Servidor AMQP

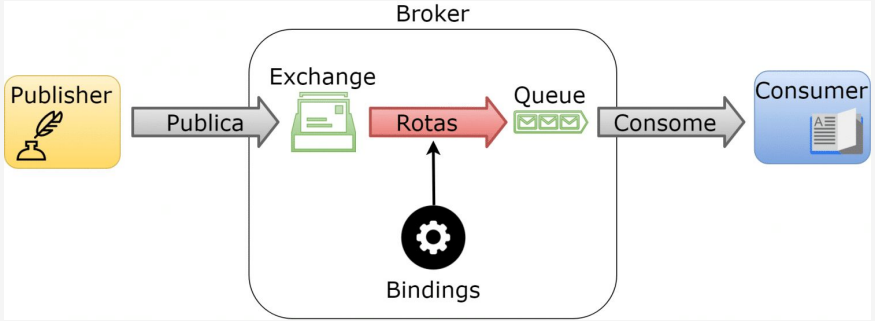
Rodrigo de Castro Michelassi

MAC0352 - Redes de Computadores e Sistemas Distribuídos
Instituto de Matemática e Estatística - USP

O protocolo AMQP

Função:

- ➔ Conexão e troca de mensagens, em formato de texto, entre um cliente e um servidor.
- ➔ Clientes devem se inscrever em filas, e receberão mensagens publicadas na fila.
- ➔ Ordem de entrega das mensagens definidas por um esquema round-robin.



O protocolo AMQP

Componentes

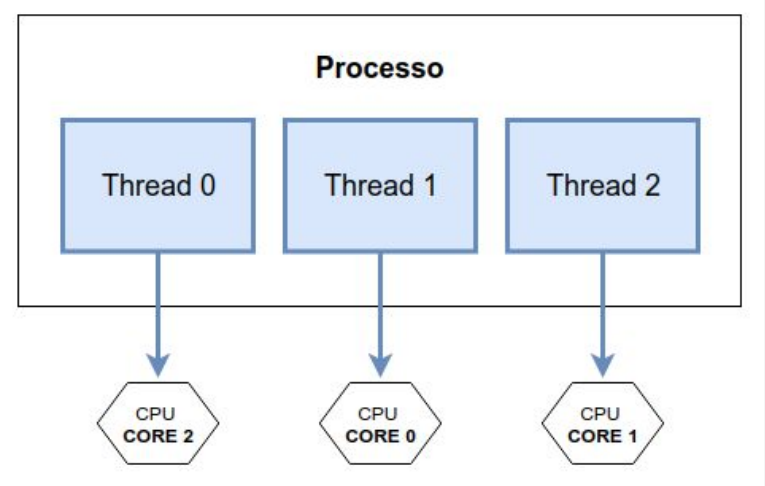
- ➔ **Fila:** ambiente utilizado para a publicação de mensagens.
- ➔ ***Consumer*:** clientes, que se inscrevem em filas e aguardam sua vez para receber alguma mensagem.
- ➔ ***Publisher*:** responsável por inscrever uma mensagem na fila.

OBS: A implementação do servidor foi feita com base no funcionamento do já implementado RabbitMQ.

Threads

- A implementação trazida pelo professor era organizada usando *Forks*, porém, a fim de facilitar a construção do EP, mudamos para *Threads*.
- As *Threads* são responsáveis por paralelizar o nosso código.
- Cada execução, de criação de fila, inscrição de consumidor ou publicação de mensagem, está em uma *Thread* diferente.
- Cada Thread carrega um *Socket*.
- Em resumo, cada operação leva a um processo diferente, executado separadamente.

```
23    struct ThreadArgs {
24        int connfd;
25    };
```



