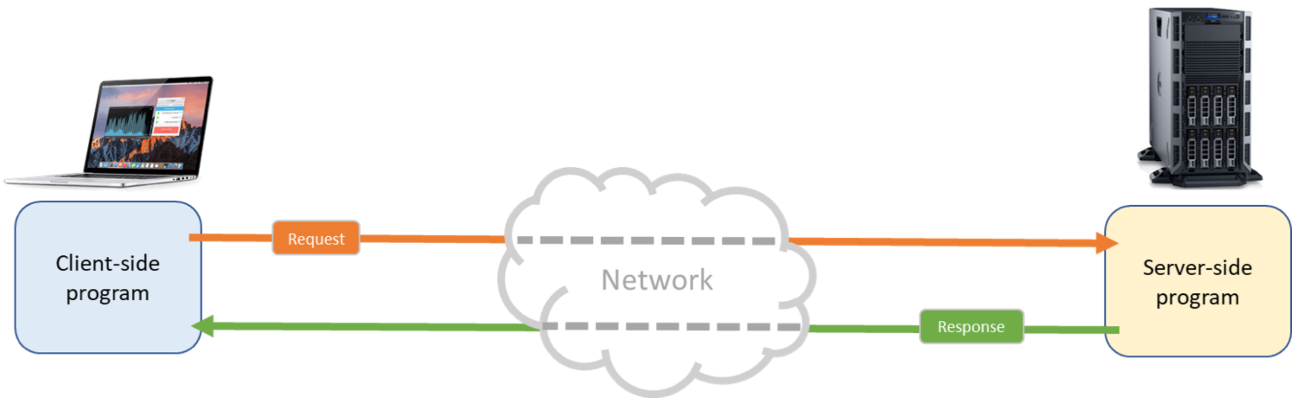
COMP3015
Data Communication and Networking
Socket Programming – Transmission Control Protocol 1

Introduction

Transmission Control Protocol (TCP) is a connection-oriented protocol. TCP requires connection establishment between two hosts before they can transfer data to each other.



Client and Server

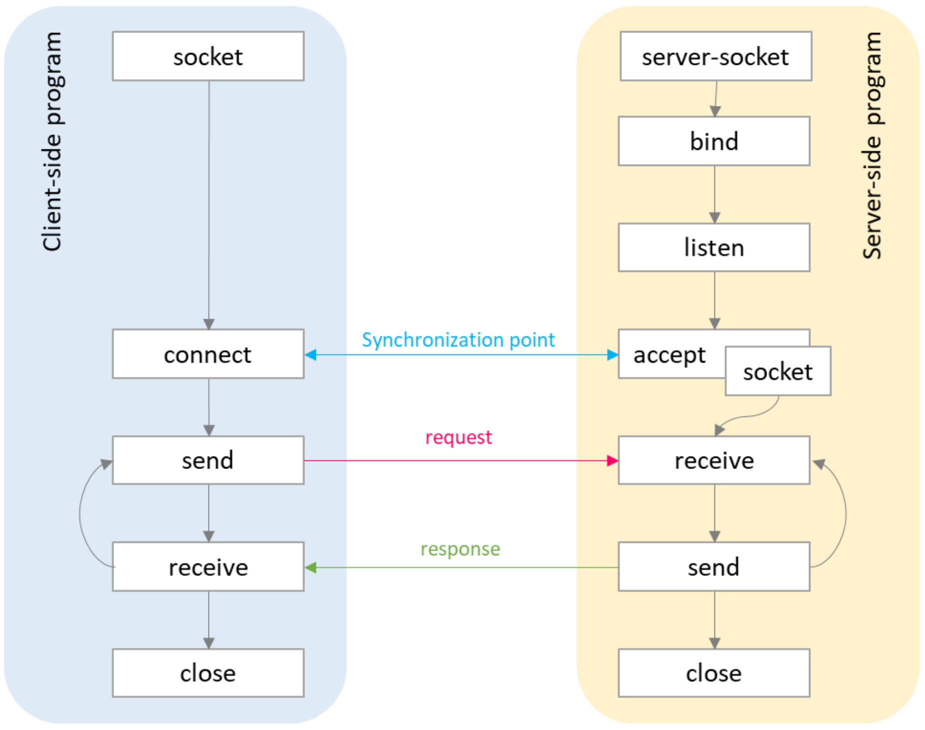
A network connects computers, mobile phones, peripherals, network devices, etc. Devices connected to your network can communicate with one another.

In a client-server application, the two main components are server and client. The server runs the server-side program that passively waits for and responds to client-side programs. The client runs the client-side program that initiates the communication to the server-side program and requests services actively.

