# Broker MQTT
## EP1 - MAC0352 - 2022

Lorenzo Bertin Salvador - 11795356

IME-USP

# Introdução:

- Código baseado no servidor de eco+pipe disponibilizado no e-disciplinas. Para o compartilhamento de mensagens existir, a comunicação entre os processos também ocorre por meio dos pipes e file descriptors.
- Broker permite conexão de um ou mais clientes simultâneos.
- Inscrição de um cliente em um tópico.
- Publicação de um cliente em um tópico.
- Desconexão de um cliente.

# Terminologia:

- Cliente sub: cliente que se inscreve em tópicos
- Cliente pub: cliente que publica em tópicos
- Mensagem: na maioria dos casos se refere as informações trocadas entre cliente/servidor, incluindo binários e textos. Também se refere ao texto enviado por um cliente pub para um cliente sub. Em geral, não será difícil diferenciar suas definições a depender do contexto.

# Ideia da implementação:

- Pipes para a comunicação entre processos.
- Criação de um arquivo de registro para armazenar os tópicos que cada cliente está inscrito.
- Quando um cliente se inscreve em um tópico, ele é adicionado ao arquivo de registro e tem seu pipe aberto para leitura pelo servidor.
- Ao receber a publicação de uma mensagem, o servidor percorre o arquivo de registro e, para cada cliente inscrito no tópico em questão, ele abre para escrita o pipe do cliente e escreve a mensagem nele.
- Para os clientes que estão inscritos em algum tópico, o servidor precisa ler do file descriptor que recebe os comandos (connfd) e do que recebe as mensagens (client.fd).

# Estrutura do cliente:

- Foi criada uma estrutura para representar cada cliente, que contém:
  - int number: número do cliente (determinado pelo próprio broker - funciona como identificador)
  - char* Id: nome do cliente (determinado pelo próprio cliente)
  - int qtd_topics: quantidade de tópicos em que um cliente está inscrito (utilizados em clientes sub)
  - int fd: file descriptor para o recebimento de mensagens (utilizado em clientes sub)
- O number é assinalado para um cliente assim que a conexão é aceita pelo broker.
- Todas as outras informações são inicializadas em 0 (ou a string vazia) e são alteradas ao longo do programa.

# Implementação: pipes e arquivo de registro

- Assim que o servidor é iniciado, ele cria um arquivo para registrar as inscrições dos clientes. O caminho do arquivo respeita o formato /tmp/broker11795356register-XXXXXX, onde cada X é um caractere aleatório.
- Exmplo do formato do arquivo de registro:
  - “29/3;1,mac0352;0,daniel;0,fatemeh;”
    - 29 = caracteres após o primeiro ';'
    - 3 = quantidade de tópicos/clientes existentes
    - 1,mac0352 = cliente 1 está inscrito no tópico mac0352
- Além disso, a cada nova conexão, cria-se um arquivo FIFO para suportar a comunicação entre processos (clientes) e o servidor. O caminho do arquivo respeita o formato /tmp/broker11795356-{num}XXXXXX, onde {num} é o número do cliente e cada X é um caractere aleatório.
- Como cada cliente recebe um arquivo e o broker não é muito complexo, ele não reutiliza os arquivos de quem já se desconectou. Dessa maneira, na macro #define MAXCLIENTS, são definidos quantos clientes podem passar pelo servidor (não simultâneos). Esse valor pode ser aumentado e por padrão seu valor é 30.000.

# Implementação: Recebimento dos pacotes (mensagens)

- A função que interpreta cada pacote recebe os bytes e, ao analisar o primeiro byte, ela decide como tratar cada mensagem. O primeiro byte de uma mensagem é chamado de MQTT Control Packet type e é definido pela documentação do protocolo.
- As possíveis mensagens recebidas são: CONNECT, PUBLISH, SUBSCRIBE, PINGREQ e DISCONNECT.
- O servidor pode enviar as seguintes mensagens aos clientes: CONNACK, PUBLISH, SUBACK e PINGRESP.