Estabelecimento de Conexão

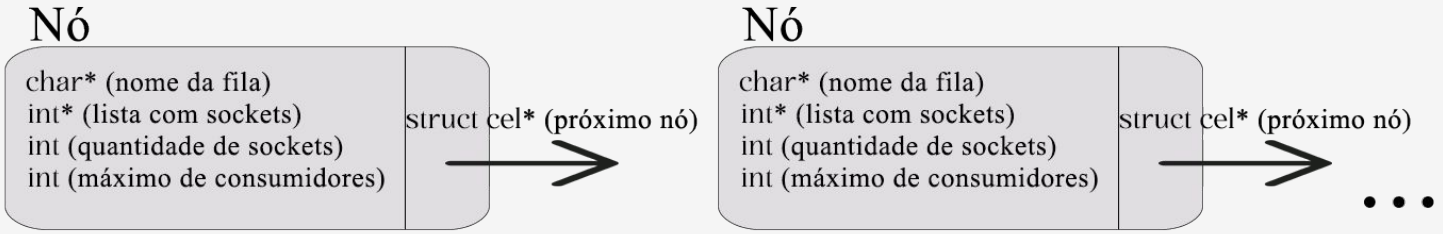
- Para conectarmos um cliente no servidor, que irá executar alguma das ações descritas, o protocolo segue um padrão.
- **Read Header:** responsável pela leitura do cabeçalho do protocolo.
- **Connection Start:** iniciar (e confirmar) e a conexão.
- .**Connection Tune:** segundo passo no estabelecimento da conexão.
- Connection Open: abrir a conexão para o cliente.
- **Channel Open:** abrir um canal de comunicação com o cliente.
- Após isso, faremos a leitura do método está sendo solicitado.

```
AMQP       74 Protocol-Header 0-9-1
TCP        66 5672 → 32854 [ACK] Seq=1 Ack=9 Win=65536 Len=0 TSval=746518335 TSecr=746518335
AMQP      586 Connection.Start
TCP        66 32854 → 5672 [ACK] Seq=9 Ack=521 Win=65024 Len=0 TSval=746518337 TSecr=746518336
AMQP      394 Connection.Start-Ok
AMQP       86 Connection.Tune
AMQP       86 Connection.Tune-Ok
AMQP       82 Connection.Open vhost=/
TCP        66 5672 → 32854 [ACK] Seq=541 Ack=373 Win=65536 Len=0 TSval=746518339 TSecr=746518338
AMQP       79 Connection.Open-Ok
AMQP       79 Channel.Open
AMQP       82 Channel.Open-Ok
```

Fila

Armazenamento

- ➔ A nossa fila é responsável apenas por receber mensagens, e transmitir para *consumers* inscritos nelas.
- ➔ Precisamos armazenar os nomes das filas e os clientes inscritos nelas.
- ➔ Para armazenarmos essas informações, escolhemos usar uma estrutura de lista ligada.
- ➔ Cada nó da lista armazena: o nome de uma fila, o socket dos clientes inscritos nessa fila, a quantidade de clientes inscritos e o máximo de *consumers* suportado.
- ➔ O máximo é subjetivo, visto que a lista de sockets é dinâmica (tem mais tamanho alocado na memória, conforme a necessidade).



Fila

- ➔ Além dos marcadores mencionados no slide anterior, implementamos, na estrutura de cada nó de fila, uma lista de *Consumer Tags* (cTag) e *Deliver Tags* (dTag).
- ➔ Esses dois marcadores não são necessários, porém salvamos assim para fins de praticidade na publicação de mensagem.
- ➔ Como cada *tag* está associada a um *consumer*, também utilizamos elas no *round-robin*.

```
27  typedef struct cel{
28      char* nomeFila;
29      int* listaSockets;
30      int qtdSockets;
31      int maxConsumers;
32      uint8_t** cTag;
33      uint64_t* dTag;
34      struct cel* prox;
35  }No;
```

Round-Robin na fila

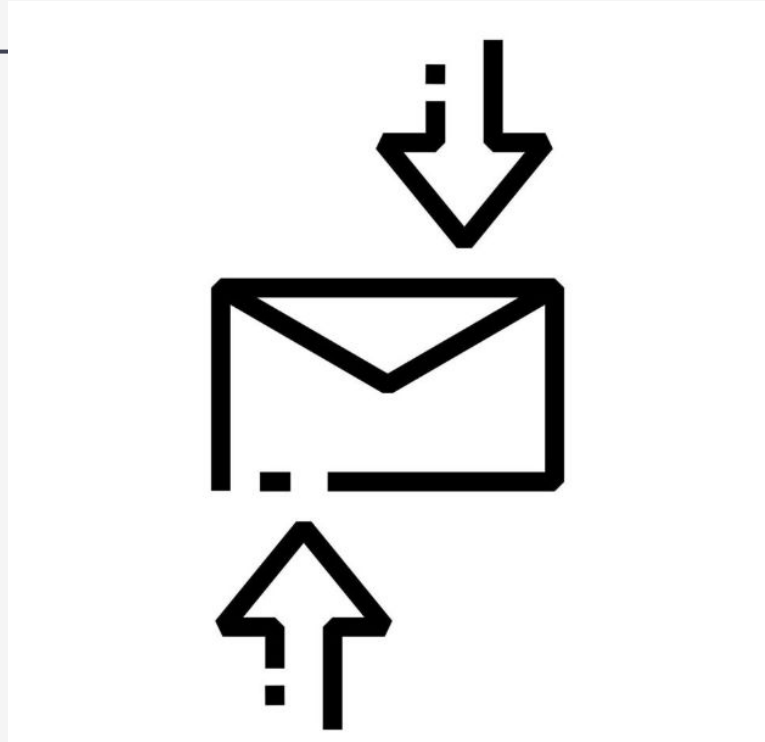
- Esquema de distribuição de mensagens da fila entre *consumers*. Administra a disposição dos *sockets* dos *consumers* em cada fila.
- A ideia para aplicar o esquema *round-robin* foi de simular uma fila circular.
- Cada vez que um cliente recebe uma mensagem, ele é mandado para o fim da fila, e os seguintes avançam uma posição.
- Todo novo cliente é inscrito no fim da fila. O primeiro cliente é inscrito na primeira posição (que também é a última).
- Toda mensagem é enviada para o primeiro cliente na fila.

