PeeKaBoo!

An Android application to share media content using open technologies

A DISSERTATION SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE MSC IN ADVANCED COMPUTING TECHNOLOGIES

BY LUIS FIDEL CAMPOY SANCHEZ

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

BIRKBECK COLLEGE, UNIVERSITY OF LONDON

SEPTEMBER 2015

Word Count: 12312

Academic declaration

This report is substantially the result of my own work except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service. I have read and understood the sections on plagiarism in the Programme Handbook and the College website.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Peekaboo is a game in which one player hides his or her face, pops back into the view of the other, and says Peekaboo!

Table of contents

TABLE OF FIGURES

TABLE OF TABLES

0 ABSTRACT

This document describes insights into the design work of PeeKaBoo!, an Android application able to share media files in a web-based manner which does not require specific software on the client side. This is achieved by running a web server in the cell phone that will make the selected files available over the network. This application has a number of different features, explained in this document in a top-down structure: starting with the presentation layer and finishing at the server lower level details.

For most aspects, a number of alternatives are discussed. The result of the programming work that is detailed in this document is a totally functional software tool.

1 DESCRIPTION OF THE TOOL

1.1 OBJECTIVE

The main aim of this project is that of implementing an Android application, PeeKaBoo!, that allows the users to graphically share media content, i.e. not in the form of an attachment or downloadable file, but displaying it using HTML, and, therefore, avoiding the necessity of installing any application on the receiver device. The application will allow the users to pick a file already stored in their phone, such as pictures, videos, or songs, and display it on the local network via HTTP protocol.

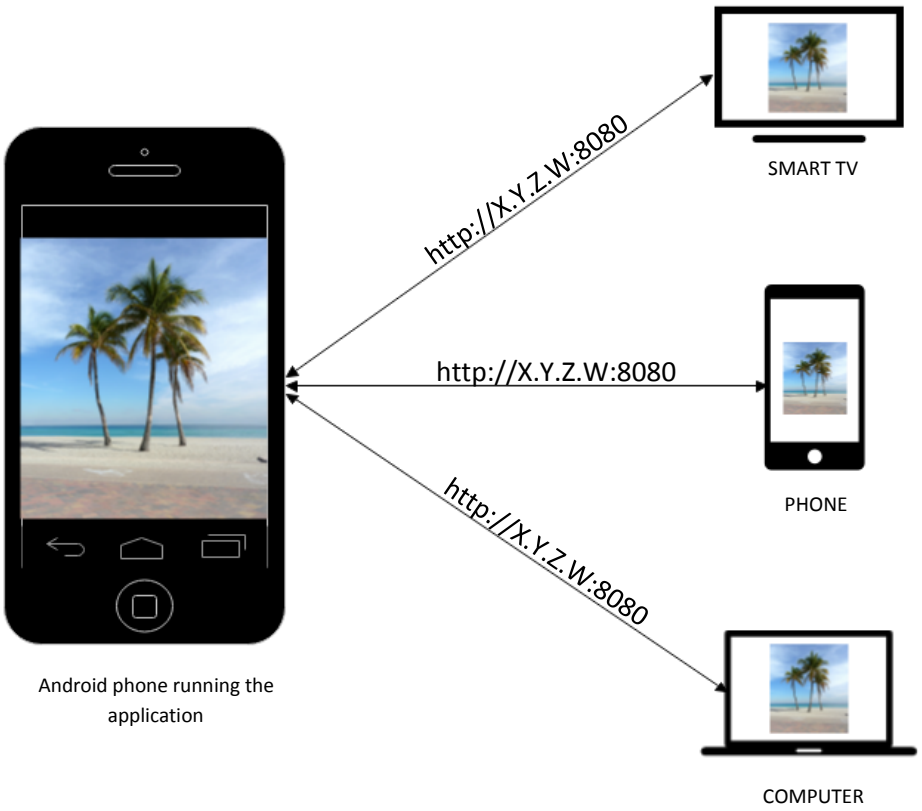


Figure 1.1: Graphic Representation of the Functionality

The application will be reachable through the local network only and able to concurrently serve more than one client. It is also a key objective that it guarantees synchronization of the clients with the content selected in the application. This content can only be pictures, video, or audio files previously stored on the phone.

1.2 SCENARIOS

PeeKaBoo! seeks to solve two common problems faced when trying to share content stored on a phone with an audience:

1.2.1.1 Need for a bigger screen

One of the biggest inconveniences of showing a picture or a video on a phone to a group of people is the screen size. This tool will provide a mechanism that can easily display the same content on a Smart TV or computer, and possibly avoid many accidental head-butts! Examples of real life scenarios for this application might be meetings with friends where there is a Smart TV available, own users' who want to watch a video or picture with a bigger display, or lecturers or people giving presentations who want to display something on a computer connected to a projector.

Under this category will also fall users who may wish to open a file in a larger device not because of its screen size but because of better or louder sound capabilities, for instance for the purposes of playing a soundtrack.

1.2.1.2 Need for multiscreen

The other model problem this tool wants tackle is that of being able to show media in contexts where the spectators might not be able to place themselves near the phone. Under these circumstances, the tool will provide a simple and synchronized method for allowing them to access the data by using their phones without the need of installing any new software.

'Real world' examples of this scenario will be formal meetings (e.g. launch meetings) where people gather at a table and are unable stand close to the person holding the phone. This will also suit situations like the ones described above where there is not the possibility of accessing a larger screen, or it is more convenient to have the content shared between the different phones.

1.3 Example Use Case

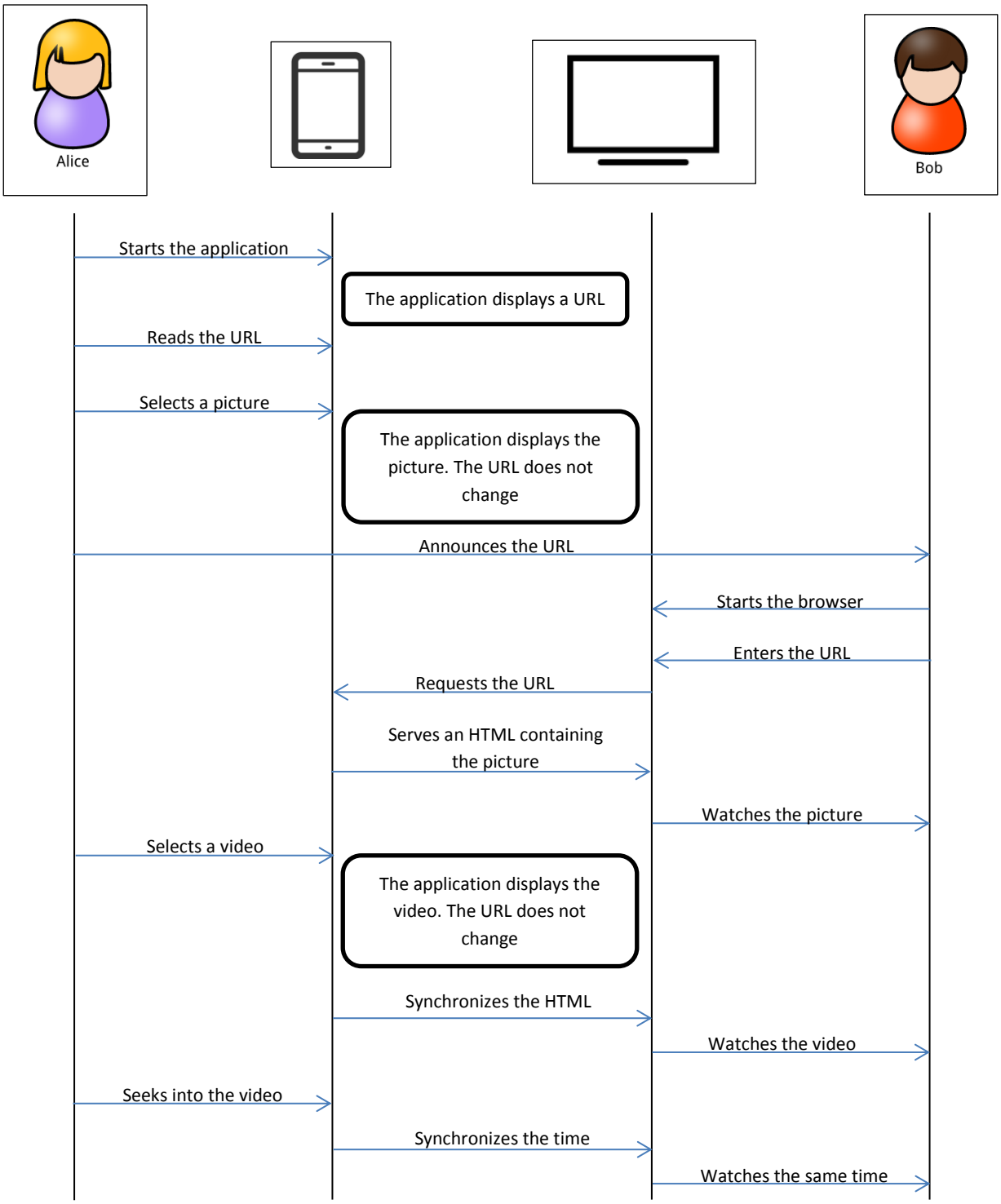


Figure 1.2: PeeKaBoo! Use Case

2 WHY ONLY LOCAL SHARING?

At this point, the reader might wonder why this tool is designed to only provide sharing through the local network. Obviously this restricts the potential number of clients only to those connected to the same LAN as the phone.

The reason is that our server is going to be announced by its IP address, not by a domain name. Public IP addresses on cellular mobile networks usually have an even shorter lease than those of home ISPs, mainly because the portability of phones makes them change their position in the cellular network topology, and, therefore, their addresses. The huge number of devices accessing a limited range of addresses is also a reason for frequent reorganizations. For communications initiated by the phone, or those that are already active, this change of address will have no implications as there are mechanisms in Mobile IP to handle this situation.

In the case where the communication is initiated in the network, not in the phone, there are several techniques (Lewerentz, et al., 2002 pp. 73-85) to resolve the issue of locating the host whatever its address might be, like SIP or Dynamic DNS. DNS is the only one supported by browsers, as it is usually used in combination with HTTP. However, that would require a domain name registration for each phone running the application, which is not only expensive but technically unviable.

In addition to these constraints, as cellular networks use Network Address Translation (NAT), phones do not have any information of what their public address is, including the port. This is also solved using the above mentioned techniques at the time the phone is registered in the SIP or DNS server, as the server will receive not the local, but the public address. Therefore, we cannot even establish the public IP address of the clients without creating a service in the middle where the application can first connect and find its address, which is expensive, as this server should be always be available and identifiable by having either a permanent public address or a domain name. Some workarounds, for example using popular sites such as *whatsmyip* can be done, which will reduce the costs, but the URL may change every few minutes, without any automatic mechanism to advise clients of the new address. That is the main issue.

In conclusion, as we are taking information from the network layer (IP address) into the application layer (making it the URL), the application will not be able to work in any environment that does not guarantee that the addresses are static, at least for the session. This only happens in LANs.

3 MODULAR DESIGN

The functionality of this tool has been achieved following a divide and conquer paradigm that has originated three different modules. There are two main blocks that look obvious: the Android application and a set of browser-compatible files that allow the visualizations of pictures, audio, or video files from a browser. The former block, the Android application, can be divided in two blocks as well: a GUI that allows users to select a file, and a web server that serves that file. This is how the application is structured and also how the document is modelled. As a brief introduction to provide the reader with very basic information about the block, I have written the following definitions that can be seen graphically in Figure 3.1, below.

