Chapter (5)

UDP

- Unreliable Datagram Protocol
- Packet Oriented, not stream oriented like TCP/IP
- Much faster but no error correction
- Must fit data into packets of about 8K or less

* Header (For packet, Segments ...) It is important and have ~~the~~ information we need it without errors :-

header → using TCP → check!

content → UDP → no check!

→ Some of applications (Real time apps)

Advantages of UDP

- UDP communication can be more efficient than guaranteed-delivery data streams.
- Unlike TCP streams, which establish a connection, UDP causes fewer overheads.
- Real-time applications that demand up-to-the-second or better performance may be candidates for UDP, as there are fewer delays due to the error checking and flow control of TCP.
- UDP sockets can receive data from more than one host machine.

- The best way to describe UDP → Having Simulator.

↓

Simulating Reality

The UDP Classes

- Java's support for UDP is contained in two classes:
  - `java.net.DatagramSocket`
  - `java.net.DatagramPacket`

  } to implement or simulate the nature of UDP.

- A datagram socket is used to send and receive datagram packets.

IP addresses

java.net.DatagramPacket

- The DatagramPacket class represents a data packet intended for transmission using the User Datagram Protocol

- Packets are containers for a small sequence of bytes, and include addressing information such as an IP address and a port.

Creating a DatagramPacket

- There are two reasons to create a new DatagramPacket:

1. To send data to a remote machine using UDP
2. To receive data sent by a remote machine using UDP

- **Constructors:** The choice of which DatagramPacket constructor to use is determined by its intended purpose. Either constructor requires the specification of a byte array, which will be used to store the UDP packet contents, and the length of the data packet.

Note about Constructor For Sending ← port number in transport layer, but here determine in datagram Packet because it's simulator not Real.

Two DatagramPacket Constructors

Very important

- Constructors for *receiving* datagram

① public **DatagramPacket**(byte[] buffer, int length)

② public **DatagramPacket**(byte[] buffer, int offset, int length)

  - **Example:**

```
byte[ ] buffer = new byte[8192];
DatagramPacket dp = new DatagramPacket(buffer, buffer.length);
```

- Constructors for *sending* datagram

public **DatagramPacket**(byte[] data, int length, InetAddress remote, int port)

public **DatagramPacket**(byte[] data, int offset, int length, InetAddress remote, int port)

  - **Example:**

```
InetAddress addr = InetAddress.getByName("192.168.0.1");
DatagramPacket packet = new DatagramPacket ( new byte[128], 128, addr, 2000);
```

DatagramPacket (Example)

```
String s = "This is a test";
byte[] data = s.getBytes();
try {
 InetAddress ia = InetAddress.getByName("www.ju.edu.jo");
  int port = 7;
 DatagramPacket dp = new DatagramPacket(data, data.length,
   ia, port);
 }
  catch (UnknownHostException e) {
   System.err.println(e);
  }
```


Using a DatagramPacket

- The DatagramPacket class provides some important methods that allow the remote address, remote port, data (as a byte array), and length of the packet to be retrieved.
  - As of JDK1.1, there are also methods to modify these, via a corresponding set method.
  - This means that a received packet can be reused. For example, a packet's contents can be replaced and then sent back to the sender.

DatagramPacket: Methods

- **InetAddress getAddress()**— returns the IP address from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination IP address.
- **byte[] getData()**— returns the contents of the DatagramPacket, represented as an array of bytes.
- **int getLength()**— returns the length of the data stored in a DatagramPacket. This can be less than the actual size of the data buffer.

DatagramPacket: Methods Cont.

- **int getPort()**— returns the port number from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination port number.
- **void setAddress(InetAddress addr)**— assigns a new destination address to a DatagramPacket.
- **void setData(byte[] buffer)**— assigns a new data buffer to the DatagramPacket. Remember to make the buffer long enough, to prevent data loss.

java.net.DatagramSocket Cont.

- Since UDP packets do not guarantee delivery, this can cause an application to stall if the sender does not resubmit packets.
- There is no distinction between a UDP socket and a UDP server socket.
- Also unlike TCP sockets, a **DatagramSocket** can send to multiple, different addresses.
- The address to which data goes is stored in the packet not in the socket.

