Relatório de Projeto

Computação Distribuída

Distributed Object Detection

Professores:

Diogo Gomes

Nuno Lau

Duarte Mortágua - 92963

Lucas Sousa - 93019

Índice

Introdução

Este projeto foi desenvolvido no âmbito da cadeira Computação Distribuída. Baseia-se em detectar objectos num vídeo de forma distribuída de forma a tornar mais rápido o processamento.

As principais tecnologias utilizadas são **Flask**, **Celery** e **RabbitMQ** - pretende-se evidenciar e contextualizar todos os conhecimentos aplicados ao longo do desenvolvimento do mesmo.

Ferramentas utilizadas

Comunicação por Flask

O Flask permitiu-nos efetuar a troca de mensagens entre o server e os workers. Métodos como o 'POST' e o 'GET' permitiram-nos passar informação facilmente entre ambos.

Task Management e Message Broker por Celery e RabbitMQ

O **Celery** é uma 'task queue' assíncrona. Esta permite-nos criar workers e atribuir-lhes tarefas. O server envia tasks (referências de frames) para a queue, o celery tem workers (cada um com uma task queue) e os workers executam essas tasks (processam frames). Quando há mais do que um worker, as tasks são distribuídas entre eles - ao servidor não interessa qual worker completa a task, apenas lhe interessa o resultado final da sua execução.

Para distribuir as tasks de forma eficaz é necessário usar um message-broker que decidia que worker recebe uma dada task - aqui entra o **RabbitMQ**. Com base no protocolo **AMQP** (Advanced Message Queuing Protocol) o RabbitMQ distribuí as tasks por cada worker.

Descrição da Solução

Upload do vídeo (Cliente) para o Server

O upload do vídeo para o servidor é feito através do comando:

```
curl -F 'file=@moliceiro.m4v' 127.0.0.1:5000/
```

Usando o curl podemos enviar o ficheiro para o servidor Flask para a route raiz (‘/’) e este acede ao conteúdo do POST através da biblioteca de ‘requests’.

Atribuição das tasks

Quando o server recebe o vídeo do client é necessário seguir estes passos:

1. Repartir o vídeo em frames (através do código fornecido).
2. Guardar a frame em .jpg no server em /static para acesso direto.
3. Enviar a referência da frame para o broker (RabbitMQ).

Os Workers

Os workers, ao receberem uma referência de uma frame, fazem o GET de server/static/frame_reference e obtêm a frame. Procedem à deteção de objetos através do código fornecido, sendo que no final elaboram um JSON com as estatísticas dessa frame e enviam o JSON para o servidor/return através de POST. O servidor procede à atualização da estatística global e, se na frame recebida for detectado um número de pessoas maior do que o número de pessoas máximo, imprime essa informação na console do server.

Configurações do Celery[1]

CELERYD_CONCURRENCY - Define o número de processos concorrentes que cada worker pode lançar. O default é o número de cores do processador.
Poderíamos ter implementado o default e lançado apenas 1 worker (que funcionaria como 4), mas optámos por lançar um número variável de workers que lançam apenas 1 processo. Optámos por este caminho uma vez que ter sempre 4 processos a fazer object detection pode ser problemático em termos de memória RAM.

