March 19 Lecture Transcript – (Socket Programming – part 1)

Slide 2:

So far in this course, we have covered the lower 4 layers of the OSI (i.e., IP/TCP) model. With today's lecture, we are starting the discussion about 'sockets' and 'socket programming', which could be considered elements of the application layer. In particular (as explained earlier in the course), a socket could be seen as an interface point through which an application is able to access the services of the transport layer - and through the transport layer also indirectly access the services of the other lower layers of the OSI (i.e., TCP/IP) model.

Slide 3:

In this slide, a formal definition of 'socket' is provided. According to this definition:

- **Socket is a 'local-host' interface**: This means that one particular socket exists (only) on one particular machine. That is, for two applications on two different machines to communicate with each other over the network/Internet, each application/machine needs to have/create its own socket. These two sockets, presenting 2 ends of a connection (i.e., communication channel), will then be sending and receiving data to/from each other.
- **Socket is 'application created'**: This implies that if an application requires a socket in order to deploy the services of the lower (TCP/IP) layers on its respective machine, the application itself is responsible for instantiating/creating the socket.
- **Socket is 'OS controlled'**: This means that even though the application is responsible for the instantiation of the socket that it deploys, the actual operation (i.e., the inner-workings) of the socket are controlled by the OS of the respective machine.

The slide also provides a reminder of the fact that each socket (e.g., a socket (S) associated with a particular application/program (A) running on a particular machine (M)) is uniquely defined with two parameters:

1) the IP address of this machine (M);
2) the port number that has been assigned by the transport layer of this machine (M) to this particular application (A).

Slide 4:

The figure in this slides illustrates the concepts explained in the previous slide, such as:

- sockets 'sit' between the application (layer) and the OS of a given machine;
- 2 applications on 2 different machines (effectively) communicate by means of their respective sockets;
- the 'instantiation' of a socket is controlled by the application (i.e., the designer/programmer of a networking-enabled application needs to include some explicit lines of code that will create/deploy a socket);
- the 'inner-workings' of a socket (i.e., the way in which data that is passed into a socket is actually stored and processed) are controlled by the OS.

Slide 5:

'**Socket-programming**' refers to the process of creating (i.e., coding) a networking-enabled program/application involving sockets. Btw, as we already know, in the Internet, there are 2 main types of networking-enabled applications – 'client' and 'server', and each of these types has very different responsibilities and characteristics. The differences in the functionalities of client and server applications, clearly, are (i.e., need to be) reflected in the way they are actually programmed, and the way they deploy their respective sockets. (We will talk more about this in the subsequent slides ...)

In all programming languages, a socket-programming developer (whether he is developing a client or a server application) has the full control over when and for which purpose a socket will be created. However, he has very little control over the transport-layer side of the socket which he creates/instantiates. In fact, in many programming languages, the only transport-layer parameter that the socket-programming developer can fully control is which transport-layer protocol the given socket will be supported with – either TCP or UDP.

When actually deciding whether to choose TCP or UDP as the transport-layer protocol of choice for a socket, the socket-programming developer needs to consider the following three characteristics/requirements of his application:

- **reliability**: is it critical for the application to have all of its data transmitted without any potential (packet) loss;
- **timing**: is it critical for the application to have its data transmitted without any delay;
- **overhead**: can the application tolerate any overhead in terms of (e.g.) extra headers or (e.g.) extra time to establish a connection with the other machine …

Slide 6:

In this slide, the pros and cons of TCP vs. UDP in terms of 'reliability', 'timing' and 'overhead' are discussed.

