|  |
| --- |
| Department of Software Engineering  Mehran University of Engineering and Technology, Jamshoro |
|  |

|  |  |  |  |
| --- | --- | --- | --- |
| Course: SWE324 – Distributing Computing | | | |
| Instructor | Rabeea Jaffari | **Practical/Lab No.** | 02 |
| Date | 02 April 2019 | **CLOs** | CLO-4: P3 & P4 |
| Signature |  | **Assessment Score** | 1 Marks |

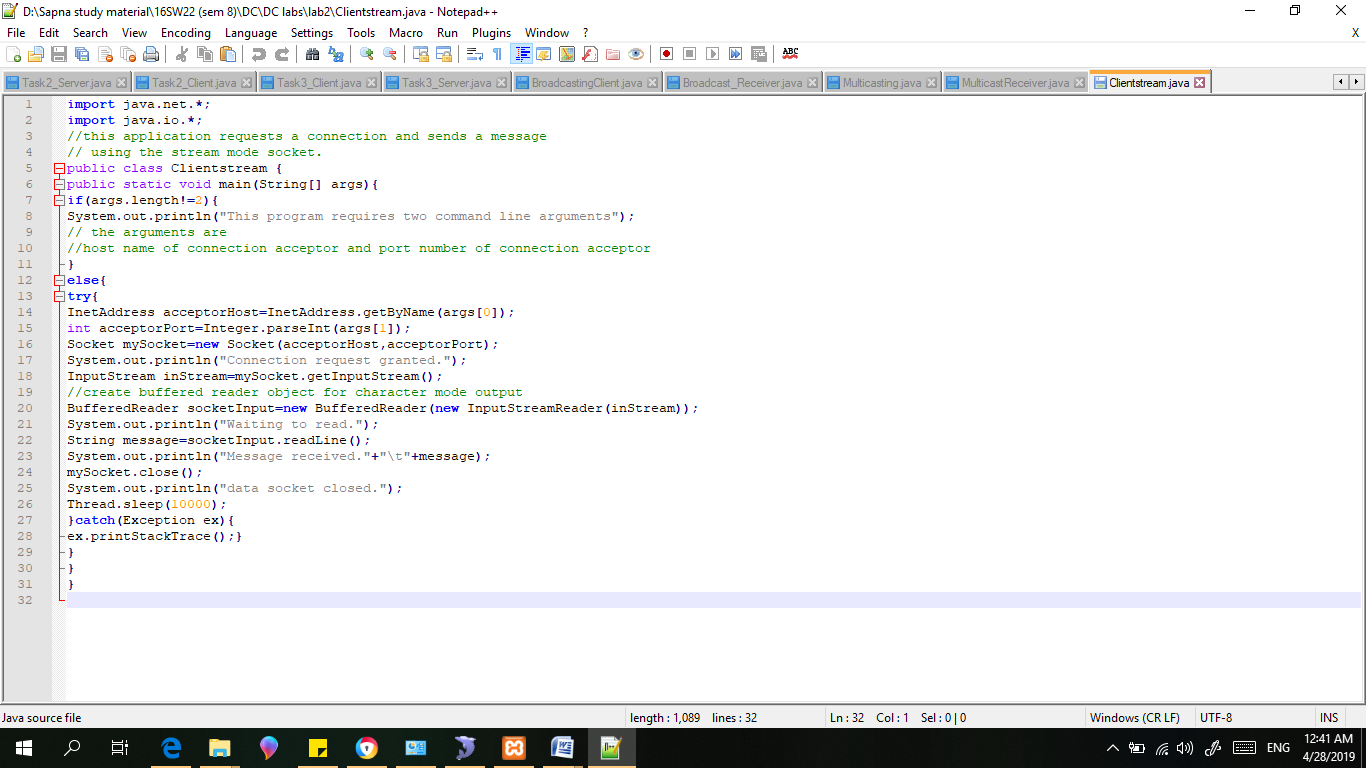
|  |  |
| --- | --- |
| Topic | To become Familiar with OLTP System Design |
| Objectives | * To Creating applications using Stream sockets Api. |

|  |
| --- |
| Lab Discussion: Theoretical concepts and Procedural steps |

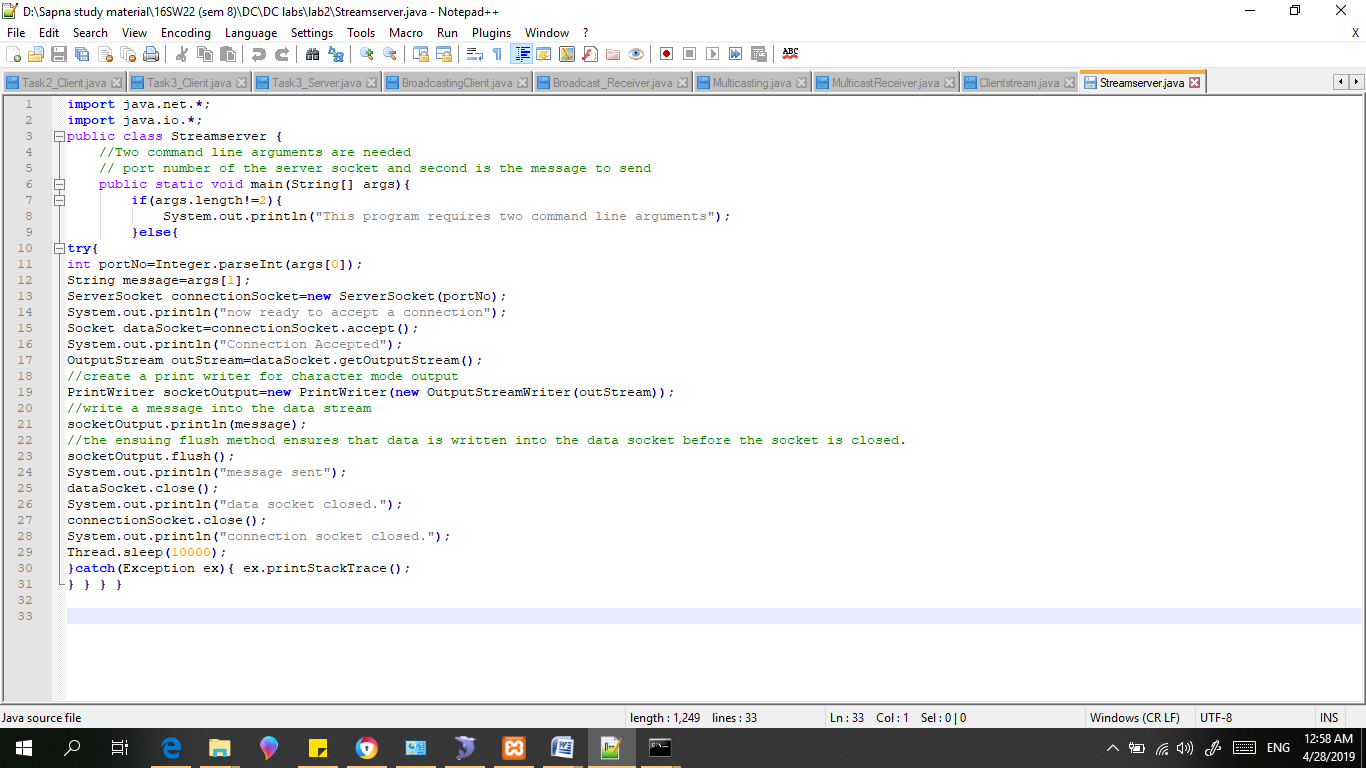
|  |
| --- |
| Lab Tasks |
| Submission Date: 09-04-19 |

1. Compile and run the above code. Start the acceptor first and then the requestor with appropriate command line arguments. Describe and explain the output.

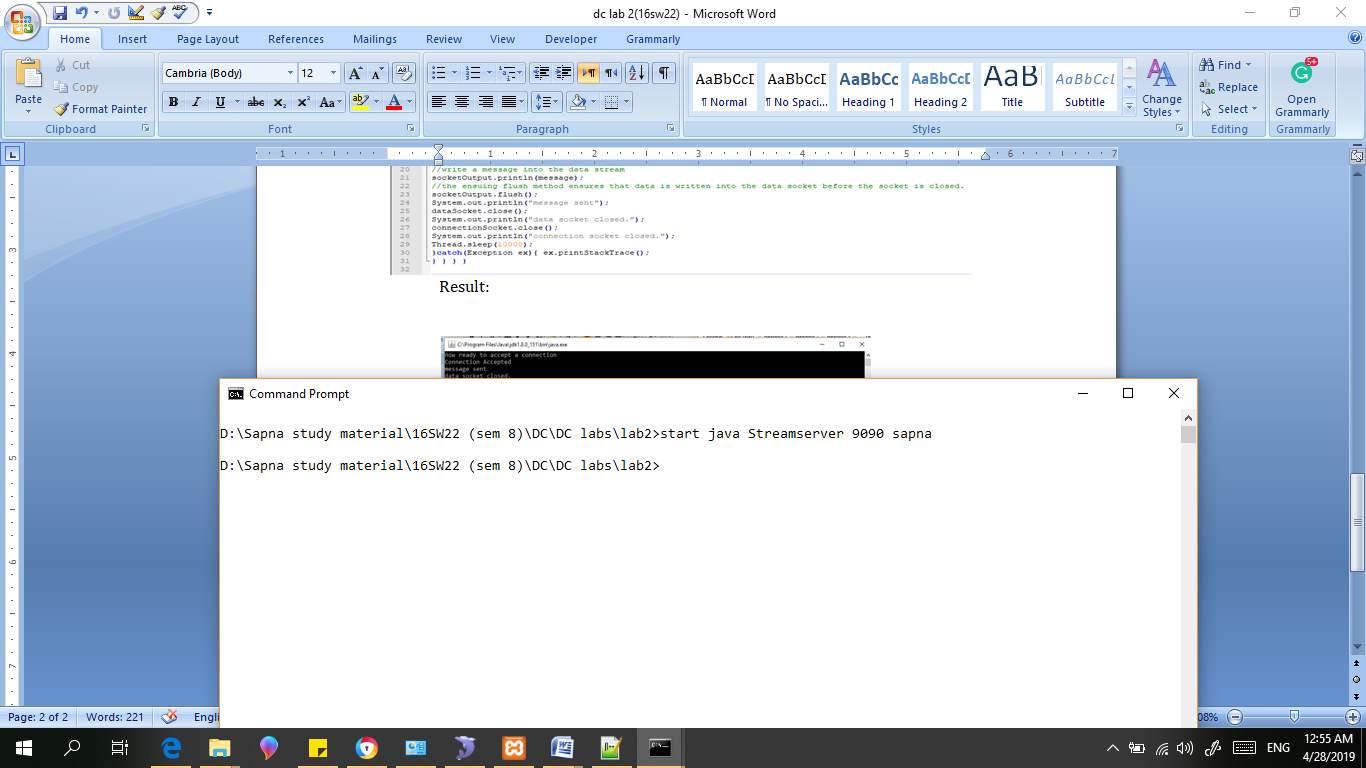
Client :

