



Trabalho Final



Estudantes: 47193 João Arcanjo, A47193@alunos.isel.pt
47256 Diogo Novo, A47256@alunos.isel.pt

Professor: Luís Assunção, luis.assuncao@isel.pt

Relatório realizado no âmbito de Computação na Nuvem

Licenciatura em Engenharia Informática e de Computadores

Semestre de verão 2021/2022

9 de junho de 2022

Índice

1	Introdução	1
1.1	Objetivos	1
1.2	Organização do relatório	2
2	Arquitetura do sistema	3
2.1	Componentes	3
2.2	Fluxo de trabalho	5
3	Implementação da arquitetura	7
3.1	Contrato	7
3.2	Cliente e Servidor <i>gRPC</i>	7
3.2.1	<i>Upload</i> de uma imagem	8
3.2.2	<i>Download</i> de uma imagem anotada	10
3.2.3	Obtenção da lista de objetos detetados numa imagem	11
3.2.4	Obtenção de ficheiros correspondentes a um conjunto de filtros	12
3.2.5	Obtenção de todos os ficheiros com paginação	13
3.2.6	Remoção de um ficheiro do sistema	15
3.3	<i>Cloud Storage</i>	15
3.4	<i>Cloud Pub/Sub</i>	16
3.5	<i>Detect Objects App</i>	18
3.5.1	Armazenamento da informação dos pedidos e resultados na <i>Firebase</i> .	18
3.6	<i>Lookup Function</i>	20
3.7	<i>Monitor Function</i>	21
4	Pontos de falha e Conclusão	23
4.1	Pontos de falha	23
4.2	Testar o sistema	23
4.3	Conclusão	24
Referências		25
A Contrato		26

Listas de Figuras

2.1	Visão geral da arquitetura do sistema	4
2.2	Fluxo de trabalho do sistema	6
3.1	Definição no contrato da operação de submissão de uma imagem	9
3.2	Definição no contrato da operação de <i>download</i> de uma imagem anotada	10
3.3	Definição no contrato da operação de obtenção da lista de objetos detetados numa imagem	11
3.4	Definição no contrato da operação de pesquisa por ficheiros através de filtros	13
3.5	Definição no contrato da operação de obtenção de todos os ficheiros com paginação	14
3.6	Definição no contrato da operação de remoção de um ficheiro do sistema	15
3.7	Organização do serviço <i>Cloud Storage</i>	16
3.8	Esquema do serviço <i>Cloud Pub/Sub</i> no sistema	17
3.9	Exemplo do formato do objeto JSON a enviar para o cliente	20

Acrónimos

GCP Google Cloud Platform

gRPC Google Remote Procedure Call

VM Virtual Machine

URI Uniform Resource Identifier

SQL Structured Query Language

JSON JavaScript Object Notation

TCP Transmission Control Protocol

API Application Programming Interface

UUID Universal Unique Identifier

HTTP Hypertext Transfer Protocol

Capítulo 1

Introdução

Para consolidar os conteúdos, lecionados na unidade curricular de Computação na Nuvem, e com o objetivo de planear e realizar um sistema para submissão e execução de tarefas de computação na nuvem, com requisitos de elasticidade, utilizando de forma integrada serviços da *Google Cloud Platform* para armazenamento, comunicação e computação, nomeadamente, *Cloud Storage*, *Firestore*, *Pub/Sub*, *Compute Engine*, *Cloud Functions* e *Vision API*, foi proposto como trabalho final o desenvolvimento de um sistema, designado por *CN2122TF*, capaz de detetar múltiplos objetos, como por exemplo, bicicletas, cadeiras, quadros, entre muitos outros, em ficheiros de imagem (PNG, JPG, etc), gerando novas imagens anotadas com as zonas onde foram detetados objetos e guardando essa informação numa base de dados *NoSQL*.

Este relatório tem como objetivo dar a entender a arquitetura do sistema e descrever como é que foi implementada, expondo as decisões tomadas, os pressupostos assumidos e as comparações com outras possíveis decisões.

O estudante Diogo Novo teve mais responsabilidade na implementação da transferência de ficheiros, e por sua vez, o desenvolvimento da *Detect Objects App* e *Monitor Function*, enquanto que o João Arcanjo dedicou-se à implementação das consultas à base de dados e também à interação *Lookup Function* - Cliente. Contudo, ambos os estudantes ajudaram-se mutuamente no desenvolvimento das suas partes.

1.1 Objetivos

Os objetivos ou as funcionalidades implementadas no sistema, as quais estão disponíveis para as aplicações cliente através de uma interface *gRPC*, são as seguintes:

- Submissão de uma imagem para deteção de objetos;
- Obtenção da lista de objetos encontrados, através do identificador retornado na submissão da imagem;
- Obtenção da imagem anotada com as zonas onde foram detetados objetos;

- Dado um intervalo de datas, um objeto e um grau de certeza, obter todos os identificadores e nomes de ficheiros armazenados no sistema que correspondem a estes parâmetros.

Além destas funcionalidades obrigatórias, foram ainda implementadas mais duas com o objetivo de praticar e facilitar algumas tarefas:

- Obtenção de todos os ficheiros armazenados no sistema com paginação;
- Remoção de ficheiros do sistema, tanto as imagens armazenadas na *Cloud Storage*, como as suas informações que se encontram na *Firestore*.

1.2 Organização do relatório

Este relatório está organizado da seguinte forma:

- No Capítulo 2 é retratada a arquitetura do sistema e são apresentados alguns detalhes acerca da implementação do mesmo, isto é, informações relevantes acerca das funções mais gerais dos componentes e, também, o fluxo de trabalho do sistema.
- No Capítulo 3 é apresentada a implementação da arquitetura exposta no capítulo anterior, sendo descritas todas as opções tomadas para as implementações das funcionalidades e de cada um dos componentes.
- No Capítulo 4 constam as conclusões sobre o trabalho desenvolvido, pontos de possível falha e informações acerca do que é preciso consultar para testar o sistema desenvolvido.

Capítulo 2

Arquitetura do sistema

Neste capítulo são apresentados mais detalhes e características do sistema de forma a serem concretizados os objetivos anteriormente mencionados. São expostos os componentes da arquitetura que possibilitam a implementação de todos as funcionalidades necessárias e o fluxo de trabalho de todo o sistema. Estes módulos encontram-se descritos na Secção 2.1 e Secção 2.2, respetivamente.

2.1 Componentes

Para que o sistema funcione é necessário compreender todos os componentes que o compõem de uma forma mais geral, desde serviços GCP utilizados, a servidores para manipularem estes serviços e interagirem com os clientes, ilustrados na Figura 2.1, e mais à frente, na Secção 3.1, são descritas com pormenor todas as suas funções e como é que foram implementadas:

- **Cliente** - Responsável por estabelecer a conexão ao servidor *gRPC* e dar início ao processamento de alguma funcionalidade disponível.
- **Lookup Function** - *Cloud function* responsável por fornecer ao cliente os endereços IP disponíveis para se conectar com o servidor *gRPC*.
- **Servidor *gRPC*** - Responsável por receber as mensagens por parte dos clientes e, após processá-las, retornar o resultado obtido. O processamento pode desencadear a publicação de mensagens no serviço *Pub/Sub*, o envio ou descarregamento de imagens do serviço *Cloud Storage* e/ou aceder aos dados persistidos na *Firebase*.
- **Serviço *Pub/Sub*** - É utilizado para a troca desacoplada de mensagens entre os componentes do sistema, nomeadamente, entre o servidor *gRPC*, a aplicação *Detect Objects App* e a *Monitor Function*.
- **Monitor Function** - *Cloud function* com o objetivo de avaliar segundo uma métrica específica, se o número de instâncias de *VMs* do *instance group* associado à *Detect*

Objects App deve ser diminuído, aumentado ou mantido.