ممكن سؤال امتحان ← data راح توصل؟ كيف بتحقق behavior

Using ↳ Constructor for recieving and first one or second ↓ below ()

17

Creating a DatagramSocket

Sender Reciever

- A DatagramSocket can be used to both send and receive packets. Constructors:- ← يقدر أعمل packet و لاعادته لأي port راجعه
  - public DatagramSocket() throws SocketException ← ولا لأي address (ما بتوصل)
  - public DatagramSocket(int port) throws SocketException
  - public DatagramSocket(int port, InetAddress laddr) throws SocketException ← for destination
    - The first is for *client datagram sockets*; that is sockets that send datagrams before receiving any.
    - The second two are for *server datagram sockets* since they specify the port and optionally the IP address of the socket

18

Using a DatagramSocket

- DatagramSocket is used to receive incoming UDP packets and to send outgoing UDP packets.
- It provides methods to send and receive packets, as well as to specify a timeout value when nonblocking I/O is being used, to inspect and modify maximum UDP packet sizes, and to close the socket.

Use: (void connect) ← connection انه طافي Check

* ازا بدنا نعمل
* الداتا الـ (Sending) بدها داخل (buffer) فارغ ازا وصلت اوك

(connection loss)

19

DatagramSocket: Methods

- **void close()**— closes a socket, and unbinds it from the local port. (to check if it is UDP)
- **void connect(InetAddress remote_addr int remote_port)**— restricts access to the specified remote address and port. The designation is a misnomer, as UDP doesn't actually create a "connection" between one machine and another. However, if this method is used, it causes exceptions to be thrown if an attempt is made to send packets to, or read packets from, any other host and port than those specified.
- **void disconnect()**— disconnects the DatagramSocket and removes any restrictions imposed on it by an earlier connect operation.

20

Listening for UDP Packets Cont.

- When an application wishes to read UDP packets, it calls the DatagramSocket receive method, which copies a UDP packet into the specified DatagramPacket. The contents of the DatagramPacket are processed, and the process is repeated as needed.

Listening for UDP Packets Cont.

The following code snippet illustrates this process:

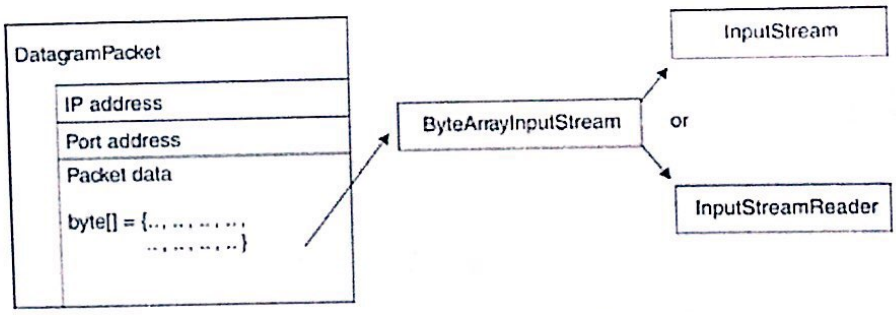
```
DatagramPacket packet = new DatagramPacket (new
byte[256], 256);
DatagramSocket socket = new
DatagramSocket(2000);
boolean finished = false;
while (! finished )
{
  socket.receive (packet);
  // process the packet
}
socket.close();
```

Listening for UDP Packets Cont.

- When processing the packet, the application must work directly with an array of bytes. If, however, your application is better suited to reading text, you can use classes from the Java I/O package to convert between a byte array and another type of stream or reader.
- By hooking a ByteArrayInputStream to the contents of a datagram and then to another type of InputStream or an InputStreamReader, you can access the contents of UDP packets relatively easily

Listening for UDP Packets Cont.

Figure 5-4. Reading from a UDP packet is simplified by applying input streams.



- For example, to hook up a DataInputStream to the contents of a DatagramPacket, the following code can be used:

```
ByteArrayInputStream bin = new
ByteArrayInputStream(packet.getData() );
DataInputStream din = new DataInputStream (bin);
// Read the contents of the UDP packet
.....
```

Sending UDP packets Cont.

