EP1 - Servidor AMQP

MAC0352 - Redes de Computadores e Sistemas Distribuídos
Beatriz Viana Costa - 13673214

Implementação

Introdução

- O presente exercício programa visa construir um servidor AMQP na linguagem C, semelhante ao *RabbitMQ*, *software* já existente que segue o protocolo;
- Para isso foram utilizadas as documentações disponibilizadas e também as capturas do *Wireshark*;
- A implementação do protocolo fez vasto uso das funções `read()` e `write()` para a manipulação dos pacotes.

Wireshark

Paralelização do programa utilizando *Pthreads*

- O código foi paralelizado utilizando a biblioteca `pthread.h`, dessa forma, substituímos o `fork()` do código original. Essa paralelização se fez necessária para que vários clientes pudessem se conectar simultaneamente ao servidor;

- Para a paralelização foi necessário criar uma nova função `void *makeConnection()`, que cuidaria das conexões dos clientes, e também uma `struct`, que seria parâmetro dessa função;

- Todos os processos de leitura de pacotes e respostas aos clientes foram feitas nessa nova função. Ou seja, a abertura de conexões e canais, declaração de filas e consumidores, entre outros, foram todos feitos de forma paralelizada.

Armazenamento das informações de filas e consumidores

- Para o armazenamento das informações da execução, como as filas declaradas, os consumidores que estão cadastrados em cada uma das filas e seus respectivos *sockets*, foram utilizadas três estruturas diferentes em conjunto;

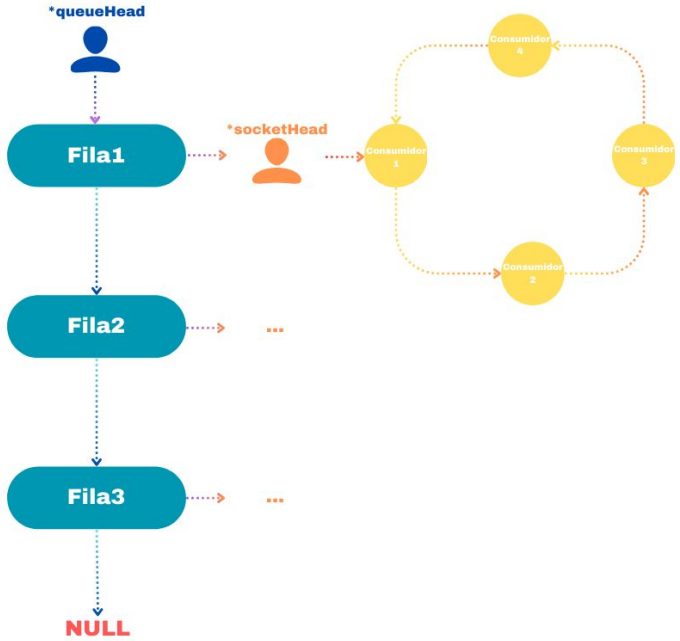
- A estrutura de dados utilizada para o armazenamento das informações de cada fila foi a lista ligada, já para guardar os *sockets* de cada fila, foi utilizada a lista ligada circular.

```
typedef struct socketNode{
    int connfd;
    uint8_t* consumerTag;
    struct socketNode* next;
    struct socketNode* ant;
    int head;
}socketNode;

typedef struct queueNode{
    uint8_t* queueName;
    struct queueNode* next;
    struct socketNode* socketHead;
    int socketSize;
    uint64_t deliveryTag;

}queueNode;

typedef struct queue{
    struct queueNode* head;
    int queueSize;
}queueList;
```

Esquema de *Round-Robin*

- Como a estrutura de dados utilizada para armazenar os consumidores de cada fila foi a lista ligada circular, para aplicar o esquema de *Round-Robin* (a rotação dos consumidores que vão receber as mensagens publicadas) foi necessário apenas mudar a head da lista para o próximo toda vez que um consumidor é utilizado;

- Quando um consumidor é adicionado, ele é colocado na última posição, ou seja, logo antes da cabeça da lista.

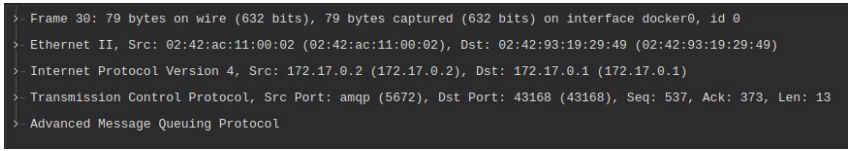


Funções de abertura e fechamento de conexões e canais

- Para a implementação das funções de abertura e fechamento de conexões e canais, como `Protocol Header`, `Connection.Start`, `Connection.Tune`, `Channel.Close`, entre outras, foi necessário apenas ler os pacotes enviados pelos clientes e enviar as respostas padrões;

- Essas respostas padrões podem ser montadas com base nas documentações e a partir das leituras feitas pelo *Wireshark* no campo *Advanced Message Queuing Protocol*;

- Para identificar quais respostas deveriam ser enviadas pelo servidor, foram observadas as portas de origem (*Src Port*) no *Wireshark*, sendo a porta do servidor a padrão do AMQP, 5672.