- **Cloud Storage** - Encarregue de armazenar as imagens originais submetidas para serem processadas e as imagens anotadas, pós processamento, com os objetos detetados.
- **Detect Objects App** - Aplicação incumbida de processar as imagens submetidas, anotando-as com os objetos encontrados e regista todas as informações necessárias na *Firebase*.
- **Vision API** - Serviço utilizado pela *Detect Objects App* para detetar objetos nas imagens.
- **Firebase** - Corresponde à base de dados *schemaless* onde estará armazenada toda a informação relevante sobre o pedido de processamento realizado e os resultados obtidos.

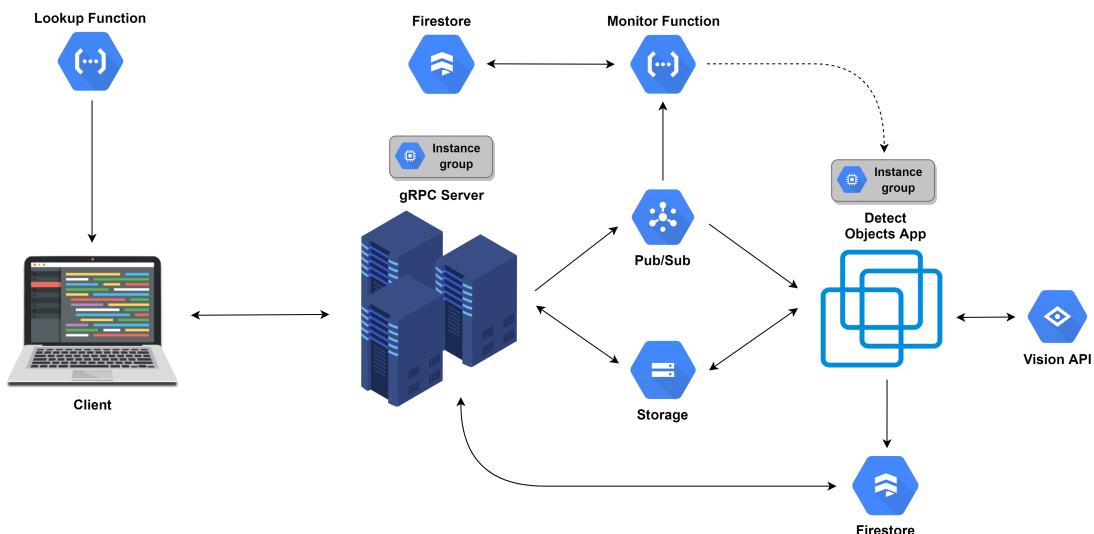


Figura 2.1: Visão geral da arquitetura do sistema

2.2 Fluxo de trabalho

De forma a clarificar todo o processo, abaixo está descrito o fluxo de trabalho do sistema, cuja numeração corresponde à mesma utilizada na Figura 2.2, de forma a ilustrar melhor o seu funcionamento desde a primeira imagem enviada por um cliente até ao envio do resultado através do servidor *gRPC*.

1. Inicialmente um cliente estabelece ligação com um servidor *gRPC*, escolhendo um dos endereços IP disponibilizados pela *Lookup Function*.
2. Após conectado ao servidor *gRPC*, o cliente irá enviar uma imagem para ser processada e serem detetados objetos presentes na mesma.
3. O servidor *gRPC* ao receber a mensagem referente ao método invocado e os seus argumentos, começa o processamento do pedido realizado. Que, para esta funcionalidade, o processamento diz respeito à receção dos blocos de *bytes* contendo os meta dados e o conteúdo da imagem. À medida que os vai recebendo vai imediatamente escrevendo-os/enviando-os para a *Cloud Storage*, onde a imagem irá ficar armazenada.
4. Após a imagem ter sido armazenada, irá ser retornado para o cliente um identificador único e estável associado ao pedido e, por sua vez, à imagem, que deverá ser utilizado para, posteriormente, realizar consultas acerca dos resultados obtidos no processamento da imagem submetida. Seguidamente, irá também ser publicada uma mensagem, através do serviço *Pub/Sub*, de forma a que tanto a *Monitor Function* como a *Detect Objects App* consumam a mensagem e fiquem informadas da chegada de uma nova imagem para processar.
5. A *Detect Objects App*, na mensagem que consumiu, irá obter a informação do *bucket* e do *blob* onde a imagem para serem detetados objetos se encontra. Desta forma, poderá obtê-la e fornecê-la à *Vision API* para realizar o seu processamento.
6. Ao mesmo tempo, a *Monitor Function* através de uma métrica, que será descrita mais adiante, irá perceber se deve aumentar, diminuir ou manter o número de instâncias de máquinas virtuais associadas ao *instance group* responsável por correr a *Detect Objects App*, obtendo e alterando valores necessários da base de dados *Firebase* de modo a conseguir realizar os cálculos necessários.
7. Quando tiver terminado o processamento da imagem, a *Detect Objects App*, irá enviar a imagem anotada com os objetos identificados para a *Cloud Storage* e também irá publicar os resultados do processamento na *Firebase* em conjunto com outras informações associadas ao pedido realizado.

8. Prontamente, quando terminarem as escritas, o cliente já poderá, através do identificador que recebeu quando enviou a imagem, consultar os resultados que foram obtidos e também transferir a imagem anotada.

Todo o tipo de funcionalidade de um componente ou de um serviço, como o tipo ou o formato da mensagem que está a ser enviada, consumida ou armazenada, que não foi especificado no fluxo de trabalho, será exposto e descrito no próximo Capítulo, onde todos estes temas serão detalhados.

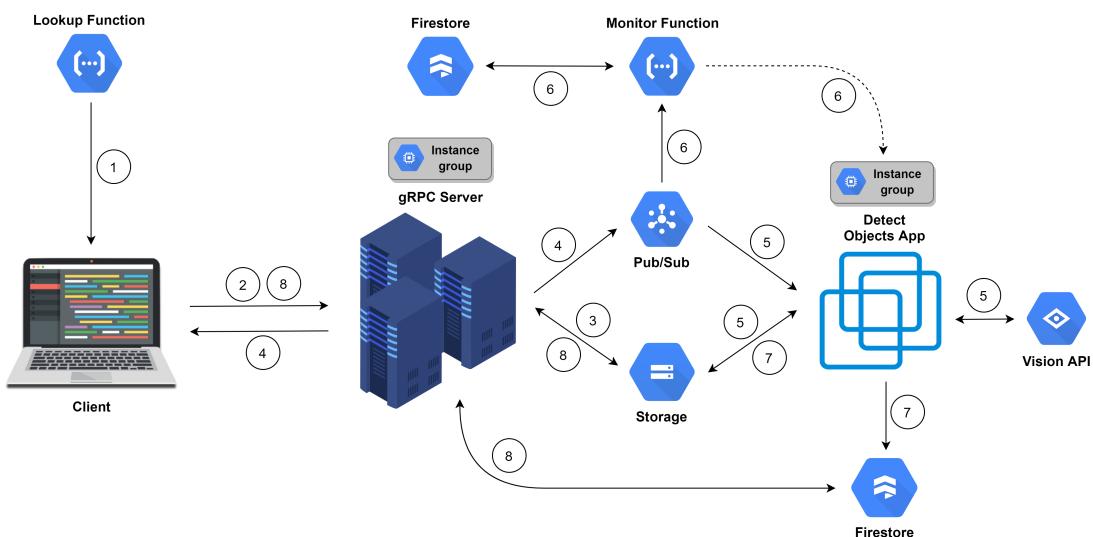


Figura 2.2: Fluxo de trabalho do sistema

Capítulo 3

Implementação da arquitetura

Apresentada toda a arquitetura e os objetivos do sistema, neste capítulo descrevem-se os detalhes e as decisões tomadas na implementação das funcionalidades e dos componentes referidos no capítulo anterior.

