Trabalho Prático 2

Distributed Object Detection

Trabalho realizado por :
Daniel Gomes nªmec 93015
Mário Silva nºmec 93430

Introdução

Para este trabalho foi-nos recomendado pelo docente da componente prática o uso e aprendizagem de diferentes “recursos” para implementar todos os pontos necessários para o trabalho.

Implementação

O primeiro objetivo foi conseguir com que o cliente comunica-se com o servidor e fizesse upload de um ficheiro, vídeo ou imagem. Para tal, recorremos a uma web framework, o Flask, onde, utilizando o protocolo **HTTP**, obtemos os vídeos através dos pedidos POST do cliente. Estes pedidos **POST** são efetuados recorrendo ao comando ***curl***. Este upload é feito através de um *endpoint* para a raiz do servidor.

Através de um objeto ***Lock*** (com o intuito de garantir a exclusão mútua) da biblioteca *Threading*, permitimos que a identificação de cada vídeo não era afetada pela concorrência de 2 vídeos em simultâneo a serem uploaded para o server. Isto porque como possuímos um contador partilhado por várias threads facilmente o valor esperado deste pode ser afetado pela concorrência.

Após o upload de cada vídeo é adicionado mais uma chave (identificadora do vídeo) para a estrutura de dados, ***video_map,*** estrutura esta que irá ser explicada a seguir neste documento.

Após esta etapa, era necessário enviar os frames aos workers de forma a que fosse possível estes serem processados e enviar a informação obtida deste processamento para o server. Inicialmente pensámos em comunicar com os workers através de Flask também e para o algoritmo de distribuição de frames criamos a classe Round Robin. No entanto rapidamente apercebemo-nos de que desta maneira teríamos de tratar todos os passos envolventes nesta operação, tal como lidar com os eventuais crashes dos workers, que em princípio seria necessário guardar os frames e o seu estado para não se perderem dados.

Posto isto para a comunicação com o(s) worker(s), decidimos utilizar a Library **Celery**.

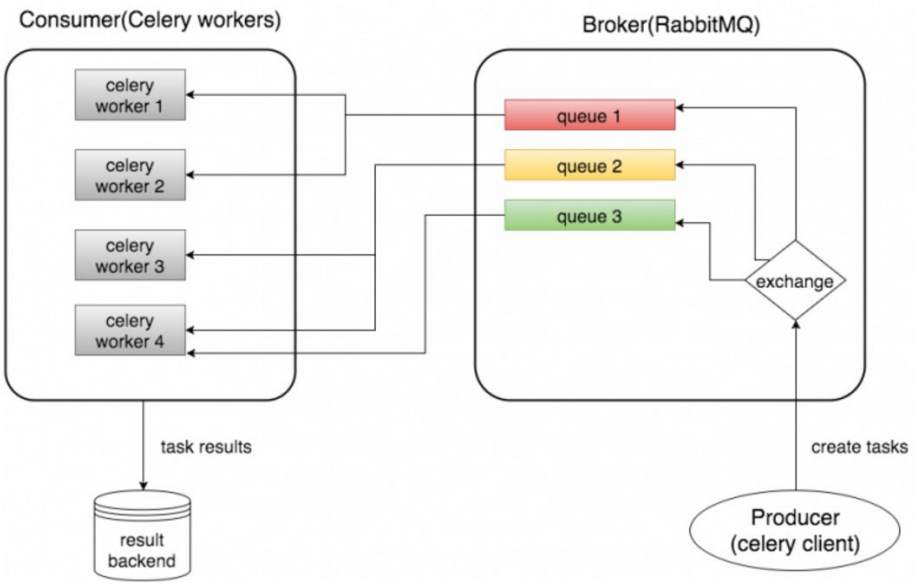
Através do Celery conseguimos implementar facilmente uma *Task Queue* onde é colocada a tarefa de processar os frames nessa queue, os argumentos passados nessa tarefa são passados para o worker através do mecanismo de serialização **Pickle**. Escolhemos este mecanismo pois permite passar as imagens de forma muito mais eficiente do que passar as imagens para base 64 e usar o mecanismo default usado pelo **Celery**, o **JSON**. É passado outro argumento, que é nada mais, nada menos que o identificador do vídeo em causa.

O Celery comunica entre o servidor e os workers através de um broker, sendo aquele que escolhemos o **RabbitMQ**. O **RabbitMQ**, como *Message Broker* implementa o **AMQP** (*Advanced Message Queuing Protocol*), protocolo este que permite a utilização de uma (ou mais) message queue(s) com a política *First In First*

Out (**FIFO**), e onde as tarefas são distribuídas pelos workers ativos com a política **Round Robin**, aplicando assim, de certa forma, o protocolo que nós mesmos tínhamos pensado em aplicar antes de utilizar Celery. Além disso, através da utilização de vários workers e utilizando as políticas de Load Balancing enunciadas agora mesmo, a concorrência dos processos é garantida na realização das suas tarefas.