$ amqp-declare-queue

- Para a implementação do método, o pacote `Queue.Declare` enviado pelo cliente é lido e então uma fila com o mesmo nome presente no campo `Queue` do pacote é criada ;

- A fila é ligada ao fim da lista ligada de filas, além disso uma `Delivery tag` já é atribuída e a sua lista de sockets é inicializada como nula;

- Como resposta a criação da fila, o servidor envia um pacote de `Queue.Declare-Ok`, retornando o nome da fila criada. Após isso o servidor fecha o canal e a conexão.

$ amqp-consume

- O pacote `Basic.Consume` enviado pelo cliente é lido pelo servidor e então os nomes da fila de cadastro do cliente e a `Consumer-Tag` são guardados;
- Em todos os casos de teste realizados, os consumidores vinham com o campo de `Consumer-tag` vazio, dessa forma o servidor cria de forma pseudo-aleatória uma `Consumer-Tag` para o cliente;
- O servidor manda como resposta um pacote `Basic.Consume-Ok` contendo a Consumer-Tag do cliente;
- Caso a fila solicitada pelo cliente não exista, uma mensagem de erro é enviada para o *consumer* e a conexão é fechada.

$ amqp-publish

- Nesse método três *frames* são enviadas pelo cliente, dela o servidor retira a `Routing-Key`, que será igual à fila de publicação da mensagem e a mensagem, que fica no campo de `Payload`;

- O servidor então pega o *socket* do cliente que deve receber a mensagem na rodada, e então envia um pacote de `Basic.Deliver`, contendo a `Consumer-Tag` e a `Delivery-Tag`;

- O servidor também é responsável por mandar a `Content header`, com o tamanho da mensagem e logo em seguida o `Content body`, contendo a mensagem em si;

- Por fim, o servidor fecha o canal e a conexão e lê o pacote `Basic.Ack` enviado pelo cliente;

- Cado a fila solicitada para a publicação da mensagem não tenha sido declarada ou não haja nenhum consumidor cadastrado nela, o servidor apenas fecha a conexão.

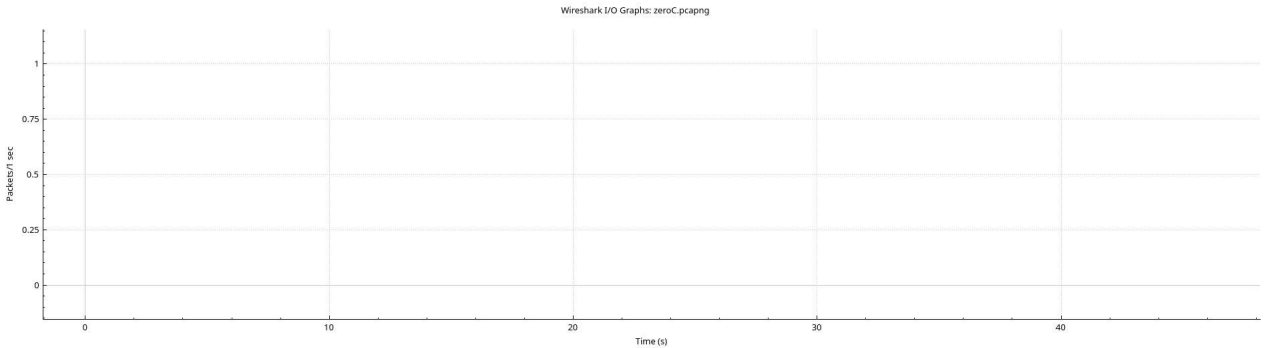
Testes

Como foram realizados os testes

- Para a realização dos testes de desempenho do servidor foi feito um *script* em bash;
- Monitoramos o uso de CPU em porcentagem. também o uso da rede em *bytes* , e a quantidade de pacotes AMQP e TCP transferidos;
- O monitoramento de CPU foi feito pelo uso da função pidstat, já o de rede pelo próprio *Wireshark*;
- Em cada teste, metade dos clientes eram *consumers* e a outra metade *publishers*, sendo que eram declaradas uma fila para cada *consumer*;
- Os dados foram coletados por 60 iterações, de forma que uma mensagem fosse publicada em cada fila a cada 2 iterações;
- Em todos os gráficos retirados do *Wireshark* foi utilizado o filtro tcp.port == 5672 representado pelo linha verde e amqp representado pela linha vermelha.

