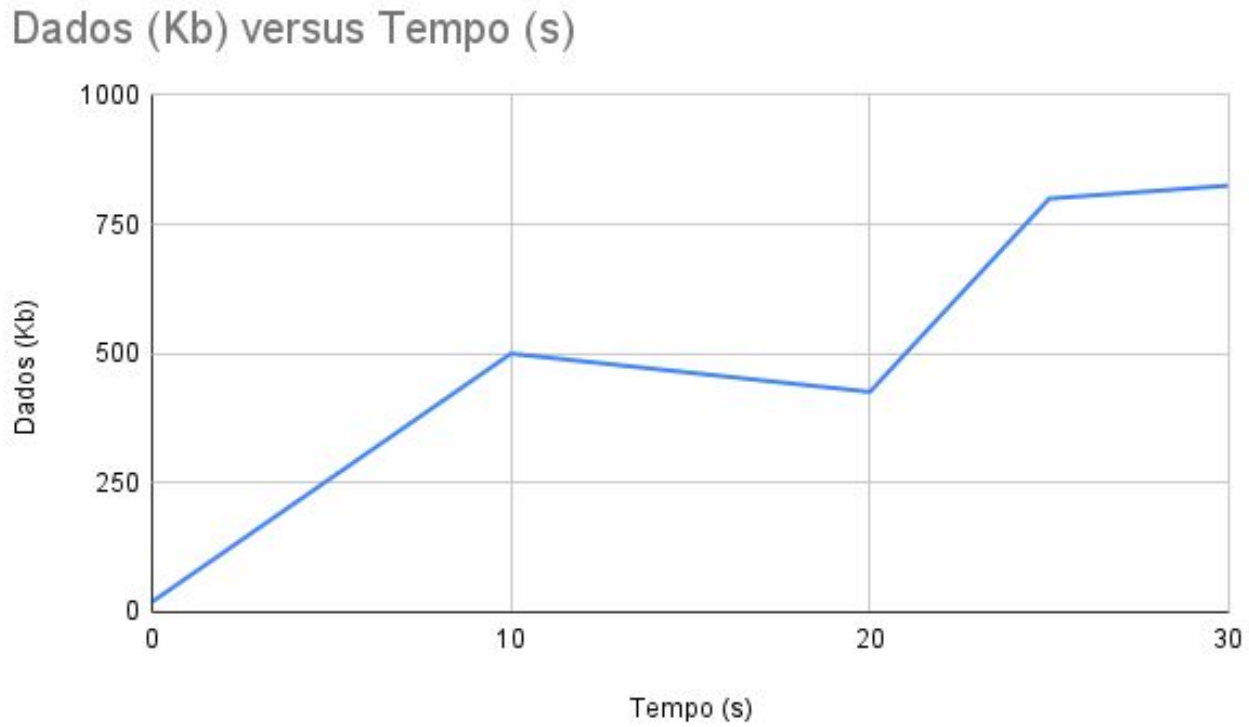
Resultados do cliente conectado ao servidor sem jogar (gráfico de I/O do Wireshark)