3.1 Contrato

Para que fosse possível existir comunicações entre o cliente e o servidor *gRPC* foi definido um contrato/interface *gRPC* através da linguagem *protocol buffers (protobuf)*. Este contrato apresenta as definições dos serviços/funcionalidades que o sistema disponibiliza, ficando também definido os argumentos e o que retornam nas suas respostas. Para que tudo funcione como esperado e os intervenientes (cliente e servidor) consigam trocar mensagens com sucesso, este contrato tem de ser respeitado por ambos.

O contrato elaborado, dado à sua dimensão, pode ser consultado na íntegra no final deste relatório, no Apêndice A.

3.2 Cliente e Servidor *gRPC*

Tal como foi referido na secção anterior, o cliente e o servidor *gRPC* têm de implementar o contrato definido para que seja possível trocar mensagens entre ambos.

O **cliente**, não usando os serviços da plataforma *GCP*, é o componente que irá realizar o ponto de entrada no sistema, o qual ao efetuar um pedido de uma operação ao servidor, irá despoletar seguidamente um conjunto de ações/procedimentos por parte de outros módulos do sistema. Mas, para que seja possível conectar-se com um servidor terá, primeiramente, que obter os endereços IP disponíveis do *instance group* responsável pelo servidor *gRPC*. Para tal, irá utilizar os serviços da *Lookup Function*, que irá retornar uma resposta para o cliente no formato JSON, de forma a que seja possível mapear a representação retornada para o objeto de domínio correspondente, com o objetivo de facilitar a introdução destes argumentos, bastando ao utilizador que está a operar o cliente, introduzir apenas o número correspondente ao endereço IP a que se deseja conectar, não havendo a necessidade de copiar o endereço IP e

seguidamente colá-lo, sendo desta forma muito mais prático. Ao estar conectado ao servidor *gRPC*, o cliente já poderá executar as funcionalidades disponíveis, inserindo o número daquela que pretende realizar e preenchendo os parâmetros necessários.

O servidor *gRPC* corresponde ao intermediário de todas as operações do sistema, o qual já irá utilizar os serviços da plataforma GCP, nomeadamente, os serviços: *i) Cloud Storage* *ii) Cloud Pub/Sub*; e *iii) Firestore*. Após receber as mensagens vindas dos clientes, irá processá-las e retornar os resultados obtidos. O processamento poderá desencadear a publicação de mensagens através do serviço *Pub/Sub*, no caso de ser uma submissão de uma imagem, o envio ou descarregamento de imagens a partir do serviço *Cloud Storage*, e/ou consultar os dados persistidos na base de dados *schemaless Firestore*.

Uma vez apresentadas as funções quer do cliente quer do servidor *gRPC*, nas próximas subsecções serão descritas todas as funcionalidades implementadas e as consequentes decisões tomadas.

3.2.1 *Upload* de uma imagem

A submissão de uma imagem para ser processada e anotada com zonas onde foram detectados objetos é a operação principal do sistema, uma vez que sem ela não existe razão para os restantes componentes exercerem as suas funções.

Esta funcionalidade corresponde, dentro dos modelos de interação cliente/servidor, ao caso 3, isto é, à chamada com *streaming do cliente*, uma vez que o cliente terá de enviar o conteúdo de uma imagem em *stream* de blocos. Visto que a capacidade máxima de um pacote TCP/IP são 64K [1], e os *headers* podem ocupar no máximo 120 *bytes* (60 *bytes* do IP e do TCP) [2], optámos por realizar o envio em *stream* de blocos de 32K, uma vez que caso utilizássemos 64K iria existir fragmentação dos dados a serem enviados, uma vez que só estariam disponíveis, no pior dos casos, 64K - 120 *bytes* (dimensão dos *headers*), problema que apenas aconteceria para imagens com dimensão superior a estes valores.

Para o envio das mensagens para o servidor, tendo em conta que se trata do terceiro caso, foi utilizado um *stub não bloqueante*, a partir do qual o cliente irá chamar o método correspondente, nomeadamente, *uploadImage*, para não só receber o *stream observer* do servidor para o qual deve enviar a *stream* de blocos, como também enviar para o mesmo o seu *stream observer* para, após ter sido enviada a imagem na sua totalidade, receber a devida resposta. Ao não ser bloqueante, enquanto a imagem é enviada, o cliente pode realizar outras operações, sem interferir com o envio da imagem para o servidor.

Primeiramente, irá ser enviado para o servidor apenas os meta dados associados à imagem, com o objetivo de estes serem enviados apenas uma vez e o servidor poder verificar se estão em concordância com os seus requisitos, isto é, se o tipo da imagem e a sua dimensão são válidos, e, de seguida, é que serão enviados os blocos de 32K correspondentes ao conteúdo da mesma. Para isto ser conseguido, no contrato, a mensagem *ImageUploadDownload*, que

é o tipo utilizado no argumento da definição do método *uploadImage*, implementa o campo ***oneof***, para que numa instância sejam enviados os meta dados e noutra os conteúdos da imagem (Figura 3.1).

```
// Upload an image
rpc uploadImage(stream ImageUploadDownload) returns (ImageResponse);

message ImageUploadDownload {
    oneof request {
        ImageMetadata metadata = 1;
        bytes content = 2;
    }
}

message ImageResponse {
    string id = 1;
    string name = 2;
    int32 objectsFound = 3; // Optional field
}
```

Figura 3.1: Definição no contrato da operação de submissão de uma imagem

Caso surja algum erro ou caso alguma validação não se verifique, a partir do método *onError* da interface *StreamObserver* e utilizando a classe *StatusException*, será enviado tanto para o cliente como para o servidor (dependendo do componente onde ocorreu o problema), o *Status* e a mensagem correspondente ao erro que foi despoletado.

Quando o envio da imagem tiver terminado, o servidor irá enviar para o cliente um **identificador único e estável**, nomeadamente, um UUID, associado ao pedido, e, por sua vez, à imagem submetida, que poderá ser utilizado para consultar os resultados obtidos após o processamento da mesma, tal como obter a imagem anotada com as zonas onde foram detetados objetos, obter a lista de objetos detetados ou também efetuar a remoção dos resultados obtidos do sistema.

O **servidor**, tal como já foi referido, à medida que for recebendo os meta dados e o conteúdo, irá realizar as devidas verificações e retornar os erros que aconteçam para o cliente. Ao receber os meta dados, caso não haja nenhum problema, irá gerar um UUID, definir o nome do *blob* onde irá ficar armazenada a imagem e criar o canal de escrita para que, conforme cheguem os blocos com o conteúdo da imagem, possam ser logo enviados/escritos na *Cloud Storage*. Concluído o envio da imagem, isto é, quando for enviado um *onCompleted* por parte do cliente, o canal de escrita será fechado, será enviado para o cliente o identificador criado associado ao pedido e, por fim, será publicada uma mensagem no serviço *Pub/Sub* com a informação do *bucket*, do *blob*, do nome e tipo da imagem para que seja possível os outros componentes do sistema serem notificados da existência de trabalho para processar, nomeadamente, a *Monitor Function* e a *Detect Objects App*.

Mais informações acerca da forma como as imagens são armazenadas na *Cloud Storage* e acerca da criação de tópicos, subscrições e formato das mensagens publicadas no serviço *Pub/Sub*, pode ser consultada na Secção 3.3 e Secção 3.4, respetivamente.

3.2.2 Download de uma imagem anotada

Após uma imagem ser processada e armazenada na *Cloud Storage*, esta funcionalidade permite ao cliente obter a imagem anotada, armazenando-a na diretoria fornecida.