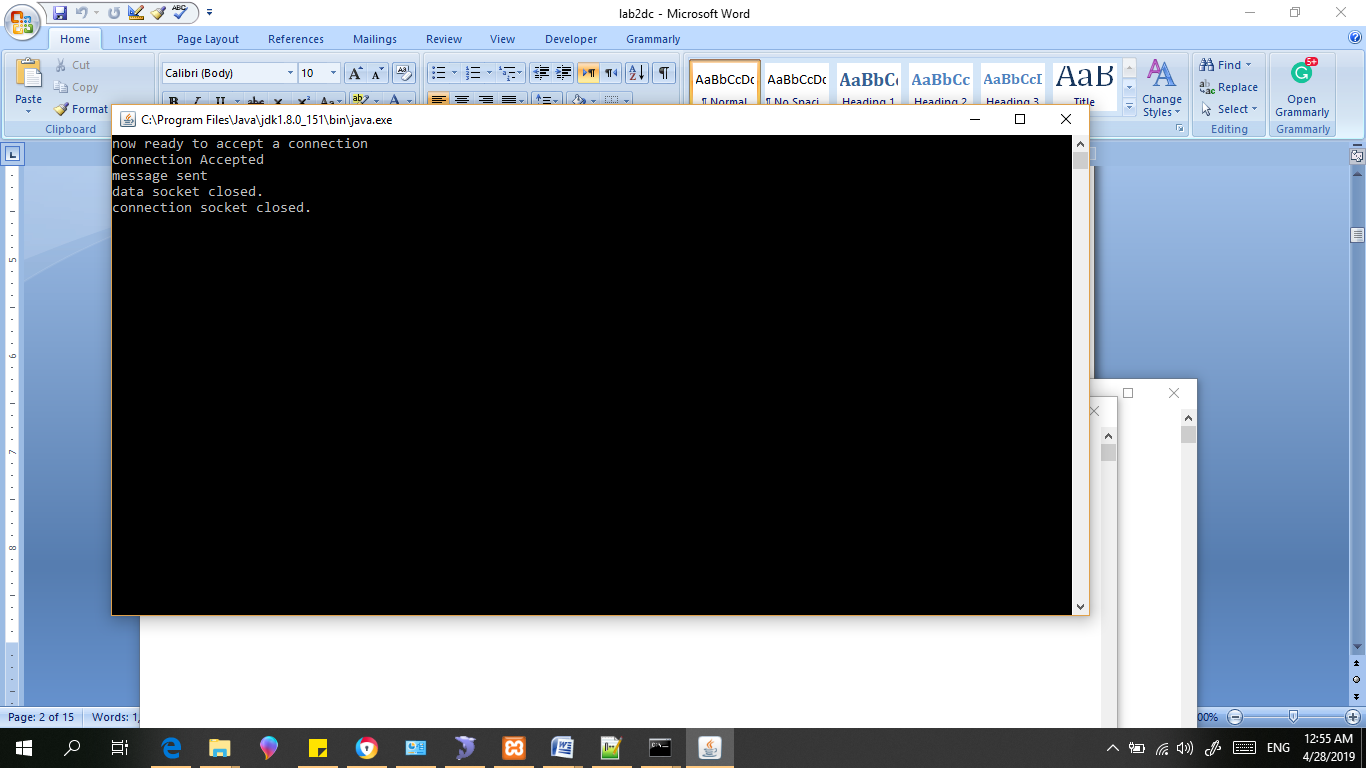


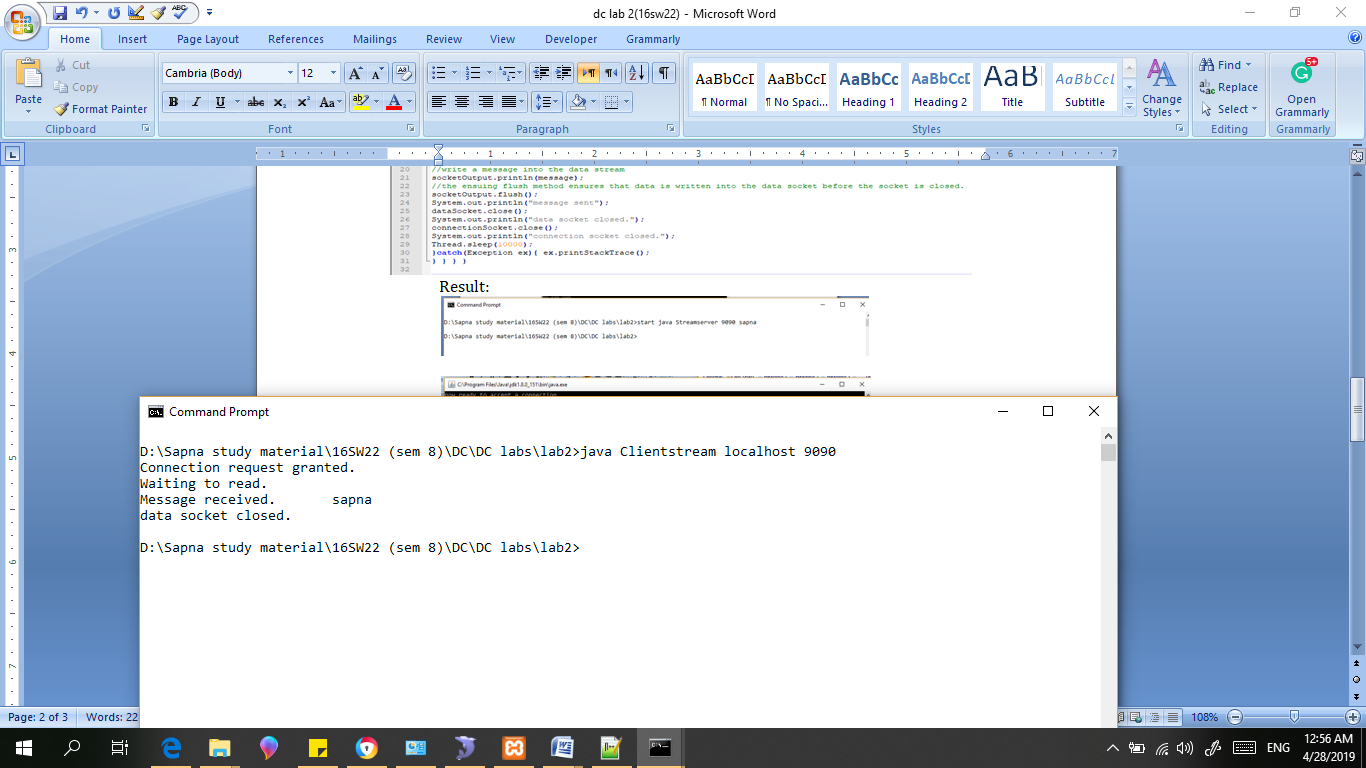
Server :



Results:

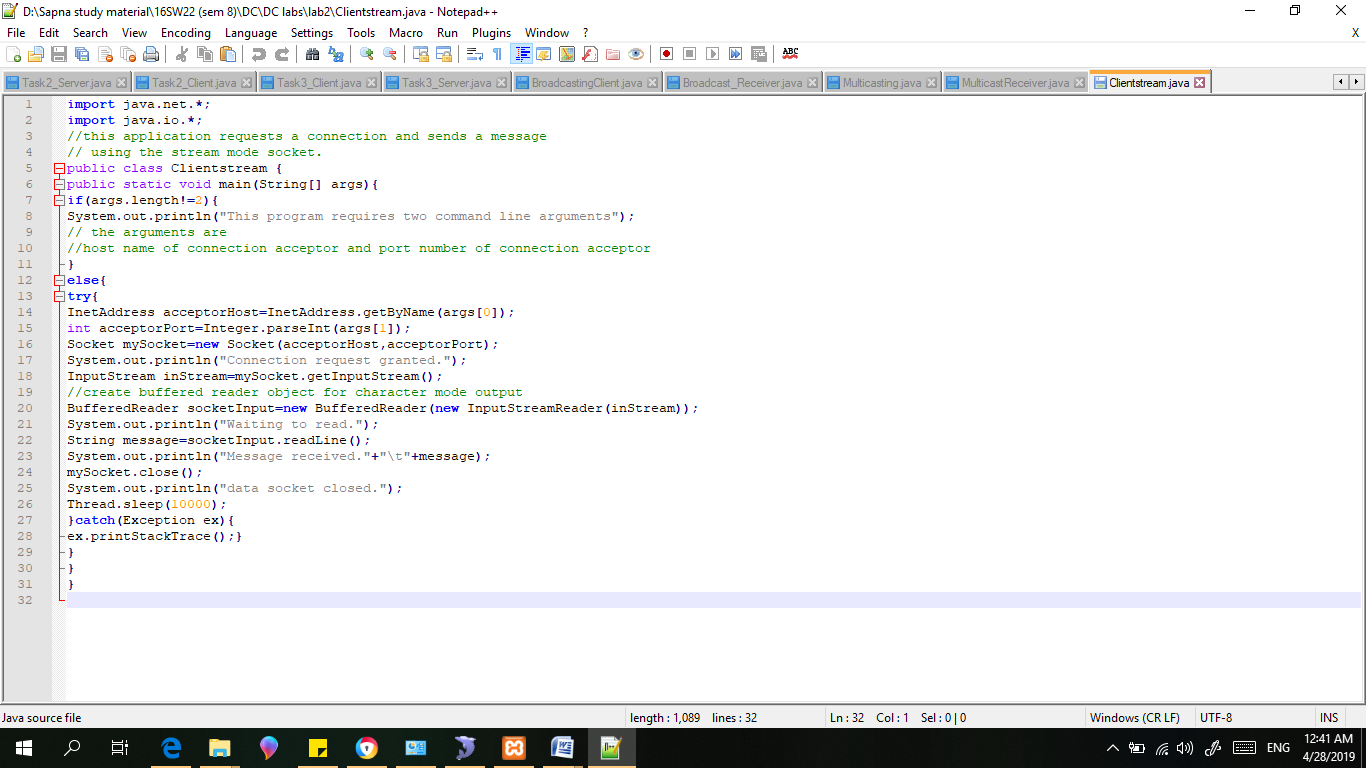
Res



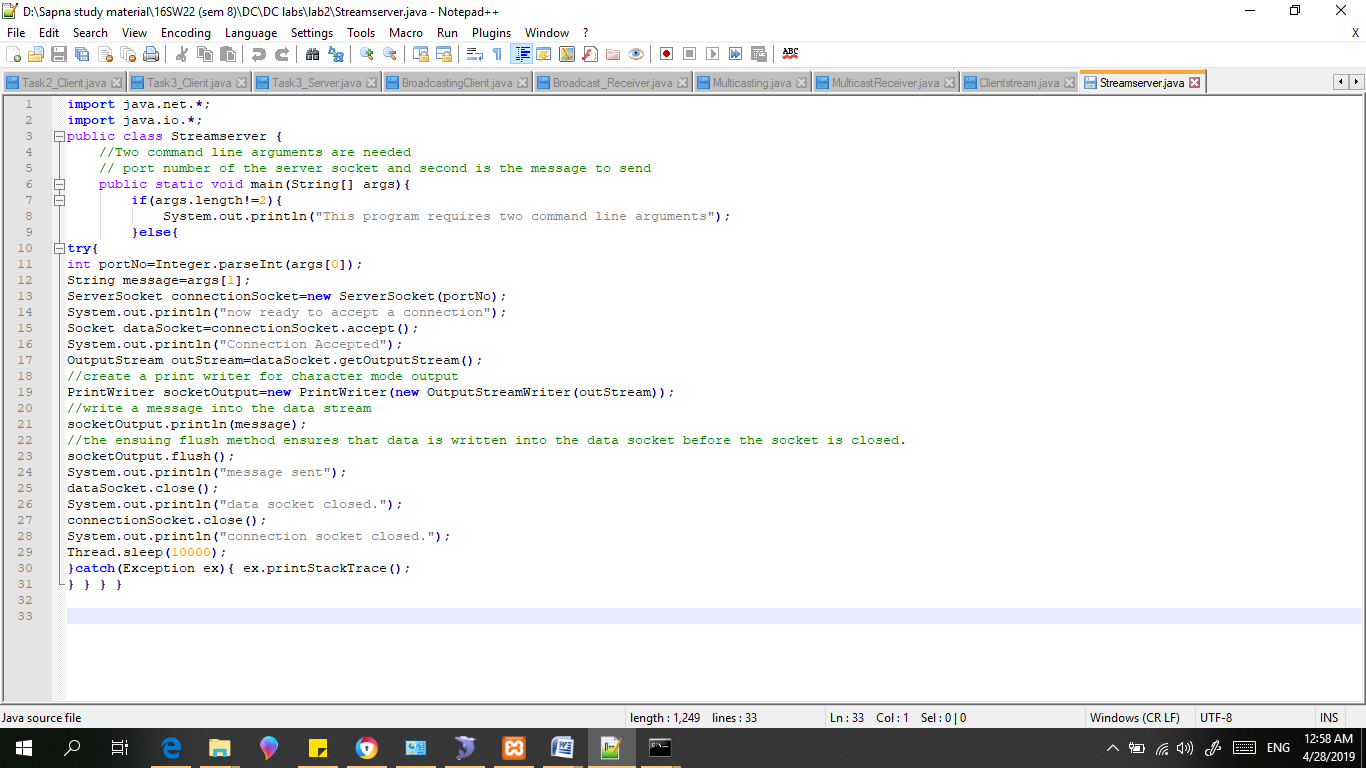


1. Now run the code again, but reverse the order of program’s execution. Start the requestor first and then the acceptor. Describe and explain the outcome.

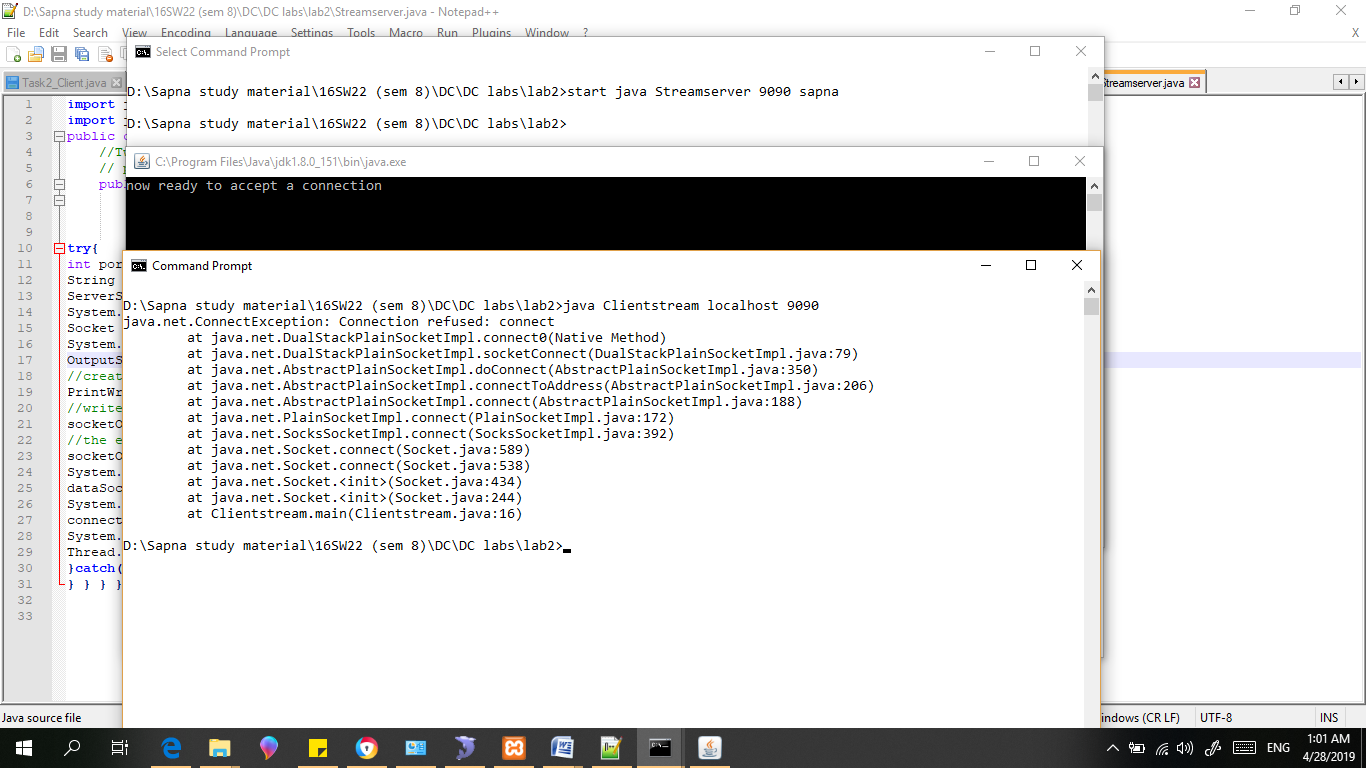
Client :



Server :

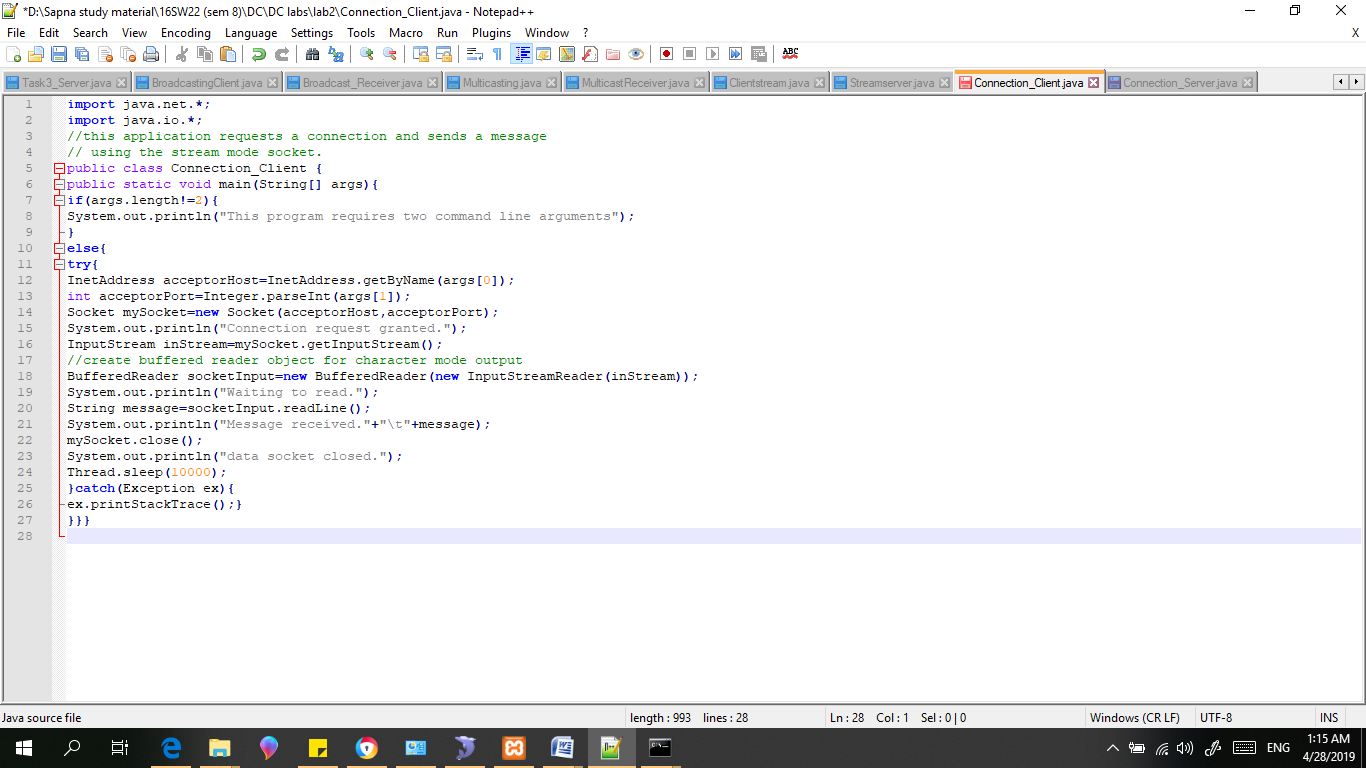


Results:

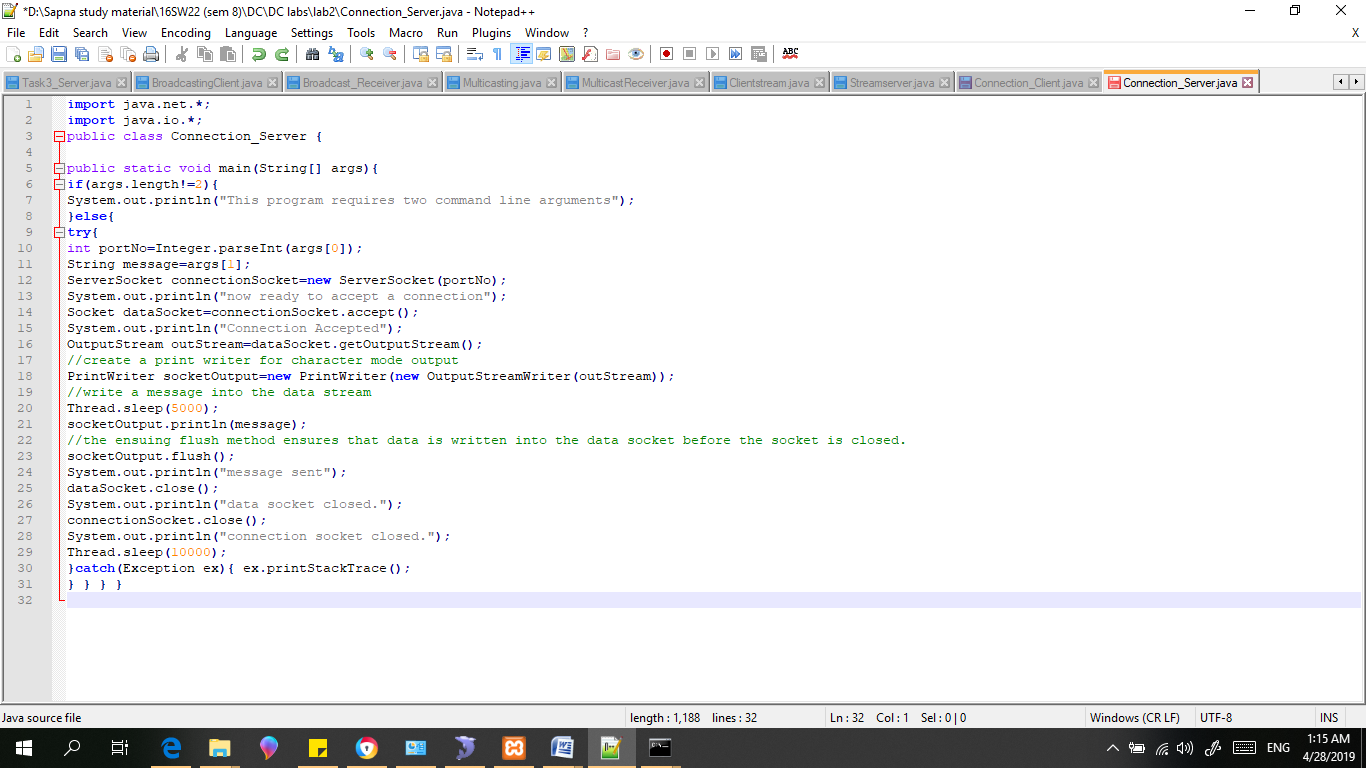


1. Add a time delay of 5 seconds in the Connection Acceptor process just before the message is written to the socket, then run the program. This will show you the blocking at the receiver. Show a trace of the output of the processes.

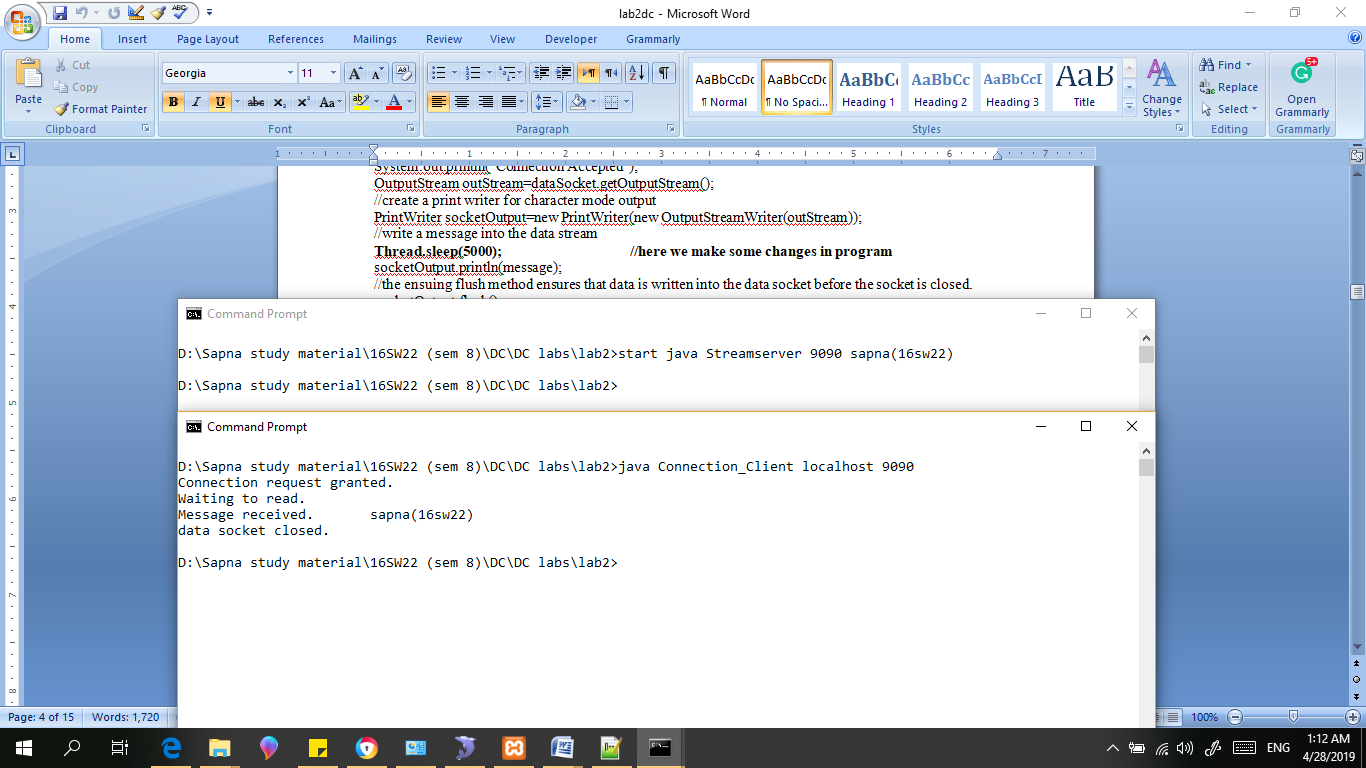
Client:



Server:



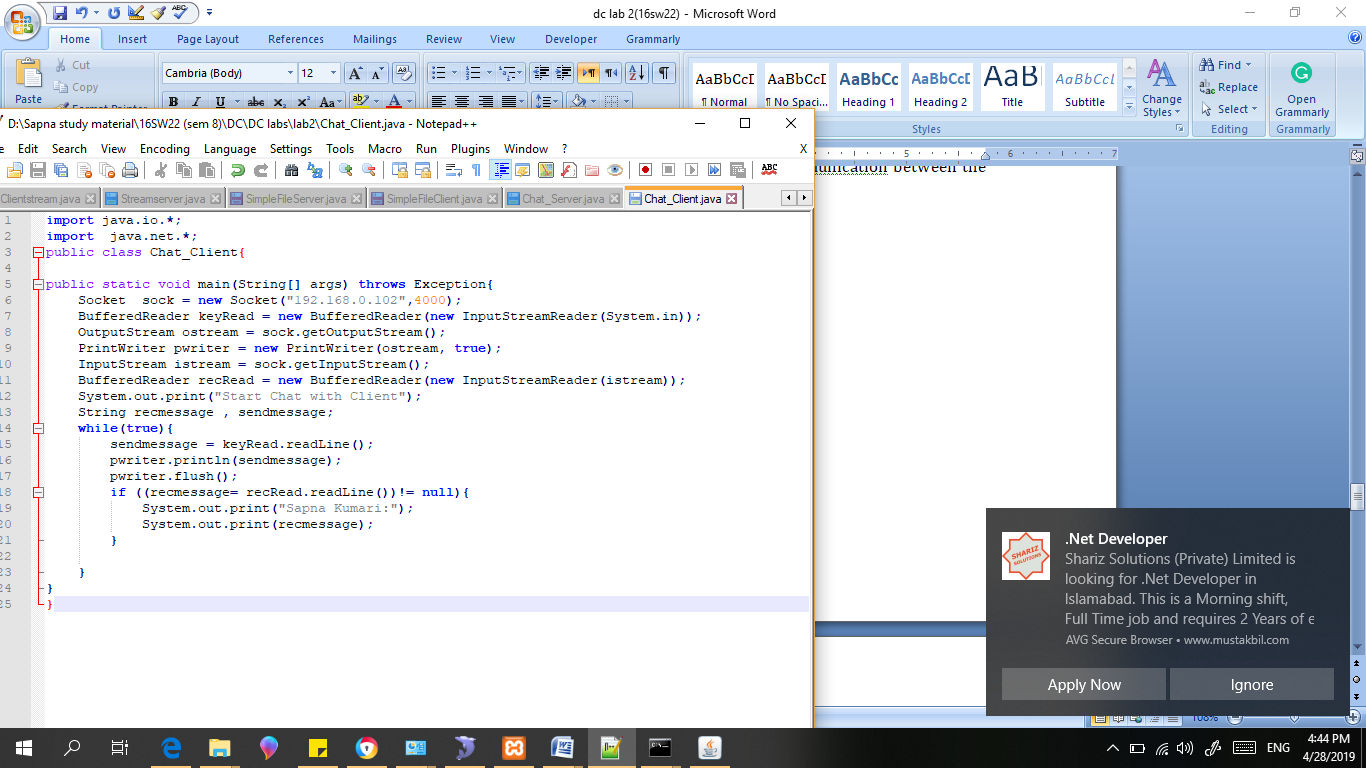
Result: No any change is observer by changing the time (delay of 5 seconds).



