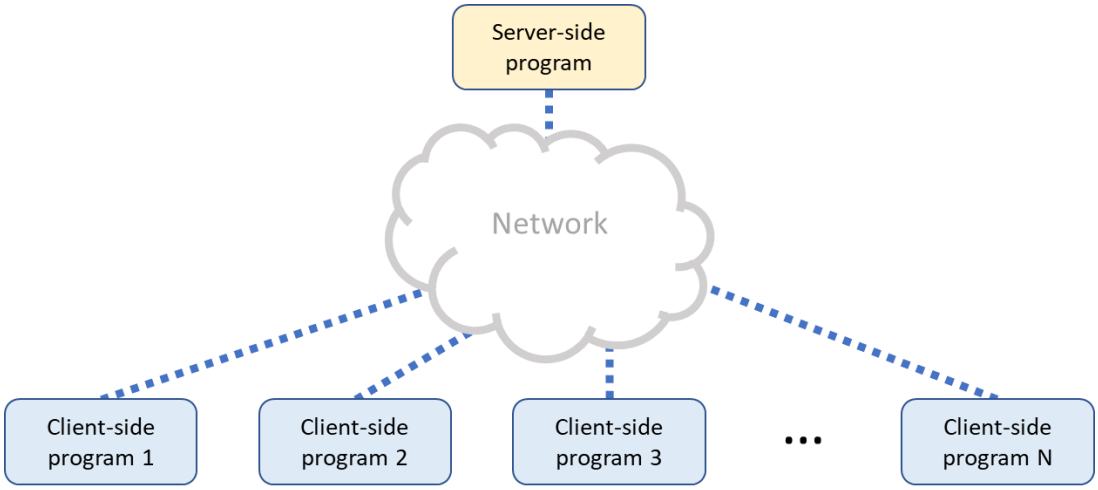
COMP3015
Data Communication and Networking
Socket Programming –
Transmission Control Protocol 2

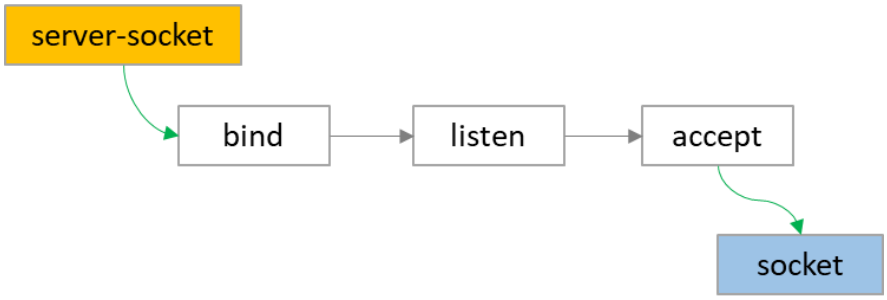
Server-side Program

The server-side program runs in the server machine, which establishes connections to the client-side program and transmits data between the client-side program and itself. A server-side program should be able to serve multiple clients.



Listening Connection Requests and Establishing Connections

When a server-side program starts, a **ServerSocket** instance is created and used for listening to the connection requests from clients. After a connection is established. A new **Socket** instance is created in the server-side program for transmitting data between the server-side program and client-side programs.



ServerSocket class

To listen to the connection requests, we first need the **ServerSocket** instance to bind to a free TCP port.

```
ServerSocket srvSocket = new ServerSocket(port);
```

Then, we execute its accept method to listen to the requests.

```
Socket clientSocket = srvSocket.accept();
```

After getting the socket, the server-side program then can communicate with the client-side program. The methods of sending and receiving data are the same as the methods we used in the client-side program.

The following class is a TCP server that listens to the connection requests from clients.

```
public class EchoServer {
ServerSocket srvSocket;

public EchoServer(int port) throws IOException {
   srvSocket = new ServerSocket(port);

   while(true) {
      System.out.printf("Listening at port %d...\n", port);
      Socket clientSocket = srvSocket.accept();
      serve(clientSocket);
   }
}

private void serve(Socket clientSocket) throws IOException {
   byte[] buffer = new byte[1024];
   System.out.printf("Established a connection to host %s:%d\n\n",
        clientSocket.getInetAddress(), clientSocket.getPort());

   DataInputStream in = new DataInputStream(clientSocket.getInputStream());
   DataOutputStream out = new DataOutputStream(clientSocket.getOutputStream());

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

   String str = "ECHO: " + new String(buffer, 0, len);

   out.writeInt(str.length());
   out.write(str.getBytes(), 0, str.length());

   clientSocket.close();
}
}
```

In the sample code above, we obtain the input stream and output stream respectively by calling the *getInputStream()* and *getOutputStream()* methods provided by the client socket. Then, we can use them to send and receive data, just like we write and read the content of a file.