Consideramos também em vez de passar na task a imagem diretamente codificada enviar apenas a referência desta e guardá-la numa pasta local do servidor, e ao processar a task o worker faria um *GET* com essa referência e receberia o frame vindo do servidor por um request. No entanto, não achamos que fosse melhorar o programa em si e dessa maneira também não se retiraria o máximo proveito das funcionalidades que o Celery nos disponibiliza.

No Diagrama a seguir é exemplificado, o funcionamento base de um sistema que implemente Celery (utilizando o RabbitMQ como Message Broker): a cada queue podem estar associados vários Celery Workers, que podem, ou não, guardar os resultados das tasks no *BackEnd*.



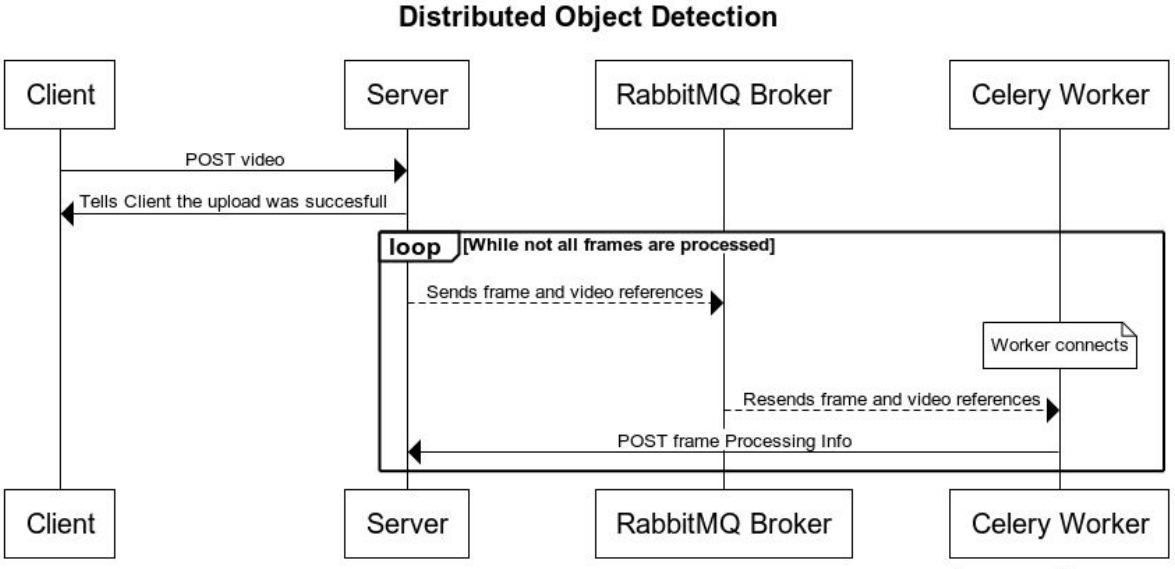
A informação que é devolvida ao servidor consiste num dicionário com as classes encontradas no frame e o número de vezes que cada uma é encontrada, o identificador do frame, o identificador do vídeo e por fim o tempo necessário para o processamento. Para devolver esta informação de volta ao servidor, é usado novamente o protocolo **HTTP**, onde é feito um **POST** para o endpoint *result*. O servidor posteriormente guarda os dados na estrutura de dados ***video_map.***

Através da fila de tarefas assíncronas do Celery e do RabbitMQ como Broker, é possível tornar o servidor resistente a falhas caso um dos workers crash, as tarefas são repostas na fila caso tal se verifique. Como o processamento das frames pode ser realizado por qualquer ordem não nos preocupamos com a prioridade de frames.

Por esta razão também, podemos garantir que quando, num processo, o número de vezes que uma resposta é recebida vinda de um worker com os dados do processamento de um frame for igual ao número de frames totais do vídeo (guardados inicialmente ao receber um upload de um vídeo recorrendo a um método do OpenCV) significa que todas as frames já foram processados e daí todos os alarmes correspondentes impressos na consola.

Mal recebe um pedido do cliente, são enviadas tarefas para quando um worker estiver disponível realizar essas tarefas e também como o servidor utiliza flask multithread não só para lidar pedidos do cliente e do worker nunca fica preso em nenhum momento portanto são evitadas as situações de possível busy-waiting.

No *Message-Sequence Chart,* apresentando abaixo é demonstrado o funcionamento do protocolo usado para a realização deste Trabalho.