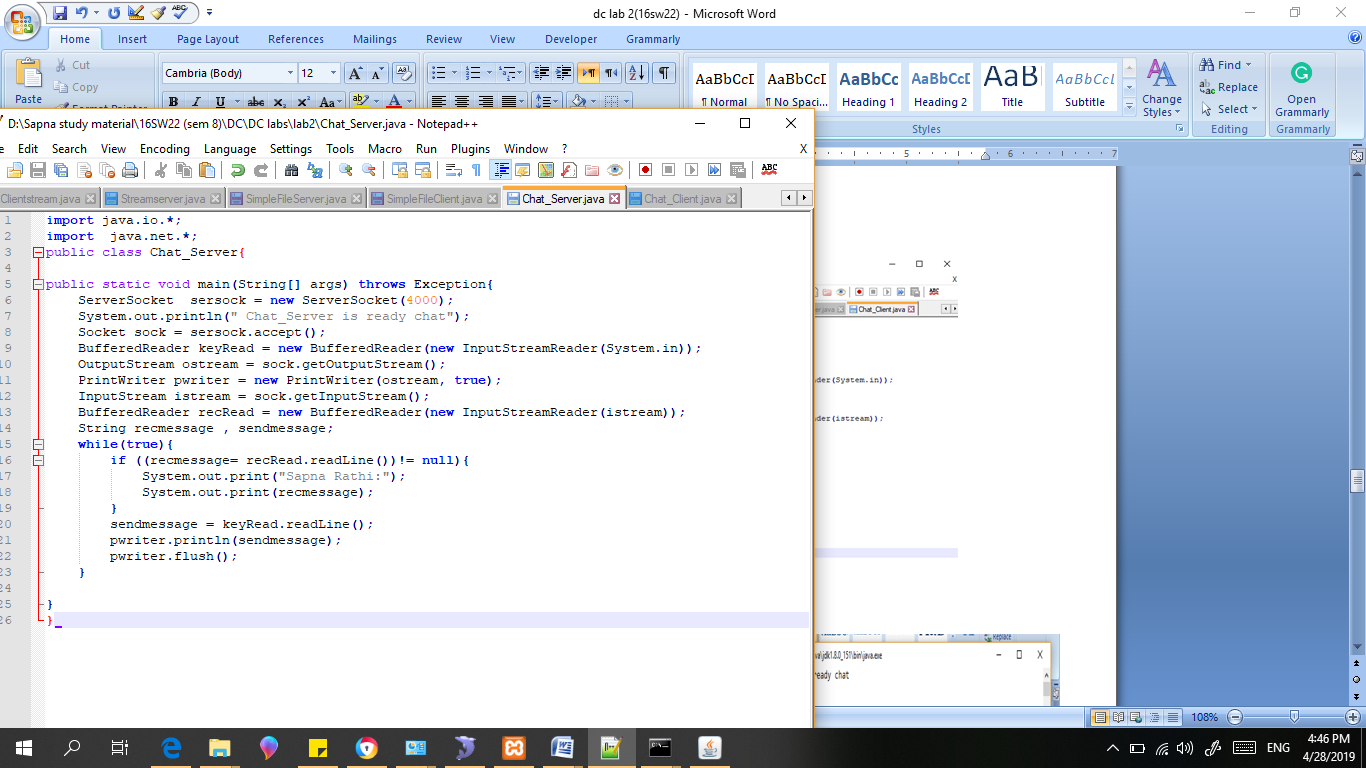
**BONUS TASKS for PRACTICAL#02 (+1)**

1. Modify the sample code to include two way communication between the client and the server.

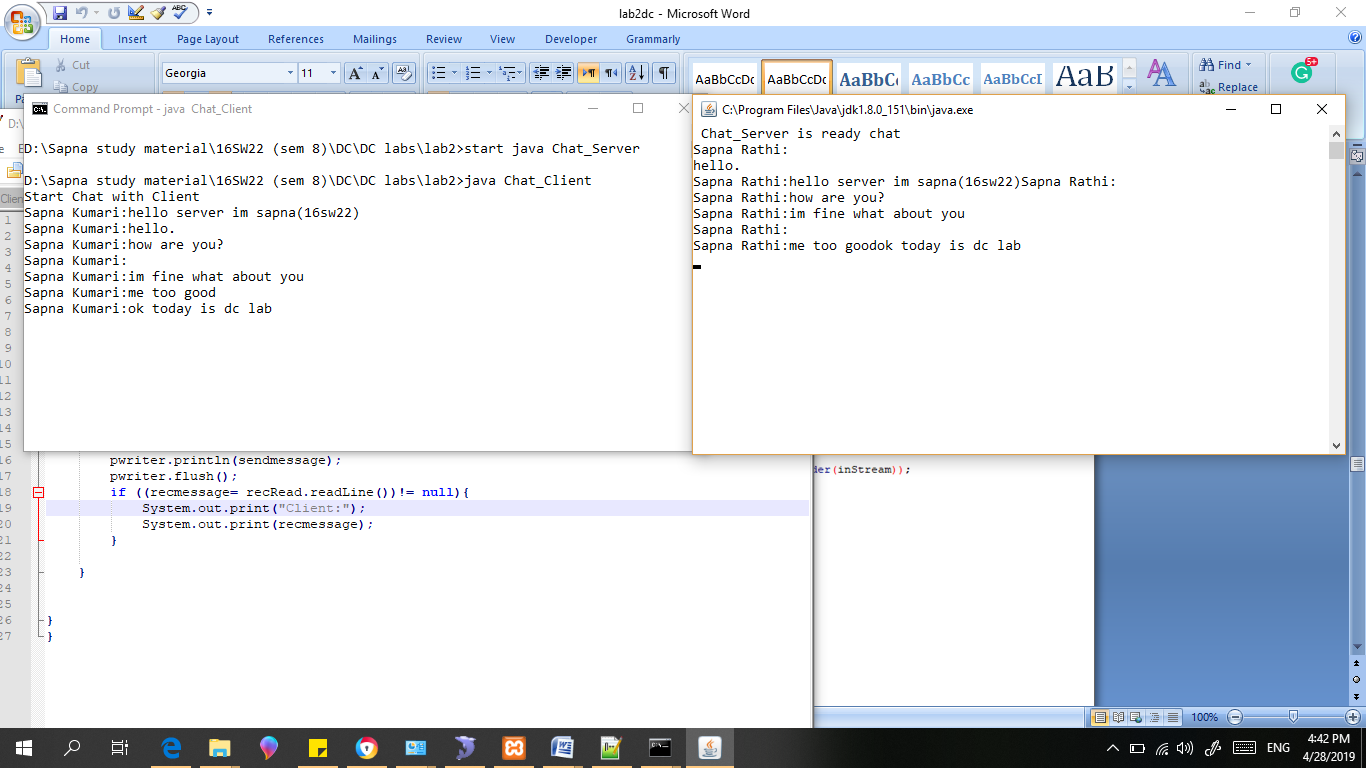
Client:



Server:

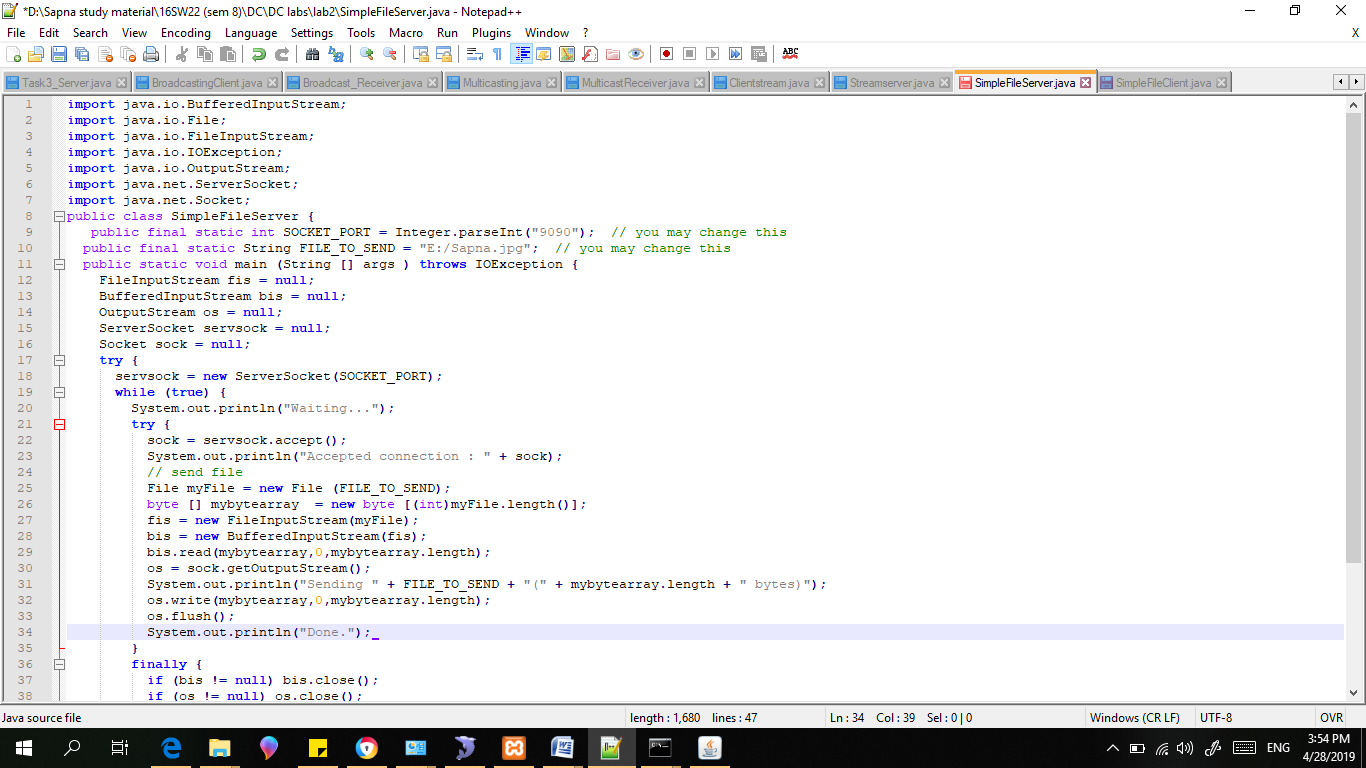


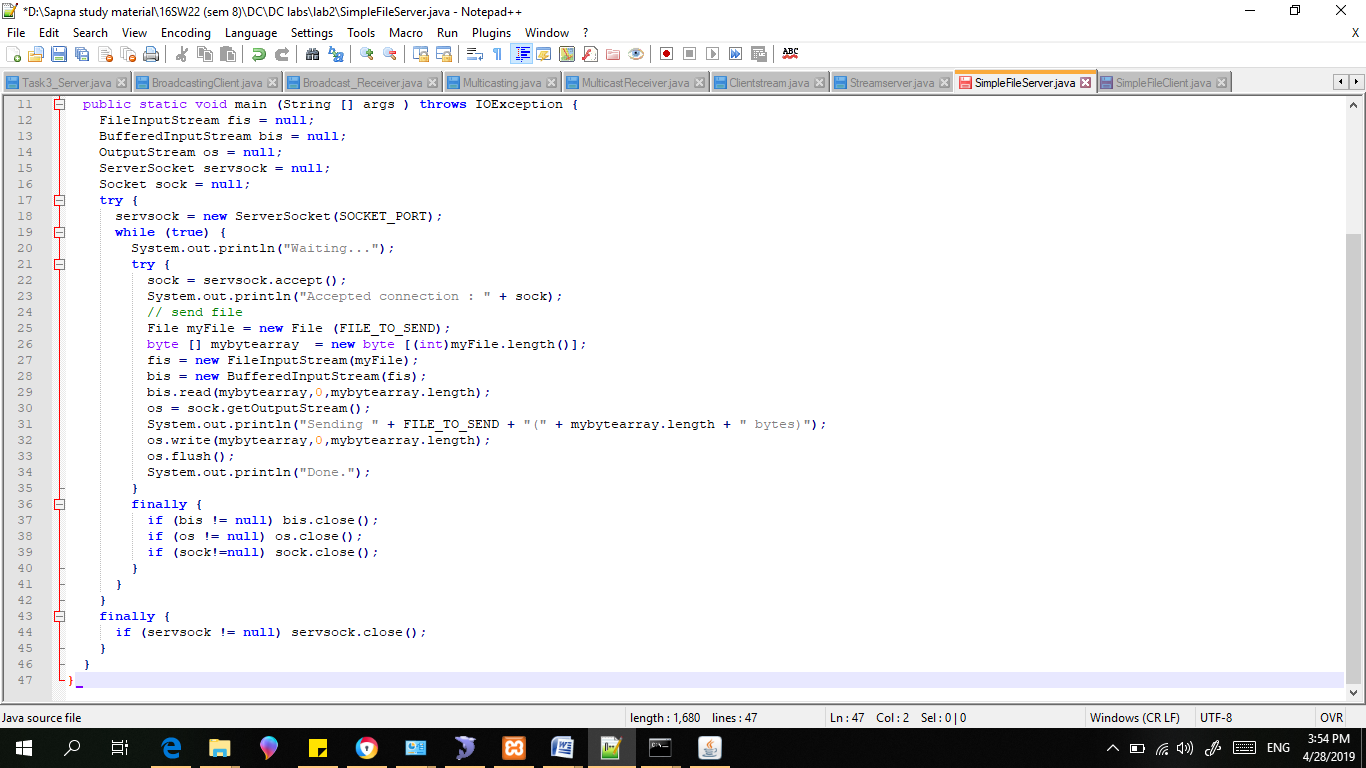
Result:



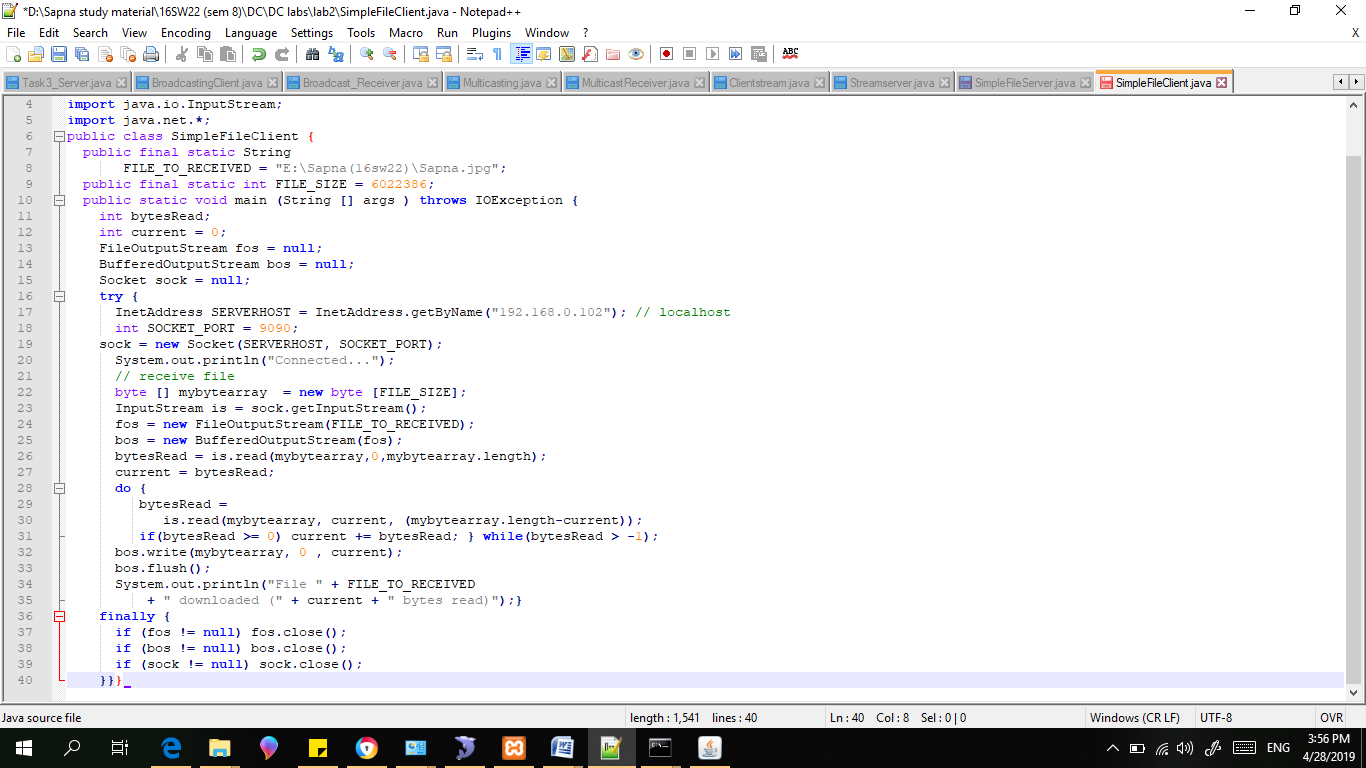
1. Modify the sample code to send complete files between the client to the server.

Server:

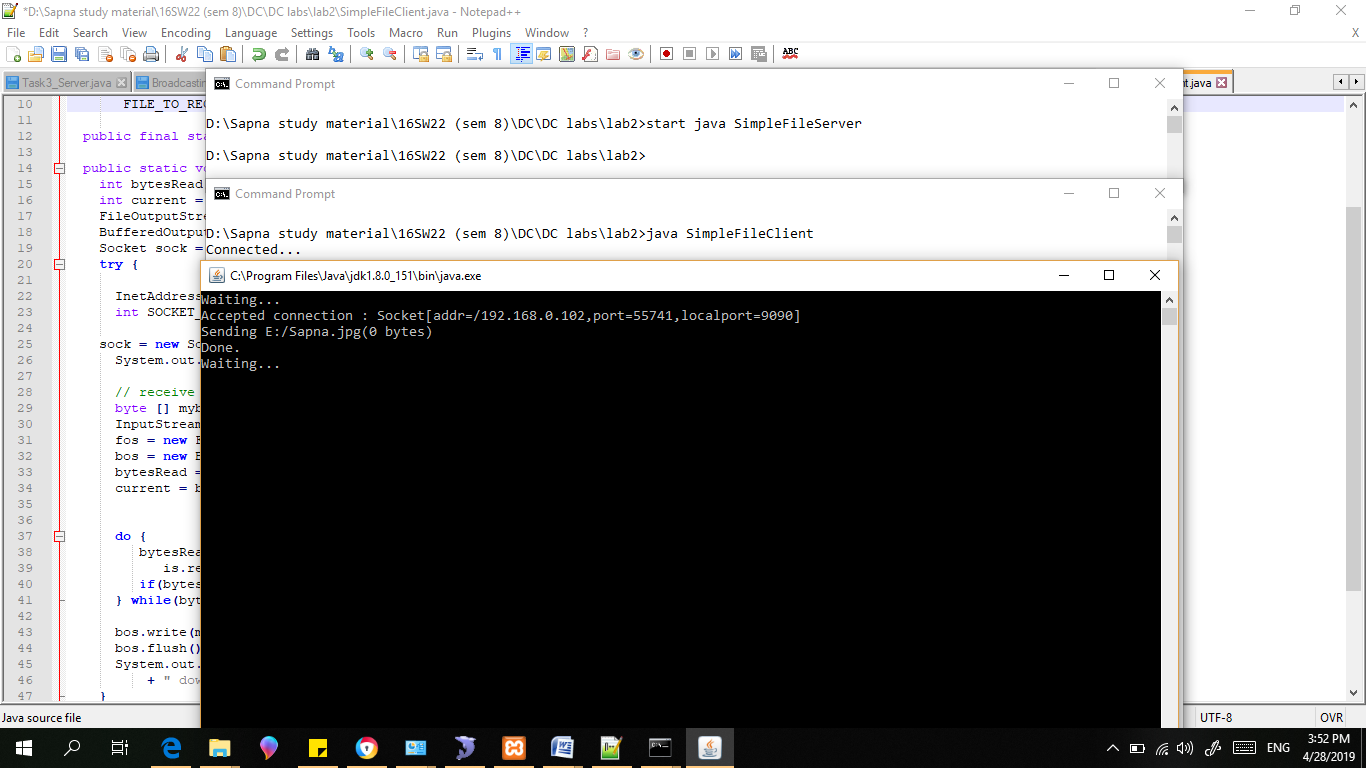




Client:



Result:



1. Explore the non-blocking java socket API in the nio package and implement a sample program.

**A brief history of Java NIO**

The capability to work with files and byte streams has been available since JDK 1.0. This first version of the java.io package, however, did not fully support character streams. This was added to JDK 1.1 with the reader and writer classes, and the introduction of character encodings. This stream API, however, was blocking and did not scale well. A web server handling thousands of simultaneous requests, for example, needed a separate thread for each connection.

