MAC0352 - EP1

Nome: David de Barros Tadokoro
NºUSP: 10300507
Professor: Daniel Macedo Batista
Abril de 2022

Visão Geral

- Este EP1 consistia em implementarmos um broker MQTT simplificado que, de forma geral, apenas aceitasse conexões e desconexões de clientes e também recebesse e encaminhasse mensagens;
- Muitas decisões de projeto foram tomadas em virtude desta solução simplificada, como a implementação de apenas os 'MQTT Control Packet Types' (a partir de agora vamos nos referir a eles como comandos/pacotes MQTT) que eram essenciais para essas funcionalidades;
- Esta apresentação contém uma explicação em alto nível das implementações desses comandos, dissertação sobre experimentos de performance realizados com o broker e comentários adicionais;
- O código fonte do broker (arquivo brokerMQTT.c, que deve ser encontrado no mesmo diretório desta apresentação) contém comentários que explicam ponto a ponto a implementação num nível mais técnico;
- Nosso broker oferece QoS = 0, como é o padrão dos clientes `mosquitto`.

Processo comum a todos os MQTT Control Packets implementados

- Todos os comandos MQTT implementados possuem uma etapa em comum, uma vez que todos estes possuem um header fixo ('Fixed Header'):
  - 1º byte: os 4 bits mais significativos enumeram os 14 comandos MQTT (os comandos de número 0 e 15 são reservados). Os 4 bits menos significativos são flags específicas a cada comando.
  - 2º - 5º byte: codificam o tamanho, em bytes, do resto do pacote ('Remaining Length'). Esse campo pode ter de 1 a 4 bytes de comprimento, a depender do tamanho do pacote.
  - Obs.: Os 4 bits menos significativos do 1º byte foram desconsiderados, pois, nos comandos implementados, essas flags se referiam a serviços não previstos na implementação simples que foi proposta.

Processo comum a todos os MQTT Control Packets implementados

- Após a conexão TCP ter sido estabelecida com um cliente, o processo filho do broker que cuida deste cliente (exclusivamente) entra em um loop infinito que processa um comando MQTT por iteração, no seguinte esquema:
  - 1º) Lê o primeiro byte do pacote, faz um shift de 4 bits para a direita para extrair o código do comando que é guardado numa variável;
  - 2º) Lê o segundo byte do pacote e decodifica o tamanho restante do pacote. Se necessário lê os próximos bytes (mais 3 no máximo) para determinar o tamanho restante. Este valor também é armazenado em uma variável;
  - 3º) Usando a variável do primeiro passo, o processo entra em um switch-case que o direciona ao tratamento do comando correspondente (próximos slides).

Variable Header

- Além do header fixo, cada pacote tem um header variável que depende do tipo do pacote;
- Alguns comandos implementados utilizam parte deste header para sinalizar serviços não considerados, pois, como dito antes, saem do escopo das funcionalidades essenciais, como QoS > 0. Assim, alguns bytes foram desconsiderados, já que não influenciam na implementação proposta (nos próximos slides haverá um caso em que isto não é tão verdade).

MQTT Control Packets implementados

`CONNECT` (1):

- O pacote `CONNECT` é o primeiro a ser enviado (da camada de aplicação) em uma conversa entre o broker e o cliente. Este pacote é enviado pelo cliente para o servidor;
- Nosso broker, primeiro lê os 7 bytes do header variável que indicam, de forma simples, a versão do protocolo MQTT ('Protocol Length', 'Protocol Name' e 'Protocol Level'). Na RFC do MQTT 3.1.1 é dito que essa sequência de bytes deve ser (em hexadecimal)

  `x00 x04 x4d x51 x54 x54 x04,`

  que se referem a:

  - 'Protocol Length' = 4 (`x00 x04`)
  - 'Protocol Name' = MQTT (`x4d x51 x54 x54`)
  - 'Protocol Level' = 4 (`x04`)

- Esses bytes são checados pelo nosso broker para garantir que o cliente está utilizando a mesma versão do MQTT que nosso broker. Se não for, fecha a conexão.