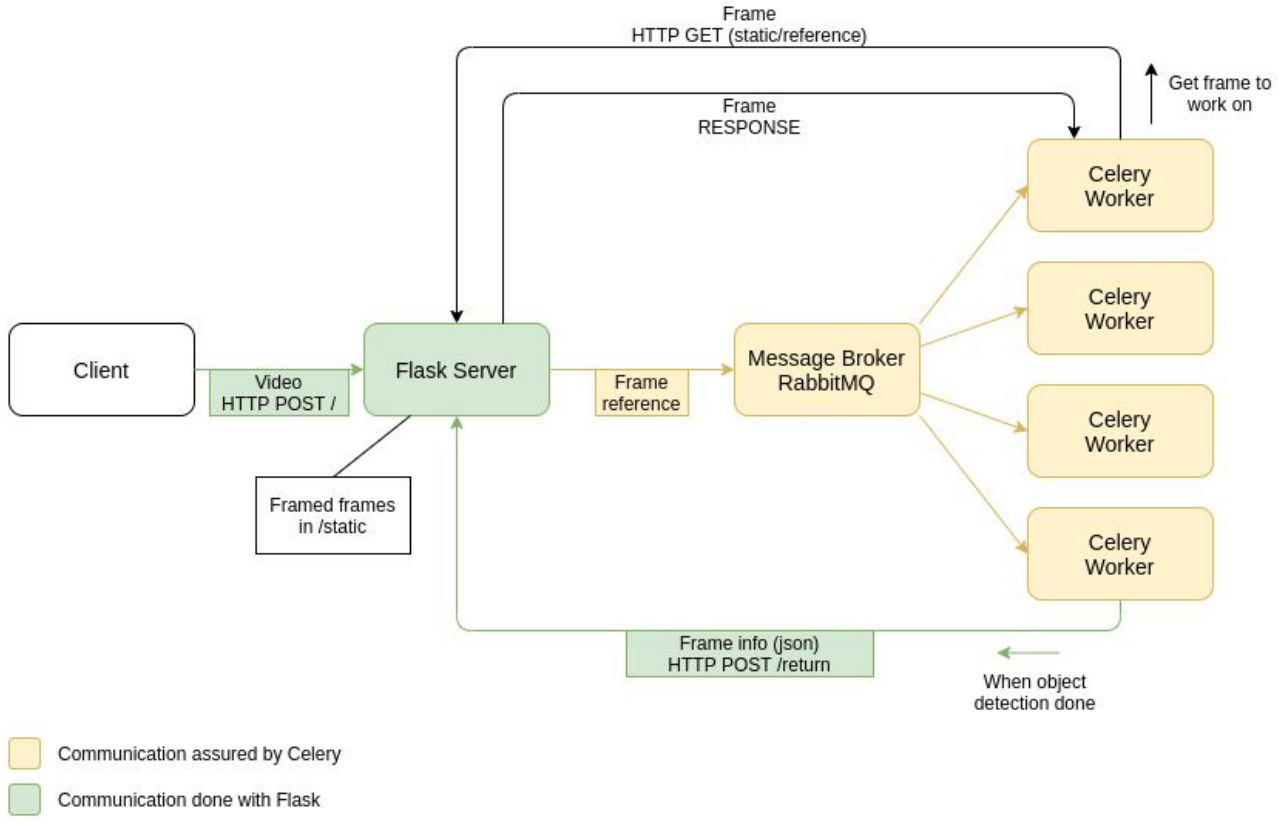
[1] "Configuration and defaults — Celery 4.4.5 documentation." https://docs.celeryproject.org/en/stable/userguide/configuration.html. Data de acesso: 13 jun.. 2020.

Normalmente a detecção de objetos de apenas 1 frame custa cerca de 700MB de memória RAM. Ter sempre 4 processos a correr implicaria ter sempre 4 vezes essa memória RAM consumida em simultâneo, o que pode trazer problemas dependendo da RAM disponível.
Assim, damos a hipótese de lançar menos workers na mesma máquina e não correr o risco de esgotar a memória.

