Client Server Application Programming

Week 5: Transmission Control Protocol (Overview of TCP, Socket Class, Creating a TCP Client etc.)

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

Summary of previous Lecture

1. User Datagram Protocol(Overview
2. Datagram Packet Class,
3. DatagramSocket Class,
4. Listening for UDP Packets,
5. Sending UDP packets,
6. Building a UDP Client/Server,

Agenda

1. Overview of Transmission Control Protocol
2. Advantages of TCP over UDP
3. Communication between Applications Using Ports
4. Socket Operations
5. Socket Class,
6. TCP Socket and java
7. Creating a TCP Client,

Transmission Control Protocal

The Transmission Control Protocol (TCP) is a stream-based method of network communication that is far different from any discussed previously. This lecture discusses TCP streams and how they operate under Java.

TCP provides an interface to network communications that is radically different from the User Datagram Protocol (UDP) discussed in the previous lecture.

The properties of TCP make it highly attractive to network programmers, as it simplifies network communication by removing many of the obstacles of UDP, such as ordering of packets and packet loss.

While UDP is concerned with the transmission of packets of data, TCP focuses instead on establishing a network connection, through which a stream of bytes may be sent and received.

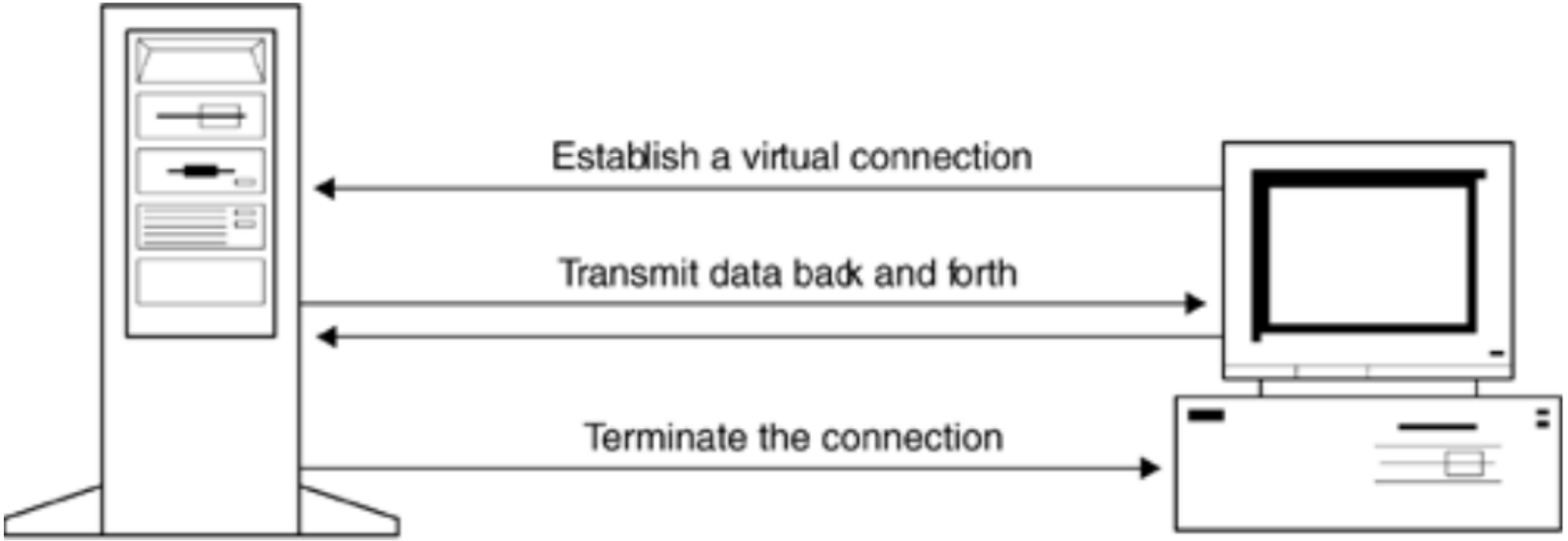
Transmission Control Protocal

When discussing UDP, we saw that packets may be sent through a network using various paths and may arrive at different times. This benefits performance and robustness, as the loss of a single packet doesn't necessarily disrupt the transmission of other packets. However, such a system creates extra work for programmers who need to guarantee delivery of data.