- The GUI

  Goal: Allow the user to select a media file and to customize some basic parameters of the application.

  Technologies: Java and Layout definitions.

  Interactions: Provides the Server with the selected file.

- The Player

  Goal: To display a web version of the media file.

  Technologies: HTML, CSS and JavaScript.

  Interactions: Requests information from the Server about how to display it.

- The Server

  Goal: To handle the HTTP requests from the Player and respond according to information retrieved from the GUI. It is also in charge of facilitating the Player files when a client first connects.

  Technologies: Java.

  Interactions: Queries the GUI and serves the Player(s).

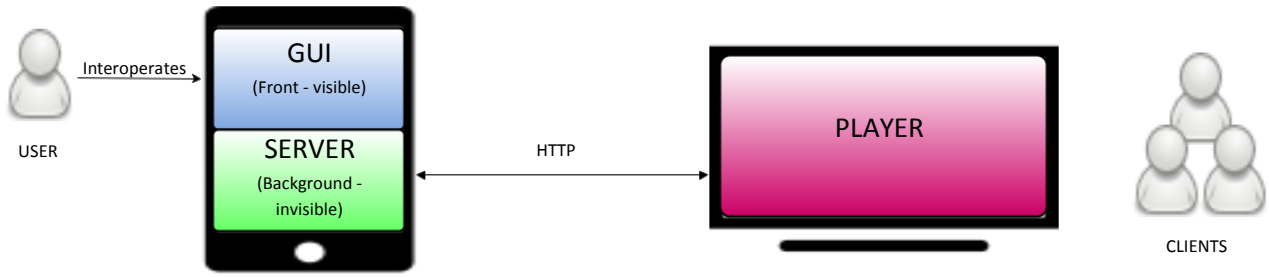


Figure 3.1: Representation of the Different Modules

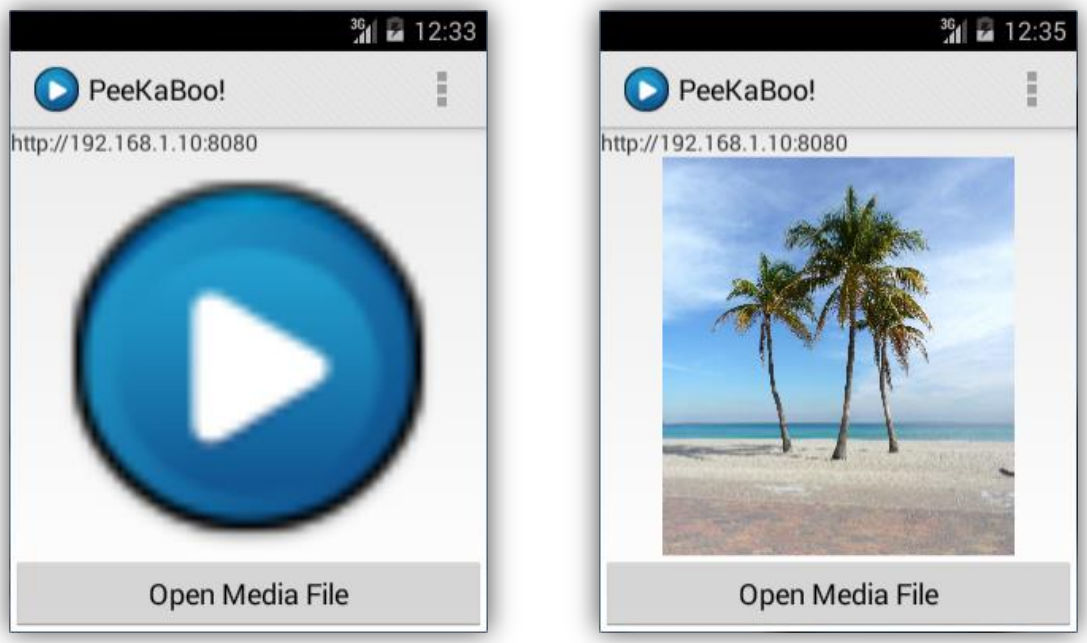
4 THE GUI

The GUI, or Graphical User Interface, is the part of the application that will allow the user to determine the content they want to share and to obtain the URL. This block of the application has a display adapted to and compatible with Android OS.

4.1 GRAPHIC DESIGN

The interface has a simple design on the basis that simplicity usually favours intuition. Thus, it will only have three elements:

- A label containing the URL
- A central element where the selected picture, audio, or video will be displayed.
- A button to search for a media file.



A B

Figure 4.1: (A) GUI When First Opened (B) GUI Once the User Selects a Picture

4.2 FUNCTIONALITY

The utility of this block of the application is that of allowing the user to control the presentation they wish to carry out. The three actions offered by the GUI are:

- Open a media file: Execution of this action is by clicking on the 'Open Media File' button. From there the phone library will be prompted to select any picture, video, or audio file. The user selects the file to display in the presentation (see Figure 4.1).

- Manage time-dependent media: the basic commands for controlling the reproduction of any video or audio file are also offered. This is on a separate menu that will only be visible for these types of files. This menu will be automatically hidden after a threshold of 5 seconds until the users touches the phone's screen again (see Figure 4.2).

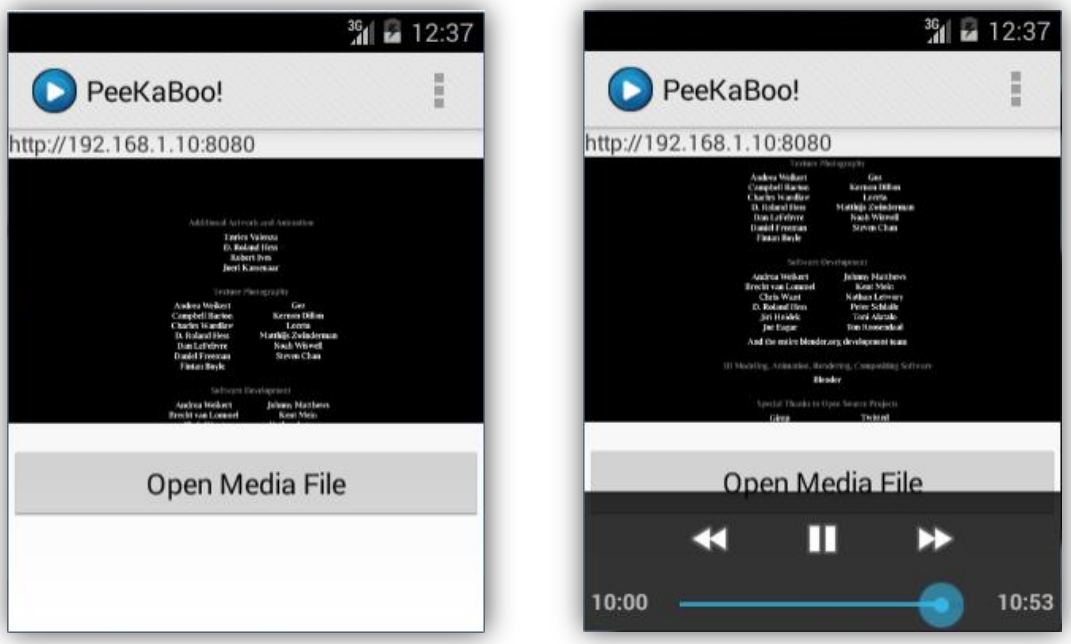


Figure 4.2: (A) Video Play (B) Video Play Showing Controls

- Configure the presentation: This action is started by clicking on the Settings menu (see Figure 4.3). A separate window to set the possible application parameters will be prompted. These parameters are:
    - Password: if filled will require spectators to enter a password.
    - HTTP port: port number to be used when using HTTP protocol.
    - HTTPS port: port number to be used when using HTTPS protocol.

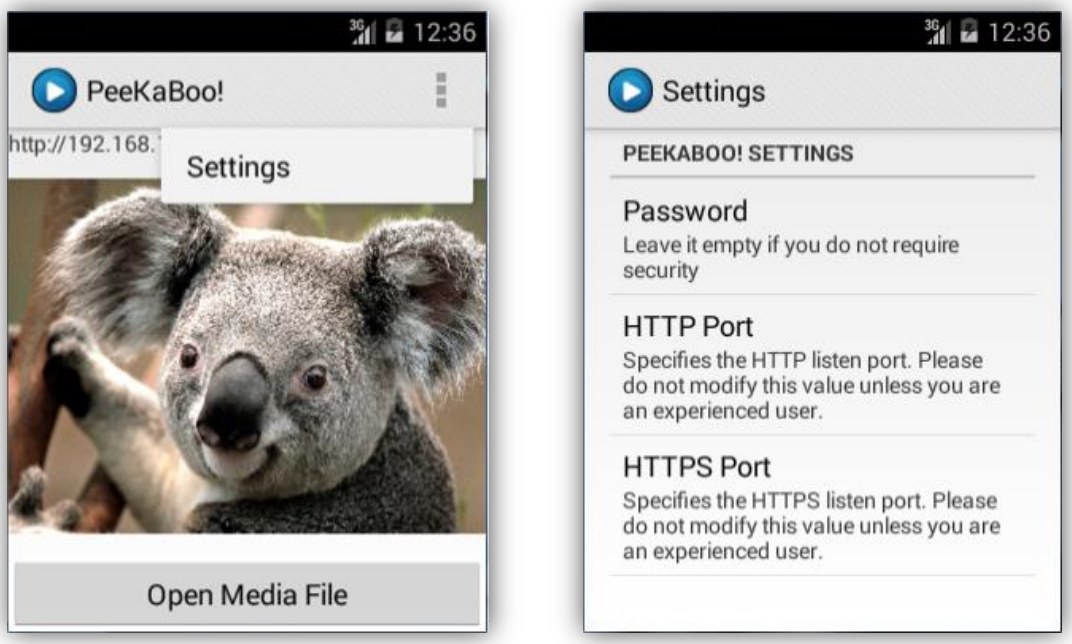


Figure 4.3: Setting Configurations

5 THE PLAYER

The Player is the interface of the application that will be visible to the clients (cf. Figure 3.1). It is based on HTML and JavaScript and stylized using Cascading Style Sheets (CSS). It can be downloaded and displayed from any browser which supports these technologies.

5.1 GRAPHIC DESIGN

As there is no functionality to be performed from the client side, this interface of the application is totally maximized to represent the content. That is, because there is no need for menus or buttons, the visual design aims to make the most of the available space on the screen (see Figure 5.1).

