Android-based Crowd Sourcing of Landmark Metadata

github.com/alexdamian/AndroAR

Alexandru Marian Damian

16 June 2012

Abstract

Augmented Reality has gained much interest in recent years, mostly backed by the smartphone surge, and many applications focusing on it have been created. These applications rely on sensor-generated data from inputs such as sound, video or GPS positioning. Most mobile applications rely on just a small subset of inputs (*i.e. only GPS positioning data, or only the camera's video feed*).

We wanted to create an application that will help people get around unknown environments. Typical use-cases include freshmen getting around their University campuses, or tourists travelling to new cities.

At the same time, we wanted to experiment with a combination of *image recognition algorithms* and *GPS positioning data*, in order to achieve a high-quality user experience.

We designed and implemented an augmented reality application for Android-based smartphones which allows users to receive information about the landmarks (*i.e. buildings, statues, etc.*) in their line of sight. Information is attached to bounding boxes drawn around the landmarks. Detection occurs using a combination of GPS positioning and object detection, based on the training examples we have stored in our database. These examples are generated by crowdsourcing annotated landmark data from users.

Keywords: *augmented reality, image recognition, pattern recognition, object detection, live, crowdsourcing, Java, Android, Cassandra, OpenCV, database, feeds, AndroAR*

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Proposed Application – AndroAR . . . 2
- 1.3 Structure of Thesis . . . 2

2 State of the Art **3**
- 2.1 Layar . . . 3
- 2.2 Google Maps / Street View . . . 3
- 2.3 Wikitude . . . 4

3 Goals for AndroAR **6**

4 System Design **8**

5 Information Flow Through the System **10**
- 5.1 Queries . . . 10
- 5.2 Store Requests . . . 12

6 Performance Issues **15**
- 6.1 Query Issues . . . 15
  - 6.1.1 Sparse Queries . . . 15
  - 6.1.2 Caching . . . 16
- 6.2 Storage issues . . . 16
  - 6.2.1 Passive example generation . . . 16
- 6.3 Write estimations . . . 16

7 Implementation **17**
- 7.1 Overview . . . 17
- 7.2 Communication . . . 18
- 7.3 Server . . . 19
  - 7.3.1 Caching . . . 20
- 7.4 Storage . . . 20
  - 7.4.1 CAP Theorem . . . 20
  - 7.4.2 Database options . . . 21
  - 7.4.3 MongoDB vs. Apache Cassandra . . . 22
  - 7.4.4 Database Choice . . . 23
  - 7.4.5 Database Schema . . . 23
- 7.5 Image Recognition Component . . . 25
  - 7.5.1 Why a Separate Server . . . 25
  - 7.5.2 The OpenCV Library . . . 25

7.5.3 Feature detection . . . 25
7.5.4 Feature Matching . . . 26
7.5.5 Optimization . . . 27
7.6 Testing GUI . . . 30
7.7 Mobile Application . . . 31
7.7.1 Android Platform . . . 31
7.8 Tools . . . 33

8 Results and Conclusions **34**
8.1 Installation . . . 34
8.2 Results . . . 35
8.3 Problems . . . 35
8.4 Conclusions . . . 36

9 Future Work **38**

A Directory Structure **43**

Chapter 1

Introduction

1.1 Motivation

Ambient intelligence and, in particular, augmented reality are becoming increasingly popular, mainly due to the surge of smartphones and the increasing Internet connectivity options users have. Researchers and companies are offering applications that are more **interactive**, **easier to use** and better **integrated** with what the user sees and feels.

In February 2001, ISTAG (*IST Advisory Group*) proposed 4 scenarios for how ambient intelligence applications will interact with users in 2010[1]. While those scenarios are far from being implemented today, much research has been conducted in this field and numerous applications have been developed. Two of the most recent and most interesting examples are:

1. **Google Glasses**[1]. This project is currently in development and plans to offer augmented reality directly in the user's line of sight, on a transparent glass cube, without them needing a smartphone or any kind of screen. The latest developments and demos have been presented at the Google IO 2012 conference (June 27 - 29). Planned features are:
   - phone and video calls;
   - video capture, along with sharing;
   - navigation;
   - notifications (messages, emails, calendar events);
2. **BMW head-up display**. BMW offers the option of augmenting information on the windshields of their high-end cars, in order for the driver to be more focused on the road ahead. Augmented information includes:
   - current speed, gear, etc.
   - turn-by-turn navigation;
   - speed limitations, radars;
   - car notifications.

However, most applications currently tend to pivot around *just* one of the two most common ideas:

1. augmenting different layers of information (*i.e. social-media information – friends' check-ins, tweets, Facebook posts, Google+ posts –, tourist information – hotels, restaurants,*

[1] https://plus.google.com/111626127367496192147

parks, museums –, even historical events – wikipedia articles –) mainly based on GPS positioning;

2. augmenting information based on a snapshot of what the user sees (*i.e. a frame in the phone's camera feed*).

The use-cases we must to deal with will force us to implement a mixture of the two ideas stated above.

1.2 Proposed Application – AndroAR

We propose a mobile application called AndroAR that will augment landmark information on the current camera feed. Queries containing the camera frame, GPS position, orientation and other sensor data will be sent to a server. It will retrieve, from a database, all the possible landmarks that can appear in the received camera frame, and will forward their instances (*i.e. images*) to an image recognition component, which will decide if they are present or not in the frame. Metadata for these objects will be returned to the user's mobile application, in order to be augmented on the screen.

Our application will support and will encourage crowdsourced contributions to annotate landmarks. We will offer a way for the user to freeze the camera feed on a particular frame, crop a landmark and annotate it with metadata. We will also be using different techniques to passively generate additional correct instances of objects that have already been annotated.

An example use-case of our application is: a tourist is travelling to a foreign country and uses AndroAR to find his way around an unknown city. Information about landmarks *that he sees* will be augmented in the application, on top of the camera feed.

We would like to thank Andrei Petre[2] for developing the Android application.

1.3 Structure of Thesis

The structure of this thesis is as follows:

- chapter 1 presents the motivation for choosing this subject;
- chapter 2 offers information regarding the most widely used and most innovative augmented reality applications;
- chapters 3, 4 and 5 present the goals for our application, the design and the workflow of our system;
- chapter 6 presents the problems that arise given the system design and the goals of our application;
- chapter 7 presents details regarding the implementation of this system;
- chapters 8 and 9 present the results, conclusions and future work.

[2]student at the Polytechnic University of Bucharest, Faculty of Automatic Control and Computers, second year

Chapter 2

State of the Art

Augmented reality is very popular in the mobile industry. However, most applications interact very little with the video feed they overlay the information on. Typically, applications will:

- interact with a small part of the current view, such as a *logo* or *banner*;
- take advantage of the localization capabilities of the phone, such as *GPS positioning* or *compass*, but place information on the screen with disregard to what the user's line of sight actually is (*e.g. a user might be facing a wall and be overrun with information, even though he doesn't see anything relevant*).

We will present the most popular and most innovative products that currently exist on the market, also stating what their disadvantages are *relative to* the implementation of our application's use-cases.

2.1 Layar

Layar[1] is a mobile application built for Android and iOS. It is the most well-known augmented reality mobile application in the industry. Having pivoted a few times, it now focuses on bringing the augmented reality experience to printed publications.

However, Layar's current focus makes it unsuitable. One can try to associate a current view of the world (*i.e. what the user sees at a particular moment through the smartphone's camera*) with a *printed magazine page*. But the world is dynamic and tridimensional, and, therefore, problems will arise when trying to extend the Layar behavior.

We must note that Layar has limited support for displaying 3D objects.

2.2 Google Maps / Street View

Google Maps[2] enables users to navigate and receive directions on an extremely detailed map. Street View, a subproduct of Google Maps, offers navigation at street level, allowing the user to asociate information with their surroundings.

[1] http://www.layar.com/
[2] http://maps.google.com/

Figure 2.1: Screen caption of the Street View web application

However,

- Street View is not live. Street View imagery was captured using dedicated equipment. Efforts are being made to keep the imagery as up-to-date as possible, but this can not be done all around the world. For example, several areas in Romania contain images that are 4 years old.
- only recently has Street View begun offering imagery outside of street range (*i.e. pedestrian roads, parks, campuses, indoor*).
- users can navigate through Street View using their smartphone, but the Street View imagery will completely replace what the user sees.

2.3 Wikitude

Wikitude[3] is one of the first implementations of an augmented reality environment. Wikitude is more feature-rich than Layar, offering more than 3500 so-called *worlds* (which are, basically, layers of information that will be displayed on top of the current camera feed, *i.e. tweets, nearby hotels, events, etc.*). One of the most interesting aspects of it is that it offers a SDK for development of new features.

Figure 2.2: Screen caption of the Wikitude Drive application

[3]http://www.wikitude.com/

However, even though there are many sources of information that can be presented to the user, *Wikitude* still doesn't take advantage of what the user actually sees, but only their position and orientation.

We must note that an application called **Wikitude Drive** exists, offering *turn-by-turn navigation*, augmented over the current view of the road.

Chapter 3

Goals for AndroAR

AndroAR is designed to offer a *true* augmented reality experience, by interleaving the *localization features* with *image recognition* algorithms. While designing AndroAR, we focused on several requirements, which we will present below, along with reasons why the current solutions can not be used or are difficult to extend.

Live

We need to focus on displaying information relevant to what the user sees, not just to the user's position and orientation. Most augmented reality applications will place an overlay of information on top of the current camera feed; the two are related only by the current localization features of the user (*i.e. GPS position, compass orientation*) and not by what the camera feed actually shows.

A typical example in which such applications perform poorly is when two buildings are collinear and viewed from an angle such that the second one is fully covered by the first one. Most applications will offer information related to both buildings, even if the user can only see one. Although this can be solved by testing for collinearity, there is no solution when the covering object is not annotated, or if it is temporary located there (*i.e. mobile advertising panel*).

Layar offers a live view, but is only focused on paper.