TCP eliminates this extra work by guaranteeing delivery and order, providing for a reliable byte communication stream between client and server that supports two-way communication. It establishes a "virtual connection" between two machines, through which streams of data may be sent. See the picture below.

Transmission Control Protocal

TCP establishes a virtual connection to transmit data



David R. and Michael R(2002).

Transmission Control Protocal

TCP uses a lower-level communications protocol, the Internet Protocol (IP), to establish the connection between machines. This connection provides an interface that allows streams of bytes to be sent and received, and transparently converts the data into IP datagram packets hence guaranteed delivery of bytes of data.

Of course, it's always possible that network errors may prevent delivery, but TCP handles the implementation issues such as resending packets, and alerts the user only in serious cases such as if there is no route to a network host or if a connection is lost.

Transmission Control Protocal

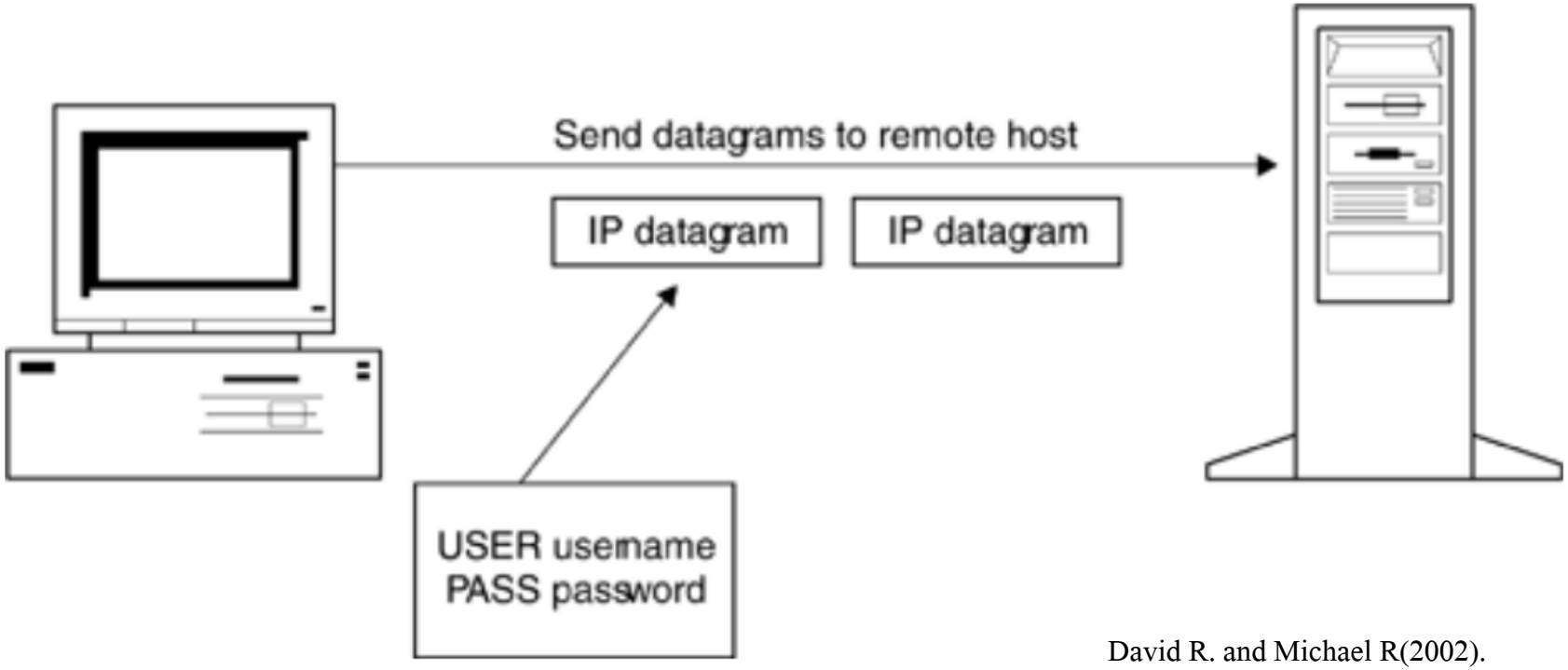
The virtual connection between two machines is represented by a socket. Sockets, introduced in previous lecture, allow data to be sent and received; there are substantial differences between a UDP socket and a TCP socket, however.

