Computação Distribuída

Diogo Nuno Pereira Gomes
José Nuno Panelas Nunes Lau

Distributed Object Detection

Hugo Paiva, 93195
Carolina Araújo, 93248

DETI
Universidade de Aveiro
25-06-2020

1 Introdução

Este trabalho visa a criação de um sistema distribuído de deteção de objetos, sendo necessário estabelecer comunicação entre *workers* e o servidor, concebido com base na *framework Flask*, e entre este e os clientes. Apresentar-se-ão, então, as escolhas tomadas relativamente aos protocolos de comunicação em prol do funcionamento do sistema e quais as razões que levaram às mesmas.

2 Protocolo

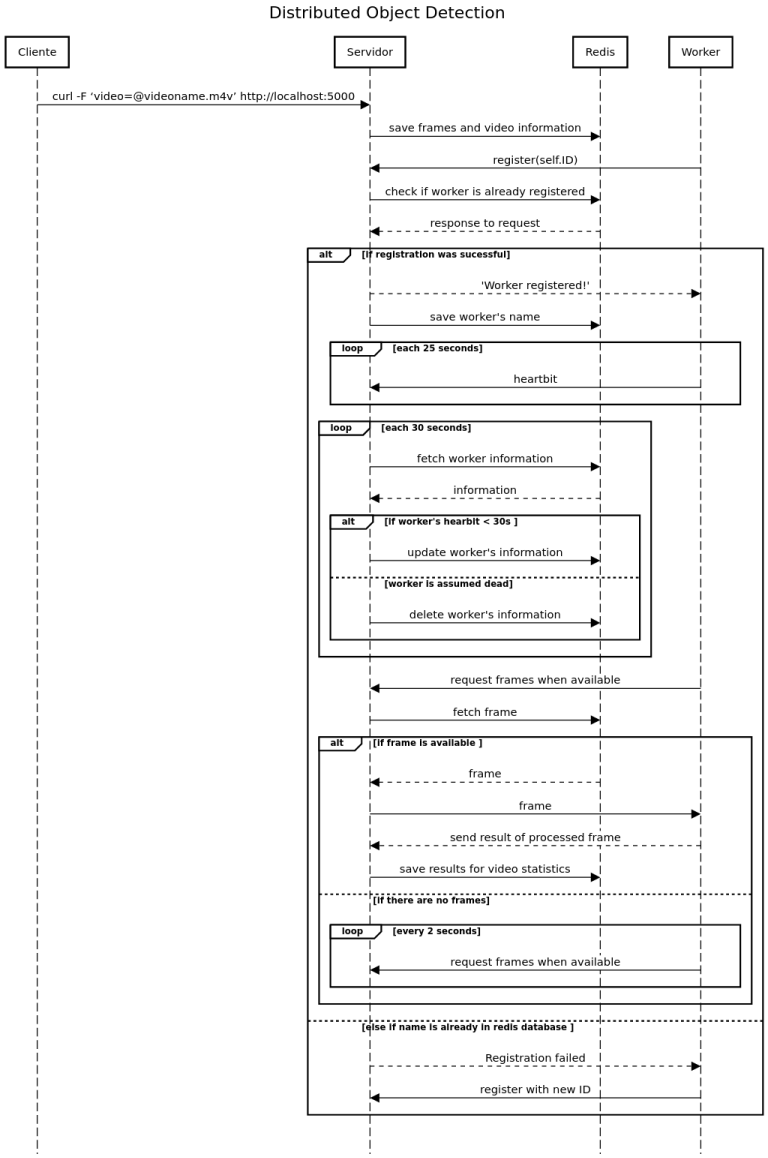

Figure 1: Message Sequence Chart entre as 3 entidades e o Redis

A figura acima retrata o fluxo das mensagens principais trocadas entre as várias entidades que constituem o sistema.