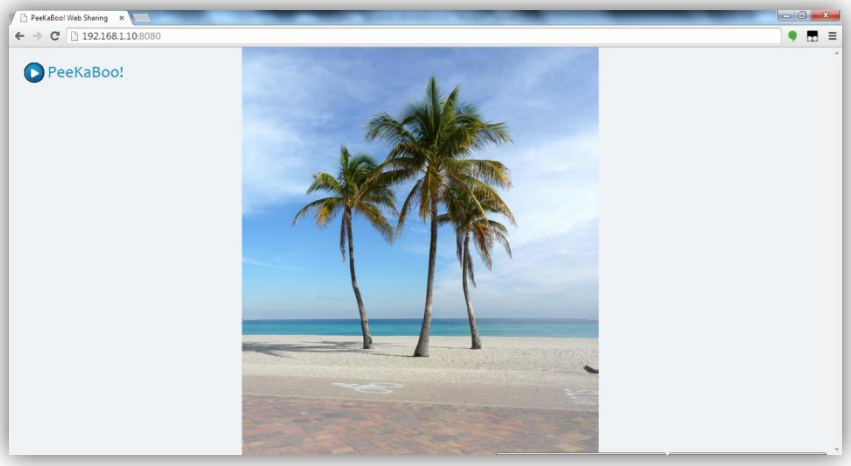


Figure 5.1: Player reproducing a picture

5.1.1 Picture representation

Images have different sizes and proportions, so that there is no static way of defining a maximized space on our HTML. This constraint can be solved by using a simple JavaScript API that calculates the size for the image element taking account of two factors: the window size and the image proportions. The use of the API called Supersized (Build Internet, 2012) with a GPL license, results in a responsive design, i.e. our HTML will be displayed differently when the application is shared with other phones, PCs, or projectors. Compare Figure 5.2 to Figure 5.1.

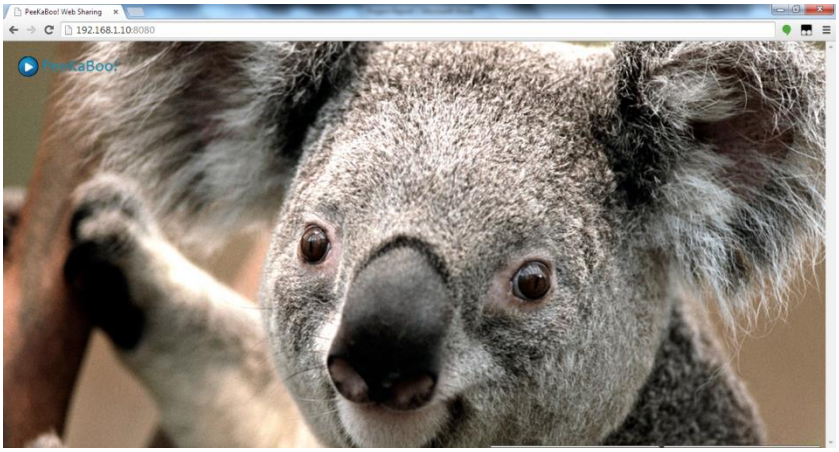


Figure 5.2: Player reproducing a picture with proportions suitable to fill the full window

5.1.2 Video and Audio Representation

The latest version of the HTML language, HTML5, provides tags to mark video and audio contents. However, the usage of these tags in our syntax would exclude a good number of Smart TVs that do not support this version. As discussed in the proposal for this project (Campoy, 2014 p. 6) one of the main goals is to design a player that can be deployed across the largest number of different devices, and for that reason a JavaScript player has been integrated.

There is a variety of JavaScript player implementations; the three main benefits that they can provide to this project are as follows:

- **Flash fallback**: the default behaviour of the player would be to deploy the audio or video using HTML5. If the device does not support this language, a Flash instance of the player will be created.
- **Uniform access:** the methods of an HTML5 and Flash player are not the same. The JavaScript API abstracts from the underlying type of reproduction and simplifies the control the player. Controlling the player is crucial for synchronization purposes, and, regardless of whether HTML5 or Flash reproduction is used, our code will be the same.
- **Uniform style:** that prevents from dealing with the different styles and properties of the different browsers. JavaScript players wrap this complexity to provide a unique graphic interface.

From the possible range of players such as Video.js, JW Player, and Projekktor, the one selected is Flowplayer (Flowplayer, 2015). The main reasons are that a free version is issued under the GPL license, there is good documentation, and it is widely used, providing a good repository of examples and discussions. I would also like to take the opportunity to mention that the makers have an efficient and helpful support system that has assisted me with this project.

Flowplayer does not decode any multimedia container; therefore, the types of videos supported depend on the client itself. The most commonly supported video formats are MP4, WEBM, and OGG for browsers implementing HTML5 and MP4, and FLV and F4V for Flash. Both types support MP3 for audio. Videos and audios are also displayed using the full window size, as can be seen in Figure 5.3.

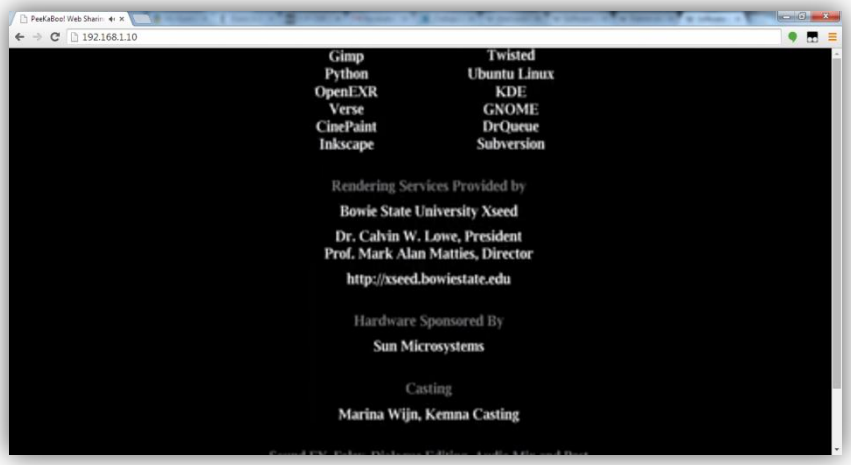


Figure 5.3: Flowplayer reproducing a video

5.2 FEATURES

5.2.1 Cross-platform

The use of HTML and JavaScript makes the application platform independent; that is, the player can be opened regardless the hardware used (e.g. computer, cell phone, TV) or operating system (e.g. Windows, Macintosh, iOS, Android OS, and TV OS). Provided that the client device has a web browser, then nothing else is required.

5.2.2 Adaptive

There are three types of content to be displayed, namely: pictures, audio, or video. This variety demands a correct mark-up for each type. For example, images in HTML should be tagged with the keyword *img*, whereas a video will require a totally different label. Therefore, the generated HTML should be adapted to the nature of the content which it will represent.

HTML5, the latest version of the language, has extended the number of available tags to include ones specific for video and audio. Using these tags will limit the number of possible client devices to those that support this version. As studied in the proposal for this project (Campoy p. 6), a good approach is to use a JavaScript web player, such as Flowplayer. This tool provides a way for setting up a video or audio player using HTML5 tags if the browser supports this version, or alternatively sets up a Flash instance of the player.

Therefore, our Player code should change depending on the nature of the content: it will use *img* tags for pictures and create a Flowplayer instance for audio or video.

5.2.3 Dynamic

Adapting the Player to the different types of media content can be done statically or dynamically. The first approach will consist of having different URLs that the users will need to request, containing the specific HTML version for that content. Dynamic adaption allows for having only one URL that will vary its HTML depending on the nature of the content. This later approach is more complex, but it does provide a higher level of abstraction to the users as they will not have to deal with different URLs.

The four options considered to achieve this feature are:

5.2.3.1 On server HTML modification

This technique consists of modifying the HTML file dynamically before sending it to the client; that is, appending strings to include the HTML image, or a Flowplayer instance in the case of a video or audio file.

This is a basic and intuitive mechanism but is demanding in terms of processing requirements for the application. Also, it is important to bear in mind that the processing will take place in a cell phone that should be able to serve several clients.

5.2.3.2 JSP: Java Server Pages

JSP is a mechanism for achieving the above by a more efficient usage of the CPU as the HTML is not modified but created dynamically, saving the work of looking up and appending strings as occurs with the above-mentioned technique.

The drawback of this method is that it introduces the need for the server to be able to interpret and process this sort of code, which introduces more dependencies on the libraries needed in the application.

As it will be discussed later on this document, it is one of the aims of this project to have its own HTTP server, and introducing the logic on the server for dealing with JSP servlets will make it far more complicated.

5.2.3.3 PHP

This approach is similar to the use of JSP and introduces the same benefits and drawbacks; it is an efficient technique but requires specialized logic on the server for its deployment. The main difference to JSP is the higher number of server implementations that support this mechanism and existing modules on the Internet. However, JSP would be better in this particular case, as we would be able to have the logic written in the same code as the server, instead of having an independent module for part of the logic. This would introduce complexity into the communication between the server's logic and the PHP-HTML generation logic.

5.2.3.4 In-client modification

The last option is that of making the client to modify the HTML. This can be achieved by sending a simple version of the webpage to the clients that they complete themselves, using JavaScript functions that are also provided. When the client retrieves the page it has to call a function that simply retrieves from the server the type of content being displayed at the application. From that information a proper HTML is generated dynamically to suit that content. After that, the content is then retrieved. This is the implemented technique, which makes use of a RESTful API that will be also discussed.

5.2.4 Content-synchronized

This property refers to the desirable characteristic that allows all the clients to get the new content being displayed in the application if the user decides to open a new file.

The HTTP protocol is based on a request-response mechanism. Therefore, there is no possibility for the server to instantiate the communication. Thus, in the event that the server content changes, it is the client who has to find this out. There are numerous general approaches and existing commercial solutions to make the server able to push a message to one or more of its clients[1].

5.2.4.1 HTML5 WebSockets

The last version of HTML introduces WebSockets, that allows for establishing a persistent connection between the client and the server in a full-duplex (bidirectional) low-latency way. Using this API, the server is able to asynchronously send any message to the client, such as an event notification. The main problem of this technique is that it requires having a HTML5 compatible browser on the client, and, thus, may restrict the portability of the application.

5.2.4.2 NODEJS + SOCKET.IO

Socket.IO is a JavaScript library that creates HTML5 WebSockets as a means for establishing communication between the client and the target server. If this mechanism is not available in the client then it will employ other alternatives, such as Flash sockets or JavaScript Polling, but it will provide a persistent communication regardless of the available technologies.

To provide this adaptability, the server also needs to be ready for handling persistent connections in any mechanism. An existing server version, also written in JavaScript, is Node.js. This is already configured to handle all the possible ways that Socket.io works, and removes the need for the developer to consider other scenarios.

The problem with this approach is that we must run Node.js on the Android cell phone, which cannot be instantiated by the application. That is, Node.js can be installed but only as an independent application. It is not a library. Our application will not have the rights to install third-party applications, and in the event it was installed, the user would have to manually start the Node.js server.

5.2.4.3 Polling

Polling refers to the technique of continuously querying the server to get information about the possible events. In this strategy, the connection is always initiated by the client but it is flexible in terms of technologies needed, as it is enough having JavaScript. The classical way for implementing the polling is to periodically query the server at a

[1] Any application that allows the server to transmit an event or message to the client is usually said to be implement the Comet paradigm. Every client is referred as a comet.

certain time, usually a few seconds. A more sophisticated approach, known as long polling, consists of keeping the client waiting for the HTTP response until an event happens or a timeout if there is no response. This mechanism significantly reduces the number of requests between the client and the server. Both mechanisms are detailed in Figure 5.4 and Figure 5.5.