Tolerância a Falhas e Redundância

Além disso, recorrendo ao Celery e RabbitMQ, conseguimos tratar da tolerância a Falhas (*Fault* Tolerance), colocando a variável **ACK_LATE** a True, ou seja, as tarefas só irão ser "acknowledged" após a task ser concluída com sucesso. Desta forma, mesmo que qualquer worker falhe, o RabbitMQ irá guardar a mensagem no fim da queue, sendo processada por outro worker(redundância passiva) . Podemos assim, utilizar o Celery como uma espécie de *Fault Tolerant Scheduler.*

Por vezes, por falta de memória no sistema, podem haver algumas possíveis falhas de processamento de frames, dando origem a SIGKILL da tarefa atribuída ao worker, isto foi resolvido recorrendo a retries da tarefa que falhou. Se não fosse feito isto, o worker iria prosseguir para o processamento de outro frame, ignorando o que falhou.

Estruturas:

video_map - dicionário responsável por armazenar para cada vídeo uploaded, possuindo como chave o identificador do vídeo e como valor um **dicionário** com : total de frames deste, ***total***, o número de frames já processadas, ***count***, e dicionário com as classes e o total de ocorrências destas no vídeo, ***classes***. Assim sempre que é processado um frame, o valor de ***count*** é incrementado, e a lista atualizada. Caso o valor de count iguale o número de frames registado para o vídeo em causa, então é impressa no terminal do Servidor as estatísticas deste, que foram requisitadas pelo docente no guião deste trabalho. A utilização de um dicionário, classes, para armazenar os dados das classes e sua frequência nesta estrutura de dados, deve-se ao facto da pesquisa de classes como "person" ser apenas de complexidade O(1), assim evitamos ter de percorrer toda a estrutura. Por outro lado, ocupamos ligeiramente mais memória.

Resultados

Como o Celery por predefinição, corre cada worker com um número de threads equivalente ao número de cores do computador, decidimos correr o(s) processo(s) de worker(s) com concurrency de 1. Assim, apresentamos agora os resultados obtidos com 1,2 e 4 workers ativos, com a função de processamento de frames. Como o vídeo *moliceiro.m4v*, apresentava um número considerável de frames, decidimos cortar o vídeo para 94 frames, com vista a conseguirmos testar melhor os resultados obtidos em menos tempo de execução (pois quantos mais frames, maior o tempo de execução).

1 worker com concurrency 1:

```
Processed frames: 431
Average processing time per frame: 709ms
Person objects detected: 17
Total classes detected: 6
Top 3 objects detected: person, boat, car
```

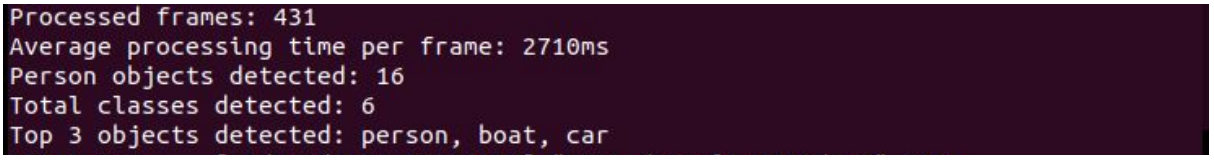
Onde o tempo médio de execução por frame foi cerca de 0.709 segundos, tendo sido precisos, cerca de 5.09 minutos para processar todo o vídeo.

2 workers com concurrency 1:

```
Processed frames: 431
Average processing time per frame: 1345ms
Person objects detected: 17
Total classes detected: 6
Top 3 objects detected: person, boat, car
```

Onde o tempo médio de execução por frame foi cerca 1.345 segundos, tendo sido precisos, cerca de 4.83 minutos para processar todo o vídeo.

4 workers com concurrency 1 :



Onde o tempo médio de execução por frame foi cerca 2.701 segundos, tendo sido precisos, cerca de 4.86 minutos para processar todo o vídeo.

Analisando estes resultados podemos concluir que à medida que se aumenta o número de workers o tempo total de processamento de um vídeo mantêm-se ou diminui pouco (é difícil notar a diferença devido às pequenas variações de tempo de processamento), mas o tempo médio por frame aumenta, devido à carga de trabalho que aumenta no CPU.

Conclusão

Com este trabalho, foi-nos possível adquirir o conhecimento de Libraries externas do Python como Flask, RabbitMQ, Celery que se mostram úteis no desenvolvimento de Sistemas Distribuídos. Além disso, permitiu-nos aprofundar os conhecimentos dados nas aulas teóricas nomeadamente dos temas de Tolerâncias a Falhas, comunicar de forma assíncrona entre processos, threads… entre outros. Também aumentou a nossa capacidade de aprendizagem e adaptação a novas ferramentas.

Referências

https://stackoverflow.com/questions/27070485/initializing-a-worker-with-arguments-using-celery

https://flask.palletsprojects.com/en/1.1.x/

https://docs.celeryproject.org/en/stable/

https://www.rabbitmq.com/