Sockets

The socket is an abstraction through which an application may send and receive data. A socket is uniquely identified by the internet address (IP address in IP network), end-to-end protocol (e.g., TCP or UDP), and a port number. Java API (**java.net**) provides two types of sockets – Stream sockets (TCP) and Datagram sockets (UDP). In our labs and project will cover the Stream sockets only.

The socket classes, **Socket** class and **ServerSocket** class provided by **java.net**, are used to represent the connection between a client-side program and a server-side program.

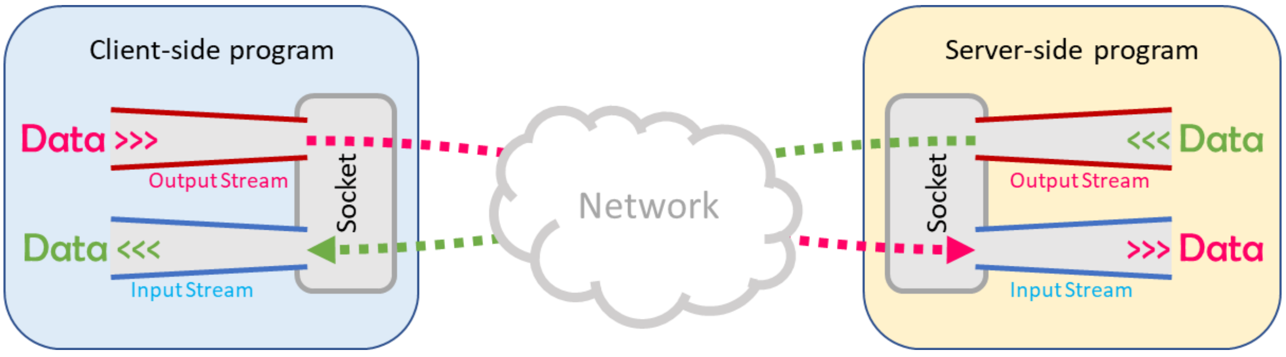


Server

The server-side program has a server socket that is bound to a specific port number. This socket just waits and listening to the connection request sent by a client to make a connection. If the connection established successfully, a new socket object is used for communicating with the client. Then, the output stream and input stream of the newly created socket will be used for sending and receiving messages to and from the client respectively.

Client

The client-side program must know IP address (perhaps hostname if DNS is available) and port the number of the server (a machine that runs the server-side program). Then, the client-side program sends a connection request to the server and establish a connection. If the connection established successfully, the output and input streams of the socket will be used for communicating with the server-side program.



Connection Establishment and Receiving Data

Imagine that a server-side program transmits a character to its client every 1 second through a TCP connection. The IP address of the machine running the program is *zzz.yyy.xxx.www*, and the program is listening to the TCP port *vvv*. We need to write a client-side program to receive the characters transmitted from the server-side program.

Firstly, the client-side program needs to establish a connection to the server-side program. The **Socket** class will be used in the program.

```
Socket socket = new Socket("zzz.yyyy.xxx.www", vvv);
```

When successfully establishing the connection, we can then obtain its input stream for receiving data.

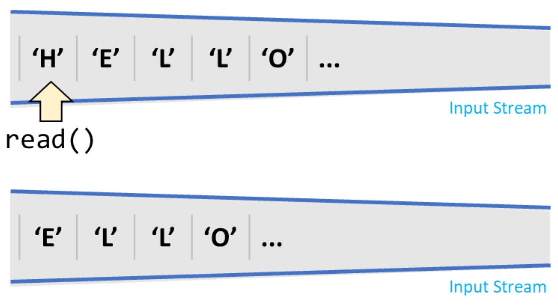
```
InputStream in = socket.getInputStream();
```

Using read()

Consider the following code segment:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Server IP: ");
String ip = scanner.nextLine();
System.out.print("TCP Port: ");
int port = scanner.nextInt();