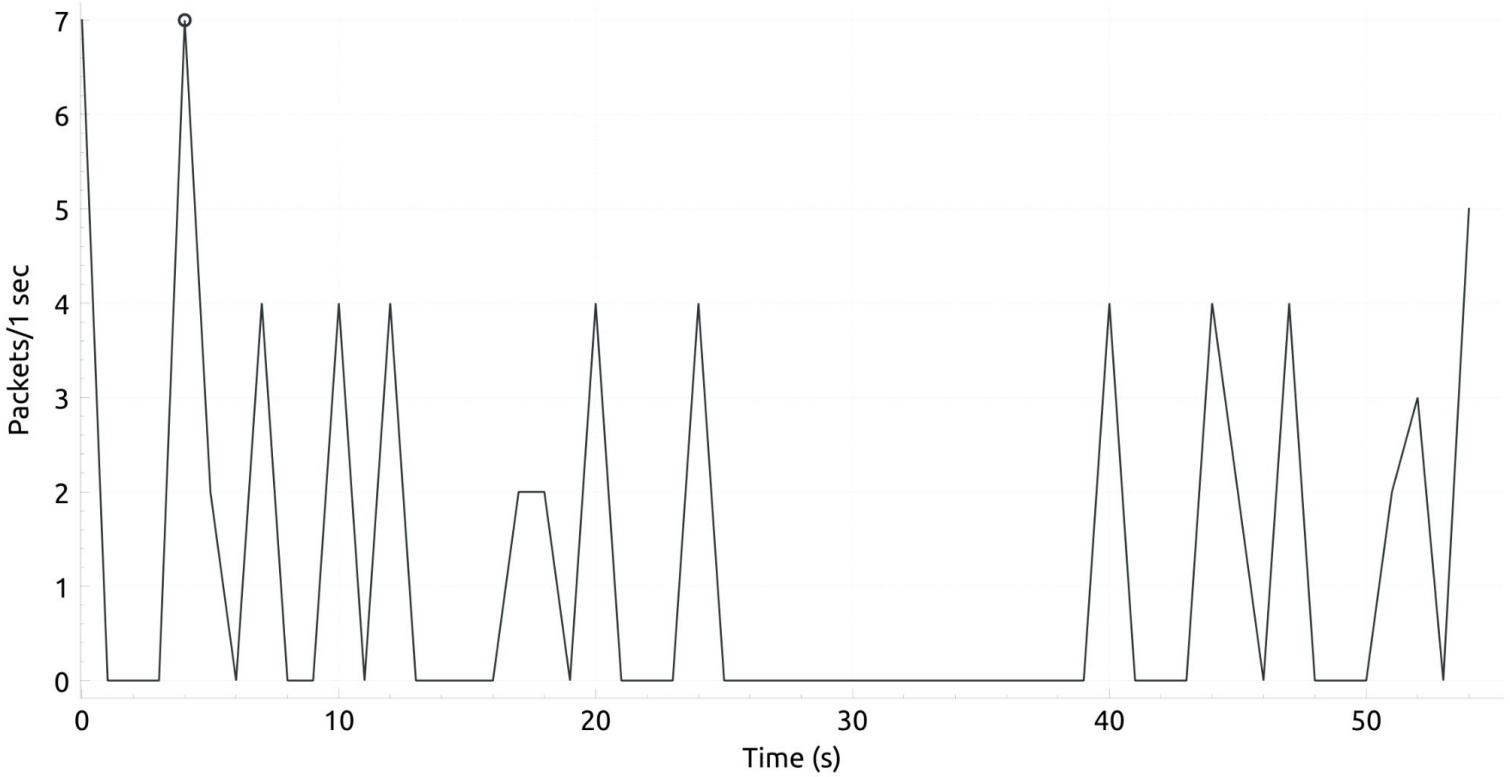
Resultados do servidor conectado a dois clientes jogando



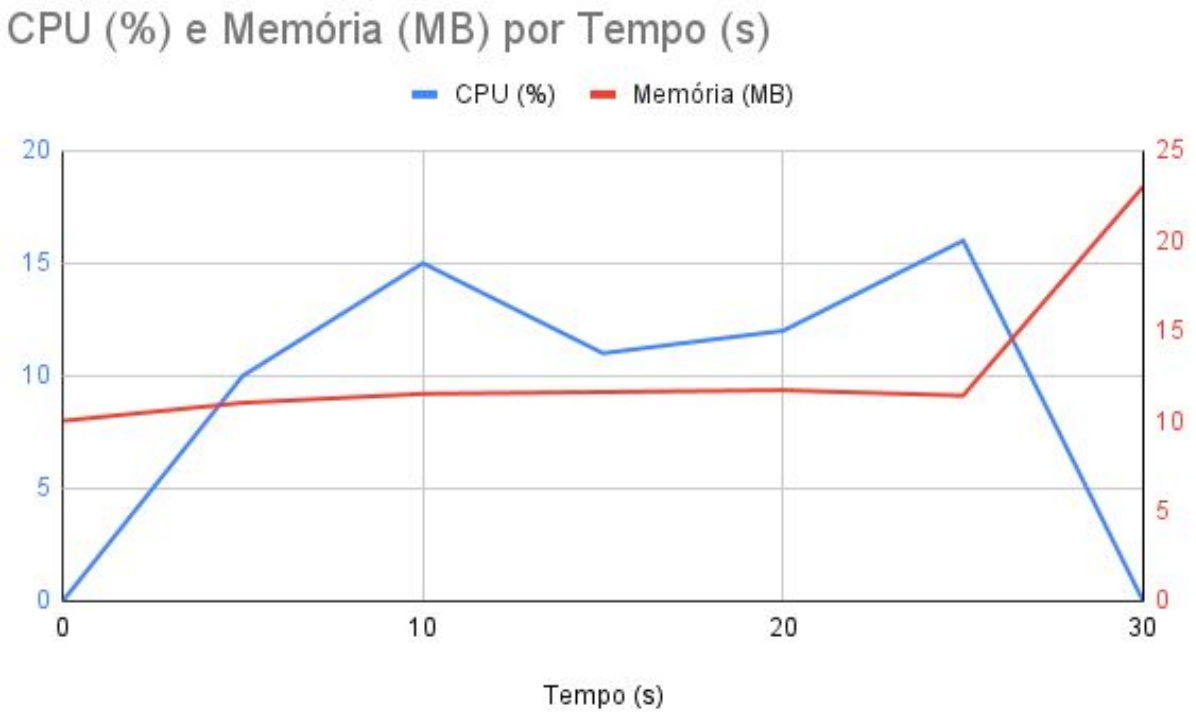
Resultados do servidor conectado a dois clientes jogando