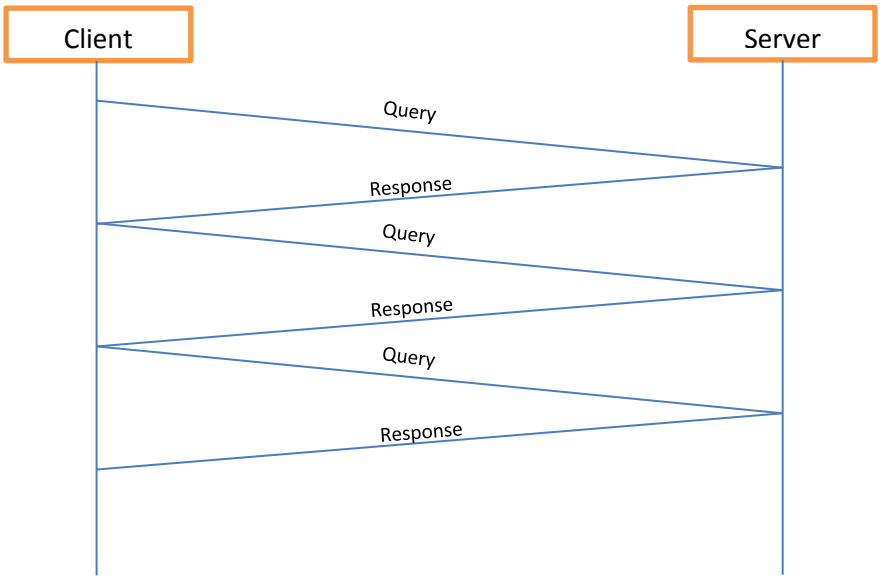


Figure 5.4: Periodic Polling

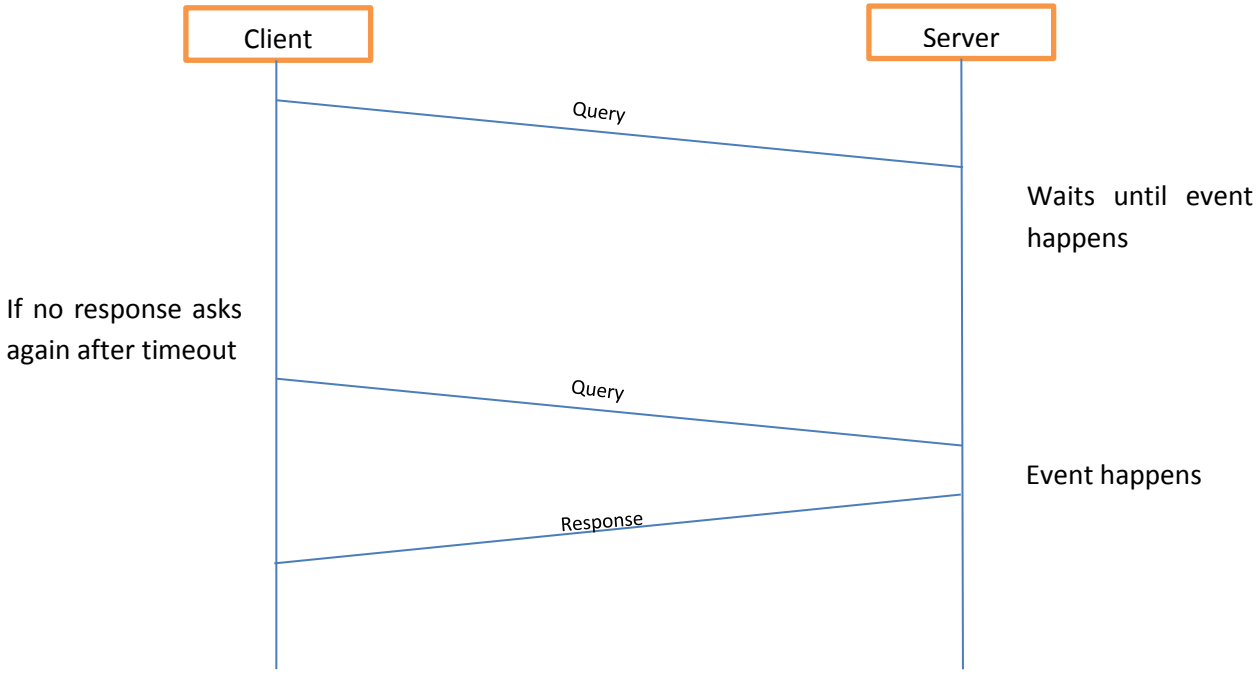


Figure 5.5: Long Polling

The long polling technique is the one implemented here because it fits best with our scenario, with a time between requests of 2 seconds. This is enough to provide a real time impression without overloading the server.

5.2.5 Time-synchronized

This property refers to the ability to ensure everybody is listening to or watching the same moment of a song or video. Our architecture must be able to transmit taking into account video seeks, that is, avoiding the complete stream of a video when a seek happens. Three ways that can provide a selective stream of a video have been studied:

5.2.5.1 RTP

RTP is a standard that describes a way for delivering audio and video. It runs typically over UDP but it can also work on TCP. The main reason for this is that the protocol was designed for live streaming scenarios, which, by nature, are flexible to packet-loss in order to keep the reproduction going, and only UPD provides a multicasting.

RTP is operated using RTCP or RTSP in most cases as well; the former for the data, and the latter handling the control information. These protocols manage synchronization and quality control.

To use this mechanism an additional server is needed. This service should be implemented according to the standards (which are outside the objective of this project) or taken from a third-party library, as there is no Android built-in mechanism.

It is worth mentioning that these communications can be tunnelled on HTTP, as these protocols have evolved to pass through proxies. However, our web player does not support this mode.

5.2.5.2 RTMP

RTMP is a proprietary protocol developed by Adobe, recently made public, to stream between a server and a Flash player. This protocol has several variants and features. The most important one for our purposes is that it has an HTTP variant that would allow us to have a single server. This mode, RTMPT, is supported by Flowplayer as the protocol is one of the most popular for video streaming. The tunnelling makes specific use of HTTP headers so that a specialized server is needed. The main drawback of this approach comes from the need to use an existing implementation of an HTTP server able to handle and serve RTMPT requests.

5.2.5.3 Improved pseudo streaming

I have given the name 'improved pseudo streaming' to my own version of pseudo streaming or progressive download. Pseudo streaming is the technique commonly used by big websites such as YouTube for streaming videos on demand over HTTP. This approach consists of reproducing the file in the player before it has been fully downloaded. The technique can be extended by making the server handling requests for video information from a point in the file not yet downloaded. The server should be able to handle byte-range requests; that is, requests where one of the headers defines a position in bytes. That same information can also be sent as a parameter in the query string named *start*. The former approach is how a HTML5 client will request that part of the video, whereas the latter is how a Flash player will behave.

This will allow a user to get the video reproduced at any position instantaneously. Indeed, it is important to note that the control relies on the client, not on the server. The previous mentioned technologies have control mechanisms and can also make use of video multicasting that allow a precise control from the server of what should be displayed on the client. However, this technique does not provide any control of the server over the clients. To address this problem, the previously explained polling technique is used for synchronizing the cell phone video play and all the clients.

All clients will be informed of the current second being played on the application. They will never have more than a seven-second difference with this value, avoiding unnecessary synchronization due to polling delays or network issues. It means that if the time differs by more than seven seconds they will request from the server the part of video currently being played in the application - unless it is already cached - and the web player will stream that part.

6 THE SERVER

The server is the part of the application in charge of connecting the clients with the application, i.e. delivering what it is on the GUI to all the client browsers. Therefore, it could be seen as a software element with two interfaces: one to communicate with the clients, and another to communicate with the GUI.

6.1 CLIENT INTERFACE

The communication with clients will be done through HTTP, so the purpose of this interface is that of handling requests coming from any client on this protocol. As already discussed (cf. 5.2.2), to achieve adaption and synchronization, the player needs to have certain information from the server, such as the type of file being displayed on the GUI. Therefore, this interface cannot only provide the basic HTTP server functionality, i.e. serving files according given a URL, but also a specific service for providing this information. This section will focus on studying the technique that the server has to implement to provide the pseudo streaming (cf. 5.2.5.3).

6.1.1 The web service

A web service is simply a method of communication over a network. In a classical programming environment, where all our objects are on the same machine, we can use a classical object messaging approach, but in this context the Android GUI and the Player's JavaScript logics are on different machines, so we need a web service to allow them to communicate. There are two main different schemas for building a web service:

6.1.1.1 SOAP

SOAP is a protocol that provides a way for exchanging messages between remote processes. It uses XML for formatting the information, which is divided in a header, containing the method being called, and the body, containing any information needed by that method.

It is used to be the first major protocol after RPC for facing web service scenarios, and it is really powerful in the sense that it provides ways for error handling, message security, and service discovery. This last feature allows the client to discover the method that a service is implementing, which makes it versatile since clients do not need to change if the method's signature changes. Another key feature of this protocol is that it is totally independent of the transport protocol - it can travel over SMTP, HTTP, UDP, TCP and more.

The main drawback of this approach is that the XML envelope payload introduces heavy processing costs.

6.1.1.2 REST

A RESTful service is said to have the REST properties: scalability, performance, security, reliability and extensibility. These properties where defined in parallel with HTTP, so they are both parts of the same idea. REST is not a protocol, but an architectural paradigm that can only be deployed to solve a problem in certain circumstances and by complying with some constraints (Fielding, 2000), being the clearest example of its application the Web. Web services built using this paradigm run over HTTP, but any protocol is valid to build a REST compliant service as long as it follows the URI schema.

Therefore, building a REST service is easy if the scenario allows the use of HTTP. URIs are used to refer to a method and the arguments for it, if needed. For example, we could invoke a method in our server doing 192.168.1.10/mymethod?field1=value1&field2=value2. This would be in the event we were using HTTP GET requests. The same can be done if we call that method but pass the arguments in the body of a POST request. There are different alternatives, but the client must always query the server in the way it is expecting the calls. There is no way provided for discovering the different methods as we are using HTTP, which does not provide this functionality.

This principal disadvantage does not create any problems in our case, as the client – the player - is always delivered from the server. Before it gets to the client browser and starts acting, it is previously hosted in the Android application. In the event we want to extend the functionality of the API, or change the signature of any of its methods, we must change the code on the clients as well for the new release. Scenarios where the clients download a client application may also have to updating them if they modify the API, for which they may want to consider a SOAP service.

To make it simple, since we already have an HTTP server that can be extended to support these methods, and in our scenario it is pointless to deal with the XML overload from SOAP as it would only introduce complexity without any real benefits, the implemented solution is a REST API. This is the same reason that makes REST more popular than SOAP and the almost unique solution on web environments, understanding the term web as that is related to web pages and servers, and generic network applications.

The list of methods is described in Table 6.1 below. It can be seen none of them is expecting argument. This is a read-only service as there are no methods by which the clients can update or delete any information (Richardson, et al., 2007).

Method	Result
getImage	Returns the displayed image or an error message if there is no image if there is no image being displayed
getVideo	Returns the displayed video or an error message if there is no video if there is no video being displayed
getAudio	Returns the displayed audio or an error message if there is no audio being displayed
getType	Returns the type of the file: image, video, audio.
getStatus	Returns the unique id of the file, followed by an ampersand symbol, followed by the current time in millisecond of the reproduction (for videos and audios) or the character 'p' if the reproduction is paused.

Table 6.1: Methods of the service API

6.1.2 Serving Videos

As seen in section 5.2.4, the player can request the video not only from the beginning, but from any point on the file (See Appendix 1 – Basics of video encoding).