A sua implementação é praticamente o inverso do que foi realizado e mencionado para a operação referente à submissão de uma imagem. O modelo de interação cliente/servidor utilizado diz respeito ao caso 2, ou seja, ao caso de **streaming do servidor**, já que desta vez é o servidor a enviar em *stream* de blocos a imagem desejada para o cliente. Cada bloco enviado tem, também, a dimensão máxima de 32K e a técnica utilizada para o envio da informação também é a mesma, através da mensagem *ImageUploadDownload* e do campo *oneof* definidos no contrato, será primeiro enviado para o cliente os meta dados da imagem e, posteriormente, os blocos do conteúdo da mesma.

Caso surja algum erro no envio dos blocos, utilizando a mesma técnica referida para a submissão de uma imagem, será enviado para o cliente através do método *onError*, da interface *StreamObserver*, uma mensagem indicando, resumidamente, o problema que ocorreu. Dois problemas típicos que podem ocorrer são: *i*) ser inserido um identificador de uma imagem que não existe no sistema; *ii*) a imagem anotada, referente ao identificador enviado para o servidor, ainda não se encontrar disponível, uma vez que estará ainda a ser processada. Cada um destes problemas, ao ocorrem, irá despoletar o envio de uma mensagem específica para o cliente com o erro detetado.

Enquanto o servidor envia os blocos, de forma a que o cliente não fique bloqueado à espera da sua conclusão e possa realizar outras operações, foi utilizado um **stub não bloqueante** para realizar a chamada ao método responsável por dar início ao *download* da imagem desejada, denominado no contrato por *downloadAnnotatedImage* (Figura 3.2).

```
// Get the original image annotated with the zones where the objects were found
rpc downloadAnnotatedImage(ImageIdentifier) returns (stream ImageUploadDownload);

message ImageIdentifier {
    string id = 1;
}

message ImageUploadDownload {
    oneof request {
        ImageMetadata metadata = 1;
        bytes content = 2;
    }
}
```

Figura 3.2: Definição no contrato da operação de *download* de uma imagem anotada

No **cliente**, ao chegar a mensagem com os meta dados da imagem, é preparada a criação de um ficheiro (caso ainda não exista nenhum com o mesmo nome), com a opção *APPEND*, que irá fazer com que os *bytes* que vão sendo recebidos sejam escritos no final do ficheiro

em vez de serem no início. O ficheiro imagem irá ter como nome a conjunção do nome e do tipo vindos na mensagem e estará disponível na diretoria passada na linha de comandos pelo utilizador. À medida chegam os blocos com o conteúdo, a sua escrita vai sendo logo realizada no ficheiro criado. Quando o envio por parte do servidor chegar ao fim e for chamado o método *onComplete*, o ficheiro será fechado e o utilizador na aplicação cliente receberá uma mensagem indicando que a imagem já se encontra disponível na respetiva diretoria.

3.2.3 Obtenção da lista de objetos detetados numa imagem

Em qualquer momento, a partir da altura em que as imagens anotadas são armazenadas, o utilizador da aplicação cliente poderá consultar os objetos que foram detetados nas mesmas. Para tal, basta enviar o identificador da imagem que pretende obter a lista de objetos detetados.

Uma vez que esta operação se trata de uma consulta na base de dados *Firebase*, e deverá, idealmente, obter uma resposta rápida em comparação com o *upload* ou *download* de uma imagem, foi utilizado um ***stub* bloqueante**, ficando, desta forma, a aplicação cliente bloqueada enquanto não receber a resposta por parte do servidor. Relativamente ao tipo de chamada, foi utilizado o caso 1, ou seja, a **chamada unária**, já que só irá existir um pedido por parte do cliente e uma resposta, com os resultados obtidos, por parte do servidor.

Tal como no *download* de uma imagem anotada, para obter os resultados referentes aos objetos encontrados, o cliente terá que fornecer ao servidor o identificador da imagem desejada. Após ter obtido os resultados necessários, irá enviar para o cliente a mensagem *ImageObjects*, representada na Figura 3.3, a qual inclui o identificador da imagem, o nome da mesma e um mapa, cujas chaves correspondem ao nome do objeto encontrado e os valores ao número de vezes que cada um foi detetado.

```
// Get the list of objects found in the desired image
rpc getImageDetectedObjects(ImageIdentifier) returns (ImageObjects);

message ImageIdentifier {
    string id = 1;
}

message ImageObjects {
    string id = 1;
    string imageName = 2;
    // Key = objectName, Value = times that the object was detected
    map<string, int32> objectsNames = 3;
}
```

Figura 3.3: Definição no contrato da operação de obtenção da lista de objetos detetados numa imagem

Da mesma forma que na funcionalidade mencionada na subsecção anterior, caso a imagem com o identificador fornecido não exista, será retornado no método *onError*, uma mensagem com a informação correspondente. Do lado do cliente, quando, eventualmente, surgir um erro do género, uma vez que não é enviado nenhum *StreamObserver* explicitamente para o servidor,

a mensagem será apanhada através de um *try/catch* da exceção *StatusRuntimeException*.

O servidor, para obter os resultados referentes ao pedido realizado, começa por obter o documento, cujo identificador corresponde àquele passado por parâmetro pelo cliente, com o objetivo de verificar se o identificador providenciado está, realmente, associado a uma imagem. E, caso se verifique, realiza uma *query* à coleção dos objetos detetados (*DetectedObjects*) utilizando a premissa *whereEqualTo*, para, desta forma, obter todos os documentos que estão associados ao identificador desejado, através do campo *requestId*. Posteriormente, utilizando os recursos fornecidos pela API da *Firebase*, cada documento recolhido a partir da *query*, é mapeado para o objeto de domínio *DetectedObject*, e o objeto detetado, que o documento representa, é adicionado ao mapa que será retornado ao cliente. Caso, mais tarde, se queira adicionar o grau de certeza ou os vértices associados à deteção do objeto, basta apenas colocar estas propriedades na mensagem de retorno do contrato, e no servidor adicioná-los à mensagem correspondente, uma vez que todas estas informações já se encontram disponíveis no objeto mapeado.

As informações acerca das coleções e formato dos documentos podem ser consultadas na Subsecção 3.5.1.

3.2.4 Obtenção de ficheiros correspondentes a um conjunto de filtros

Esta funcionalidade permite obter todos os ficheiros armazenados entre duas datas fornecidas, os quais incluem o objeto desejado com um grau de certeza acima daquele incluído no pedido. Por exemplo, obter todos os ficheiros entre 01/06/2022 e 02/06/2022, os quais incluem o objeto *Bicycle* com um grau de certeza superior a 0,85.

Todas as funcionalidades que realizam apenas consultas à base de dados *Firebase*, foram implementadas através de uma chamada unária (caso 1) e com um *stub bloqueante*. As justificações para esta decisão são as mesmas às apresentadas na subsecção anterior (Subsecção 3.2.3).