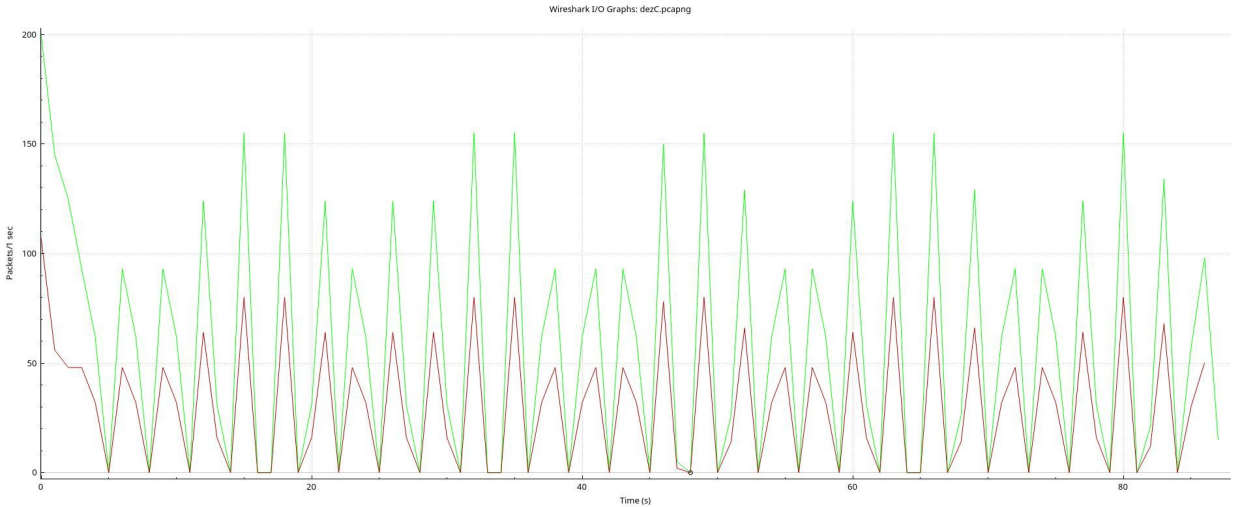
Servidor ocioso

- Quantidade de pacotes transferidos: 0;
- Quantidade de *bytes* transferidos: 0;
- Uso médio da CPU: 0.00%.



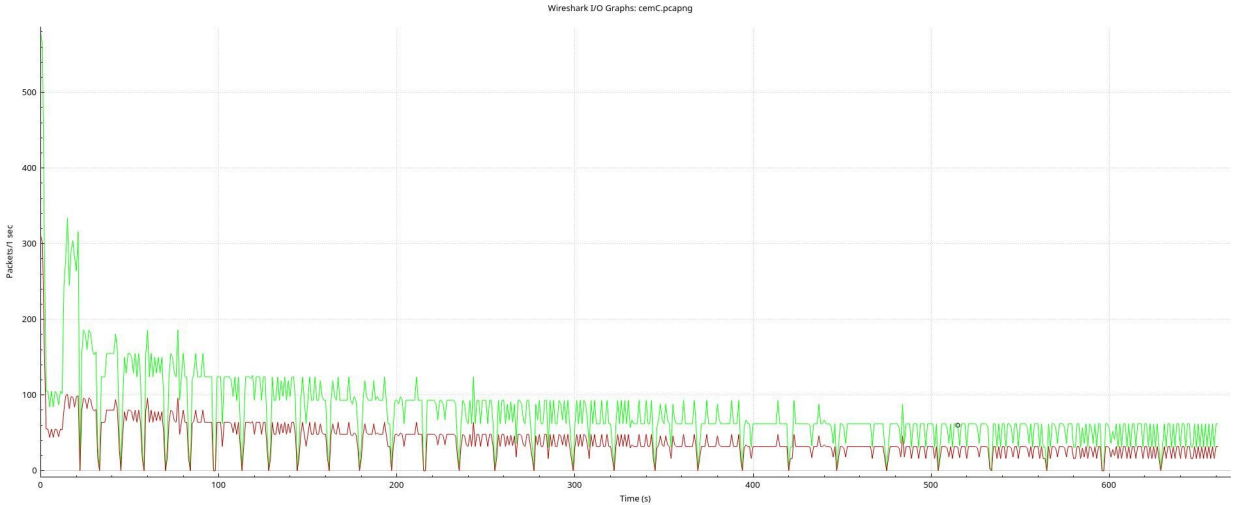
Servidor com 10 clientes publicando e recebendo mensagens simultaneamente

- Quantidade de pacotes transferidos: 4.805;
- Quantidade de *bytes* transferidos: 522000;
- Uso médio da CPU: 0.00%.



Servidor com 100 clientes publicando e recebendo mensagens simultaneamente

- Quantidade de pacotes transferidos: 48.184;
- Quantidade de *bytes* transferidos: 5000000;
- Uso médio da CPU: 0.01%.



Conclusão

- ➔ Em relação aos dados de quantidade de *bytes* e pacotes transferidos, os resultados foram sólidos e condizem com o aumento de uso do servidor de acordo com o crescimento de clientes cadastrados;

- ➔ Já os testes relacionados à CPU mostraram um baixo uso da mesma pelo servidor, mesmo no cenário em que haviam 100 clientes conectados.