When an MP4 video is first requested by an HTML5 player, the metadata is read first. If the user seeks content that it is not loaded in the player, using the metadata it is able to calculate the closest previous keyframe for that time point, and request the file starting at that byte using a byte range request.

Flash players unfortunately do not provide byte range request support, but they implement a similar approach using the *start*[2] parameter in the query string, which for MP4 files will be expressed in seconds. It is not the standard, but most implementations also handle start=0 requests, which demand the keyframes metadata information in case it is not at the beginning of the file (see Appendix One).

```
http://192.168.1.10:8080/video.flv?start=105024
```

Figure 6.1 Start parameter for requesting byte 105024

```
http://192.168.1.10:8080/video.flv?start=0
```

Figure 6.2 Start parameter for requesting the meta information

[2] This is the convention, although it can be changed as long as the player and the server use the same keyword.

Serving a Flash player in the pseudo streaming context is not as easy as for an HTML5 player. A Flash player requires more information from the server in any case, but especially when dealing with MP4 files. For instance, it requires that the moov atom is always sent, and therefore recalculated in the server, for the new block. That is, if a certain offset is requested, the server should send all the keyframes in that block in a new moov atom.

Most commercial server implementations have plug-ins that support MP4 pseudo streaming for HTML5 and Flash video players in case it is not natively implemented. In our case, we are just going to support it for HTML5 player due to the complexity in doing it for Flash. Flash is just the alternative rather than main option in the player. Only a reduced number of clients will require the use of Flash, that unfortunately will miss this feature.

The implication of this is that our server should be able to handle file requests from any given time point.

6.1.3 Handling HTTP

Implementing an HTTP server inside an Android machine is not a normal task. In almost all the literary scenarios, Android devices are the clients. As already discussed in the project proposal (Campoy, 2014) there are some lightweight implementations that are suitable for Android, such as Jetty and Nanohttpd. However, the aim of this project is learn and understand all areas deeply, and there is little that can be gained from using existing software. In addition, having a bespoke server guarantees achieving the maximum lightness as we are only going to have the necessary modules for our application.

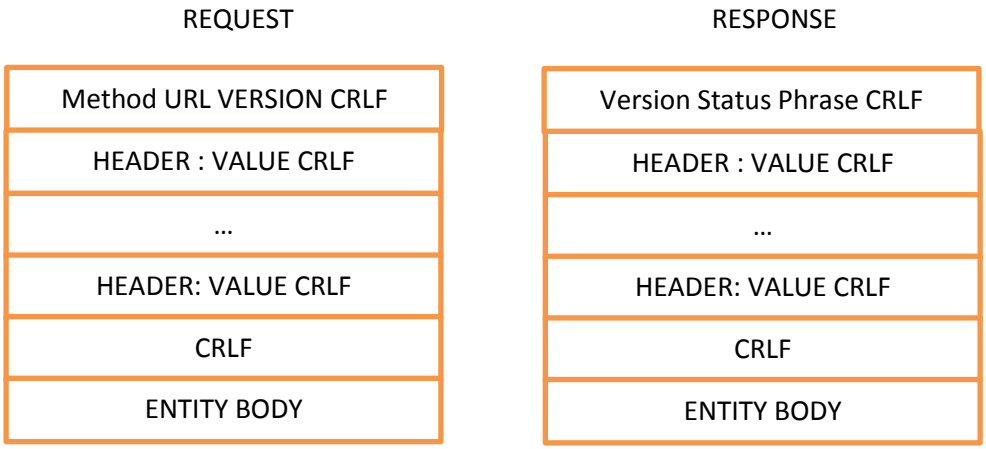


Figure 6.3: HTTP Request and Response Structure

In this project, the Apache HTTP Components (Apache Software Foundation, 2015) library for Java has been used. This tool is not a server but a HTTP wrapper, meaning that from a TCP Socket it is able to wrap all the HTTP information in objects, making it easier to retrieve the body or a particular header. To reply to an HTTP communication the library allows us to easily create a wrapper response object that will be translated later on according to the HTTP standard. This technique therefore

avoids dealing with the protocol at a low level, but the logic for processing and replying requests, i.e. what defines a server, is not provided at all.

The designed server is far from being a generic HTTP server in the sense it does not consider all the possible scenarios that generic servers do. It lacks reusability but provides efficiency. Its features are:

- Support for GET and POST methods
- Implementation of a custom RESTful API (See Restful API, page 26)
- Cookie authentication (See Authentication, page 33)
- Special HTTP request headers for video pseudo streaming support (See Byte-Range requests, page 24)
- Response headers Content-Type, Content-Length and Cache-Control: No Cache
- Error response generation:
    - 401 Unauthorized
    - 503 Service Unavailable

This implies that the methods like PUT or DELETE or the majority of the headers are not going to be processed, like the popular If-Modified-Since. This header might help frequently visited servers with static content, but we have a server that is going to run occasionally and usually serving different content, with a tiny static part. Most of the HTTP headers (Fielding, et al.) are designed for a classical scenario. Skipping them simplifies the response generation process; something we have to bear in mind as it is going to be done on a cell phone's hardware.

6.1.4 Multithreading

As a critical part in the user perception of the quality of the whole application, the server must run smoothly. The paradigm design in the literature (Leondes, 2004 pp. 655-656) is that of using a multithread server so there is a main thread listening to the requests and redirecting them to be answered in independent threads. Thus, requests can be processed in parallel, being able to serve more than one client simultaneously. Graphically this approach is represented in Figure 6.4.

However, we must take into account that threads come at a cost (Goransson, 2014). When a new thread is created, a memory area is allocated for it. The more running threads, the bigger this overhead will be. Our designed thread is composed by a Socket and a Handler references. This is the approach the Apache Library suggests, which basically proposes having a single object, the handler, with a public method to handle a request. Such technique significantly reduces the number of objects and constants in memory, as can be seen in Figure 6.5, considering the size of a handler (Figure 6.6). But even though we have light threads, they do intense I/O operations, which results in heavy processing for the system and high memory costs.

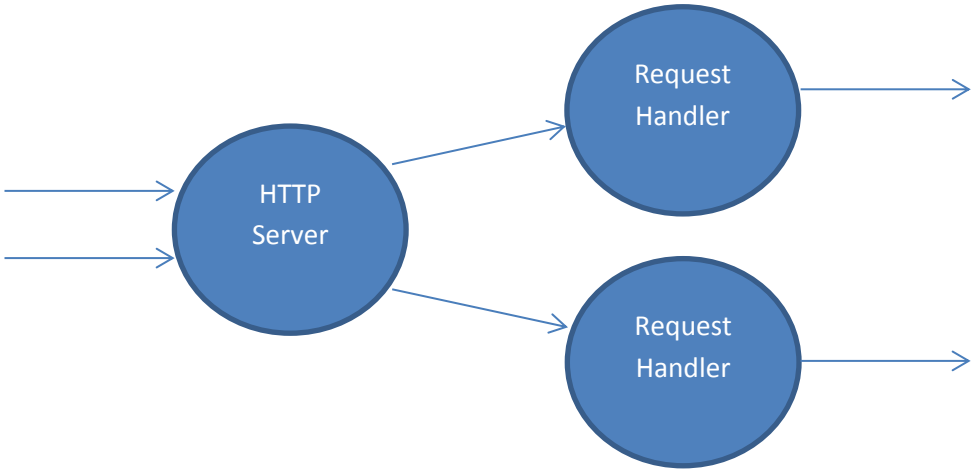


Figure 6.4: Multithread Server

A thread will typically serve any of the static files of the web player or the displayed media file. This information must be read and loaded to memory and then sent through the Socket, blocking the thread while the disk is being accessed and increasing the size of its stack once the information is retrieved. It looks clear that we cannot afford to have too many threads running on a cell phone device doing I/O operations after the point that the average response time goes over the user's acceptability level.

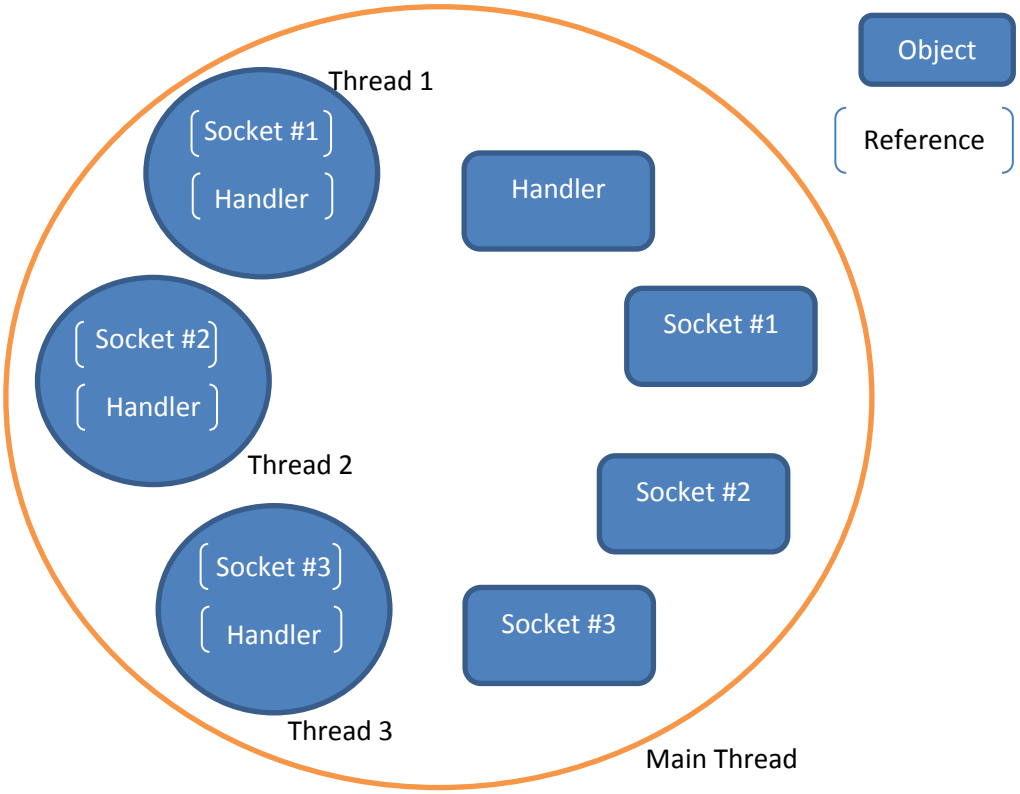


Figure 6.5: Main Thread/Process and its Threads and Objects

To prevent a large number of simultaneous threads we can use a thread pool. This is a design thread technique that is mainly used for scenarios where there is a need for restricting the number of threads or to reuse them in case they are expensive to create. Our pool is going to create threads until the maximum of 6 is reached. It will only create a thread if there is not a free one to reuse, and will delete those that have not been used in the last 60 seconds.

On average, the application will have one client where the user wants to display his media on a bigger screen, or several in the context of a meeting with other people and each of them is a client. In both cases, we will first have a peak until we reach the 6 threads, because all the files of the player will be possibly retrieved by different threads[3]. After that, on average we will need one thread per client to send the video or the images (if they change), and one extra thread to handle the periodicals calls to the RESTful API. Therefore the maximum ideal number of clients will be five. Any number of concurrent connections beyond that is not guaranteed to be served instantaneously. Even a number below five might be poorly served if the cell phone hardware is not powerful enough.

