UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CÁLCULO DE DESCRIPTORES DE IMÁGENES EN DISPOSITIVOS ANDROID PARA UN SERVICIO DE BÚSQUEDA DE VIDEOS

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

FELIPE ANDRÉS QUINTANILLA MATEFF

PROFESOR GUÍA:
BENJAMÍN BUSTOS CÁRDENAS

MIEMBROS DE LA COMISIÓN:
JAVIER BUSTOS JIMÉNEZ
AIDAN HOGAN

SANTIAGO DE CHILE
2016

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: FELIPE ANDRÉS QUINTANILLA MATEFF
FECHA: 07/03/2016
PROF. GUÍA: SR. BENJAMÍN BUSTOS CÁRDENAS

CÁLCULO DE DESCRIPTORES DE IMÁGENES EN DISPOSITIVOS ANDROID PARA UN SERVICIO DE BÚSQUEDA DE VIDEOS

Desde sus inicios en el cine el material audiovisual sólo se ha masificado y multiplicado. Luego del cine, la aparición de la televisión hace que el consumo este tipo de contenido sea aún más accesible que en épocas anteriores y con las primeras cámaras de video caseras el producir contenido audiovisual se vuelve de a poco más y más común. Gracias a los dispositivos móviles, cámaras de video y cámaras fotográficas cualquier persona puede generar contenido audiovisual. Además a lo anterior se le suman las producciones oficiales de videos musicales, películas, series de televisión, etc. por lo que la cantidad de contenido es enorme y poder identificarlo se vuelve un problema. Hoy en día gran parte de este contenido se encuentra sin etiquetar, es decir no contiene información asociada que lo identifique, haciendo difícil su búsqueda y reconocimiento.

La búsqueda de videos cuando estos no se encuentran etiquetados requiere el uso de técnicas como la *búsqueda por similitud*. Esta búsqueda utiliza descriptores para encontrar videos. Los descriptores son vectores que contienen información del contenido del video, estos describen ciertas características del contenido como puede ser la distribución de colores, intensidad de luz, la ubicación y dirección de los bordes contenidos dentro de una imágen, etc. De esta forma la *búsqueda por similitud* se basa en comparar videos usando descriptores para establecer relaciones de semejanza entre videos. De acuerdo a estas relaciones de semejanza es posible encontrar videos que son similares entre sí.

Actualmente existen soluciones disponibles capaces de realizar la búsqueda por similitud para archivos de video en un mismo computador, sin embargo esta solución no es de fácil acceso para las personas. En esta memoria se propone e implementa un sistema de búsqueda de videos que utiliza dispositivos móviles Android para grabar un video y calcular sus descriptores respectivos, para luego enviar los descriptores a un servidor para realizar una búsqueda por similitud haciendo uso de las herramientas existentes. Se evaluó la eficiencia del sistema propuesto comparando su tiempo de respuesta y uso de la red de datos con una implementación alternativa del sistema, que envía videos completos al servidor para ser procesados. Por otro lado se evaluó la eficacia del sistema implementando tres tipos de descriptores de video distintos, y midiendo el porcentaje de respuestas correctas obtenido al usar cada uno.

Los resultados obtenidos indican que los dispositivos móviles son capaces de realizar la extracción de descriptores eficientemente. El sistema propuesto es más eficiente que la alternativa de enviar videos completos al servidor, con respecto a este sistema alternativo se redujo el tiempo de respuesta del sistema en 30 %, mientras que la cantidad de datos enviada al servidor pasó de más de 6 Megabytes a aproximadamente 20 Kilobytes. Con respecto a la eficacia del sistema, dos de los descriptores implementados mostraron resultados apenas mejores que el azar, mientras que el restante obtuvo una precisión cercana al 50 %.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Motivación

Día a día crece la cantidad de contenido audiovisual disponible gracias en parte a la televisión, a la proliferación de medios de captura de video de bajo costo y a la creciente oferta de servicios de almacenamiento de video en internet como Youtube o Vimeo. Sin embargo, gran parte de este contenido no está etiquetado, es decir, no presenta metadatos que faciliten su rápida identificación.

Para buscar dentro de colecciones de video no etiquetadas comúnmente se usa el enfoque de búsqueda por similitud. Este tipo de búsqueda se basa en comparar (esto es, medir cuanto se parecen) dos videos. Para medir esta similitud se utilizan *descriptores*, que son vectores que representan características del contenido un video y bajo los cuales las comparaciones están definidas mediante funciones de distancia.

La masificación de los *smartphones* o teléfonos inteligentes ha aportado al aumento de la cantidad de videos, y por ende al aumento de la cantidad de contenido no etiquetado, sin embargo por otro lado pueden ofrecernos una solución para su identificación. Existen ejemplos en los que estos dispositivos móviles ayudan a encontrar contenido no etiquetado como es el caso de la aplicación *shazam*[1]. Esta aplicación le permite a sus usuarios identificar contenido de audio que no contiene los datos necesarios para su identificación inmediata, por ejemplo, canciones en una radio. Sin embargo no existen soluciones similares para el caso de contenido audiovisual -como películas o series de televisión- a pesar de que estos dispositivos móviles permiten grabar videos fácilmente, y recientemente han aumentado su poder de cómputo de manera tal que les hace posible realizar tareas previamente relegadas a computadores de escritorio.

La presente memoria se enmarca dentro del siguiente problema: identificar la *fuente* de un segmento de video a partir de su contenido utilizando una búsqueda por similitud. El objetivo que se plantea es que un usuario pueda grabar parte del video no etiquetado usando

[1] `http://www.shazam.com`

su celular y, a través de una búsqueda por similitud, identificar el video. Adicionalmente se desea probar diversas implementaciones del sistema para medir sus diferencias en términos de eficiencia y eficacia, donde eficiencia se refiere al uso de recursos como la red de datos y el tiempo de procesamiento, y eficacia a la habilidad para retornar resultados correctos. Para el correcto desarrollo de este trabajo son necesarios conocimientos de procesamiento de imágenes y búsqueda en contenido multimedia para llevar a cabo búsquedas por similitud del contenido que se está analizando. También es necesario manejar conceptos de programación concurrente de forma de poder implementar los descriptores que, a su vez, permiten llevar a cabo la búsqueda. Finalmente es necesario ser capaz de desarrollar aplicaciones móviles para dispositivos Android.

1.2. Objetivos

El objetivo general de esta memoria es implementar un sistema de búsqueda de videos en donde un usuario graba un video corto con un dispositivo móvil, luego se comunica con un servidor de consultas que identifique la fuente del video y finalmente le comunique sus resultados de vuelta a la aplicación. Se propone implementar el cálculo de descriptores de video para la búsqueda en el dispositivo móvil y comparar su eficiencia y uso de recursos con la alternativa de enviar el video completo para ser procesado por el servidor. Además se desea implementar distintos tipos de descriptores y evaluar su eficacia para reconocer el video de búsqueda.

Para conseguir el objetivo principal de la memoria se pueden detallar varios objetivos específicos los cuales son expuestos a continuación y agrupados en tres partes:

Aplicación Android

- Implementar una aplicación Android capaz de capturar frames usando la cámara del dispositivo.
- Implementar un módulo que reciba frames de la cámara y calcule descriptores de imágenes, el módulo debe ser capaz de calcular descriptores de Histograma de grises, Histograma de bordes y Distribución de colores.
- Obtener resultados de la búsqueda ejecutada por el servidor y mostrárselos al usuario.

Servidor

- Descargar películas de dominio público e indexarlas para crear una base de datos de videos sobre la cual se realizarán consultas.
- Implementar un servidor que reciba descriptores de la aplicación móvil y ejecute una búsqueda de copias en base a ellos.
- Comunicarle los resultados de la búsqueda de vuelta a la aplicación.

Experimentos

- Comparar los descriptores implementados en términos de su eficacia para la búsqueda.
- Implementar una versión alternativa del sistema que reciba videos completos de la aplicación móvil y realice tanto el cálculo de descriptores como la búsqueda en el servidor.
- Comparar las dos versiones del sistema en términos de sus eficiencia para cuantificar las ganancias comparativas logradas al realizar el cálculo de descriptores en el dispositivo móvil.

1.3. Estructura de la memoria

El capítulo 1 contiene la seccion 1.1 que presenta una breve introducción y motivación de los problemas relevantes para la memoria y la sección 1.2 donde se detalla el objetivo principal y una lista de objetivos específicos de esta memoria. En el capítulo 2 define conceptos útiles en 2.1, para luego explicar el concepto de descriptores de video en 2.2 y detallar el funcionamiento de la detección de copias en 2.3. En las siguientes secciones del capítulo se describen los programas y herramientas usadas, en 2.4 se describe el sistema Android, mientras que en 2.5 se discuten herramientas disponibles para procesamiento de imágenes en Android, en 2.6 el framework RenderScript y en 2.7 el programa P-VCD. El capítulo 3 expone los descriptores escogidos para implementar en esta memoria, "Histograma de grises por zona" en 3.1, "Distribución de colores" en 3.2 e "Histograma de bordes" en 3.3. En el capítulo 4 se detalla la implementación del sistema, se divide dos secciones para describir las alternativas de arquitectura del sistema, una centralizada 4.1 y otra distribuida 4.2. El capítulo 5 describe los experimentos realizados para evaluar el sistema en 5.1, para luego mostrar y discutir sus resultados en 5.2. Finalmente el capítulo 6 presenta conclusiones finales de la memoria en 6.1, y lista posible trabajo futuro en 6.2.

Capítulo 2

Conceptos

En este capítulo se presentan conceptos teóricos necesarios para la comprensión del trabajo realizado en la memoria, así como la descripción de programas y frameworks utilizados cuyo funcionamiento y restricciones influyeron en decisiones de diseño tomadas a lo largo del desarrollo de la memoria.

2.1. Definiciones

1. Imagen: Una imagen se puede considerar una matriz de pixeles. El contenido de cada pixel depende del tipo de imagen, en el caso de imágenes en tonos de grises cada pixel corresponde a un valor entero entre 0 y 255, en el caso de imágenes a color cada pixel corresponde a tres números enteros cuyo significado depende del espacio de color que se utilice. En esta memoria las imágenes a color siempre corresponderán al espacio RGB, donde los tres valores del pixel corresponden a su intensidad de rojo, verde y azul respectivamente.
2. Video: Corresponde a una secuencia de imágenes (de aquí en adelante *frames*) presentadas cada cierto intervalo de tiempo regular (por ejemplo 24 *frames* cada segundo).

2.2. Descriptores

Un *descriptor* [3] de imagen es una forma representar dicha imagen por sus características (como su distribución de colores, orientación de bordes, textura, etc). Los descriptores son usualmente representados como vectores multidimensionales. Para compararlos es común usar una función de distancia, como por ejemplo la distancia euclideana entre vectores.

El descriptor de una imagen puede ser global o local dependiendo de si describe la imagen completa o solo sectores de ella.
Dado que un video es una secuencia de imágenes se puede describir usando una secuencia de

descriptores de imágenes, para hacer esto típicamente se submuestrean los frames del video, es decir, aunque el video muestre 24 frames cada segundo, se pueden considerar solo algunos de ellos a intervalos regulares para describir el video. Esto funciona debido que en la mayoría de los casos dos frames consecutivos son muy similares entre sí, por lo que tomar todos los frames del video para describirlo aporta información redundante.
Es necesario comparar videos o imágenes utilizando descriptores en vez de comparar sus archivos bit a bit debido a que se pueden realizar transformaciones sobre una imagen que cambien totalmente los bits de su archivo sin modificar mucho el contenido de esta, como por ejemplo iluminar la imagen sumándole un valor fijo al valor de todos sus pixeles.

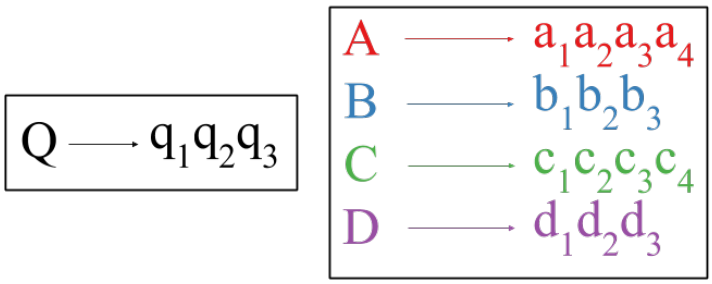
2.3. Detección de copia basada en contenido

La detección de copia basada en contenido o *Content Based Copy Detection* (CBCD) se define como [4]:
Sea V una colección de documentos multimedia (imágenes, audio, video) y sea q un documento de búsqueda. Una *detección de copia basada en contenido* consiste en encontrar y recuperar todos los documentos $v \in V$ de la colección tales que q "es una copia" de v.

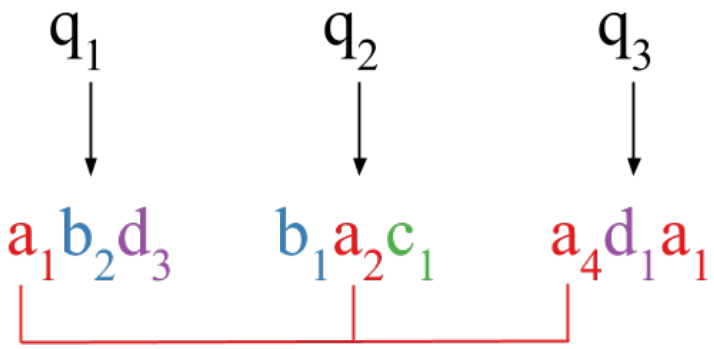
En el contexto de esta memoria los documentos corresponden a videos, y la relación "es una copia" se refiere a que el video de consulta corresponde a una grabación del video original.

Para establecer si dos videos cumplen la relación de copia es necesario tener una manera de comparar su contenido. Es muy probable que la grabación no sea idéntica al video original debido a transformaciones de color y perspectiva inherentes a la grabación, sin embargo es posible comparar los videos por las *características* presentes en ellos, como sus colores, formas, etc. Esta comparación de características se realiza por medio de *descriptores*.

La Figura 2.1 ilustra las etapas del proceso de detección de copia. El primer paso, correspondiente a la Figura 2.1a, es extraer descriptores del video de consulta y de cada uno de los videos de la colección de referencia (estos últimos pueden ser precalculados y utilizados en múltiples consultas). El siguiente paso, ilustrado en la figura 2.1b, es realizar una *búsqueda por similitud*, que corresponde a identificar por cada descriptor del video de consulta, los descriptores más *similares* entre los de la colección de referencia. En este contexto los descriptores *más similares* corresponden a los que presenten menor distancia en el espacio vectorial. Finalmente se toma la información de descriptores más similares y se buscan secuencias crecientes de descriptores que correspondan al mismo video. Se razona que si para cada descriptor del video de consulta, se encuentra entre sus más cercanos a descriptores de un mismo video, entonces el video de consulta representa una copia de un segmento del video de la colección de referencia.



(a) Extracción de descriptores. Con mayúscula se representan los videos y con mínuscula y subíndice sus descriptores



(b) Búsqueda por similitud y detección

Figura 2.1: Etapas del proceso de detección de copias en video.

2.4. Android

Android es un sistema operativo desarrollado por Google [1], es principalmente usado en dispositivos móviles como smartphones y tablets. Las aplicaciones para Android son programas escritos en el lenguaje de programación Java que hacen uso de las bibliotecas del *Software Development Kit* (SDK) de Android para realizar llamadas al sistema operativo. Esta sección presenta brevemente conceptos del sistema Android necesarios para entender la implementación del trabajo realizado.

Actividades y Fragmentos

Una actividad corresponde a cualquier clase que extienda de la clase Activity[1] o alguno de sus descendientes, asimismo un fragmento corresponde a clases que extiendan de Fragment[2]. Las actividades y fragmentos controlan elementos de la interfaz gráfica de la aplicación. La vista de la interfaz gráfica es definida usando XML para especificar los elementos de la interfaz y sus posiciones. El comportamiento en cambio se controla desde actividades y fragmentos, que asignan métodos *listener* a los distintos elementos de la interfaz. En general se usa una actividad para manejar un proceso dentro de la aplicación y varios fragmentos se encargan cada uno de una etapa del proceso.