Para serem obtidos resultados com os filtros pretendidos, o cliente utiliza a mensagem *SearchProperties*, para enviar para o servidor todas as propriedades necessárias a utilizar na pesquisa. O formato desta mensagem está representado na Figura 3.4, na qual pode ser observada a presença de dois campos *timestamp* referentes à data inicial e final fornecida. Foi utilizado este tipo para representar a data no contrato devido ao facto de na *Firebase* serem armazenados apenas *timestamps*, e, desta forma, podermos tirar partido deste formato. Um problema que poderia surgir, caso fossem fornecidas apenas as datas, seria por exemplo, supondo que existem dois ficheiros que foram armazenados no mesmo dia, mas a horas diferentes, um de manhã e outro à noite, e as datas inseridas para pesquisar eram iguais, ou seja, eram ambas do mesmo dia, apesar de estarem a ser utilizadas apenas datas para fazer a pesquisa, iria ser acrescentada às mesmas um horário por *default*, resultando em que, ao usar as datas iguais para pesquisar ficheiros do mesmo dia, só iria ser possível apanhar

os ficheiros cujo *timestamp* seria igual ao das datas. Para combater este problema, foram utilizados *timestamps*, mas cada um representando horários distintos. O *timestamp* referente à data inicial representa o início do dia da data inserida, ou seja, apresenta as horas: 00 horas, 00 minutos, 00 segundos e 000 nano segundos, enquanto que o *timestamp* da data final representa o fim do dia da data inserida, isto é, às 23 horas, 59 minutos, 59 segundos e 999 nano segundos. Desta forma, nunca irá existir o problema referido.

```
// Get the stored files between two dates, which contain a specific object with a score greater than 't'
rpc searchForFiles(SearchProperties) returns (FilesResponse);

message SearchProperties {
    google.protobuf.Timestamp initialTimestamp = 1;
    google.protobuf.Timestamp lastTimestamp = 2;
    string objectName = 3;
    double score = 4;
}

message FilesResponse {
    repeated ImageResponse responses = 1;
}

message ImageResponse {
    string id = 1;
    string name = 2;
    int32 objectsFound = 3; // Optional field
}
```

Figura 3.4: Definição no contrato da operação de pesquisa por ficheiros através de filtros

O servidor, após receber as propriedades de pesquisa, realiza uma *query* à coleção *DetectObjects*, para obter os documentos/objetos que correspondem aos filtros desejados, ou seja, se têm um *timestamp* maior ou igual àquele que representa a data inicial (*whereGreaterThanOrEqualTo*), um *timestamp* menor ou igual àquele que representa a data final (*whereLessThanOrEqualTo*) e se o objeto corresponde ao inserido (*whereEqualTo*). Uma vez que a *Firebase API* não permite a pesquisa utilizando premissas iguais àquelas já realizadas, só foi possível realizar a filtragem pelo grau de certeza após a obtenção dos documentos que cumpriam as propriedades acima mencionadas. À medida que é verificado o grau de certeza de documento em documento, uma vez que caso um ficheiro tenha pelo menos dois objetos iguais com o nome desejado, será retornado o seu documento duas vezes, para prevenir tal situação, foi utilizado um *HashSet* para ir armazenando os identificadores dos ficheiros já analisados.

Após todos os documentos processados, será retornada a mensagem *FilesResponse* para o cliente, observável na Figura 3.4, que irá representar uma lista de mensagens *ImageResponse*.

3.2.5 Obtenção de todos os ficheiros com paginação

Esta operação foi realizada com o objetivo de serem apresentados todos os ficheiros armazenados sem haver necessidade de inserção de nenhuns filtros, sendo bastante útil para obter rapidamente o identificador referente a algum documento. A implementação desta funcionalidade é basicamente a mesma que foi realizada para a operação descrita na subsecção

anterior, com a diferença que já não é necessário filtrar os documentos, uma vez que o objetivo é obter todos os existentes. Uma vez que pode existir uma grande quantidade de documentos armazenados, foi implementado um pequeno mecanismo de paginação, para que não sejam retornados todos os ficheiros de uma vez.

Tal como foi referido, esta funcionalidade ao ser idêntica à da pesquisa de ficheiros por filtros, é implementada também através de uma **chamada unária** (caso 1) e um **stub bloqueante**. O cliente fornece ao servidor a paginação através da mensagem *Pagination* (Figura 3.5), que apresenta dois campos, *limit* para saltar um determinado número de ficheiros e *offset* para obter o número máximo de ficheiros que este campo indicar. Para efeitos de teste, foi colocado um *limit* de 10 ficheiros.

```
// Gets all the available documents within a limit
rpc getAllFiles(Pagination) returns (FilesResponse);

message Pagination {
    int32 limit = 1;
    int32 offset = 2;
}

message FilesResponse {
    repeated ImageResponse responses = 1;
}

message ImageResponse {
    string id = 1;
    string name = 2;
    int32 objectsFound = 3; // Optional field
}
```

The diagram illustrates the flow of the code. It starts with the `getAllFiles` RPC definition. An arrow points down to the `Pagination` message definition. Another arrow points down to the `FilesResponse` message definition. A final arrow points down to the `ImageResponse` message definition.

Figura 3.5: Definição no contrato da operação de obtenção de todos os ficheiros com paginação

O mecanismo de paginação foi implementado no cliente, e foi construído através de um ciclo infinito que só termina quando o utilizador já não quiser obter mais páginas ou quando já não houver mais páginas para serem obtidas. Para ser identificado que já não existem mais páginas, em todos os pedidos, é acrescentado ao *limit* mais um valor, ou seja, $10 + 1$ ficheiros que podem ser obtidos. Desta forma, caso haja, pelo menos, mais um ficheiro, será exibida a possibilidade de ser consultada mais uma página.

Do lado do servidor, a pesquisa é feita sobre a coleção *Requests* e na *query* é tirado partido das funcionalidades da *Firebase API*, que fornece interrogações específicas para paginação com os mesmos nomes dados aos campos da mensagem *Pagination*, ou seja, *limit* e *offset*. Obtidos os documentos, é construída a lista de mensagens *ImageResponse* e enviada a resposta para o cliente através da mensagem *FilesResponse* (Figura 3.5), tal como na operação da subsecção anterior.

3.2.6 Remoção de um ficheiro do sistema

Com o objetivo de facilitar a remoção de todos os recursos armazenados referentes a uma imagem, quer na *Cloud Storage*, quer na *Firebase*, foi desenvolvida esta operação.

Tal como as anteriores, é implementada através de uma **chamada unária** (caso 1) e um **stub bloqueante** pelas mesmas razões.

O servidor ao obter o identificador da imagem, obtém os documentos associados à mesma, tanto da coleção *Requests* como da coleção *DetectedObjects*, caso, realmente, existam, caso contrário, envia a mensagem de erro apropriada para o cliente, através do método *onError*, como já foi mencionado anteriormente. Para efetuar a remoção destes documentos de forma simultânea e atómica da base de dados *Firebase*, foi utilizada a técnica de ***Batched writes*** [3], a qual permite realizar um conjunto de operações de escrita em diferentes documentos e coleções. Desta forma, tal como numa transação, é garantida a atomicidade, não existindo dados instáveis na base de dados, como por exemplo, caso ocorresse um erro na remoção do segundo documento, após o primeiro já ter sido completo/removido com sucesso. Infelizmente, não existe uma forma que permita realizar simultaneamente a remoção de ficheiros armazenados em diferentes serviços, e como tal, para realizar a remoção dos *blobs* referentes à imagem original e à anotada, foi utilizada a mesma técnica, mas desta vez com um *batch* dedicado apenas para a *Cloud Storage*. Após todas as remoções terem sido concluídas é enviado para o cliente uma resposta com a mensagem *ImageResponse* (Figura 3.6), como forma de confirmação que todos os ficheiros foram removidos com sucesso.

```
// Delete a specific file from the Cloud Storage and Firestore
rpc deleteFile(ImageIdentifier) returns (ImageResponse);

message ImageIdentifier {
    string id = 1;
}

message ImageResponse {
    string id = 1;
    string name = 2;
    int32 objectsFound = 3; // Optional field
}
```

Figura 3.6: Definição no contrato da operação de remoção de um ficheiro do sistema

3.3 *Cloud Storage*

A *Cloud Storage* é o serviço onde ficarão armazenadas as imagens, quer as enviadas inicialmente, quer as anotadas com os objetos detetados, escritas através do servidor *gRPC* e da *Detect Objects App*, respetivamente.

