Name: _____

IB Computer Science 2
Transport-layer protocols worksheet

Download `Transport.zip` to the folder where you usually write activity code, and extract it.

The transport layer provides end-to-end communication between two processes (running programs), regardless of whether they are located on the same physical computer.

The most common protocols at this layer are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Both provide **ports**, numeric addresses used to identify separate processes. At this network layer, packets begin with a TCP or UDP header containing the packet's source and destination ports. Thanks to ports, multiple programs can use the network at the same time: the operating system maintains a mapping between ports and processes, which it uses to route traffic arriving at (or looping back to) the local machine to the process it is intended for. For this reason, both TCP and UDP are implemented within the operating system itself, which provides a programmatic interface for server programs to "bind" to, or listen on, specific ports and client programs to "connect," or set the target of their data transmission, to another process's port. The operating system automatically adds and removes a TCP or UDP header to or from each packet before transmitting or after receiving it, so higher-level protocols do not see these headers.

UDP is a barebones protocol: besides the port numbers, its protocol header contains only the packet length and an **error checking** "checksum" that is used to verify that neither the header nor the payload have been corrupted during transmission across the network. TCP provides four further features to establish **data integrity**, ensuring that the sent and received data match.

Ports

1. Connect to your own TCP port 1024 with the command: `nc -v localhost 1024`
   What happens and why?

2. Start a server process and bind it to this TCP port: `nc -l 1024`
   With the server still running, split the terminal and run: `hostname`
   On a second computer, replace `hostname` with that output in: `nc hostname 1024`
   What happens when you type into either `nc`?

3. Predict what would happen if you bound a server to port 1024 on the second computer with the first connection still open.

4. On the second computer, split the terminal and run: `hostname`
   On the second computer, split the terminal and run: `nc -l 1024 | tee tcp.txt`
   Use the file explorer to view the contents of the folder. What do you notice?

5. On the first computer, replace `hostname` in: `seq 100000 | nc hostname 1024`
   Were the processes able to communicate? What did the `seq` command do?

6. On the second computer, open the `tcp.txt` file. What did the `tee` command do?

7. The port numbers for popular services are standardized. Run this command on either computer to see the port assignments: `open /etc/services`
   Which ports do the `http://` and `https://` at the beginning of URLs correspond to?

8. What does the file say about port 1024? What do you think this means?

You should have found that when you connected to a remote port, you got a two-way connection, implying that *the client process had a port of its own* so that it could receive data. You should also have found that binding to port 1024 on the second machine established a separate connection, implying that *the client program on that machine was using a different port*. Indeed, when a client connects to a remote port, the operating system assigns it an arbitrary “ephemeral,” or temporary, port. Ephemeral ports are important because we might want multiple processes to connect to the same remote port, as when duplicating a Web browser tab.

TCP killer feature: reliable delivery

Thanks to its four killer features, TCP makes transferring a file between computers seamless. To understand the significance of those features, we will now see what goes wrong when we try to accomplish the same trivial task with the simplistic UDP.

9. Use Ctrl+C to close your existing `nc` TCP connections. Establish a UDP connection:
   On the second computer, run: `nc -ul 1024 | tee udp.txt`
   On the first, replace `hostname` in: `seq 100000 | nc -uw1 hostname 1024`
   Once the client exits, hit Ctrl+C to stop the server. What do you notice?

10. On the second computer, run `cmp tcp.txt udp.txt` to find the first line that differs between the two files. Open `udp.txt` in your editor; what happens at that line?

11. Continue scrolling through the file. Are there other instances of this phenomenon?

You should have found that you did not receive all of the data! Not only was `udp.txt` shorter than `tcp.txt`, it probably also had gaps in the middle. You have just seen that, unlike TCP, UDP does not ensure that transmitted data is ever received!

TCP killer feature: flow control

There are many reasons why data might be lost during network transmission. A common one is packet loss, which occurs when one or more packets do not make it across the network from the sender to the recipient. Hopefully, this is not the reason for what we just saw: your computers are directly connected, and the network should not be very busy!

To understand what's happening, let's look at (roughly) what `nc` is doing programmatically to manage the UDP connection. Open `UdpServer.java` and `UdpClient.java`, short examples of Java server and client applications. The two programs communicate back and forth. Start reading through the server comments and source code, but whenever you get to a comment saying that the program waits, switch to reading the source code of the other program.

12. Run `java UdpServer.java`. Then, in another terminal on the same computer, run `java UdpClient.java`. Enter `localhost` followed by a short one-word message. What happens?

13. Repeat the above steps, but this time send a longer message consisting of a sentence of at least four words. How much of the message does the server receive and why?