O *cliente* terá de enviar um vídeo para que o servidor possa dividir o mesmo em *frames*. Entretanto, espera-se que os *workers* se registem no servidor para poderem receber tarefas, só depois poderão realmente trabalhar. O servidor comunica maioritariamente com o *Redis*, verificando, introduzindo ou indo buscar dados à base de dados para todas as funções que realiza.

A cada 25 segundos, o *worker*, depois de registado, enviará um *heartbit*, a partir do qual informa o servidor de que ainda se encontra ativo. Já o servidor, a cada 30 segundos, verifica todos os *workers*, eliminando aqueles que se consideram mortos.

Paralelamente, os *workers* estão constantemente a pedir *frames* para processar. Caso haja, ser-lhes-á enviada a mesma e eles devolverão o resultado, se não forem mortos entretanto; caso contrário, apenas repetem o *request* dentro de 2 segundos.

2.1 Estrutura de Dados

De modo a permitir que o servidor seja executado por múltiplos processos, ou até mesmo existirem múltiplos servidores com endereços diferentes, todos os dados necessários para a execução do mesmo são guardados com recurso a ***Redis***, uma base de dados não relacional que guarda os dados na memória principal e que permite o armazenamento de dados de texto ou binários até 512*mb*, cumprindo todos os nossos requisitos. Desta forma, o acesso aos dados continua extremamente rápido.

Para armazenar os dados, foram utilizados os seguintes tipos de dados do *Redis*:

- **Strings** - Utilizado para guardar informações de um vídeo e *worker* após codificação em *JSON*
- **Lists** - Permitiu o armazenamento dos *paths* de cada frame e o *MD5* correspondente ao seu vídeo, também com codificação em *JSON*, e os diversos vídeos que ainda necessitam de processamento (para o *Load Balancer*)
- **Sets** - Para guardar os nomes dos diferentes *workers*, não permitindo duplicação em termos de nomes

2.2 Server

É função do servidor gerir todos os *workers* que se encontram ativos, atribuindo *frames* aos mesmos quando estes se mostram disponíveis para tal. Para além disto, deve ainda processar toda a informação vinda dos *workers* referente às diferentes *frames* dos vários vídeos, para poder alertar a presença excessiva de pessoas e, no final, apresentar as estatísticas de cada vídeo. O servidor pode receber vídeos diferentes de vários clientes, sendo sua função dividi-los depois em *frames*.

2.2.1 Receção e Processamento de Vídeos

A receção de um vídeo é feita utilizando o *endpoint* **/** onde são feitas algumas verificações como a confirmação da extensão do ficheiro, sendo que apenas é suportado *.mp4* e *.m4v*, e a validação da existência de um ficheiro.

Passando estas verificações, é criado um ficheiro temporário utilizando a biblioteca *tempfile* onde é guardado o vídeo recebido para posteriormente ser processado. O processamento foi realizado com recurso a *daemon threads*, algo necessário para garantir a execução do servidor em pleno e para que sejam terminadas juntamente com este, caso seja necessário.

No início do processamento, é gerada uma *MD5 hash* do vídeo com o objetivo de o identificar indubitavelmente. Assim, caso um vídeo igual esteja a ser processado ou já tenha sido processado, o servidor não necessita de lidar com o que acabou de chegar, imprimindo na consola os resultados, caso o outro vídeo igual já tenha impresso

anteriormente. Como referido anteriormente, o processamento do vídeo gera as várias *frames* deste, guardando na base de dados *Redis* o *path* do ficheiro temporário onde se encontram as diferentes frames. No final da divisão em *frames*, são também guardadas na base de dados as informações do vídeo e, posteriormente, colocando-o em lista de espera para os *workers* começarem a trabalhar.

2.2.2 Envio de Frames aos Workers

Acedendo ao *endpoint* ***/worker/<name>/request***, será atribuída, ao respetivo worker, uma frame, caso ainda exista alguma por processar.

