**Lab no 02**

**Objective: To become familiar with Stream Socket API.**

**Task 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.**

**Requestor:**

import java.net.\*;

import java.io.\*;

//this application requests a connection and sends a message

// using the stream mode socket.

public class ConnectionRequestor {

public static void main(String[] args){

if(args.length!=2){

System.out.println("This program requires two command line arguments");

// the arguments are

//host name of connection acceptor and port number of connection acceptor

}

else{

try{

InetAddress acceptorHost=InetAddress.getByName(args[0]);

int acceptorPort=Integer.parseInt(args[1]);

Socket mySocket=new Socket(acceptorHost,acceptorPort);

System.out.println("Connection request granted.");

InputStream inStream=mySocket.getInputStream();

//create buffered reader object for character mode output

BufferedReader socketInput=new BufferedReader(new InputStreamReader(inStream));

System.out.println("Waiting to read.");

String message=socketInput.readLine();

System.out.println("Message received."+"\t"+message);

System.out.println("Message received."+"\t"+message);

mySocket.close();

System.out.println("data socket closed.");

Thread.sleep(50000);

}

catch(Exception ex){

ex.printStackTrace();

}

}

}

}

**Acceptor:**

import java.net.\*;

import java.io.\*;

public class ConnectionAcceptor {

//Two command line arguments are needed

// port number of the server socket and second is the message to send

public static void main(String[] args){

if(args.length!=2){

System.out.println("This program requires two command line arguments");

}

else{

try{

System.out.println("Aslamoalikum");

int portNo=Integer.parseInt(args[0]);

String message=args[1];

ServerSocket connectionSocket=new ServerSocket(portNo);

System.out.println("now ready to accept a connection");

Socket dataSocket=connectionSocket.accept();

System.out.println("Connection Accepted");

OutputStream outStream=dataSocket.getOutputStream();

//create a print writer for character mode output

PrintWriter socketOutput=new PrintWriter(new OutputStreamWriter(outStream));

//write a message into the data stream

socketOutput.println(message);

//the ensuing flush method ensures that data is written into the data socket before the socket is closed.

socketOutput.flush();

System.out.println("message sent");

dataSocket.close();

System.out.println("data socket closed.");

connectionSocket.close();

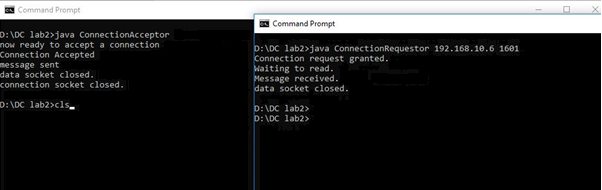
System.out.println("connection socket closed.");

Thread.sleep(50000);

}

catch(Exception ex){ ex.printStackTrace();

} } } }

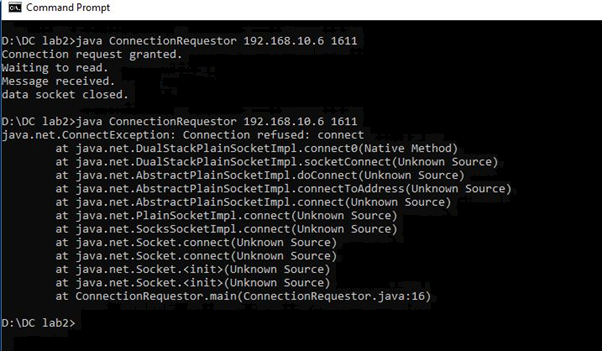


**Description:**

First we start the acceptorclass that accepts the request and responds accordingly and then requestor that request any service. In above case acceptor starts and prepares its service that is a message in this case and then requestor is started in order to make a request that required the name and port number of acceptor in order to establish a connection and get the data or service after it receives the message the acceptor flushes all the data (some like session ends) and close the socket and ends the connection. We can say here the acceptor class is working as a server while requestor is working as receiver.

**Task2:**

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.