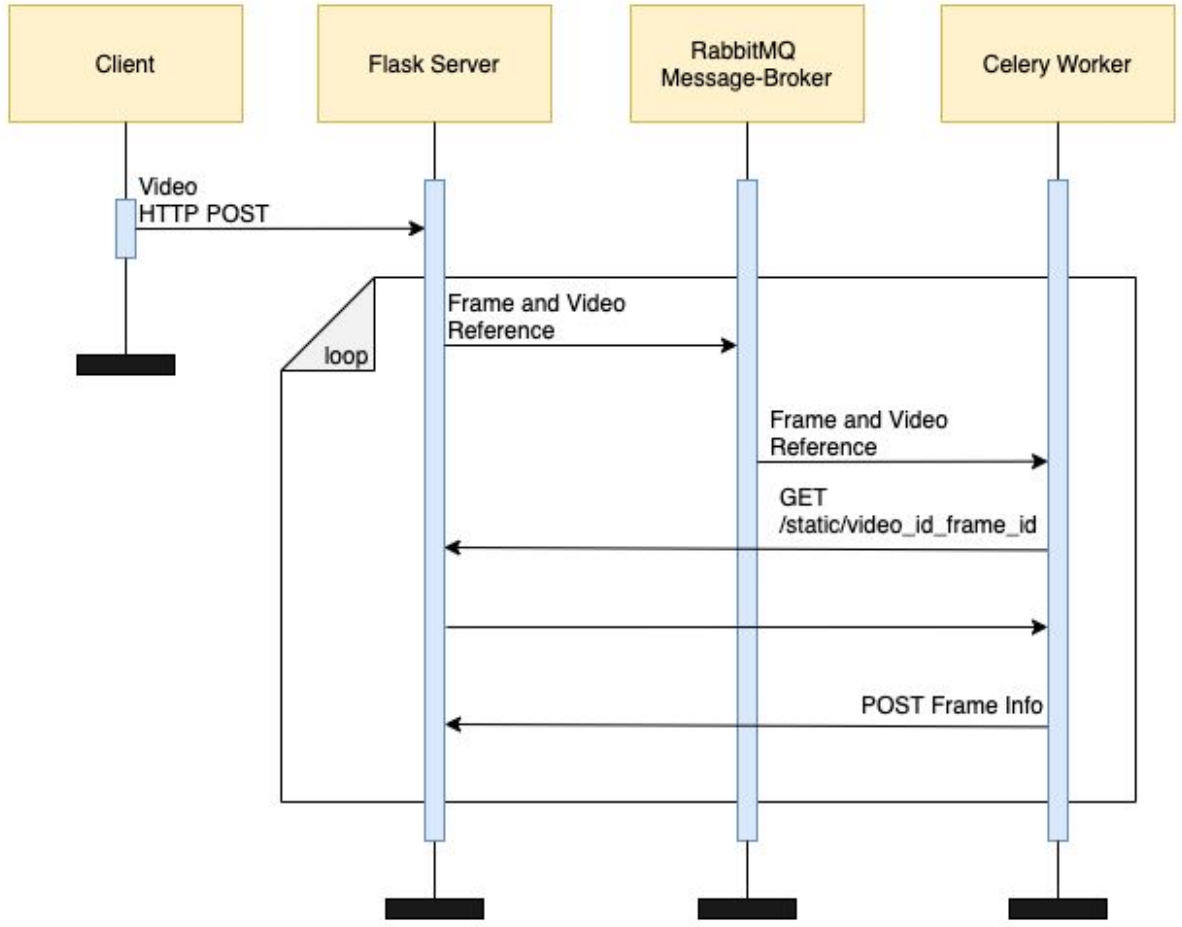
CELERYD_PREFETCH_MULTIPLIER - Esta configuração define quantas tarefas é que um worker pode reservar para si de cada vez.
O default é 4, mas decidimos alterar este valor para 1 uma vez que os nossos workers lançam apenas um processo cada um (mencionado acima).

CELERY_ACKS_LATE - Ver Fault Tolerance*.