Consumer

- ➔ Quando executado, aguarda pela chegada de uma mensagem.
- ➔ Todo consumer inscrito em uma fila, fica inscrito até que sua thread seja fechada.
- ➔ Só pode ser inscrito em uma fila já existente. Não é capaz de criar uma fila nova.
- ➔ Quando chega sua vez, recebe a próxima mensagem da fila.
- ➔ É a única função AMQP que não encerra conexões, canais e o socket, para que se mantenha ligado no sistema de thread.

Publisher

- ➔ Responsável por publicar uma mensagem em uma fila.
- ➔ Recebemos por leitura o nome da fila e a mensagem a ser publicada.
- ➔ Escrevemos, no socket do usuário prioritário dessa fila, a mensagem, conforme o protocolo.
- ➔ Após isso, esse usuário é direcionado ao fim da fila.
- ➔ Ao fim de cada mensagem, a conexão e os canais são encerrados.



Funções Auxiliares

- ➔ **generateCTAG:** responsável por gerar um texto aleatório no formato de uma consumer tag, para enviarmos na mensagem do *consumer*.
- ➔ **char2int:** feita com a ajuda do Filipe Tressman, faz um bitshift de 8 bytes para converter uma sequência de chars para um int.
- ➔ **char2LongLong:** igual a char2int, porém retornamos um valor long long int.
- ➔ **getConsumerSock:** retorna o socket do *consumer* que está na posição 0 da fila.
- ➔ **existeFila:** retorna 1, caso uma fila de determinado nome exista, 0 caso contrário.
- ➔ **iniciarLista:** “construtor” da lista ligada descrita anteriormente.
- ➔ **adicionaFila:** adiciona uma nova fila a lista ligada, ou um socket novo, caso a fila já exista.
- ➔ **realocaEspaco:** aumenta a capacidade do armazenamento de *consumers*.
- ➔ **imprimeFilas:** imprime o nome das filas que estão registradas, usada somente para teste.
- ➔ **roundRobin:** realiza o esquema round-robin de rotação dos sockets em uma fila.
- ➔ **threadConnection:** carrega toda a execução do servidor, aqui chamamos todas funções AMQP no sistema de *threads*, que foi adotado em detrimento do *fork*, proposto pelo professor.

Testes da CPU

- Para os testes, fizemos um arquivo em bash, responsável por executar nosso servidor.
- Esse arquivo inicia o servidor e abre um certo número de terminais, cada um com um consumer (número decidido pelo usuário, no código).
- Após isso, o servidor executa também um certo número de *publishers*, que enviarão mensagens para os *consumers*.
- No fim, o terminal principal exibe os dados de uso da CPU a cada alguns cadastros de *publishers*.
- Para obtermos esses dados, usamos os seguintes comandos:

```
top -b -n 1 | grep "Cpu" | awk -F '[, ]+' '{printf "%s,%s ", $2, $3}'
```

```
top -b -n 1 | grep "Cpu" | awk -F '[, ]+' '{printf "%s,%s", $5, $6}'
```

```
free -h | grep "Mem:" | awk '{printf "%s/%s", $3, $2}'
```

- Esses comandos são responsáveis por entregar, respectivamente, uso da CPU por usuário, pelo sistema e uso da memória.
- Para nossos testes, o uso da CPU por usuário possui muito mais impacto, e apresentaremos ele nos gráficos.

