

---

---

# Broker MQTT

## EP1

**Autor:**

Thiago Guerrero, 11275297

**Professor:**

Daniel Macedo Batista

---

# Simplificações

Como não devemos nos preocupar com falhas ou autenticação, o processamento de alguns pacotes foi simplificado:

**CONNECT:** O conteúdo do variable header e do payload são ignorados.

**CONNACK:** O conteúdo do variable header é sempre zero.

**PUBLISH:** As flags são ignoradas. É assumido que o pacote não possui Packet Identifier no variable header.

**SUBSCRIBE:** O QoS é ignorado no payload. É assumido que só existirá um tópico por pacote.

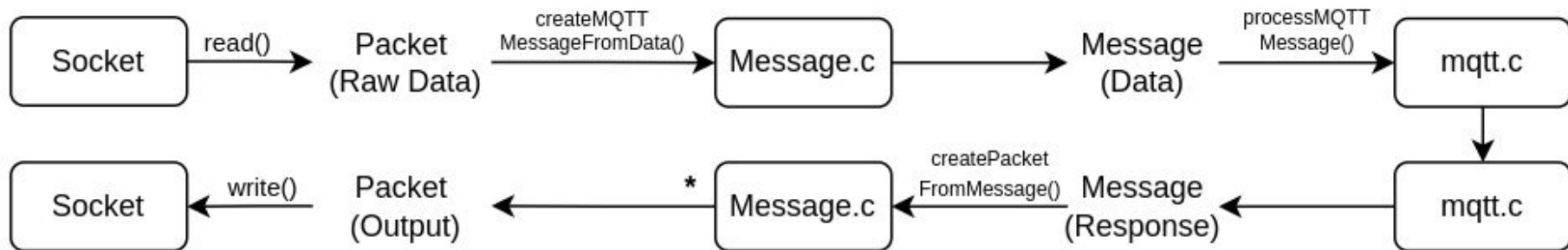
**SUBACK:** O Return Code é sempre zero.

---

Obs: Os conteúdos que não são ignorados não são validados.


# Fluxo dos dados

# Fluxo de dados




\* Foi criado um pacote de controle especial chamado **SPECIAL** que interrompe a execução do processo, quando desejado. Ou seja, o fluxo é interrompido aqui.

# Fluxo de dados



```
typedef struct {  
    int packetType;  
    int flags;  
    int remainingSize;  
    char * remaining;  
} Message;
```



```
typedef struct {  
    char * data;  
    int size;  
} Packet;
```

# Implementação

---

# Implementação

- Todo o gerenciamento do broker é realizado no arquivo `mqtt.c`
  - Os pacotes de controle CONNECT e PINGREQ são simplesmente respondidos com CONNACK e PINGRESP, respectivamente.
  - Os pacotes de controle PUBLISH e DISCONNECT são “respondidos” com o pacote de controle especial SPECIAL para informar que deve encerrar a conexão. Ou seja, após processar o pacote recebido, o processo encerra a sua execução sem enviar nada para o cliente.
-

---

# SUBSCRIBE

- É criado um pipe para cada pacote de SUBSCRIBE recebido.
  - Todos os pipes são armazenados através da seguinte estrutura:
    - `<TMP_FOLDER>/<TOPIC_NAME>/<getpid()>`
    - Onde TMP\_FOLDER está definido no arquivo config.h e TOPIC\_NAME é processado através de dados do pacote.
  - O processo é separado em duas threads.
-



---

# SUBSCRIBE

- **Primeira thread (main thread)**
    - Responsável por continuar a troca de mensagens com o cliente.
    - Responsável pelo envio do SUBACK para o cliente.
    - Responsável por responder os pings enviados pelo cliente.
    - Responsável por finalizar a outra thread após receber o pacote de controle DISCONNECT.
  - **Segunda thread (created thread)**
    - Responsável por ouvir mensagens do tópico e enviar para o cliente.
    - Após ouvir cada mensagem, fecha e reabre o tópico.
    - Com isso, a thread fica travada até que algum outro processo abra o tópico para escrever novamente.
    - Esta estratégia ajuda a sincronizar as escritas e leituras no pipe.
-