In terms of **reliability**, as we know, TCP is far superior over UDP, as TCP has built-in mechanisms for detection and retransmission of lost and delay packets. UDP, on the other hand, has no such mechanisms, so if an application relies on UDP and a packet gets lots, the application itself has to have/make provisions to detect and recover the lost packet …

When it comes to **timing**, UDP is (in fact) superior to TCP, as it allows an application to keep transmitting packets/data at a constant rate – regardless of the actual conditions (e.g., potential congestion) in the network or at the receiving machine. On the other hand, as we have seen when discussing TCP, TCP has several different built-in mechanisms specifically intended to detect network and receiving-end congestion, and these mechanisms can/will trigger automatic reduction in the rate of transmitted packets. So, if an application (e.g., real-time video, as in the case of Skype) requires that packets/data be transmitted at a constant rate (even if the are certain packet/data losses occurring in the network), then TCP should not be the transport-layer protocol of choice.

From the perspective of **overhead**, both TCP and UDP have their own 'disadvantages'.

- For example, in the case of an application that deploys a UDP socket (as we will see later in this lecture), every packet is created and sent through the (UDP) socket on its own, which implies that the application has to explicitly 'attach' a header to each data segment before the segment can actually be passed to the socket. This is an example of 'processing overhead' …

- In contrast, TCP sockets do not require this – i.e., when using a TCP socket, the application can simply pass data into the (TCP) socket, without any processing overhead. However, TCP sockets require that a connection between the client and the server be established before any application data could be exchanged (recall the three-way handshake). And, this clearly results in a time-related overhead (i.e., initial communication delay) …

Slide 7:

Taking into account the discussion from the previous slide, this slide gives some specific recommendations (for socket-programming developers) about when to use a TCP vs. when to use a UDP socket.

So, clearly, applications that require reliable and in-order delivery of packets, and at the same time can tolerate potential and occasional packet delays, TCP should be the protocol (socket-type) of choice. Examples of applications that do use TCP sockets are WWW client and servers, FTP and TELNET client and servers, etc.

On the other hand, for applications that cannot tolerate (significant) fluctuations in the rates of sent/received packets, but are not overly impacted by occasional loss of data, UDP should be the protocol (socket-type) of choice. Examples of applications that do use UDP sockets are real-time video and audio streaming client/servers.

Slide 8:

This slide is just a refresher of 'client-server' framework – something we've discussed several times earlier in the course.

One point to note here, however, is the fact that servers generally can be:

- **Iterative** – they serve only one client request at a time, and are typically programmed as 'single-threated' applications.
- **Concurrent** – they can server multiple client requests at a time, and are typically programmed as 'multi-threated' applications.

Slide 9:

The figure in this slide illustrates the idea of an 'iterative server'. Clearly, in such a server, all the clients' requests/data are received through the same (shared) socket and are placed into the same (shared) buffer. And these requests are then served on 'one-by-one' (typically FIFO) basis.
UDP servers are typically iterative.

Slide 10:

The figure in this slide illustrates the idea of a 'concurrent server', which is how some TCP servers are designed to operate.
So, a concurrent TCP server has one 'parent socket' through which the initial connection-establishment requests are received. Then, for each such request received, the server instantiates a separate/dedicated socket and continues to exchange data with the respective client through this dedicated socket …

Slide 11:
The figure in this slide provides a more detailed look into the communication process/stages between one (single) UDP server and (multiple) UDP clients.
It should be noted here that the server operates in an 'iterative' manner.

Slide 12:
The figure in this slide reinforces the concepts from slides 9 and 11.

Slide 13:
The figure in this slide provides a more detailed look into the communication process/stages between one concurrent TCP server and one TCP clients.

Here it is important to note that TCP, itself, requires that the entire communication process between the client and the server be split into (i.e., seen as consisting of) 2 different stages:

1) the initial connection establishment (think of the 3-way handshake) – shown in the upper part of the provided figure;
2) the actual exchange of data – shown in the lower part of the provided figure.

Also note that the second (data-exchange) stage considers the possibility of both: a) the client sending data to the server, and b) the server sending data to the client. However, due to the nature of the two application, it is expected that the server will be the FIRST one receiving data from the client (i.e., receiving a client request), while the client will be the NEXT (second) receiving data from the server (i.e., receiving a server response).

Slide 14:
The figure in this slide reinforces the concepts from slides 10 and 13.