14. What do you think happened to the rest of the message?

Here, we lost the remainder of the message because the server did not allocate a large enough buffer to store the packet's full payload. However, even if a recipient application's buffer is large enough to hold a packet, it is common for multiple packets to arrive before it has a chance to call `.receive()` again. To prevent packet loss in this situation, the operating system stores incoming packets in a larger receive buffer, and `.receive()`s by the application copy from this buffer. The operating system then knows it is safe to reuse that portion of the receive buffer.

Unfortunately, the receive buffer's size must still be limited, or the operating system would waste a lot of memory. If an application is not `.receive()`'ing data as fast as it arrives, the receive buffer can become full. In this case, the operating system discards any additional packets that arrive until the application catches up and leaves space in the receive buffer.

15. In the previous section, what was the `nc` server doing that caused it to receive data more slowly than the client was sending it?

To prevent the receive buffer from becoming full, TCP provides a feature called **flow control** that automatically manages the sender's transmission rate. When sending more than a small amount of data over UDP, an application must implement this itself. Briefly skim the `FlowControlServer.java` and `FlowControlClient.java` programs, focusing on the conditional inside their loops.

Copy `FlowControlServer.java` to the second computer, either by using `nc` in TCP mode or by redownloading the ZIP file.

16. On the second computer: `java FlowControlServer.java 1024 | tee udp.txt`
    On the first: `java FlowControlClient.java hostname 1024 100000`
    What does the client output and why? What do you notice about the server's output?

17. On the second computer, run `wc -l tcp.txt udp.txt` to compare the number of lines in the two files. Did everything make it?

18. What would happen if the network lost one of the server's resume messages?

Our example takes a simplistic approach to flow control, periodically pausing transmission. TCP is smarter about it, giving notifications to continue sending in advance based on speed of arrival.

TCP also takes steps to avoid **deadlock**, which occurs when each side is waiting for the other. As you may have realized, packet loss can send our example programs into such a state.

TCP killer feature: consistent ordering

So far, we have been sending data between two machines that are directly connected over the same network link. Let's see what happens when UDP traverses multiple network segments.

Your instructor will give you a link to visit in the browser of your second computer. Use it to start a flow control server; it will provide you with a hostname and port. Replace `hostname` and `port` in the command: `java FlowControlClient.java hostname port 100000`

Back in your browser, click the link to download the resulting `udp.txt` file. Move it into the folder with the others, replacing the old one.

19. Rerun `wc -l tcp.txt udp.txt`. What do you notice about the files' lengths?

20. Run `cmp tcp.txt udp.txt` to find the first line that differs. (If there is no output, the files are the same. In this case, hit your browser's back button and retry the experiment.) Open `udp.txt` in your text editor; what happens at that line?

21. Search the file for the missing numbers. What do you discover?

Although the sender transmitted the packets in order, they took different paths through the network. This caused some to arrive out of order at the receiver!

Compared to the UDP header, the TCP header primarily adds a packet sequence number. TCP uses this number both to detect lost packets that must be resent (reliable delivery) and to rearrange packets that arrive out of order (consistent ordering).

TCP killer feature: congestion control

Let's return to using TCP to demonstrate its final killer feature. The Web's application-layer HTTP protocol is a *very* thin layer on top of TCP: the only thing it adds is an HTTP header as part of the first TCP packet's message.

The provided `http_header.txt` file contains such a header. On your first computer, run the command `yes 1 | cat http_header.txt - | nc -l http` to start a Web server hosting a file of infinite size.

22. Download the file on the second computer by replacing `hostname` in the command:
    `curl -o /dev/null http://hostname`
    Watching the current speed, what do you notice happening over time?

The speed of transmission is limited not only by the hardware, but also by **congestion**, or other traffic on the network. Each network link can only carry so much data per second. When links are saturated, data has to wait in buffers on **routers**, the computers between the links. Just like the destination computer's receive buffer, routers' buffers can become full and drop packets.

Fortunately, TCP can tell when this happens thanks to its sequence numbers. It uses this information not only to resend the dropped packets, but also to manage its transmission speed, with the goal of sending as fast as possible without causing packet loss. You probably noticed the download speed slowly increasing for the first few seconds of the transfer. This was TCP's "slow start" behavior, where it gradually increases its transmission speed until it converges to the network capacity.

TCP makes communication easy

Examine `TcpServer.java` and `TcpClient.java`, TCP server and client applications. They do the same thing as our earlier UDP examples; try running them as you did with those.

23. Compared to `UdpServer.java`, what constant is missing from `TcpServer.java`?

24. How can the programs work without being aware of byte buffers or their sizes?

25. Let's unpack that. In your own words, what problems can you ignore if you use TCP?
    a.

    b.

    c.

    d.