Diagrama geral de funcionamento da solução



Message Sequence Chart



Funcionalidades implementadas

Fault Tolerance

De modo a prevenir eventuais falhas dos workers (processo morrer a meio da execução), foi ativada a configuração **CELERY_ACKS_LATE**, que de acordo com a documentação do Celery, previne isso mesmo: *"The acks_late setting would be used when you need the task to be executed again if the worker (for some reason) crashes mid-execution."*[2]

De modo a testar esta configuração, foram submetidas 10 frames para os workers trabalharem e, a meio da execução, executou-se o seguinte comando:

```
ps axjf | grep '[c]elery' | awk '{print $3}' | xargs kill -9
```

É feito o kill através do group ID e não do PID uma vez que o PID dos workers está em constante mudança (comet process), e o group ID mantém-se constante.

Os resultados confirmam o funcionamento da prevenção de falhas, uma vez que os workers foram todos mortos repentinamente, lançados novamente e no final todas as frames foram trabalhadas.

Concorrência de vídeos

A concorrência de vídeos é possível, assim como a concorrência de vídeos com o mesmo nome, sendo que se se carregar duas vezes o mesmo vídeo, e tendo como exemplo o caso do *moliceiro.m4v*, o servidor irá carregar o *moliceiro.m4v*, o *moliceiro1.m4v, moliceiro2.m4v,…* etc.

A concorrência de vídeos é possível uma vez que cada frame é identificada através da lógica *videoID_frameID.jpg*, sendo as frames dos múltiplos vídeos tratadas como independentes pelo message broker.

O servidor guarda um dicionário com keys os video_ID submetidos, e values de cada uma das keys o número total de frames e o número de frames já retornadas. Quando estes dois últimos valores são iguais, significa que o processamento daquele vídeo terminou e imprime-se as estatísticas do mesmo na console do servidor.

[2] "Configuration and defaults — Celery 4.4.5 documentation." https://docs.celeryproject.org/en/stable/userguide/configuration.html. Data de acesso: 13 jun.. 2020.

Testes

Ferramentas auxiliares

- **RabbitMQ Management API (Management page) -** De modo a monitorizar as filas de mensagens, os estados (Ready, Unacked, Total) das mensagens e memória RAM consumida pelo broker em tempo real.
- **Flower - Celery monitoring tool** - De modo a visualizar a pool de workers, a execução das tasks por parte dos workers e de modo a fazer uma monitorização mais detalhada através de gráficos de Succeeded Tasks, Task Times, Failed Tasks e Queue Tasks ao longo do tempo.