First, TCP sockets are connected to a single machine, whereas UDP sockets may transmit or receive data from multiple machines.

Second, UDP sockets only send and receive packets of data, whereas TCP allows transmission of data through byte streams(represented as an InputStream and OutputStream). They are converted into datagram packets for transmission over the network, without requiring the programmer to intervene (as shown in below)

Transmission Control Protocol

TCP deals with streams of data such as protocol commands, but converts streams into IP datagrams for transport over the network.



David R. and Michael R(2002).

Advantages of TCP over UDP

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are two different protocols used for transmitting data over a network. Here are some advantages of TCP over UDP:

1. **Reliable and Ordered Data Delivery:** TCP ensures reliable and ordered delivery of data packets. It uses mechanisms like acknowledgements, retransmissions, and sequence numbers to ensure that data packets are received in the correct order and without loss. In contrast, UDP does not guarantee reliability or order of delivery, as it is connectionless and does not have built-in error checking or retransmission mechanisms.
2. **Error Checking and Correction:** TCP includes error checking and correction mechanisms to detect and correct errors in data transmission. If a packet is received with errors, TCP requests retransmission of that packet, ensuring accurate data delivery. UDP, on the other hand, does not have built-in error checking or correction mechanisms, and any errors in data transmission are not automatically detected or corrected.
3. **Flow Control:** TCP includes flow control mechanisms to regulate the rate of data transmission between sender and receiver, preventing data overflow and congestion. This ensures that data is transmitted at a pace that the receiver can handle. UDP does not have flow control, and data packets can be sent at the sender's rate, which may overwhelm the receiver or result in data loss.

Advantages of TCP over UDP

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are two different protocols used for transmitting data over a network. Here are some advantages of TCP over UDP:

4. **Connection-oriented:** TCP is a connection-oriented protocol, which means that a reliable and established connection is maintained between the sender and receiver throughout the data transmission. This ensures that data packets are received in the correct order and without loss. UDP, on the other hand, is connectionless, and each packet is treated independently, without establishing a connection.

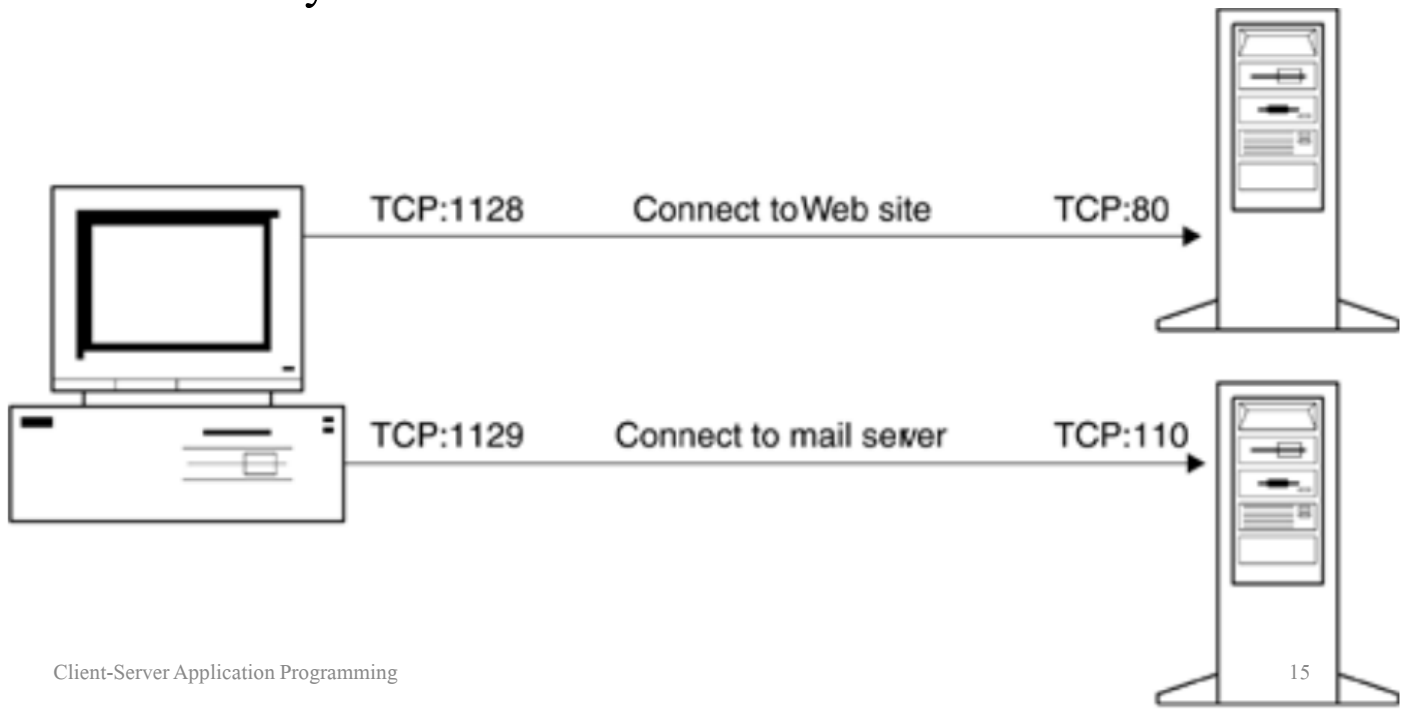
5. **Widely Supported:** TCP is a widely supported protocol and is commonly used for applications that require reliable and ordered data delivery, such as file transfer, email, and web browsing. Most network devices and operating systems support TCP by default. UDP, on the other hand, is used in scenarios where low-latency and real-time data transmission are more important than reliable data delivery, such as online gaming, video streaming, and VoIP.