# Implementação: Recebimento dos pacotes (mensagens)

- Ao receber cada pacote, antes de decidir qual comando está presente no primeiro byte, o servidor transforma a lista de bytes do pacote em uma lista de bits, convertendo cada byte em uma lista de 8 inteiros, onde cada inteiro representa um bit (0 ou 1). Isso é feito para que seja possível interpretar cada flag que pode estar presente em um byte.
- Exemplo: bytes = {32,2,3,0}

  bytes = {{0,0,1,0,0,0,0,0},{0,0,0,0,0,0,1,0},{0,0,0,0,0,0,1,1}}

- Na prática, para esse broker mais básico, veremos que nenhuma flag vai alterar o comportamento do servidor pois ele as ignora, porém, caso o broker seja expandido, a interpretação das flags ocorre facilmente dessa maneira.

# Implementação: CONNECT

- O comando CONNECT inicia a conexão entre um cliente e o servidor.
- O servidor sempre responde um CONNECT com um CONNACK, estabelecendo assim a conexão.
- Em geral, no CONNECT algumas flags são enviadas para descrever como a interação deve ocorrer, porém, no caso desse broker mais básico, esses casos não são tratados.
- No CONNECT, o cliente envia seu Id para o servidor, que armazena essa string na estrutura cliente.Id.

# Implementação: SUBSCRIBE

- O SUBSCRIBE é o segundo comando enviado por um cliente sub, depois do CONNECT. Nele, está codificado o tópico que o cliente está se inscrevendo.
- O servidor responde o SUBSCRIBE com um  SUBACK, para confirmar que a inscrição foi feita. A cada SUBSCRIBE de um cliente, a estrutura cliente.qtd_topics é incrementada em uma unidade.
- Após o CONNECT, o servidor adiciona o número do cliente e o tópico no arquivo de registro e abre o pipe do cliente para leitura, onde serão escritas as mensagens destinadas a esse cliente.
- Nesse momento, a estrutura do cliente também armazena o inteiro que representa seu file descriptor (para abrir seu pipe) em cliente.fd.

# Implementação: cliente sub monitora dois file descriptors.

- É importante ressaltar que o servidor não consegue monitorar dois file descriptors ao mesmo tempo.  Dessa maneira, quando o cliente sub se inscreve em um tópico, o servidor passa a ter que ler tanto o file descriptor que recebe os comandos, quanto o file descriptor do cliente, que recebe as mensagens para seu tópico.
- Por esse motivo, para os clientes subs é utilizada duas vezes a função input_timeout(filedes,seconds), que recebe os files descriptors e checa se há conteúdo neles a cada intervalo de seconds. Por padrão, o valor de seconds é 3, porém isso pode ser alterado na macro #define COOLDOWN.
- Se o file descriptor das mensagens possui informação, o servidor a lê e exibe na tela do cliente. Se o file descriptor dos comandos (connfd) possui informação (em geral um PINGREQ), o servidor processa esse comando e volta a esperar pelas mensagens.

# Implementação: PUBLISH (receber)

- O comando PUBLISH é o segundo comando de um cliente pub. Nele, o cliente envia a mensagem e o tópico em que ela será publicada.
- No caso desse broker, o servidor não devolve nada confirmando a publicação para o cliente, apenas envia um outro PUBLISH para os clientes sub (explicado no próximo slide).
- Muitas vezes, dentro do próprio pacote de PUBLISH, o cliente envia o sinal de DISCONNECT. (que também será explicado em breve).

# Implementação: PUBLISH (enviar)

- Quando recebe um PUBLISH do cliente pub, o servidor  abre o arquivo de registro para leitura e, para cada cliente sub que possui inscrição no tópico a ser publicado, o servidor escreve a mensagem em seu pipe usando seu file descriptor.
- Se não houver nenhum cliente inscrito no tópico ninguém receberá a mensagem.