---

# PUBLISH

- Após identificar o tópico, escreve a mensagem em todos os pipes no diretório:
    - <TMP\_FOLDER>/<TOPIC\_NAME>/
    - Onde TMP\_FOLDER e TOPIC\_NAME são os mesmos definidos antes.
  - Caso o diretório não exista, ou seja, não existe nenhum SUBSCRIBE para aquele tópico, o processo não faz nada.
  - Após todo o envio de mensagens, o processo é encerrado.
-

# Experimentos

---

# Experimentos

- Para fins de experimentação, todos os processos do broker estarão em um contêiner do Docker.
  - Este contêiner estará rodando em uma rede virtual gerenciada pelo Docker.
  - Os clientes estarão rodando no terminal da minha máquina.
  - Para ocorrer a comunicação Cliente/Servidor, a porta 1883 do contêiner estará vinculada com a porta 1883 do localhost.
-

---

# 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)

# Mudança nos cenários

## Os experimentos serão realizados nos três cenários:

1. apenas com o broker, sem nenhum cliente conectado.
2. com o broker e com cem clientes conectados recebendo mensagens simultaneamente de cem clientes sequenciais.
3. Mesmo cenário do item anterior, mas com mil clientes.

- Todos os processos estarão rodando na mesma máquina, ou seja, todas as conexões feitas com o broker, todos os clientes de subscribe e todos os clientes de publish.
  - Por conta disso, experimentos em que N clientes estão ouvindo e N clientes estão publicando ao mesmo tempo faz com que os clientes de publish sofram com *starvation*, para N grande.
    - Isso ocorre, pois, o S.O pode escolher entre todas as conexões em aberto ou todos os clientes que estão ouvindo em vez de escolher um cliente para publicar a mensagem, fazendo com que passe o tempo de *timeout*.
  - Para solucionar este problema sem afetar a relevância dos experimentos, foi considerado que apenas um cliente irá publicar por vez de forma bloqueante para outros N clientes escritos no mesmo tópico.
-

---

# 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 paralelo, enquanto os clientes realizam a sua comunicação.
    - 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 100 vezes.
  - Será feito um intervalo de confiança com nível de confiança de 95%.
-



---

# Scripts

- Para cada cenário, o seguinte algoritmo é executado:
    - A rede virtual e o contêiner do Docker são levantados.
    - O script que simula os clientes é executado em background.
    - O script que coleta as medições é executado em primeiro plano.
    - Após a simulação dos clientes ser finalizada, a coleta das medições é finalizada e a rede virtual e o contêiner do Docker são derrubados.
  - É importante notar que não são usados contêineres exclusivos para cada processo, ou seja, todos os processos rodam no mesmo contêiner.
    - Isso vai contra as boas práticas do uso do Docker, o ideal é cada processo ter os seus próprios recursos computacionais.
    - Porém, esta separação iria trazer poucos benefícios em relação ao trade off Complexidade x Resultados.
-