Comunicación entre aplicaciones

Existen ciertos procesos que pueden ser necesarios para diversos tipos de aplicaciones, como por ejemplo mostrar los archivos de una carpeta para que el usuario elija uno, o reproducir un archivo de video. Si bien es posible que cada aplicación implemente su propia solución a estos problemas un dispositivo Android usualmente cuenta con aplicaciones predefinidas que realizan estos procesos. Cuando una aplicación requiere tales procesos puede realizar una llamada al sistema que invoque una aplicación predefinida y, una vez se obtenga un resultado (se seleccionó un archivo, se reprodujo el video, etc) se le entregan los resultados de vuelta a la aplicación original. Esta llamada se realiza a través de un objeto *Intent*[3], que representa una descripción abstracta del trabajo que se quiere realizar. Android provee una lista de intents estándar para acciones comunes. El siguiente código muestra como se puede invocar la aplicación por defecto de la cámara para grabar un video y guardarlo a archivo:

```
1     Intent intent = new
         Intent(MediaStore.ACTION_VIDEO_CAPTURE);
2     videoFile = getOutputFile();
3     intent.putExtra(MediaStore.EXTRA_OUTPUT, videoFile);
4     intent.putExtra(MediaStore.EXTRA_DURATION_LIMIT, 5);
5
```

[1] http://developer.android.com/guide/components/activities.html
[2] http://developer.android.com/guide/components/fragments.html
[3] http://developer.android.com/reference/android/content/Intent.html

```
6     startActivityForResult(intent,
          CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE);
```

En la primera línea se selecciona el intent apropiado. Es posible agregarle opciones dependiendo de la acción requerida, en el ejemplo se especifica el archivo de salida y la duración máxima del video. Finalmente se invoca la aplicación usando el método `startActivityForResult` que recibe al intent y un código que identifica a la aplicación original. Para obtener el resultado de la acción basta implementar el método `onActivityResult` que será llamado automáticamente por el sistema.

2.5. Procesamiento de imágenes en Android

Los dispositivos móviles disponen de limitada memoria y capacidad de computo en comparación con un computador de escritorio. Es por esto que para implementar la extracción de descriptores en Android fue necesario buscar alternativas que permitieran hacer uso eficiente de los recursos disponibles, después de investigar se encontraron dos opciones prometedoras. Por un lado está la librería OpenCV[4] de procesamiento de imágenes y visión por computador. Esta librería es ampliamenta usada en la industria y se encuentra disponible para multiples plataformas, incluido el sistema Android. Está implementada en C/C++ y cuenta con interfaces para Java y Python. Se utilizó un libro escrito por los autores de la librería como apoyo para la implementación [7]. Por otro lado Android cuenta con un framework propio para aplicaciones que requieran alta capacidad de computo llamado RenderScript[5].

Antes de comenzar la implementación de la memoria se pusieron a prueba ambas alternativas. Se implementaron dos operaciones típicas de procesamiento de imágenes, convertir una imágen de color a gris y realizar una convolución, además se implementó unos de los descriptores usados en la memoria. Se implementaron de manera que fueran capaces de recibir *frames* de la cámara del dispositivo mientras esta realiza una grabación. Se midió la cantidad de tiempo requerido por cada implementación para realizar cada operación, además se usaron dos calidades de video distintas para medir el efecto del aumento en el tamaño de la imagen. Los resultados de estas pruebas se resumen en la Figura 2.2, podemos ver que en todos los casos la implementación en Renderscript resultó ser más rápida que la de OpenCV. Dados estos resultados se decidió utilizar el framework RenderScript para implementar las tareas de procesamiento de imágenes en el dispositivo móvil.

En la siguiente sección se detalla el funcionamiento del framework RenderScript.

[4] `http://opencv.org/`
[5] `http://developer.android.com/guide/topics/renderscript/compute.html`

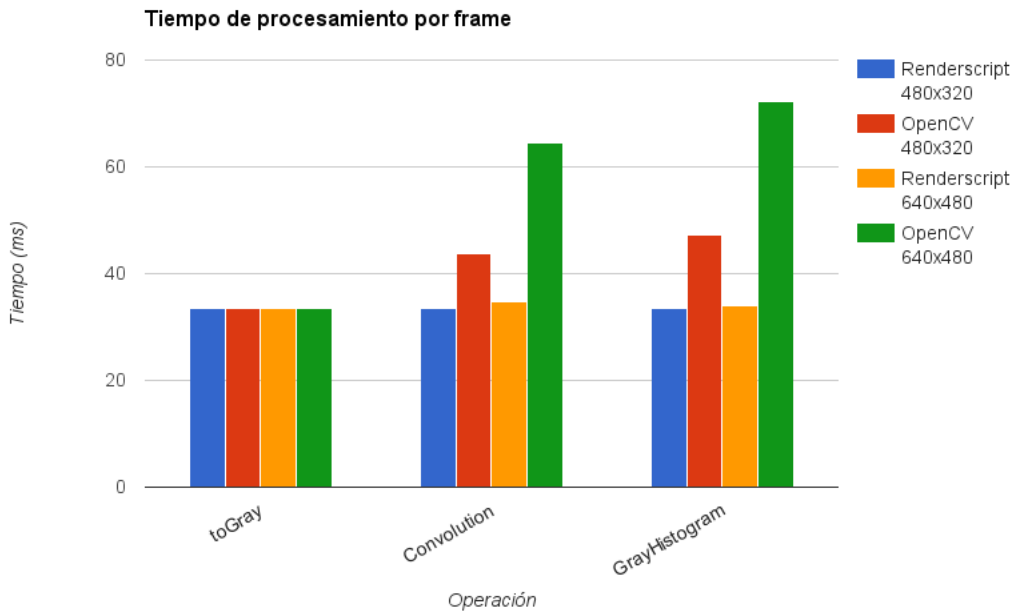


Figura 2.2: Resultados de la comparación entre RenderScript y OpenCV.

2.6. RenderScript

RenderScript es un framework de Android diseñado para lograr alto rendimiento al correr código en paralelo, aunque algoritmos secuenciales también pueden verse beneficiados por él. El funcionamiento de RenderScript se basa en escribir un *kernel* de procesamiento que el sistema puede distribuir a los procesadores disponibles en el dispositivo, como los múltiples núcleos de la CPU, o la GPU. Los kernel se escriben en un lenguaje derivado de C99, a continuación se presenta un ejemplo de un kernel que convierte una imagen en RGBA a escala de grises.

```
1  #pragma version(1)
2  #pragma rs java_package_name(fquintan.renderscripttest)
3  #pragma rs_fp_relaxed
4
5  void rgba_to_grayscale(uchar4 *v_in, uchar *v_out,
6                         uint32_t x, uint32_t y) {
7      uchar red = (*v_in).r;
8      uchar green = (*v_in).g;
9      uchar blue = (*v_in).b;
10     uchar intensity = (uchar) 0.2989*red + 0.5870*green + 0.1140*blue;
11     *v_out = intensity;
12 }
```

Las primeras tres líneas corresponden a configuración de Renderscript. La función `rgb_to_grayscale` recibe 4 parámetros. El primero es un puntero a un `uchar4`, un tipo nativo de RenderScript que representa un pixel con cuatro elementos, este corresponde a un pixel de

la imagen de entrada. El segundo es un puntero a un `uchar` que corresponde a un pixel de la imágen de salida. El tercer y cuarto parámetro corresponden a las coordenadas que ocupan los pixeles de entrada y salida en sus respectivas imágenes. El cuerpo de la función extrae los valores de rojo, verde y azul del pixel de entrada, calcula el valor de gris correspondiente y lo asigna al pixel de salida.

El código anterior se ejecutará una vez para cada pixel de la imagen de entrada, creando así la imagen de salida. El sistema operativo distribuirá esta ejecución de la mejor manera posible haciendo uso de los múltiples núcleos de la CPU o GPU para poder realizar el cálculo de varios pixeles en paralelo.

El caso anterior es simple dado que todos los pixeles son independientes entre sí, por lo que no requiere esfuerzo del programador sincronizar su ejecución. Para casos más complejos RenderScript cuenta con primitivas de sincronización que permiten tener acceso exclusivo a una variable compartida para modificar su valor.

Suponiendo que el código anterior se guarda en el archivo `RGBA2Gray.rs` al compilar se generará la clase Java `ScriptC_RGBA2Gray` con código que permite ejecutar el script. Para ejecutar primero debemos pasar el input a un formato que RenderScript pueda recibir, para esto el framework provee la clase `Allocation`[6]. Una Allocation es un contenedor para los elementos de entrada y salida de un script. Para el ejemplo anterior debemos crear dos Allocations como sigue:

```
1  Allocation inAllocation;
2  Allocation outAllocation;
3
4  rs = RenderScript.create(context);
5
6  Type.Builder inType = new Type.Builder(rs, Element.U8_4(rs));
7  inType.setX(imageWidth);
8  inType.setY(imageHeight);
9
10 inAllocation = Allocation.createTyped(rs, inType.create(),
11             Allocation.USAGE_SCRIPT);
12
13 Type.Builder outType = new Type.Builder(rs, Element.U8(rs));
14 outType.setX(imageWidth);
15 outType.setY(imageHeight);
16
17 outAllocation = Allocation.createTyped(rs, outType.create(),
18             Allocation.USAGE_SCRIPT);
```

Notemos que en las líneas 8 y 13 se define el tipo de elemento de cada Allocation, en este caso, la entrada corresponde a pixeles de tipo `uchar_4`, que se traducen en Java al tipo `Element.U8_4`, mientras que la salida contenía pixeles de tipo `uchar`, que se traducen al tipo `Element.U8`. El tamaño de las Allocation debe ser definido al momento de crearlas,

[6] `http://developer.android.com/reference/android/renderscript/Allocation.html`

en este caso ambas Allocations tienen el mismo tamaño, pero RenderScript permite definir Allocations de entrada y salida de distintos tamaños.

Finalmente hay que poblar de datos la Allocation de entrada, correr el script y extraer los datos de la Allocation de salida como muestra el siguiente código:

```
1 Bitmap imageRGBA;
2 Bitmap imageGray;
3 // ... inicializar los bitmaps
4 inAllocation.copyFrom(imageRGBA);
5 ScriptC_RGBA2Gray script = new ScriptC_RGBA2Gray(rs);
6 script.forEach(inAllocation, outAllocation);
7 outAllocation.copyTo(imageGray);
```

La clase Allocation cuenta con métodos `copyFrom` para poblar Allocations a partir de múltiples estructuras de datos como Bitmaps o arreglos numéricos, por otro lado se cuenta con métodos `copyTo` para extraer los resultados de la Allocation de salida. El método `forEach` de la clase `ScriptC_RGBA2Gray` ejecuta el script sobre las Allocations recibidas.

Se utilizó el framework RenderScript para implementar el cálculo de descriptores en el dispositivo móvil, los detalles de esta implementación se discuten en el capítulo 4 de este informe.

2.7. P-VCD

P-VCD es un software de detección de copias en video [5], es decir, detecta segmentos de video similares dentro de una colección. Fue implementado como parte de una tesis de doctorado y fue probado participando en la competencia de Content-based copy detection de TRECVID en 2010 y 2011 [6].

P-VCD logra detectar videos similares siguiendo el siguiente algoritmo:

1. Calcular descriptores para una base de datos de videos. El programa cuenta con múltiples tipos de descriptores tanto locales como globales que pueden ser utilizados. Se guardan en distintos archivos tanto los descriptores como la información de que frame representa cada descriptor.
2. Dado un video de consulta se calculan y guardan sus descriptores.
3. Para cada descriptor del video de consulta se encuentran en la base de datos sus vecinos más cercanos, esto es, los descriptores que tengan menor distancia al descriptor de consulta según una función de distancia apropiada, como lo es por ejemplo la distancia euclideana entre vectores.
4. Usando los vecinos más cercanos encontrados, se usa la información de a que frame y a que video corresponden para identificar secuencias donde los vecinos más cercanos correspondan a una secuencia creciente de frames del mismo video. Si se encuentra tal secuencia se guarda información del video original y de la ubicación de la secuencia de

frames. Pueden encontrarse múltiples posibles copias para una mismo video de consulta, así como se puede no encontrar ningún candidato a copia.

El programa está construido de manera modular de forma que cada etapa del proceso de búsqueda se puede realizar independiente de las otras, esto hace que P-VCD sea útil para el desarrollo de esta memoria, pues se puede crear e indexar una base de datos de videos de forma offline, para luego saltarse la etapa de extracción de descriptores del objeto de búsqueda, entregándole al sistema los descriptores calculados por otro programa y formateados de la misma manera y prosiguiendo a la etapa de detección de copias.

P-VCD se invoca usando comandos de consola para cada etapa del proceso de búsqueda. A continuación se detallan los comandos necesarios para realizar la extracción de descriptores y la búsqueda de videos similares.

Suponiendo que todos los videos que se utilizarán como referencia están ubicados en el directorio `videos_dir`, se procede a inicializar una base de datos con el comando:

```
> pvcd_db -new -db database videos_dir
```

Este comando crea el directorio `database` que contiene el archivo `files.txt`, este archivo contiene información de todos los videos encontrados en `videos_dir`, como sus dimensiones, tipo de archivo, cantidad de frames, etc. De ahora en adelante para trabajar con los videos basta referenciar la base de datos `database`. El paso siguiente es calcular una segmentación con el comando:

```
> pvcd_db -segment -db database -seg SEGCTE_0.25
```

La segmentación define el conjunto de frames que se usará para describir cada video, en el ejemplo anterior se extraen frames a intervalos regulares de 0.25 segundos. Como resultado de este comando se crea el directorio `segmentations` dentro de `database`, que contiene un archivo de texto con extensión .seg por cada video de la base de datos. Cada línea del archivo .seg contiene el momento exacto al cual corresponde el frame extraído dentro del video original. El siguiente paso es calcular los descriptores, para esto se usa el comando:

```
> pvcd_db -extract -db database -seg SEGCTE_0.25 -desc [DESCRIPTOR] -alias [ALIAS]
```

Este comando utiliza la segmentación creada anteriormente para calcular descriptores a cada frame seleccionado, el parámetro `[DESCRIPTOR]` corresponde a la definición de alguno de los descriptores disponibles en P-VCD, mientras que `[ALIAS]` corresponde a un parámetro opcional que puede usarse para renombrar un descriptor, de ahora en adelante se puede referir al descriptor usando su alias. Como resultado del comando anterior se crea un directorio `descriptors` dentro de `database`, que contiene un archivo binario de extensión .bin para cada video de la base de datos. Cada archivo .bin contiene todos los descriptores calculados para el video correspondiente. Con esto ya tenemos calculados los descriptores de la base de datos.

Para iniciar una búsqueda el primer paso es repetir los pasos anteriores para el video de consulta, con los comandos:

```
> pvcd_db -new -db database video_de_consulta.mp4
> pvcd_db -segment -db consulta -seg SEGCTE_0.25
> pvcd_db -extract -db consulta -seg SEGCTE_0.25 -desc [DESCRIPTOR] -alias [ALIAS]
```

Cabe destacar que el tipo de descriptor y el alias deben ser iguales a los usados anteriormente para poder realizar la búsqueda. El siguiente paso es crear un perfil del búsqueda

```
> pvcd_search -new -profile perfil -query consulta
    -reference database -desc [ALIAS] -distance L2
```

Este ejemplo crea un nuevo perfil de consulta, que usará la base de datos ubicada en `database` como referencia y la ubicada en `consulta` como objeto de consulta, se usará el descriptor renombrado como `[ALIAS]` para la búsqueda y la distancia `L2` (distancia euclideana) como función de distancia. Seguido esto podemos ejecutar la búsqueda de vecinos más cercanos para cada descriptor:

```
> pvcd_search -ss -profile perfil -knn 3
```

Este comando ejecuta la búsqueda, encontrando para cada descriptor de la base de datos de consulta, sus tres vecinos más cercanos. Los resultados se guardan por defecto en el archivo de texto `ss,segments-knn_3.txt` del directorio `perfil`. El paso final es encontrar secuencias entre los vecinos más cercanos usando el comando:

```
> pvcd_detect -detect -ss "perfil\ss,segments-knn\_3.txt"
    -minLength 1s -out "perfil\detections.txt"
```

Con este comando se puede especificar el archivo que contiene los vecinos más cercanos, así como un archivo de salida donde se escribirá la información de los candidatos a copia detectados. También se puede especificar con el parámetro `minLength` mínimo largo que debe tener una secuencia de frames del mismo video para ser considerado un candidato a copia. Después de ejecutar esta serie de comandos el archivo `detections.txt` contendrá los candidatos a copia.