- The following code snippet illustrates this process:
- DatagramSocket socket = new DatagramSocket(2000);
- DatagramPacket packet = new DatagramPacket (new byte[256], 256);
- packet.setAddress (InetAddress.getByName (somehost));
- packet.setPort (2000);
- boolean finished = false;
- while !finished)
- {
- // Write data to packet buffer
- .....
- socket.send (packet);
- // Do something else, like read other packets, or check to
- // see if no more packets to send
- .....
- }
- socket.close();

Disadvantages of UDP

- **Lack of Guaranteed Delivery**
- **Lack of Guaranteed Packet Sequencing**
- **Lack of Flow Control**

*Very important Example

User Datagram Protocol Example

A UDP Server:

```
//
// UDPlistener.java based on code in Dietel and Dietel
// Server reads a datagram packet from a client and responds
// with a datagram packet.
//
import java.io.*;
import java.net.*;

public class UDPlistener {
private DatagramSocket mySocket;
private DatagramPacket receivePacket;
private DatagramPacket sendPacket;
public UDPlistener() {
System.out.println(" \nListening very carefully!\n\n");
try {
mySocket = new DatagramSocket( 4567);
}
catch( SocketException sE) { // error so bad server dies!
System.err.println( "Could not open a datagram socket");
sE.printStackTrace();
System.exit( 1);
}
} // constructor
//
// Wait for packets from client, display them and respond.
//
public void waitForPackets() {
do { // wait for 'END' from client!!
// receive a packet, display it and echo
try {
// set up a packet ready to receive the msg
byte data[] = new byte[ 100];
receivePacket = new DatagramPacket( data, data.length);
// now wait for the packet
mySocket.receive( receivePacket);
displayPacket();
sendPacketToClient( "I see you sent: ");
}
catch( IOException ioE) {
System.err.println( "Some receive error!");
System.err.println( ioE.toString());
}
} while( !(receivePacket.toString()).equals( "END"));
} // waitForPacket()
```

Socket ? → port number

why 2 packets ? Server will recieve packet & resend it to client.

define Objects. — Constructor — Create object — port — msg to start listening

Server ⟷ client: Server لحين ما يبعت ال (end) بيستقبل كو

length → (100) بيستقبل كو ١٠٠ واذا زاد عن ١٠٠ رح يعطي (exception)

معرفة globaly. — إلي وصلني

شو ممكن بالامتحان ؟

١. تغيير رقم (port) إلي برسل وبستقبل منه [للواحد رقم] ، ما رح يطلع (exception) بس ما رح يوصل اشي.

٢. (connect function) : ما بيشتغل بال (UDP)

٣. Server ← buffer ← client

لـ يدل انه (connectionless)

```
//
// Display the received packet
//
public void displayPacket() {

System.out.println( "Packet received: ");
System.out.println( "\tFrom host: " + receivePacket.getAddress());
System.out.println( "\tHost port: " + receivePacket.getPort());
System.out.println( "\tLength: " + receivePacket.getLength());
System.out.println( "\tContaining: ");
System.out.println( "\t\t" + new String( receivePacket.getData(), 0,
receivePacket.getLength()));
} // displayPacket()
//
// echo packet
//
private void sendPacketToClient( String msg) throws IOException {
// construct the packet to send
String sMsg = msg + " ";
sMsg += new String( receivePacket.getData(), 0, receivePacket.getLength());
// create the packet to send
sendPacket = new DatagramPacket( sMsg.getBytes(), sMsg.length(),
receivePacket.getAddress(), receivePacket.getPort());
// send the packet
mySocket.send( sendPacket);
System.out.println( "Packet sent back");
} // sendPacketToClient()
//
// execute the application
//
public static void main( String args[]) {
UDPlistener app = new UDPlistener();
app.waitForPackets();
} // main
} // class UDPlistener
```