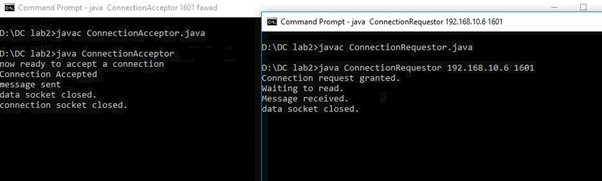


**Description:**

It will create an exception that has no acceptor class to accept the requestor.

**Task 3:**

Add a time delay of 5 seconds in the ConnectionAcceptor 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.



**Task4:**

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

**Client:**

import java.io.\*;

import java.net.\*;

public class ClientChat

{

public static void main(String[] args) throws Exception

{

Socket s = new Socket("127.0.0.1", 3000);

// reading from keyboard (keyRead object)

BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));

// sending to client (pwrite object)

OutputStream ostream = s.getOutputStream();

PrintWriter pwrite = new PrintWriter(ostream, true);

// receiving from server ( receiveRead object)

InputStream istream = s.getInputStream();

BufferedReader receive = new BufferedReader(new InputStreamReader(istream));

System.out.println("Start the chatting ");

String receiveMessage, sendMessage;

while(true)

{

sendMessage = keyRead.readLine(); // keyboard reading

pwrite.println(sendMessage); // sending to server

pwrite.flush(); // flush the data

if((receiveMessage = receive.readLine()) != null) //receive from server

{

System.out.println(receiveMessage); // displaying at DOS prompt

}

}

}

}

**Server:**

import java.net.\*;

import java.io.\*;

public class ServerChat

{

public static void main(String[] args) throws Exception

{

ServerSocket sersock = new ServerSocket(3000);

System.out.println("Server ready for chatting");

Socket s = sersock.accept( );

// reading from keyboard (keyRead object)

BufferedReader keyRead = new BufferedReader(new InputStreamReader(System.in));

// sending to client (pwrite object)

OutputStream ostream = s.getOutputStream();

PrintWriter pwrite = new PrintWriter(ostream, true);

// receiving from server ( receiveRead object)

InputStream istream = s.getInputStream();

BufferedReader receive = new BufferedReader(new InputStreamReader(istream));

String receiveMessage, sendMessage;

while(true)

{

if((receiveMessage = receive.readLine()) != null)

{

System.out.println(receiveMessage);

}

sendMessage = keyRead.readLine();

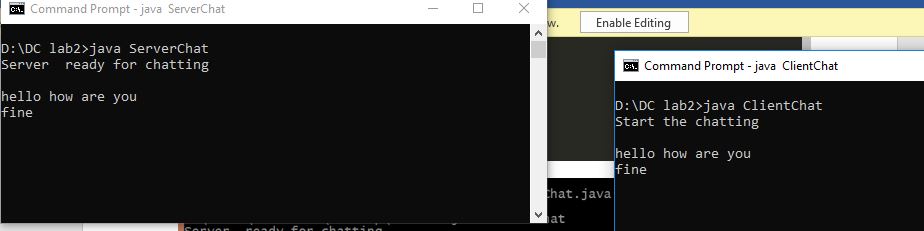
pwrite.println(sendMessage);

pwrite.flush();

}

}

}



**Task 5 :**

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

**Client.java**

import java.io.BufferedOutputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.net.Socket;

public class FileClient {

public final static int SOCKET\_PORT = 13267; // you may change this

public final static String SERVER = "127.0.0.1"; // localhost

public final static String

FILE\_TO\_RECEIVED = "D:/MulticastReceiver.txt"; // you may change this, I give a

// different name because i don't want to

// overwrite the one used by server...

public final static int FILE\_SIZE = 6022386; // file size temporary hard coded

// should bigger than the file to be downloaded

public static void main (String [] args ) throws IOException {

int bytesRead;

int current = 0;

FileOutputStream fos = null;

BufferedOutputStream bos = null;

Socket sock = null;

try {

sock = new Socket(SERVER, SOCKET\_PORT);

System.out.println("Connecting...");

// receive file

byte [] mybytearray = new byte [FILE\_SIZE];

InputStream is = sock.getInputStream();

fos = new FileOutputStream(FILE\_TO\_RECEIVED);

bos = new BufferedOutputStream(fos);

bytesRead = is.read(mybytearray,0,mybytearray.length);

current = bytesRead;

do {

bytesRead =

is.read(mybytearray, current, (mybytearray.length-current));

if(bytesRead >= 0) current += bytesRead;