MQTT Control Packets implementados

`CONNECT` (1):

- O próximo byte se refere às flags de conexão. Analisando o Wireshark numa sessão "básica" do broker e clientes `mosquitto`, percebi que estas flags não são setadas por default, apenas a sessão limpa ('Clean Session'), o que já é considerado pelo nosso broker por padrão;
- Assim, este byte foi desconsiderado e apenas lido (mas não usado). Isto vale também pro payload do pacote, já que, usando o Wireshark, pude ver que este se refere a dados sobre as flags que são desativadas por padrão;
- Os próximos dois bytes após o byte de flags representam o tempo da conexão ('Keep Alive'). Estes bytes também foram lidos, porém desconsiderados, pois, por padrão o broker deve fechar a conexão após uma vez e meia do tempo anunciado, mas é responsabilidade do cliente enviar um pacote `PINGREQ` (não implementado) para manter a conexão ativa. Quando nosso broker recebe um pacote não implementado, este apaga o pipe do subscriber (se for um) e fecha a conexão. Como uma decisão de projeto, pela ótica da comunicação, isto não afeta a funcionalidade, porém não é ideal (parte do header desconsiderada mencionada em um slide anterior);
- Por fim, tornamos verdadeiro um booleano que guarda o estado conectado/desconectado do cliente.

MQTT Control Packets implementados

CONNACK (2):

- Após receber e processar o pacote `CONNECT` de um cliente, o broker (no mesmo case) deve responder com um pacote `CONNACK`;
- Por padrão (a partir do Wireshark) esse pacote em resposta ao primeiro pacote sempre segue a forma (em hexadecimal)

  `x20 x02 x00 x00`,

  em que o último byte indica o código de retorno 'Connection Accepted';

- A implementação deste comando simplesmente prepara um buffer com estes bytes e os envia para o cliente.

MQTT Control Packets implementados

PUBLISH (3):

- O broker primeiro verifica se o cliente já está conectado, isto é, se já enviou um CONNECT e recebeu um CONNACK. Senão, a conexão é fechada. Na realidade, pela RFC, o cliente pode começar a enviar outros pacotes em seguida do envio do CONNECT, então nosso broker seta o estado de conectado após o recebimento e processamento do CONNECT;
- O comando PUBLISH é enviado por um cliente (publisher) para o broker, mas também do broker para um cliente (subscriber);
- Lemos os dois primeiros bytes que se referem ao 'Topic Length' e guardamos seu valor numa variável;
- Usando a variável do 'Topic Length' lemos esse número de bytes para obtermos o tópico ('Topic Filter');
- Subtraímos da variável que guarda o 'Remaining Length' o valor 2 e 'Topic Length' (número de bytes do pacotes lidos sem contar o header fixo).

MQTT Control Packets implementados

`PUBLISH` (3):

- Tendo o nome do tópico, preparamos uma string com o path `/tmp/<nome_topico>`;
- Lemos a mensagem de fato publicada pelo publisher (payload), que tem 'Remaining Length' bytes (número calculado anteriormente) e a guardamos num buffer;
- Utilizando a string com o path do diretório que representa o tópico, iteramos pelos arquivos (que são pipes referentes aos inscritos no tópico) os abrimos e escrevemos a mensagem enviada do buffer. Também fechamos cada um dos pipes e o diretório;
- Não há resposta padrão para o publisher pelo broker em QoS = 0. No entanto, a escrita da mensagem nos pipe engloba parte do processo do envio do pacote `PUBLISH` do broker para os subscribers do tópico.

MQTT Control Packets implementados

`SUBSCRIBE` (8):

- Assim como no `PUBLISH`, nosso broker verifica se o cliente enviando a mensagem está no estado conectado. Senão, encerra a conexão;
- Este pacote é enviado de um cliente (subscriber) para o broker;
- Os primeiros dois bytes se referem ao 'Message Identifier' e são guardados em uma variável;
- Os próximos bytes se referem ao 'Topic Length' e ao 'Topic Filter' (que no nosso caso é o próprio tópico, já que não são aceitos wildcards). Assim como no `PUBLISH`, os lemos, extraímos suas informações e as guardamos em variáveis;
- Usando o tópico, criamos seu diretório referente `/tmp/<nome_topico>` (se já não tiver sido criado);