A UDP Client:

```
//
// UDPclient.java based on code in Dietel and Dietel
// Sends a datagram packet to a server and
// waits for a reply.
//
import java.io.*;
import java.net.*;

import java.util.Scanner;
public class UDPclient {
private DatagramSocket mySocket;
private DatagramPacket sendPacket;
private DatagramPacket receivePacket;
public UDPclient() { // constructor
try {
mySocket = new DatagramSocket();
```

```
}
catch ( SocketException sockE) {
System.err.println( "Problem opening datagram socket!");
sockE.printStackTrace();
System.exit( 1);
}
} // constructor
//
// Send packets, wait for responses from server and display them.
//
public void sendPackets() {
Scanner kbd = new Scanner( System.in);
String message = null;
byte rdata[] = new byte[ 200]; // to hold response
do{ // wait for 'END'
try {
// set up datagram packet.
System.out.println( "\nEnter a message to send (END to disconnect): ");
// next line of input from keybd.
message = kbd.nextLine(); // what happens if we use next()?
if( !message.equals( "END")) { // if more messages to send
// set up the packet
byte sdata[] = message.getBytes();
sendPacket = new DatagramPacket( sdata, sdata.length,
InetAddress.getLocalHost(), 4567);
mySocket.send( sendPacket);
// wait for response!
receivePacket = new DatagramPacket( rdata, rdata.length);
mySocket.receive( receivePacket);
displayPacket();
} // if msgs to send
}
catch( IOException ioE) {
System.err.println( "Receive error!");
ioE.printStackTrace();
}
} while( !message.equals( "END"));
System.out.println( "Client finished");
mySocket.close();
} // sendPackets()
//
// display the received packet
//
public void displayPacket() {
System.out.println( "Packet received: ");
System.out.println( "\tFrom host: " + receivePacket.getAddress());
System.out.println( "\tHost port: " + receivePacket.getPort());
System.out.println( "\tLength: " + receivePacket.getLength());
System.out.println( "\tContaining: ");
System.out.println( "\t\t" + new String( receivePacket.getData(), 0,
receivePacket.getLength()));
} // displayPacket()

//
// start the application!
//
public static void main( String args[]) {
```

Networks and Internet Programming (0907522)

CHAPTER 6

Transmission Control Protocol (TCP)

Instructor: Dr. Khalid A. Darabkh

* Tcp
1
- Reliable
- Flow & Error Control.
- End to End protocol.

* Check damage & Drop not only Drop like UDP

*Datagrams

- Before data is sent across the Internet from one host to another using TCP/IP, it is split into packets of varying but finite size called *datagrams*.
- Datagrams range in size from a few dozen bytes to about 60,000 bytes.
- Packets larger than this, and often smaller than this, must be split into smaller pieces before they can be transmitted.

2

Packets Allow Error Correction

- If one packet is lost, it can be retransmitted without requiring redelivery of all other packets.
- If packets arrive out of order they can be reordered at the receiving end of the connection.

* Datagram → when Using UDP
* Packet → " " TCP
  ↳ Using them to Split msg into pieces.

Abstraction

- Datagrams are mostly hidden from the Java programmer.
- The host's native networking software transparently splits data into packets on the sending end of a connection, and then reassembles packets on the receiving end.
- Instead, the Java programmer is presented with a higher level abstraction called a *socket*.

Sockets

- A socket is a reliable connection for the transmission of data between two hosts.
- Sockets isolate programmers from the details of packet encodings, lost and retransmitted packets, and packets that arrive out of order.
- Sockets are more likely to throw IOExceptions

Socket Class

- The Socket class represents client sockets, and is a communication channel between two TCP communications ports belonging to one or two machines. A socket may connect to a port on the local system, avoiding the need for a second machine, but most network software will usually involve two machines. TCP sockets can't communicate with more than two machines

Socket Operations

- There are four fundamental operations a socket performs. These are:
  1. Connect to a remote machine
  2. Send data
  3. Receive data
  4. Close the connection
- A socket may not be connected to more than one host at a time.

How java programs use client sockets?

- Create new socket
- Socket attempts to connect to the remote host
- Once connection is established, get input/output stream: *Full duplex* connection
- When transmission is complete, close connection

The java.net.Socket class

- The `java.net.Socket` class allows you to create socket objects that perform all four fundamental socket operations.
- You can connect to remote machines; you can send data; you can receive data; you can close the connection.
- Each `Socket` object is associated with exactly one remote host. To connect to a different host, you must create a new `Socket` object.

∴ every Socket for every Connect.

Constructing a Socket

- Connection is accomplished through the constructors.
  - public Socket(InetAddress address, int port) throws IOException
  - public Socket(InetAddress address, int port, InetAddress localAddr, int localPort) throws IOException
  - public Socket(String host, int port) throws UnknownHostException, IOException
  - public Socket(String host, int port, InetAddress localAddr, int localPort) throws UnknownHostException, IOException

src

(i) (ii) - multihomed machine

public Socket(String host, int port) throws UnknownHostException, IOException

```
try {
   Socket yahoo = new
                     Socket("www.yahoo.com", 80);
   // send and receive data... }

catch (UnknownHostException ex) {
   System.err.println(ex); }

catch (IOException ex) {
   System.err.println(ex);
}
```

String; Port number; From Data Streams.

15

{Very important}

- You cannot just connect to any port on any host. The remote host must actually be listening for connections on that port.

