EP1 - Redes

Aluno: Francisco Eugênio Wernke
NUSP: 11221870

Funcionamento

Conexão TCP

Para este EP, usei o código exemplo disponibilizado no e-disciplinas. Logo, toda a parte da conexão TCP foi resolvida pelo código disponibilizado. Esta abstração faz sentido dado que estamos mais preocupados com a camada de aplicação neste EP.

Assim, não entrarei em detalhes sobre a conexão TCP, basta dizer que usei a lógica já disponibilizada pelo Professor.

Funcionamento dividido por operações

Ao desenvolver a camada de aplicação MQTT do servidor, separei algumas operações que deveriam ser realizadas pelo servidor, elas são:

- Reconhecimento de mensagens MQTT
- Resposta ao cliente (*Suback*, *Connack, Pingresp*)
- Criação de tópicos
- Publicação de uma mensagem para um tópico
- Leitura de tópicos

Reconhecimento de mensagens MQTT

- As mensagens enviadas pelo cliente MQTT são lidas no vetor recvline e então podemos descobrir a operação desejada pelo header (byte 1).
- O cliente envia:
  - **Connect**: Espera uma mensagem de volta que diga que o servidor entende o MQTT versão 3.1.1 e responderá as ações do cliente.
  - **Publish**: Envia um tópico e uma mensagem a ser propagada neste tópico. Passará pela ação **publishToTopic**.
  - **Subscribe**: Envia um tópico e espera por mensagens do mesmo. Passará pelas ações **createTopic** e **waitForMessages**. Também espera uma confirmação de sucesso.
  - **Disconnect**: Sinaliza que não deseja receber outras mensagens. Fechamos a conexão do cliente.
  - **Pingreq**: Espera uma sinalização de que a conexão do cliente com o servidor está saudável.

Resposta ao cliente

- Ao enviar mensagens, o cliente pode querer algumas respostas para a confirmação de que o servidor entendeu o pedido.
- As mensagens que precisam de resposta são:
  - Connect: Espera receber de volta um Connack. Pacote produzido na função sendConnack.
  - Subscribe: Espera receber de volta um Suback. Pacote produzido na função sendSuback.
  - Pingreq: Espera receber de volta um Pingresp. Pacote produzido na função sendPingresp.
- Essas mensagens são criadas e logo enviadas ao cliente após a realização das ações envolvidas, caso necessárias.

Criação de tópicos

- Ao receber um Subscribe, verificamos se existe um diretório para o tópico desejado dentro do sistema de arquivos, em um caminho base, e criamos caso não exista. Depois, criamos uma FIFO que será acessada pelo processo que cuida do cliente.
- Para isto, usamos a função **mktemp** que cria um novo arquivo no diretório do tópico usando um template de 6 caracteres para manter a unicidade do nome.
- Após a criação do arquivo de nome único, precisamos transformá-lo em um arquivo especial FIFO, usando a função **mkfifo**.
- Criada essa fifo, precisamos somente armazenar o caminho para ela e está feito o registro do cliente naquele tópico.

Publicação de mensagens

- Ao recebermos um Publish, o servidor desvendará o tópico e a mensagem de dentro do pacote.
- Então, acessa o diretório do tópico dentro do caminho base e lista todas as FIFOs dentro deste diretório.
- Cada FIFO dentro do tópico deve ser aberta para a escrita e então escreve-se a mensagem para o arquivo.
- Este comando de abertura deve ser sincronizado com a abertura para a leitura. Como vamos ver adiante, o subscriber está sempre esperando por essa abertura.

Leitura de tópicos

- Assim que um subscriber cria a sua FIFO dentro do diretório do tópico, ele deve ficar em um laço de abertura da mesma FIFO em modo de leitura, para que um publisher consiga abrir a mesma FIFO para escrita e escrever nela.
- Ao mesmo tempo, precisamos responder o cliente que envia pacotes de Pingreq ou Disconnect no socket.
- Para realizarmos as duas ações, o servidor cria duas threads, uma para a leitura do socket e outra para a leitura da FIFO.
- Como só paramos de ler mensagens da FIFO ao recebermos um Disconnect, a thread que lê do socket será esperada pela thread principal (via pthread_join). Enquanto isso, a thread que lê da FIFO será cancelada assim que a thread do socket acabar.