} while(bytesRead > -1);

bos.write(mybytearray, 0 , current);

bos.flush();

System.out.println("File " + FILE\_TO\_RECEIVED

+ " downloaded (" + current + " bytes read)");

}

finally {

if (fos != null) fos.close();

if (bos != null) bos.close();

if (sock != null) sock.close();

}

}

}

**Server.java**

import java.io.BufferedInputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

public class FileServer {

public final static int SOCKET\_PORT = 13267; // you may change this

public final static String FILE\_TO\_SEND = "D:/MulticastReceiver.txt"; // you may change this

public static void main (String [] args ) throws IOException {

FileInputStream fis = null;

BufferedInputStream bis = null;

OutputStream os = null;

ServerSocket servsock = null;

Socket sock = null;

try {

servsock = new ServerSocket(SOCKET\_PORT);

while (true) {

System.out.println("Waiting...");

try {

sock = servsock.accept();

System.out.println("Accepted connection : " + sock);

// send file

File myFile = new File (FILE\_TO\_SEND);

byte [] mybytearray = new byte [(int)myFile.length()];

fis = new FileInputStream(myFile);

bis = new BufferedInputStream(fis);

bis.read(mybytearray,0,mybytearray.length);

os = sock.getOutputStream();

System.out.println("Sending " + FILE\_TO\_SEND + "(" + mybytearray.length + " bytes)");

os.write(mybytearray,0,mybytearray.length);

os.flush();

System.out.println("Done.");

}

finally {

if (bis != null) bis.close();

if (os != null) os.close();

if (sock!=null) sock.close();

}

}

}

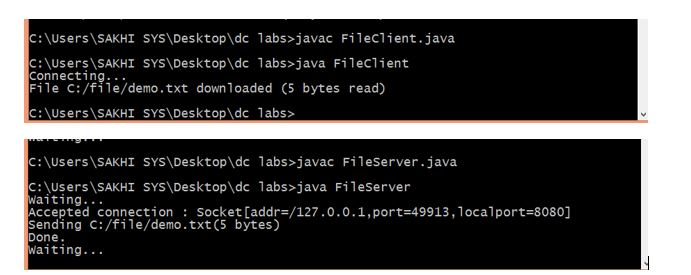
finally {

if (servsock != null) servsock.close();

}

}

}



**Task 6 :**

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

**Server:**

import java.io.IOException;

import java.net.InetSocketAddress;

import java.net.Socket;

import java.net.SocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.\*;

public class ServerNio {

private Selector selector;

private Map<SocketChannel,List> dataMapper;

private InetSocketAddress listenAddress;

public static void main(String[] args) throws Exception {

Runnable server = new Runnable() {

@Override

public void run() {

try {

new ServerNio("localhost", 8090).startServer();

} catch (IOException e) {

e.printStackTrace();

}

}

};

Runnable client = new Runnable() {

@Override

public void run() {

try {

new ClientNio().startClient();

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

new Thread(server).start();

new Thread(client, "client-A").start();

new Thread(client, "client-B").start();

}

public ServerNio(String address, int port) throws IOException {

listenAddress = new InetSocketAddress(address, port);

dataMapper = new HashMap<SocketChannel,List>();

}

// create server channel

private void startServer() throws IOException {

this.selector = Selector.open();

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

// retrieve server socket and bind to port

serverChannel.socket().bind(listenAddress);

serverChannel.register(this.selector, SelectionKey.OP\_ACCEPT);

System.out.println("Server started...");

while (true) {

// wait for events

this.selector.select();

//work on selected keys

Iterator keys = this.selector.selectedKeys().iterator();

while (keys.hasNext()) {

SelectionKey key = (SelectionKey) keys.next();

// this is necessary to prevent the same key from coming up

// again the next time around.

keys.remove();

if (!key.isValid()) {

continue;

}

if (key.isAcceptable()) {

this.accept(key);

}

else if (key.isReadable()) {

this.read(key);