مستحيل تعمل (Connect) على اي شي مش لازم يكون On (listening) ... TCP.

- You can use the constructors to determine which ports on a host are listening for connections.

public Socket(InetAddress host, int port) throws IOException

```
try {
 InetAddress yahoo =
            InetAddress.getByName(www.yahoo.com);

Socket ySocket = new Socket(yahoo, 80);
    // send and receive data... }

catch (UnknownHostException ex) {
   System.err.println(ex); }

catch (IOException ex) {
   System.err.println(ex);
}
```

Picking an IP address

- There are constructors also specify the host and port you're connecting from.
- On a system with multiple IP addresses, like many web servers, this allows you to pick your network interface and IP address.

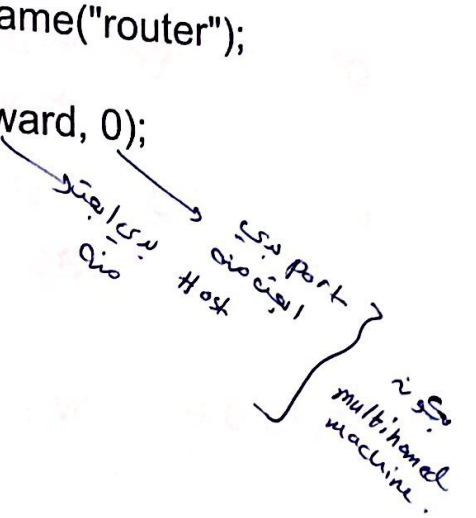
public Socket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException

```
try {
InetAddress inward = InetAddress.getByName("router");

Socket socket = new Socket("mail", 25, inward, 0);
  // work with the sockets... }

catch (UnknownHostException ex) {
  System.err.println(ex); }

catch (IOException ex) {
  System.err.println(ex); }
```



public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException

```
try {
InetAddress inward = InetAddress.getByName("router");

InetAddress mail = InetAddress.getByName("mail");

Socket socket = new Socket(mail, 25, inward, 0);
// work with the sockets... }

catch (UnknownHostException ex) {
System.err.println(ex); }

catch (IOException ex) { System.err.println(ex); }
```

...nding and Receiving Data

- Data is sent and received with output and input streams.
- There are methods to get an input stream for a socket and an output stream for the socket.

```
public InputStream getInputStream() throws
IOException: returns an input stream, which reads from the
application this socket is connected to.
public OutputStream getOutputStream() throws
IOException: returns an output stream, which writes to the
application that this socket is connected to.
```

- There's also a method to close a socket.

```
public void close() throws IOException
```

Reading Input from a Socket

- The `getInputStream()` method returns an InputStream which reads data from the socket.
- You can use all the normal methods of the `InputStream` class to read this data.
- Most of the time the input stream is chained to some other input stream or reader *(ex.DataInputStream or InputStreamReader)* object to more easily handle the data. Chaining the input to buffers *(BufferedInputStream or BufferedReader)* for performance reasons

Example

- The following code fragment connects to the daytime server on port 13 of metalab.unc.edu, and displays the data it sends.

```
try {
  Socket s = new Socket("metalab.unc.edu", 13);
  InputStream in = s.getInputStream();
  InputStreamReader isr = new InputStreamReader(in);
  BufferedReader br = new BufferedReader(isr);
  String theTime = br.readLine();
  System.out.println(theTime);
}
catch (IOException e) {
  return (0);
}
```

Writing Output to a Socket

- The `getOutputStream()` method returns an output stream which writes data from the application to the other end of the socket.

- Most of the time you'll chain the raw output stream to some other output stream or writer class *(ex. DataOutputStream or OutputStreamWriter)* to more easily handle the data.

Socket Options (attributes)

(data)

- SO_LINGER
- SO_TIMEOUT } available in all Versions.
- SO_SNDBUF (Java 1.2 and later)
- SO_RCVBUF (Java 1.2 and later) to 1.12
- SO_KEEPALIVE (Java 1.3 and later)

* every attribute → Set.
→ get.

Other Important Methods

How long is gonna Stay?

- boolean getKeepAlive() throws java.net.SocketException— returns the state of the SO_KEEPALIVE socket option.
- void setKeepAlive(boolean onFlag) throws java.net.SocketException— enables or disables the SO_KEEPALIVE socket option.