# Implementação: PINGREQ

- O PINGREQ é um comando enviado pelo cliente que pretende apenas saber se o servidor ainda está ativo. Por isso, o servidor apenas responde um PINGRESP quando recebe esse comando.
- O PINGRESP é um comando de 2 bytes, sendo o primeiro o identificador do comando e o segundo sendo o byte 0.
- Se a resposta não ocorrer em um determinado intervalo de tempo, o cliente reinicia a conexão enviando um CONNECT novamente.

# Implementação: DISCONNECT

- Quando um cliente envia um DISCONNECT ao servidor, o servidor altera o número do cliente para -1 a fim de indicar ao laço que monitora os files descriptors que o cliente deseja se desconectar. Então, ele sai desse loop, termina de liberar as variáveis alocadas dinamicamente, fecha o pipe, o deleta do diretório /tmp e encerra o processo.

# Makefile e arquivos temporários:

- Por criar alguns arquivos no diretório temporário da máquina do servidor (/tmp), é preciso ter cuidado para remover todos os arquivos no fim da conexão.
- O problema é que o servidor possui um laço infinito e, portanto, apenas terminado com um CTRL+C (ou kill). Por esse motivo, o arquivo de registro não é deletado do diretório temporário.
- Além disso, quando um cliente sub não possui a opção -C (no mosquitto_sub), que o disconecta automaticamente após certo número de mensagens recebidas, seu término também só ocorre com um CTRL+C (ou kill), o que também impede a remoção do arquivo do cliente do /tmp.
- Para resolver isso, no makefile há a opção “make clean”, que remove tudo relacionado aos arquivos temporários do servidor do diretório /tmp.

# Performance - método e ambiente

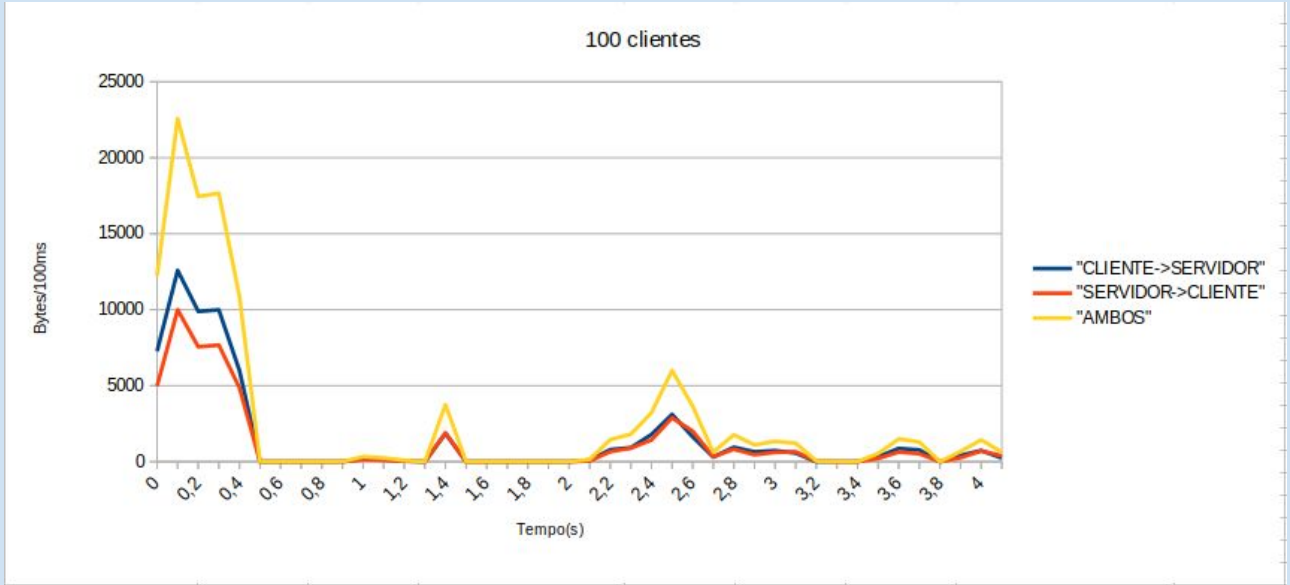
- Para os testes de 100 clientes, foram utilizados 99 clientes subs se inscrevendo no mesmo tópico e por último 1 cliente pub publicando neste tópico.
- Para os testes de 1000 clientes foi utilizado o mesmo método, porém com 999 subs e 1 pub.
- Foi considerada a conexão, o recebimento de mensagem e desconexão dos clientes subs (apenas é feito kill nos processos de clientes sub, não são todos que enviam a mensagem DISCONNECT explicitamente).
- O código do broker está rodando na máquina real UBUNTU com 8GB de RAM e os clientes se conectam por uma máquina virtual XUBUNTU com 2GB de RAM.
- CPU: Intel(R) Core(TM) i5-4460  CPU @ 3.20GHz
- A conexão da máquina real e a internet se dá por um cabo ethernet.
- A porta utilizada nos testes é a 12345.