However, it's important to note that UDP also has its advantages, such as lower overhead, faster transmission speed, and better suitability for certain types of applications. The choice between TCP and UDP depends on the specific requirements of the application and the trade-offs between reliability, speed, and other factors.

Communication between Applications Using Ports

Communication between Applications Using Ports

When a TCP socket establishes a connection to another machine, it requires two very important pieces of information to connect to the remote end—the IP address of the machine and the port number. In addition, a local IP address and port number will be bound to it, so that the remote machine can identify which application established the connection (as illustrated in Figure below). After all, you wouldn't want your e-mail to be accessible by another user running software on the same system.



Communication between Applications Using Ports +

Ports in TCP are just like ports in UDP—they are represented by a number in the range 1–65535. Ports below 1024 are restricted to use by well-known services such as HTTP, FTP, SMTP, POP3, and telnet. Table below lists a few of the well-known services and their associated port numbers.

Protocols and Their Associated Ports

Well-Known Services	Service Port
Telnet	23
Simple Mail Transfer Protocol	25
HyperText Transfer Protocol	80
Post Office Protocol 3	110

Socket Operations

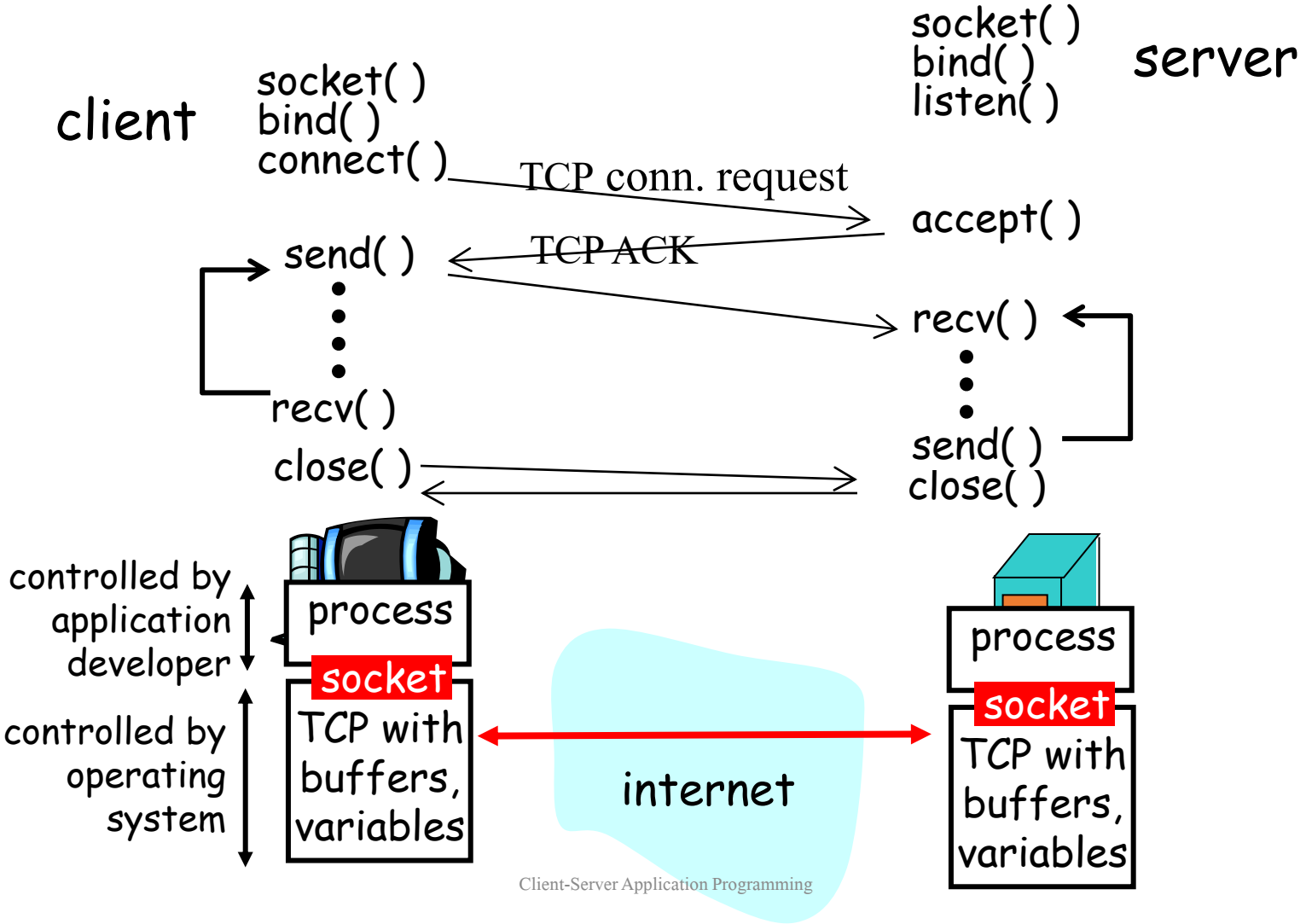
Socket Operations

TCP sockets can perform a variety of operations. In addition to the above special type of socket that provides a service that binds to a specific port number. Which type of socket is normally used only in servers, and can perform the following operations:

1. Establish a connection to a remote host
2. Send data to a remote host
3. Receive data from a remote host
4. Close a connection
5. Accept incoming connections from remote hosts
6. Unbind from a local port
