Mp3 Transfer With Java

Server and Client

By
Ismael Mosquera Rivera

To my sister Ana, and her great family.

Abstract

The JDK (Java Development Kit) comes with a really huge API (Application Program Interface) with classes already tested and ready to use; these set of classes can help you to implement software to do almost any task that you can imagine. In this work, we demonstrate several of those classes in the java.net, java.io and javax.sound packages. Actually, what we did are two applications: Mp3Server and Mp3Client to transfer mp3 data from the server to a client; the idea is so simple:
A MP3Server has a repository with *.mp3 files. The server runs listening in an already known port to response requests from multiple clients. The clients comunicate with the server connecting to it in that port. A client makes requests for concrete mp3 files; when a client makes such a request, the server checks if the requested file is in its repository; if the server has the concrete mp3 file, loads the file into a stream, ask for relevant parameters (number of channels, sample rate...) and sends that information to the client; the client takes it and configures a local player according the information transfered by the server; the server starts transfering mp3 data to the client; the client reproduces the audio using the player configured according the parameters sent from the server; when the server finishes transfering the mp3 file the client is notified, then, the client can ask for another file or enter ‘quit’ which causes the client to shutdown.

This paper describes in more detail how these applications work and how they are implemented. On the other hand, from this initial work, an interested reader can add more features or even build a GUI (Graphical User Interface) using, for instance, the Java Swing toolkit or a web page to interact with each side, that is, server and client.

From this concept, it can be easy to transfer another kind of information as can be a file transfer; we hope that this work could be useful to somebody.

Contents

1 Description

This is an implementation suitable to transfer mp3 data from a server to be played by a client. So, there are two applications involved, one for the mp3 server, and another one for the mp3 client. In this section we are going to explain how these two applications were implemented.

The idea is to run a server capable to response requests from multiple clients so, the server creates an independent thread to response requests for each client which connects to it. The server runs listening in an already known port by the client, and the client knows also the machine name where the server runs.

When a client connects to the server, a socket is accepted by the server to the client, that socket is shared for both server and client to comunicate. Each side also make instances for input and output streams in order to transfer data with relevant information. The concrete class to code the server side is called AudioStreamSender and the class to implement the client is named AudioStreamReceiver; both take a socket as parameter, that is, the concrete socket shared by server and client.

These applications run from the command line so, we first open a shell to execute the server; once the server is running, we can open another shell to execute a client in the same machine, or run a client from another machine, for instance, if you have a local network with more than one computer; the host where the server is running can execute in a different platform than another computer in the net, since these applications are coded in Java, which is a programming language compiled in a bytecode which is independent of the platform, that is, the same bytecode can run under any platform. If the client successfully connects to the server, it asks you for a file; you can type a file name or “quit” which causes the client to shutdown. You must enter the file name and also the .mp3 extension:

```
Filename: sid.mp3
```

Or

```
Filename: quit
```

to exit. Following, we are going to describe the server side first, then, we will explain the client side; afterwards, we present an example to show how server and client interact.

Mp3Server

The server side application declares a java.net.ServerSocket object and makes an instance creating a new ServerSocket object passing as parameter to its constructor the port number where the server will be listening for incoming requests from clients:

```
ServerSocket server = null;
try
{
server = new ServerSocket(StreamTransferPort.getPort());
}
```

```
catch(IOException e)
{
System.err.println(e);
System.exit(1);
}
```

if the creation of the ServerSocket object success, the application enters a loop where requests from clients are accepted and a java.net.Socket object is returned and passed as parameter to a new instance of the AudioStreamSender, which is the class used to do the task needed from the server side; that class implements the Runnable interface so, each request from a client causes the creation of an independent thread to comunicate with the client through a shared socket. The sentence to create such a thread and start it is as follows:

```
(new Thread(new AudioStreamSender(server.accept()))).start();
```

notice that the server.accept() call returns a java.net.Socket, which is the shared socket used to perform the server/client conversation. The class declration to do the task is declared as follows:

```
class AudioStreamSender implements Runnable
```

and the declaration for its constructor, as you maybe already guessed is:

```
public AudioStreamSender(Socket socket)
```

since this class implements the Runnable interface, it has a ‘run’ method which does the task; in that method, a java.io.BufferedReader object is created for input, taken as parameter an input stream from the client socket; for output, a java.io.PrintWriter object is also created passing to its constructor an output stream from the client socket; now, we have the basics to start the server/client conversation.