Socket socket = new Socket(ip, port);
InputStream in = socket.getInputStream();

while(true) {
   char c = (char) in.read();
   System.out.println(c);
}
```

In the code above, we use the **read()** method to retrieve a byte from the input stream of the socket. When the input stream is empty but we execute the **read()** method, the program will be blocked until the input stream becomes not empty. If the input stream is not empty, the **read()** method removes the first byte of the data from the input stream and returns that byte in the integer data type.



To test the code segment above, you need to create a new class with the main method. Then, put the code segment to the main method and run it. But, of course, there must be a server-side program running on a machine. A server-side program will be started in the lab lesson, so you can use it to test your program. If you want to test your program after this lesson, you can follow the steps below to run the server-side program on your computer.

1. Download the **CharServer.class** file through the following URL and save it in your Downloads directory.

   http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab3

2. Open the Command Prompt (Windows) or Terminal (Mac or Linux).

3. Change the current directory to the Downloads directory.

   ```
   > cd Downloads
   ```

4. Run the following command to start the server-side program.

   ```
   > java CharServer
   ```

5. Go back to your client-side program and run it.

6. Input the IP address and TCP port number shown by the server-side program.

7. Check the outputs of the client-side program.

Using read(byte[], int, int)

How about the server-side program transmits several bytes as a message every 2 seconds? Too many overheads will be generated if the client-side program uses the **read()** method reads bytes one-by-one. Instead, we use the **read(byte[], int, int)** method to read multiple bytes at once.

Consider the following code segment:

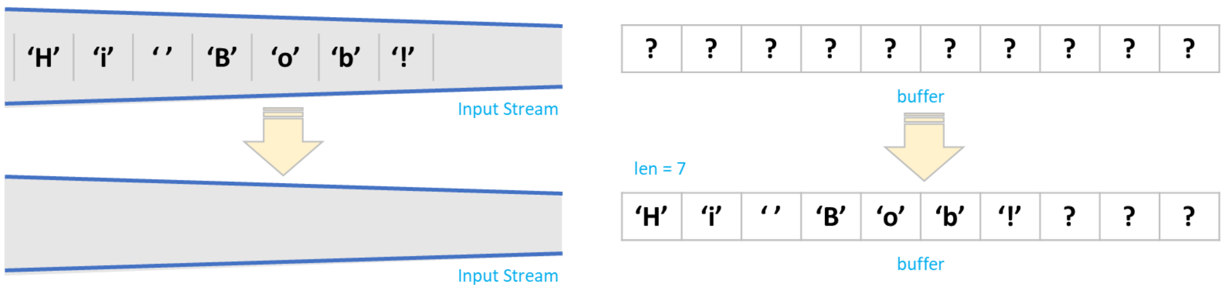
```
Scanner scanner = new Scanner(System.in);
System.out.print("Server IP: ");
String ip = scanner.nextLine();
System.out.print("TCP Port: ");
int port = scanner.nextInt();

byte[] buffer = new byte[1024];

Socket socket = new Socket(ip, port);
InputStream in = socket.getInputStream();

while(true) {
   int len = in.read(buffer, 0, buffer.length);
   System.out.println(new String(buffer, 0, len));
}
```

Because we do not know the length of the message sent by the server-side program, we use a large-enough byte array as a buffer for reading messages from the input stream. We assume that the length of each message is less than 1024.

In addition, we use **in.read(buffer, 0, buffer.length)** to read the input stream. Same as the **read()** method, the **read(byte[], int, int)** method blocks the client-side program if the input stream is empty. If the input stream is not empty, the bytes will be read from the input stream to the buffer.

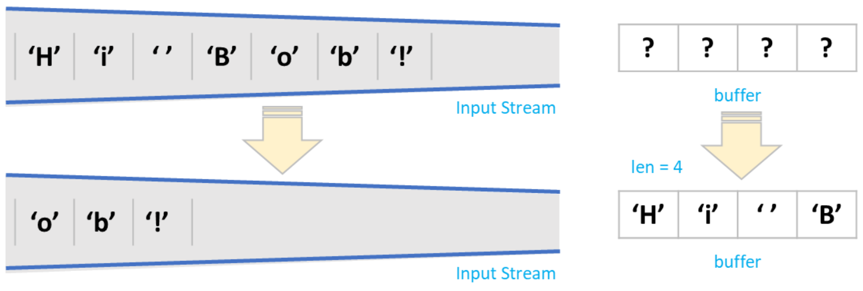
If the buffer is greater than the data stored in the input stream, all bytes will be read to the buffer and the **read** method returns the number of bytes read.

```
int len = read(buffer, 0, buffer.length)
```



If the buffer is smaller than the data, the part of the data will be read and fill the buffer until the buffer becomes full. The remaining part will be kept in the input stream. The **read** method returns the number of bytes read, same as the size of the buffer.