# Performance: Rede e CPU - Apenas o broker

- Quando o broker está rodando, mas nenhum cliente se conecta, não há nenhuma transferência de dados ocorrendo pela rede.
- Sem conexão de clientes, o uso da CPU também fica sempre em 0%.

# Performance: Rede - 100 clientes

Média de bytes transmitidos: 113569 bytes




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| "CLIENTE->SERVIDOR" | | | "SERVIDOR->CLIENTE" | | | | "AMBOS" | | |
| | | | | | | | | | |
| MÉDIA (Bytes/100ms): | 267,675 | | MÉDIA (Bytes/100ms): | 1208,57619 | | | MÉDIA (Bytes/100ms): | 2704,02381 | |
| DESVIO PADRÃO: | 3027,21113 | | DESVIO PADRÃO: | 2344,29672 | | | DESVIO PADRÃO: | 5366,81784 | |

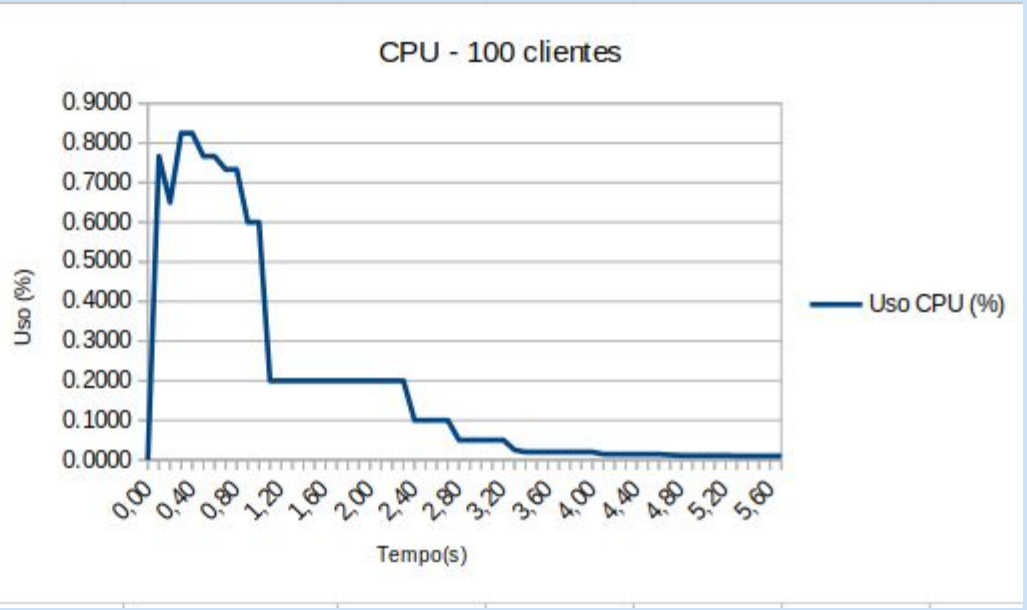
# Performance: Rede - 1000 clientes

Média de bytes transmitidos:  1133708 bytes


| | "CLIENTE->SERVIDOR" | "SERVIDOR->CLIENTE" | "AMBOS" |
|---|---|---|---|
| MÉDIA (BYTES/100ms) | 5780,51153015873 | 4672,35324444444 | 10452,8647746032 |
| DESVIO PADRAO | 2636,08201792662 | 1915,99446149388 | 4525,60573405798 |