The steps followed by the server are:

Read the name of the mp3 file requested from the client and check whether the file is in its repository; in case of the read string were “quit”, the server shuts down that client. On the other hand, if the server has the requested file in its repository, loads the file into a javax.sound.sampled.AudioInputStream object; inside the method used to load the audio file, a java.io.DataOutputStream object is created passing as parameter an output stream from the client socket, and relevant data is sent to the client in order to configure a player suitable to reproduce the mp3 data sent from the server to the client. In case that the server does not have the mp3 file in its repository, the client is also notified. Following we show some sentences involved in those steps:

```
if(hasFile(filename))
{
        out.println("found");
loadAudio(FILE_REPOSITORY+filename);
System.out.println("AudioStreamSender: dispatch " + filename + "...");
// send audio stream
sendAudioStream(audioInput, new DataOutputStream(client.getOutputStream()));
}
else
```

```
{
        out.println("!found");
}
```

the methods involved are private of the AudioStreamSender class and here they are their signatures:

```
private boolean hasFile(String filename)
private void loadAudio(String audioFile)
private void sendAudioStream(AudioInputStream input, DataOutputStream output)
throws IOException
```

these methods are self explanatory, the first checks if the server has a mp3 file in its repository, the second loads the audio into an AudioInputStream object and sends the frame size, number of channels and sample rate in order that the client can configure a local player to reproduce the mp3 data send by the third method.
now that we already described in some detail the server side, let us to explain the work done from the client side.

Mp3Client

The client side application receives the name of the host where the server is running as parameter; the client already knows the port where the server is listening for incoming requests; when the client creates a socket in order to connect with the server, that socket is also accepted by the server so, the server and client share that socket during its conversation. The creation for such a socket is as follows:

```
Socket s = null;
try
{
        s = new Socket(HOST, StreamTransferPort.getPort());
}
catch(UnknownHostException e)
{
System.err.println(e);
}
catch(IOException e)
{
System.err.println(e);
}
```

the class which does the task in the client side is named AudioStreamReceiver, and its constructor takes as parameter the shared socket, the declaration for that class is:

```
class AudioStreamReceiver
```

and the signature for its constructor:

```
public AudioStreamReceiver(Socket s)
```

the main method to do the task is called ‘run’, although this class does not implement the Runnable interface, we decided to use this name just for simetry with the server side. As in the case of the server, there are some private methods of the AudioStreamReceiver which collaborate to do the work.

After the socket is created, it is passed as parameter to an instance object of the AudioStreamReceiver and its run() method is called.

```
(new AudioStreamReceiver(s)).run();
```

the ‘run’ method declares some streams to comunicate to the server and another one used by the user to read from the command line; remember that the client ask the server for mp3 files or exits if the file name is “quit; following are some sentences to show these initial steps:

```
BufferedReader in = null;
PrintWriter out = null;
BufferedReader userInput = null;
try
{
userInput = new BufferedReader(new InputStreamReader(System.in));
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true);
System.out.println("Connected to mp3 server in port " + StreamTransferPort.getPort());
```

afterwards, the user types a mp3 file name or “quit” to shutdown; if the file requested from the client exists in the server repository, the client is notified and the steps to transmit and reproduce the mp3 audio data start, those steps are the following:

```
String filename;
while(true)
{
        System.out.println("<enter \"quit\" to shutdown>");
        System.out.print("File name: ");
filename = userInput.readLine();
out.println(filename);
if(filename.equals("quit"))
{
System.out.println("Mp3Client: shutdown!");
break;
}
if(in.readLine().equals("found"))
{
        // get parameters from server
        getParameters();
        // now we have parameters to setup the player
initPlayer();
// play audio stream
play();
}
else
{
        System.out.println("Sorry, " + filename + " file not found.");
}
```

As we can see, there are three private methods to perform the main task; one gets the needed parameters from the server in order to configure a local player then, the player is initialized and finally, the mp3 audio data is played. The signaturs for these methods are:

```
private void getParameters() throws IOException
private void initPlayer()
private void play() throws IOException
```

You can pick inside the code (server and client); it is so easy to understand. After this description, we demonstrate how the Java classes used in this implementation work, and we hope that you enjoy these two applications. Following, we present a little example to show how these two applications interact; first, we run the server in a shell; once the server is running, we open another shell to run the client; the client asks for a pair of files and, finally, the client finishes typing the “quit” string as a file name. Notice that the server stills running listening for requests for other clients.

```
Mp3Server version 1.0
Transport protocol: TCP
Author: Ismael Mosquera Rivera

Mp3Server listening in  port 11105
Mp3Server: press ctrl+c to quit.
AudioStreamSender: dispatch beatles.mp3...
AudioStreamSender: dispatch acdc.mp3...

AudioStreamSender: shutdown!
```