De la descripción anterior podemos ver que el proceso de búsqueda de P-VCD está separado en etapas autocontenidas, cualquiera de las cuales puede ser fácilmente reemplazada por otro programa que obtenga los mismos resultados, siempre y cuando se respete el formato de los archivos de entrada y salida de cada etapa. Esto fue especialmente útil para el desarrollo de la memoria, pues permitió realizar el cálculo de descriptores del video de consulta de forma independiente, integrando los resultados de este cálculo con las siguientes etapas de la búsqueda sin mayores problemas.

Capítulo 3

Descriptores

En este capítulo se detalla el funcionamiento de los descriptores que fueron implementados y probados en la memoria.

3.1. Descriptor de Histograma de grises por zona (GHD)

El descriptor de histograma de grises o *gray histogram descriptor* busca capturar la distribución espacial de la luz en la imagen usando histogramas de grises [4]. Un histograma de grises, o histograma de intensidad, representa la cantidad de pixeles con cierto rango de intensidad, suponiendo que un pixel tiene intensidad entre 0 y 255 y que el histograma tiene m bins entonces el valor del bin i del histograma representa la cantidad de pixeles en la imagen con intensidad en el intervalo $[\frac{255*\mathrm{i}}{m}, \frac{255*(\mathrm{i}+1)}{m})$.

Para calcular el GHD de una imagen primero se pasa la imagen a escala de grises, luego se divide en $W \times H$ zonas, para cada zona se calcula un histograma de m bins con la intensidad de sus pixeles, para formar el descriptor global de la imagen se concatenan los histogramas de cada zona. La figura 3.1 ilustra el proceso del cálculo del descriptor. El descriptor final es un vector de $W \times H \times m$ dimensiones.

3.2. Descriptor de distribución de colores (CLD)

El descriptor de distribución de colores o *color layout descriptor* busca representar la distribución de color en la imagen [8, 9]. Para lograr esto se divide en $W \times H$ zonas, para cada zona se calcula el color promedio de sus pixeles, calculando por separado el promedio en cada canal de la imagen RGB, así para cada zona se obtiene tres valores correspondientes a los valores promedio de rojo, verde y azul en la zona. La figura 3.2 ilustra el resultado de el proceso anterior. Se concatenan los resultados de cada zona para formar el descriptor global. El descriptor es un vector de $W \times H \times 3$ dimensiones.

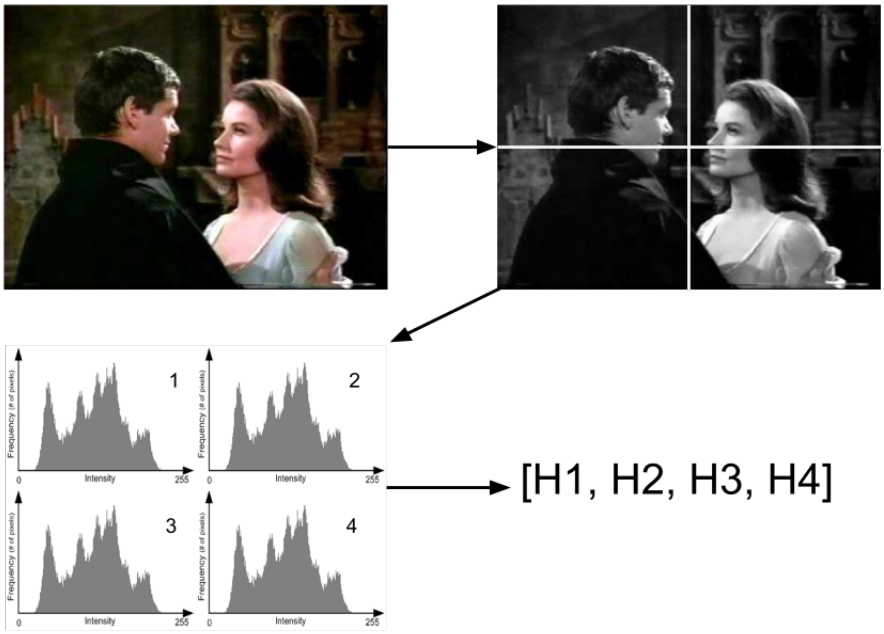


Figura 3.1: Ilustración del algoritmo para calcular el descriptor GHD.

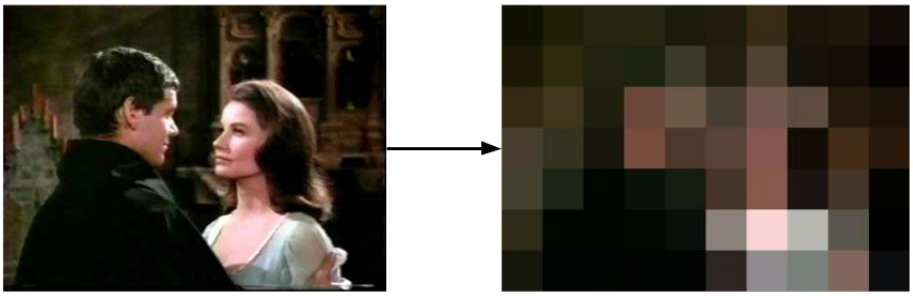


Figura 3.2: Ilustración del algoritmo para calcular el descriptor CLD.

3.3. Descriptor de histograma de bordes por zona (EHD)

El descriptor de histograma de bordes o *edge histogram descriptor* busca representar la orientación de los bordes presentes en la imagen [10, 9]. Se construye pasando la imagen a escala de grises y particionándola en $W \times H$ zonas, cada zona se subdivide en $M_w \times M_h$ bloques y cada bloque se subdivide en cuatro sub-bloques como muestra la Figura 3.3. De aquí en adelante se trata al sub-bloque como la unidad mínima de la imagen, y su valor corresponde al promedio de intensidad de gris de los pixeles contenidos en el sub-bloque. Cada bloque se clasifica según la orientación de los bordes presentes en él. Se definen cinco tipos de borde ilustrados en la figura 3.3, cada tipo de borde tiene asociado un filtro. Para identificar el tipo de borde presente en un bloque calculamos la *energía* de cada filtro como:

$$E_{\mathrm{i}} = \sum_{\mathrm{i}=0}^{1} \left(\sum_{j=0}^{1} \mathrm{filtro}_{\mathrm{i}}(\mathrm{i}, j) * \mathrm{subbloque}(\mathrm{i}, j) \right)$$

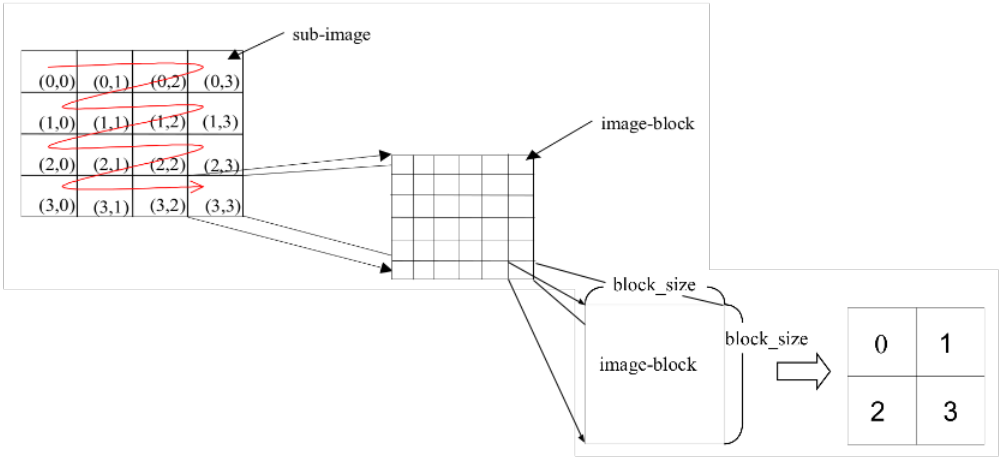


Figura 3.3: Subdivision de la imagen usada por el algoritmo del descriptor EHD.

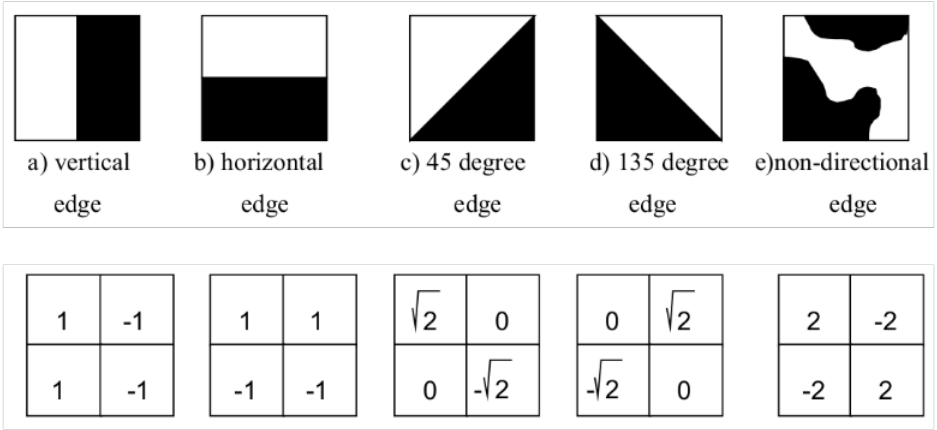


Figura 3.4: Tipos de borde y sus respectivos filtros para el descriptor EHD.

Para cada bloque calculamos la energía de cada filtro, si el valor de la mayor energía supera un umbral t entonces el bloque se clasifica con el tipo de borde correspondiente, si el mayor no alcanza el valor umbral el bloque se clasifica como sin bordes.

Luego para cada zona se calcula un histograma de cinco bins donde cada bin corresponde a un tipo de borde, y el valor del bin corresponde a la cantidad de bloques dentro de la zona que se clasificaron con ese borde, los bloques clasificados como sin borde no cuentan en el histograma. Finalmente el descriptor global está dado por la concatenación de los histogramas de cada zona de la imagen. El descriptor final tiene dimensión $W \times H \times 5$.

Estos tres tipos de descriptores fueron implementados y comparados en términos de eficacia, es decir, la cantidad de resultados correctos que produjo cada uno al usarlos en la búsqueda.

Capítulo 4

Implementación

En este capítulo se describe la implementación del sistema desarrollado durante la memoria. Se implementaron dos posibles arquitecturas para un sistema de reconocimiento de videos. La primera alternativa graba videos con el dispositivo móvil y los envía completos al servidor, para que este calcule sus descriptores y realice la búsqueda, llamamos a esta la alternativa *centralizada* pues todos los cálculos son realizados en el servidor. La segunda alternativa del sistema realiza los cálculos de descriptores en el mismo dispositivo móvil que graba el video, luego solo es necesario enviar los descriptores calculados al servidor que ejecuta la búsqueda, llamamos a esta alternativa *distribuida* pues los cálculos están repartidos entre el cliente y el servidor.

En ambas versiones del sistema el cliente corresponde a una aplicación Android implementada en el lenguaje de programación Java, mientras que el servidor fue implementado en Python usando Flask[1], un framework minimal para programación de servicios web. Se decidió este ambiente de programación para el servidor dada la familiaridad del alumno memorista con el framework y porque la implementación del servidor no requiere funcionalidad compleja que justifique usar herramientas más completas. Todo el código descrito en este capítulo está disponible en Github en un repositorio para el cliente[2] y otro para el servidor[3].

El capítulo se divide en una sección para la alternativa centralizada y otra para la distribuida. A su vez cada sección se divide en la implementación del cliente Android y el servidor.

4.1. Sistema Centralizado

La figura 4.1 muestra la arquitectura de la versión centralizada del sistema. En esta versión el dispositivo móvil es usado solo para grabar un video corto. El archivo de video resultante es enviado al servidor (1), el cual calcula sus descriptores (2) y realiza la búsqueda en la base de datos (3), devolviendo los resultados al cliente (4). A continuación se detalla la

[1] `http://flask.pocoo.org/`

[2] `https://github.com/fquintan/MovieDetector`

[3] `https://github.com/fquintan/MovieDetectorServidor`

implementación de esta versión del sistema, separando el cliente Android del servidor.

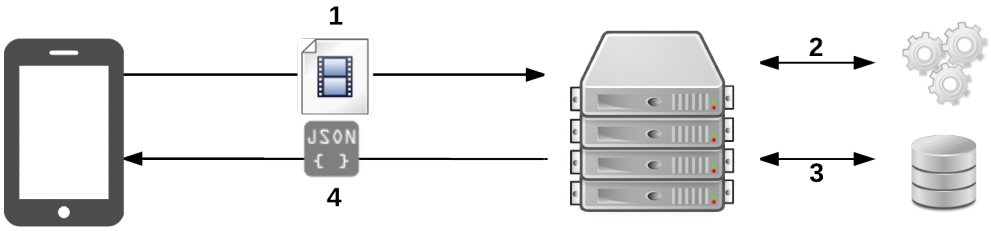


Figura 4.1: Diagrama de la arquitectura del sistema centralizado.

4.1.1. Cliente Android

En esta versión el cliente Android es responsable de grabar un video usando la cámara del dispositivo, enviarlo al servidor y de mostrarle al usuario los resultados de la búsqueda realizada por el servidor. Para cada una de estas responsabilidades se implementó un módulo.

Grabación

El módulo de grabación es responsable de grabar un video usando la cámara del dispositivo y guardar el archivo resultante. Para lograr esto se creó el fragmento `VideoRecordFragment`, esta clase presenta al usuario un breve mensaje de instrucciones y un botón para comenzar la grabación. La figura 4.2 corresponde a la pantalla que ve el usuario al iniciar la aplicación. Al hacer click en el botón *grabar* el sistema invoca a la aplicación por defecto de grabación de videos de Android por medio de un Intent. La aplicación de grabación presenta un preview de la cámara, el usuario inicia la grabación presionando un botón, la aplicación graba por un máximo de 5 segundos y guarda en un archivo el video resultante. Después de terminar la grabación el sistema le cede de vuelta el control a la aplicación llamando al método `onActivityResult`, con esto se procede a enviar el video resultante al servidor.

Conexión con el servidor

El módulo de conexión con el servidor es responsable de enviar al servidor el video capturado por el módulo anterior. Se implementó la clase `FromVideoSearchRequest` que recibe un archivo y ejecuta un petición HTTP Post al servidor usando la clase `HttpUrlConnection` de Java. Con la petición Post se envía el archivo grabado y un parámetro con el tipo de descriptor para usar en la búsqueda. Al crear un objeto de la clase `FromVideoSearchRequest` se le debe entregar un `ResponseHandler`, que corresponde a una interfaz con los métodos `onSuccessResponse` y `onFailure`, alguno de estos métodos se llamará dependiendo de la

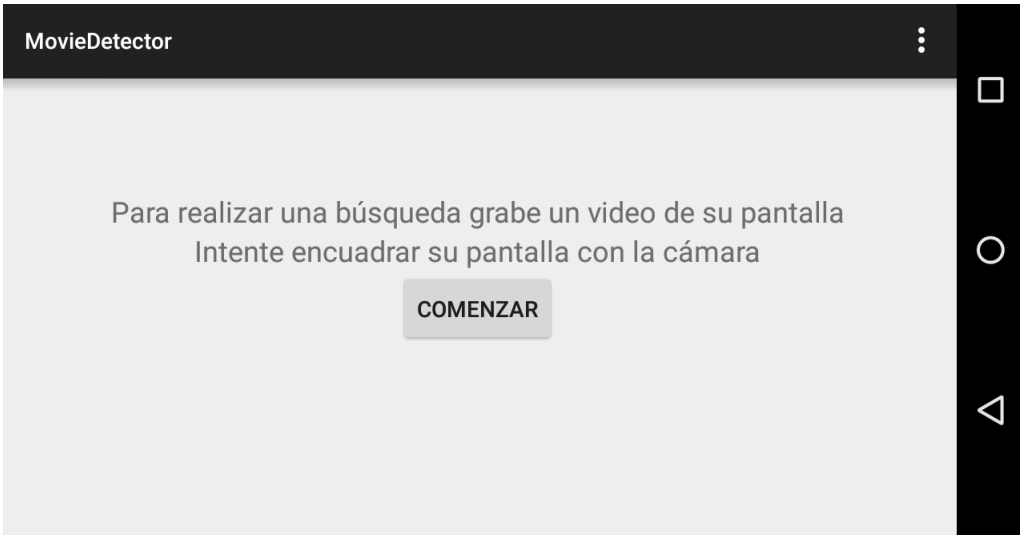


Figura 4.2: Vista del cliente el inicio de la versión centralizada.

respuesta del servidor. Una vez se termina el envío de datos al servidor se cambia la interfaz por una pantalla inicialmente vacía donde se mostraran los resultados.