→ ON or off

كم بدك تضل alive.

1. connect
2. Send

- int getReceiveBufferSize() throws java.net.SocketException— returns the receive buffer size used by the socket, determined by the value of the SO_RCVBUF socket option.

→ not as buffer in UDP

- void setReceiveBufferSize(int size) throws java.net.SocketException— modifies the value of the SO_RCVBUF socket option, which recommends a buffer size for the operating system's network code to use for receiving incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a BufferedInputStream or a BufferedReader.

→ There is handshaking

Other Important Methods Cont.

- must be listening first → not as buffer in UDP.

- int getSendBufferSize() throws java.net.SocketException— returns the send buffer size used by the socket, determined by the value of the SO_SNDBUF socket option.
- void setSendBufferSize(int size) throws java.net.SocketException— modifies the value of the SO_SNDBUF socket option, which recommends a buffer size for the operating system's network code to use for sending incoming data. Not every system will support this functionality or allows absolute control over this feature. If you want to buffer incoming data, you're advised to instead use a BufferedOutputStream or a BufferedWriter.
- int getSoLinger() throws java.net.SocketException— returns the value of the SO_LINGER socket option, which controls how long unsent data will be queued when a connection is terminated. (in Sec) → queue
- void setSoLinger(boolean onFlag, int duration) throws java.net.SocketException java.lang.IllegalArgumentException— enables or disables the SO_LINGER socket option (according to the value of the onFlag boolean parameter), and specifies a duration in seconds. If a negative value is specified, an exception is thrown.

Other Important Methods Cont.

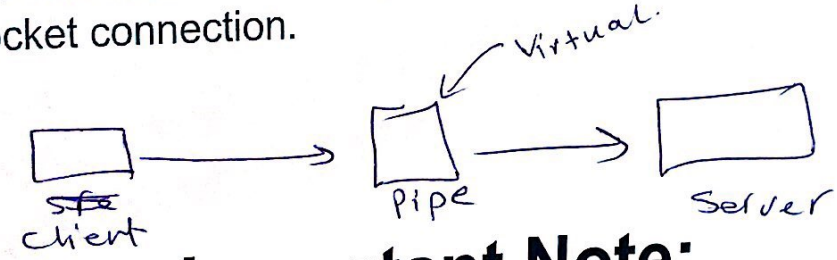
- int getSoTimeout() throws java.net.SocketException— returns the value of the SO_TIMEOUT socket option, which controls how many milliseconds a read operation will block for. If a value of 0 is returned, the timer is disabled and a thread will block indefinitely (until data is available or the stream is terminated). → in ms
- void setSoTimeout(int duration) throws java.net.SocketException— modifies the value of the SO_TIMEOUT socket option, which controls how long (in milliseconds) a read operation will block. A value of zero disables timeouts, and blocks indefinitely. If a timeout does occur, a java.io.IOInterruptedException is thrown whenever a read operation occurs on the socket's input stream. This is distinct from the internal TCP timer, which triggers a resend of unacknowledged datagram packets.
- void shutdownInput() throws java.io.IOException— closes the input stream associated with this socket and discards any further information that is sent. Further reads to the input stream will encounter the end of the stream marker.
- void shutdownOutput() throws java.io.IOException— closes the output stream associated with this socket. Any data previously written, but not yet sent, will be flushed, followed by a TCP connection-termination sequence, which notifies the application that no more data will be available (and in the case of a Java application, that the end of the stream has been reached). Further writes to the socket will cause an IOException to be thrown.

like close input and output stream

Closing the Socket

- Till now it is assumed that sockets close on their own
- Socket closes automatically when one of its two streams closes, the program ends or it's garbage collected.
- To close a socket, the following should be called

public void close( ) throws IOException: Closes the socket connection. Closing a connect may or may not allow remaining data to be sent, depending on the value of the SO_LINGER socket option. Developers are advised to flush any output streams before closing a socket connection.

