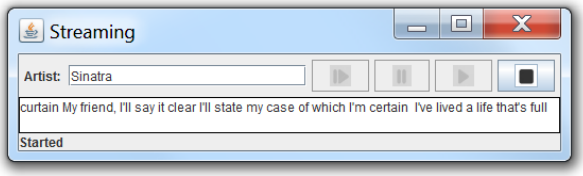
Assignment 5: Networking and Channels (100 Punkte)

In this assignment you should implement a client-server application by using the channel and buffer API in Java's NIO implementation.

The program to implement is a client-server application for streaming text data from a server to a client. In the client application it is possible to specify a name of the content file and then the data in this file should be streamed piecewise from the server to the client and displayed there.

Remark: This example program actually should simulate streaming of multimedia data from a server to a client (to keep the program simple, simple text data is used). Further, the assumption is that the client program runs on a small device and only a small portion of the content can be kept at the client. The client therefore has to request the data from the server as it is displayed.

The client has a simple UI where the name for the content file can be specified, streaming can be started, and canceled. Optionally (not required in the assignment), you can implement an action for pausing and continuing the streaming process. In the UI the text floats with a specific speed from left to right on a single line. The following figure shows how the UI should look like. The program is currently streaming a song text from Frank Sinatra.



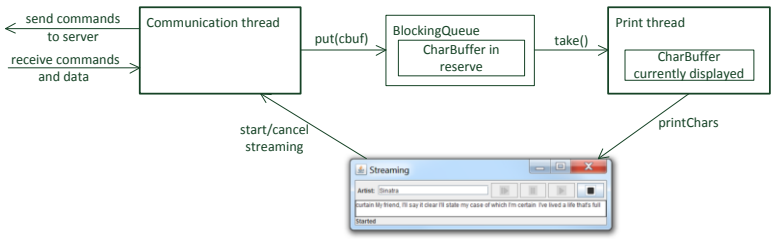
Client program:

Thus, the main idea of the client program is to keep a portion of the content data buffered. In fact, the client program should use two `CharBuffer`s, one whose content is currently displayed, and one in reserve. As soon as the current is finished, the one in reserve should be started and a new one is requested from the server.

A skeleton of the client application is provided for download. It contains a method `printChars(CharBuffer cbuf)` which will slowly put out the text of the given `CharBuffer` in a `JTextArea`.

Client architecture

The following figure sketches the architecture of the client application. There are two threads, one for handling the communication with the server, the other one for putting out the current text buffer. They synchronize on a shared blocking queue which holds the buffer in reserve.



Requirements:

- Use `SocketChannel` for sending and receiving commands and data.
- Use `ByteBuffer` and `CharBuffer` of appropriate size for communication of commands and data.

Hints:

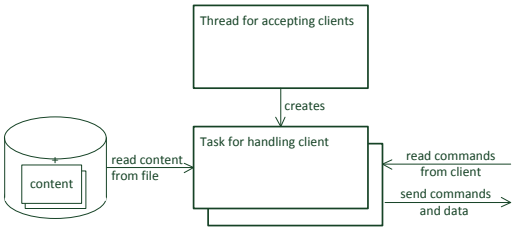
- You can use a `BlockingQueue<CharBuffer>` (with capacity 1) for holding the buffer in reserve.

Server program:

The server uses `ServerSocketChannel` for accepting clients. It has to handle each client in a separate task. Use a `CachedThreadPool` executor for executing tasks. A sketch of a possible client-server communication protocol is given below.

Server Architecture:

The following figure shows the architecture of the server. In one thread the client connections are accepted. For each client a new task is created which is executed by a `CachedThreadPool` executor. It is handling the communication with the client. In particular, it waits for request from the client for new data and only then reads the next buffer from the file and sends it to the client.



Requirements:

- Use `ServerSocketChannel` for accepting client connections.
- Use `SocketChannel` for sending and receiving commands and data.
- Use `ByteBuffer` and `CharBuffer` of appropriate size for communication of commands and data.

Hints:

- For reading the data from the file, you best use a `BufferedReader` and the `read(CharBuffer cbuf)` operation which reads into a `CharBuffer`, e.g.,

```
BufferedReader fileReader = Files.newBufferedReader(filePath);
CharBuffer cbuf = CharBuffer.allocate(20); // buffer size is 20 characters
int nRead = fileReader.read(cbuf);
```

Communication protocol

In the following, a sketch of a possible communication protocol between client and server is given.

Scenario 1: Normal course

Client	Server
Connects to the server and opens channel	
	Server accepts **connect** request and opens channel
	Server sends **hello**
Client receives **hello**	
Sends a **get** request with name for the content file	
	Receives **get** request and checks if content available
	Sends **ackn** if content available
Sends **next** request for data	
	Receives **next** request
	Sends **data package** (one buffer)
Receives **data package**	
As soon as more data needed, sends **next** request for data	
	Receives **next** request
	Sends **data package** (one buffer)
Receives **data package**	
As soon as more data needed, sends **next** request for data	
	Receives **next** request

	Sends **data package** (one buffer)
Receives **data package**	
…	…
	When no more data available **shuts down output**
Recognizes end of data, sends **bye** message and terminate	
	Receives **bye** message and client handler terminates

Scenario 2: Cancel streaming by client

Client	Server
Connects to the server and opens channel	
	Server accepts **connect** request and opens channel
	Server sends **hello**
Client receives **hello**	
Sends a **get** request with name of the content file	
	Receives **get** request and checks if content available
	Sends **ackn** if content available
Sends next request for data	
	Receives next request
	Sends data package (one buffer)
Sends **next** request for data	
	Receives **next** request
	Sends **data package** (one buffer)
Receives **data package**	
…	…
Sends **cancel** message and terminate	
	Receives **cancel** massage and client handler terminates

Scenario 3: Content not available

Client	Server
Connects to the server and opens channel	
	Server accepts **connect** request and opens channel
	Server sends **hello**
Client receives **hello**	
Sends a **get** request with name of the content file	
	Receives **get** request and checks if content available
	Sends **not_ackn as content is not available**
Receive **not_ackn** and sends bye message and terminate	
	Receives **bye** message and client handler terminates