Here we can see the server running, notice that each request from a client is dispatched from the server showing a message. When the client enters “quit”, the thread for that concrete connection terminates but, actually, the server continues listening for requests from other clients. To quit the server you must press Ctrl+c

```
Mp3Client version 1.0
Transport protocol: TCP
Author: Ismael Mosquera Rivera

Connected to mp3 server in port 11105
<enter "quit" to shutdown>
File name: beatles.mp3
<enter "quit" to shutdown>
File name: acdc.mp3
<enter "quit" to shutdown>
File name: quit
Mp3Client: shutdown!
```

As we said before, the client makes a pir of requests and, when the “quit” string is typed for the user, causes the client to shutdown.

Following, we recommend some bibliography to learn more about things related to this work.

Elliotte Rusty Harold: Java Network Programming, Third Edition; O’Reilly
Java I/O 1st edition; O’Reilly 1999
Ron Hitchens: Java NIO 1st edition; O’Reilly 2002
Java Sound API Programmer's Guide; Sun Microsystems Inc.

2 Conclusions

We implemented a simple mp3 transfer system, both server and client sides; in this work, we demonstrate some classes in the java.net, java.io and javax.sound packages which come with the JDK API.

From this initial idea, we can improve this system adding more features, as can be to show a list with the content of the server repository and, since these two applications run from the command line, build a GUI (Graphical User Interface) or a web page for server and client interaction. In this case, the server side creates a new thread for each incoming request from a client; we can improve that creating, for example, a thread pool in the server side to get a better performance; another improvement can be to add support for more audio file formats.

These two applications (server and client), have been tested in a small local network, running the server in one host, and multiple clients, one on each computer in that network and, even with a client running in the same machine than the server; we made requests to the server from each client concurrently, and it runs so good with a nice performance.

In the appendix, we explain in som detail how to run the server and multiple clients and some notes about the author.

3 Appendix

Although we already explained how to run these two applications (server and client) for this mp3 transfer system, we are going to make a little review in this section. There is a folder ‘mp3_transfer’ which contains two folders, ‘mp3_server’ and ‘mp3_client’ and a ‘readme.txt’ file that we encourage you to read, since it explains also how to execute this simple mp3 transfer system.

You have just to copy the ‘mp3_server’ in the computer that you decide the server will be running; then, open a shell and go to that folder; there are simple scripts to build and run the server, since the application is already compiled, you do not need to build it but, if you prefer, you can rebuild it executing the ‘build.bat’ file or the ‘build.sh’ depending if you are under a MSWindows command line shell or under a Linux or a Linux like shell. The same is true to run the server using the ‘run.bat’ or the ‘run.sh’ file. The first time that you execute the server side in a host, the system will ask you for permission to listen in the port number 11105, you must accept in order to run the mp3 server.

You can run the mp3 client in the same machine than the server, but if you want to execute it in other computer, you just have to copy the ‘mp3_client ’ folder in it. As in the case of the server side, you will see simple scripts to build and run the application; in this case, you must change the host name in the ‘run.bat’ and ‘run.sh’ files by the one where the server is running since the host name where the server runs is passed as parameter to the client application; in our case, that name, as you will see, is CAMPANILLA.

Once all of that is done, open a shell window and go to that folder (mp3_client), and execute the client side application using the ‘run.bat’ or ‘run.sh’ file depending in the shell type under you are working.

Notice that there are 3 mp3 files in the server repository which you can use to test this mp3 transfer system; each time a song is requested, you have to wait until the playback finishes to ask for another one so, an improvement can be to add a way to stop or pause playing.

About The Author

Ismael Mosquera Rivera has a degree in Computer Science by Pompeu Fabra University http://www.upf.edu, Barcelona – Spain, 2004. He also was involved in the Clam project, http://www.clam-project.org (C++ Library for Audio and Music); unfortunately, he is affected by a severe eye disease (Retinitis Pigmenthosa), which caused him to get totally blind a number of years ago. Anyway, he finally decided to get back to the computer world, so, he installed a screen reader in his old computer. The one he installed is only available for Windows OS, but he got the MinGW compiler which has also a command line shell from where you can have a little Linux in your Windows system. So, all the textual information in a computer can be read, and he began to do research. Among the things he has done until that, the development of this simple mp3 transfer system encouraged him to write a paper explaining how it was done, and this is the result. He apologizes the reader to the lack of graphics, diagrams and identation in the source code. Anyway, his expectation is that an interested reader can take benefit from this article.