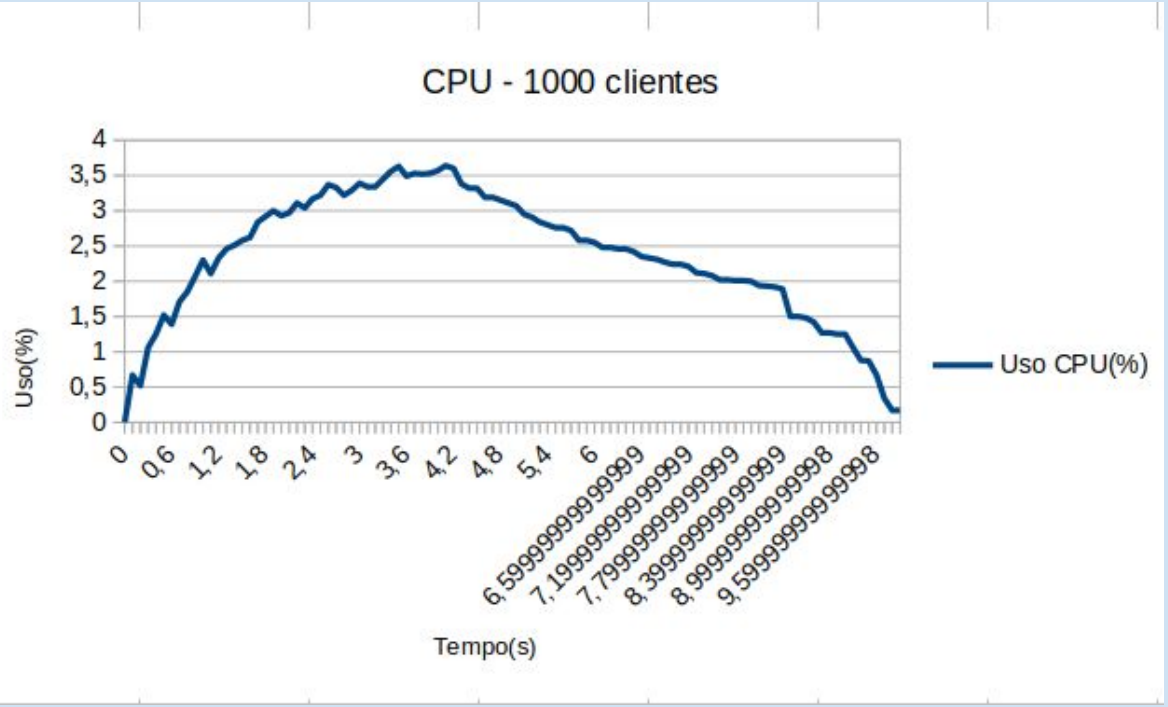
# Performance: CPU - 100 clientes

| | |
|---|---|
| MÉDIA | 0.19 |
| DESVIO PADRÃO | 0.26 |



# Performance: CPU - 1000 clientes

| | |
|---|---|
| MÉDIA | 2,3553 |
| DESVIO PADRÃO | 0,910827635 |



# Conclusão:

- Mesmo com 1000 clientes, o baixo uso da CPU mostra justamente o principal objetivo do protocolo MQTT, que é a simplicidade na transmissão de informação entre o broker e os clientes.
- Apesar de ser baseado TCP, a ideia é que o overhead de bytes na transmissão não seja alto e por isso é possível utilizar o protocolo com aparelhos com pouca capacidade de processamento ou pouca bateria.
- Além disso, é importante notar que o protocolo baseia-se em poucos comandos, todos essenciais para a existência de uma conexão publish/subscribe, o que auxilia na implementação do broker, dos clientes e na simplicidade da transmissão de dados.