MQTT Control Packets implementados

SUBSCRIBE (8):

- Guardamos em um buffer a string `/tmp/<nome_topico>/pipe.XXXXXX`;
- Essa string é usada pela função `mkstemp()` para criar um arquivo com nome único, mas também altera a string para o path desse arquivo;
- Como só queremos o nome único do arquivo, o apagamos e usamos o `mkfifo()` para criar um pipe de mesmo nome e localização que o arquivo deletado (note como esta string "localiza" o pipe e permite que esse seja aberto/lido no futuro);
- Tornamos verdadeiro o booleano que diz que o cliente é um subscriber (necessário para apagar seu pipe quando desconectado);
- Aqui vale dizer que consideramos que cada subscriber só se inscreve em um tópico, e, portanto, cada subscriber é associado a um único pipe;

MQTT Control Packets implementados

`SUBSCRIBE` (8):

- Nesse momento, o broker cria outro processo filho (lembre-se que este será o processo filho do processo filho do processo inicial do broker) que ficará em um loop infinito esperando escreverem no pipe, isto é, uma publicação no tópico que o publisher está inscrito. Enquanto isso, o processo pai (processo filho do processo inicial) segue para o envio do pacote `SUBACK` e consequente tratamento de outros pacotes;
- De forma geral, este filho, ao receber uma mensagem do pipe, prepara o mesmo pacote `PUBLISH` que foi enviado pelo publisher (em outra conexão) usando as variáveis do tamanho restante, tamanho do tópico e nome do tópico e o envia para o subscriber.

MQTT Control Packets implementados

SUBACK (9):

- Similar ao CONNACK, o pacote SUBACK é enviado (no mesmo case) para o subscriber após o recebimento e processamento do pacote SUBSCRIBE;
- O padrão do pacote segue a forma (em hexadecimal)

  x90 x03 x<MSB_msgid> x<LSB_msgid> x00

  em que <MSB_msgid> e <LSB_msgid> foram os dois primeiros bytes lidos do pacote SUBSCRIBE que foram guardados em uma variável;

- O último byte refere-se ao código de retorno para o QoS = 0.
- A implementação deste comando simplesmente prepara um buffer com estes bytes e os envia para o subscriber.

MQTT Control Packets implementados

DISCONNECT (14):

- Este pacote é enviado pelos clientes ao broker, para que haja a desconexão;
- Quando este pacote é recebido, o broker somente remove o pipe referente ao cliente, se este for um subscriber, e fecha a conexão;
- Obs.: Este mesmo esquema ocorre quando um pacote recebido pelo broker não corresponde ao código 1, 3, 8 ou 14, isto é quando recebe um pacote não implementado (case default).

Experimentos

Ambiente:

- Para medir o desempenho do broker, em questão de uso de CPU e de rede, realizamos experimentos em três cenários:
  - Apenas o broker rodando;
  - O broker mais 100 clientes rodando;
  - O broker e mais 1000 clientes rodando.
- Estes experimentos foram feitos colocando o broker em um container e criando uma rede virtual usando o `docker-compose`;
- Os clientes rodavam na minha máquina local enquanto o broker rodava "fora". Na prática, estão na mesma máquina, mas logicamente não, pois da visão do docker estes processos clientes não residem no mesmo sistema (há uma postagem no fórum, em que o professor confirma a validade desta abordagem como sendo "dois computadores diferentes").

Experimentos

Método :

- Para as medições da porcentagem da CPU (medida para desempenho de CPU) e total de bytes recebidos/enviados pelo o broker (medida para desempenho de rede), foi usado o `docker stats`. O `docker stats` mostra informações diversas sobre o container, incluindo essas duas medidas, chamadas, respectivamente, de `CPU %` e `NET I/O`;
- No caso da CPU, como somente no momento em que há processamento de pacotes o broker utiliza da CPU (de forma não desprezível), não faz sentido capturar o seu uso apenas no final de uma bateria de experimento. Assim, seus valores foram coletados periodicamente ao longo da execução dos clientes;
- Por outro lado, no caso da rede, faz sentido considerarmos o tráfego total após o fim da bateria, uma vez que este número sumariza os dados enviados/recebidos pelo broker;