Resultados

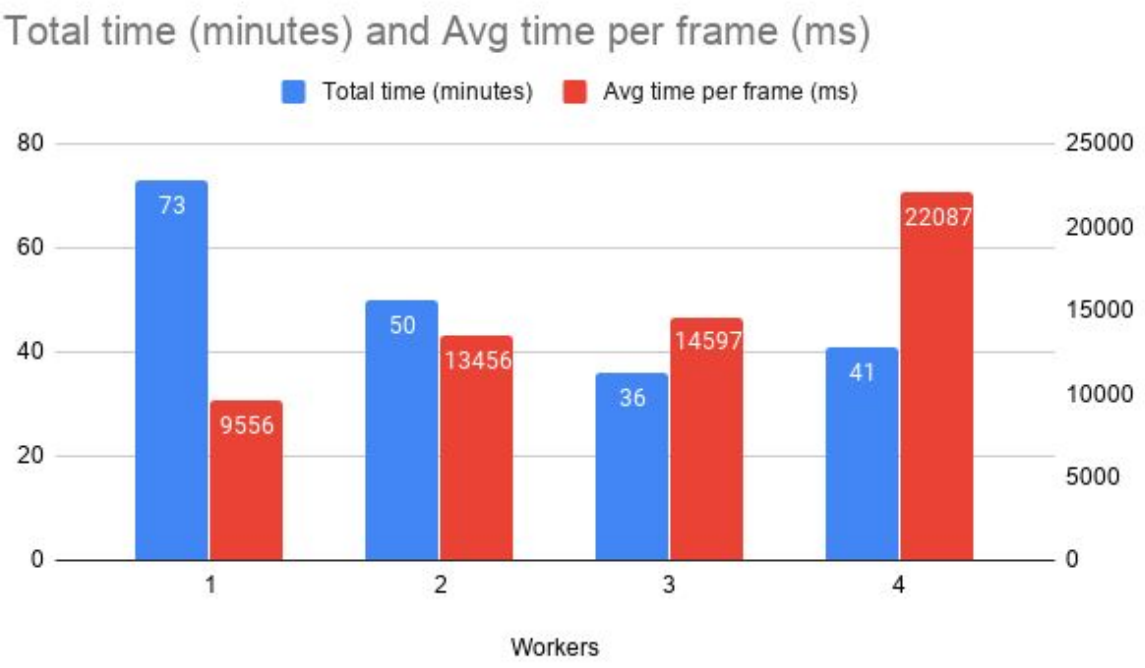
Estatísticas globais

Apresentamos abaixo as estatísticas globais do processamento do vídeo moliceiro.m4v fornecido, trabalhado por 3 workers (melhor resultado).

```
Total time: 0:36:24.383783


Processed frames: 431
Average processing time per frame: 14597ms
Person objects detected: 8557
Total classes detected: 6
Top 3 objects detected: person, boat, car
```

Tempos de Processamento



O gráfico acima refere-se aos tempos de processamento do vídeo *moliceiro.m4v*. Os resultados neste aspeto foram de encontro ao que esperávamos. À medida que adicionamos workers, o tempo de execução diminui de forma constante até certo ponto, isto porque esta concorrência de workers está a ser feita apenas numa máquina, com memória RAM e poder de processamento limitados (ver Memória RAM e CPU). Como resultado disso mesmo, temos o aumento gradual do tempo de processamento por frame e, consequentemente, o aumento do tempo total de execução de 3 para 4 workers, quando num sistema bem distribuído por várias máquinas se esperaria um tempo de processamento médio por frame constante e um tempo total de execução tão mais pequeno como o número de workers.

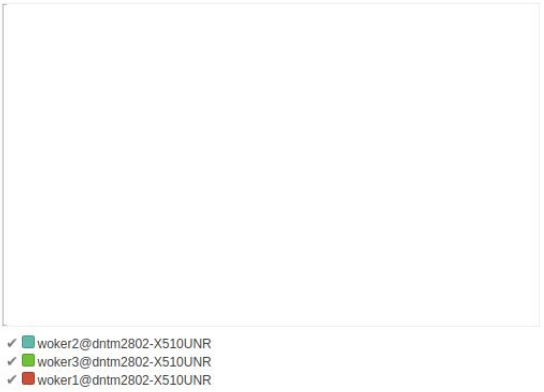
Memória RAM e CPU

Por si só, o Object Detection despende, por frame, entre 700MB-1GB de RAM e de bastante poder de processamento. Naturalmente que correr 4 workers em concorrência, numa máquina limitada a 4 cores e 8GB de RAM como é o nosso caso, limita bastante a capacidade de cada worker, tanto que, de acordo com o gráfico acima, o tempo de processamento por frame aumenta em 51% de 3 workers para 4, e, portanto, não faz sentido utilizar mais do que 3 workers nestas condições.

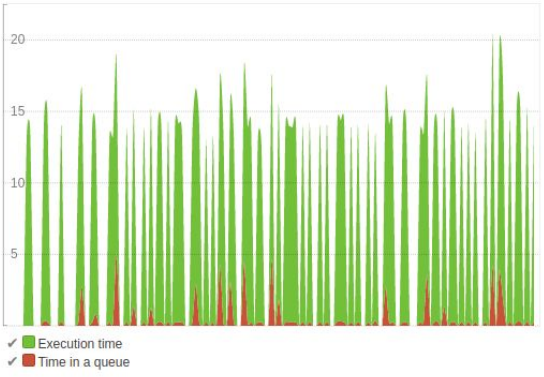
Estatísticas Flower Monitor

Succeeded tasks

Failed tasks



Task times



Queued tasks