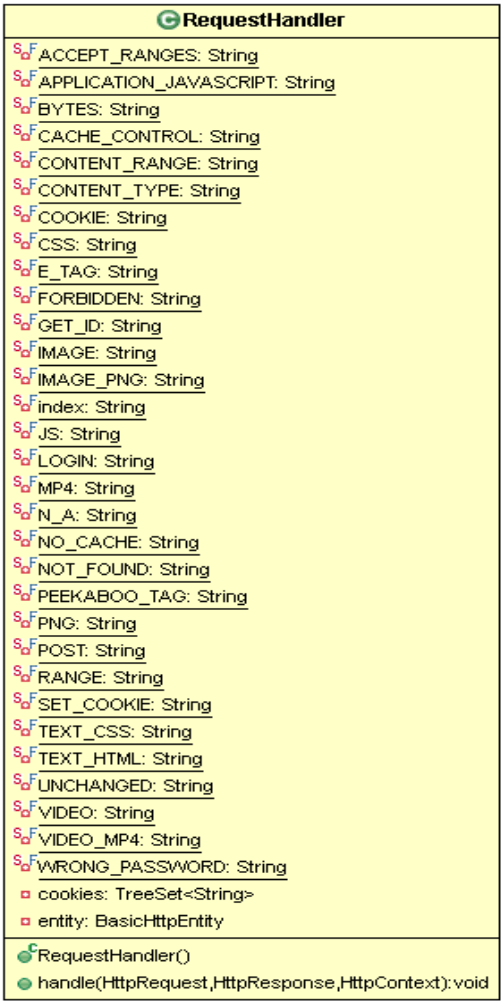


Figure 6.6: Request Handler Inside Details

[3] Even in the single client scenario. A browser will retrieve the index and will get the URLs for all the different files of the player and open several simultaneous connections to retrieve them.

6.2 File System and GUI interface

This is the interface that translates the URI requests from the clients into the right sequence in the application internal file space system or phone gallery.

Hence, we have two scenarios to consider: assets and media files. An asset in the Android context is a raw file of the application, i.e. our server static files, such as player files, JavaScript, Flowplayer libraries etc., and can be accessed through an AssetManager (Android Developers, 2015). Media files displayed on the GUI are retrieved calling the Android gallery. These last ones follow a particular URI notation (see Figure 6.7), that allows to access a File, or an element in a database like a phone number, but that introduces the necessity of using a ContentResolver (Android Developers).

```
content://com.android.providers.media.documents/document/image%3A11
```

Figure 6.7: Android Image URI example

Both the AssetManager and the ContentResolver are classes from the Android package, and dealing with them in the threads might break the independency and modularity design. Although we have already stated that the threads' code cannot be reused for any other application, not isolating them from the Android methods would be bad general practise.

To isolate the threads, an interface between the Android FS and the threads was designed (described in Figure 6.8). This interface is first loaded by the main thread, setting the AssetManager and the ContentResolver. Every time a media file is opened, its URI is also registered in the interface. Once everything is set, it becomes usable by the threads. If a thread receives a regular URI, like *index.html*, it will simply call the interface to resolve that asset. If it has to serve the media file being played it will also make a call to the interface for the information about the file and how to deal with the particularities of the Android File System. Other details useful for pseudo streaming, such as file type and size, or methods for getting the media content from a particular position, are also provided.

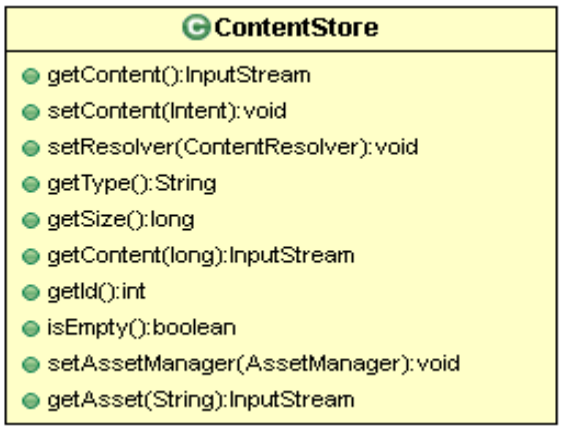


Figure 6.8: ContentStore Interface

7 SECURITY

A crucial aspect that arose when planning the solution was that of security. We are providing a tool for sharing media over a local network, where we usually can assume only trusted devices are interconnected. It can also be presumed that if someone is sharing media, that those files were not intended to be kept secret. Therefore, due to the context of the application, its default behaviour is will be that of not using any security mechanisms.

However, to expand its use to a broader range of situations, a security schema that guarantees to the user that only his trusted clients can access the content has been implemented. In this section we are going to study its key features and details.

7.1 FEATURES

7.1.1 Authentication

This feature refers to the capacity of proving entity's identity. In our particular context this can be translated into the ability of establishing that a given client forms part of the user's group of trusted clients. Because we do not need to know the individual identity but a group identity, having just a shared secret or password, without usernames, is sufficient.

This mechanism has a number of vulnerabilities (Forouzan, 2007 pp. 417-419), all of them well known, such as susceptibility to password stealing or dictionary attacks. Possibly more sophisticated mechanisms could be implemented, like IP or even MAC validation, but there should be a compromise between complexity and risk prevention. In my opinion password authentication is the best approach.

For its implementation, a field in the application preferences has been created (page 14). From there, the user can set the password they want and can share with his trusted clients. If a password is set, the application will enforce the user to enter the password. Only if both the password entered and the one set on the application match will the authentication be successful. Because our clients are going to retrieve more than one file from our server, the subsequent requests after a successful authentication must avoid the requirement of having to enter the password again.

To achieve this, a cookie-based authentication mechanism that follows that of the password has been implemented. Each client gets a cookie that lasts until the application is closed or the password is changed, thus guaranteeing the authentication of a user for a certain sharing session or presentation.

7.1.2 Authorization

This feature refers to the capacity for determining the permission of a certain user. In our scenario the authorization policies are very simple: non-authenticated clients are not authorized to access any content other than the password form, whereas authorized clients will be able to request all the content. There will be no subcategories as these are not required in our context. Thus, authorized and authenticated are synonyms and have no functional difference.

The server will reject any connection not attempting to get the form and display a 401 Unauthorized HTTP message.

7.1.3 Confidentiality

This feature refers to the capacity for preventing information being disclosed to unauthorized clients. Thus, we should not only focus on the information access as we have done so far but in its transmission as well.

Even though we have prevented non-trusted clients from retrieve any media content from the server, this does not implies that they cannot access it by 'sniffing' the local network. What would be trivial if they have access to the router or any hub on the chain from the server to a trusted client. Such a hack would allow the attacker to read all the information or even obtain a valid cookie to cheat the server.

To provide this feature the server has been extended to implement HTTPS.

7.2 HTTPS

HTTP Secure, commonly referred as HTTPS, is simple HTTP sent over the Transport Layer Security (TLS) protocol. This protocol is able to exchange a ciphering key using asymmetric cryptography that is only known to the client and the server. Although the study of details of this protocol is not an aim of this project, we will analyse some facts that are relevant for us.

7.2.1 Ciphering algorithm

As it has already been stated, TLS uses an asymmetric cryptographic mechanism to securely exchange another key that temporarily ciphers that session. The main reason for not continuing to use the asymmetric encryption for the whole session relates to the fact that asymmetric algorithms are much more CPU consuming (Forouzan, 2007 p. 296) as they are based on complex mathematics models (trapdoor functions and elliptic curve) whereas symmetric algorithms usually perform less demanding confusion and diffusion operations.

As asymmetric encryption can be achieved by Diffie-Hellman key exchange, which allows the secure generation of a shared secret key over an insecure channel, my first attempt was to use this method. Unfortunately, this mechanism does not prevent a 'man-in-the-middle' attack as it does not authenticate the counterparty in the way that Public Key Infrastructure does; that is, the well-known X.509 Certificates. Therefore, popular browsers do not support its use (Firefox Archive, 2006). This introduces the necessity of having to create our own certificate.

There are many different algorithms that work with certificates. The main method for classifying them is to check whether or not they support Perfect Forward Secrecy; that is, whether the communication is still confidential even in the event that an attacker has access to the certificate's private key. An interesting study shows that most browsers generally do not use these cipher suites, and some do not even support them (Netcraft, 2013). This could be dangerous if we are supplying a certificate inside an application that everyone can hold of.

In the recent version of Java SE 8 , a new method has been introduced to specify the preferred order for using a cipher suite (Java 8 setUseCipherSuitesOrder); however Android's Java version and Java SE are not the same thing and this method is not available yet in the former one. In the event that it was, its use would restrict our application to updated phones and would still not guarantee security if the client's browser refused to use these algorithms.

What we can do with our current Java version is to restrict the list of valid suites to those that support PFS. However, this will reduce of number of potential clients as it is not supported in every browser.

7.2.2 The Certificate

The other key aspects that we need to consider are the certificates. As we have already stated, TLS is able to securely exchange a key without a certificate, but that will not provide any authentication. Thus, it is used in conjunction with the PKI and X.509 certificates in most of its implementations, and all those that are applied to browsers. It is important to add that this is not the only means of authentication the standard provides, as it can be used in conjunction with Kerberos (RFC 2712, 1999) or a pre-shared key (RFC 4279, 2005).

To obtain a valid X.509 certificate for the PKI, a supported Certification Authority for most browsers, e.g. Verisign or GoDaddy, must be issued and signed over to us. This costs money, and also incurs the risk of storing our certificate in an application than can be downloaded by anyone who compromises the security of its private key. In addition, when checking the validity of a certificate, browsers do not only check if it has been signed by a trusted CA but also if it has a valid date, that it has not been revoked, and especially if the credentials on the certificate match those of the entity presenting it. That is, a certificate will usually be issued for a specific Common Name (CN) which in the case of the web has to match the host of the address. Recall that in our case the host address is going to be an IP address, and, therefore, has potentially many possible values. This last constraint is the confirmation that there is no scenario where we could get a valid certificate that any browser permits.

The solution to this problem is dynamically generating a self-signed certificate. We will issue it for the correspondent CN and the private key will not be compromised, as it will last only for that session. If an attacker runs the application the private key will not be the same as the previous session.

This approach solves the issue we faced about PFS on page 35. Because we are deleting the certificate and its keys after each session, the scenario where a previous communication can be decrypted for a future key is removed. Therefore, security can only be compromised where an attacker is able to hack the phone and access its key while it is still transmitting and given that for

that session the browser and the server did not negotiate a PFS compliant algorithm. Previous communications from earlier (and subsequent) sessions will still be PFS if the attack succeeds.

Besides the good level of security achieved with this schema, not having a certificate issued by a trusted CA will result in a warning in most commercial browsers, thus discouraging access to the page. This is because according to the PKI model our server cannot be authenticated. But from a technical point of view we are providing authenticity the moment the user tells his clients his IP address. Therefore, this will prevent a man-in-the-middle attack if someone tries to emulate the server to get the password since clients know the address to authenticate and enter the password.

7.3 PERFORMANCE

Securing our application with the discussed mechanisms will result in an overload of our software, and we should always keep in mind the limited processing capacity of our potential clients, i.e. cell phones. Moving from HTTP to HTTPS will increase the overhead of the communication and its computation (Iyer, et al., 2005). The two main factors for this increase are the SSL handshake, where the keys and algorithms are negotiated, and the actual ciphering process that comes after that. In their article, Iyer et al. showed that the 70% of the time spent for small transactions, i.e. those ones below 32KB, is in the handshake. As the transactions get larger, the bottlenecks moves towards the encryption process.