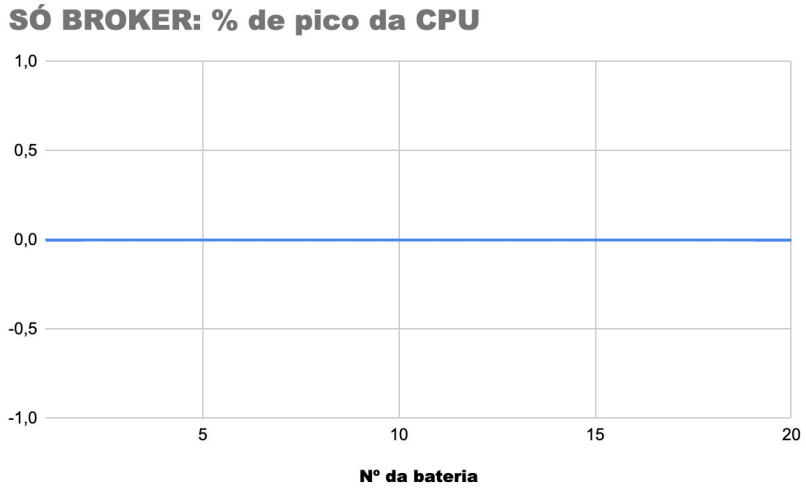
Experimentos

Método:

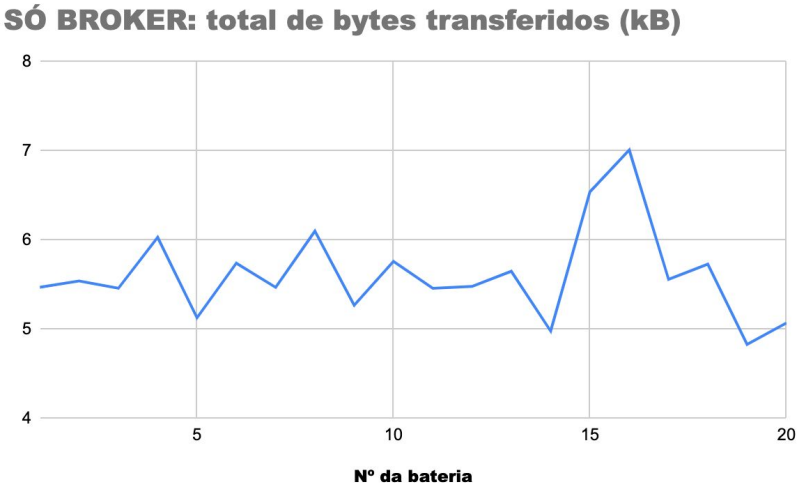
- No caso da CPU, mesmo com medidas periódicas, a maioria acusavam 0% de uso. Porém haviam certos picos (consistentes para cada cenário) de forma que adaptei a medida de desempenho da CPU para porcentagem de pico da CPU;
- A coleta de dados foi feita com 20 baterias de testes para cada um dos três cenários;
- Cada bateria consistia em 30 execuções do `docker stats` a partir do momento que o script de 100 ou 1000 clientes começava (ou a partir do momento que o broker entra no ar);
- Os scripts de 100 ou 1000 clientes funcionavam da seguinte maneira:
  - Inscreviam metade dos clientes com o `mosquitto_sub`, com um ¼ dos clientes inscritos em um tópico e o outro ¼ dos clientes inscritos em outro tópico;
  - Conectavam metade dos clientes com `mosquitto_pub`, com um ¼ dos clientes publicando em um dos tópicos e o outro ¼ dos clientes publicando no outro;
  - Desconectavam os subscribers com `killall mosquitto_sub` (já que estes estavam rodando em segundo plano). Note como não foi preciso mandar os publishers desconectarem, pois estes enviam um pacote `DISCONNECT` após publicarem a mensagem, por padrão;
  - Exemplo para 100 clientes: 25 inscritos no tópico `a`, 25 inscritos no tópico `b`, 25 publicando no tópico `a` e 25 publicando no tópico `b`.