Interfaz de resultados

Para mostrarle los resultados de la búsqueda al usuario se implementó el fragmento `QueryResultsFragment` que implementa la interfaz `ResponseHandler` definida en la parte anterior. Inicialmente el fragmento muestra un texto indicando que se están esperando los resultados de la búsqueda, cuando se obtiene respuesta del servidor se ejecuta alguno de los métodos de la interfaz. Si el servidor responde con un error se ejecuta el método `onFailure` que muestra un mensaje al usuario informándolo del error. De lo contrario se ejecuta el método `onSuccessResponse` que recibe un String con los resultados del servidor codificados como JSON. Los resultados son deserializados usando la biblioteca GSON[4] de manera que cada candidato de video identificado se convierte en un objeto de la clase `Detection` y se ingresa en una lista. El fragmento contiene un `ListView`, un elemento de interfaz de Android para mostrar listas de objetos. Se programó la clase `QueryResultAdapter` que recibe la lista de detecciones y crea una vista con cada una como muestra la figura 4.3. El diagrama de la figura 4.4 muestra las clases descritas en la sección anterior, detallando sus contenidos e interacciones.

4.1.2. Servidor

El servidor debe recibir una petición Post que contenga un video y el tipo de descriptor que se desea usar en la búsqueda. Debe usar el programa P-VCD para calcular los descriptores especificados y realizar la búsqueda para identificar el video original, finalmente debe entregarle al cliente en formato JSON una lista de los candidatos que encuentre.

[4]GSON, biblioteca para convertir objetos Java a sus representación en JSON y vice-versa: `https://github.com/google/gson`

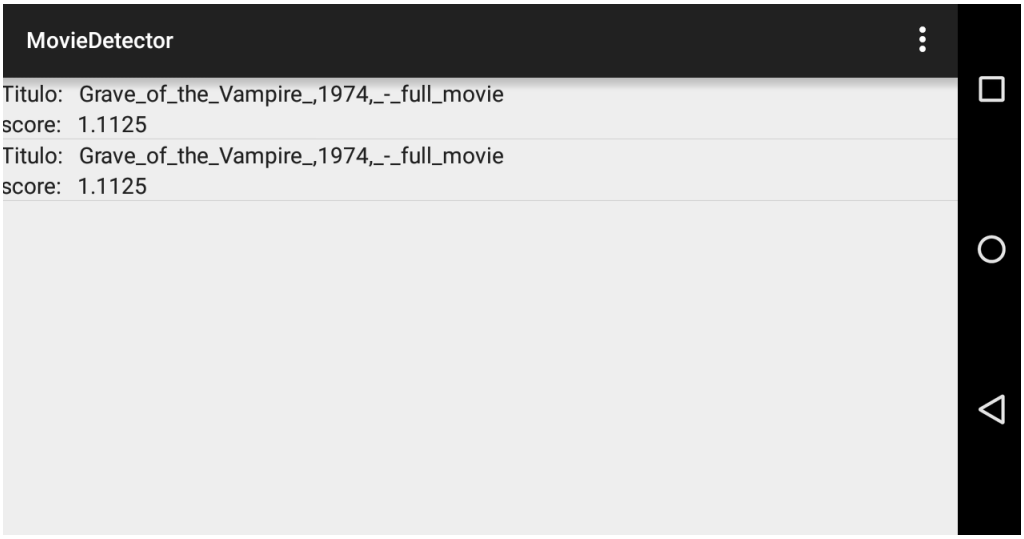


Figura 4.3: Vista del cliente mostrando los resultados de la búsqueda.

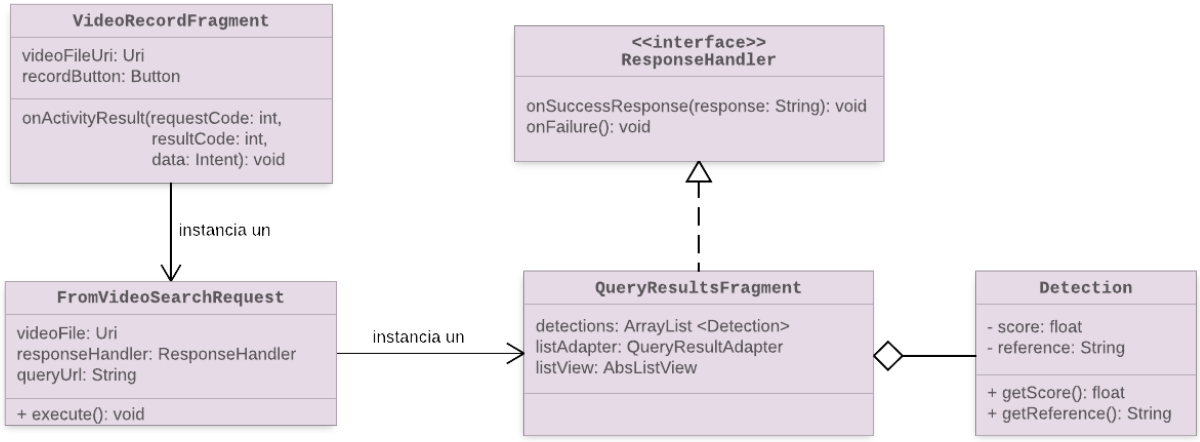


Figura 4.4: Diagrama de clases del cliente Android centralizado.

Para recibir peticiones Post se usó el framework Flask, que permite programar servicios web con muy poco código extra. El siguiente ejemplo muestra todo el código necesario para definir y responder una petición Post:

```
1 from flask import Flask
2 app = Flask(__name__)
3
4 @app.route("/api/example/post", method=['POST'])
5 def hello():
6         return "Hello World!"
7
8 if __name__ == "__main__":
9         app.run(host='0.0.0.0')
```

La primera línea importa el módulo Flask, luego se define el objeto `app` que representa el

contexto global de la aplicación. Para definir una *request* define una función con la anotación `@app.route`, explicitando la ruta y el tipo de petición. La función resuelve peticiones a la ruta especificada, dentro de la función se puede utilizar el objeto `app` para obtener el contenido de la petición. El string retornado por la función se convierte automáticamente en la respuesta del servidor.

Para servir la petición Post requerida se creó la función `search_by_video_file` que escucha la URL host/api/search/search_by_video_file.

Comunicación con P-VCD

Como se detalló en capítulos anteriores P-VCD consta de múltiples módulos, cada uno con sus propio ejecutable que puede ser llamado por línea de comandos. Para usarlo en conjunto con el servidor para realizar el cálculo de descriptores y la búsqueda fue necesario encontrar la manera de ejecutar un programa externo desde Python. Se encontró el módulo `subprocess`[5] de Python que permite lanzar nuevos procesos pesados de manera síncrona o asíncrona y obtener sus resultados. En particular se utilizó la función `call` de este módulo, que recibe una lista de strings con el nombre del programa y sus argumentos, lanza un nuevo proceso y espera a que termine, retornando el código de retorno del proceso.

Usando lo anterior se creó el módulo `PVCD_Wrapper` que envuelve llamadas a las distintas etapas del proceso de búsqueda de P-VCD en funciones de Python, a continuación se decriben las funciones usadas.

- **compute_descriptors**:
  La función recibe un archivo de video, un string correspondiente al tipo de descriptor a utilizar y un alias para el descriptor. Con esto realiza las llamadas necesarias para la extracción de descriptores de P-VCD, esto es, se llama a `pvcd_db -new` para crear una nueva base de datos con el video correspondiente, luego se llama a `pvcd_db -segment` para calcular su segmentación y finalemente se llama a `pvcd_db -extract` para extraer el descriptor especificado.
- **new_search_profile**:
  Esta función recibe dos strings, uno con el nombre de la base de datos con los descriptores de consulta, y otro con el alias del descriptor y realiza la llamada a `pvcd_search -new` para crear un nuevo perfil de búsqueda.
- **search**:
  Esta función no recibe argumentos, llama a `pvcd_search -ss` que toma los descriptores del video de consulta y para cada uno busca sus vecinos más cercanos en la base de datos de referencia.
- **detect**:
  Esta función no recibe argumentos, llama a `pvcd_detect` que usa la información de los vecinos más cercanos del paso anterior para detectar secuencias al mismo video. P-VCD escribe en un archivo detections.txt la información de los videos que se consideran candidatos a ser el video original. La función lee este archivo y retorna una lista con el

[5]Documentación del módulo subprocess: `https://docs.python.org/2/library/subprocess.html`

nombre de cada video y el puntaje de confianza que le asigna el sistema.

Usando estas funciones el código de la función `search_by_video_file` queda como sigue:

```
1  @app.route("/search/api/search_by_video_file", methods=['POST'])
2  def search_by_video_file():
3    if request.method == 'POST':
4      uploaded_file = request.files['uploaded_file']
5      if uploaded_file and allowed_file(uploaded_file.filename):
6        db_name = 'query'
7        descriptor = request.form['descriptor']
8        alias = request.form['alias']
9        file_path = save_video_file(uploaded_file)
10       PVCD_Wrapper.compute_descriptors(file_path, descriptor, alias)
11       PVCD_Wrapper.new_search_profile(db_name, alias)
12       PVCD_Wrapper.search()
13       detections = PVCD_Wrapper.detect()
14
15       return jsonify(detections=detections)
```

Primero se verifica que la petición sea de tipo Post y que se haya recibido un video con formato válido. Luego la función ejecuta secuencialmente los pasos de la búsqueda y retorna los resultados usando la función `jsonify` de Flask, que codifica una lista de Python como JSON.

Esta versión del sistema destaca por su simplicidad y facilidad de implementar. Además el sistema resultante es flexible, puesto que tiene a su disposición todos los descriptores disponibles en P-VCD y solo se requieren cambios de configuración del servidor para cambiar el descriptor usado por el sistema. Sin embargo esta arquitectura presenta dos grandes problemas, por un lado enviar el video completo al servidor significa un considerable gasto de la red de datos del dispositivo. Por otro lado, el realizar todas las operaciones en el servidor se crea un posible cuello de botella que limita la escalabilidad del sistema. Ambos problemas se originan por el uso ineficiente de los recursos computacionales del sistema, ya que no se está aprovechando la capacidad de computo del dispositivo móvil. En la siguiente sección se propone una arquitectura alternativa que busca solucionar estos problemas.

4.2. Sistema Distribuido

La figura 4.5 ilustra la arquitectura de esta versión del sistema. El cálculo de descriptores para la búsqueda es realizado por el cliente Android (1) utilizando las herramientas para computación paralela descritas en secciones anteriores. Una vez calculados, se envían solo estos descriptores al servidor (2) el cual los usa para realizar la búsqueda por similitud (3) y luego comunica los resultados de vuelta al cliente (4).

A continuación se detalla la implementación de tanto el cliente Android como el servidor y se comentan las diferencias con respecto a la versión descrita anteriormente.

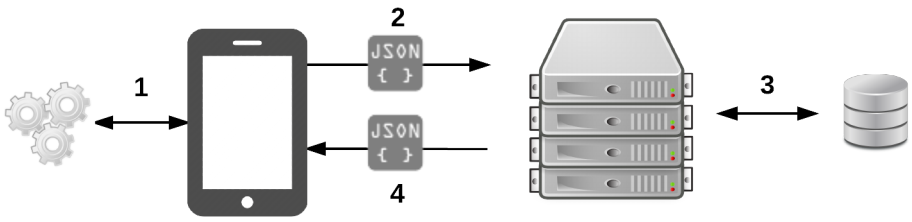


Figura 4.5: Diagrama de la arquitectura del sistema distribuido.

4.2.1. Cliente Android

A diferencia de la versión anterior del sistema, en esta versión el cliente debe calcular los descriptores del video de consulta y realizar un petición Post al servidor enviando solo los descriptores y no el archivo de video completo. Al igual que la versión centralizada, una vez obtenida la respuesta del servidor debe mostrarle los resultados de la búsqueda al usuario. Se implementó esta versión cambiando el módulo de grabación de la cámara y el de conexión con el servidor, además se añadió un módulo de cálculo de descriptores.

Grabación

En esta versión ya no es necesario obtener un archivo de video resultante, sino que es necesario realizar un procesamiento del video. Dada esta diferencia se decidió reimplementar este módulo, creando un nuevo fragmento `CameraPreviewFragment` de manera que se puedan obtener frames de la cámara mientras se graba el video para poder realizar el cálculo en tiempo real. La interfaz gráfica del fragmento `CameraPreviewFragment` consta de tres elementos que

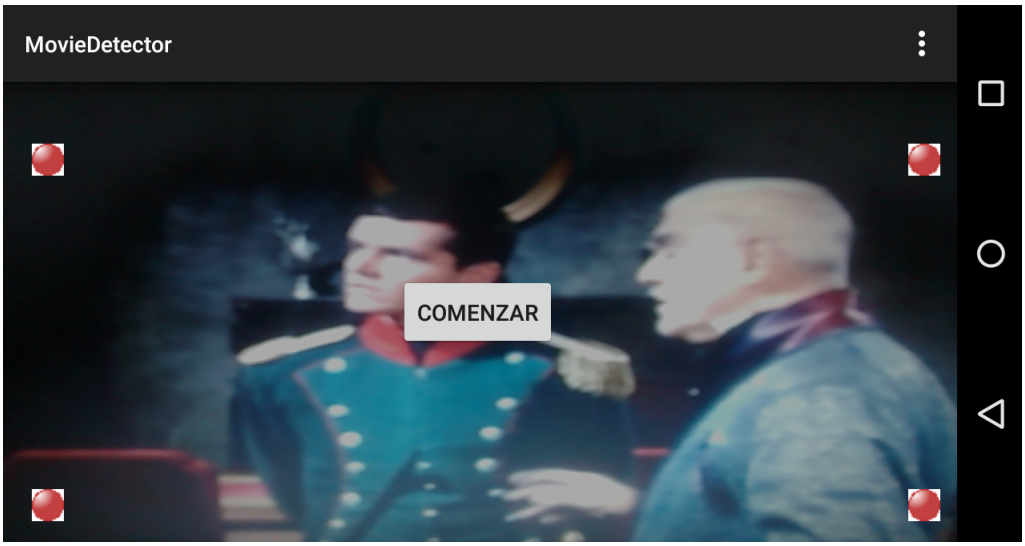


Figura 4.6: Vista inicial del cliente en la versión distribuida.

podemos identificar en la figura 4.6, el preview de la cámara, los cuatro puntos con los que

el usuario puede marcar los límites de la pantalla y finalmente un botón para comenzar la grabación. A continuación se detallan aspectos de la implementación de cada uno de estos elementos.

A diferencia de la versión anterior, en que la cámara era manejada por una aplicación predefinida, ahora es necesario manejar los eventos del ciclo de vida de la cámara. Para esto se utilizó la funcionalidad de la clase `Camera`[6] de Android. Esta clase permite mostrar un preview y obtener frames para procesarlos en tiempo real. Se programó la clase `CameraContainer`, que representa un elemento gráfico que contiene el preview. La clase además encapsula las llamadas al sistema operativo necesarias para controlar la cámara, a través de las funciones `startCameraPreview`, `stopCameraPreview` y `releaseCamera`.

Se programó la clase `CropView` que controla los cuatro puntos vistos en la interfaz. Estos puntos permiten al usuario ajustar el video al tamaño de su pantalla, esto permite un mejor desempeño de los descriptores utilizados dado que todo lo grabado corresponde al video de consulta y no a objetos exteriores como el marco de la pantalla. La clase contiene cuatro puntos representados por objetos de la clase `Point` e implementa el método `onTouchEvent`, que permite ser notificado por el sistema operativo cuando el usuario toca la pantalla. Cuando esto sucede se verifica si el usuario está arrastrando alguno de los puntos y se ajusta la interfaz gráfica para reflejar el cambio. Al procesar la imagen para calcular sus descriptores se especifica la posición de los cuatro puntos de manera que el cálculo solo considere la sección de la imagen que está dentro de sus límites.