7. Bind to a local port

These two sockets are grouped into different categories, and are used by either a client or a server (since some clients may also be acting as servers, and some servers as clients). However, it is normal practice for the role of client and server to be separate.

Socket-programming using TCP



TCP Sockets and Java

Java offers good support for TCP sockets, in the form of two socket classes, java.net.Socket and java.net.ServerSocket. When writing client software that connects to an existing service, the Socket class should be used. When writing server software that binds to a local port in order to provide a service, the ServerSocket class should be employed.

This is different from the way a DatagramSocket works with UDP—the function of connecting to servers, and the function of accepting data from clients, is split into a separate class under TCP.

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, however. If this functionality is required, a client application should establish multiple socket connections, one for each machine.

Socket Class Constructors

There are several constructors for the java.net.Socket class. However, two constructors, which allowed a **boolean parameter** to specify whether UDP or TCP sockets were to be used, have been **deprecated**. These constructors SHOULD NOT be used and are not listed here—if UDP functionality is required, use a DatagramSocket (covered in previous lecture).

The easiest way to create a socket is to specify the hostname of the machine and the port of the service. For example, to connect to a Web server on port 80, the following code might be used:

Socket Class Constructors+

```
try {
// Connect to the specified host and port
Socket mySocket = new Socket ( "www.awl.com", 80);
// ..... }
catch (Exception e) {
System.err.println ("Err - " + e);
}
```

However, a wide range of constructors is available, for different situations. Unless otherwise specified, all constructors are public.