Experimentos

Resultados (Só Broker) :



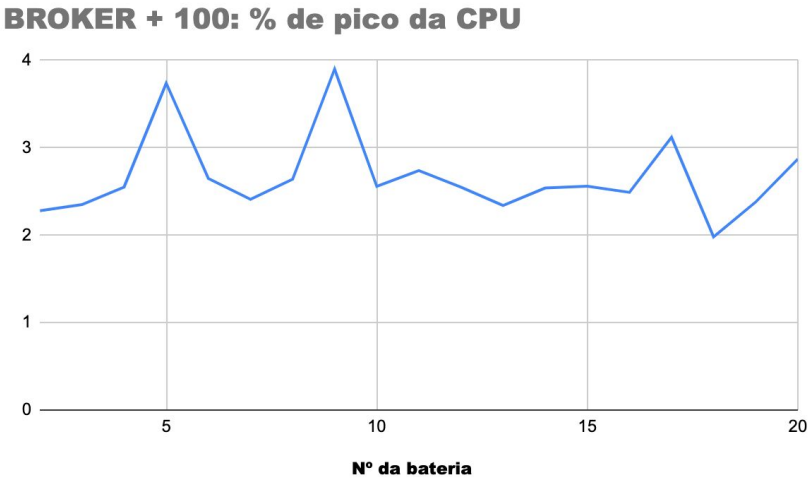
Média: 0,0 %
Desvio Padrão: 0,0 %



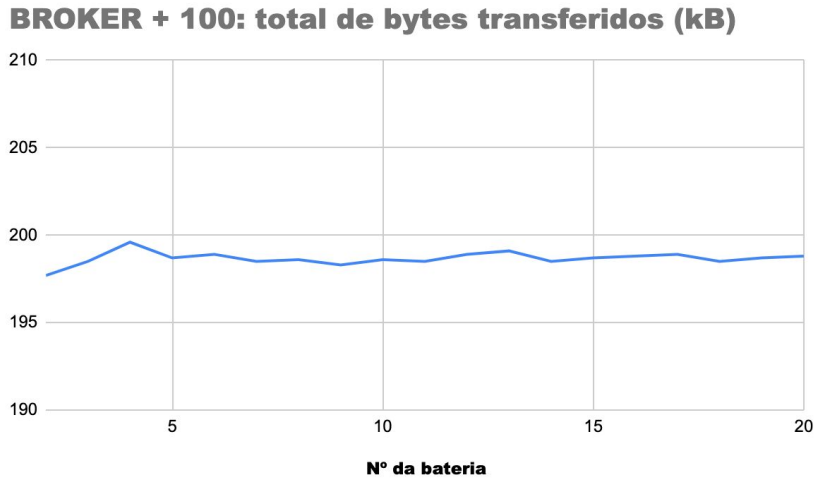
Média: 5,614 kB
Desvio Padrão: 0,5 kB

Experimentos

Resultados (broker + 100 clientes) :