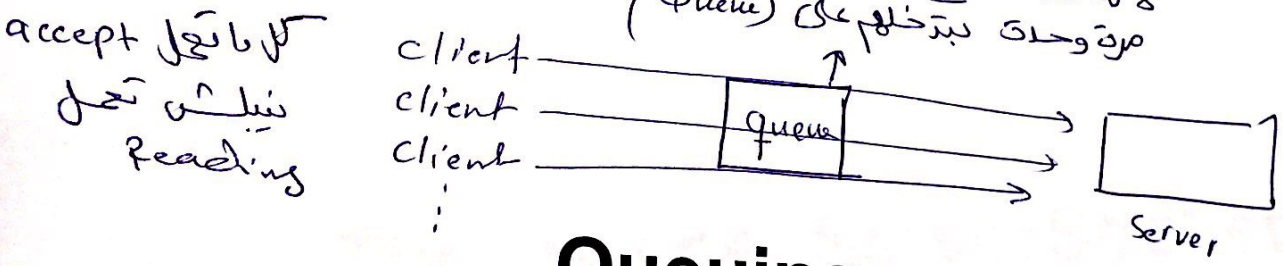


Important Note:

- From the application's perspective, the TCP connection is a direct virtual pipe between the client's socket and the server's connection socket. The client process can send arbitrary bytes into its socket; TCP guarantees that the server process will receive (through the connection socket) each byte in the order sent. Furthermore, just as people can go in and out the same door, the client process can also receive bytes from its socket and the server process can also send bytes into its connection socket.

Multiple Clients

- Multiple clients can connect to the same port on the server at the same time.
- A server may choose to accept only one connection at any given moment
- Incoming data is distinguished by the port to which it is addressed and the client host and port from which it came.
- The server can tell for which service (like http or ftp) the data is intended by inspecting the port.
- It can tell which open socket on that service the data is intended for by looking at the client address and port stored with the data.



Queuing

- Incoming connections are stored in a queue until the server can accept them.
- On most systems the default queue length is between 5 and 50.
- Once the queue fills up, further incoming connections are refused until space in the queue opens up.

The java.net.ServerSocket Class: Constructors

- **ServerSocket(int port)** throws java.io.IOException, java.lang. SecurityException— binds the server socket to the specified port number, so that remote clients may locate the TCP service. If a value of zero is passed, any free port will be used—however, clients will be unable to access the service unless notified somehow of the port number. By default, the queue size is set to 50, but an alternate constructor is provided that allows modification of this setting. If the port is already bound, or security restrictions (such as security polices or operating system restrictions on well-known ports) prevent access, an exception is thrown.

The java.net.ServerSocket Class: Constructors Cont.

- **ServerSocket(int port, int numberOfClients)** throws java.io. IOException, java.lang.SecurityException— binds the server socket to the specified port number and allocates sufficient space to the queue to support the specified number of client sockets. This is an overloaded version of the ServerSocket(int port) constructor, and if the port is already bound or security restrictions prevent access, an exception is thrown.

The java.net.ServerSocket Class: Methods Cont.

- **void close()** throws java.io.IOException— closes the server socket, which unbinds the TCP port and allows other services to use it.
- **InetAddress getInetAddress()**— returns the address of the server socket, which may be different from the local address in the case of a multihomed machine (i.e., a machine whose localhost is known by two or more IP addresses).
- **int getLocalPort()**— returns the port number to which the server socket is bound.

The java.net.ServerSocket Class: Methods Cont.

- **int getSoTimeout()** throws java.io.IOException— returns the value of the timeout socket option, which determines how many milliseconds an accept() operation can block for. If a value of zero is returned, the accept operation blocks indefinitely.
- **void setSoTimeout(int timeout)** throws java.net.SocketException— assigns a timeout value (specified in milliseconds) for the blocking accept() operation. If a value of zero is specified, timeouts are disabled and the operation will block indefinitely. Providing timeouts are enabled, however, whenever the accept() method is called a timer starts. When the timer expires, a java.io.InterruptedIOException is thrown, which allows a server to then take further actions.

↳ blocking Function (must be connect first).

- The `accept()` and `close()` methods provide the basic functionality of a server socket.
  - **public Socket accept() throws IOException**
  - **public void close() throws IOException**

- A server socket can't be reopened after it's closed

49

* flush → any thing in buffer will Send.
* if you don't close → wait for (timeout) to finish then will close :

Reading Data with a ServerSocket

- `ServerSocket` objects use their `accept()` method to connect to a client.
  - public Socket accept() throws IOException
- There are no `getInputStream()` or `getOutputStream()` methods for `ServerSocket`.
- `accept()` returns a Socket object, and its `getInputStream()` and `getOutputStream()` methods provide streams.