The next milestone in streaming processing was introduced in Java 4 with the new Java NIO API ([JSR 51](https://jcp.org/en/jsr/detail?id=51)). This API promised to provide fast and scalable I/O operations by taking advantage of non-blocking I/O advancements to operating systems. Buffers, Channels and Selectors in the java.nio package made the core of this API. Additionally, it also provided regular expressions for efficient parsing, printf-style formatting, and Charset encoders and decoders

Java 7 enhanced the Java NIO APIs further ([JSR-203](https://www.jcp.org/en/jsr/detail?id=203)) by adding efficient file handling, a new file system interface, asynchronous I/O, support for socket binding and multicasting. The makeover of this API was so substantial that was rebranded as Java NIO.2.

Java 8 enhanced the NIO.2 API further by adding support for the new java.util.stream API.

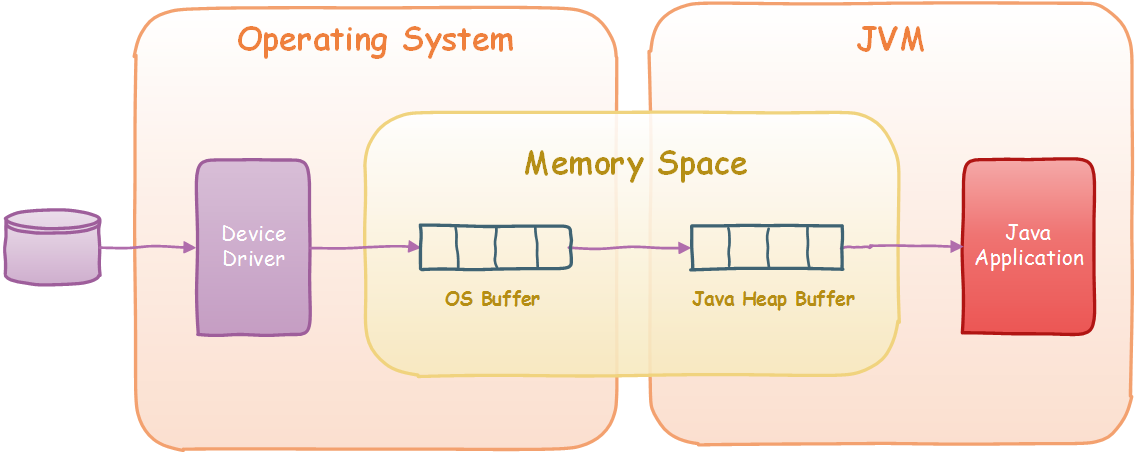
Buffers

Buffers are the workhorse of Java NIO. At its heart, NIO processing is about moving data in and out of buffers. The [java.nio.Buffer](https://docs.oracle.com/javase/8/docs/api/java/nio/Buffer.html) class is the main abstraction, while [java.nio.ByteBuffer](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) is the main implementation class, representing a buffer containing bytes.

There are two types of buffers, direct and non-direct buffers.

A non-direct buffer uses an intermediate buffer managed by the operating system.  This intermediate buffer is needed because the operating system cannot efficiently access buffers in the Java heap.  All objects in the Java Heap are managed by the VM and might change memory location or not be optimally page aligned.

As an example, imagine an application that needs to read data from a file in local disk. It will request to read this data from the operating system, which in turn will instruct the disk controller to copy bytes from storage to the intermediate buffer.  The data from this buffer will then be copied to the Java heap buffer. The application can then access data from this buffer via a [java.nio.ByteBuffer](https://docs.oracle.com/javase/8/docs/api/java/nio/ByteBuffer.html) instance.

Java NIO Non-Direct Buffer

A direct buffer is one that removes this intermediate buffer by moving the underlying buffer memory outside of the Java heap.  As the memory address is fixed for the lifetime of the buffer, and outside of the reach of the Garbage Collector, the operating system can safely access it. While these buffers are more efficient when reading and writing, they are also more expensive to create and destroy.

Channels

A channel represents a connection to a resource that supports I/O operations such as reading and writing from files, network sockets, hardware devices or other applications.  Channels are used in conjunction with buffers to achieve efficient streaming of data. While channels coordinate with the operating system to access the resource, buffers are used to transfer and hold the data.

Channels are defined in the java.nio.channel package, and abstracted by the [java.nio.channel.Channel](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/Channel.html) interface.

These are the three main channel implementations:

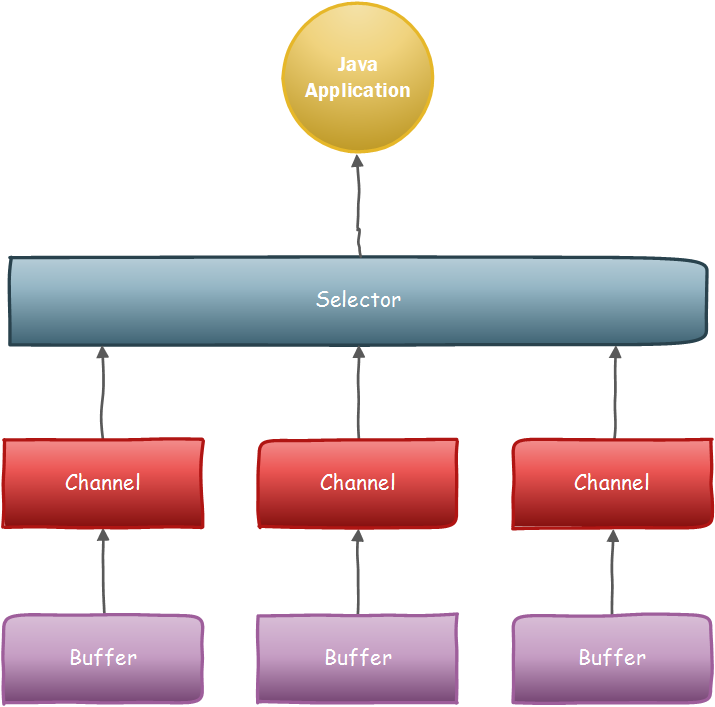
* [nio.channel.FileChannel](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/FileChannel.html): a channel to read and write to files
* [nio.channel.SocketChannel](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/SocketChannel.html): a channel to stream data from and to sockets
* [nio.channel.ServerSocketChannel](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/ServerSocketChannel.html): a channel that can bind to a socket and listen for connections.

Selectors

Selectors enable the efficient monitoring of many Java NIO channels and recognise when they are ready to transmit or receive data.  This allows a single thread to manage large number of channels, thus reducing the system load by reducing thread context overhead and thread memory utilisation.

Selectors make use of a modern operating system feature called non-blocking mode access. Non-blocking access allows the operating system to check the readiness of a stream without blocking. The selector can then instruct the operating system to monitor a group of channels and get notified once one or more are ready for processing.

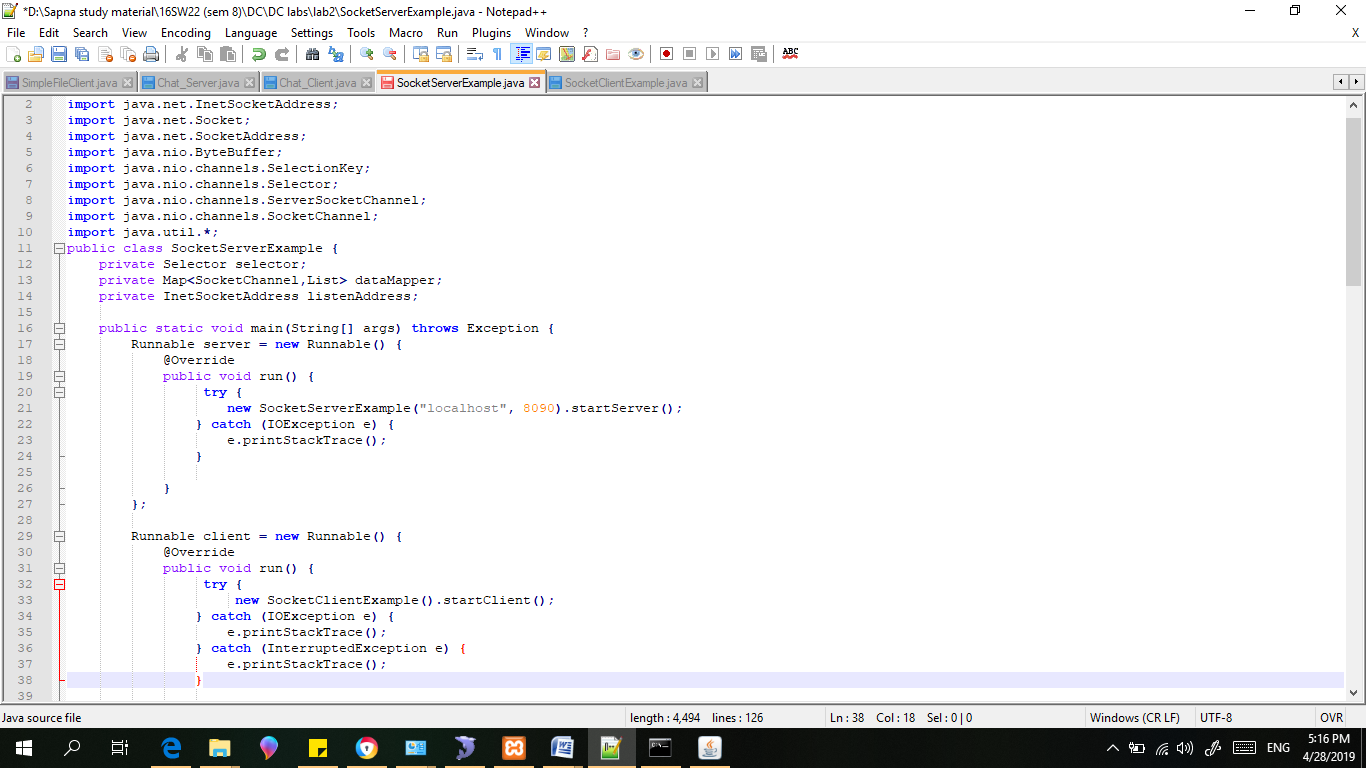
The main Selector class in the [java.nio.channel.Selector](https://docs.oracle.com/javase/8/docs/api/java/nio/channels/Selector.html), and are used mainly for network servers to listen from multiple socket connections.