Socket Class Constructors+

1. protected Socket ()— creates an unconnected socket using the default implementation provided by the current socket factory. Developers should not normally use this method, as it does not allow a hostname or port to be specified.
2. Socket (InetAddress address, int port) throws java.io.IOException, java.lang.SecurityException— creates a socket connected to the specified IP address and port. If a connection cannot be established, or if connecting to that host violates a security restriction (such as when an applet tries to connect to a machine other than the machine from which it was loaded), an exception is thrown.
3. Socket (InetAddress address, int port, InetAddress localAddress, int localPort) throws java.io.IOException,java.lang.SecurityException— creates a socket connected to the specified address and port, and is bound to the specified local address and local port. By default, a free port is used, but this method allows you to specify a specific port number, as well as a specific address, in the case of multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses).

Socket Class Constructors+

4. protected Socket (SocketImpl implementation)— creates an unconnected socket using the specified socket implementation. Developers should not normally use this method, as it does not allow a hostname or port to be specified.
5. Socket (String host, int port) throws java.net.UnknownHostException, java.io.IOException, java.lang.SecurityException— creates a socket connected to the specified host and port. This method allows a string to be specified, rather than an InetAddress. If the hostname could not be resolved, a connection could not be established, or a security restriction is violated, an exception is thrown.

Socket Class Constructors+

6. Socket (String host, int port, InetAddress localAddress, int localPort) throws java.net.UnknownHostException,java.io.IOException, java.lang.SecurityException— creates a socket connected to the specified host and port, and bound to the specified local port and address. This allows a hostname to be specified as a string, and not an InetAddress instance, as well as allowing a specific local address and port to be bound to. These local parameters are useful for multihomed hosts (i.e., a machine where the localhost is known by two or more IP addresses). If the hostname can't be resolved, a connection cannot be established, or a security restriction is violated, an exception is thrown.

Socket Class Methods

Sockets can perform a variety of tasks, such as reading information, sending data, closing a connection, and setting socket options. In addition, the following methods are provided to obtain information about a socket, such as address and port locations:

1. void close() throws java.io.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.
2. InetAddress getInetAddress()— returns the address of the remote machine that is connected to the socket.
3. InputStream getInputStream() throws java.io.IOException— returns an input stream, which reads from the application this socket is connected to.
4. OutputStream getOutputStream() throws java.io.IOException— returns an output stream, which writes to the application that this socket is connected to.
5. boolean getKeepAlive() throws java.net.SocketException— returns the state of the SO_KEEPALIVE socket option

Socket Class Methods+

6. InetAddress getLocalAddress()— returns the local address associated with the socket (useful in the case of multihomed machines).
7. int getLocalPort()— returns the port number that the socket is bound to on the local machine.
8. int getPort()— returns the port number of the remote service to which the socket is connected.
9. 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.
10. 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.
11. 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.

Socket Class Methods++

12. 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).
13. boolean getTcpNoDelay() throws java.net.SocketException— returns "true" if the TCP_NODELAY socket option is set, which controls whether Nagle's algorithm (will be discussed in the next lecture) is enabled.
14. void setKeepAlive(boolean onFlag) throws java.net.SocketException— enables or disables the SO_KEEPALIVE socket option.
15. 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.

Socket Class Methods+++

16. void setSendBufferSize(int size) throws ava.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.
17. static void setSocketImplFactory(SocketImplFactory factory)throws java.net.SocketException, java.io.IOException java.lang.SecurityException — assigns a socket implementation factory for the JVM, which may already exist, or may violate security restrictions, either of which causes an exception to be thrown. Only one factory can be specified, and this factory will be used whenever a socket is created.
18. 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.

Socket Class Methods+++

19. 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 (seen lecture 6, dealing with error control).
20. void setTcpNoDelay(boolean onFlag) throws java.net.SocketException— enables or disables the TCP_NODELAY socket option, which determines whether Nagle's algorithm is used.
21. 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.
22. 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.

Reading from and Writing to TCP Sockets

Creating client software that uses TCP for communication is extremely easy in Java, no matter what operating system is being used. The Java Networking API provides a consistent, platform-neutral interface that allows client applications to connect to remote services. Once a socket is created, it is connected and ready to read/write by using the socket's input and output streams. These streams don't need to be created; they are provided by the Socket.getInputStream() and Socket.getOutputStream() methods. As was shown in Lecture 4 on I/O streams, filtered streams provide easy I/O access.