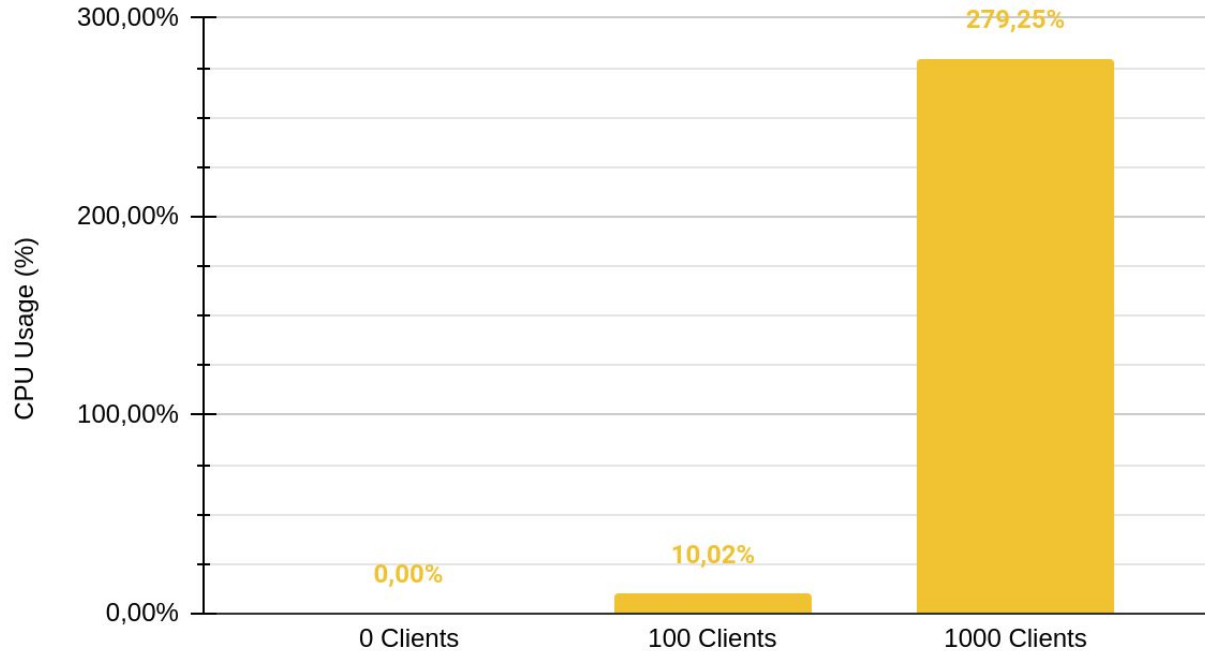
---

# Resultados

---

# Uso de CPU

CPU Usage Per Scenario Size



**Intervalos de confiança:**

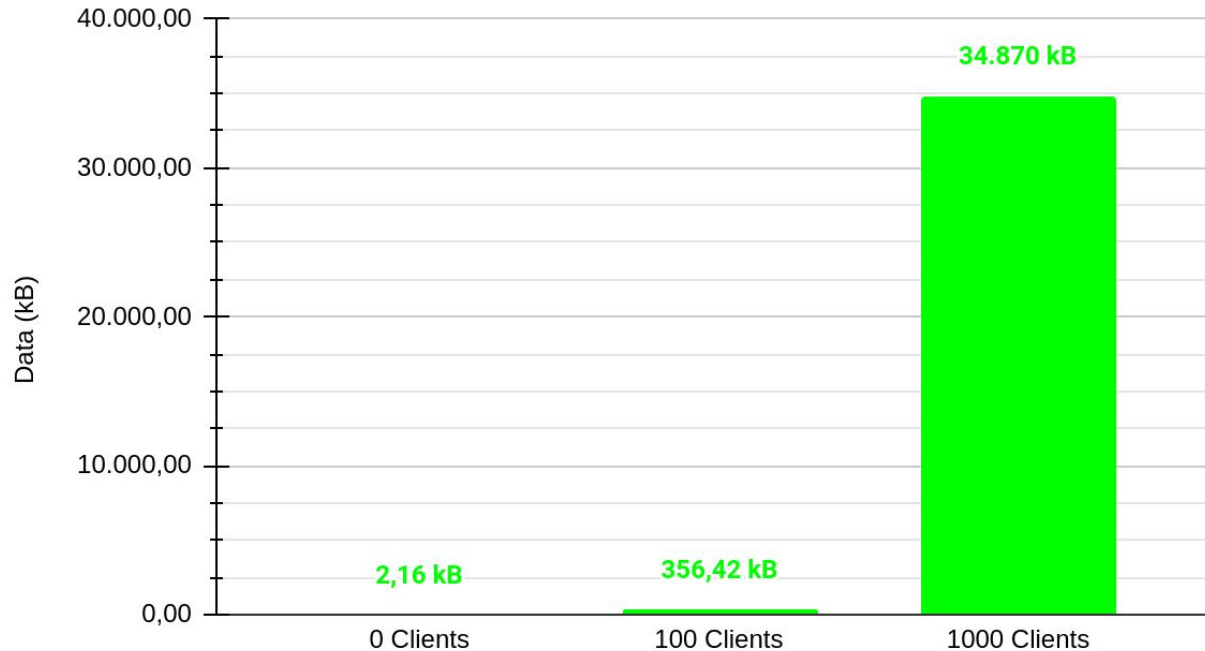
[0; 0]

[0; 31,82%]

[169,13%; 389,37%]