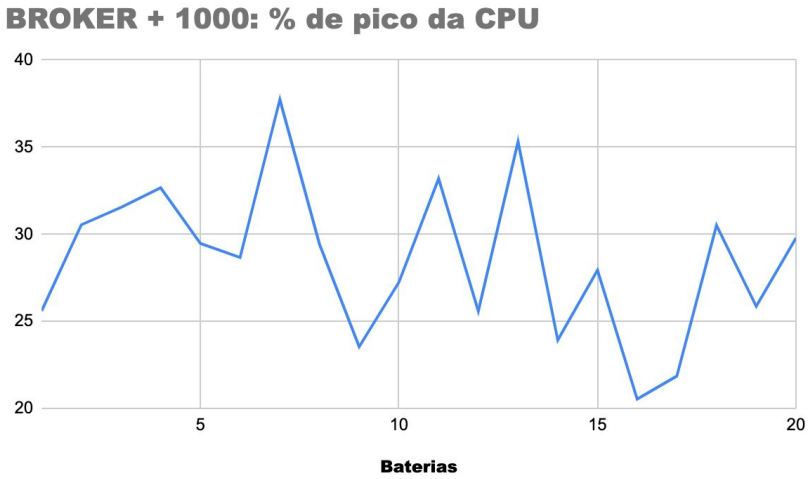
Média: 2,693 %
Desvio Padrão: 0,5 %



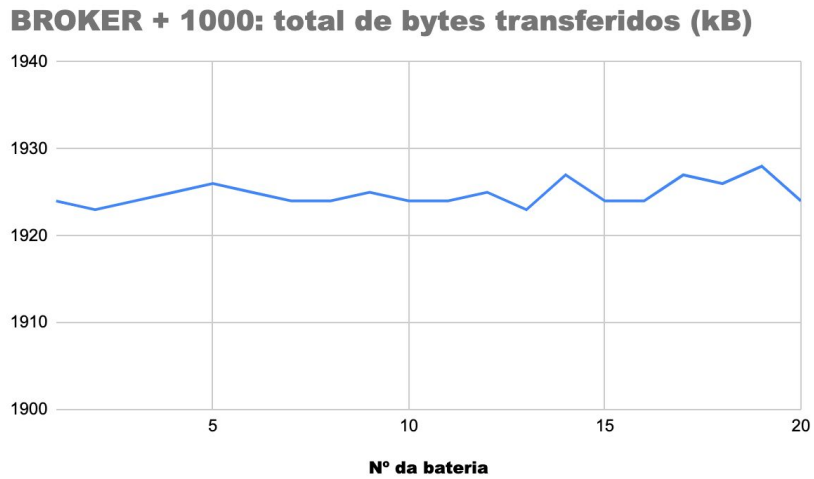
Média: 198,685 kB
Desvio Padrão: 0,6 kB

Experimentos

Resultados (broker + 1000 clientes) :



Média: 28,545 %
Desvio Padrão: 4,4 %



Média: 1924,8 kB
Desvio Padrão: 1,4 kB

Experimentos

Comentários :

- No cenário em que há somente o broker, o `docker stats` acusou 0,0 % uso da CPU em todas iterações. Isto era de certa forma esperado, pois o uso do tempo de CPU do broker sem conexões é praticamente zero (em dado momento ele fica "travado" esperando conexões). Curiosamente, houve tráfego de bytes chegando no broker (net input), e, apesar de ser um tráfego leve em torno de 5,6 kB, não esperava que houvesse uso de rede nesse cenário;
- No cenário em que há o broker e 100 clientes, vemos que há alguns momentos de pico não nulos, apesar de consumirem pouco a CPU, relativamente. Vemos que nesse caso o tráfego aumentou bastante em relação ao cenário só com broker (~40 vezes);
- No cenário em que há o broker e 1000 clientes, as porcentagem de pico chegaram a valores bem altos. O tráfego também aumentou bastante;

Experimentos

Comentários:

- Vale dizer que, comparando os cenários do broker com 100 e 1000 clientes, parece haver uma relação de linearidade entre o número de clientes e a porcentagem de pico da CPU, assim como entre o número de clientes e o total de bytes transferidos. No caso do total de bytes, isso faz bastante sentido, uma vez que a única diferença entre os scripts dos cenários é que temos uma ordem de grandeza entre as instâncias de clientes, ou seja, os tópicos e as mensagens são os mesmos. Assim, faz sentido o tamanho dos pacotes ser linear com o número de clientes (a flutuação a cada bateria pode ser justificada com pacotes de camadas inferiores que variam). Já a aparência linear da porcentagem de pico da CPU, não me parece tanto intuitiva;
- Importante mencionar como, mesmo não sendo ideal, a falta da implementação do `PINGREQ`, que faz um "refresh" na conexão, não fez falta, pois todas baterias demoraram menos que 1 minuto ('Keep Alive' padrão).
- A forma de registrar o desempenho da CPU não se mostrou tão acurada, pois conta com a ocasionalidade de observar o uso da CPU em instantes propícios. Mesmo considerando apenas a porcentagem de pico, não há como ter certeza se, de fato, esses valores modelam a realidade.