Delivery number 3 (Draft Version)

Lab #7 - Delivery number 3 (Draft Version)

A Server to Send Video over IP/UDP Streams to Clients

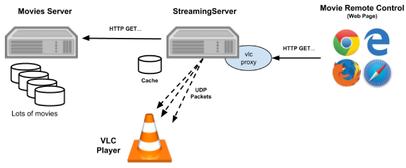
Motivation

Live or stored multimedia content is often delivered over the Internet. One way to achieve this is to use streams of UDP packets containing encoded video frames. In this case, the player must be able to show the content to be played, sent correctly by the server, in order to allow for a good experience to the user.

Goal

The goal of this assignment is to implement a video delivery infrastructure over the Internet. This will be achieved by repurposing work done on the previous delivery and the latest exercise on multicast streaming to glue together the necessary components in a integrated solution.

The figure below illustrates the intended service architecture.



Architecture Components

- **Player: VLC**

  VLC will serve as the media player for the user. It will listen for UDP packets with the media content in some IP+port location.
- **Movies Server**

  A standard HTTP server will host the movie files.
- **Movie Remote Control (Web Page Client)**

  This is a web page that will allow the user to control the streaming of movies.
- **UDP StreamingServer**

  This is the only component that needs to be developed and delivered.

  This server provides an HTTP interface to accept streaming requests from **Client** (**via vlcproxy**). To that end, it will have to fetch the movie contents from the Movies Server (or local cache) and send the stream (unicast or multicast UDP packets) to the **Player**.
- **VLC Proxy -** performs HTTP requests to UDP StreamingServer (and to VLC Player) on behalf of the **Movie Remote Control (Web Page Client)**

Detailed Specification

UDP StreamingServer

A movie playback request consists of a **HTTP GET** request. The **StreamingServer** must parse the request and obtain the movie contents either from local storage (cache) or from the remote **Movies Server**. Once the movie contents is available, the StreamingServer must stream it, as a sequence of UDP packets, to the **Player.** The address and port of the **Player** is part of the HTTP request. The movies will be encoded using the format already used in Lab #7, detailed in the next section of this document.

Basic Solution:

StreamingServer receives HTTP requests formatted as follows**:**

- **GET** /monsters.dat?ip=224.0.0.1&port=1234 **HTTP/1.0**

  In this case, the movie "**monsters.dat**" is to be retrieved from **local storage** and streamed to the **<224.0.0.1:1234>** UDP endpoint, where VLC will be expecting the stream.

Advanced Solution (Optional):

StreamingServer receives HTTP requests formatted as follows**:**

- **GET** http://<ip:port>/<path>/monsters.dat?ip=224.0.0.1&port=1234 **HTTP/1.0**

In this case, the movie **"monsters.dat"** must be retrieved from the Movies Server pointed by http://**<ip:port>/path/...** and streamed to the **<224.0.0.1:1234>** UDP endpoint, where VLC will be expecting. **Note,** if the movie is already present in the local cache of the streaming server, the cache will be used to serve the request.

Here's an actual example:

GET http://asc.di.fct.unl.pt:80/rc/movies/monsters.dat?ip=224.0.0.1&port=1234 **HTTP/1.0**

Running

Your **StreamingServer** should be started in the following way:

java -cp ..vlcproxy.jar StreamingServer port

Where **port** is the port where your streaming server accepts the requests as specified above. **vlcproxy.jar** is a library that provides some hidden code to control the VLC player from **Movie Remote Control** web page. You should also add it to the **Java Build Path** of your Eclipse project.

VLC Player - In order to allow the **Movie Remote Control** web page to control the **VLC Player**, (1) you must enable in VLC the (LUA) Web interface; (2) set a password; (3) make sure port tcp 8080 is free for VLC.

Media Contents Format

Files containing video streams are encoded according to the same rules as the ones used in Lab #7 (UDP and MULTICAST Streaming): a sequence of records. Each record is composed by the number of bytes (a **short**) of the payload encoding the frame contents; followed by the timestamp of the frame (a **long**), relative to the first frame (whose timestamp is not 0); followed by the payload (a **byte array** of the given size) containing the encoded frame to be streamed to the player.

n (short)	Timestamp (long)	Payload - movie frame (byte[n])

Minimal and Optative Goals of Your Work

Your simplest server should be able to serve player requests of movies available in its local directory, iteratively, i.e. playing the next movie after having fully streamed the previous one.

Optionally, your server should be able to stream several different streams in parallel, to play movies from the local disk or from the movies server, to cache locally movies got from the movies server, etc.