A filter can easily be connected to a socket stream, to make programming simpler. The following code snippet demonstrates a simple TCP client that connects a BufferedReader to the socket input stream, and a PrintStream to the socket output stream.

Reading from and Writing to TCP Sockets

```
try {
// Connect a socket to some host machine and port
Socket socket = new Socket ( somehost, someport );
// Connect a buffered reader
BufferedReader reader = new BufferedReader ( new
InputStreamReader ( socket.getInputStream() ) );
// Connect a print stream
PrintStream pstream = new PrintStream(
socket.getOutputStream() );
}
catch (Exception e) {
System.err.println ( "Error – " + e);
}
```

Socket Options

Socket Options

Socket options are settings that modify how sockets work, and they can affect the performance of applications (both positively and negatively) . Support for socket options was introduced in Java 1.1, and some refinements have been made in later versions (such as support for the SO_KEEPALIVE option in Java 2 v 1.3).

Socket options are typically used to configure the behavior of the underlying TCP/IP stack or network protocols, and they can be set using setOption() method provided by the java.net.Socket and java.net.ServerSocket classes.

Below are some common socket options available in Java:

1. Socket Type (**SO_TYPE**): This option specifies the type of socket, such as **SOCK_STREAM** for TCP sockets or **SOCK_DGRAM** for UDP sockets.

Socket Options+

Below are some common socket options available in Java:

2. Socket Timeout (**SO_TIMEOUT**): This option sets the maximum time in seconds that a socket will block while waiting for data to be received or sent. If no data is received or sent within the specified timeout period, an error or timeout condition will occur.
3. Socket Reuse Address (**SO_REUSEADDR**): This option allows multiple sockets to bind to the same local address and port combination. This can be useful in cases where multiple processes or threads need to bind to the same address and port, such as in server applications.
4. Socket Keepalive **(SO_KEEPALIVE):** This option enables or disables the keep-alive mechanism, which periodically sends keep-alive packets to detect if a connection is still alive. This can be useful in cases where the underlying network may be unreliable or prone to disconnects.

Socket Options++

6. Socket Buffer Sizes (**SO_SNDBUF** and **SO_RCVBUF):** These options set the size of the send and receive buffers for a socket, which determine the amount of data that can be sent or received in a single operation. Adjusting these buffer sizes can impact the performance and efficiency of data transmission.
7. Socket Multicast **(IP_MULTICAST_*):** These options are used for multicast communication, allowing a socket to join or leave a multicast group, set multicast TTL (Time-to-Live), or specify the multicast interface, among others.

NOTE! These are just some examples of common socket options, and the available options may vary depending on the specific operating system, network stack, and programming language being used. It's important to refer to the documentation of the specific socket implementation or library being used for a complete list of available socket options and their usage.

Creating a TCP Client- TCPClient.java

Having discussed the functionality of the Socket class, we will now examine a complete TCP client. The client we'll look at here is TCPClient code, which, as its name suggests, connects to a TCPServer to read the current day and time. Establishing a socket connection and reading from it is a fairly simple process, requiring very little code. By default, the daytime service runs on port 13. Not every machine has a daytime server running, but a Unix server would be a good system to run the client against.

server, code for a TCPserver is given in week 6 lecture—the client can be run against it.

Creating a TCP Client- TCPClient.java

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

public class TCPClient {

    public static final int SERVICE_PORT = 13;

    public static void main(String[] args) {

        // Set the host name

       String hostname = "localhost";

        try {

            // Get a socket to the daytime service

            Socket socket = new Socket(hostname, SERVICE_PORT);