O formato utilizado para armazenar e organizar todas as imagens submetidas para a *Cloud Storage* está ilustrado na Figura 3.7, na qual é possível observar a existência de um iden-

tificador único e estável, um UUID, criado pelo servidor aquando da submissão da imagem original, ao qual ficará associado ambas as imagens, a submetida inicialmente e a anotada. Conseguindo ter uma ilusão de divisão por pastas ao colocar no nome dos *blobs* o carácter '/', enriquecendo a interface das aplicações que usam o serviço. Por exemplo, para obter o resultado da Figura 3.7, o servidor criou o *blob* da imagem submetida com o nome "dbf12b99-807e-4487-83f0-fe29496ef5bc/visionimage", e a *Detect Objects App*, após anotar a imagem, cria um *blob* com o nome "dbf12b99-807e-4487-83f0-fe29496ef5bc/visionimage-annotated" (utilizando o mesmo identificador), ficando assim, as imagens associadas ao mesmo pedido organizadas.

Buckets > cn2122tf

Filter by name prefix only							<input type="checkbox"/> Filter objects and folders
	Name	Size	Type	Created	Storage class	Last modified	
<input type="checkbox"/>	dbf12b99-807e-4487-83f0-fe29496ef5bc/	—	Folder	—	—	—	

Buckets > cn2122tf > dbf12b99-807e-4487-83f0-fe29496ef5bc

Filter by name prefix only							<input type="checkbox"/> Filter objects and folders
	Name	Size	Type	Created	Storage class	Last modified	Public access
<input type="checkbox"/>	visionimage	93.8 KB	image/jpg	Jun 11, 20...	Standard	Jun 11, 20...	Not public
<input type="checkbox"/>	visionimage-annotated	51.1 KB	image/jpg	Jun 11, 20...	Standard	Jun 11, 20...	Not public

Figura 3.7: Organização do serviço *Cloud Storage*

Caso um nome de um *bucket* seja passado por argumento ao iniciar o servidor *gRPC*, será este o utilizado nas operações, senão, será utilizado o nome *cn2122tf* pré-definido no servidor. Para ambos os casos, é verificado se o *bucket* está criado no projeto correspondente, caso não esteja, uma vez que o nome de um *bucket* tem de ser único globalmente e não é verificado se os nomes inseridos são realmente únicos, irá ser criado um *bucket* utilizando o nome pré-definido em conjunção com um UUID gerado, de forma a tornar o nome único. Um *bucket* ao ser criado é configurado com localização regional (*EUROPE-WEST1* - situado na Bélgica), com tipo de acesso padrão (*STANDARD*) e com a política da lista de controlo de acessos definida como *fine-grained*, para caso seja necessário alterar alguma permissão, mais tarde, já haja essa possibilidade disponível.

3.4 Cloud Pub/Sub

Este serviço é o que irá permitir a troca desacoplada de mensagens entre componentes do sistema, nomeadamente, entre o servidor *gRPC*, que irá publicar as mensagens, e a *Monitor Function* e *Detect Objects App*, que serão os consumidores destas mensagens a partir de duas subscrições distintas.

O servidor *gRPC*, sempre que acabar de receber uma imagem e caso não ocorram quaisquer tipos de erros, irá publicar uma mensagem no tópico ***detectionworkers*** (caso ainda não exista será criado em *runtime*), o qual, quando receber a mensagem, irá retornar para o produtor um identificador único como confirmação da receção da mesma. Este identificador no servidor é apenas utilizado para questões de *logging*, de forma a que se possa verificar que a publicação foi concluída com sucesso. Na mensagem publicada estão contidos os seguintes dados:

- campo ***data*** que contém o UUID associado ao pedido criado pelo servidor;
- campo ***attributes*** que inclui no seu mapa o nome do *bucket* (***bucket***), o nome do *blob* (***blob***), o nome da imagem (***imageName***) e o tipo da mesma (***imageType***).

O tópico através do padrão *Fan-out* irá entregar a mensagem às duas subscrições disponíveis, uma associada à *Monitor Function*, a qual ao ser *deployed* para a GCP irá criar uma subscrição ***push***, e outra à *Detect Objects App*, que, caso ainda não exista a sua subscrição, irá criá-la com o identificador ***workers***, com o tipo de entrega ***pull***, o *acknowledgement deadline* de 30 segundos e as restantes configurações por *default*. Ambos os componentes utilizam para consumir as mensagens disponíveis nas suas subscrições o padrão *Work-queue*. O esquema deste serviço pode ser observado abaixo na Figura 3.8.

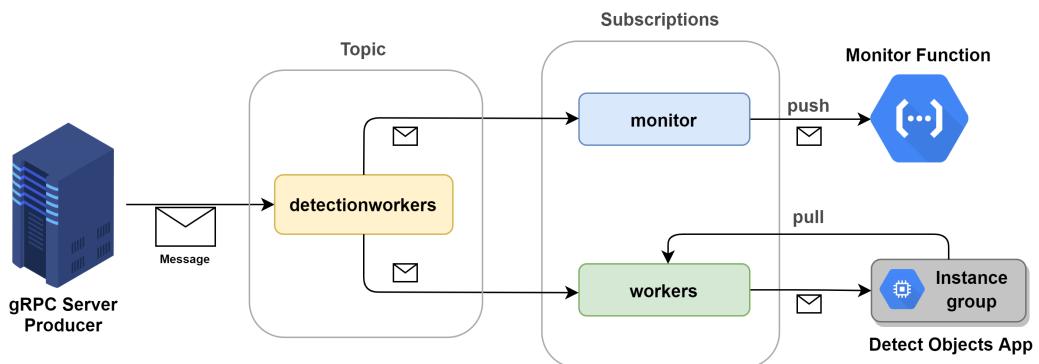


Figura 3.8: Esquema do serviço *Cloud Pub/Sub* no sistema

Na *Detect Objects App*, quando é criado o consumidor para a subscrição *workers*, o *handler* que consome e processa as mensagens está associado a um *ExecutorProvider*, que disponibiliza a **mesma e única thread** para o *handler* realizar o processamento das mensagens consumidas. Contribuindo para que desta forma, ao não existirem duas mensagens a serem processadas ao mesmo tempo, não é possível haverem conflitos de envios de *acknowledges*, só quando uma acabar e tiver sido realizado *acknowledge* positivo ou negativo, é que poderá ser processada outra mensagem ou até a mesma, no caso de ser negativo, uma vez que só existe uma *thread* para as receber. Por exemplo, caso existissem mais *threads* a processar

as mensagens, e supondo que existiam duas mensagens a serem simultaneamente processadas, se a segunda mensagem a chegar emitisse primeiro um *acknowledge* positivo, e depois, a primeira mensagem tivesse, devido a um erro, enviado um *acknowledge* negativo, poderia causar muitos problemas no sistema, uma vez que o *acknowledge* da segunda mensagem seria interpretado como se fosse para a primeira mensagem, que momentos depois obteve um erro, e não conseguiu terminar com sucesso a sua tarefa, e desta forma, como foi recebido um *acknowledge* positivo, a sua mensagem não iria retornar para a fila, ficando o sistema num estado desconhecido e instável.

3.5 Detect Objects App

A aplicação *Detect Objects App* tem como principal função o processamento das imagens submetidas, de forma a anotá-las com as zonas onde foram detetados objetos, tirando partido dos serviços disponibilizados pela *Vision API*. Posteriormente irá armazenar a imagem resultante na *Cloud Storage*, e na *Firebase* irão ficar todas as informações vindas no pedido, isto é, na mensagem que irá ser consumida, e os resultados obtidos do processamento.

Para que esta aplicação seja notificada da existência de uma nova imagem para ser processada, a mesma é consumidora de mensagens publicadas na sua subscrição *workers*, do tópico *detectionworkers*. Desta forma, sempre que chega uma mensagem, a partir do serviço *Cloud Pub/Sub* (descrito na Secção 3.4), são utilizados os campos *bucket* e *blob* para ser fornecida a imagem a processar, armazenada no serviço *Cloud Storage*, à *Vision API*. Após terminado o processamento, irá retornar o nome dos objetos detetados e, associado a cada um, os vértices da zona em que foram detetados e o grau de certeza. Seguidamente, a partir destas informações, será anotada a imagem com o nome dos objetos e as zonas onde foram detetados, e depois armazenada na *Cloud Storage* no formato descrito na Secção 3.3. Por fim, as informações tanto recebidas da mensagem consumida, como as resultantes do processamento da imagem, serão armazenadas na *Firebase*, cujo formato das coleções e documentos é descrita na próxima Subsecção 3.5.1.