Leitura de tópicos

- Para a leitura do socket, o código se parece bastante com a leitura das operações que fazíamos antes. A diferença é que só aceitamos Pingreqs e Disconnects.
- Para a leitura da FIFO, tentamos constantemente abrir a FIFO para leitura. Isso se deve ao fato de que, se uma FIFO for fechada por todas as threads, ela é excluída pelo Sistema Operacional, logo precisamos manter a FIFO sempre ativa. Além disso, precisamos abrir novamente a FIFO para toda leitura.
- Quando o cliente envia um Disconnect para o servidor, a FIFO é excluída e fechamos a conexão com o cliente.

Testes

Premissas básicas

- Os experimentos foram feitos com 0, 100 e 1000 clientes, sendo eles 50% subscribers e 50% publishers.
- O experimento visa entender o que ocorre com o servidor em regime estacionário (sem clientes) e depois com um pico de clientes se conectando (100 ou 1000)
- Foram usadas 3 métricas:
  - Uso de CPU do container;
  - Saída de dados do container;
  - Entrada de dados para o container.

Computador usado nos testes

O computador usado tem as seguintes propriedades:

- macOS Monterey versão 12.2.1
- Processador 2.6GHz Intel Core i7 6-core
- Memória 16 GB 2667 Mhz DDR4

Preparando as estruturas

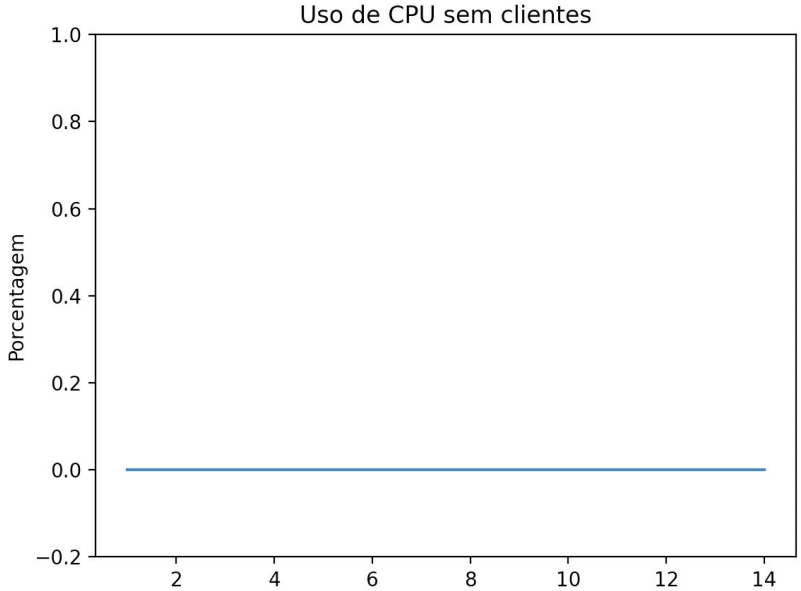
- O experimento começa levantando toda a estrutura do servidor no Docker, com somente um container executando o código do servidor.
- Nesta network criada pelo Docker, ligamos uma porta interna do container a uma porta do Sistema Operacional. Assim, conseguimos nos comunicar com o servidor.
- Foi usada a função **docker stats** da própria interface do Docker para retirarmos as métricas. Criamos um processo que retira essas métricas de 1 em 1 segundo durante a execução do teste.