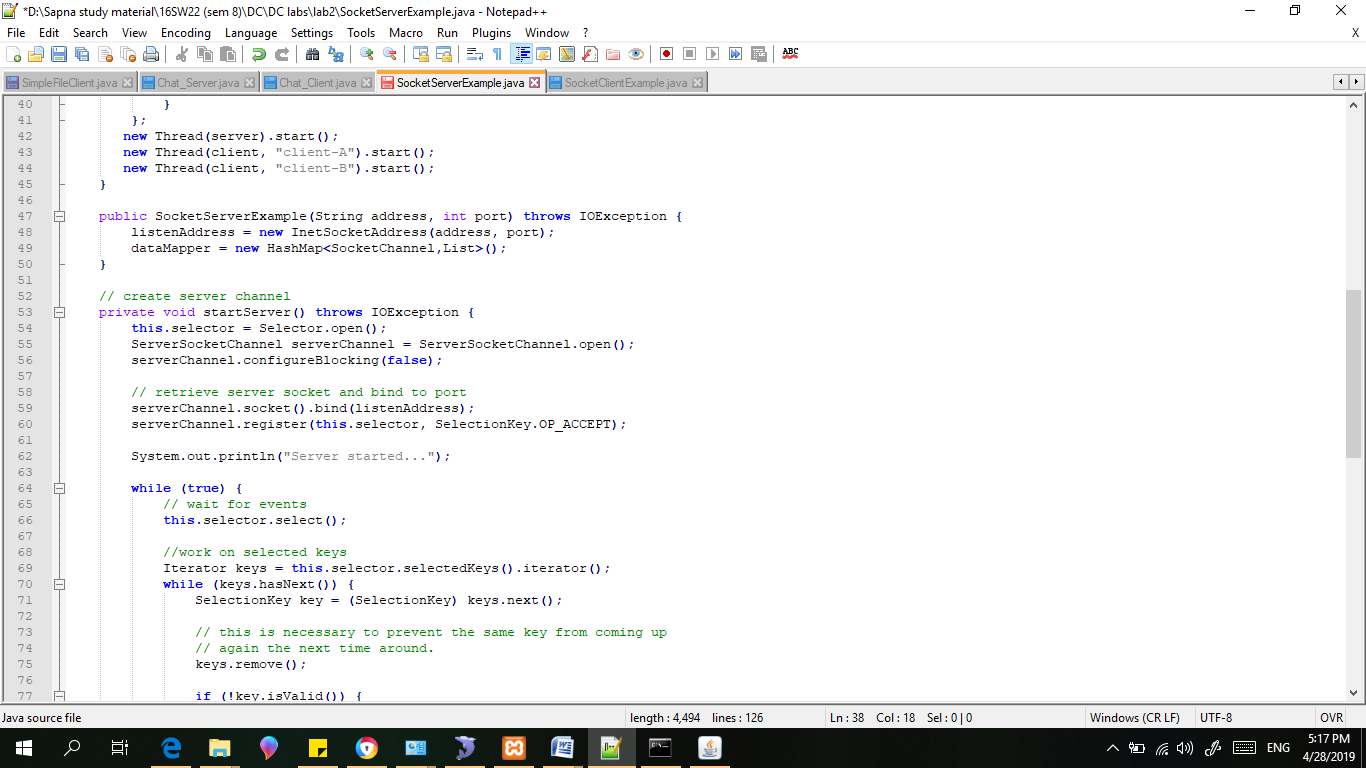
NIO Selector

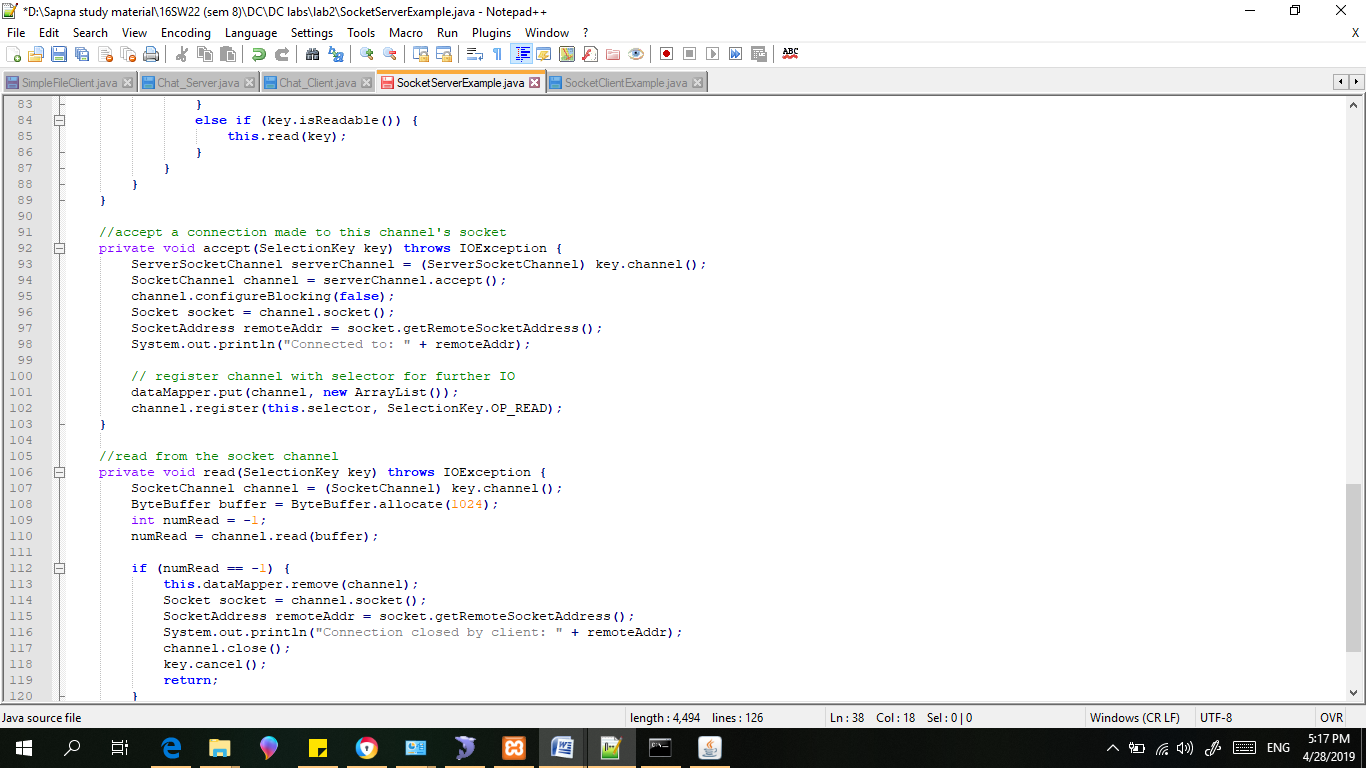
Summary::

This article has presented a brief history of Java NIO and introduced the main concepts of the Java NIO API that enable efficient I/O streaming: Buffers, Channels and Selectors.  Later articles will dive into these concepts and explain how to use the Java NIO for networking programming.

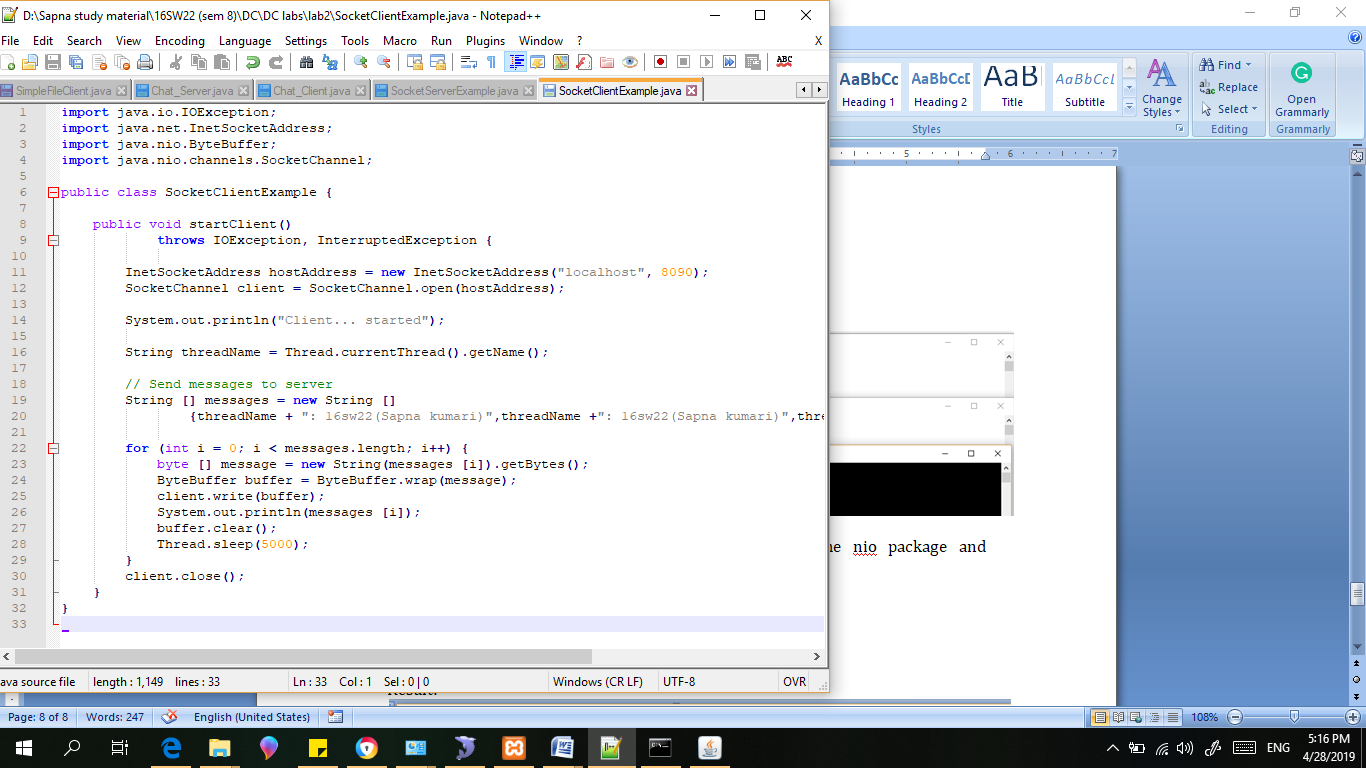
Server:







Client:



Result:

