# Projeto 1 - Message Broker

Computação Distribuída

Neste trabalho foi desenvolvido um **Message Broker**. Para além de sockets, foram utilizados selectors no broker, para lidar com várias comunicações em simultâneo.

Para os tópicos hierárquicos, decidimos usar uma abordagem em árvore através de classes (topics.py). Cada objeto (*topic*) tem uma lista com referências para os "tópicos filho", um set com as sockets dos Consumers subscritos, uma referência para o "tópico pai" e o valor da última mensagem publicada nesse tópico.

Na inicialização de cada entidade é criada uma *Queue*, sendo posteriormente enviada uma mensagem para o broker de apenas um byte, com a informação do tipo de serialização utilizada pela entidade (1=JSON, 2=XML, 3=Pickle). O broker, ao receber esta mensagem, associa, utilizando um dicionário (*serializacoes*), a socket do remetente ao número correspondente ao tipo de serialização. No tipo de codificação XML foi utilizado o xml.etree. Adoptamos este mecanismo de guardar o tipo de serialização de cada entidade no broker, invés de essa ser enviada antes de cada mensagem, para não sobrecarregar a rede e ocupar menos largura de banda.

Assim, sempre que o broker recebe uma mensagem, ou se pretende enviar uma, consulta este dicionário, para saber qual o tipo de serialização usado na mensagem, ou então para que tipo de serialização é que a mensagem deve ser convertida, cumprindo assim o seu papel de tornar possível comunicação entre entidades com tipos diferentes de serialização.

Como o protocolo TCP é orientado à ligação, é sempre enviada, excepto na inicialização, uma mensagem com 5 bytes, contendo o tamanho da mensagem que será enviada de seguida, para permitir que todos os bytes da mensagem seguinte sejam lidos, independentemente do seu tamanho. Todas as mensagens enviadas para o broker (excetuando as referidas acima) possuem também um campo '**OP**', com o objetivo de indicar que tipo de mensagem se trata.

Logo após o envio da primeira mensagem que contém informação sobre a serialização, é enviada outra, com o objetivo de obter informação acerca dos tópicos que o Consumer deseja subscrever, ou de indicar em que tópico o Producer vai publicar. O campo '**OP**' desta mensagem tem o valor '**join**'. Para além desse, possui também o campo '**TOPIC**', cujo valor é o nome do tópico onde a entidade se pretende subscrever/publicar, e '**TYPE**', que pode ter os valores 0 ou 1, dependendo se se trata de um Producer ou Consumer, respetivamente. Após a recepção da mensagem, no caso de ter sido enviada por um Producer, o broker associa, num dicionário (*produtores*), a sua socket ao tópico indicado na mensagem. Se o tópico não existir, é criado e inserido no sítio adequado na estrutura (árvore). Escolhemos guardar a informação sobre os tópicos associados a cada Produtor no broker, em vez de os especificar em cada mensagem enviada, para que estas tivessem tamanho mais reduzido e ocupar menos largura de banda. No caso do remetente ser um Consumer, este é adicionado na lista de subscritores do objeto tópico correspondente. Quando num tópico é adicionado um novo subscritor, esse subscritor é também adicionado às listas de subscritores dos seus tópicos filho, de modo a facilitar a recolha das sockets necessárias para o envio de publicações. Após um Consumer ter sido adicionado à lista de subscritores, é-lhe enviado a última mensagem publicada no tópico, assim como a dos seus tópicos filho.

Outros valores que o campo '**OP**' pode ter são:

'**publish**' → Com o objetivo de publicar uma mensagem (cujo valor se encontra no campo '**VALUE'**), sendo por isso os Producers a única entidade responsável pelo envio deste tipo de mensagens. O broker, ao receber este tipo de mensagem, envia o seu valor para todos os subscritores daquele determinado tópico. A mensagem enviada possui 2 campos, um '**TOPIC**' que indica o nome do tópico a que pertence, e outro '**VALUE**', que contém a mensagem em si.

'**topics_request**' → Ambos os tipos de entidades podem enviar uma mensagem com este tipo, sendo que receberão como resposta uma mensagem com um campo '**OP**' com valor '**topics_list**', e um '**LIST**' com uma string contendo os tópicos que existem no broker. De salientar que o objetivo deste tipo de mensagem é puramente de debug.

‘**leave_topic**‘ → Uma mensagem com este valor no campo ‘**OP**’ é enviado quando o Consumer pretende deixar de subscrever um tópico. Ao receber esta mensagem o broker percorre a estrutura (árvore) começando no tópico com nome ‘/’ (*root*), verificando se cada tópico tem como subscritor o Consumer que enviou a mensagem, sendo posteriormente removido da lista de subscritores do tópico caso a condição anterior se verifique . Esta abordagem viu-se necessária devido ao facto de os “tópicos filho” herdarem os subscritores do pai.