Após todos os processos efetuados, será realizado *acknowledge* positivo à mensagem consumida. Caso surja alguma exceção durante o processamento da imagem, será retornado um *acknowledge* negativo, voltando a mensagem para a fila da subscrição. O *deadline* configurado para sinalizar com *acknowledge* positivo é de 30 segundos, sendo muito complicado chegar ao valor ideal para esta configuração.

3.5.1 Armazenamento da informação dos pedidos e resultados na *Firebase*

Pensando nas consultas que são supostas existirem, nomeadamente a obtenção dos objetos detetados e a pesquisa por ficheiros com determinadas propriedades, foram definidas duas coleções para facilitar estas pesquisas. A coleção **Requests** para armazenar a informação re-

ferente ao pedido e a coleção ***DetectedObjects*** para conter em cada documento a informação relacionada com cada objeto encontrado nas imagens processadas.

Para facilitar a obtenção de um determinado documento da coleção *Requests*, cada um destes tem como identificador o UUID criado e retornado pelo servidor ao cliente quando é submetida uma imagem. Contido em cada um, têm a informação referente aos seguintes campos:

- ***id*** - Identificador do pedido, igual ao identificador do documento;
- ***imageName*** - Nome do ficheiro imagem submetido;
- ***imageType*** - Tipo do ficheiro imagem submetido;
- ***bucket*** - Nome do *bucket* do serviço *Cloud Storage* onde estão armazenadas as imagens;
- ***originalBlob*** - Nome do *blob* correspondente à imagem submetida inicialmente para ser processada;
- ***annotatedBlob*** - Nome do *blob* correspondente à imagem anotada com as zonas onde foram detetados objetos;
- ***detectedObjects*** - Número de objetos que foram detetados na imagem original;
- ***creationTimestamp*** - Marco temporal do momento em que foi criado o documento em questão.

Já os documentos referentes à coleção *DetectedObjects*, uma vez que o identificador de cada não é relevante, a sua criação foi deixada ao critério da *Firebase API*. Cada documento contém os seguintes campos:

- ***objectName*** - Nome do objeto detetado na imagem associada ao documento em questão;
- ***score*** - Grau de certeza da *Vision API* quanto ao objeto detetado;
- ***vertices*** - *Array* com os vértices da zona quadrangular onde foi detetado o objeto;
- ***requestId*** - Identificador do pedido, isto é, o identificador do documento da coleção *Requests*, correspondente à imagem na qual foi detetado o objeto;
- ***creationTimestamp*** - Marco temporal do momento em que foi criado o documento em questão.

Para efetuar a escrita **simultânea** e **atómica** dos dois documentos na *Firebase* foi utilizada a técnica de ***Batched writes*** [3], descrita na Subsecção 3.2.6, onde foi utilizada para

realizar um conjunto de remoções, e neste componente foi usada para realizar um conjunto de operações de escrita em diferentes documentos e coleções.

Inicialmente, de forma a que todas as informações referentes aos pedidos e objetos detetados estivessem num único documento e coleção, foi realizada a tentativa de colocar as informações dos objetos encontrados num *array* dentro do documento respetivo, o qual iria conter para cada posição do *array*, um objeto composto pelo nome do objeto detetado, o grau de certeza e um *array* com os vértices da área quadrangular onde o objeto foi detetado, mas uma vez que não foi possível descortinar o mecanismo utilizado pela Google na *Firebase* para realizar pesquisas sobre propriedades de objetos contidos dentro de *arrays*, foi decidido optar pela opção mencionada no início desta subsecção, realizando a divisão em duas coleções.

3.6 *Lookup Function*

A *Lookup Function* corresponde a uma *Cloud Function* cuja responsabilidade é obter os endereços IP das *VMs* do *instance group* onde estão a correr os servidores *gRPC*. A qual retorna para o servidor um objeto num formato JSON específico, ilustrado na Figura 3.9, para no cliente ser mapeado diretamente para um objeto de domínio através da biblioteca *JSON* [4] da Google.

```
{
  "instances": [
    {
      "name": "instance-1",
      "ip": "35.184.76.147"
    },
    {
      "name": "instance-2",
      "ip": "35.184.76.148"
    }
  ]
}
```

Figura 3.9: Exemplo do formato do objeto JSON a enviar para o cliente

Esta função é invocada através de um *trigger* HTTP, e para que saiba em que *instance group* deve filtrar os endereços IP das máquinas virtuais que estão a correr, o cliente precisa de inserir no URL base, criado quando a *Lookup Function* foi *deployed*, na secção da *query string* o nome (com a *key instance-group*) e a zona (com a *key zone*) do *instance group* a que os servidores *gRPC* pertencem. Um exemplo deste URL é o seguinte: "<https://europe-west1-cn2122-t2-g09.cloudfunctions.net/lookup-function?zone=europe-west1-b&instance-group=grpc-server-instance-group>".

Após obtidos os parâmetros mencionados, a *Lookup Function* utiliza a API referente ao serviço *Compute Engine* para obter todas as instâncias disponíveis na zona fornecida, e, consequentemente, filtrar aquelas que pertencem ao *instance group* providenciado e estão no

estado *RUNNING*. Construindo, de seguida, o objeto JSON a ser enviado ao cliente.

Caso surjam erros durante alguma operação e para que seja possível estar a par do desenvolvimento das operações mais relevantes, foi utilizada a classe *Logger* para ser possível observar na GCP as *logs* com as respetivas informações. E, para o cliente também ser informado de algum erro que tenha acontecido, a representação em JSON retornada, contém também um campo *error*, que será preenchido com a mensagem do problema sucedido.

3.7 Monitor Function

A *Monitor Function* representa uma *Cloud Function*, cujo objetivo é contabilizar as mensagens enviadas para o tópico *detectionworkers*, sem afetar o processamento das mesmas pelos *workers*, isto é, as instâncias que executam a *Detect Objects App*, e atuar sobre o número de instâncias de VM pertencentes ao *instance group* correspondente, diminuindo, aumentando ou mantendo este número de acordo com a métrica utilizada.

Para cumprir estes objetivos, a função irá ser invocada através de um *trigger* do tipo *Pub/Sub*, a qual ao ser *deployed* para a GCP, é criada automaticamente uma *push subscription* para o tópico que foi passado como argumento, nomeadamente o tópico *detectionworkers*.

De forma a que a função consiga decidir se deve aumentar, diminuir ou manter o números de *running VMs* do *instance group*, foi utilizada como referência a seguinte equação:

$$ref = \frac{\text{número de mensagens}}{\text{intervalo de monitorização (s)}}$$

Para efeitos de teste, para a referência foram considerados os valores por *default* de **3 mensagens por 60 segundos**, resultando numa **referência de 0,05**. Para o **threshold foi usado o valor 0,02**, significando que sempre que a referência seja inferior ao valor 0,05 (*ref*) - 0,02 (*threshold*) = **0,03 deve ser diminuída** uma instância, caso a referência se encontre superior a $0,05 + 0,02 = 0,07$ **deve ser aumentada** uma instância, caso contrário deverá ser mantido o número de instâncias do *instance group*. Caso o número de VMs a correr se encontre num dos extremos da sua capacidade, quer mínima (1 VM a correr), quer máxima (4 VMs a correr), é salvaguardado que estes valores não diminuem nem aumentem apenas quando a referência indica para uma diminuição ou aumento do número de instâncias, respetivamente, mas somente quando todas as condições se verificam para a respetiva alteração. Quando deve ser realizado o redimensionamento, este só irá ter efeito quando for recebida uma nova mensagem no novo intervalo de monitorização.