```
int len = read(buffer, 0, buffer.length)
```



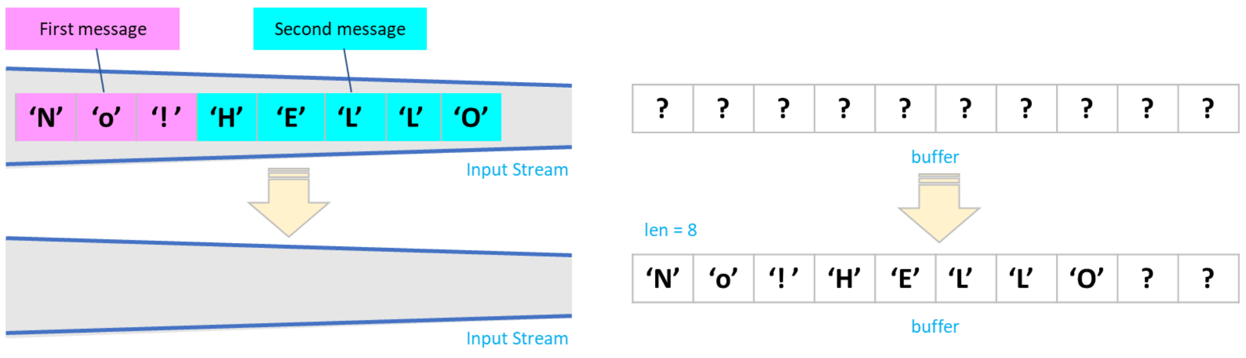
You can start a message server using the **MessageServer1a.class**[1] file. The server-side program sends a message every 2 seconds.

[1] **MessageServer1a.class** can be found in the page http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab3

Application Layer Protocol

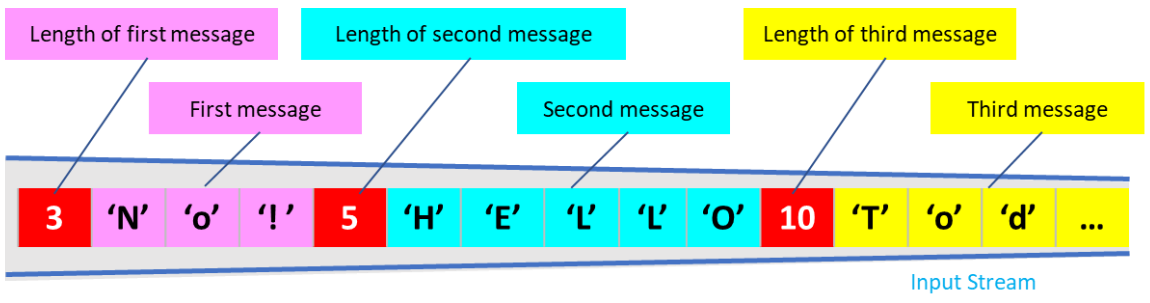
However, we cannot guarantee that the client-side program can pick the message immediately when a new message has just arrived. A client-side program of a normal network application should have some tasks, it should not just monitor the input stream. Therefore, multiple messages may have already arrived before the client-side program reads the input stream.

```
int len = read(buffer, 0, buffer.length)
```



When the buffer is large enough, the client-side program will read multiple messages to the buffer. It is because the **read** method does not know where the end of a message is. You may run the **MessageServer1b.class**[2] file and your client-side program (based on the example code shown on page 4) to simulate the problem.

To avoid mixing multiple messages together, we need to set an application-level protocol (the communication method) to the server-side program and the client-side program. For example, the server-side program every time sends the length of the message in integer data type before sending the message. When the client-side program receives the length of a message, it will know how many following bytes belong to that message.



[2] **MessageServer1b.class** can be found in the page http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab3

DataInputStream

If the server-side program sends an integer to the client-side program, the client-side program must have a way to receive the integer. The default input stream (**InputStream** class) has the methods for reading a single byte or multiple bytes. It does not have a method to receive integers. So, we use the **DataInputStream** class instead.

```
DataInputStream in = new DataInputStream(socket.getInputStream());
```

The **DataInputStream** class provides a set of methods for reading the data in different data types – *int* (**readInt**), *long* (**readLong**), *float* (**readFloat**), *char* (**readChar**), *boolean* (**readBoolean**), etc.