Por último el bóton para iniciar la grabación corresponde a un objeto de la clase `Button` de Android, al presionar el botón se utiliza el método `setPreviewCallback` del cameraContainer para registrar un *listener* que será llamado por el sistema cada vez que la cámara obtiene un nuevo frame de video, esto será usado por el siguiente módulo para realizar el cálculo de descriptores. La interfaz gráfica cambia para reflejar que se están calculando descriptores, el botón de iniciar se reemplaza por una barra de progreso que muestra cuanto tiempo de grabación queda como muestra la figura 4.7.

La figura 4.8 muestra un diagrama que detalla las clases utilizadas descritas anteriormente.

Cálculo de descriptores

La figura 4.9 muestra las clases que modelan los descriptores. Se creó la clase abstracta `ImageDescriptor` que corresponde a un descriptor de imágenes genérico, contiene un arreglo de doubles, un timestamp y un frameNumber que identifican el frame al cual corresponde el descriptor. Los métodos abstractos `getType` y `getSerializedOptions` son usados para serializar el descriptor al enviarlo al servidor. La clase tiene tres implementaciones concretas que corresponden a los tres tipos de descriptores que el sistema puede calcular.

Por otro lado está la clase `VideoDescriptor` que representa un descriptor de video, esta clase contiene una lista de descriptores de imágenes y cuenta con los métodos `addDescriptor` que recibe un descriptor de imagen y lo añade a la lista, y el método `toJson` que serializa el

[6] `http://developer.android.com/guide/topics/media/camera.html`

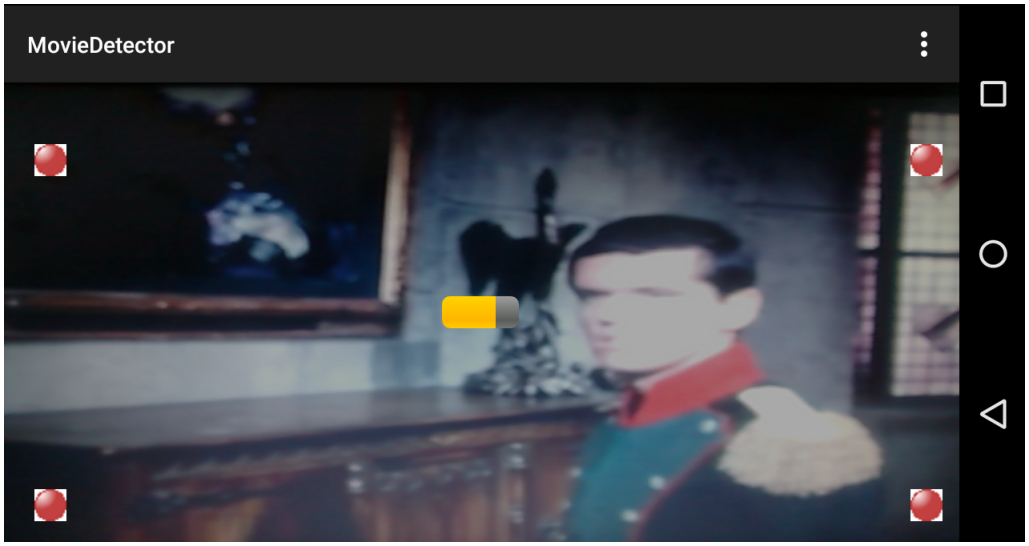


Figura 4.7: Vista inicial del cliente en la versión distribuida.

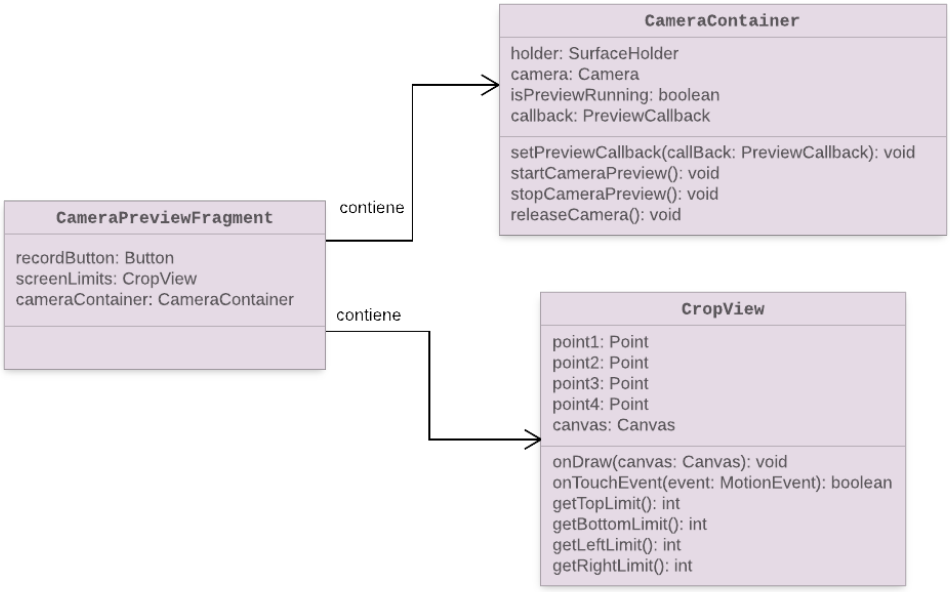


Figura 4.8: Diagrama de clases del módulo de grabación del sistema distribuido.

descriptor en formato JSON para enviarlo al servidor.

El cálculo de descriptores de imágenes es realizado por clases que implementan la interfaz `DescriptorExtractor`. Esta interfaz define el método `extract`, que recibe el arreglo de bytes con los datos de un frame de video y el timestamp y frameNumber del frame correspondiente y entrega su descriptor de imagen. La figura 4.10 muestra el diagrama de clases con las clases que implementan la interfaz. El diagrama además muestra la clase `VideoDescriptorExtractor`, esta clase se usa en conjunto con los extractores para cálcular los descriptores de un video.

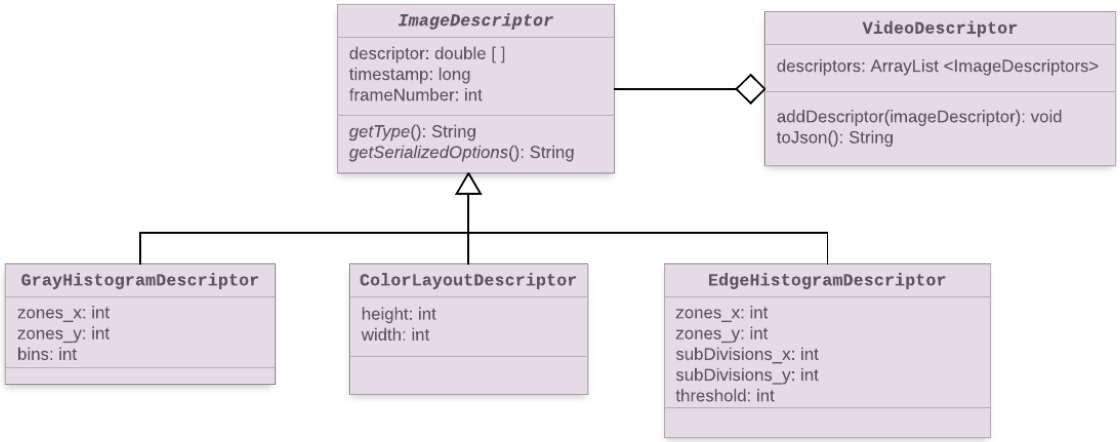


Figura 4.9: Diagrama de las clases usadas para modelar descriptores.

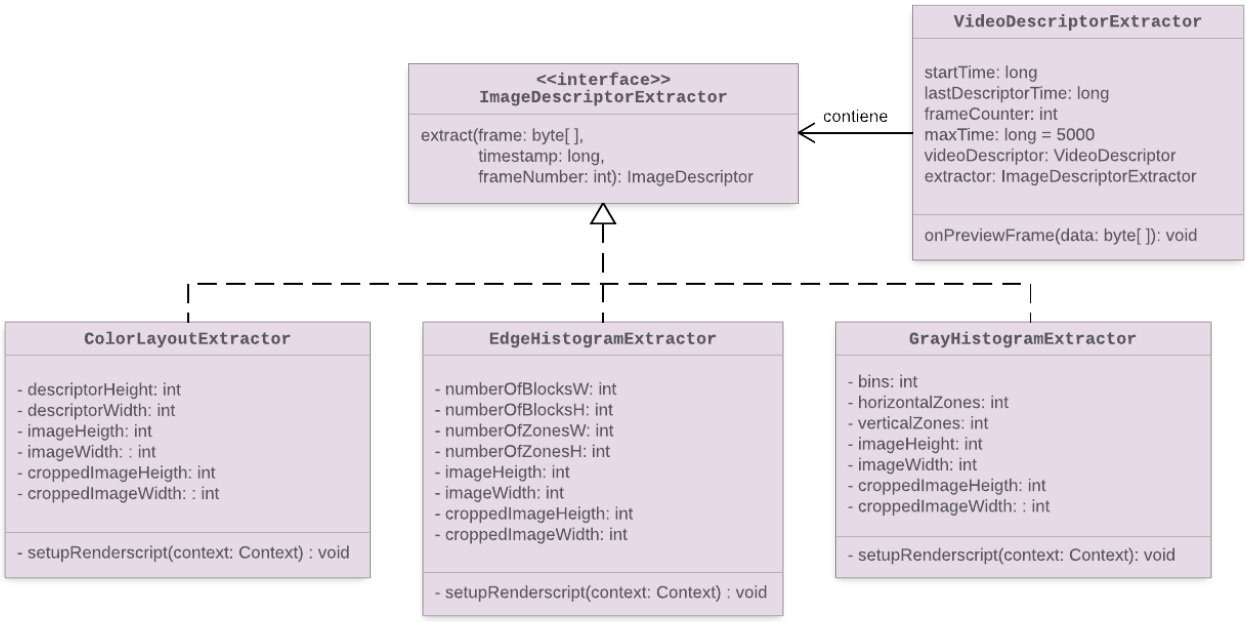


Figura 4.10: Diagrama de clases de extractores de descriptores.

VideoDescriptorExtractor implementa la interfaz `PreviewCallback` con lo que se puede registrar para obtener frames de la cámara. Cada frame se recibe como un arreglo de bytes en formato YUV. En este cada pixel se divide en valores de *luma* (Y) que indica la intensidad de luz y *chroma* (UV) que indican el color. Específicamente se usa el formato YUV420 NV21 [2], que establece un ordenamiento de los valores como indica la figura 4.11. Los primeros $W \times H$ bytes corresponden a los valores de *luma* de cada pixel, donde W y H corresponden al ancho y alto de la imagen.

VideoDescriptorExtractor intenta procesar frames a un ritmo constante de 4 frames por segundo. Al recibir un nuevo frame primero verifica si se está procesando un frame anterior,

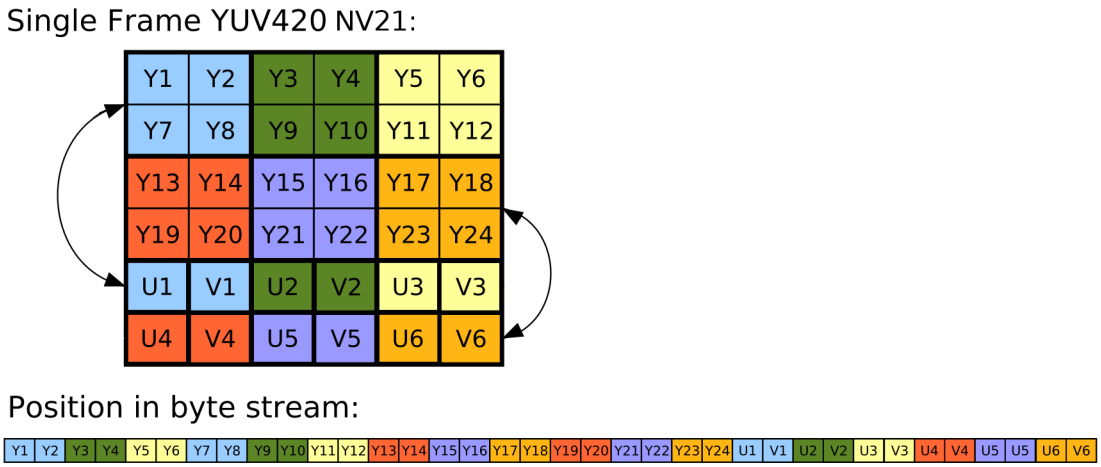


Figura 4.11: Especificación del formato YUV420 NV21.

de ser así se ignora el frame recibido sin procesarlo, de lo contrario se revisa el tiempo transcurrido desde el último frame procesado. Si ha pasado muy poco tiempo desde el último frame (menos de 0.25 segundos) entonces se ignora y se sigue esperando. Para procesar un frame se usa alguno de los extractores definidos en la figura 4.10, al invocar su método `extract` se obtiene un objeto de tipo `ImageDescriptor`, este descriptor de imagen se añade a un descriptor de video mantenido por el VideoDescriptorExtractor usando el método `addDescriptor` de la clase `VideoDescriptor`.

A continuación se explica en detalle el funcionamiento de cada extractor de descriptores de imágenes implementado.

`GrayHistogramExtractor` maneja el cálculo de histogramas de grises por zona. El algoritmo para calcular el descriptor consiste de cuatro etapas:

1. Transformar imagen de color en formato YUV a escala de grises.
2. Recortar imagen para ajustarse a los límites establecidos por el usuario.
3. Calcular histograma, esto es, contar la cantidad de pixeles con cierta intensidad en cada zona.
4. Normalizar histograma.

La primera etapa se logra extrayendo los primeros $W \times H$ bytes del arreglo de bytes, donde W y H son el ancho y alto de la imagen, en el formato YUV utilizado por Android estos corresponden a la intensidad de luz de los pixeles de la imagen, es decir, sus valores de gris. La extracción de bytes se realiza con el método `copy2DRangeFrom` de la clase `Allocation` de RenderScript, que recibe un arreglo de bytes, valores de offset horizontal y vertical, alto y ancho de la imagen y extrae los valores del arreglo para poblar una Allocation. Al final de esta etapa se obtiene una Allocation con los valores de gris de la imagen.

Para la segunda etapa nuevamente se usa el método `copy2DRangeFrom`, esta vez se usa una versión del método que recibe una Allocation, y valores de offset, alto y ancho para extraer

un subconjunto de valores de la Allocation recibida. Al final de esta etapa se obtiene una nueva Allocation distinta a la de la etapa anterior, que contiene solo los valores de gris de la zona demarcada por el usuario.

La tercera etapa requiere el uso de RenderScript para realizar el cómputo. Esto se logró con el script `GrayZoneHist.rs` que itera sobre cada pixel de la imagen, calcula su bin correspondiente basado en su valor de gris y su posición y aumenta en 1 el valor del bin. Para aumentar el valor del bin se debe usar una primitiva de sincronización, de lo contrario podrían ocurrir *data races* dado que el script se ejecuta en parlelo. Se utilizó la función `rsAtomicInc` que recibe una dirección de memoria e incrementa su valor en 1 de forma atómica. La allocation de salida del script contiene el histograma sin normalizar. El código del script `GrayZoneHist.rs` se encuentra adjunto en la sección de Anexos.

En la cuarta etapa primero se extrae el contenido de la Allocation que contiene el histograma y se copia a un arreglo de ints. Esto se logra con el método `copyTo` de la clase `Allocation`. Luego se crea un nuevo arreglo de doubles que contendrá el histograma normalizado. Se itera sobre el histograma dividiendo cada valor por la cantidad de pixeles en su zona correspondiente y asignando el resultado al histograma normalizado. Finalmente se crea un objeto `GrayHistogramDescriptor` con el histograma normalizado y se retorna.

`ColorLayoutExtractor` maneja el cálculo del descriptor de distribución de colores. El algoritmo para calcularlo toma los siguientes pasos:

1. Transformar imagen de formato YUV a formato RGB.
2. Recortar imagen para ajustarse a los límites establecidos por el usuario.
3. Calcular el promedio de color por canal en cada zona.

El primer paso se logra usando la clase `ScriptIntrinsicYuvToRGB` de RenderScript. Esta clase pertenece a la biblioteca de RenderScript y permite decodificar una imagen en YUV y pasarla a formato RGB, solo es necesario crear Allocations de entrada y salida del tamaño correspondiente. El resultado es una Allocation que contiene la imagen en formato RGB.