Street View doesn't offer a live view, but Google Maps supports uploading of user images and tagging; these images can be augmented onto the Street View imagery, but that only works on the desktop web version.

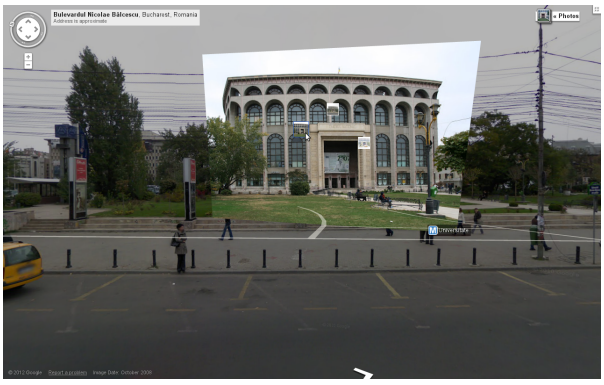


Figure 3.1: Example of augmentation of a user-submitted photo in Street View

Wikitude is live and shows information originating in an area around the user. We can use their SDK to retrieve the subset of objects around the user's current position and then use our own algorithms to select which objects' information to display. But even so, we must deal with aquiring the data (either by user input, or by crawling different sources that have the mapping *object information* → *object image* available) and storing it in a Wikitude-friendly way.

Latency

Using the camera feed brings up some issues regarding speed and bandwidth since displaying information is not a matter of only querying a database with the user's current position anymore. We need to find a way to divide the computational effort between the server and the mobile application. We must also consider the effect of latency on the query replies (*i.e. by the time the user receives the query reply, the detected objects might have translated).*

By its design, **Layar** users are expected to wait for the reply to the query. A typical usecase of Layar is: *a user will see the message* Scan for a making-of video *on the cover of a magazine and will wait for the application to serve the video.*

Since it's not live, **Street View** can precache the information it needs in order to serve information to the user. By its design, Street View queries will not be affected by what the users see, but only by their position.

Wikitude will definitely have a speed advantage over our application, since they involve only querying a database and not executing any other CPU-intensive operations (*i.e. image recognition*).

Correct

We must make sure that the information (*i.e. objects associated with images and metadata*) is correct. Since we are using a crowdsourced approach, we should consider rating of images to allow removal of incorrect associations. Also, we must deal with spam.

Layar has fewer problems with this since they are using a business-to-consumer model, in which the information is provided by the business and the queries are made by the consumer (*i.e. a magazine creates the layers and the reader uses the application*). Layar can correctly assume that businesses will provide high-quality information.

Street View already has correct and high-quality imagery. Images added by users can be easily filtered by first querying them against the existent database.

Information available in **Wikitude** has a very high probability of being correct, since it is backed by a community, but we are interested in *object information* → *object image* associations, and Wikitude contains *object information* → *object localization* associations.

Chapter 4

System Design

The basic design of the system has 5 components:

1. server,
2. storage,
3. image recognition component,
4. mobile application,
5. communication layer,

connected as follows:

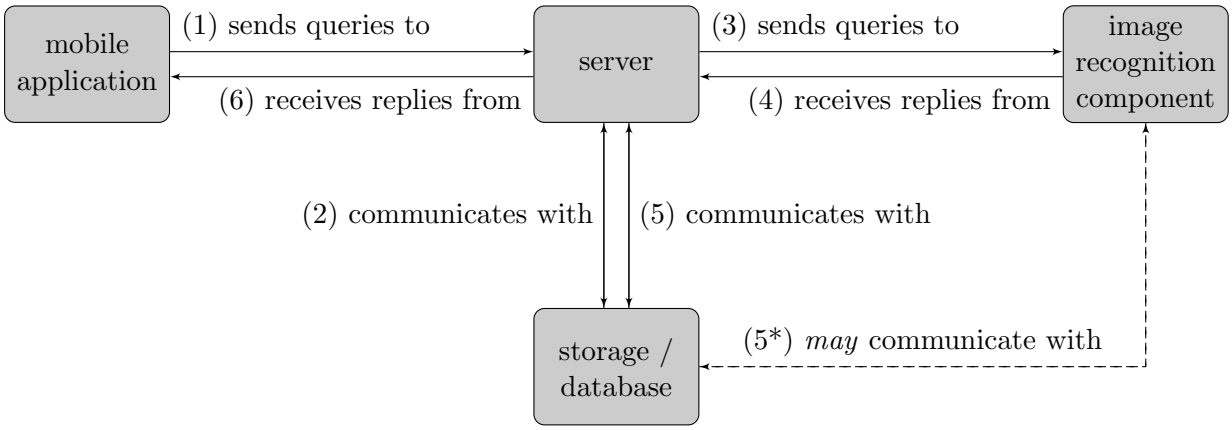


Figure 4.1: Basic system design

Note that, although *arrows (2)* and *(5)* both share the same caption, the transferred messages they refer to are different:

- *arrow (2)* refers to messages related to queries that are received by the *server* from the *mobile application*. These messages will aid the server in preparing the more complex query that will be sent to the *image recognition component*;
- *arrow (5)* refers to messages related to queries and replies that are received by the *server* after being parsed by the *image recognition component*.

The system design can be easily scaled, by implementing one of the following:

1. *Servers*, *image recognition components* and *database instances* are selected from different pools.

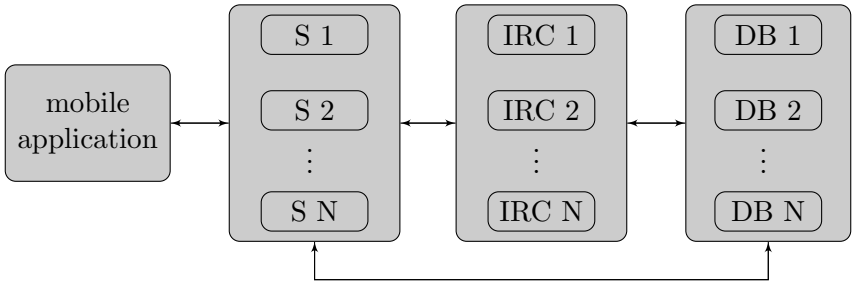


Figure 4.2: Design option for scaling the system, using *3* pools for *servers*, *image recognition components* and *databases*

.

2. Each *server* is associated with its own *image recognition component*. The *server + image recognition component* and the *database* are chosen from their respective pools. Each time an *image recognition component* needs to query the database, it will forward the request to the *server*, which will act as a *transit component*, therefore reducing the implementation time, while still keeping high performance.

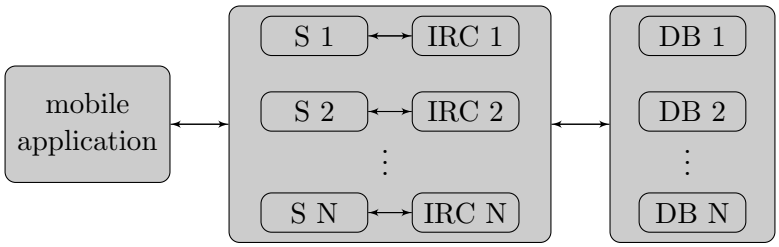


Figure 4.3: Design option for scaling the system, with each *server* being connected to the corresponding *image recognition component*, therefore using only *2* pools.

When scaling the system, we must take into account several aspects:

1. the state of the system is completely stored in the *database*. The *server* and the *image recognition component* are stateless and can therefore be easily scaled, by using more instances;

2. the *database* is stateful and is, therefore, more difficult to scale. Also, we must take into account the high number of writes our database must accomodate.

   For example, *Netflix* has chosen **Apache Cassandra** as the database for their system. Experiments conducted by them showed excellent performance under high amounts of write requests, achieving 1.1 million writes per second (or 3.3 million writes per second after replication with a factor of 3) when using a cluster of 96 medium sized Amazon EC2 instances (or 288, after replication)[1].

[1] http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html

Chapter 5

Information Flow Through the System

A user can interact with our application in two ways:

1. by making queries while using the application;
2. by sending images annotated with objects and metadata.

We will illustrate the overall workflow for each of them below.

5.1 Queries

Let us assume that at one point in time, while the user is using the application, it needs to find all the objects that appear in a frame, along with their metadata. To accomplish this, we will send a request to the server that will be completed in the following steps:

1. The mobile application will encapsulate the current frame, along with localization information (GPS position and compass orientation) and send it to the server;
2. The server will forward the query to the database which will fetch from storage, all the possible objects in the user's line of sight. We define the user's *line of sight* as a *cone* in front of the user, as shown below. All the objects that are in the user's line of sight are marked in *green*. The objects that we consider as being not observable are marked in *red*:

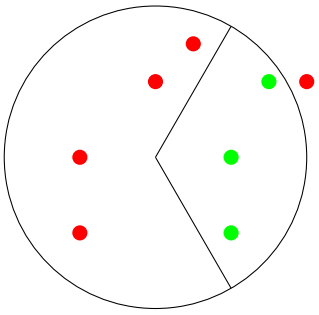


Figure 5.1: Example of using the line of sight to retrieve a subset of possible matches

3. The database will return the objects that might pe present in the frame. For each object, information that will help match the particular object to an image will also be returned.

Note that each object may (and should) be present in more than one image. We will return matching information for the best performing subset of these images, which may contain:

(a) image features (*e.g. lines, corners, areas of constrast, etc.*);

(b) illumination information;

(c) etc.

4. We will forward the initial query, along with the possible objects to the image recognition component. This component will:

   (a) compute the features for the query image;

   (b) compare the features for the query image to the features of every possible object and assign a rate of success (or certainty) to each statement
   X is present in the query image I, where X is a *possible object*

5. Using a threshold for the rate of success, we will end up with a subset of the possible objects being classified as detected objects. These detected objects, along with the corresponding positions in the query image, will be sent back to the server by the image recognition component.

6. The server will then forward the reply to the mobile application.

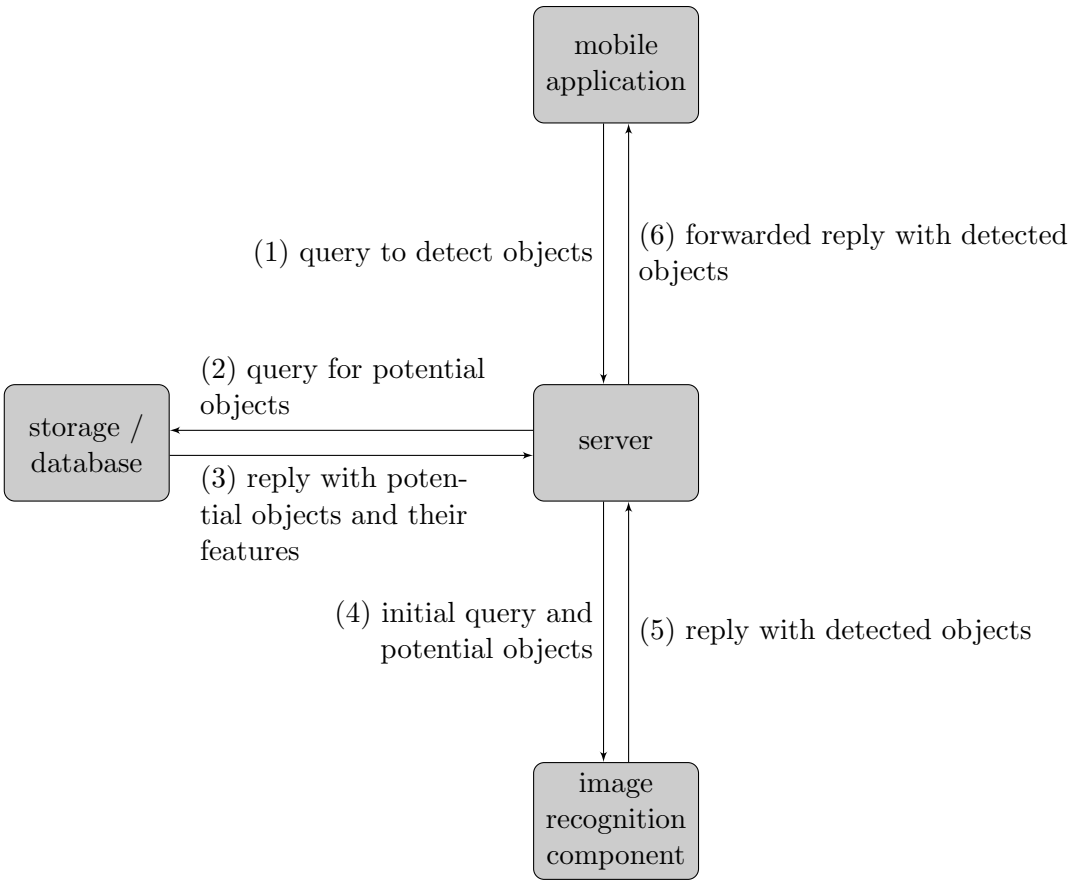


Figure 5.2: Life of a query.

Example

We will illustrate the previous workflow with an example:

- Let us assume that a tourist is travelling through Paris and, when a query is made to the server, he is looking at the *Eiffel Tower*.
- The server will receive a query containing an image of the *Eiffel Tower*, along with the tourist's position and orientation.
- The server will then request, from the database, all the possible objects that might be present in the tourist's line of sight. For example, the server might request the top (at most) 10 ranking instances of all the objects in a 100 meter radius and in a 90° cone in front of the user.
- A possible reply from the server contains the features for 10 images of the *Eiffel Tower* and the features for 6 images of *Les Invalides*.
- These will be forwarded to the image recognition component which will try to match 2 possible objects with the query image. The confidence rates that result can be:

Object	**Confidence**
Eiffel Tower	.85
Les Invalides	.5

- With a threshold of .75, only the *Eiffel Tower* will be selected as a valid match and will be returned as the query reply.

5.2 Store Requests

Our application supports and encourages crowdsourced contributions to annotate new landmarks. Should a user want to annotate a landmark, the application will allow them to freeze on a particular camera frame, create bounding boxes around landmarks and annotate them with information. We will need to store this information into our database.

To accomplish this, we will send a request to the server that will be completed in the following steps:

1. The mobile application will encapsulate the current frame, along with localization information (GPS position and compass orientation), and the bounding boxes and metadata for each input object; it will then send this request to the server.
2. The server will forward the request to the database, which will store the data (*i.e. image, localization, objects ids, objects metadata, cropped objects images, etc.*).
3. The server will then forward the request to the image recognition component which will compute the features for:
   - the large image;
   - the small images (*i.e. images cropped according to the provided bounding boxes for the detected objects*).
4. The image recognition component will then send the features back to the server.

5. The server will forward them to the database, which will store them.

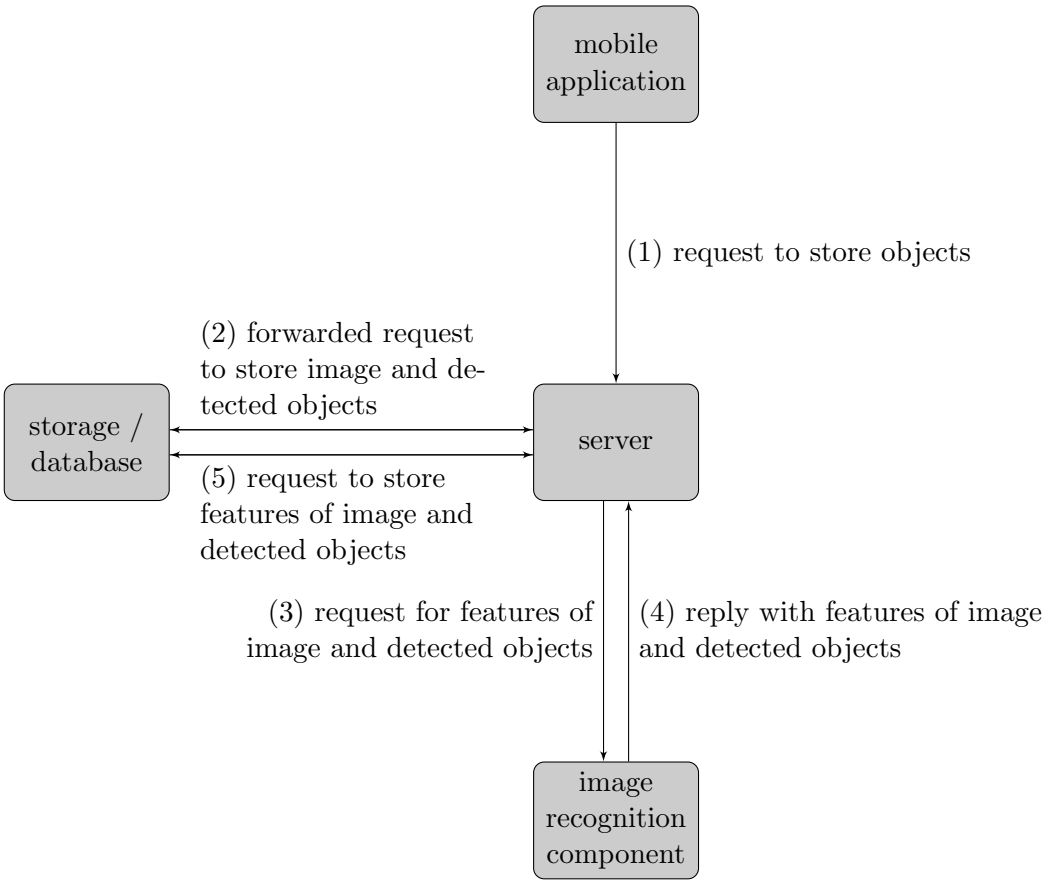


Figure 5.3: Life of a storage request.

Example

We will illustrate the previous workflow with an example:

- Let us assume that the same tourist has now arrived home, in Romania. He now wants to use the application in Bucharest, but sees that very few buildings are tagged. Even when he is pointing his smartphone at the *National Theatre* building, he sees that no information is being displayed. He decides to annotate this building with metadata.
- He clicks on the *Capture* button and the camera feed freezes on the current frame. He must then select the building by positioning and resizing a bounding box over it and insert the landmark's name and description.
- The server will receive a store request containing the image of the *National Theatre* and its surroundings, a cropped image of the *National Theatre*, metadata (name, description) and the tourist's position and orientation.
- The server will then forward the request to the database, which will store all the information.

- Additionally, the server will request all features to be computed for both the original image and all the smaller, cropped images. The server will then forward this information to the database to be stored accordingly.

This example only illustrates basic functionality of the *Storage* component. A more likely workflow is:

- compute all the frames in a predefined range before and after the current frame that contain the tagged objects;
- detect the object's translation in order to automatically adjust the bounding box in each frame;
- send each frame, along with its cropped images, metadata and localization information to the server, when the mobile phone connects to a WiFi link.

We will explain ideas for capturing additional valid examples (*i.e. correct object instances*) in chapter 6.

Chapter 6

Performance Issues

The workflows presented in chapter 5 pose several performance issues.

6.1 Query Issues

Ideally, we would want to answer queries at a rate equal to the *fps* value of the current video feed. A sufficient value for the *fps* would be $5-10$. We do not need the performance of a higher *fps* value (*i.e. the cinematic value of 24*).

However, even intuitively, it will be extremely difficult to answer queries at a rate of $5-10$ *fps*, for *each* connected client. Moreover, we must consider the limited amount of traffic a mobile phone has available on 3G / 4G networks. With this in mind, we should limit the amount of queries issued by the client.

6.1.1 Sparse Queries

We can achieve a good trade-off between quality and performance by using a simple observation: *in most scenarios, a detected building will not disappear from the user's line of sight in less than* $1-2$ *seconds*; that is because the user will not pan the camera fast enough so that the entire landscape changes.

We can take advantage of this by:

1. issuing sparse queries (*i.e. a query every 5 seconds*);
2. in between these queries, using the orientation (mainly) and the GPS position of the device to translate or scale the bounding boxes accordingly;

We should note that:

1. there will be a delay between the moment when the query is issued and the moment when the response is received; we should translate and scale the bounding boxes according to the relative move of the device;
2. translation and scaling according to localization capabilities will allow for higher quality user experience; let us imagine that the user is looking at a building for which the right half is very rich in features and the left side is very poor in features, and that the building is correctly detected. Should the user pan the camera to the left and have only the left half of the building visible, in the 5 second range, a bounding box will still be shown around the building, even if the image recognition component can not further detect it.

We will also ilustrate this in chapter 7, but it important to note why we chose the 5 second range. Using a simple implementation with the following features:

- *SURF* detector[1];
- *Brute-force* matcher;
- 2 potential objects (totalling 4 instances),

the average round-trip time for a query was 3.5 seconds, using a *high-quality, color, 8 megapixel* image.

6.1.2 Caching

Given that the users will change their GPS position so sightly every-time a query is issued (*i.e. a typical user will travel by foot; assuming their average speed is* $4km/h$*, they will travel* 1 *meter every second*), we can take full advantage of *caching*. We could query the database for *all* the objects surrounding the users, not just those in their cone of sight and cache their features. Should the user move, we only need to query the database for the objects in the areas previously not covered.

6.2 Storage issues

As stated above, we are using a crowdsourced approach to getting new data. However, this is often not enough, since most of the users will submit very little information.

6.2.1 Passive example generation

We have come up with some ideas to increase the number of correct examples (*i.e. object instances*) that are saved to our database:

1. when a user freezes on a frame and tags several objects, we can safely assume that some frames before and some frames after will also contain that object. We can therefore send them in a batch to the server to store in the database. The server should decide which of them it should keep (*i.e. frames that actually contain the object and have good quality, and are not repetitive*);
2. when a mobile application instance issues a query and a reply that contains objects rated with high confidence is returned, than we can also assume that some frames before and some frames after will also contain those objects. We can then proceed as above.

6.3 Write estimations

We must also take into account the amount of writing that needs to be supported by the database: assuming *50 concurrent clients*, all with a framerate of *10 fps*, each frame being sent to the servers as a 800×600 *JPEG image* of size roughly *300 kilobytes*, we require a write speed of *150 megabytes per second*, a speed that is difficult to achieve using traditional *SQL* databases.

[1]Speeded Up Robust Feature detector

Chapter 7

Implementation

7.1 Overview

In chapter 4 we presented the *high-level system design*. We will illustrate our choice of technologies for all of the system components, followed by an in-depth explanation of the reasons and implementation details.

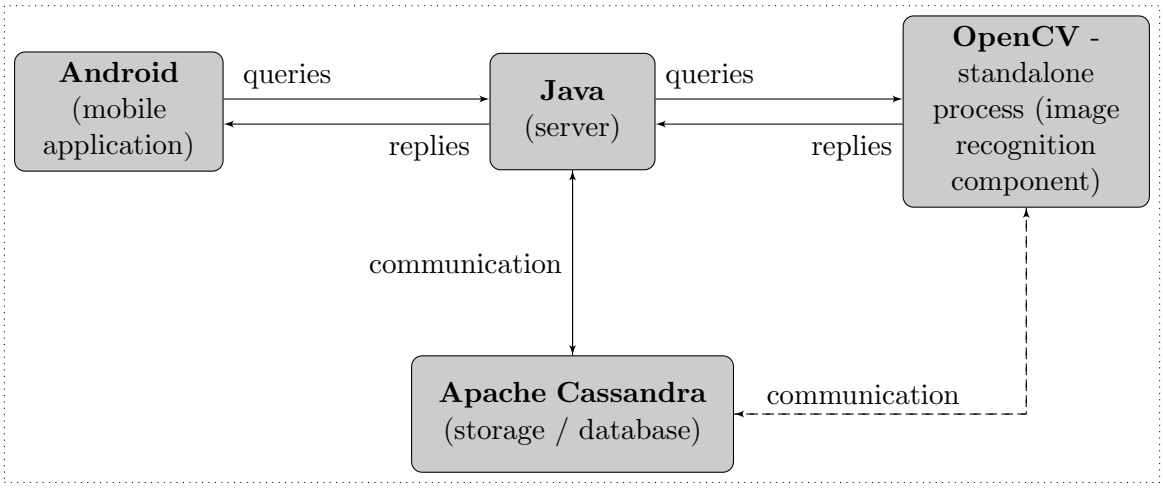


Figure 7.1: Technologies used in the system implementation

The main reasons for choosing these technologies are:

- **Java**™ (server) → advanced network I/O and multithreading;
- **Android**[1] (mobile application) → open-source, well documented, written in Java™;
- **OpenCV**[2] (image recognition component) → complex, efficient, versatile;
- **Apache Cassandra**[3] (storage) → excellent distributed performance, fast writes, caching;

[1] http://developer.android.com/
[2] http://opencv.willowgarage.com/
[3] http://cassandra.apache.org/

- **Protocol Buffers**[4] (communications layer) → high compression, works well with binary data.

7.2 Communication

For the *Communication Layer*, we decided to use *Google Protocol Buffers*. *Protocol Buffers* are an extensible and efficient way to encode structured data, used by Google for most of their internal RPC protocols. This library has several advantages over the more widely used *XML* or *JSON*:

- it is less verbose; setting values to keys is similar to *XML* or *JSON*, but the keys are not encapsulated in the final message;
- it is typed; this allows for data to be optimally encapsulated (*e.g. 32-bit integer types will only take 4 bytes to encode*);
- it works well with binary data (we will be using it to transfer images); on the other hand, *XML* and *JSON* work better when trying to encode markup text;
- it has built-in validation.

Also, given our current setup, we can easily avoid their disadvantages:

- *Protocol buffers* offer support only for C++, Java™ and Python;
- a human-readable form can be obtained by parsing the message (*e.g. in Java™, calling the* toString() *method, or in C++, calling the* DebugString() *method*).

Below is an extract from an example message used by our application:

```
// Next id = 5
message DetectedObject {
        enum DetectedObjectType {
                UNKNOWN = 1;
                BUILDING = 2;
        }
        optional DetectedObjectType object_type = 1;
        optional ObjectMetadata metadata = 2;
        required string id = 3;
        optional bytes cropped_image = 4;
}
```

The definitions for the *Communication layer* messages are stored in the **/proto** directory. They are compiled using the **protoc** compiler. Below is an extract from the *Protocol Buffers Makefile*:

```
java:
        protoc --java_out=. comm.proto image_features.proto
        cp -r com/ ../AndroARComm/src/
```

Issuing the command *make java* will result in the creation of *Java classes*, stored in corresponding *.java* files. All the objects representing messages will be *immutable* and will be created with the help of a dedicated *Builder*. Below are some examples of methods created for the *DetectedObject* class:

[4] https://developers.google.com/protocol-buffers/

```
// builder
public DetectedObject.Builder newBuilder();
// getters
public boolean hasObjectType();
public boolean hasCroppedImage();
public DetectedObjectType getObjectType();
public String getId();
public ByteString getCroppedImage();
// serialization and communication
public byte[] toByteArray();
public static DetectedObject parseFrom(byte[]);
public void writeTo(OutputStream output);
public static DetectedObject parseFrom(InputStream input);
```

The *DetectedObject.Builder* class has all the above methods, plus their corresponding *setters* and a few more *serialization and communication* methods:

```
// setters
public DetectedObject.Builder setObjectType(DetectedObjectType value);
public DetectedObject.Builder setId(String id);
public DetectedObject.Builder setCroppedImage(ByteString value);
// serialization and communication
public DetectedObject.Builder mergeFrom(Message other);
```

Creating the *simplest*, but still *valid*, DetectedObject message can be done as follows:

```
DetectedObject detected_object =
DetectedObject.newBuilder().setId("hello world").build();
```

7.3 Server

The server must be able to deal with the following aspects:

1. be easy to distribute;
2. allow multiple, simultaneous remote connections;
3. easily implement an interface with the database;
4. implement or interface with the image recognition component;

The first three aspects are easier to implement using Java™, by making use of its more advanced *network I/O* and *multithreading* capabilities.

We decided to decouple the fourth aspect in a separate component, implemented in C++. We had the following aspects in mind:

- optimization of CPU-bound components;
- most image recognition libraries and frameworks offer C/C++ support.

The server's main attributions are to *ensure communication* and to *correctly modify and forward messages* between all components.

The entire *communication layer* is implemented by the server uses **TCP**. We are using reliable connections to be able to parse the messages we receive (binary encoded) and to ensure consistency of data throughout the system.

When a new client attempts to connect to one of the server instances, a new object is chosen from a *threadpool* of **Client Connections**. Also, a new object is chosen from a *pool*

of **Database Connections**. Instead of the *Client Connection object* creating its own *database connection*, we have opted to use a pool of *database connection* clients to allow for more instances to be used with each server, mainly to reduce the overhead of creating connections when issuing *small and short-lived queries*.

For our current prototype, we have chosen to implement a *many-to-one* communications model between the *server* (more specifically the *client connections*) and the *image recognition component*. This has to do mainly with threading and synchronization being tedious to implement in C++. However, communication in the Java™ server-side has been implemented with consideration to a future transition to a *many-to-many* communications model. We are currently using a *thread-safe queue* in order to serialize requests sent to the *image recognition component*.

7.3.1 Caching

We implemented a caching system to take advantage of the fact that database queries for objects in a user's line of sight change very little over time, due to slow movement speed.

However, in order to use the caching system to its full potential, we must ignore the user's orientation when querying the database. Our implementation for the caching system queries the database for all the objects surrounding the user. It then stores them in the cache, but only serves the objects that are located in the user's line of sight (*Figure* 7.2, *left*). When another query is issued from the mobile application, the caching system only needs to query the database for objects in the unknown area (*Figure* 7.2, *right*).

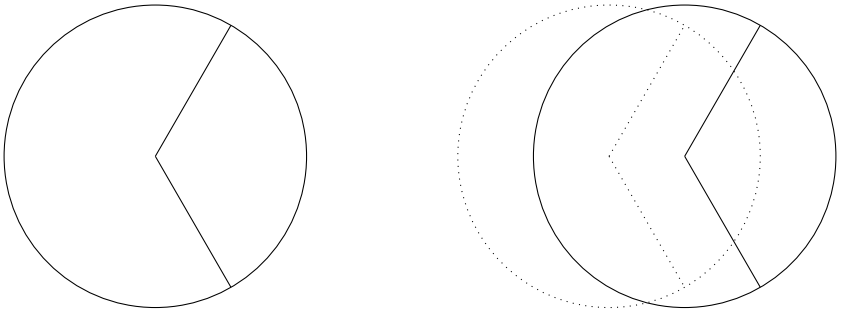


Figure 7.2: Example of how the caching system works

7.4 Storage

7.4.1 CAP Theorem

Theorem 1 *It is impossible for a distributed computer system to provide all of the following three guarantees:*

- *Consistency*
- *Availability*
- *Partition-tolerance*

This theorem was proposed as a conjecture by Eric Brewer at the 2000 Symposium on Principles of Distributed Computing (PODC). The formal proof was published in 2002[4].

7.4.2 Database options

There are two directions of design and implementation for databases:

1. **ACID**, which stands for *atomicity, consistency, isolation and durability.* Databases that implement *ACID* include *MySQL, SQLite*;

2. **BASE**, which stands for *basically available, soft state, eventually consistent.* Databases that implement *BASE* include *Apache Cassandra, MongoDB.*

There is also another classification of database types, based on the model they adhere to. The two most common classes of databases are:

1. **RDBMS** (stands for *relational database management system*). Databases adhering to this model will have ACID properties and will typically use SQL (structured query language);

2. **NoSQL**. NoSQL databases do not adhere to the RDBMS model. They typically do not use SQL as their query language and don't offer ACID properties. Most of these databases focus on the *availability* and *partition-tolerance* aspects stated in the CAP theorem, providing *eventual consistency* within a *distributed, fault-tolerant architecture.*

Most database implementations fall into only one of the two classes. However, we must note that some implementations are *hybrid*, such as the latest version of **PostgreSQL**.

When choosing the database model, we must keep in mind several aspects:

1. **What the queries that we must reply to are**. The possible database queries are:

   (a) *Return all the objects in the line of sight given a GPS position and an orientation.*

   (b) *Return images and features for an object, optionally sorted by rank.*

   (c) *Return metadata for an object (or a list of objects).*

   (d) *Store an image and the detected objects that appear in it.*

2. **What is the rate of occurrence of these queries**. Most of the queries that will be sent to the databases are

   (a) *Return all the objects in the line of sight given a GPS position and an orientation.*

   (b) *Return (the best) images and features for an object.*

   (c) *Return metadata for an object (or a list of objects).*

   with the most common being a combination between the first 2: *Return (the best) images and features — for all the objects in the line of sight given a GPS position and an orientation.*

3. **Consistency**. Before we decide whether we always need consistency or we can manage with eventual consistency, we must observe how the database can change over time.

   The database **replies** might change if one of the following events occur:

   (a) a store request appears and a new image is stored in the database;

   (b) after a query, the ranking for the instances of objects changed (*i.e. an instance of an object becomes more or less successful*).

   Given these possible situations, we can manage with *eventual consistency*, because:

- a new instance of an object will not greatly affect the performance of the system, since determining whether an object appears in an image or not is done using more than one instances;
- ordering of object instances based on their success will not be drastically changed, since we can safely assume that the *best* instances tend to remain at the top and the *worst* instances tend to remain at the bottom.

4. **Latency**. The most common type of query needs to be answered as fast as possible, in order to reduce round-trip time. A distributed database is preferrable because we can easily ensure load balance.

Due to all the aspects stated above, we decided to use a *NoSQL* database. There are many *NoSQL* database implementations, some of which being presented below, along with their features:

MongoDB[5]	best used for dynamic queries over data that changes little over time
Cassandra[6]	BigTable-like, highly-distributed, fast-write database
HBase[7]	best used in conjunction with Hadoop and HDFS
Redis[8]	best used for data that changes rapidly (*i.e. communication*), but is limited in size
Riak[9]	the *light* version of *Cassandra*, with a trade-off in scalability
VoltDB[10]	a scalable, consistent and highly available multi-master database

Given the use-case and restrictions of our application:

- Java™ API,
- scalability,
- fast queries,

the top selections were *MongoDB* and *Apache Cassandra*. We will compare the two in the following section.

7.4.3 MongoDB vs. Apache Cassandra

API; Supported Languages

MongoDB is the best supported database from all those presented above. It has support for most of the programming languages and offers an API that retains some features from SQL. **Cassandra** offers a Java™ API using *Thrift*, a custom binary protocol. Database operations are rather tedious, but wrappers exist that easen up most of the operations (*i.e. Hector*[11]).

[5] http://www.mongodb.org/
[6] http://cassandra.apache.org/
[7] http://hbase.apache.org/
[8] http://redis.io/
[9] http://wiki.basho.com/
[10] http://voltdb.com/
[11] http://hector-client.github.com/hector/

Scalability

MongoDB is easy to set-up if only a single server is used.
Cassandra, however, has the advantage of being more reliable when using more server instances. It has a no-single-point-of-failure architecture and more options for better management of data (*i.e. replication factor, replication strategy, read/write quorum*). Also, Cassandra is easier to set-up in a multi-server environment, since all nodes are homogenous.

Fast Queries

MongoDB has more advanced support and flexibility for indexes. However, the main disadvantage of MongoDB is that it has a **global write-lock**, making large amounts of writes more problematic.
Cassandra's architecture performs extremely well when many *writes* occur, offering constant-time writes, no matter how large the current database is.

7.4.4 Database Choice

After comparing the top two choices for the database, we decided to use **Apache Cassandra** [5] as the database for our application.

7.4.5 Database Schema

Given that we are using a *NoSQL* database, before we explain the *database schema*, we need to offer definitions to the terms we will use:

column	→	a tuple of *key*, *value* and *timestamp*; imagine this as a cell inside a SQL table;
supercolumn	→	a group of **columns** with the same *key*; imagine this as a tuple of *key*, *value* and *timestamp*, where the *value* field is a collection of columns (*i.e. a map*);
column family	→	a collection of **columns**; imagine this as a SQL table;
keyspace	→	a collection of **column families** and **supercolumn families**; imagine this as a SQL database.

We can now illustrate the *database schema*. Our project's *keyspace* will contain :

- *1 column family* to store images and their features;
- *1 supercolumn family* to store objects associated to images.

Image features	
key	row key. This is the image's hash (string)
image-contents	the binary representation of the image (bytes)
localization	the binary representation of the localization features *Protocol Buffers* message, containing GPS position and compass orientation (bytes)
gps-latitude	the GPS latitude where the image was taken (float) (**index**)
gps-longitude	the GPS longitude where the image was taken (float) (**index**)
num-objects	number of detected objects that appear in this image
objectX	the id of detected object X (string)
croppedX	the cropped image of detected object X (bytes)
cvX	the features for the cropped image of detected object X. This is the binary representation of a *Protocol Buffers* message that encapsulates all computed features (bytes)

Figure 7.3: Image features column family

Object instances		
key	row key. This is the object's ID (string)	
metadata	object-name	name of the object (string)
	object-description	a short description of the object (string)
image-index	first-available-image-id	the first available image id (integer)
imageX	image-hash	the hash of the cropped image containing the current object (string)
	image-contents	the contents of the large image containing the current object (bytes)
	cropped-image-contents	the contents of the cropped image representing the current object (bytes)
	distance-to-viewer	the distance between the viewer and the object (float) (*optional*)
	inferred-gps-position	the inferred GPS position of the object – a binary representation of the *Protocol Buffers* localization features message (bytes) (*optional*)

Figure 7.4: Object instances column family

For the implementation of the database queries, we decided to use *Hector*, a high-level Java client for *Apache Cassandra*. Its main advantages are:

- object oriented interface;
- type-safety;
- encapsulation of the *Thrift* API.

7.5 Image Recognition Component

7.5.1 Why a Separate Server

As stated above, we decided to implement the image recognition component as a separate server, written in C++. This is mainly because:

- the image recognition component is CPU intensive. It is easier and more cost-effective to differentiate between *I/O intensive* and *CPU intensive* components. For example, Amazon EC2 offers highly specialized instances for such types of applications.
- we are using the OpenCV library for image recognition, written in C++.

The *image recognition component* contains implementations of:

1. classes that implement the *communication layer* between the *server* and the *image recognition component*, using the **TCP** stack;
2. classes that extract and match features, giving a *score* (or *a level of confidence*) to the pair (*query image, object instance*);
3. classes that remove incorrect matches.

7.5.2 The OpenCV Library

OpenCV[6] is a library of programming functions for computer vision software. Supported functions range from *general image processing functions* and *transformations* to *feature extraction* and *machine learning*.

OpenCV also provides a version for Android, which offers similar features. This will ease the process of splitting the computational work between the mobile application and the server.

The process of detecting whether an object appears in an image or not is done by extracting the features from the image and matching them against available object instances.

7.5.3 Feature detection

For feature extraction we experimented with two feature detectors:

1. SIFT (*Scale-Invariant Feature Transform*) detector;
2. SURF (*Speeded-Up Robust Feature*) detector;

The SIFT Detector

This detector was designed to be **invariant** to *uniform scaling* and *changes of orientation* and **partially invariant** to *changes in illumination* and *affine transformations*.

The *SIFT* detector uses several techniques, algorithms and heuristics to extract features ([7],[8],[9]). These can be divided into the following steps[12]:

1. **generate scale spaces**. *SIFT* expands on the idea of correctly removing unnecessary details by using *gaussian blur*. It computes *scaled-down* sets of *increasingly blurred* images, called *octaves*;

[12] http://www.aishack.in/2010/05/sift-scale-invariant-feature-transform/

2. **compute the LoG - Laplacian of Gaussians**. This is achieved using an approximation of the laplacians (*i.e. second derivative*), using *DoG - Difference of Gaussians*;
3. **find keypoints**. This is done by computing local maxima and local minima on the previously computed *DoG* images;
4. **remove uninteresting keypoints**. Most of these are located on edges, or are in low-contrast areas;
5. **associate orientation information with keypoints**. This is done by computing a *histogram* for the *magnitude* and *orientation* values. The orientation will be chosen according to the highest values in the histogram (we should note that, in this step, a single keypoint can be duplicated, should more than one histogram maxima exist);
6. **compute features**. A window of 16×16 pixels is chosen, with the keypoint in the center. This is then divided into 16 4×4 windows; gradient magnitudes and orientations are computed for each pixel in each window and placed into an 8-bucket histogram, scaled according to the magnitude (each bucket corresponds to a 45° arc). 128 values will result (16 windows $\times 8$ buckets) which, after normalization, form the *feature vector*.

The SURF Detector

This detector was designed as an improvement to the *SIFT* detector. It is claimed to perform reasonably well in terms of quality, compared to the *SIFT* detector, while also being several times faster [10]. Much of this performance increase is due to:

- using local intensity differences around keypoints, *i.e. they apply the following two kernels*:

-1	1

-1
1

- in an intermediary step, computing the *Integral Image*, that will allow constant computation of the sum of intensities (*i.e. area*) of any rectangular section.

7.5.4 Feature Matching

We experimented with a *brute-force, k-nearest-neighbor* algorithm to select features, as well as with the *FLANN-based* matcher. We found that, although the *FLANN-based* matcher is randomized, it achieves similar results in terms of quality, and is roughly 3 times faster than the *brute-force* matcher.

Number of images in test	**Brute-force matcher**	**FLANN-based matcher**
8 images	10 seconds	3 seconds
12 images	14 seconds	4 seconds

Figure 7.5: Comparison of run-time for a test suite

This is because the *FLANN* matcher uses more optimized data structures, which allow for faster *k-nearest-neighbor* algorithms to be implemented:

- *K-D trees* and *Randomized K-D trees*;
- *K-means trees*.

7.5.5 Optimization

The use-case of our application makes it difficult for feature detectors and matchers to achieve good performance:

- the target objects we are matching against are buildings. While there are several landmarks and buildings that have distinct features (*i.e. The Eiffel Tower, The Guggenheim Museum*), most of them are similar, especially if they are removed from their context (*i.e. GPS position, background*). For example, consider all the apartment buildings in Eastern Europe, or the similarities between *L'Arc de Triomphe* in Paris and the one in Bucharest,

or good latency:

- feature matchers work in nearly $O(nlogn)$ *time*, with some (i.e. *FLANN*) managing to work in .1 of that. This means that it is impossible to try and find each and every possible object in an image (*e.g. if we had* 10000 *tagged objects and* 10 *instances for each of them, we would have to run the matching algorithm against* 100000 *pairs*).

We therefore need to improve both the accuracy and the latency of the matching process.

Finding correct subsets of images

As discussed above, we implemented a heuristic to detect which objects can be present in a query image. This takes into account the *GPS position* and the *compass orientation* of the user. In addition to that, we implemented a system for ranking instances of objects. For the objects that are in the user's line of sight, we will choose the top X instances, based on this ranking system. The ranking system will work as follows:

- when the rate of valid matches between the query image and a particular instance of an object is higher than some threshold T, then the image receives a boost B_+;
- when the rate of valid matches between the query image and a particular instance of an object is lower than that threshold, then the image receives a penalty of B_-.

The ratio $B_+ : B_-$ should be approximately $10 : 1$. We must take into account the fact that we will have instances representing different sides (angles) of the object; therefore, matching between a query image representing an angle of the building and a valid instance representing a completely different angle should not incur a significant penalty. Conversely, the bonus should be significant enough to boost a valid instance even if it received penalties for images from different angles.

Removing uninteresting matches

We implemented a system which allows purging features and matches that aren't relevant. For example, buildings' corners might perform extremely well (*i.e. corners from the query image and an object instance might be matched with extremely low distance — or high confidence*), but that might not relevant for the matching process. Another example are features that have very high distance — or low confidence when matched (*i.e. an advertising panel that appears in both the query image and the object instance, but shows different ads*).

We were interested in implementing and experimenting with several ideas of *purging* uninteresting features and matches, therefore we are using a generic approach by using the *abstract* **MatchPurger** *class* to allow for easy injection of more algorithms and heuristics.

Clustering of features

One of the most basic ways of removing invalid feature matches is clustering. We are computing the *mean* distance of matches, as well as the *standard deviation*:

$$mean = \frac{\sum_{m \in Matches} distance(m)}{|Matches|}$$

$$std = \sqrt{\frac{\sum_{m \in Matches} (distance(m) - mean)^2}{|Matches|}}$$

and then removing the matches outside the range:

$$(mean - threshold * std, mean + threshold * std)$$

Removing low-performance features

Even if we associate objects not only with the images they appear in, but also with their geographic context, problems still remain with similar architecture. With the same example in mind, of the architecture of buildings in Eastern Europe, we can observe two main issues that might result in incorrect matching:

1. adjacent buildings with similar features in the query image;
2. repetitive sections in the buildings (*e.g. windows*).

To cope with these problems, we have implemented a way to remove the matches that have the following property: *the feature from the query image can be best matched with 2 or more features in the object's instance image, with roughly the same probability.* We are removing the matches from both the query image's point of view, and the object's instance point of view, as follows:

1. compute all the possible matches between the *query image* and an *object's instance*. For each possible match, we will store at most 2 pairs (F, f), where F is the feature from the *query image*, and f is a possible matched feature in the *object instance*. These represent the top 2 choices, based on the distance between F and all possible f;
2. we ignore all matches for which the top 2 pairs (F, f) have a relative difference in distance lower than some threshold:

   $$\frac{\min(dist(F, f1), dist(F, f2))}{\max(dist(F, f1), dist(F, f2))} < 1 - \varepsilon$$

   This means that are throwing out matches for which the top choices are equally probable;
3. we execute the steps above for the reversed pair of images (*i.e. the object instance and the query image*;
4. we remove all matches from the first set, that do not have a corresponding match in the second set (*i.e. all matches that were found for the pair* (query image, object instance)*, but not for the pair* (object instance, query image)*)*

In our tests, we achieved a rate of removal of $90\% - 96\%$ after the first 3 steps. The final step resulted in an additional removal of roughly 50% of the remaining matches.

Interpreting the geometric properties of matches

Another aspect we must consider is the possible transformations on objects. In most of the use cases (the typical, normal use-case), it is impossible for the following transformations to occur:

- major skew;
- mirror;

very unlikely for:

- rotation;

and very likely for:

- minor skew due to the camera lens;
- translation;
- scale.

This will allow us to impose some geometric restrictions on the matches. If we were to place the two images next to each other and draw lines between the matching key points, then the lines corresponding to the best matches will be parrallel (if the object and its instance in the query image are the same size) or skewed (if one of them is larger). We can use the simple clustering algorithm stated above for the slopes.

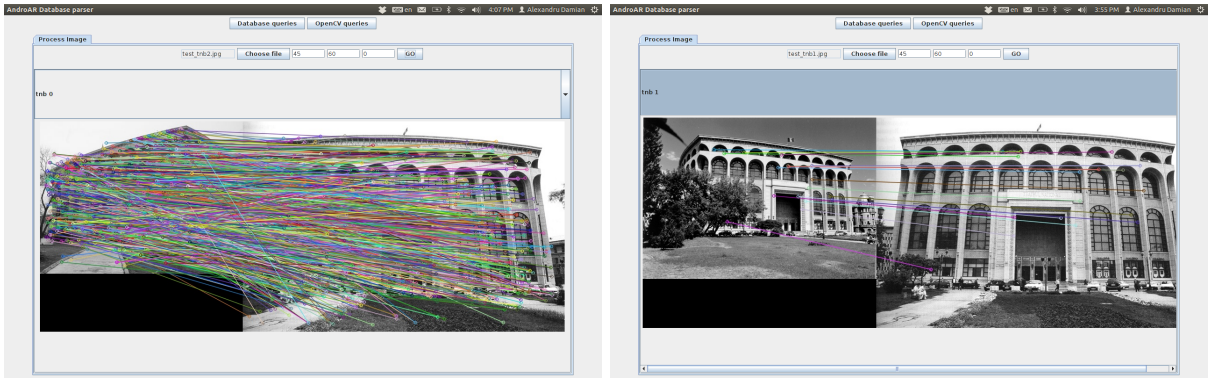
Fundamental Matrix. Epipolar lines. RANSAC

For further match purging, we can take advantage of a simple observation: *if matching is correct, the query image and a particular object instance can be seen as made by two cameras* looking *at the same object*. This means we can take advantage of algorithms based on *stereo-vision*.

The **epipolar line** of a particular point X in the first image is an imaginary line on which the same point X, seen by the second camera, can lie on. The **fundamental matrix** is the mathematical relation between how a point from the first image is mapped in the second image.

Therefore, assuming that we know what the fundamental matrix is for a pair of images, we can purge all invalid matches. However, OpenCV needs at least **7 valid matches** in order to compute the fundamental matrix. We will use the **RANSAC** (RANdom SAmpling Consensus) algorithm to compute an estimation of the fundamental matrix. The idea behind RANSAC is to randomly select as few matches as possible, compute the fundamental matrix and then select all the other valid matches. Should any of the randomly picked matches be invalid, the RANSAC algorithm will ouput very few valid matches.

After applying the **Repetitive Features Purger** and the **RANSAC Match Purger**, we obtained excellent results:



left: before purging matches, right: after purging matches

Figure 7.6: Amount of invalid matches purged

7.6 Testing GUI

We needed a *debug* version of the system that allowed visual inspection of the matching of features between the query image and all the instances of all the possible objects present in the user's line of sight.