O servidor recorre à lista de vídeos por processar, guardada no *Redis*, para ir buscar o identificador de um vídeo que ainda não tenha tido as suas *frames* processadas na totalidade, retirando o *md5* do vídeo que esteja na cabeça da lista. A partir do *md5*, irá buscar a *frame*, também na cabeça da lista de *frames* desse vídeo, que será então retornada. É, portanto, atualizada a informação, *heartbit* e *frame* atual, do próprio *worker*.

Feito isto, se depois da atribuição ainda houver mais *frames* para processar relativas ao vídeo em questão, voltará a colocar-se o identificador desse mesmo vídeo no fim da lista de vídeos por processar. Deste modo, garante-se o processamento de vídeos em concorrência, sendo que um vídeo nunca tem de esperar que outro acabe de processar totalmente para chegar a sua vez na lista.

2.2.3 Receção dos Resultados dos Workers

Cada vez que um *worker* acaba de processar a *frame* que lhe foi atribuída, acede ao *endpoint* ***/worker/<name>/result***.

O servidor liberta a *frame* associada ao *worker*, para que, caso esse mesmo *worker* morra, a *frame* não seja atribuída a um outro, visto que já foi processada. De seguida, o servidor guarda numa lista do *Redis* os resultados para posteriormente serem processados durante a execução de uma *daemon thread*. Esta vai buscar a informação armazenada relativa ao vídeo a que a *frame* pertence e, também, a informação enviada pelo *worker*.

É atualizada toda a informação, desde o tempo total de processamento do vídeo, o número de *frames* já processadas, bem como a contagem de todas as entidades que foram detetadas. Caso o número de pessoas detetadas exceda o limite estipulado, será impressa o número da *frame* em que tal se sucedeu, bem como a quantidade de pessoas que nela se detetaram.

Em termos da informação impressa perante a entrega dos resultados da última *frame* de um vídeo, são apresentadas as estatísticas que os docentes solicitaram.

2.2.4 Manutenção dos Workers ativos

Em qualquer uma das interações entre servidor e *worker*, o servidor atualiza a informação que mantém sobre esse mesmo, para poder, periodicamente, verificar se ele ainda se encontra ativo. No entanto, os próprios *workers* acedem, também de modo periódico (a cada 25 segundos), ao *endpoint* ***/worker/<name>/alive***, para manifestar ao servidor que ainda se encontram vivos.

Na função ***main***, ao ser iniciado o servidor, corre-se pela primeira vez a função ***checkWorkers()*** que, desde então, de 30 em 30 segundos, iniciará uma *thread* cuja função é percorrer a lista de *workers* que se consideram ativos e verificar se isso é ainda, de facto, verdade.

Quando o *worker* se regista com sucesso, é adicionado o seu nome ao *Set* de *workers*, e embora não lhe seja automaticamente atribuída uma *frame*, é atribuído o primeiro *heartbit*, *i.e*, a hora a que o registo sucedeu, sendo então guardada a informação na base de dados. A partir daí, cada vez que a *thread* é chamada, o servidor verifica se, entre a última hora associada a cada *worker* e a hora atual, passaram 30 segundos. Se sim, assume-se que o *worker* morreu e a *frame* que lhe estava associada é colocada novamente na lista de *frames* por processar, mas no início da mesma, uma vez que era suposto já ter sido processada e tem, portanto, mais prioridade. De seguida, apagar-se-ão referências a esse *worker* na base de dados .

2.3 Worker

O *worker* é uma entidade cujas únicas funções, após um contacto inicial com o servidor, são obter *frames*, processá-las, devolver os resultados e manter a comunicação com o servidor ativa.

2.3.1 Registo no Servidor

Ao executar o *worker*, este regista-se no servidor de modo a dar-se a conhecer para permitir a sua tolerância a falhas. Para este registo, é gerado um identificador com o comprimento de 8 caracteres, contendo letras e números aleatórios. Caso, mesmo assim, o identificador já estiver registado no servidor, este gera novamente um novo e tenta realizar novamente o registo ao fazer um pedido *GET* ao *endpoint* ***/worker/<name>***.