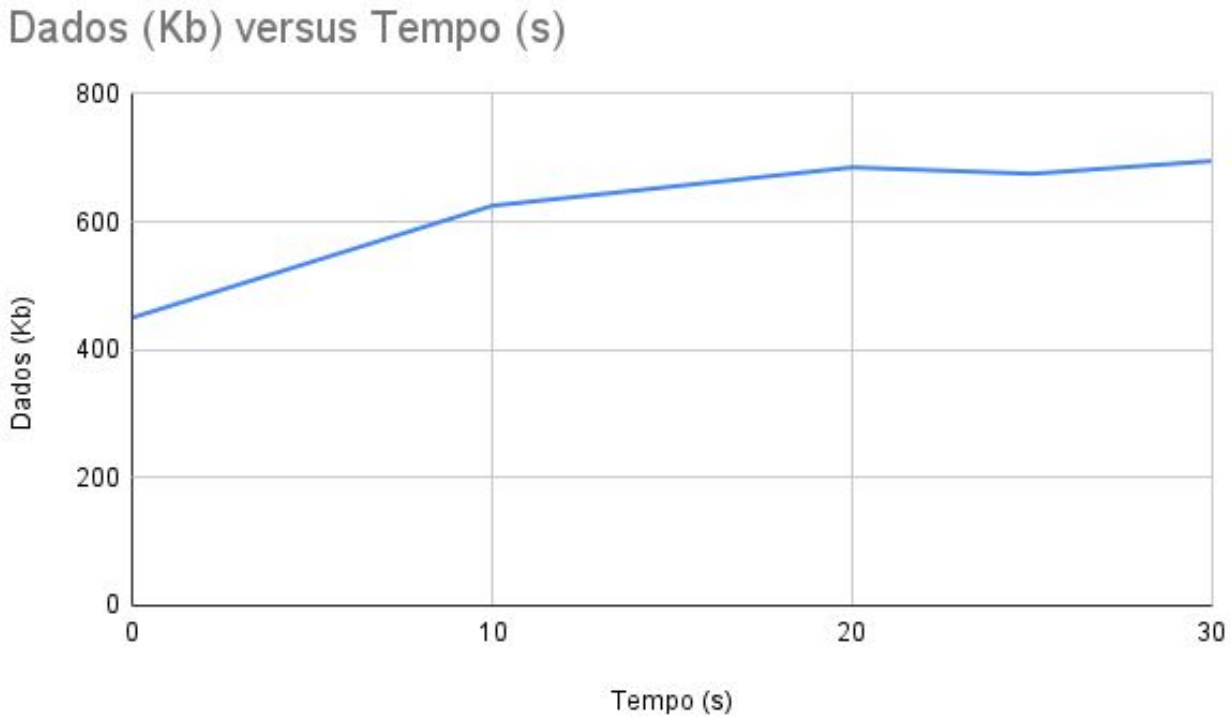
Resultados do servidor conectado a dois clientes jogando (gráfico de I/O do Wireshark)