Testes da CPU

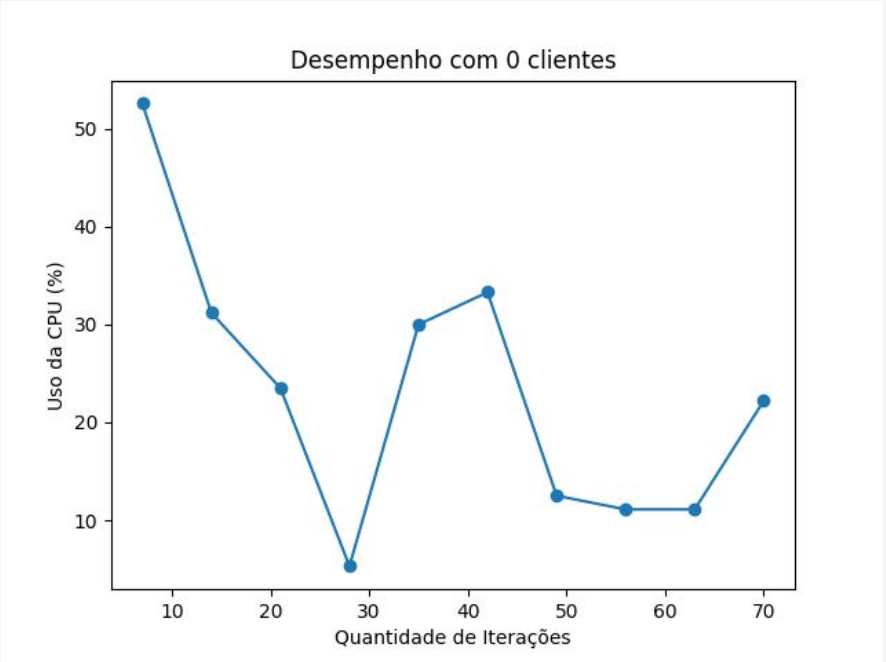
- Os gráficos gerados foram gerados a partir de um arquivo .csv, em Python.
- Os testes que serão apresentados a seguir foram feitos em uma máquina virtual (Virtual Box), utilizando Ubuntu 22.04, com 3GB de memória disponível.
- **Teste com 0 clientes:** rodamos o servidor e um Loop com 70 iterações, a cada 10 expusemos o uso da CPU.
- **Teste com 10 clientes:** executamos o servidor com 3 *consumers* e 7 *publishers*, mostramos o uso da CPU na adição de cada *publisher*.
- **Teste com 100 clientes:** executamos o servidor com 30 *consumers* e 70 *publishers*, mostramos o uso da CPU após a adição de 10 *publishers*.

- Segue o código responsável por gerar os gráficos, em Python:

```python
import pandas as pd
from matplotlib import pyplot as plt

data = pd.read_csv('data.csv', sep=';')

x = data['pub']
y = data['cpu']

data = data.sort_values(by='pub')

plt.plot(x,y, marker='o')
plt.xlabel("Quantidade de Publishers")
plt.ylabel("Uso da CPU (%)")
plt.title("Desempenho com 10 clientes")
plt.show()
```

Uso da CPU com 0 clientes

- ➔ Vemos que a maior atividade foi logo que o servidor foi executado.
- ➔ O uso da CPU começa a cair rapidamente, um tempo após a execução do servidor.
- ➔ Vemos uma pequena oscilação, que pode ter ocorrido devido a execução de processos em segundo plano.
- ➔ Em geral, o servidor consome pouco da CPU, muitas vezes abaixo de 30%.