Para demonstrar a utilização de estes dois últimos tipos de mensagens, foi alterado o código do Consumer. Na função **run**, criamos uma variável *i* com o valor 15, que é decrementada por cada mensagem que recebe. Quando *i* tiver o valor 10, é chamada a função **getTopicsList()** da *Queue*, que irá enviar uma mensagem com o valor ‘**topics_request**’ no ‘**OP**’ ao broker. O Consumer recebe resposta a este pedido através da função **pull()**, sendo que, ao contrário do que acontece quando recebe uma publicação normal, o campo *topic* retornado tem o valor None. Quando *i* tiver o valor 0 é chamada a função **leaveTopic()**, que enviará a mensagem com o valor ‘**leave_topic**’ no ‘**OP**’, fazendo com que o consumer deixe de subscrever ao tópico associado à queue onde a função foi chamada.

Foi desenvolvida uma implementação alternativa para os tópicos em árvore, em que os tópicos correspondem a *nodes*, identificados pelo seu nome, que possui também informação sobre o tópico pai, um conjunto com as sockets dos subscritores, a última mensagem publicada e também informação sobre a “camada” onde se encontra o tópico. A principal diferença entre esta implementação em relação à final é na forma como se obtém todas as sockets para onde é necessário enviar as publicações, sendo que nesta implementação a árvore era percorrida com o objetivo de reunir todas as sockets necessárias. Esta forma seria benéfica no ponto de vista da memória, mas não tanto no ponto de vista do tempo, visto ser necessário percorrer a árvore sempre que houvesse uma nova publicação. Por esta razão, decidimos utilizar a outra implementação. O ficheiro tree.py contem o código desta classe e, apesar de não ser utilizado atualmente, decidimos entregá-lo porque foi referido neste relatório.

À medida que avançávamos no projeto verificamos que vários ficheiros necessitavam das mesmas funções, por isso, por uma questão de organização e para evitar replicação de código, foi criado o ficheiro utils.py, contendo implementação dessas funções.

Para verificarmos que a solução estava correta fomos testando por etapas:

1. 1 Producer, 1 consumer e um tipo de serialização (JSON)
2. 1 Producer, 1 Consumer e 2 tipos de serialização (JSON e Pickle)
3. 1 Producer, 1 Consumer e os 3 tipos de serialização
4. 1 Producer, vários Consumers e os 3 tipos de serialização
5. Vários Producers, vários Consumers e os 3 tipos de serialização
6. 1 Producer, 1 Consumer e os 3 tipos de serialização usando tópicos hierárquicos
7. Vários Producers, vários Consumers e os 3 tipos de serialização usando tópicos hierárquicos

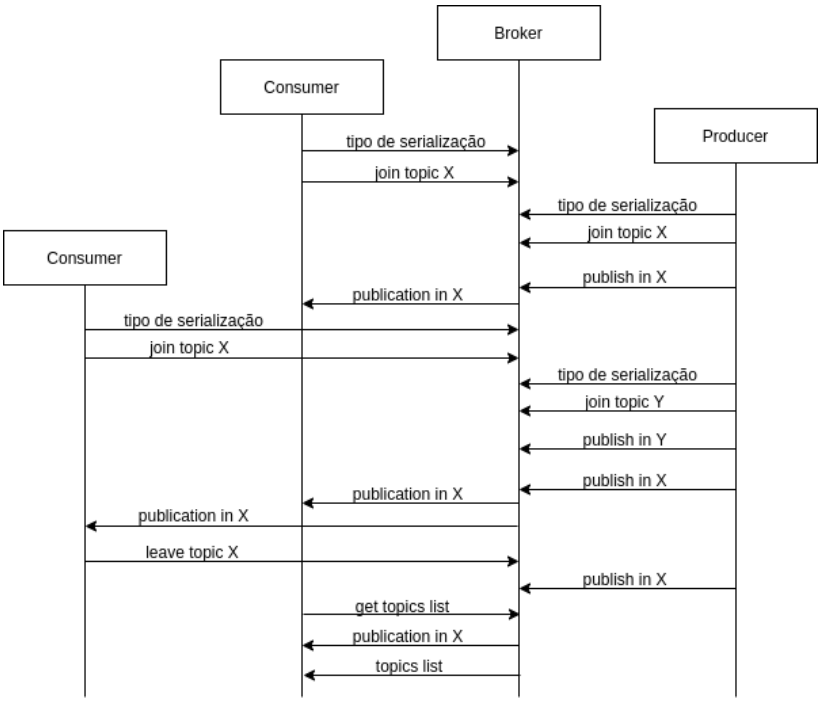


*Imagem 1: Message Sequence Chart*

Mariana Santos (P4) - 93257
Miguel Almeida (P2) - 93372