}

}

}

}

//accept a connection made to this channel's socket

private void accept(SelectionKey key) throws IOException {

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

SocketChannel channel = serverChannel.accept();

channel.configureBlocking(false);

Socket socket = channel.socket();

SocketAddress remoteAddr = socket.getRemoteSocketAddress();

System.out.println("Connected to: " + remoteAddr);

// register channel with selector for further IO

dataMapper.put(channel, new ArrayList());

channel.register(this.selector, SelectionKey.OP\_READ);

}

//read from the socket channel

private void read(SelectionKey key) throws IOException {

SocketChannel channel = (SocketChannel) key.channel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

int numRead = -1;

numRead = channel.read(buffer);

if (numRead == -1) {

this.dataMapper.remove(channel);

Socket socket = channel.socket();

SocketAddress remoteAddr = socket.getRemoteSocketAddress();

System.out.println("Connection closed by client: " + remoteAddr);

channel.close();

key.cancel();

return;

}

byte[] data = new byte[numRead];

System.arraycopy(buffer.array(), 0, data, 0, numRead);

System.out.println("Got: " + new String(data));

}

}

**Client:**

import java.io.IOException;

import java.net.InetSocketAddress;

import java.net.Socket;

import java.net.SocketAddress;

import java.nio.ByteBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.\*;

public class ServerNio {

private Selector selector;

private Map<SocketChannel,List> dataMapper;

private InetSocketAddress listenAddress;

public static void main(String[] args) throws Exception {

Runnable server = new Runnable() {

@Override

public void run() {

try {

new ServerNio("localhost", 8090).startServer();

} catch (IOException e) {

e.printStackTrace();

}

}

};

Runnable client = new Runnable() {

@Override

public void run() {

try {

new ClientNio().startClient();

} catch (IOException e) {

e.printStackTrace();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

new Thread(server).start();

new Thread(client, "client-A").start();

new Thread(client, "client-B").start();

}

public ServerNio(String address, int port) throws IOException {

listenAddress = new InetSocketAddress(address, port);

dataMapper = new HashMap<SocketChannel,List>();

}

// create server channel

private void startServer() throws IOException {

this.selector = Selector.open();

ServerSocketChannel serverChannel = ServerSocketChannel.open();

serverChannel.configureBlocking(false);

// retrieve server socket and bind to port

serverChannel.socket().bind(listenAddress);

serverChannel.register(this.selector, SelectionKey.OP\_ACCEPT);

System.out.println("Server started...");

while (true) {

// wait for events

this.selector.select();

//work on selected keys

Iterator keys = this.selector.selectedKeys().iterator();

while (keys.hasNext()) {

SelectionKey key = (SelectionKey) keys.next();

// this is necessary to prevent the same key from coming up

// again the next time around.

keys.remove();

if (!key.isValid()) {

continue;

}

if (key.isAcceptable()) {

this.accept(key);

}

else if (key.isReadable()) {

this.read(key);

}

}

}

}

//accept a connection made to this channel's socket

private void accept(SelectionKey key) throws IOException {

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

SocketChannel channel = serverChannel.accept();

channel.configureBlocking(false);

Socket socket = channel.socket();

SocketAddress remoteAddr = socket.getRemoteSocketAddress();

System.out.println("Connected to: " + remoteAddr);

// register channel with selector for further IO

dataMapper.put(channel, new ArrayList());

channel.register(this.selector, SelectionKey.OP\_READ);

}

//read from the socket channel

private void read(SelectionKey key) throws IOException {

SocketChannel channel = (SocketChannel) key.channel();

ByteBuffer buffer = ByteBuffer.allocate(1024);

int numRead = -1;

numRead = channel.read(buffer);

if (numRead == -1) {

this.dataMapper.remove(channel);

Socket socket = channel.socket();

SocketAddress remoteAddr = socket.getRemoteSocketAddress();

System.out.println("Connection closed by client: " + remoteAddr);

channel.close();

key.cancel();

return;

}

byte[] data = new byte[numRead];

System.arraycopy(buffer.array(), 0, data, 0, numRead);

System.out.println("Got: " + new String(data));

}

}

**Output:**