We decided to implement the testing GUI using the Java™ Swing API and use it as a standard desktop application. This way, we have several advantages:

- we can control and fine-tune queries;
- by using a *debug* version of the communication layer between the system's components, we can receive a reply containing significantly more information than the production version. This will allow us to compute image recognition performance statistics and to visually inspect the success of the matching algorithms we use.

and we remove the disadvantages of using a *debug* mobile application:

- visual inspection of the matching algorithms is tedious when using a mobile application, given fewer controls and smaller screen size;
- a mobile application will incur another step in the data's path from query to reply;
- we do not need the features we would normally use in the production version: real localization capabilities, real camera feed, etc.

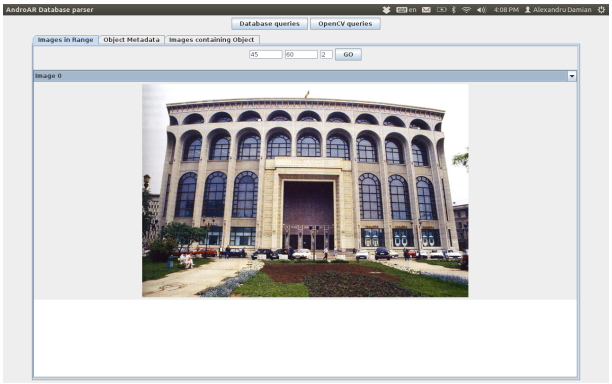


Figure 7.7: Database queries screen in the Testing GUI

7.7 Mobile Application

7.7.1 Android Platform

Mobile platforms are currently dominated by the *Android* and *iOS* platforms. We decided to use the *Android* platform to implement the mobile application, mainly because:

- the platform is open-source, well documented and supported by a large community;
- applications are written in Java™ , which is consistent with our choice for the Communication library.

We implemented the mobile application to support the *Gingerbread (2.3)* version of the *Android* platform. More than 50% of the *Android* smartphones run the *2.3* version, and 25% more run newer versions. In addition to this, so far we do not need any of the new functionality of the newer versions. The mobile application is fully compatible with future versions, but not backwards.

The **interface** was created using XML layouts, as is recommended for Android to reduce load times.

Figure 7.8: Main screen of the AndroAR prototype

Camera integration was implemented using SurfaceView[13] and several callbacks issued when the camera view is initialized, changed or destroyed. Everytime the main camera preview (activity) is interrupted, the camera is stopped and a new camera instance is obtained when the activity is brought back to foreground.

GPS positioning and **compass orientation** were obtained using the android internals, through listeners that update them when changed.

The **user experience** and **user interface** are clear and straightforward, allowing the user to receive replies from the servers on the camera preview main activity, but in the same time permitting screen captures to allow cropping and tagging of buildings. After the user has provided proper keywords and a description for the selected landmark, the information is sent to the server, using a TCP connection. Image cropping was done using an android built-in intent and, thus, before and after actions were needed, in order to parse the newly obtained data. Also, as an improvement, saving an image to the storage space is done asynchronously, in background, by extending the AsyncTask[14] class.

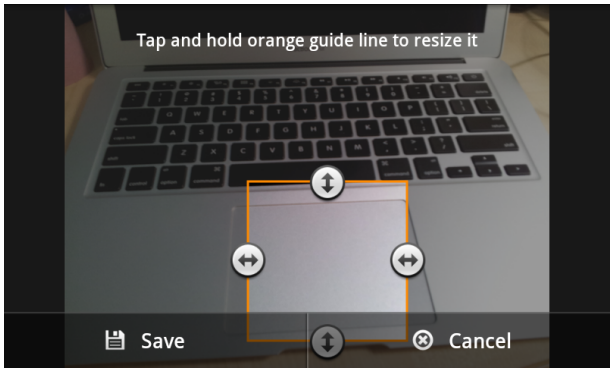


Figure 7.9: Screen caption of the image crop step in the AndroAR prototype

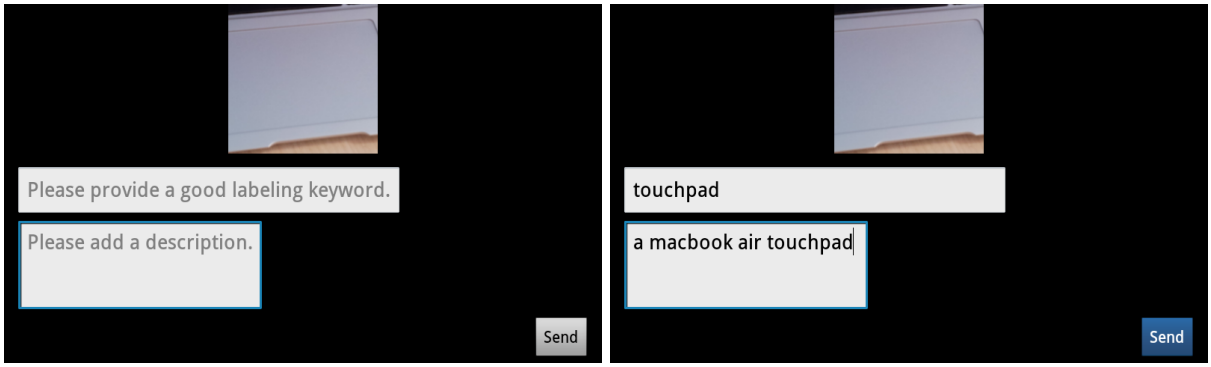


Figure 7.10: Screen captions of the tagging step in the AndroAR prototype

Note: development of the Android application has been done by Andrei Petre[15].

[13] http://developer.android.com/reference/android/view/SurfaceView.html
[14] http://developer.android.com/reference/android/os/AsyncTask.html
[15] student at the Polytechnic University of Bucharest, Faculty of Automatic Control and Computers, second year

7.8 Tools

Apart from the techologies used in the implementation (*Apache Cassandra, Protocol Buffers, OpenCV, Android*), we will present other tools used during development in the following list, along with some details:

Git. GitHub

Git is currently one of the most feature-rich and easy-to-use version control systems. It has support for advanced features, such as branching and tagging. We used *git* in conjunction with *github.com*, a service that offers free hosting of open-source projects, using git. Features of *github.com* include a visual interface to track changes in commited files, tickets and milestones.

Eclipse IDE[16]

As one of the most popular IDEs on the market, Eclipse has numerous features:

- code highlighting;
- auto-completion;
- refactoring tools (*i.e. renaming of classes, functions and variables*);
- auto-build;
- class-hierarchy navigation and search;
- plugins.

Ant[17]

Ant can be considered a replacement (or an enhancement) of the *make* utility, designed specifically for Java™ projects. It has the same functionality as *make*, but uses XML and offers many shortcuts to ease the Java™ build process.

[16] http://www.eclipse.org/
[17] http://ant.apache.org/

Chapter 8

Results and Conclusions

We have successfully implemented and tested a prototype of the proposed application.

8.1 Installation

Installation of the application is relatively straightforward, using Ant, but only assuming all prerequisites are met:

- sun-java-1.6 (Java^{TM} compiler);
- libprotoc 2.4.* (Protocol Buffers compiler);
- opencv 2.3.* (OpenCV library);
- Cassandra 1.* (Cassandra database);
- Hector 1.* (Hector API, working on top of Cassandra).

For testing purposes, the following requirements are needed:

- Android SDK, version 2.3 or higher;
- (*optionally*) a smartphone running Android 2.3 or higher.

We successfully installed and ran end-to-end tests on:

- **remote VPS**: quad-core Intel Xeon @ 3.2GHz, 2GB RAM;
- **localhost**: quad-core Intel I5M @ 2.4GHz, 4GB RAM.

Test results showed a reply latency of approximately 3 seconds when a query using a 1920×1080 resolution image was issued. Time spent on a query was divided as follows:

- 40% interaction between the server and database, along with forwarding queries;
- 40% OpenCV processing;
- 20% network time.

8.2 Results

We are extremely satisfied with the results we obtained in terms of latency and we target a 25% to 50% reduction when using smaller images and better caching.

In terms of performance, using the purgers presented in chapter 7 we have managed to achieve excellent results, even when matching *difficult* buildings (*i.e. buildings with common architecture or repetitive features*).

Latency tests were conducted using a verbose version of the mobile application, while performance tests were conducted using the Testing GUI, for visual inspection, backed by a suite of bulk tests ran against a separate, testing database.

8.3 Problems

During the development process, we encountered several problems:

- **difficulty using Cassandra and Hector**. Cassandra is, by design, difficult to use programmatically, and the most efficient and easy-to-use solution is Hector. However, the API doesn't support all possible types of queries and is poorly documented. We were forced many times to reverse engineer test-cases in order to understand the correct usage and behaviour of Hector.

  Also, debugging correct insertions in the database was tedious since a GUI to explore the database is not offered. Only a CLI client is available, which has its limitations, mainly because Cassandra is not typed (keys and values are stored as *byte arrays*). We were forced to create unit tests for all types of queries (see **/AndroARServer/src/test/-com/androar/CassandraDatabaseConnectionTest.java**).

- **matching**. The OpenCV library has many features and algorithms implemented to extract and match features. However, there is no out-of-the-box solution for our use-case. We had to research and implemenent heuristics, adapted to our use-case, in order to extract subsets of correct and relevant features and match those.

- **Protocol Buffers**. Protocol Buffers are extremely efficient and message definitions are intuitive and fast to write. However, setting up the environment needed to support them is rather tedious. Also, creating messages in Java™ is more complex and more verbose than in C++.

- **communication between Java™ and C++**. We had several issues with communication between the Java™ and C++ components, mainly with data delivery through sockets.

- **debugging**. Creating and running tests to determine whether heuristics yield correct classifications was not difficult. However, we needed a more detailed view on the matching process, apart from the *binary yes/no* results, a way to visually inspect the features that were matched. Thus, we implemented the testing GUI, along with a *DEBUG*, *more verbose* version of the communications layer.