Uma vez que as *Cloud Functions* não mantêm estado, os dados relativos às referências, aos cálculos e ao número de mensagens no intervalo de monitorização atual, têm de ser armazenadas na base de dados *Firestore*, para que numa próxima instância da função seja possível utilizar os valores calculados anteriormente. Para tal foi definida a seguinte estrutura de dados para o **único documento** (com o identificador *properties*) da coleção **Monitor**:

- ***refMessages*** - Referência do número de mensagens para um intervalo de monitorização;
- ***refSeconds*** - Referência em segundos do intervalo de monitorização;
- ***threshold*** - Valor a somar ou subtrair à referência para decidir se o número de instâncias deve ser alterado;
- ***firstMessageTimestamp*** - Marco temporal do momento em que chegou a primeira mensagem associado ao intervalo de monitorização atual;
- ***runningInstances*** - Número de instâncias do respetivo *instance group* que se encontram a correr;
- ***currentMessages*** - Número de mensagens recebidas durante o intervalo de monitorização atual.

Caso se queira utilizar outros valores de referência, o mais fácil é alterar os valores dos respetivos campos na *Firestore*, uma vez que sempre que a função é invocada o documento da *Firestore* é mapeado para um objeto de domínio, sendo utilizada as informações mapeadas para fazer novamente todos os cálculos necessários, mas é de notar que, caso a coleção ou o documento seja apagado da base de dados, quando a *Monitor Function* realizar a próxima operação, irá criar a coleção e o documento com os valores por *default* mencionados anteriormente.

Nas ocasiões em que o intervalo de monitorização ainda não terminou, é apenas atualizado no documento o campo *currentMessages*, e quando ocorre o redimensionamento do *instance group* são apenas atualizados os campos *currentMessages*, *runningInstances* e *firstMessageTimestamp*. Para ser realizado o redimensionamento, a *Monitor Function* utiliza a API disponibilizada pelo serviço *Compute Engine*, fazendo uso do método *resizeAsync* da classe *InstanceGroupManagersClient*.

Tal como para a *Lookup Function*, de forma a que seja possível seguir as operações que estão a acontecer na *Monitor Function* através das *logs*, foi utilizada a classe *Logger* para registar as informações relevantes das operações e as causas dos eventuais erros que possam surgir.

Capítulo 4

Pontos de falha e Conclusão

4.1 Pontos de falha

Tendo em conta o tempo e o conhecimento que temos, a implementação do sistema terá, naturalmente, falhas que não conseguimos identificar e outras que nem sequer sabíamos que poderiam existir, mas ainda assim temos consciência de dois possíveis pontos de falha:

- Uma vez que não é possível realizar escritas simultâneas e atómicas na *Firestore* e na *Cloud Storage*, tanto na criação como na remoção, caso uma escrita na *Firestore* tenha sucesso e depois na *Cloud Storage* não tenha sido completada devido a um erro, ou vice-versa, o estado do sistema irá ficar instável, resultando, eventualmente, na existência de erros;
- Caso a rede esteja muito congestionada, quer com muitos pedidos ou outros problemas não associados diretamente ao sistema, poderá acontecer que caso o processamento da deteção de objetos numa imagem submetida esteja a levar muito tempo, leve ao *deadline* do *acknowledge* da mensagem consumida do serviço *Pub/Sub*, uma vez que o processamento da imagem não é cancelado, este irá, eventualmente, chegar ao fim, e como tal, serão realizadas as escritas dos resultados na *Cloud Storage* e *Firestore*. Uma vez que a mensagem não teve *acknowledge* positivo, voltou para a fila de mensagens da subscrição, e por isso, a dada altura, será novamente consumida, resultando na repetição da operação, reescrevendo os possíveis resultados ou gerando exceções inesperadas.

4.2 Testar o sistema

Para testar o sistema, deve ser utilizado o ficheiro README fornecido em conjunto com este relatório, onde estará descrito todas as configurações necessárias para o sistema ser iniciado e utilizado como previsto, como as chaves de serviço necessárias e argumentos que devem ser inseridos ao iniciar os componentes.

4.3 Conclusão

Ao desenvolver este trabalho, foi nos dada a oportunidade de meter em prática todos os conteúdos lecionados, implementando um sistema que fosse capaz de, a partir de uma imagem inicialmente submetida, processá-la e criar uma nova, anotada com as zonas onde foram detetados objetos, para posteriormente ser possível realizar consultas dos objetos que foram detetados. Utilizando, durante o caminho de toda a implementação, os serviços da *Google Cloud Platform* para cimentar todo o conhecimento aprendido. Chegando ao final, e fazendo uma retrospectiva, reparamos que todo o esforço compensou, e que, agora, já temos noções fundamentais para que no futuro consigamos, mais facilmente, entender problemas, resolvê-los e explorar mais sobre o grande mundo da computação nuvem.

Referências

- [1] TCP Packet size. <https://www.baeldung.com/cs/tcp-max-packet-size>. Consultado a 09/06/2022.
- [2] TCP/IP Headers size. <https://www.oreilly.com/library/view/internet-core-protocols/1565925726/ch07s01s05s04.html>. Consultado a 09/06/2022.
- [3] Batched writes. <https://firebase.google.com/docs/firestore/manage-data/transactions#batched-writes>. Consultado a 09/06/2022.
- [4] Google GSON. <https://github.com/google/gson>. Consultado a 09/06/2022.

Apêndice A

Contrato

```
1 syntax = "proto3";
2 import "google/protobuf/timestamp.proto";
3 option java_multiple_files = true;
4 option java_package = "grpcserver";
5
6 package grpcserver;
7
8 // The gRPC server service definition.
9 service Server {
10     // Upload an image
11     rpc uploadImage(stream ImageUploadDownload) returns (ImageResponse);
12     // Get the list of objects found in the desired image
13     rpc getImageDetectedObjects(ImageIdentifier) returns (ImageObjects);
14     // Get the original image annotated with the zones where the objects were found
15     rpc downloadAnnotatedImage(ImageIdentifier) returns (stream ImageUploadDownload);
16     // Get the stored files between two dates, which contain a specific object with a score greater than 't'
17     rpc searchForFiles(SearchProperties) returns (FilesResponse);
18     // Gets all the available documents within a limit
19     rpc getAllFiles(Pagination) returns (FilesResponse);
20     // Delete a specific file from the Cloud Storage and Firestore
21     rpc deleteFile(ImageIdentifier) returns (ImageResponse);
22 }
23
24 message ImageMetadata {
25     string name = 1;
26     string type = 2;
27     int64 size = 3;
28 }
29
30 message ImageUploadDownload {
31     oneof request {
32         ImageMetadata metadata = 1;
33         bytes content = 2;
34     }
35 }
36
37 message ImageResponse {
38     string id = 1;
39     string name = 2;
40     int32 objectsFound = 3; // Optional field
41 }
```

```
42
43 message ImageIdentifier {
44     string id = 1;
45 }
46
47 message ImageObjects {
48     string id = 1;
49     string imageName = 2;
50     // Key = objectName, Value = times that the object was detected
51     map<string, int32> objectsNames = 3;
52 }
53
54 message SearchProperties {
55     google.protobuf.Timestamp initialTimestamp = 1;
56     google.protobuf.Timestamp lastTimestamp = 2;
57     string objectName = 3;
58     double score = 4;
59 }
60
61 message FilesResponse {
62     repeated ImageResponse responses = 1;
63 }
64
65 message Pagination {
66     int32 limit = 1;
67     int32 offset = 2;
68 }
```