# Uso de Rede: Recebimento de pacotes

Network Input Data Per Scenario Size



**Intervalos de confiança:**

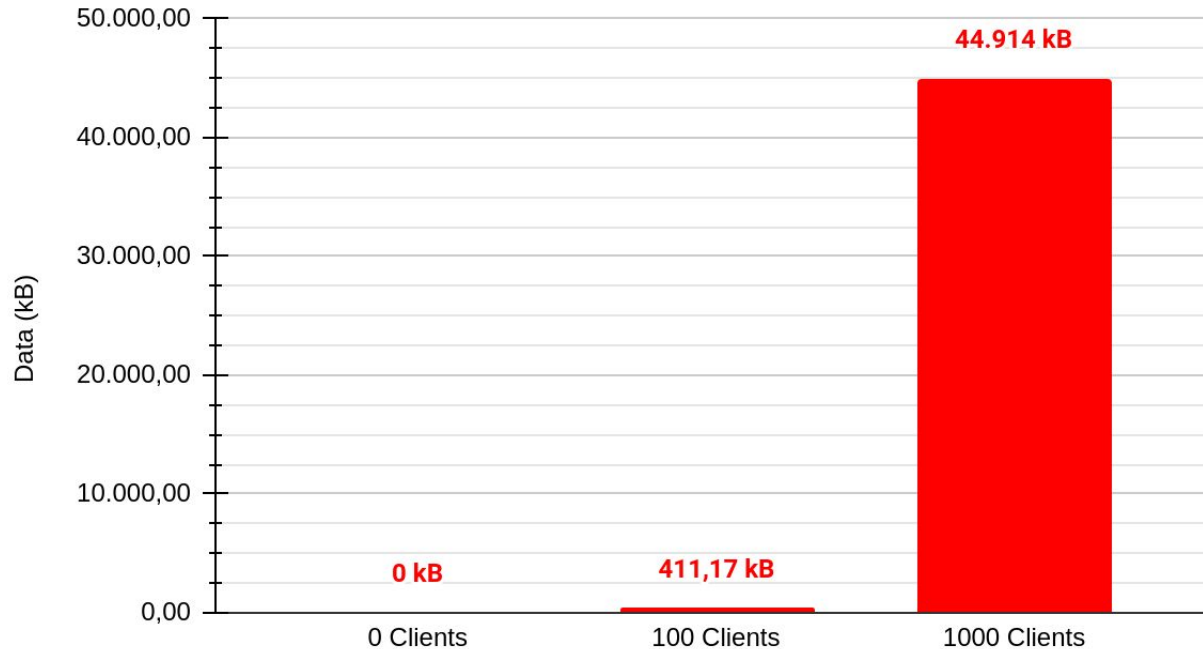
[0,65 kB; 3,66 kB]

[0; 937,46 kB]

[24.871 kB; 44.869 kB]

# Uso de Rede: Envio de pacotes

Network Output Data Per Scenario Size



**Intervalos de confiança:**

[0; 0]

[0; 1.117,1 kB]

[31.745 kB, 58.083 kB]

---

# Resultados

- Embora o aumento do uso de CPU não tenha sido exatamente linear, é possível observar que esse aumento tem uma tendência linear.
  - Entretanto, isso está longe de ser verdade para o uso de rede.
    - O que evidencia que a quantidade de pacotes trocados usando o protocolo MQTT não cresce de forma linear ao número de clientes.
  - Ainda nesse ponto, é notório o impacto no uso dos recursos de uma rede com o aumento do número de usuários que a usam.
    - Isso mostra que é sempre muito importante levar esses números em conta na hora de arquitetar uma solução.
-

---

## Resultados

- Como o broker é responsável por fazer o broadcast das mensagens para os clientes inscritos em algum tópico, é notório que o número de bytes enviado foi maior que o número recebido.
  - Mesmo com zero clientes conectados, ainda houve um pequeno recebimento de mensagens.
    - Isso se deve a algumas trocas de mensagem que acontecem visando o funcionamento da rede virtual do Docker.
  - Por algum motivo os intervalos de confiança do uso de rede para o cenário com 100 clientes ficou muito grande.
    - Não soube identificar se há um motivo.
-