  Below is an extract of some tests created for the matching process:

```
********** TEST 1 **********
image {
  image_hash: "src/test/test_intercontinental.jpg"
}
detected_objects {
  ...
  id: "intercontinental"
}
********** TEST 2 **********
image {
  image_hash: "src/test/test_tnb1.jpg"
}
detected_objects {
  ...
  id: "tnb"
}
********** TEST 3 **********
image {
  image_hash: "src/test/test_tnb2.jpg"
}
detected_objects {
  ...
  id: "tnb"
}
```

- **Android**
  - **communication**. When we implemented the query loop, we were faced with restrictions on the Android platform regarding blocking socket reads.
  - **camera**. All Android application are required to serialize access to the camera by using *callbacks*. This poses several problems, because we have no control over when the callbacks will be executed and when camera frames will become available.

8.4 Conclusions

During the development process of this system, we have gained deeper understanding of augmented reality research and image recognition algorithms. We have drawn the following conclusions:

1. ambient intelligence and, in particular, augmented reality, have gained much momentum in the past years; research and development in these fields are offering more advanced hardware and software (*i.e. Google Glasses, BMW head-up display*) and are moving towards integration of data from as many sensor types as possible;

2. acquiring sensor data is not straightforward and is prone to errors; ambient intelligence applications must also deal with these errors;

3. implementation of queries using the Cassandra database was tedious. Had we known that, we would have most likely chosen MongoDB to store our data.

4. the system design was successfuly implemented and a prototype is available. It works as expected and offers high quality results. However, latency is still an issue and future work is needed to ensure a seamless experience for future users.

5. the direction we chose for the design of this system is also pursued by major companies and research institutions, which contributes to proving the validity of our idea.

Chapter 9

Future Work

Our plan is to make *AndroAR* production ready. This will require both the implementation of new features and the optimization of existent ones.

Support for Sparse Queries

As noted in the previous chapters, it is impossible to issue queries for every frame of the camera feed, whether our target is *30 frames per second* (because of the amount of queries) or *1 frame per second* (because, even then, the reply will arrive delayed and will, thus, be out-of-sync). We need to implement a way to track objects in the camera feed; alternatively, we can understand how the phone moves and then infer object movement.

Detecting how the user moves their phone seemed the easier to implement choice. Our initial thoughts were to use the internal *accelerometer* to detect the direction and speed of the phone movement, since there are many smartphones with this hardware embedded. We would then have had to compute the distance covered and translate the bounding boxes accordingly. However, we found it too tedious and unsecure to compute distances based on the accelerations: if we are issuing queries every $4-5$ seconds, then we mostly rely on computing how the images translated to correctly display the bounding boxes; even the smallest error in the accelerometer sampling will result in a huge offset during these $4-5$ seconds *(due to integrating acceleration* twice *to obtain distance)*.

Therefore, we decided to detect translations using the *OpenCV* library for Android to track features in images. This will allow for more accurate computation of translations.

Passive Example Generation

Using the *OpenCV* library for feature tracking will also allow us to generate more examples, by storing instances of objects in frames adjacent to the one that generated a valid match, or to the one that was selected by the user to tag a building with metadata. A similar approach, called *Predator*[1], was already implemented, and achieved excellent results.

Image Normalization

Most image recognition algorithms have low tolerance to lighting changes. The *OpenCV* library transforms images to grayscale and normalizes intensities before attempting to extract any features.

[1] http://info.ee.surrey.ac.uk/Personal/Z.Kalal/tld.html

However, we can do more and, for example, estimate the amount of lighting based on the current time of day (a feature we have available on any smartphone).

Inferring GPS Position

Currently, when a storage request is issued, we assume that all objects that appear in the camera frame are located at the GPS position provided by the phone (*i.e. all objects are stacked on top the current user's position*). It would be nice to be able to estimate the GPS position of objects, based on the user's viewport and the camera's focal length.

Database Improvements

The current system design is homogenous. All information is stored in a single database (we refer here to a collection of machines, acting as a distributed database), even though it is clear that any query will interact with only a fraction of the database (*e.g. queries that originate in Romania will not interact with the part of the database storing information for buildings located in the US*).

Given these geographical issues, we can take advantage of sharding:

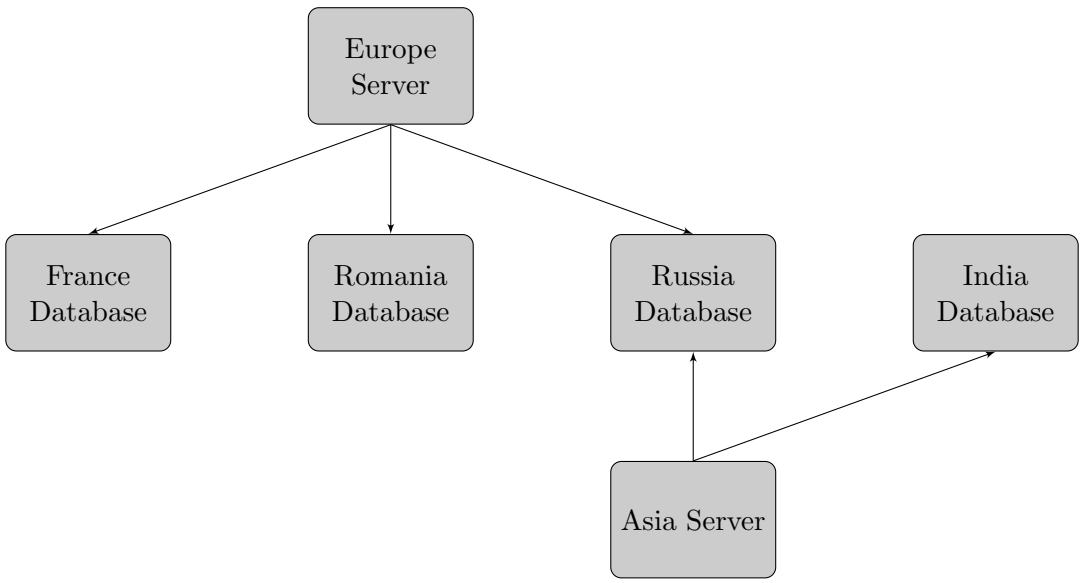


Figure 9.1: Geographic explanation of the advantages of sharding.

Other Improvements

At present, communication is done using *Protocol Buffers*, over *TCP*. We are planning to transition to *UDP* or *RTP*, protocols that are more suitable for media delivery. If so, *Protocol Buffers* might become unsuitable and other solutions will have to be researched.

Also, security aspects will have to be considered. Some of the most common security precautions that will have to be implemented are:

- prevention of *denial of service* attacks;

- spam purging, when issuing store requests:
  - if the building is already tagged in the database, we will treat the store request as a normal query to ensure validity and quality; we will store the instance only if the confidence of it representing the proposed building surpasses a certain threshold;
  - if this is the first time the building is tagged, then we will rely on user validation. It would be nice to be able to search the web for the proposed building name and match the provided image against the top search results.
- user accounts, most likely by using *open authentification* (*i.e. OAuth, Facebook login, Google login*)

List of Figures

2.1 Screen caption of the Street View web application 4
2.2 Screen caption of the Wikitude Drive application 4

3.1 Example of augmentation of a user-submitted photo in Street View 6

4.1 Basic system design 8
4.2 Design option for scaling the system, using *3* pools for *servers*, *image recognition components* and *databases* 9
4.3 Design option for scaling the system, with each *server* being connected to the corresponding *image recognition component*, therefore using only *2* pools. 9

5.1 Example of using the line of sight to retrieve a subset of possible matches 10
5.2 Life of a query. 11
5.3 Life of a storage request. 13

7.1 Technologies used in the system implementation 17
7.2 Example of how the caching system works 20
7.3 Image features column family 24
7.4 Object instances column family 24
7.5 Comparison of run-time for a test suite 26
7.6 Amount of invalid matches purged 30
7.7 Database queries screen in the Testing GUI 31
7.8 Main screen of the AndroAR prototype 31
7.9 Screen caption of the image crop step in the AndroAR prototype 32
7.10 Screen captions of the tagging step in the AndroAR prototype 32

9.1 Geographic explanation of the advantages of sharding. 39

Bibliography

[1] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, J-C. Burgelman, *Scenarios for Ambient Intelligence in 2010*, 2001

[2] Stuart Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, Third Edition, 2010.

[3] Richart O. Duda, Peter E. Hart, David G. Stork, *Pattern Classification*, Wiley-Interscience, 2001.

[4] Nancy Lynch, Seth Gilbert, *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*, ACM SIGACT News, Volume 33 Issue 2, 2002.

[5] Eben Hewitt, *Cassandra: The Definitive Guide*, O'Reilly, 2010.

[6] Robert Laganire, *OpenCV 2 Computer Vision Application Programming Cookbook*, Packt Publishing, 2011.

[7] David G. Lowe, *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, Volume 60 Issue 2, 2004.

[8] David G. Lowe, *Object recognition from local scale-invariant features*, International Conference on Computer Vision, 1999.

[9] David G. Lowe, *Local feature view clustering for 3D object recognition*, IEEE Conference on Computer Vision and Pattern Recognition, 2001.

[10] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, *SURF: Speeded Up Robust Features*, Computer Vision and Image Understanding, Volume 110 Issue 3, 2008.

Appendix A

Directory Structure

Below are details regarding the directory structure of the current system implementation:

- **AndroARComm**: contains common classes used by the Java™ components;
- **AndroARCV**: the image recognition component implementation. The *Match Purger* implementations presented in chapter 7 are:
  - **GeometryMatchPurger**: class that interprets the geometric properties of valid matches;
  - **KNNMatchPurger**: class that removes low-performance features;
  - **RANSACMatchPurger**: class that removes incorrect matches by using the fundamental matrix computed through RANSAC;
  - **STDMatchPurger**: class that uses clustering of features to remove outliers.
- **AndroARCV/test**: contains tests for the image recognition component;
- **AndroARGUI**: the mobile Android application implementation;
- **AndroAROpenCVGUI**: implementation of the Testing GUI;
- **AndroARServer**: implementation of the Java™ server;
- **AndroARServer/test**: contains unit tests for the server and database connection.