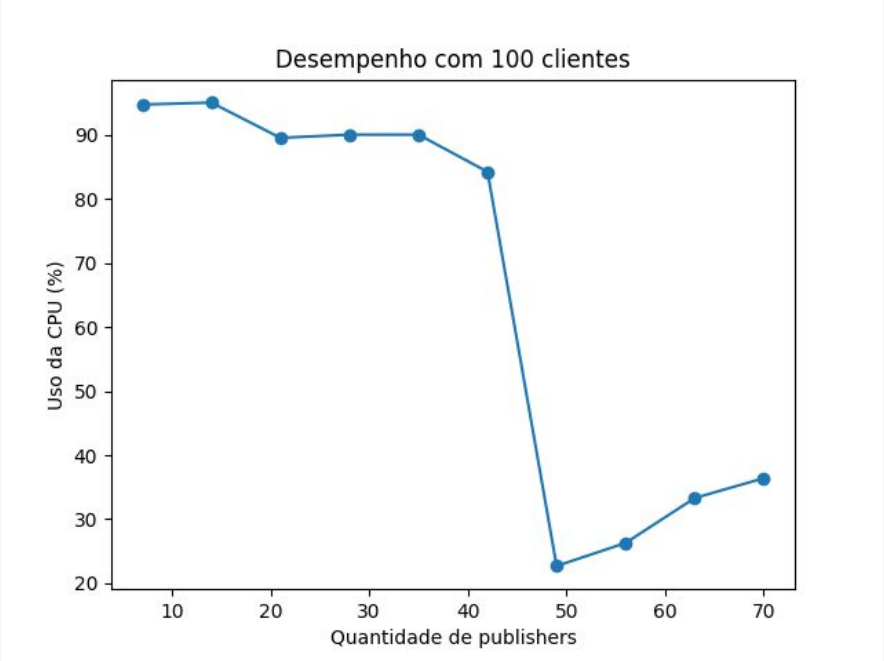


Uso da CPU com 10 clientes

- ➔ Temos 7 *publishers* e 3 *consumers*.
- ➔ A atividade com 10 clientes já é maior que com 0 clientes.
- ➔ Atingimos 90% de uso da CPU na primeira iteração.
- ➔ O mesmo perfil de decrescimento é apresentado nesses testes.
- ➔ Embora, conforme o número de publishers aumente, os valores se mantém relativamente constantes, próximos de 40% de uso da CPU.

Uso da CPU com 10 clientes

- ➔ Temos 70 *publishers* e 30 *consumers*.
- ➔ Para esse teste, o perfil do gráfico é bem diferente.
- ➔ Vemos que há um início de uso de CPU por volta de 90%, que se mantém durante a inscrição de cerca de 40 publishers.
- ➔ Após isso, há um decaimento no uso da CPU muito notável.
- ➔ Por fim, os valores de uso da CPU se estabilizam por volta dos 40%, como no caso anterior.



Testes da Rede

- ➔ Para os testes da rede, mantivemos o mesmo esquema de execução tratado anteriormente.
- ➔ Ainda estamos utilizando o arquivo em bash para executar os testes.
- ➔ Agora, não obtemos os dados de uso da rede do terminal, mas sim do wireshark.
- ➔ Dessa forma, se torna inviável apresentar gráficos, como fizemos para o uso da CPU.
- ➔ Segue então tabelas que demonstram os dados de rede obtidos pelo wireshark, em cada caso:

	0 Clientes	10 Clientes	100 Clientes
Bytes Capturados	38146	37948	355441
Execução (s)	6	6	42
Pacotes Capturados	387	384	3624

Testes da Rede

- Foram adotadas algumas medidas a fim de garantir que os dados obtidos em rede eram decorrentes do servidor:
  - Fechar todas abas presentes no computador utilizado.
  - Fechar todas as abas presentes na máquina virtual utilizada.
  - Fechar todas as abas em um notebook, utilizado simultaneamente.
  - Desligar o Wifi de todos dispositivos móveis, conectados na mesma rede.
  - Filtrar os dados no Wireshark, pela porta 5672, em modo Loopback.
  - Executar apenas o servidor, além dos processos ocultos da máquina.
- É possível notar que o número de bytes capturados é proporcional à quantidade de clientes conectados no servidor.
- Isso pois, para 100 clientes, consumimos (aproximadamente) 10 vezes mais bytes que com 10 clientes no servidor.
- Um dado interessante é que com 10 clientes, o tempo de execução é, em proporção, maior que para 100 clientes.
- Esse fato pode ser explicado pelos gráficos do teste de CPU, em que o maior uso da máquina é no início da execução.

Considerações Finais

- ➔ Durante os testes, foi possível perceber que o servidor não suportou, por exemplo, cadastrar 50 *consumers* (suportou algo entre 30 e 40).
- ➔ Podemos concluir que nossa implementação do servidor funcionou bem, apesar dessa fatalidade.
- ➔ A troca de mensagem ocorre da forma que deveria, e o sistema round-robin para gerenciamento de filas funciona corretamente.
- ➔ Todos os pacotes necessários são enviados corretamente, pelo rastreamento do Wireshark.
- ➔ O programa utiliza pouco da CPU, pelo que foi visto nos testes.
- ➔ Não conseguimos concluir muito sobre os testes de rede, sem termos outros valores de referência para o que buscamos.
- ➔ Como um todo, esse projeto foi muito importante para o aprendizado sobre o protocolo AMQP e a implementação de um servidor, em linhas gerais.