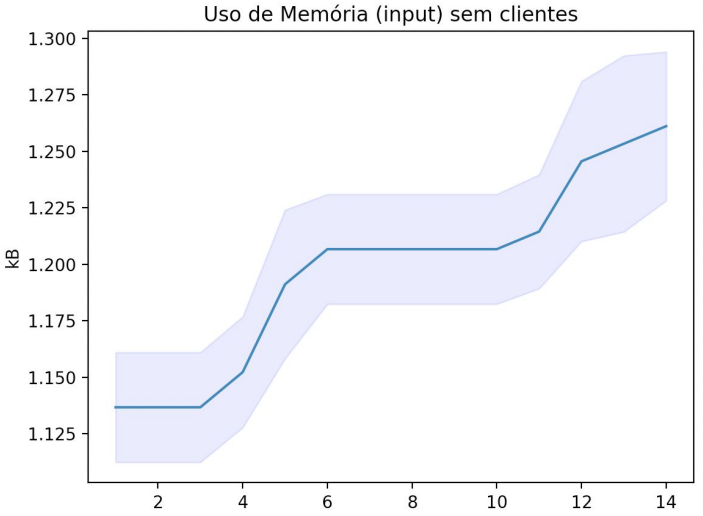
Teste

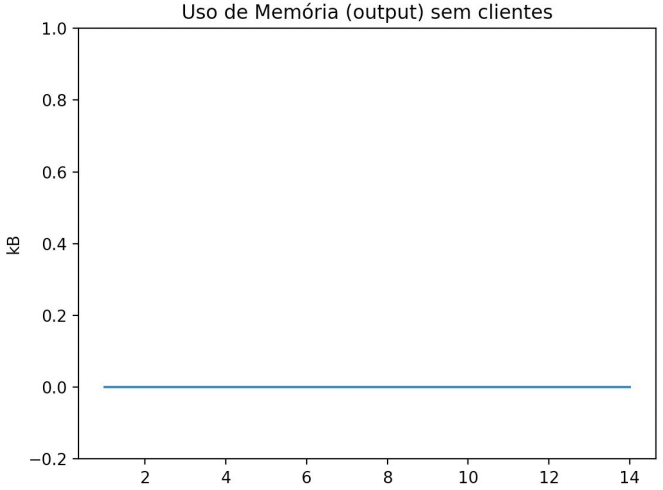
- Começamos a disparar os clientes após algumas medições do **docker stats**.
- Recolhemos então 15 medidas em arquivos txt para a análise posterior.
- Repetimos o processo 10 vezes para tirarmos a média e o intervalo de confiança em cada intervalo de tempo.
- Após o teste, fechamos todos os processos iniciados e desligamos o container docker.