Slide 15:
For real-world application designers/programmers, often one of the important decision to make is – which (actual) programming language is the best-suited for socket-programming applications. (Btw, most modern programming languages have their own socket/networking libraries and could, theoretically, be used to develop socket-programming applications.)

In this slide, the advantages and disadvantages of using Java vs. C when developing socket-programming applications are listed. (Note, in this course we will proceed using Java as the programming language of choice!)

Some of the key advantages of using Java include:

- In Java, many socket and networking related complexities (i.e., parameters) are bundled/hidden 'under the hood' – e.g., within a single class. As a result:
  1) the development of socket-programming applications is much simpler in Java, and
  2) the actual code of these applications is much 'leaner' and 'cleaner'.
- Once a Java socket-programming application is complied into the 'bytecode', this same bytecode can be ported to and used on any computer/OS - as long as the respective computer has a Java

interpreter (i.e., JVM) built in it. (This stems from Java's 'interpreted-language' nature.) On the other hand, for any socket-programming application developed in a 'compiled language', like C, a different version of the program's executable will have to be created for every different type of operating system. (E.g., we would have to create one executable for Windows, one for Linux, one for Mac.)

Slide 16:

The figure in this slide illustrates the fact (i.e., advantages) of Java being 'interpreted'. So, as a result, the same/single Java bytecode and be used/run on different OSs.

Slide 17:

This slide illustrates the previously made point that the codes of socket-programming applications in Java are far 'leaner' and 'cleaner' than those written in C.

The figure in the upper part of the slide shows all the many lines of code that are required in C in order to instantiate / create one single socket.

The figure in the lower part of the slides shows one single line of code in Java that accomplishes the same as all the (above) lines of code in C – create one single socket.

Slide 18:

The main Java's library that deals with (i.e., enables) socket programming is **java.net**.
https://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html
This is the key library that we will be referring to in the reminder of this lectures, and you will also need to use this library for the purposes of your socket-programming project.
java.net contains many different classes; however, you will have to use and know only a few of them – specifically the ones enlisted in this slide. These classes are:

1) **InetAddress** is a class used to represent (deal) with IP addresses. It has 2 subclasses – Inet4Address and Inet6Address – with the first one representing IPv4 and the second IPv6 addresses. Each class comes with many different methods, including:
    - getHostAddress() – returns the IP address of the host machine as a string;
    - getHostName() – returns the symbolic name of the host machine as a string;
    - isLinkLocalAddress() – checks if the provided address is a link local IP address; etc.
2) **ServerSocket** is a class that can be used to implement a passive connection-oriented TCP server socket – i.e., the main server socket that 'listens' for initial client requests for connection …
3) **Socket** is a class that can be used to implement either: a) a TCP client socket, or b) a dedicated server-side TCP sockets (as in the case of concurrent TCP server shown in slide 10).
4) **DatagramSocket** is a class that can be used to implement a connectionless UDP client- or UDP server socket. Note, in case of UDP, and when looking (just) from the perspective of packet exchange, the actual functionality of the client and the server is (almost) identical – they both just send and receive sand-alone UDP packets – thus the same socket can be implemented/used on both sides.
5) **DatagramPacket** is a class that can be used to implement/create a UDP packet – i.e., a stand alone segment of data plus a header that contains the source and the destination IP address as well as the source and the destination port number.

6) **MulticastSocket** is a class that can be used to create a (connectionless) socket for sending and receiving of data to/from multiple other users (i.e., a multicast group).

Slide 19:

This slide provides a simple Java code that utilizes java.net library.

The purpose of this code/program is:

1) For a given URL (i.e., symbolic name) of a server – in this case **javacodegeeks.com** - determine the respective IP address of the given server (i.e., server machine) by using the method getByName(). Note, this method will return the IP address as an **Inet4Address** object!
2) Convert/extract the Inet4Address object (**address**) obtained in the previous step into an IP address in the String format – this is done by applying/executing getHostAddress() on **address**.