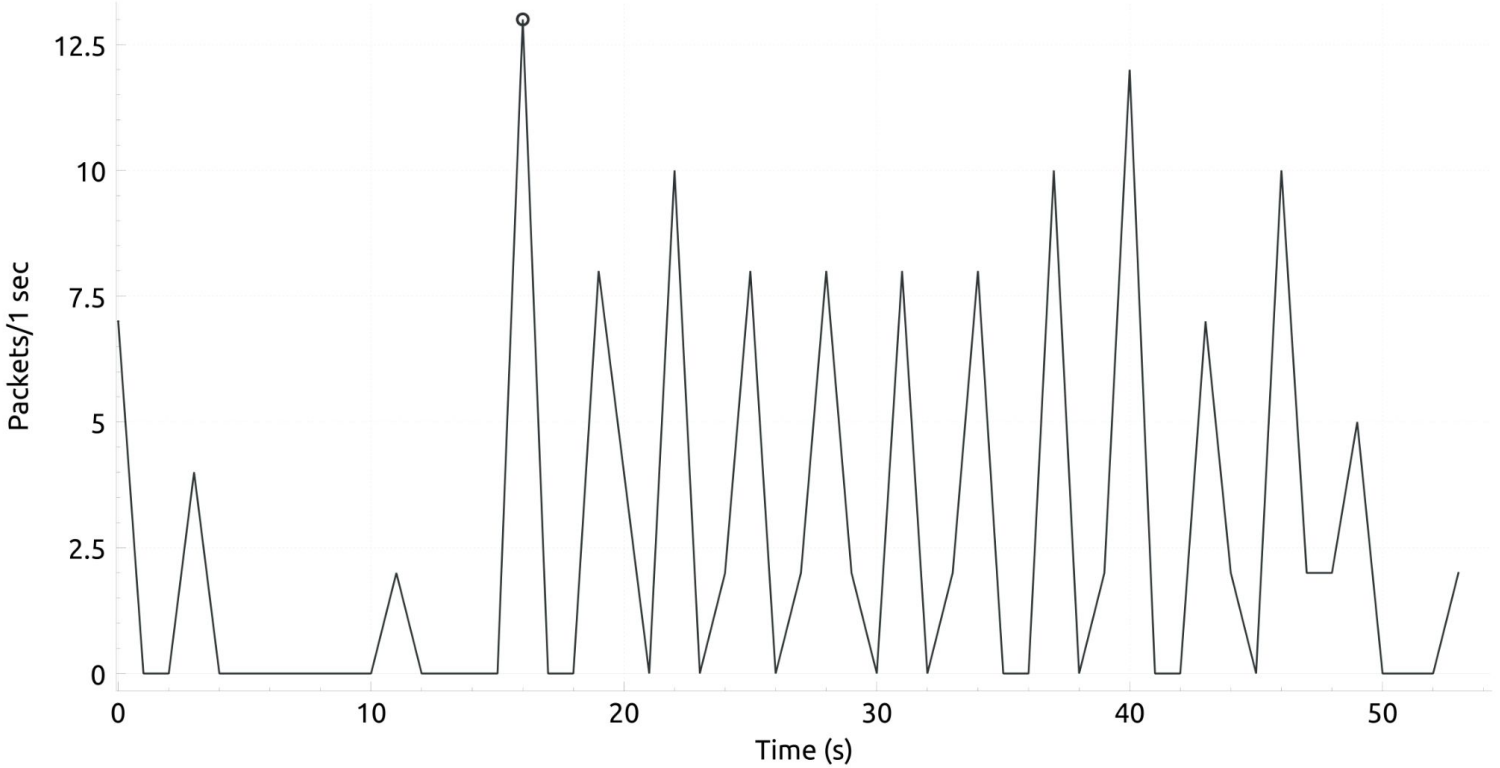
Resultados do cliente conectado ao servidor e jogando



Resultados do cliente conectado ao servidor e jogando



Resultados do cliente conectado ao servidor e jogando (gráfico de I/O do Wireshark)



Comentários

- Quando o servidor não possui nenhum cliente conectado, não existe praticamente nenhum uso de rede (apenas as checagens padrões do Docker). Por isso não foi exibido nenhum gráfico de rede nesse cenário.

- Quando o servidor está com dois clientes conectados, mas sem nenhuma operação por parte deles, o uso de rede é bem baixo, vemos que o servidor possui um pico maior no envio de pacotes quando os dois clientes se conectam e depois existem alguns picos mais baixos referentes às operações de ping do clientes. Por parte do cliente, vemos o mesmo padrão: um pico maior ao se conectar e picos menores referentes aos pings.

- Quando os clientes começam a jogar e enviar comandos nota-se um uso de rede mais intenso (porém ainda bem baixo). Por parte do servidor, o padrão anterior quase se mantém. Por parte do cliente, observamos um uso grande da rede e envio de pacotes, que ocorre quando os jogadores estão jogando e trocando mensagens entre si.