Faculdade de Engenharia da
Universidade do Porto
Sistemas Distribuidos de Larga Escala
M.EIC004

Projeto 1 - Reliable Pub/Sub service

Amanda Silva - up201800698
Filipe Pinto - up201907747
Pedro Rezende - 201900513
Victor Nunes - up201907226

2022/2023
semestre de Outono

.

Conteúdo

1 Descrição do problema

O problema proposto é a construção de um sistema confiável de comunicação entre *publisher* e *subscribers*. Aonde os *publishers* enviam mensagens referentes a um tópico, para um servidor PUB/SUB, independentemente de como ou quando essas mensagens devem ser processadas. Os *publishers* se comunicam com *subscribers* de forma assíncrona e intermediada por um servidor usando um *Middleware* orientado a mensagens, o zeroMQ.

O sistema deve suportar as seguintes operações:

- put() para publicar uma mensagem num tópico
- get() para consumir uma mensagem de um tópico
- subscribe() para subscrever um tópico
- unsubscribe() para anular a subscrição um tópico

Como não nos foi imposta nenhum tipo de restrição no que toca a linguagem de programação, escolhemos Python para a implementação deste projeto.

2 *Sockets* e *Patterns* usados

Para a comunicação entres entidades são utilizadas as *sockets* modelo *bind/connect* do zeroMQ, uma vez que elas se encaixam bem num modelo cliente servidor, dado que o servidor não precisa saber o endereço dos seus clientes. Para a comunicação foi utilizado o *pattern* interno do zeroMQ o REQ/REP.

Essa escolha foi feita especialmente pela necessidade de garantirmos (dentro do que é razoável) uma comunicação confiável. Preferimos utilizar REQ/REP ao invés do pattern PUB/SUB, que veem com zeroMQ, pela necessidade de uma comunicação bidirecional. PUB/SUB é especialmente útil para publicarmos uma informação a vários *subscribers*, particulamente vantajoso no caso de *streaming* de dados, porém isso vem associado a um custo, para conseguir essa eficiência zeroMQ não permite *back chatter*, ou seja, não permite que as mensagens fluam na direção do *publisher*, necessário para garantir uma comunicação confiável com o servidor intermediário.

Implementamos ainda, um sistema de armazenamento persistente, onde as informações sobre os clientes, e respetivos tópicos inscritos, eram gravadas em disco. No caso de falha por parte do server, quando este reiniciar, vai ler

dois ficheiros(topics.pickle e offsets.pickle), tornando possível a preservação do estado antes do erro.

Para além disso também foi aplicada a *pattern lazy pirates* para cobrir a omissão de resposta do servidor. Onde caso o Servidor não responda o cliente, no tempo estipulado (timeout), ira reenviar a mensagem até tres vezes. Após as três tentativas o cliente assume que o servidor não se encontra mais disponível. Mais informações quanto a aplicação da *pattern* se encontram na secção 4 - Rigor referente a solução encontrada.

3 Protocolo implementado

3.1 Menssagens

As menssagens, enviadas através o recurso tomultipart do ZMQ, seguem o seguinte protocolo implementado para este projeto:

Protoco de Menssagens		
Comando	Tópico	Valor
SUB	Nome do Tópico	Identificador do Cliente
UNSUB	Nome do Tópico	Identificador do Cliente
GET	Nome do Tópico	Identificador do Cliente
PUT	Nome do Tópico	Valor a ser publicado
ERROR	Nome do Tópico	Identificador do cliente

As mensagens de Request são enviadas pelo cliente para o servidor, onde o cliente ficará pendurado até receber um *Reply* com uma mensagem caracterizada por "ERROR"caso o pedido encontre-se irregular. Ou, em caso de sucesso, é recebido com os valores referidos na tabela.

Valores devolvidos		
Comando	Erro	Sucesso
SUB	*ERROR: client already inscribed*	Identificador do Cliente
UNSUB	*"ERROR: topic does not exists"* ou *"ERROR: client is not inscribed"*	Identificador do Cliente
GET	*"ERROR: topic does not exists"* ou *"ERROR: client is not inscribed"* ou *"ERROR: client already received all the values available on this topic"* *"ERROR: topic doesn't exist"*	Identificador do Cliente
PUT	N/A	Valor a ser publicado

3.2 Implementação Server

Em ordem de garantir os requerimentos propostos para esse projeto o Servidor utiliza dois dicionários para armazenar as ações realizadas pelas mensagens. Sendo esses:

topics = "topic1": (last, 1: value1, 2: value2..., [clientId1, clientId12])
offsets = client1$_t$*opic*1 : 2, *client*2$_t$*opic*1 : 1, *client*1$_t$*opic*2 : 3...

Onde *topics* tem como chave o nome do tópico e como valor uma tuple em que: o primeiro elemento é a chave do último valor inserido. o segundo elemento é um dicionário que contém como chave, id's com valores incrementais, e como valor o conteúdo do publisher. E o terceiro elemento é uma lista com todos os id's dos clientes (id passados pelos mesmos) inscritos no tópico.

Em *offsets* é guardado como valor o id da publicação que o cliente inscrito

naquele tópico terá de receber ao fazer um get.

4 Rigor referente a solução encontrada

Para implementar uma entrega que respeite a "exactly once", a implementação deve ter em conta vários cenários falhas.

4.1 Falha no client

Começamos por abordar a possibilidade de uma falha no *client*, ou seja, ficar inacessível ao server por um período indefinido, seja porque desapareceu momentaneamente ou permanentemente.

Com esta informação, quando um client subscreve (SUB) um tópico, o seu id e armazenado no dicionário e associado a esse mesmo tópico.

Caso a menssagen enviada pelo cliente não chege ao servidor, após um periode de 2.5 segundos,Caso não obtenha resposta por parte do server, estabelece uma nova ligação com o mesmo e reenvia a mensagem. Entretanto, na eventualidade do server responder a mensagem inicial que ultrapassou o tempo de resposta estipulado pelo cliente, o cliente receberar duas mensagens seguidas (falha quanto idempotência) já que nao há como cliente verificar se o conteúdo da resposta é o correto.

4.2 Falha no servidor

Para combater possíveis falhas no servidor implementamos um sistema de armazenamento persistente, como já mencionado. Ao inicializar, o servidor deve recuperar esses dados de modo a garantir uma tolerância adicional a falhas do sistema.

4.3 Falhas Não Cobertas

A nossa solução não consegue cobrir todos os casos de falhas possíveis, mas julgamos ter solucionado os casos prováveis de falha.

Uma situação possível é um *crash* entre a transcrição dos dados em disco e o envio das mensagens, nesse caso mesmo que o servidor se recupere o cliente que faz o pedido *get* perderá a resposta do pedido original, uma vez que para o servidor a mensagem já foi enviada com sucesso e qualquer novo pedido do cliente faz referência ao próximo elemento do tópico. O mesmo se repete a um cliente que faz um pedido *put*, mas nesse caso ao risco de uma mensagem ser duplicada, dado que o cliente faz o pedido novamente visando receber uma confirmação.

Conclusão

O grupo conseguiu desenvolver o sistema que cumpre os requisitos especificados no enunciado. A entrega "exactly-once"foi cumprida com para os cenários de falhas mais comuns.

Além disso, durante a implementação, tivemos sempre em consideração que era necessária uma solução simples com qualidade de software adequado a estes quatro anos de estudo.

Estamos satisfeitos com o resultado obtido.