50

Example

```
try {
 ServerSocket ss = new ServerSocket(2345);
 Socket s = ss.accept();
 PrintWriter pw = new
  PrintWriter(s.getOutputStream());
 pw.print("Hello There!\r\n");
 pw.print("Goodbye now.\r\n");
 s.close();
}
catch (IOException e) {
 System.err.println(e);
}
```

(to use print .but print writer)

(timeout) Connection [accept] → listening

Server Socket Options

- SO_TIMEOUT
  - public void setSoTimeout(int timeout) throws SocketException
  - public int getSoTimeout( ) throws IOException
- SO_RCVBUF →
  - public void **setReceiveBufferSize**(int size) throws SocketException
  - public int **getReceiveBufferSize**() throws SocketException

{ Pipe بكتب على client
Pipe من بيقرأ ← Server }

Creating a TCP Client

Code for DaytimeClient:

```
import java.net.*;
import java.io.*;

public class DaytimeClient
{
public static final int SERVICE_PORT = 13;
public static void main(String args[])
{

try
{
// Get the hostname of server
InetAddress hostname = InetAddress.getLocalHost();
//String hostname = args[0];


// Get a socket to the daytime service
Socket daytime = new Socket (hostname,SERVICE_PORT);
System.out.println ("Connection established");

// Set the socket option just in case server stalls
daytime.setSoTimeout ( 2000 );

// Read from the server
BufferedReader reader = new BufferedReader (new
InputStreamReader(daytime.getInputStream()));

System.out.println ("Results : " +
reader.readLine());
// Close the connection
daytime.close();
}
catch (IOException ioe)
{
System.err.println ("Error " + ioe);
}
}
}
```

- Client & Server at the same machine

if put → 12 & server still 13 → (error)

(IP) (port)

used the Constructor which takes the (IP + port no.)

gonna stay longer ← daytime.setSoTimeout (2000); → it is: if I don't close socket → after 200ms → termination. Socket.

in ms

byte.

low level → so: filter ↓ Buffered Reader. Uchar

From Pipe ← reader.readLine(); → Safe to use it after Bridge

(getInputStream is byte based and BufferedReader is char)

↳ So Bridge → new buffered Reader.

Creating a TCP Server

Code for DaytimeServer

```
import java.net.*;
import java.io.*;

public class DaytimeServer
{
public static final int SERVICE_PORT = 13;
public static void main(String args[])
{
try
{

// Bind to the service port, to grant clients
// access to the TCP daytime service
ServerSocket server = new ServerSocket(SERVICE_PORT);

System.out.println ("Daytime service started");

// Loop indefinitely, accepting clients
for (;;)
{
// Get the next TCP client
Socket nextClient = server.accept();

// Display connection details
System.out.println ("Received request from " +
nextClient.getInetAddress() + ":" + nextClient.getPort() );

// Don't read, just write the message
OutputStream out =nextClient.getOutputStream();
PrintStream pout = new PrintStream (out);

// Write the current date out to the user
pout.print( new java.util.Date() );

// Flush unsent bytes
out.flush();

// Close stream
out.close();

// Close the connection
```

13 as client port.

no accept here, no connection started.

as many client as possible.

(IP + port for client)

to Send. for client

Filter

For output stream

First: Server is listening then Creat Object from Socket

```
nextClient.close();
}
}
catch (BindException be)
{
System.err.println ("Service already running on port "
+ SERVICE_PORT );
}
catch (IOException ioe)
{
System.err.println ("I/O error - " + ioe);
}
}
}
```

Functions system dependant
- get Send buffer
- = received buffer
- getInput Stream
- = Output Stream.
- = Inet Address
- = port

Another TCPServer Program:

Client / Server → Read and write

```
import java.io.*;
import java.net.*;
class TCPServer {
public static void main(String argv[]) throws Exception
    {
String clientSentence;
String capitalizedSentence;
ServerSocket welcomeSocket = new ServerSocket(6789);

while(true) {

Socket connectionSocket = welcomeSocket.accept();

BufferedReader inFromClient = new BufferedReader(new
InputStreamReader(connectionSocket.getInputStream()));

DataOutputStream  outToClient = new
DataOutputStream(connectionSocket.getOutputStream());

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';

outToClient.writeBytes(capitalizedSentence);
        }
    }
```

(port) server is listening to.

filter + Bridge (in one sentence)

Bridge

low Level → filter → buffered Reader.

(to Send)

no buffered writer. So use: data output Stream

Filter of (Output Stream)

String not bytes.