2.3.2 Processamento de Frames e Envio dos Resultados

Após o registo com sucesso do *worker*, este faz pedidos de *frames* para processar à medida que esteja disponível. Para isso, são feitos pedidos *GET* ao *endpoint* ***/worker/<name>/request***, os quais são respondidos com o ficheiro correspondente ao *path* guardado em *Redis*, que contém a *frame* que irão processar. Com isto, o *worker* gera os resultados, enviando-os ao servidor através de um pedido *POST*.

Caso o *worker* tenha sido considerado morto pelo servidor, este volta a efetuar o registo, de modo a conseguir pedir novas *frames* para processar. Se porventura o servidor não tiver *frames* para processamento, o *worker* espera 2 segundos para voltar a efetuar o mesmo pedido, repetindo-se a mesma sequência até existir conteúdo para trabalhar.

Para tornar o processamento o mais rápido possível, passou-se a criação do modelo *keras* e a leitura dos pesos para o *scope* global, tendo assim de serem realizados apenas uma vez.

2.3.3 Heartbit

Do mesmo modo que o servidor, o *worker* irá chamar periodicamente uma função, depois de se registar com sucesso, que informa, através do *endpoint* ***/worker/<name>/alive***, que ainda se encontra ativo.

Isto deve-se ao facto de, caso o *worker* tenha sido morto ou tenha sofrido *crash*, a *frame* que lhe estava atribuída tem de ser atribuída a um outro, para que o processamento dos vídeos se mantenha coerente e breve.

Chamando uma *thread* para realizar o envio do *heartbit*, mesmo que o *worker* esteja, de momento, a demorar demasiado tempo a processar a *frame*, é possível avisar o servidor de que ele ainda se encontra ativo.

3 Estratégia de Validação da Solução

Os testes que foram feitos à medida que o projeto ia sendo construído permitiram que o grupo detetasse certas situações que foram então tratadas, como por exemplo: lidar com os casos de tentativas de acesso por parte de *workers* quando o servidor não está ativo, registos falhados, parâmetros incorretos no *curl*, *endpoints* não existentes, ficheiros de tipos inválidos, tentativa de registos sob um nome já existente, *etc*.

Para correr a solução criada, deve-se instalar o *Redis*, seguindo os comandos de instalação do *README.txt*.

3.1 Resultados

O Total Time nos *screenshots* abaixo contabiliza o tempo desde que se envia o vídeo para o server até os resultados finais serem impressos.