Smart management of the cache: assuming the cache is limited in size, or any other smart caching features you invent, will also be considered as an extra added value of your solution.

If you are up to the challenge, you can also extend the specification and the player to also consider the case where a certain requested content is being already multicasted towards a certain IP group multicast and port accessible to the player, like when you have a box playing live TV channels broadcasted over UDP packet streams.

Your server program must also obey a certain skeleton, see the last section.

Grading (in 0-20 scale)

- A **StreamingServer** that is capable of receiving the **Client** request and send it the requested stream while reading the video file from the local disk, will be graded at most 11/20. (This is the context of the **Basic Solution** as described above)
- A **StreamingServer** that is capable of receiving the **Client** request and send it the stream, reading it from the local disk (cached movies), or if not available, obtaining it from the HTTP movie server, will be graded at most 14/20. (This is the context of the **Advanced Solution**, as described above.)

Other Improvements (on top of the Advanced Solution)

- The **StreamingServer** can serve several clients in parallel, playing the same or different movies. This will be graded an additional 1/20 points.
- The **StreamingServer** implements a cache. If a movie is not found in the cache it is fetched from the server and stored in the cache. Subsequent request will be served off the cache. This will be graded an additional 2/20 points.
- Challenges for an Ultimate Excellent Solution - the **StreamingServer** could have:
  - Smart Cache Management (limited size, eviction policies, etc.)
  - Robustness
  - Impeccable coding quality
  - Any other feature you can argument as a differentiation factor of your own solution

  Up to 3/20 points are reserved for this category.

Work Delivery

Via this Google Form.

Look at an example of a submited form.

Rules:

- **Development**: Groups of 1 or 2 students.
- **INDIVIDUAL DELIVERY:** Each student must deliver his/her own copy of the work and results, using the supplied Google form.
- **DEMO:** A demo of the work may be requested by the instructors. This will take place before the second frequency test.
- **DEADLINES:**
  - **December, 7, at 23:59.**
  - **December, 9, at 23:59. Late deliveries will be penalized 1/20 points per day.**

Additional materials and suggestions

All the required materials (movie files and player program/components) are available in this folder. You have the movie monsters.dat to start your developments but more movie files can be added afterwards. These files will have always the extension ".dat".

Provisional: File / script Player is the player. However, for the basic solution you can provisionally use a browser to make the request to your Streaming Server and running VLC by hand in the expected endpoint <ip address, port>.

Provisional: a **Movie Server** as a standard HTTP server is available in: http://asc.di.fct.unl.pt/rc/movies (the monsters.dat movie is available in this URL). However, provisionally you can use a local HTTP server (from previous classes) if you want to have the Movies Server running in your development computer. It should be run in a different directory then your **StreamingServer**, otherwise all movies will be local to your server.

File **Http.java** contains methods to parse HTTP requests/responses.

File **StreamingServer.java** is a skeleton of your solution. If you only deliver a simple solution (basic + caching + parallelism) you do not need to modify it. This will help instructors during the evaluation process. More improved solutions may need to change it deeply.

A sketch of its content follows:

```
import java.net.*;

public class StreamingServer {

  // receives and handles a client request
  void handlePlayerRequest(Socket s) {...}


// sends an stream of movie to the UDP socket(s) at ip+port
// it assumes that movie.dat exists in the local file system
  void sendStream(String movie, String ip, int port ) {...}


// obtains movie from an HTTP server at ip+port
  void getMovie(String movie, String ip, int port ) {...}

  public static void main(String[] args) throws Exception {
      int port = ...

      proxy.VlcProxy.start( port ); //entry point of the vlcproxy.jar

      try(ServerSocket ss = new ServerSocket ( port )) {
        Socket s = ss.accept();
        new StreamingServer().handlePlayerRequest( s );
      } catch (Exception e) {
          e.printStackTrace();
      };
  }
}
```

Method **handlePlayerRequest** parses the player HTTP request, analyzes if it is suitable, optionally gets the movie from the HTTP **Movies Server** using the method **getMovie**, and finally calls method **sendStream** to start streaming the file contents.

Suggested Roadmap:

1. Program the method **sendStream.** Test it, by calling it directly from the main method, while serving a movie from the local disk of the **Streaming Server** to a some <ip, port> endpoint pointing to VLC (as you did in Lab #7);
2. Program and test method **handlePlayerRequest** using the **sendStream** method to serve your clients;
3. Complete the full program and possibly the proposed improvement challenges, up to your choice, with all correct specified options.