El segundo paso se logra de la misma manera que en el extractor anterior, usando el método `copy2DRangeFrom` de la clase Allocation. El resultado de este paso es un Allocation que contiene los valores RGB de la sección de la imagen indicada por el usuario.

El tercer paso se realiza usando RenderScript con el script `color_layout.rs`. Este script itera sobre cada pixel de la imagen y separa el pixel en sus valores de rojo, verde y azul. Cada valor es sumado a un contador correspondiente a su zona. Para realizar esta suma se usó la función `rsAtomicAdd` de RenderScript que permite realizar sumas de manera atómica. La Allocation de salida contiene la suma de los valores de rojo, verde y azul de los pixeles de cada zona. Para calcular el valor promedio de cada color basta dividir el valor obtenido por el script por la cantidad de pixeles de cada zona. Para esto se extrae el contenido de la Allocation usando el método `copyTo` al igual que en el extractor anterior. Luego se itera por cada valor y se divide por la cantidad de pixeles de la zona. Finalmente se crea un objeto `ColorLayoutDescriptor` y se retorna. El código del script `color_layout.rs` se encuentra adjunto en la sección de Anexos.

La clase `EdgeHistogramExtractor` se encarga del cálculo del descriptor de histograma de bordes. Los pasos del algoritmo para el computo son los siguientes:

1. Transformar imagen de color en formato YUV a escala de grises.
2. Recortar imagen para ajustarse a los límites establecidos por el usuario.
3. Reducir la imagen a una cantidad fija de bloques, donde el valor de cada zona corresponde al promedio del valor de los pixeles dentro de la zona.
4. Calcular la energía de cada filtro aplicándolo sobre grupos de cuatro bloques. Se selecciona el mayor y se revisa si la energía correspondiente excede un valor umbral.
5. Calcular el histograma de bordes, que indica por zona cuantos grupos de bloques presentan cada uno de los tipos de bordes posibles.
6. Normalizar el histograma.

Las primeras dos etapas son idénticas a las primeras etapas del cálculo del descriptor de histograma de grises y se realizan de la misma forma. El resultado al final de la segunda etapa es un Allocation con los valores de gris correspondiente a la sección de la imagen seleccionada por el usuario.

La tercer etapa es similar al cálculo del descriptor de distribución de colores y se realiza de forma análoga. El script `reducer.rs` divide la imagen en bloques e itera sobre cada pixel sumando su valor de gris al de la zona correspondiente. El resultado del script es una Allocation con la suma de los pixeles en cada zona. Al igual que en el descriptor anterior se extraen los resultados a un arreglo usando el método `copyTo` y se divide cada valor por la cantidad de pixeles por zona, con esto se obtiene un arreglo con el valor de gris promedio en cada zona.

Para la cuarta etapa se toma el arreglo resultante del paso anterior y se usa para poblar una Allocation. Estos datos se usan como entrada para el script `borderDetector.rs`, que toma grupos de cuatro bloques y calcula el valor de energía de cada filtro, selecciona el mayor y se prueba si pasa el valor umbral. La Allocation de salida contiene un número del 0 al 5 por cada bloque, si el filtro de mayor energía pasó el valor umbral, el valor irá de 0 a 4 indicando el indice del filtro de mayor energía, en caso que el mayor no supere el umbral se marcará con un 5.

La quinta etapa es similar al cálculo de histograma de grises. El script `edgeHist.rs` itera sobre cada bloque revisando cual fue el filtro de mayor energía, para cada zona se cuenta la cantidad de bloques con cada tipo de borde. Si para algún bloque el valor en la Allocation de entrada es 5 (correspondiente a los bloques en que la energía no excedió el umbral) no se considera para la construcción del histograma. La Allocation de salida contiene el histograma con la cantidad de bloques con cada tipo de borde por cada zona de la imagen.

La última etapa es idéntica a la última etapa de la construcción del descriptor de histograma de grises. Se extraen los valores de la última Allocation a un arreglo y se normaliza dividiendo cada valor por la cantidad de pixeles presentes en cada zona. El arreglo resultante contiene el histograma normalizado, que se usa para construir un objeto `EdgeHistogramDescriptor` y retornarlo.

Todos los script usados por EdgeHistogramExtractor se adjuntaron en la sección de anexos.

El proceso de cálculo de descriptores se realiza por un tiempo determinado por el VideoDescriptorExtractor. Una vez terminado el proceso se procede a enviar el descriptor de video resultante al servidor para realizar una consulta.

Conexión con el servidor

El módulo de conexión con el servidor es análogo al de la versión anterior. Se implementó la clase `FromDescriptorsVideoSearch` que maneja el envío de datos al servidor. A diferencia de la versión anterior esta clase recibe un objeto de la clase `VideoDescriptor` para reapizar la búsqueda. El descriptor de video puede ser serializado con su método `toJson`, este método usa los métodos `getType` y `getSerializedOptions` de los descriptores de imágenes para obtener propiedades del descriptor necesarias para la búsqueda como el tipo de descriptor y su tamaño. Una vez serializado se envía al servidor usando la clase `HttpUrlConnection` de Java. Al igual que la versión anterior, una vez que se termina de enviar datos al servidor se cambia la interfaz gráfica de la aplicación, a una pantalla inicialmente vacía donde se mostrarán los resultados.

Interfaz de resultados

Este módulo no presenta diferencias con respecto a la versión anterior, se reutilizó la clase `QueryResultsFragment` para mostrar los resultados obtenidos del servidor.

La figura 4.12 muestra un diagrama de clases resumido con las clases que componen la versión distribuida del cliente.

4.2.2. Servidor

La implementación de la versión distribuida del servidor es análoga a la versión centralizada. Se creó la función `search_by_descriptors` que sirve la URL usada por el cliente para enviar los descriptores. Se reutilizaron las funciones `new_search_profile`, `search` y `detect` del módulo `PVCD_Wrapper`.

Dado que esta versión recibe los descriptores en vez del video completo basta con guardar los descriptores con el formato apropiado para realizar la búsqueda. Además de esto es necesario crear los archivos que identifican la base de datos del video de búsqueda, ya que a pesar que el video no está en el servidor (ni se utiliza en ningún paso de la búsqueda) P-VCD exige que exista una base de datos con su correspondiente segmentación. Se crearon funciones que imitan el funcionamiento de P-VCD, creando una base de datos, una segmentación y un archivo de descriptores a partir de la información recibida por el servidos. A continuación se

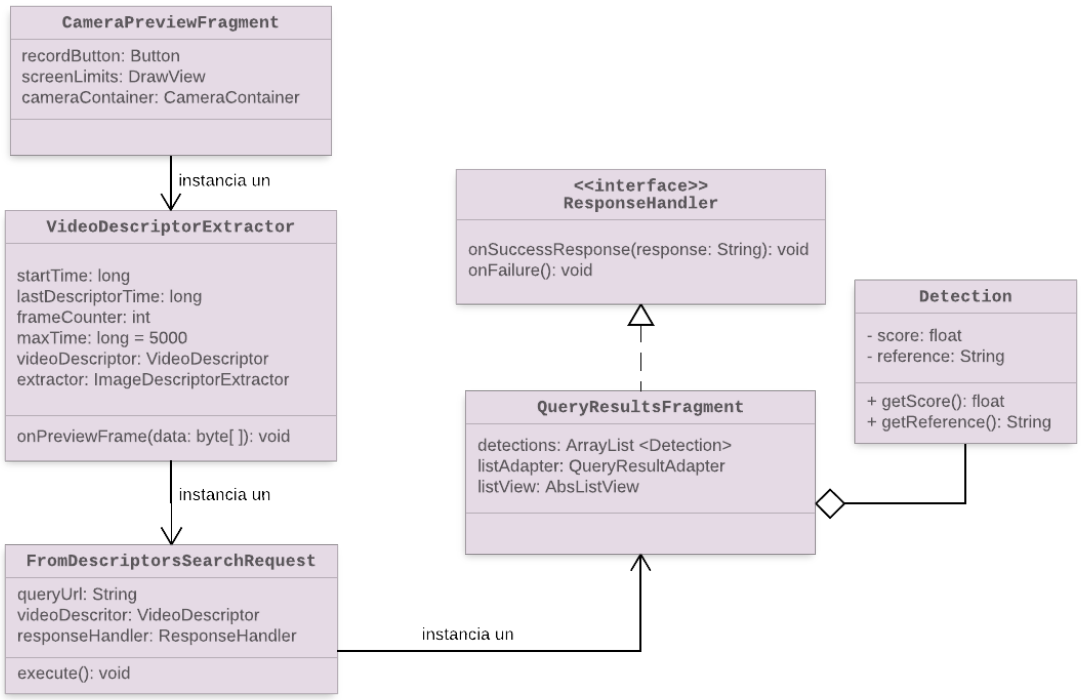


Figura 4.12: Diagrama de clases del cliente distribuido.

describen las nuevas funciones implementadas.

- `create_database`

  Esta función crea una base de datos vacía. Crea una carpeta con un archivo `files.txt`, normalmente este archivo contiene información del video, como la ruta a su archivo, tipo de archivo, etc. Esta información es usada por P-VCD para extraer la segmentación y los descriptores. Dado que estas operaciones no son necesarias en esta versión del sistema se puede crear el archivo `files.txt` con los headers apropiados pero sin información.

- `create_segmentation`

  Al crear una segmentación P-VCD extrae información de cuales frames serán usados por el sistema para calcular descriptores. Se especifica el número del frame correspondiente y el instante en que este aparece en el video. Toda la información necesaria para crear la segmentación es enviada al servidor junto con los descriptores del video. El cliente envía una lista con los números de frame y el instante en que aparecieron, la función usa esta información para crear el archivo `query.seg` requerido por el sistema.

- `write_descriptors`

  Los descriptores requieren dos archivos, uno con información sobre el tipo de descriptor y sus parámetros, y otro con los valores del descriptor mismo. La función crea ambos archivos, el primero, `descriptor.des`, se escribe con la información recibida desde el cliente con las opciones del descriptor. El archivo `query.bin` corresponde al contenido de los descriptores, este se escribe usando la clase `array`[7] de Python, esta clase permite escribir un arreglo numérico en un archivo binario con la función `to_file`.

[7] `https://docs.python.org/2/library/array.html`

La función `search_by_descriptors` ejecuta las tres nuevas funciones para crear los archivos necesarios para la búsqueda. Después de esto la búsqueda procede idénticamente a la versión anterior.

Esta versión del sistema soluciona los problemas expuestos en la sección anterior. Al enviar solo los descriptores de video al servidor se disminuye significativamente el uso de la red de datos, lo cual además reduce el tiempo de respuesta del sistema. Por otro lado se disminuye la carga del servidor al realizar la extracción de descriptores en el dispositivo móvil.

En el siguiente capítulo se exponen las pruebas realizadas con el sistema, que buscan determinar empíricamente las diferencias entre ambas versiones del sistema en términos de tiempo de respuesta y uso de recursos del sistema. Además se busca comparar la eficacia de los descriptores usados para realizar la búsqueda.

Capítulo 5

Evaluación Experimental

Este capítulo se divide en dos secciones. En la primera se describen los experimentos realizados para medir la eficiencia y eficacia de los sistemas implementados. En la segunda se muestran los resultados obtenidos.

5.1. Dataset y Experimentos

Para probar el sistema se recopiló una colección de videos correspondientes a películas de dominio público. Las peliculas fueron descargadas de un canal de Youtube que se especializa en subir recolectar y subir películas de dominio público[1]. En total se descargaron 119 películas que corresponden a alrededor de 110 horas de video. El dataset contiene películas publicadas desde 1920 y hasta la década de 1970 cuya protección de derechos de autor expiró, además de algunas más recientes que renunciaron a tales derechos. Debido a esto el dataset presenta videos con un amplio rango de tamaños y calidad de video, además de contener tanto filmes a color como en blanco y negro.

Para comparar la eficacia de los distintos descriptores se usó la versión distribuida de la aplicación. Esto debido que es la que cuenta con el preprocesamiento encargado de ajustar el video a los límites indicados por el usuario, sin esto los descriptores globales no pueden funcionar para videos de tamaño indeterminado, como en la base de datos recopilada.

Para realizar las pruebas se seleccionaron aleatoriamente 10 videos de la base de datos, y para cada uno se realizaron 4 consultas con cada descriptor, esto nos da un total de 120 consultas. Para cada consulta se revisó si el video correcto aparecía en cualquier lugar en la lista de resultados, de ser así se considera que la consulta obtuvo una respuesta correcta. Se estudió el comportamiento de los descriptores al variar la cantidad de objetos en la base de datos de referencia. Para aislar el efecto de la cantidad de películas en la base de datos se guardaron, para cada consulta, los descriptores enviados por el cliente, de esta forma al repetir la consulta con una base de datos mayor podemos estar seguro que cualquier cambio

[1] https://www.youtube.com/user/BestPDMovies

en la efectividad fue debido al cambio en la base de datos y no a cambios del descriptor mismo. Se repitieron las consultas variando la cantidad de películas en la base de datos entre 15 y 119. Para cada consulta se midió la proporción de resultados correctos en comparación con la cantidad total de consultas. Finalmente se repitió todo este proceso tres veces y se promediaron los resultados obtenidos.

Para probar la eficiencia del sistema distribuido en comparación con el centralizado se midió tanto el tiempo de respuesta total del sistema como la cantidad de datos enviada por el cliente, las pruebas se realizaron con el dispositivo móvil conectado a una red wifi.

El tiempo de respuesta se midió como el intervalo de tiempo desde que el cliente empieza a enviar datos hasta que termina de recibir la respuesta del servidor. Resulta interesante desglosar este tiempo en etapas para saber que operaciones son más costosas para el sistema. Para esto se midió en el servidor el tiempo gastado en distintas operaciones. En el caso del sistema centralizado se midió el tiempo usado por la extracción de descriptores y por la búsqueda en la base de datos. Para el caso del sistema distribuido se midió el tiempo usado escribiendo los archivos necesarios por P-VCD y los descriptores, así como el tiempo de la búsqueda en la base de datos. Finalmente se calculó el tiempo usado transmitiendo datos bajo el supuesto que:

$$Tiempo\ total\ \text{de}\ respuesta = Tiempo\ \text{de}\ transmisión + Tiempo\ total\ \text{del}\ servidor$$

Adicionalmente se midió el tiempo requerido por el dispositivo móvil para realizar el cálculo de cada tipo de descriptor.

5.2. Resultados

A continuación se presentan los resultados de los experimentos realizados y su interpretación.

5.2.1. Eficacia

La figura 5.1 muestra los resultados de medir la precisión del sistema, esto es, la proporción de resultados correctos con respecto al total de consultas. Podemos apreciar en la figura que solo uno de los descriptores obtuvo resultados favorables. El descriptor CLD alcanzó una precisión cercana al 50 % en los experimentos realizados, mientras que los otros no superaron en ningún caso el 5 %. Todos los descriptores mostraron una disminución en su precisión al aumentar el tamaño de la base de datos.