File	Size	HTTP	HTTPS	Increase
API Message	175 B	61 ms	609ms	898%
Index	6.7 KB	90 ms	542ms	502%
Picture	763 KB	1.68 s	4.14s	146%
Video	11 M	1.9 min	3.3 min	73%

Table 7.1: HTTP vs HTTPS

Table 7.1 contains a comparison for the most common elements the clients and the server exchange. This study has been done emulating a Nexus One. The results show the penalty of using the security, which comes to a critical point for the API calls, something that happens every two seconds, and therefore, a big percentage of the total number of requests (recall long polling, page 21). But despite the huge overload, the response time is still below the users' tolerance to delay as indexes or API calls are served in under a second.

8 Software Engineering

8.1 Life Cycle

Peekaboo! has been developed under an incremental prototype life cycle. The decision was to implement a basic model and systematically build the rest of the requirements. The initial circumstances, where there was only a theoretical proposal based on open technologies deployed over Android, suited this model. Building a prototype which incrementally controls the viability of the different features in terms of technical feasibility and time was seen as the best solution. Finally, all the desired features have been implemented but it was not initially possible to determine the final status of the software.

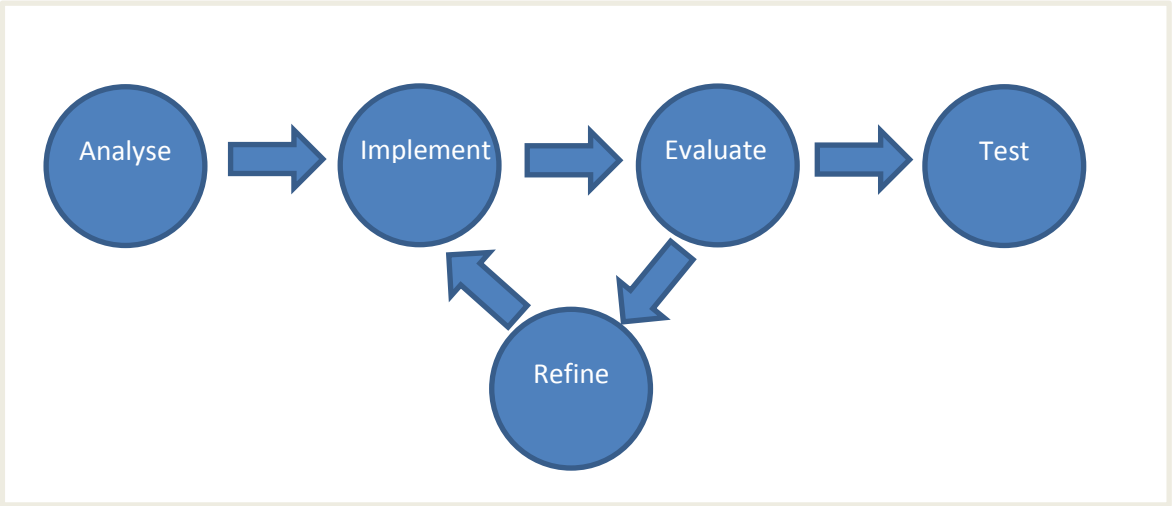


Figure 8.1: Development Cycle

There have been a number of refinements cycles:

- **Version 0.0**: initial prototype based on a basic interface and running a basic implementation of an HTTP server. This first prototype was useful to test that an HTTP server running in the background of an Android application was indeed reachable over the network.
- **Version 0.1**: the prototype is extended to open images from the gallery. When the server was queried over the network the displayed picture was served.
- **Version 0.2**: the content is now wrapped on a basic HTML website, the player. The RESTful design is implemented and the player can request the picture calling the RESTful API using JavaScript. The server is extended to serve queries requesting files like HTML, CSS, or any other content and answer with different HTTP statuses.
- **Version 0.2.1**: the website JavaScript functionality and the RESTful API are extended. The long polling mechanism is implemented and now if the picture is changed on the GUI it gets reflected on the website.
- **Version 0.3**: additional functionality is added to open and display videos in the GUI. Flowplayer is integrated in the player and the API extended. The videos can now be visualized on the HTML.

- **Version 0.3.1**: the website's JavaScript is extended along with the RESTful API for synchronizing the time in video reproduction. The server is also extended to support byte-range requests to serve partial content.
- **Version 0.4**: Security is implanted. The server now implements HTTPS and cookie control, and generates a new certificate depending on the current IP address. The GUI is extended to define some settings including a password and HTTP and HTTPS ports. The player now has a log-in page that will be displayed by the server if the clients are not signed in.
- **Version 0.5**: The player and the GUI are modified to handle MP3. No need to modify the server.
- **Version 1.0**: The presentation of the player is polished and the Supersized module included. This is the final version of the software.

8.2 Design Patterns

8.2.1 Facade

This pattern is meant to provide a unified access to a set of interfaces in the system. It can be used to solve different issues, but, in our scenario, it has been used to decouple the system. An example of this use can be found in the way that the GUI and server communicate, illustrated in Figure 8.2. As it can be observed the ContentStore class decouples the server from the graphic interface and from the file system.

In the event that the server requires information from the file system or the displayed media element on the GUI, a request to the ContentStore will be sufficient to get that information. This both hides and eliminates the complexity between the different blocks. Avoiding having strong dependencies between the elements warranties that any modification on the server will not affect the GUI, and the other way around. Similarly, if in the future the application files' structure wants to be changed or new functionality added for accessing files, the server will not require any modification.

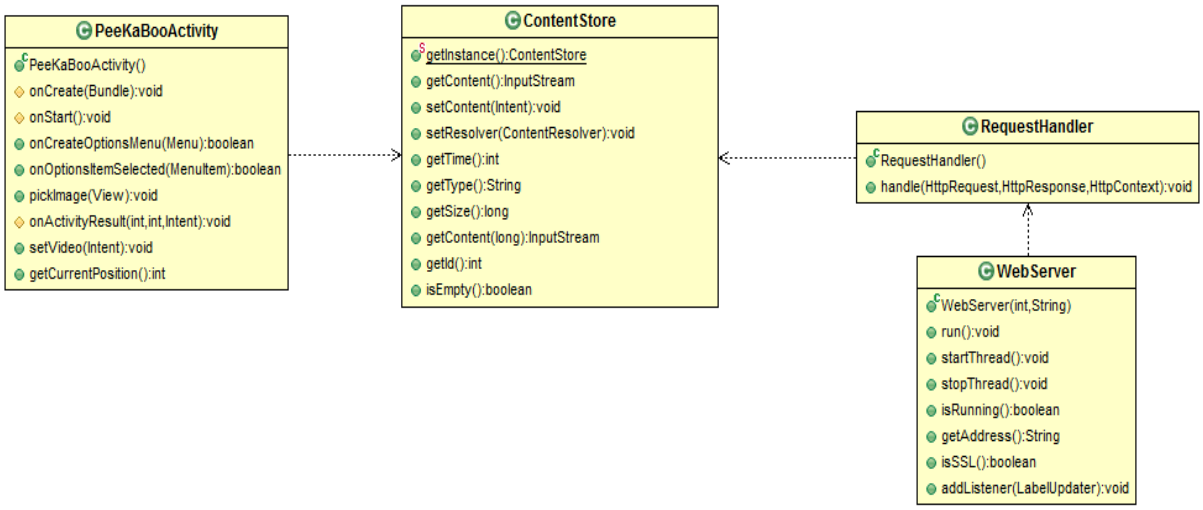


Figure 8.2: Facade Pattern Implemented in Classes ContentStore and LabelUpdater

8.2.2 Singleton

The singleton pattern when applied on a given class restricts it to be only instantiated in one object. This is useful when an object has to do a central job and we want to ensure that all its clients point to the correct one, which is indeed the case for the ContentStore. For this class, we have to warrant that both the server and the GUI refer to the same instance; otherwise they will not really be interconnected.

As it can be seen in Figure 8.2 the static getInstance method is the only way for creating and retrieving a ContentStore, and will guarantee that only one instance is created.

8.2.3 Thread Pool

The thread pool pattern is commonly used in scenarios where there is a list of tasks and a need for controlling the number of current threads or reducing the impact of its creation. This pattern usually provides different benefits depending on its use. For our particular case this pattern is going to benefit our server in two ways as discussed on page 29:

- Control the number of threads: we want to restrict the number of concurrent threads due to the limitations of the hardware.
- Thread reutilization: creating a thread is expensive and with this technique threads are reused so we can minimize the impact of created by removing them, which is a common operation in a server as each request is treated independently and each element and API call of the player will be requested individually from the browsers.

8.3 TESTING

The testing process of the software has been done emulating a Nexus One through the iterative *Evaluate* steps (c.f. Figure 8.1). This local device has helped to test the incremental versions. Once the development was completed the test was carried on in a real LG E610. Because this consists of checking the usability of an Android application, it has been documented in a video. This video can be found in http://luiscampoy.es/peekaboo.mp4 and on the CD attached with this documentation. On it, PeeKaBoo! is tested serving four clients concurrently: one TV, one cell phone and two computers. Figure 8.3 and Figure 8.4 show a sample of it.

Figure 8.3 Real picture sharing

Figure 8.4 Real video sharing

9 FUTURE WORK

This project is open to a good number of possible developments that will boost the usability of the tool. Some of the proposals that I have are as follows:

- **Make it global**: The current design described in this document allows the PeeKaBoo! to work only on a local network. To make it global a centralized service or any other suitable software for this task as described in page 11 would required to be implemented.
- **Use an existing server solution**: an open source or commercial HTTP/S service implementation could be integrated to replace the current one. It will have the constraints of not being specific and therefore necessarily efficient, but it is always a good practice from an engineering point of view to reutilize existing software instead of building new one. Implementing an HTTP service is a recursive problem and the best approach will be taking a working implementation as it will have had stability testing by a larger number of developers that has been focused on just that issue.
- **Extend the range of file types**: PeeKaBoo! could be extended to support a new types of files. Perhaps the most interesting would be PDF or any other kind of document files. The approach to implement this new functionality would be similar to the that followed in the case of videos. An existing web document viewer should be integrated into the player, and any action occurring in the GUI, e.g. a movement from page, notified to it. This could be done implementing new functions in the API or extending the functionality of the existing ones.
- **Implement Flash pseudo streaming support**: as discussed in this work, serving partial content of MP4 files for Flash players requires following a special convention which is not currently supported.
- **Commands from the Player**: the player could be extended to have more functionality apart from that of displaying the media files. Examples are selecting the files from the Android Gallery of the player or perhaps sending a message to the phone in case some requirement needs to be transmitted to the presenter. There are many things that can be added to the player.
- **Easy sharing:** currently the users themselves need to publish the URL. Even though they are able to copy it, it will be interesting to include a sharing link that automatically shares this information through other applications like Whatsapp or email. This will ease the interconnection as both the user and his clients will be able to do the whole process through clicks, without explicitly dealing with IP-based URLs.
- **Statistics and Log**: my final suggestion is to implement some log functionality in terms of files displayed, number of connections, users' devices, etc. This would allow usage perspective to be built; for example, most displayed files , files displayed per session, average session times, and average number of users, users' devices types and many more. This information may be useful for the users running the application but also for maintenance and tool knowledge set. A central service could be implemented to gather anonymous information from all users which have installed PeeKaBoo! to provide internal statistics that help maintain and improve the tool.