```
int len = in.readInt();
in.read(buffer, 0, len);
```

We use **in.readInt()** to read an integer that represents the length of the following message. In the **in.read(buffer, 0, len)** statement, we specify the number of the bytes for reading the data from the input stream. So, the client-side program now will not mix the messages together.

If you want to test the code segment above after this lesson, you can use the **MessageServer2.class**[3] file to start the server-side program.

Sending Data through DataOutputStream

To send the data to the remote side, we use the **write()** method provided by the output stream, just like what we learned from Lab 1. As mentioned above, we usually define the application layer protocols. The server-side and client-side programs follow our defined protocols to transmit data.

Let’s reuse the protocol defined in the previous example. We first send an integer representing the length of the data, then send the data. Imagine that a server-side program is running for handling file submissions.

If the client-side program uploads a file named *song.mp3*, the data transmitted are as follows:

Name length	Filename								File length	Content				
5	**‘s’**	**‘o’**	**‘n’**	**‘g’**	**‘.’**	**‘m’**	**‘p’**	**‘3’**	**813821**	**x**	**x**	**x**	**x**	**…**
int	bytes								long	bytes				

[3] **MessageServer2.class** can be found in the page http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab3

The client-side program runs for uploading a file by performing the following procedure:

1. Establish a connection to the server.

   ```
   Socket socket = new Socket(serverIP, port);
   ```

2. Open an output stream to the socket.

   ```
   DataOutputStream out = new DataOutputStream(socket.getOutputStream());
   ```

3. Open an input stream to the file being uploaded.

   ```
   FileInputStream in = new FileInputStream(file);
   ```

4. Send an integer that represents the length of the name of the file.

   ```
   out.writeInt(file.getName().length());
   ```

5. Send the file name in bytes.

   ```
   out.write(file.getName().getBytes());
   ```

6. Send a long that represents the file size.

   ```
   long size = file.length();
   out.writeLong(size);
   ```

7. Send the file content in bytes.

   ```
   int len = in.read(buffer, 0, buffer.length);
   out.write(buffer, 0, len);
   ```

8. Repeat step 7 until the content of the file is completely sent.

9. Drop the connection.

   ```
   in.close();
   out.close();      // the socket will be closed too.
   ```

Exercise

Write a Java program with a class named **FileClient** that will be used for uploading files to the server. The follows are the details of the program:

1. **FileClient** class has a static method named **upload**. It accepts three parameters respectively are the **server IP address** (String), **server port number** (int), and **file name** (String).
2. If the file is a valid file (exists but not a directory), send the file to the server using the application-layer protocol mentioned in the section "Sending Data through DataOutputStream".
3. Otherwise, an error message should be prompted about the problem.
4. If there is any problem during the data transmission, an error message should be prompted about the problem. Then, the program should be terminated.
5. If the data transmission is completed, your program should show a message about the completion.
6. You may use the following main method for your program.

```java
public static void main(String[] args) {
   Scanner scanner = new Scanner(System.in);
   System.out.println("Please input\nServer IP: ");
   String ip = scanner.nextLine().trim();
   System.out.print("Port no:    ");
   int port = Integer.parseInt(scanner.nextLine());
   System.out.print("File:       ");
   String filename = scanner.nextLine().trim();
   scanner.close();

   FileClient.upload(ip, port, filename);
}
```

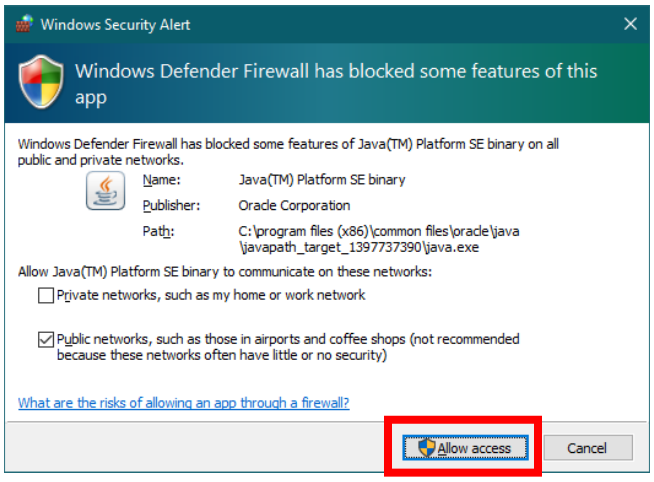
To test your program, you need to download the server-side program – **FileServer.class** from the following URL: http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab3/FileServer.class

And, run it using the following command:

java FileServer *9001*

where 9001 is the server port number

Then, click "Allow access" if ***Windows Security Alert*** is prompted.



Windows Security Alert

Windows Defender Firewall has blocked some features of this app

Windows Defender Firewall has blocked some features of Java(TM) Platform SE binary on all public and private networks.

Name:	Java(TM) Platform SE binary
Publisher:	Oracle Corporation
Path:	C:\program files (x86)\common files\oracle\java \javapath_target_1397737390\java.exe

Allow Java(TM) Platform SE binary to communicate on these networks:

- ☐ Private networks, such as my home or work network
- ☑ Public networks, such as those in airports and coffee shops (not recommended because these networks often have little or no security)

What are the risks of allowing an app through a firewall?