Para explicar la diferencia en el desempeño de los descriptores debemos fijarnos en la manera en que cada uno extrae características de la imagen y en como los afecta la distorsión provocada por grabar una pantalla con le celular. Un efecto importante que desfavorece al descriptor GHD es que al encuadrar incorrectamente el video se puede grabar accidentalmente parte del marco de la televisión por lo que al calcular el histograma se contarán pixeles

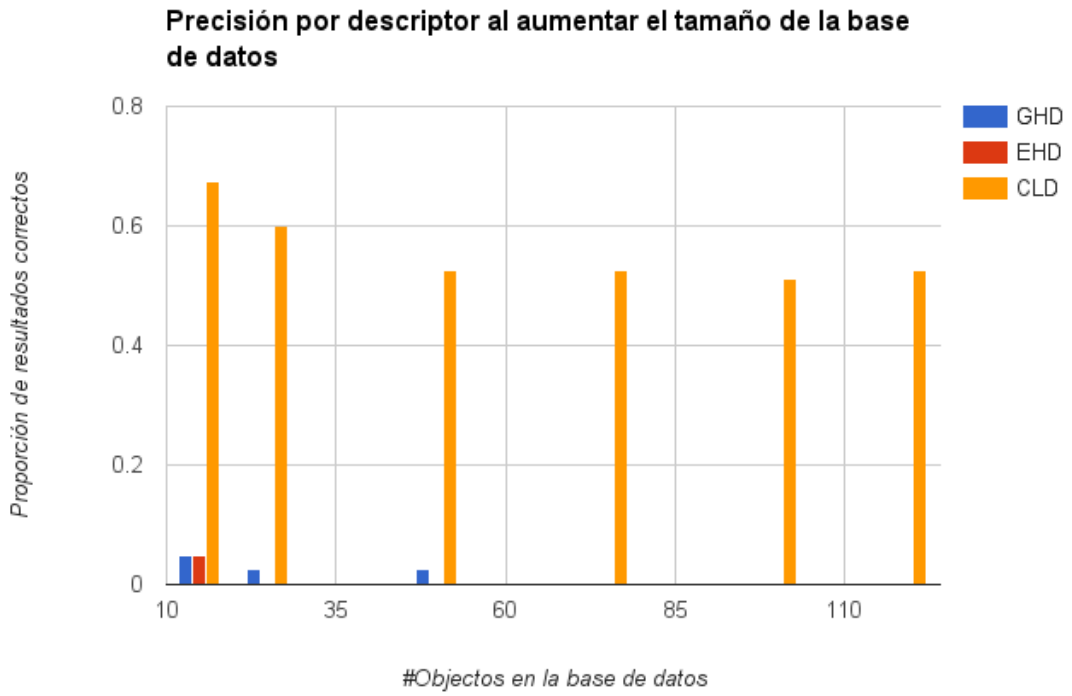


Figura 5.1: Gráfico con los resultados de medir la precisión cada descriptor

que no existen en el video original, más aún, el marco de una televisión típicamente es de un solo color, por lo que todos los pixeles extra se concentrarán en unos pocos bins, provocando en ellos un cambio significativo. Este cambio provoca un mayor efecto en el descriptor que cambiar levemente muchos bins dado que se comparan usando distancia euclideana. En contraste en el caso del CLD el mismo error (grabar pixeles que no existen en el video original) se suavizará debido a que se calculan promedios de pixeles por cada zona. En este caso además el efecto se distribuye sobre todas las zonas afectadas, a diferencia del GHD donde se concentra en unos pocos bins. En el caso del descriptor EHD, el efecto que provoca mayor error es probablemente la perdida de calidad de la imagen. Dado que el descriptor se basa en la detección de bordes cambios de calidad que difuminen la imagen hacen que sea más difícil detectar bordes y por ende que este descriptor pierda efectividad. En comparación nuevamente el descriptor CLD se muestra más resistente a tales cambios, pues solo toma en cuenta el valor promedio de los pixeles dentro de un área y no los detalles dentro de la misma.

Otra posible hipótesis para explicar el buen desempeño del descriptor CLD es que se vea favorecido por el dataset utilizado para las pruebas. La base de datos contiene muchas películas en blanco y negro, por lo que al usar el descriptor CLD (que usa el color de la imagen) todas las películas en blanco negro se encontrarán agrupadas en la base de datos, mientras que las películas a color estarán distribuidas de manera más dispersa, facilitando su identificación y mejorando en promedio la precisión del descriptor. Este efecto se observó en las pruebas, donde las películas a color en general obtuvieron mejores resultados que las en blanco y negro. De ser correcta esta hipótesis, cualquier descriptor que haga uso del color de la imagen debería verse favorecido.

5.2.2. Eficiencia

La Figura 5.2 muestra el tiempo requerido por la aplicación móvil para calcular cada tipo de descriptor. En todos los casos el tiempo fue considerablemente menor que el tiempo de refresco de la cámara, es decir el sistema es capaz de calcular el descriptor correspondiente a un frame de video antes que la cámara capture el siguiente frame.

Es posible observar que el cálculo del descriptor CLD requiere casi un 50 % más tiempo que el cálculo del descriptor EHD, esto inicialmente parece contraintuitivo puesto que el primer paso del cálculo de EHD es similar al cálculo de CLD como se explicó en la sección 4.2. Sin embargo la razón de estos tiempos de cálculo se vuelve evidente si se considera que el cálculo se realiza de forma concurrente. Si bien ambos descriptores presentan una etapa muy similar, reducir la imagen calculando para cada sección un color promedio, en el descriptor CLD la imagen se reduce a 10×10 secciones, mientras que en EHD se reduce a 64×64. Dado que el valor promedio se calcula usando una variable compartida a la cual accesan distintos *threads*, cuando la imagen se divide en más secciones habrá menos *threads* intentando acceder a la variable compartida, por lo que habrán menos sobrecosto producido por la sincronización entre *threads*.

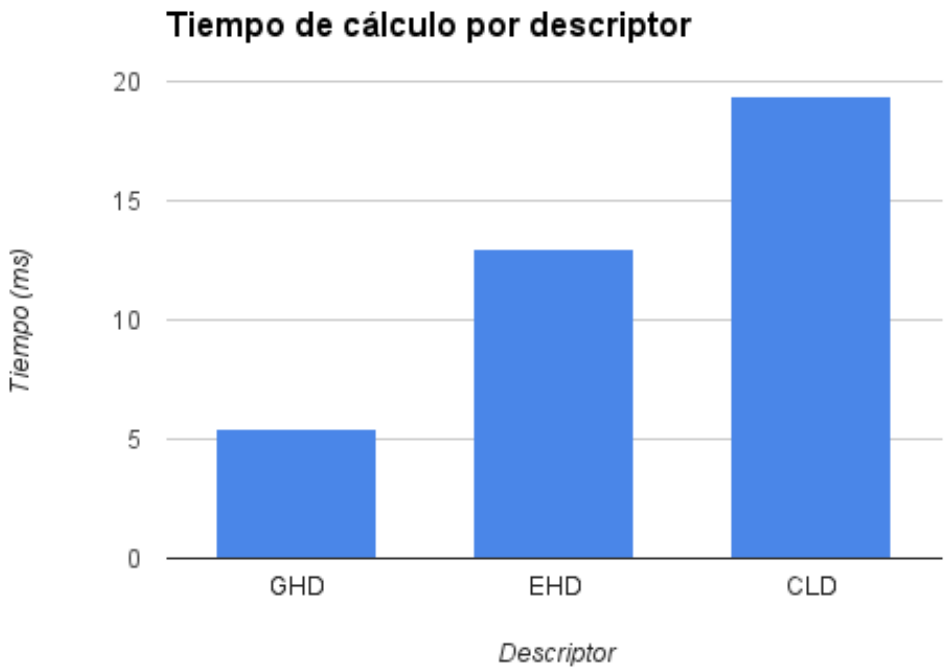


Figura 5.2: Gráfico con comparación de tiempos de cálculo de cada descriptor en el dispositivo móvil

La figura 5.3 muestra la medición del tiempo total de respuesta de cada implementación del sistema y para cada descriptor. De la figura podemos apreciar que para todos los casos la implementación distribuida requiere un tercio menos tiempo en comparación con la implementación centralizada.

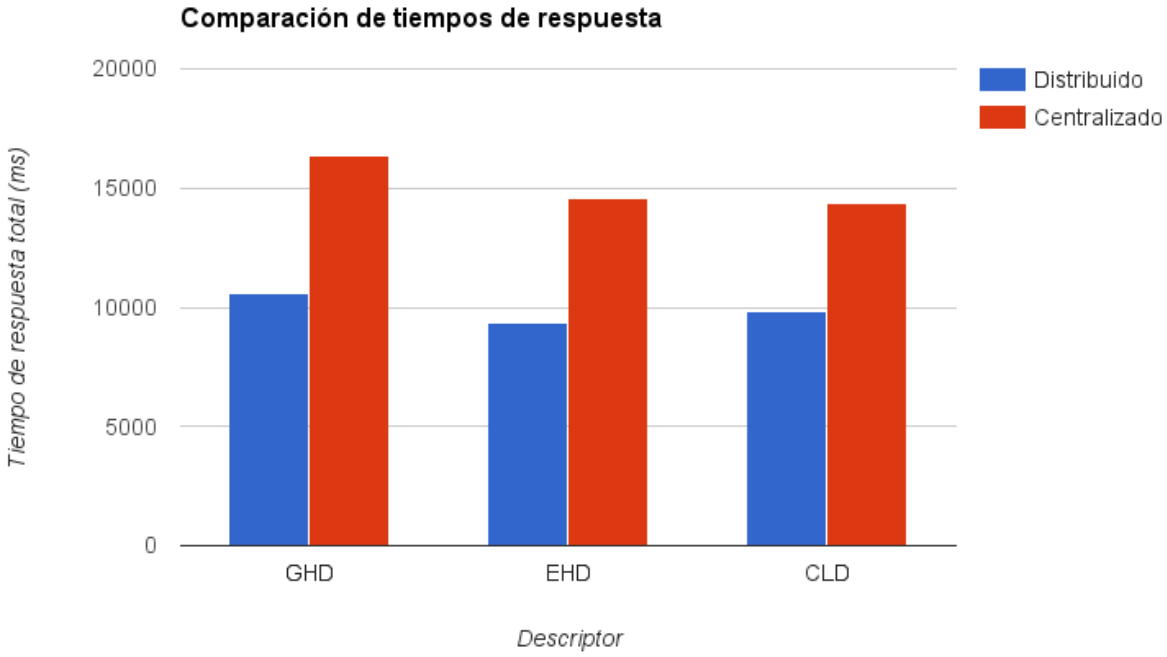


Figura 5.3: Gráfico con comparación de tiempos de respuesta de ambos sistemas

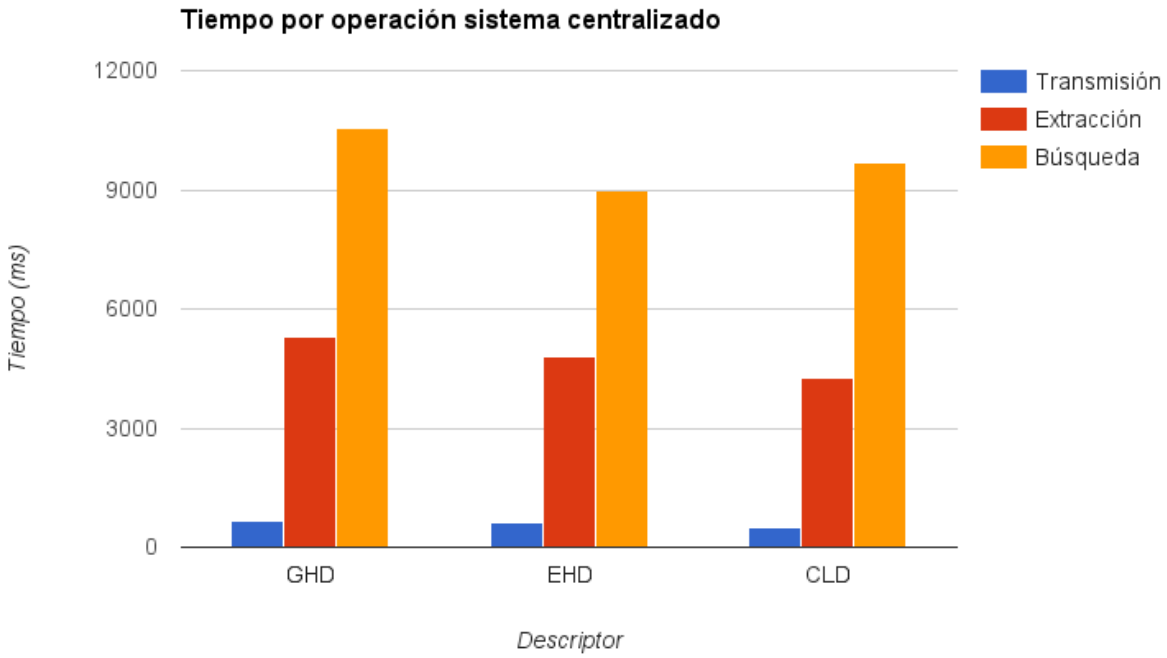


Figura 5.4: Gráfico con detalle de tiempo por operación del sistema centralizado

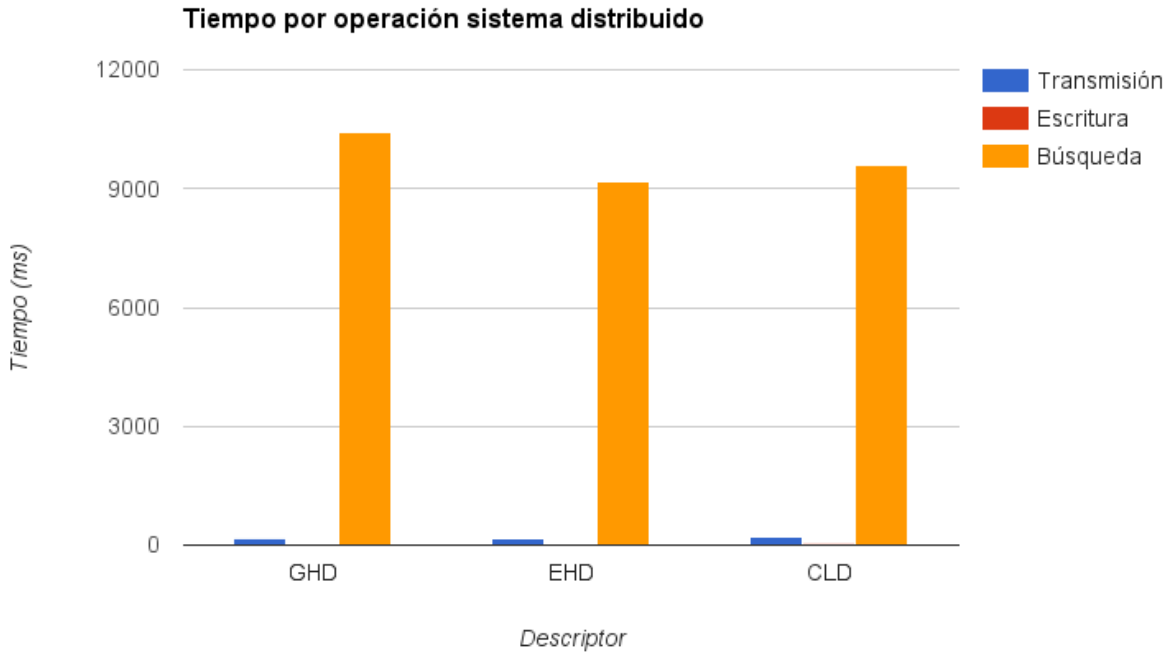


Figura 5.5: Gráfico con detalle de tiempo por operación del sistema centralizado

Esta reducción del tiempo de respuesta era de esperarse puesto que la implementación distribuida extrae descriptores mientras el video se graba, presentando un sobrecosto nulo. Para observar de manera más detallada las razones de la disminución del tiempo de respuesta se midió el tiempo requerido por operación. La figuras 5.4 y 5.5 muestran el detalle del tiempo de cada sistema. Podemos apreciar que en el caso centralizado cerca de un tercio del tiempo total del servidor es usado calculando los descriptores del video de búsqueda. En la versión distribuida este cálculo es reemplazado por simplemente escribir los descriptores recibidos a un archivo, lo cual toma una cantidad despreciable de tiempo en comparación con las otras operaciones. Por otro lado podemos apreciar la disminución del tiempo de transmisión debido a la reducción de la cantidad de datos que hay que enviar al servidor.

Las figuras 5.4 y 5.5 muestran que para ambas versiones de la aplicación la operación de búsqueda en la base de datos es la más cara en términos de tiempo de procesamiento. Por esto es interesante comprender el comportamiento del tiempo de búsqueda en la base de datos, en particular como varía al crecer el tamaño de la base de datos. La figura 5.6 muestra los resultados de medir el tiempo usado en la búsqueda al variar el tamaño de la base de datos. El tiempo varía al usar distintos descriptores pues cada uno tiene largo distinto, a más largo el descriptor más tiempo requiere la búsqueda. Del gráfico podemos deducir que el tiempo de búsqueda aumenta linealmente con el tamaño de la base de datos.