You may download the sample programs from the following URL:

http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab4/

And, run them as follows:

1. Run the server-side program in a terminal/command prompt:

java EchoServer

2. Run the client-side program in a terminal/command prompt:

java EchoClient

You may make another test as follows and see what happened:

1. Run the server-side program.

2. Run the client-side program. Then, input the IP address and the port number but do not input the message. And, keep it running.

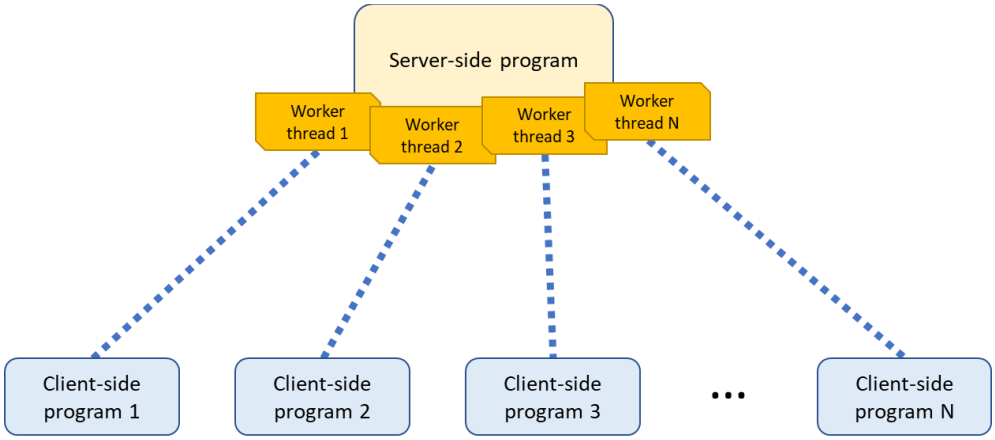
3. Run the client-side program again in another terminal (command prompt). Then, input the IP address, the port number, and a message. You should discover that no echo message will be replied from the server-side program.

4. If you go back to the first client-side program and input a message. Then, the server-side program sends the echo back to the first client-side program. Then, it sends the echo to the second client-side program.

Do you know why we get this result? It is because the server-side program can serve one client at a time. The *read()* method blocks the server-side program until the first client-side sends it a message. To serve multiple clients concurrently, the server-side program needs the **Multithreading** technique.

Multi-client Support and Multithreading

As mentioned at the beginning, the server-side program should be able to serve multiple clients. Therefore, the server-side program should have multiple threads and each thread serves a client. After the accept method of the server socket returned a socket, we need to create a new thread and pass the socket to the thread and let it serve the client. After that, the server socket is released and listens to the other connection requests.



Creating Child Thread

Thread class is used for creating a thread. One of its constructors accepts a lambda function. The lambda function will be executed when the *start()* method of the thread is invoked.

```
Thread t = new Thread(()->{

try {
   serve(cSocket);
} catch (IOException e) {
   System.err.println("Connection dropped!");
}

});

t.start();
```

We move the code for serving the client into the lambda function. Note that the lambda functions cannot throw any exceptions, so we need to use *try/catch* instead.

The main thread of the server-side program is used for handling the client connection requests only. Once the connection is established, a child thread will be created, and the client socket will be passed to the child thread.

You may convert the Echo Server program to the multi-client supported version. Then, test it with multiple clients.

Synchronization

The server-side program with multi-client support may generate data inconsistent programs because multiple threads may access the same object concurrently for reading or updating the values. The values may be overwritten incorrectly.

To avoid the kind of problems, we need to add a ***synchronized block*** to the object. The synchronized object can be accessed by one thread at a time. If another thread wants to access that synchronized object, that thread will be blocked until the previous thread finishes the access.