Resultados

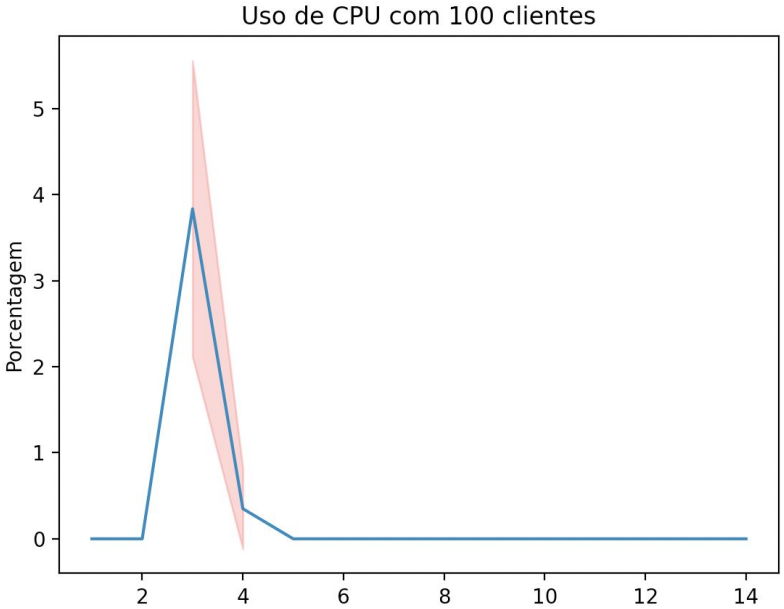
Teste sem clientes

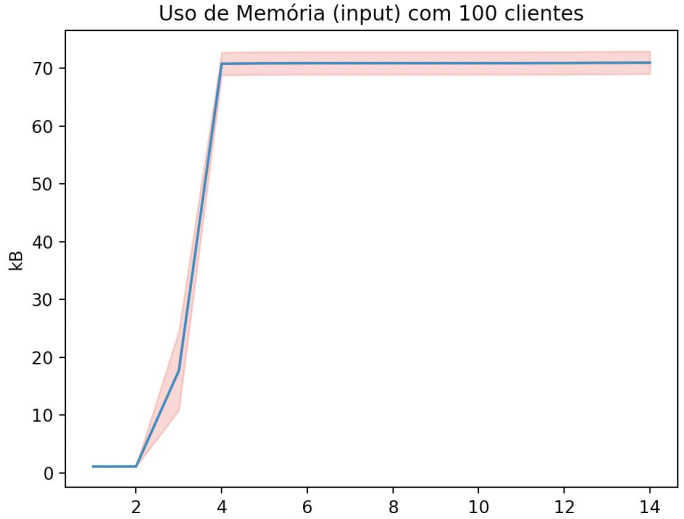


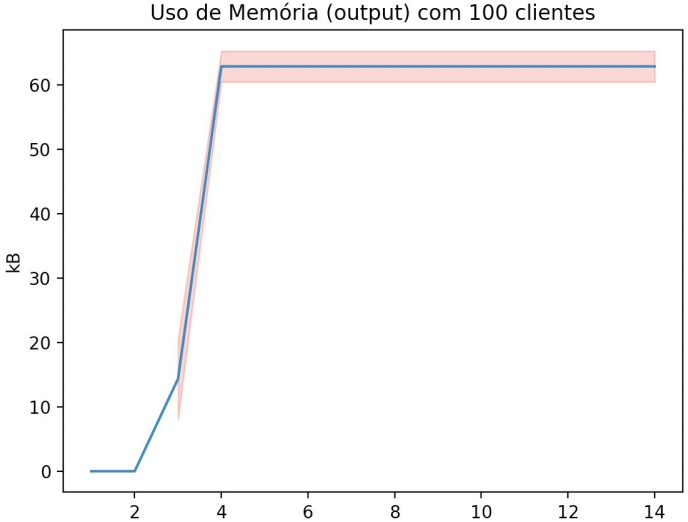




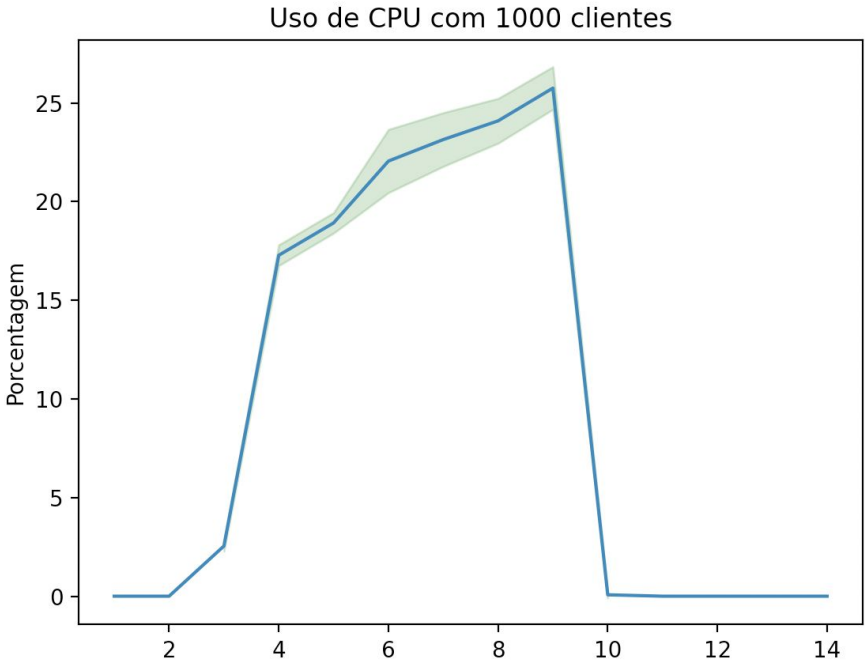
Teste com 100 clientes

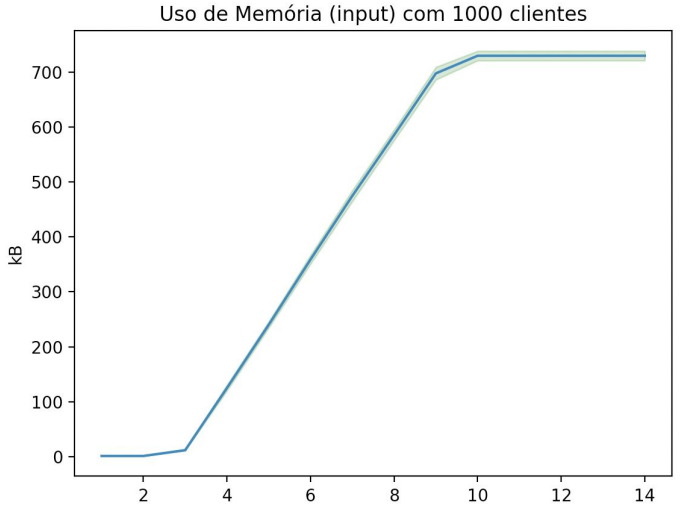


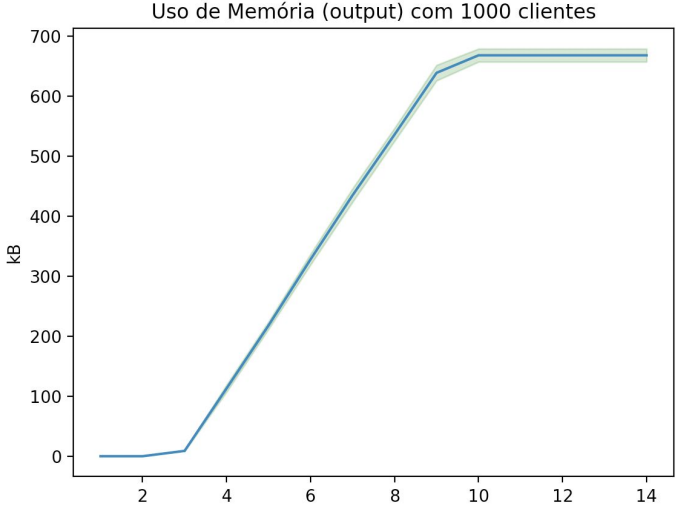




Teste com 1000 clientes







Análise dos resultados

- O uso de CPU tem grandes disparos ao conectarmos vários clientes. Existe também uma grande variação desse uso em testes de "estresse" como o de 1000 clientes conectando.
- O platô que temos ao final dos testes compõe somente os subscribers ainda conectados e os
- O servidor tem grande custo de memória e, dado um cenário de vários usuários ao longo de semanas, teríamos um problema sério de escala caso os pipes não sejam excluídos após a desconexão.
- O uso de CPU não parece ser uma grande barreira para a escala como a memória é.

Análise dos resultados

- O servidor aparenta aceitar muitas conexões por segundo sem problemas, isso pode talvez ser atribuído à máquina usada nos testes ou a velocidade da linguagem C no processamento. A divisão em dois processos, um de aceite de conexões e outro que lida com as já existentes, também tem papel fundamental na velocidade de resposta.