Finalmente analizamos la cantidad de datos enviados al servidor por cada versión del sistema. La tabla 5.1 muestra la comparación de los datos enviados por cada versión. Para la versión distribuida se detalla la diferencia entre cada descriptor. Para la versión centralizada no es necesario el detalle pues no hay diferencia en el video enviado al usar distintos des-

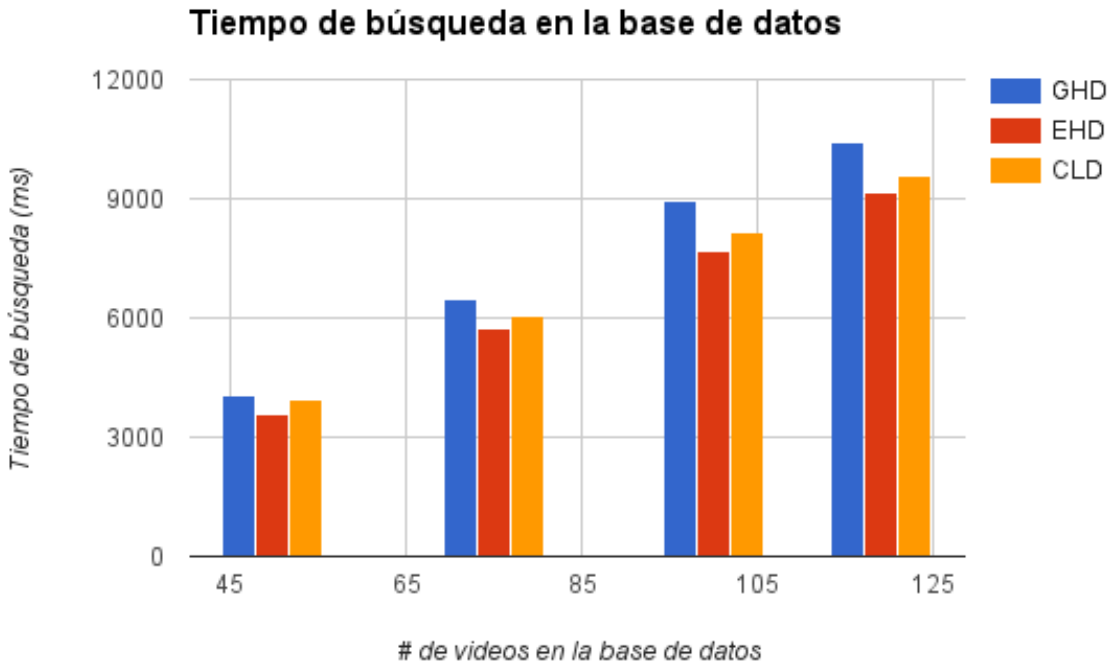


Figura 5.6: Gráfico con tiempos de consulta por descriptor al variar el tamaño de la base de datos

criptores. Podemos apreciar que se redujo significativamente la cantidad de datos enviados al servidor. La alternativa centralizada envía más de 6 megabytes por cada consulta, mientras que el sistema distribuido reduce esta cantidad al orden de 20 kilobytes dependiendo del tipo de descriptor utilizado. En términos prácticos esta reducción hace posible el usar la aplicación con la red de datos móviles sin incurrir en un gasto excesivo del usuario.

	Centralizado	Distribuido - GHD	Distribuido - EHD	Distribuido - CLD
KiloBytes Enviados	6389.64	22.69	11.98	19.60

Tabla 5.1: Cantidad de datos enviados al servidor por cada versión de la aplicación

Capítulo 6

Conclusiones

En este capítulo se presentan conclusiones finales sobre el trabajo realizado en la memoria, así como propuestas de trabajo futuro y posibles extensiones a la memoria.

6.1. Conclusiones

Se cumplió satisfactoriamente con los objetivos establecidos al inicio de la memoria, habiendo implementado un sistema de búsqueda de videos capaz de calcular descriptores usando un dispositivo móvil y enviarlos a un servidor que utiliza algoritmos de detección de copias para identificar el video de búsqueda.

Eficiencia del sistema

Entre las falencias que se vislumbraban en la versión centralizada del sistema estaba el alto uso de la red de datos del dispositivo móvil y la excesiva carga del servidor al realizar todos los cálculos necesarios para la búsqueda. Los resultados obtenidos de las pruebas confirman estas falencias y reafirman la necesidad de rediseñar el sistema para resolverlas. Por otro lado los resultados validan el diseño propuesto demostrando que la implementación del cálculo de descriptores en el dispositivo móvil reduce significativamente la cantidad de datos enviados por el cliente y la carga de trabajo del servidor. El envío de datos desde el cliente pasó de más de 6 megabytes en la versión centralizada del sistema a cerca de 20 kilobytes en la versión distribuida. La carga de trabajo del servidor también se vio reducida en la versión distribuida, al traspasar el cálculo de descriptores al dispositivo móvil se elimina una operación que representaba un tercio del trabajo total del servidor.

A pesar de la reducción en el tiempo de respuesta del sistema lograda al realizar la extracción de descriptores en el dispositivo móvil la operación más cara del sistema, la búsqueda en la base de datos, es demasiado lenta como para que el sistema sea comercialmente viable. Si se quiere continuar el desarrollo de la aplicación como producto comercial será necesario

realizar optimizaciones en el servidor que permitan reducir este tiempo de búsqueda. Ya que el tiempo de búsqueda crece linealmente con la cantidad de objetos en la base de datos, una optimización posible es extraer menos descriptores por segundo de video, esto disminuye fácilmente el tamaño de la base de datos. Sin embargo es necesario comprobar el efecto que esto tendría en la eficacia del sistema. Por otro lado se pueden estudiar maneras de optimizar el proceso de búsqueda de descriptores similares en la base de datos, por ejemplo haciendo uso de índices para espacios multidimensionales u otras técnicas que permitan reducir el tiempo de búsqueda.

RenderScript

El framework RenderScript de Android fue esencial para la implementación de los descriptores en el dispositivo móvil ya que permitió hacer uso eficiente de los recursos del teléfono. El framework permite al programador abstraerse de la arquitectura específica donde correrá el código dado que el sistema operativo se hace cargo de distribuir el código de manera eficiente para cada dispositivo. Los resultados obtenidos en términos del tiempo de procesamiento requerido para calcular cada descriptor revelan que usando RenderScript se logró extraer descriptores a una tasa mayor que el tiempo de refresco de la cámara. Este resultado indica que es posible implementar descriptores aún más complejos que los considerados en esta memoria haciendo posible distintas extensiones de la misma. Por otro lado resulta interesante analizar otros sistemas que podrían verse beneficiados al traspasar trabajo del servidor al cliente o si existen tareas comúnmente relegadas a ordenadores de escritorio por sus altos requerimientos computacionales que pudiesen ser implementados en dispositivos móviles. Sin embargo también es necesario mencionar los problemas encontrados con el uso de RenderScript durante el desarrollo de la memoria. El framework inicialmente fue desarrollado específicamente para su uso en computación gráfica y contaba con una API de rendereo de imágenes. Esta API fue deprecada pronto después de ser lanzada y el framework se relanzó como orientado a cualquier tipo de cálculo que requiera alto paralelismo. A pesar de haber sido lanzado junto con la API 4.2 de Android a principios de 2013 aún existe escasa documentación de su funcionalidad. La falta de información de parte del equipo desarrollador del framework hace que su futuro sea incierto, lo cual dificulta la decisión de utilizarlo para construir aplicaciones comerciales.

Eficacia de los descriptores

De los tres tipos de descriptores usados solo uno (CLD) mostró una efectividad satisfactoria mientras que los otros dos (GHD y EHD) mostraron resultados peores que el azar en los experimentos realizados. Es importante comprender las razones para el desempeño medido si se quiere mejorarlo en futuras versiones del sistema. Es posible por ejemplo analizar la distribución de cada tipo de descriptor en la base de datos de videos originales para confirmar si alguno presenta una distribución que naturalmente dificulte su diferenciación. Por otro lado se puede simular la transformación sufrida por el video original al ser grabado con la cámara del celular para investigar la resistencia de cada tipo de descriptor a esta transformación.

Finalmente también es posible implementar distintas estrategias para extraer descriptores del video. Por un lado se puede usar un enfoque de tipo *bag of words* usando descriptores locales. Esta estrategia se basa en extraer descriptores locales para crear un vocabulario de palabras visuales de la imagen, luego cada imagen se describe con un histograma de las *palabras* presentes en la imagen. Este tipo de estrategia se encuentra disponible en el programa P-VCD por lo que nuevamente solo sería necesario la implementación del cálculo de descriptores en Android.

Por otro lado se pueden seguir usando descriptores globales, pero automatizando el proceso de encuadrar el video. Actualmente la aplicación le pide al usuario encuadrar el video correctamente en el marco de la pantalla. Este proceso se puede automatizar realizando un procesamiento de la imagen que detecte los límites de la pantalla y que luego recorte la imagen para solo calcular el descriptor global dentro de estos límites. Al mejorar la detección de los límites de la pantalla los descriptores globales debiesen mejorar su efectividad.

6.2. Trabajo Futuro

A continuación se presentan propuestas de trabajo futuro:

- Analizar la eficacia de los descriptores usados al variar parámetros del sistema (como cuantos frames por segundo se extraen) y parámetros de cada descriptor.
- Hacer uso de índices u otras técnicas que permitan reducir el tiempo de consulta en la base de datos.
- Analizar las razones del mal desempeño de los descriptores GHD y EHD, y del comparativamente mejor desempeño de CLD estudiando sus distribuciones en la base de datos y su resistencia a las transformaciones del video provocadas por la cámara.
- Implementar nuevas estrategias de extracción de descriptores, como el uso de descriptores locales o la detección automática del marco de la pantalla e implementarlas en el dispositivo móvil.
- Estudiar otros sistemas que puedan verse beneficiados por la implementación de cálculos en dispositivos móviles.
- Estudiar en detalle el desempeño y las limitaciones de RenderScript usando algoritmos paralelos conocidos.

Bibliografía

[1] Documentación del sistema android. `http://developer.android.com/guide/index.html`. Accesado: 20-12-2015.

[2] Especificación del formato yuv 420 nv21. `http://www.fourcc.org/yuv.php#NV21`. Accesado: 20-12-2015.

[3] Y. Alp Aslandogan and Clement T. Yu. Techniques and systems for image and video retrieval, 1999.

[4] Juan Manuel Barrios. *Content-based copy detection*. PhD thesis, Universidad de Chile, Disponible en http://www.repositorio.uchile.cl/handle/2250/115521, 2013.

[5] Juan Manuel Barrios and Benjamin Bustos. P-vcd: a pivot-based approach for content-based video copy detection. In *Proc. IEEE International Conference on Multimedia and Expo (ICME'11)*, 2011.

[6] Juan Manuel Barrios, Benjamin Bustos, and Xavier Anguera. Combining features at search time: Prisma at video copy detection task. In *Proc. TRECVID 2011*, 2011.

[7] Dr. Gary Rost Bradski and Adrian Kaehler. *Learning Opencv, 1st Edition*. O'Reilly Media, Inc., first edition, 2008.

[8] Eiji Kasutani and Akio Yamada. The mpeg-7 color layout descriptor: a compact image feature description for high-speed image/video segment retrieval. In *ICIP (1)*, pages 674–677, 2001.

[9] B. S. Manjunath, J. R. Ohm, V. V. Vasudevan, and A. Yamada. Color and texture descriptors. *IEEE Trans. Cir. and Sys. for Video Technol.*, 11(6):703–715, June 2001.

[10] Dong Kwon Park, Yoon Seok Jeon, and Chee Sun Won. Efficient use of local edge histogram descriptor. In Shahram Ghandeharizadeh, Shih-Fu Chang, Stephen Fischer, Joseph A. Konstan, and Klara Nahrstedt, editors, *ACM Multimedia Workshops*, pages 51–54. ACM Press, 2000.

Anexo

A. Código

GrayZoneHist.rs

```
1
2 void setup_histogram(int32_t n_xzones, int32_t n_yzones,
3     int32_t imgWidth, int32_t imgHeigth, int32_t n_bins){
4     xzones = n_xzones;
5     bins = n_bins;
6     binMultiplier = 256 / bins;
7     zoneWidth = imgWidth / n_xzones;
8     zoneHeigth = imgHeigth / n_yzones;
9 }
10 void root(const uchar *v_in, uint32_t x, uint32_t y) {
11     int32_t bin = *v_in / binMultiplier;
12     if (bin < 5){
13         int32_t xzone = x / zoneWidth;
14         int32_t yzone = y / zoneHeigth;
15         int32_t index = (yzone * bins * xzones) + (xzone * bins) + bin;
16         volatile int32_t* addr = gOutarray + index;
17         rsAtomicInc(addr);
18     }
19 }
```

color_layout.rs

```
1 void setup_color_layout(int32_t n_width, int32_t n_height,
2     int32_t imgWidth, int32_t imgHeight){
3     height = n_height;
4     width = n_width;
5     zoneWidth = imgWidth / n_width;
6     zoneHeight = imgHeight / n_height;
7 }
8 void root(const uchar4 *v_in, uint32_t x, uint32_t y) {
9     int32_t red = (*v_in).r;
```

```
    int32_t green = (*v_in).g;
    int32_t blue = (*v_in).b;
    int32_t xzone = x / zoneWidth;
    int32_t yzone = y / zoneHeight;
    int32_t index = ((yzone * width) + xzone)*3;

    volatile int32_t* addr_r = gOutarray + index;
    volatile int32_t* addr_g = gOutarray + index + 1;
    volatile int32_t* addr_b = gOutarray + index + 2;
    rsAtomicAdd(addr_r, red);
    rsAtomicAdd(addr_g, green);
    rsAtomicAdd(addr_b, blue);
}
```

reducer.rs

```
void setup_reducer(int32_t n_width, int32_t n_height,
    int32_t imgWidth, int32_t imgHeight){
    height = n_height;
    width = n_width;
    zoneWidth = imgWidth / n_width;
    zoneHeight = imgHeight / n_height;
}
void root(const uchar *v_in, uint32_t x, uint32_t y) {
    int32_t value = *v_in;
    int32_t xzone = x / zoneWidth;
    int32_t yzone = y / zoneHeight;
    int32_t index = (yzone * width) + xzone;
    int index_value = index;
    volatile int32_t* addr = gOutarray + index;
    rsAtomicAdd(addr, value);
}
```

borderDetector

```
void setup_detector(float t){threshold = t;}
void root(uchar *v_out, uint32_t x, uint32_t y) {
    uint32_t x_index = 2 * x;
    uint32_t y_index = 2 * y;

    int A_0_0 = rsGetElementAt_int(gIn, x_index, y_index);
    int A_0_1 = rsGetElementAt_int(gIn, x_index+1, y_index);
    int A_1_0 = rsGetElementAt_int(gIn, x_index, y_index+1);
    int A_1_1 = rsGetElementAt_int(gIn, x_index+1, y_index+1);

    float max = threshold;
```

```
    uchar max_index = 5;
    float current = fabs(A_0_0*K1_0_0 + A_0_1*K1_0_1 +
        A_1_0*K1_1_0 + A_1_1*K1_1_1);
    if (current > max){
            max = current;
            max_index = 0;}
    current = fabs(A_0_0*K2_0_0 + A_0_1*K2_0_1 +
        A_1_0*K2_1_0 + A_1_1*K2_1_1);
    if (current > max){
        max = current;
        max_index = 1;}
    current = fabs(A_0_0*K3_0_0 + A_0_1*K3_0_1 +
        A_1_0*K3_1_0 + A_1_1*K3_1_1);
    if (current > max){
            max = current;
            max_index = 2;}
    current = fabs(A_0_0*K4_0_0 + A_0_1*K4_0_1 +
        A_1_0*K4_1_0 + A_1_1*K4_1_1);
    if (current > max){
            max = current;
            max_index = 3;}
    current = fabs(A_0_0*K5_0_0 + A_0_1*K5_0_1 +
        A_1_0*K5_1_0 + A_1_1*K5_1_1);
    if (current > max){
            max = current;
            max_index = 4;}
    *v_out = max_index;
}
```

edgeHist.rs

```
void setup_edge_histogram(int32_t n_xzones, int32_t n_yzones,
    int32_t imgWidth, int32_t imgHeigth){
    xzones = n_xzones;
    yzones = n_yzones;
    zoneWidth = imgWidth / n_xzones;
    zoneHeigth = imgHeigth / n_yzones;
}
void root(const uchar *v_in, uint32_t x, uint32_t y) {
    int32_t bin = *v_in;
    int32_t xzone = x / zoneWidth;
    int32_t yzone = y / zoneHeigth;
    int32_t index = (yzone*bins + xzone)*bins + bin;
    volatile int32_t* addr = gOutarray + index;
    rsAtomicInc(addr);
}
```