Let's study the following code:

```
public class SimpleChatServer {
  ServerSocket srvSocket;
  ArrayList<Socket> list = new ArrayList<Socket>();

  public SimpleChatServer(int port) throws IOException {
     srvSocket = new ServerSocket(port);

     while (true) {
        System.out.printf("Listening at port %d...\n", port);
        Socket cSocket = srvSocket.accept();

        synchronized(list) {
           list.add(cSocket);
           System.out.printf("Total %d clients are connected.\n", list.size());
        }

        Thread t = new Thread(() -> {
           try {
              serve(cSocket);
           } catch (IOException e) {
              System.err.println("connection dropped.");
           }
           synchronized(list) {
              list.remove(cSocket);
           }
        });
        t.start();
     }
  }

  private void serve(Socket clientSocket) throws IOException {
     byte[] buffer = new byte[1024];
     System.out.printf("Established a connection to host %s:%d\n\n",
          clientSocket.getInetAddress(), clientSocket.getPort());

     DataInputStream in = new DataInputStream(clientSocket.getInputStream());
     while (true) {
        int len = in.readInt();
        in.read(buffer, 0, len);
        forward(buffer, len);
     }
  }
```

```
  private void forward(byte[] data, int len) {
    synchronized (list) {
      for (int i = 0; i < list.size(); i++) {
        try {
          Socket socket = list.get(i);
          DataOutputStream out = new DataOutputStream(socket.getOutputStream());
          out.writeInt(len);
          out.write(data, 0, len);
        } catch (IOException e) {
          // the connection is dropped but the socket is not yet removed.
        }
      }
    }
  }

  public static void main(String[] args) throws IOException {
    new SimpleChatServer(12345);
  }

}
```

The simple chat server program supports multiple client connections. When a client connected, the socket for the new client will be added to a list, but another thread may be using the list for forwarding a message. To avoid the problem, we do:

```
synchronized(list) {
  list.add(cSocket);
  System.out.printf("Total %d clients are connected.\n", list.size());
}
```

We remove a corresponding socket from the list when the connection is dropped. Like the case above, we add a synchronized block to the list too.

```
synchronized(list) {
  list.remove(cSocket);
}
```

Besides, in the *forward()* method, sockets are retrieved one-by-one for forwarding messages. We need the synchronized block to guarantee that no other threads add new items to or delete existing items from the list.

Java provides the synchronized modifier for synchronizing a whole method. The usage is as follows:

```
public synchronized void doSomething(String[] someValues) {
  ...
}
```

Exercise

Download **SimpleChatServer.class** and **SimpleChatClient.class** from the following URL:

http://www.comp.hkbu.edu.hk/~mandel/comp3015/lab4/

Run the server-side program in a terminal/command prompt:

java SimpleChatServer

Run the client-side program twice in two terminals/command prompts:

java SimpleChatClient 127.0.0.1 12345

(change 127.0.0.1 to the IP address of the server if you use multiple computers to test the programs)

Following is the sample output of client A. Currently, we do not know who the senders of the messages are.

```
Please input messages:
Good morning          ← inputted by the user of client A
Good morning          ← forwarded message of client A
Hi!                   ← forwarded message of client B
How are you?          ← inputted by the user of client A
How are you?          ← forwarded message of client A
Fine! Thanks!         ← forwarded message of client B
```

Your task is to improve both client-side and server-side programs so that the user name will be shown at the beginning of each forwarded message in the client. But the messages will not be forwarded back to the senders.

```
Please input your name:
Alice
Please input messages:
Good morning
Bob: Hi!
How are you?
Bob: Fine! Thanks!
```

Alice sends “Good morning”. Alice’s message is forwarded to other users, but Alice will not receive it. Bob then sends “Hi!”. Bob’s message is forwarded. Of course, Bob does not receive his own message.

Here are the guidelines:

1. Download the source code files – **SimpleChatServer.java** and **SimpleChatClient.java**.
2. Create an Eclipse project and put the code files in.
3. Understand the program logic of both server-side and client-side programs and modify the codes.
4. Test the programs again (as mentioned above).
5. Then, zip your project and submit your zip file to Moodle.