10 CONCLUSIONS

In this last section of the document we are going to take a look at those elements that have to do with my subjective experience and have been omitted in the previous chapters.

10.1 BIGGEST COMPLEXITIES

10.1.1 Serving parts of files

Implementing the pseudo streaming in the server side turned out as a really complex task. Basically, the server should be able to return the video file from any byte the clients' request. Altering the file buffers and especially dealing with the complexity of the HTTP protocol were the key issues. There is no information about how to implement this feature to be able to interconnect with other devices, i.e. there is no standard definition for the byte-range request implementation. So the way I followed was that of analysing the traffic between my browser and public web services. From that I was able to find the core number of headers and statuses belonging to this mechanism.

10.1.2 Emulator network testing

The android emulator provided for IDE software was mainly conceived for testing the functionality of the applications. This is usually local. For this scenario I had to be able to communicate my local web browser with the application running in the emulator. Because the emulator simulates its own private network the solution was to do some NAT between my computer's network and the former one. Finding out how to do this and the rationale of what was happening behind the scenes required its own learning process.

10.1.3 On the fly self-signed certificate generation

Unfortunately the support from the Java community to generate self-signed certificates is limited. It is not a common task as they are mainly generated outside the application and then used by it (using any software like OpenSSL). In addition, once the certificate is generated it cannot be used straight away as it has to be stored in a keystore and then retrieved from there at the time of generating the secured socket. Thus, the process does not only involve its creation but also the management of a secured keystore that has to keep the certificate on disk, with the additional complexity that entails. This store is encrypted and saved in a temporary directory of the phone. As a clarification, this does not compromise the security as each new time the application is run a new certificate will be generated, changing both the private and public key (c.f. page 33).

10.2 LESSONS LEARNT

10.2.1 Android development

I had some minor experience with the Android programming but this project has given me opportunity to get a deeper insight into it and to manage different kind of containers and grids. I have also learnt how the activity lifecycle has been designed and how to implement the functionality for each status.

10.2.2 Video Encoding

I have also learnt how the MP4 container is defined in its different variants and how that has an impact on the video streaming. To understand this I had first to explore the general aspects of video encoding as that was a field I had not studied before. After doing this project I have gained substantial knowledge about what the goal of the media containers is and the differences between their compressions. It is true that when it comes to cover the general aspects the literature is quite detailed and easy to find. However, it was rather difficult to find good text that could link encoding with video streaming.

10.2.3 Byte-range requests

As a consequence of dealing with video parts and implementing a way to serve them, I have learnt how to construct such a mechanism in a way that is able to interconnect with any browser. There are specific modules to install this feature in the popular HTTP server implementations but there is no documentation that actually explains what the browsers are looking for. Finding out how to do it has not only taught me how to do it but also that the standard documentation of the protocol is not only what the browsers are expecting. That was only valid for Firefox, the rest of browsers required some more headers.

10.3 SELF-ASSESSMENT

The final result of this project has fulfilled all my expectations. As I mentioned elsewhere, this idea came to my mind when trying to find a quick and safe way to display a picture in a TV and it has now become a real software application. Nowadays devices like Chromecast and AppleTV are becoming reasonably popular and present a way to overcome this issue. Nonetheless, while they do it at OS level using physical connections with the TVs, PeeKaBoo! presents an alternative way of creating a sharing structure with the only requirement of being connected to the same network. This is also a requirement for the previously mentioned devices but the difference is that the latter one has the extra feature of serving more than one client (e.g. TV). In addition, it does not require plugging any other device.

Despite the fact that those two popular hardwares offer a wider range of features, the key idea here is the design of a different mechanism which, in the future, may have a wider scope and further implications. Besides, the application fulfils all the requirements defined in (Campoy, 2014) and as shown in page 40 it has been proved to work fairly well with four simultaneous clients.

To conclude, I would like to recognize the positive benefits for the student of implementing a whole project on its own in which they can freely take the key design decisions. I have truly learnt from this experience.

APPENDIX 1- BASICS OF VIDEO ENCODING

OVERVIEW

Videos are defined using a video container. This is a formatted description of the data and metadata for the video. The data information is the several streams that are joined on a video, e.g. image, sound, subtitle, whereas the metadata contains information about the synchronization of these stream, used codecs or video compression information.

Videos might be compressed using different mechanisms, but there is always some kind of compression. All the different frames a video is composed of are never stored as they are, but compressed making use of the information from the previous frames. This is usually referred to as the video encoding.

This is the basic approach among the different encodings, although there are important differences between them that may lead into different compression capabilities (a level of detail that we will not examine). What it is crucial to understand is that all the techniques have what it is known as keyframes where the complete image is stored. Classic containers like AVI do not provide access to future keyframes, and the only way for knowing where they are is to have the whole file already loaded, whereas encoding standards like H.263 or H.264 (best known by some of their commercial variations like FLV and MP4 respectively) Matroska (MKV), WebM or Ogg have managed to solve this issue. But the majority of the web players just have pseudo streaming support for FLV and MP4 since these are the containers for which popular HTTP server implementations provide this technique, as the server also plays a role in this mechanism. Hence these will be the formats we will be considering.

Both MP4 and FLV are quite similar, FLV was created exclusively for Flash players whereas MP4 is becoming the standard for streaming in HTML5 players as it has a good compression and is supported by most devices, which is crucial in the era of mobile communications and devices. Because Android devices do not support FLV, we are just going to consider the encoding for H.264.

H.264 ENCODING

MP4 files are divided into atoms, the more important one for us being Moov. Among other things, this atom contains pointers to all the key frames in the data part of the file. When a video file is created it is usually added at the end because it is when all the keyframe positions are known; however, the way the standard was designed also allows adding it at the beginning as the atoms are independent blocks (an example can be found in Figure A.1). There are tools that move this meta-information to the beginning, but new recorders are by default adding it first as well.

It is crucial to understand that pseudo streaming will only work if the Moov atom is downloaded before the MDAT. Most videos will be structured this way, but some servers implement their own technique to serve it first, and, also, some browser implementations might request the moov atom block in a parallel second connection if it is at the end.

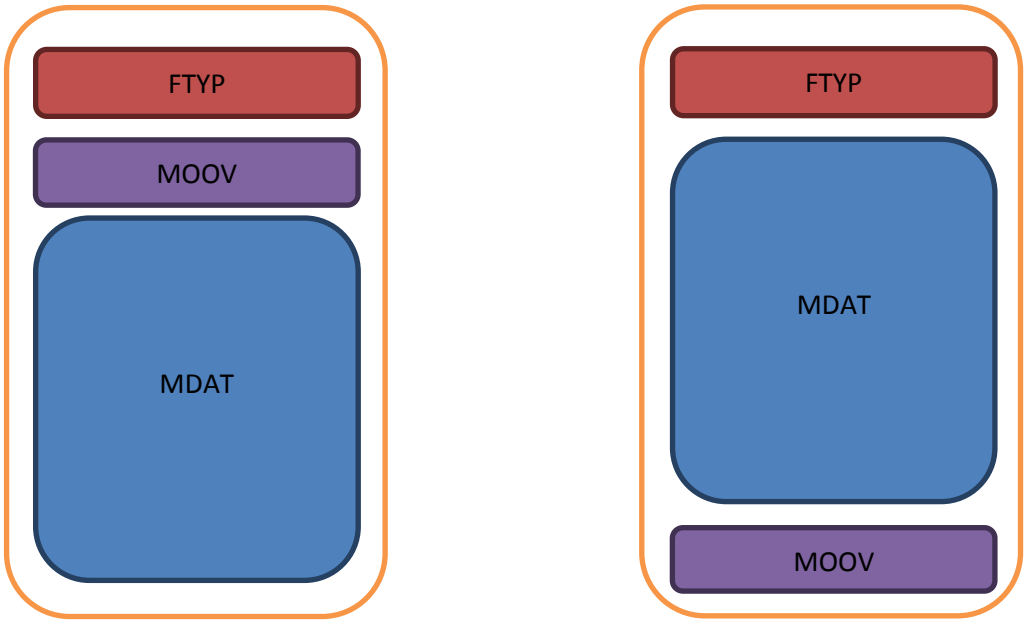


Figure A.1: Figure 11 MP4 Containers (A) Moov atom before data. (B) Moov atom after data

BIBLIOGRAPHY

Adobe's Real Time Messaging Protocol. **Adobe Systems Incorporated.** 2012.

Android Developers. 2015. AssetManager. [Online] 2015. http://developer.android.com/reference/android/content/res/AssetManager.html.

—. ContentResolver. [Online] http://developer.android.com/reference/android/content/ContentResolver.html.

Apache Software Foundation. 2015. Apache HttpComponents. [Online] 2015 https://hc.apache.org/index.html.

Build Internet. 2012. *Supersized.* [Online] 2012. http://buildinternet.com/project/supersized/.

Campoy, Luis. 2014. *An implementation of an Android OS media sharing application using open protocols.* London : s.n., 2014.

Fielding, et al. Header Field Definitions. [Online] http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html.

Fielding, Roy Thomas. 2000. Chapter 5: Representational State Transfer (REST). *Architectural Styles and the Design of Network-based Software Architectures.* s.l. : University of California, 2000.

Firefox Archive. 2006. *Are DH and anonymous DH cipher suites implemented?* [Online] 2006. http://www.mail-archive.com/mozilla-crypto@mozilla.org/msg08353.html.

Flowplayer. 2015. Flowplayer - The video player for the web. [Online] 2015. https://flowplayer.org/.

Forouzan. 2007. Cryptography and Network Security. s.l. : McGraw-Hill, 2007, pp. 417-419.

Goransson, Anders. 2014. *Efficient Android Threading: Asynchronous Processing Techniques for Android.* s.l. : O'REILLY, 2014.

Iyer, Ravi, et al. 2005. Anatomy and Performance of SSL Processing. [Online] 2005. http://www.cs.ucr.edu/~bhuyan/papers/ssl.pdf.

Java 8 setUseCipherSuitesOrder. Java Secure Socket Extension (JSSE) Reference Guide. [Online] http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html.

Leondes, Cornelius T. 2004. *Database and Data Communication Network Systems.* 2004.

Lewerentz, Sigurd and Noel, Thomas. 2002. *Wireless Mobile Phone Access to the Internet (Innovative Technology: Information Systems and Networks) .* s.l. : Hermes Penton Science, 2002.

Netcraft. 2013. SSL: Intercepted today, decrypted tomorrow. [Online] September 2013. http://news.netcraft.com/archives/2013/06/25/ssl-intercepted-today-decrypted-tomorrow.html.

New Directions in Cryptography. **Whiitfield, Diffie and Martin, Hellman E. 1976.** s.l. : IEEE, 1976.

RFC 2712. 1999. Addition of Kerberos Cipher Suites to Transport Layer Security (TLS). [Online] October 1999. http://tools.ietf.org/html/rfc2712.

RFC 4279. 2005. Pre-Shared Key Ciphersuites for Transport Layer Security (TLS). [Online] December 2005. http://tools.ietf.org/html/rfc4279.

Richardson, Leonard and Ruby, Sam. 2007. Chapter 5 - Designing Read-Only Resource-Oriented Service. *RESTful Web Services.* s.l. : O'REILLY, 2007.

RTP: A Transport Protocol for Real-Time Applications. **Network Working Group.** 2003.