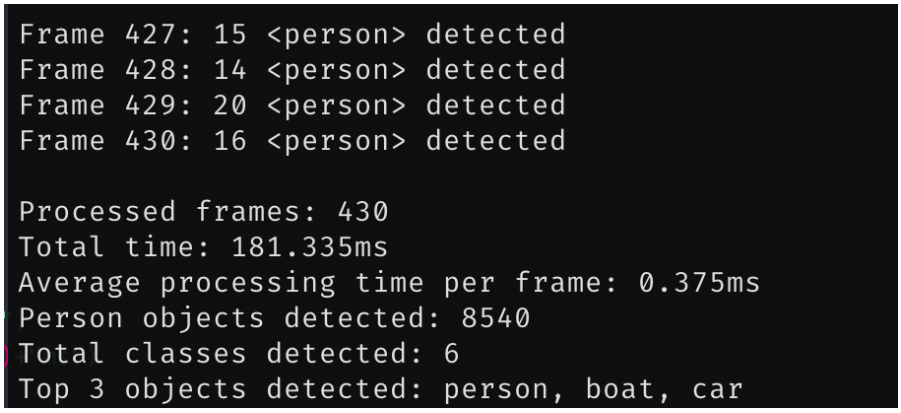

Figure 2: Resultados do processamento do vídeo moliceiros.m4v com 1 *worker*

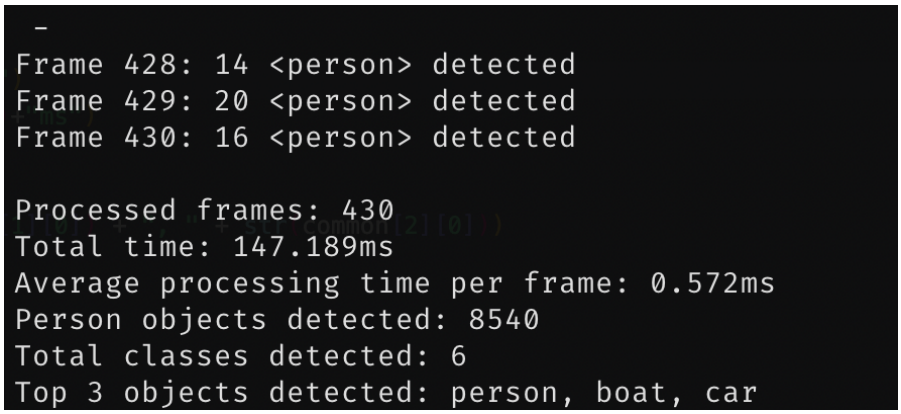

Figure 3: Resultados do processamento do vídeo moliceiros.m4v com 2 *workers*

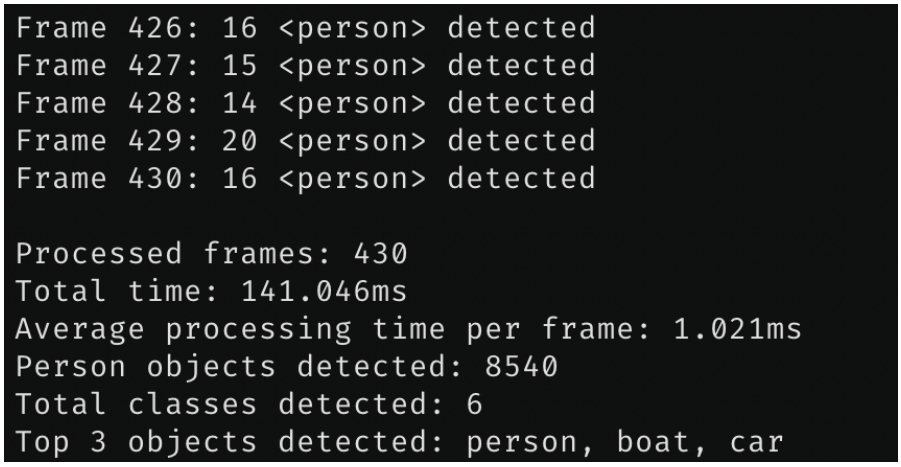

Figure 4: Resultados do processamento do vídeo moliceiros.m4v com 4 *workers*

4 Conclusão

De modo geral, considera-se que a solução criada cumpre todos os requisitos propostos e, até os ultrapassa em algumas situações, como no caso da utilização de um identificador *MD5 Hash*.

Entre as dificuldades encontradas aquando da realização do projeto, está o facto de inicialmente se ter criado um cliente que dividia ele mesmo as *frames*, enviando-as então para o servidor, em vez de um simples *curl*. Isto ocorreu devido a um mal entendido sobre qual o propósito do servidor, tendo-se pensado que se o cliente fizesse a divisão em *frames* dos vídeos, se aliviava carga do servidor, o que por sua vez acelerava o tratamento do vídeo.

Para além disso, inicialmente também foi morosa a pesquisa de quais as ferramentas que se pretendia utilizar, bem como definir a estratégia de comunicação, em prol de um processamento rápido que, ao mesmo tempo, permitisse uma boa gestão dos recursos. Julga-se que as ferramentas utilizadas são aptas para a o objetivo deste trabalho.

5 Referências

[1] https://flask.palletsprojects.com/en/1.1.x/

[2] https://redis.io/documentation

[3] https://redis-py.readthedocs.io/en/stable/

[4] https://requests.readthedocs.io/en/master/

[5] https://docs.python.org/3/library/tempfile.html

[6] https://docs.python.org/3/library/hashlib.html

[7] https://tinyurl.com/y68mthof