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

            // Set the socket option just in case server stalls

            socket.setSoTimeout(2000);
```

Creating a TCP Client- TCPClient.java

```
// Read from the server
            BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            System.out.println("Results: " + reader.readLine());
            // Close the connection
            socket.close();
        } catch (IOException ioe){
            System.err.println( "Error: No Server found.\n" + ioe); // Updated error message
        }
    }
}
```

Creating a TCP Client-TCPClient.java Explained

The given Java code is an example of a TCPClient, which connects to a TCPServer and retrieves the current date and time from the TCPServer. Let's go through the code step by step:

1.The **import** statements at the beginning of the code import the necessary Java libraries for network communication (**java.net.***) and input/output operations (**java.io.***).

2.The **TCPClient** class is defined, which contains the **main** method where the code execution starts.

3.The hostname of the TCPServer is retrieved from the command-line argument and stored in a variable named **hostname.**

Creating a TCP Client- TCPClient.java Explained

4. The code establishes a socket connection to the TCP Server on the specified hostname and the standard TCP Server port 13 using the **Socket** class. The **Socket** class represents a client-side socket that can connect to a server using an IP address and a port number.
5. The **setSoTimeout()** method is called on the **Socket** object to set a timeout of 2000 milliseconds (2 seconds) on the socket. This means that if the server does not respond within 2 seconds, an exception will be thrown. This is done to prevent the client from waiting indefinitely for a response from the server.
6. A **BufferedReader** object is created to read from the input stream of the **Socket** object. The **getInputStream()** method of the **Socket** class returns an input stream connected to the server.
7. The response from the TCP Server is read using the **readLine()** method of the **BufferedReader** class, which reads a line of text from the input stream.

Creating a TCP Client- TCPClient.java Explained

8. The retrieved results, which contain the current date and time, are printed to the console using **System.out.println()**.
9. The **close()** method is called on the **Socket** object to close the socket connection.
10. Finally Any **IOException** that may occur during the socket communication is caught using a **catch** block, and an error message is printed to the console using **System.err.println()**. **IOException** is a common exception that can occur during input/output operations, such as reading from or writing to a socket, and it needs to be handled gracefully in the code.

Creating a TCP Client- TCPClient.java code int NetBeans

```
1  package cs;import java.io.*; import java.net.*;
2  public class TCPClient {
3      public static final int SERVICE_PORT = 13; // Added missing constant SERVICE_PORT
4      public static void main(String[] args) {
5          // Set the host name
6         String hostname = "localhost";
7          try {
8              // Get a socket to the daytime service
               Socket socket = new Socket(hostname, SERVICE_PORT);
10             System.out.println("Connection established");
11             // Set the socket option just in case server stalls
12             socket.setSoTimeout(2000);
13             // Read from the server
14             BufferedReader reader = new BufferedReader(new InputStreamReader(socket.getInputStream())
15             System.out.println("Results: " + reader.readLine());
16             // Close the connection
17             socket.close();
18         } catch (IOException ioe){
19             System.err.println("Error: No Server found.\n " + ioe); //Updated error message
20         }
21     }
22 }
```

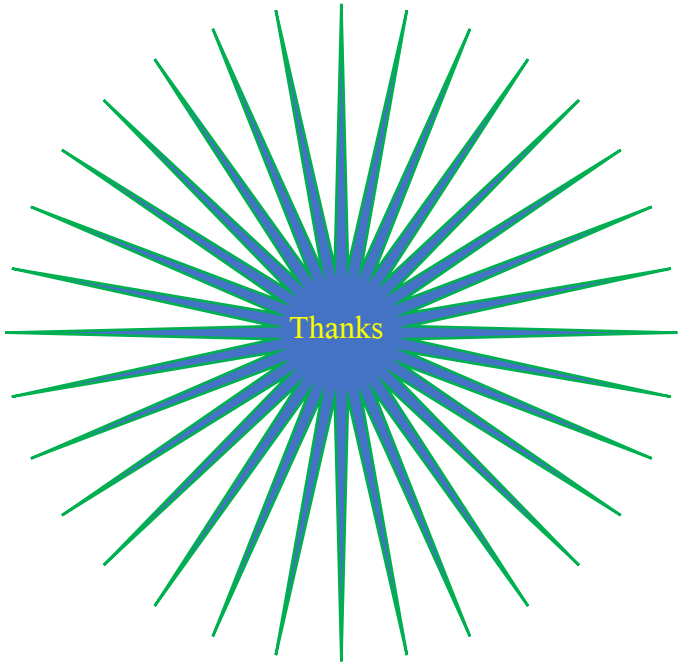
```
run:
Error: No Server found.
java.net.ConnectException: Connection refused: connect
```

Summary

Summary

1. Overview Transmission Control Protocol
2. Advantages of TCP over UDP
3. Communication between Applications Using Ports
4. Socket Operations
5. Socket Class,
6. TCP Socket and java
7. Creating a TCP Client,

Thank you for Listening



References

Java Network Programming and Distributed Computing, David R, Michael R (2002), Publisher: Addison Wesley, ISBN: 0-201-71037-4