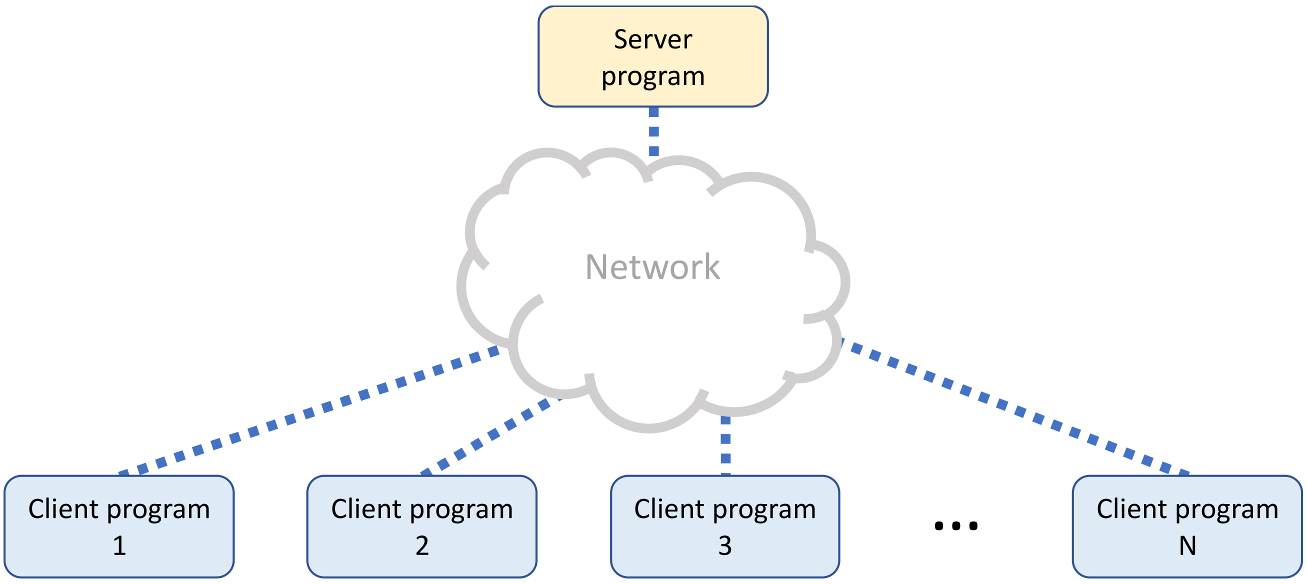
COMP3015
Data Communication and Networking
Socket Programming 2: Server Program and Synchronization

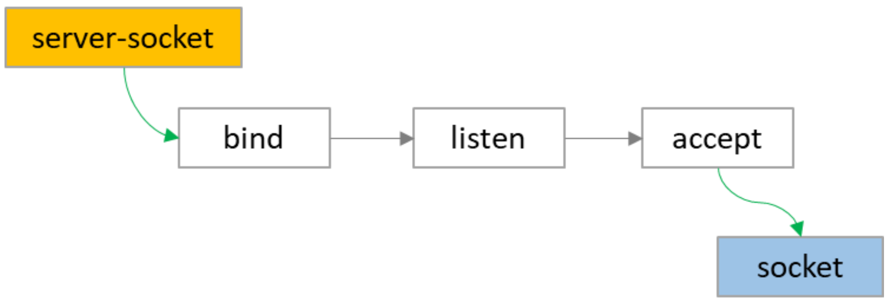
Server Program

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 EchoServer1 {

  public EchoServer1(int port) throws IOException {
     ServerSocket srvSocket = new ServerSocket(port);

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

  private void serve(Socket clientSocket) throws IOException {
    byte[] buffer = new byte[1024];
    print("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());

    while(true) {
        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());
    }
  }

  ...
}
```

Check the following files:

- EchoClient.java
- EchoServer1.java

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, compile them, and run them as follows:

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

```
java EchoServer1 12345
```

2. Run the client program in another terminal/command prompt:

```
java EchoClient 127.0.0.1 12345
```

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

1. Run the server program.

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

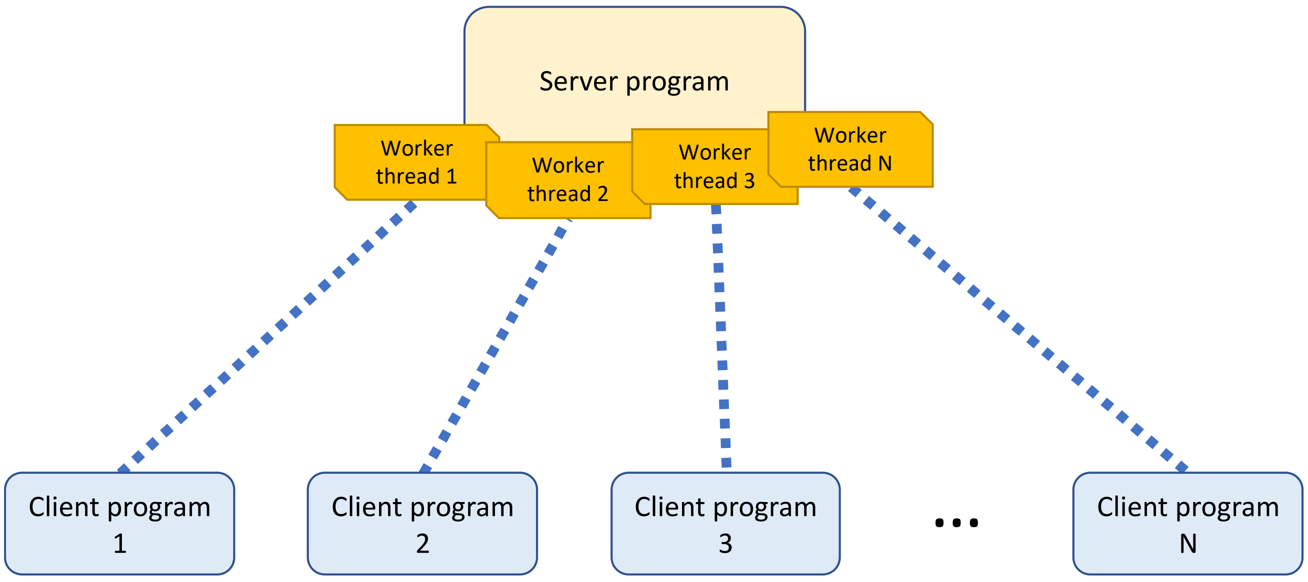
3. Run the client program again in another terminal (command prompt). Then, input the IP address, the port number, and a message. You should discover that no echo message will be replied from the server program.

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

Do you know why we get this result?

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 ex) {
     print("Connection drop!\n");
  }

});

t.start();
```

Check the following files:

- EchoClient.java
- EchoServer2.java

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 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 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:

Check the following files:

- EchoViewer.java
- EchoClient.java
- EchoServer3.java

```
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;

public class EchoServer3 {
    ArrayList<Socket> socketList = new ArrayList<Socket>();

    public EchoServer3(int port) throws IOException {
        ServerSocket srvSocket = new ServerSocket(port);

        while(true) {
            print("Listening at port %d...\n", port);
            Socket clientSocket = srvSocket.accept();

            synchronized (socketList) {
                socketList.add(clientSocket);
            }

            Thread t = new Thread(()-> {
                try {
                    serve(clientSocket);
                } catch (IOException ex) {
                    print("Connection drop!");
                }

                synchronized (socketList) {
                    socketList.remove(clientSocket);
                }

            });
            t.start();
        }
    }
```

EchoViewer is also a client program but it displays messages transferred from the server only.

The EchoServer3 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 put the add statement in a synchronized block.

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

```
    private void serve(Socket clientSocket) throws IOException {
        byte[] buffer = new byte[1024];
        print("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());

        while(true) {
            int size = in.readInt();
            String msg = "FORWARD: ";
            while(size > 0) {
                int len = in.read(buffer, 0, Math.min(size, buffer.length));
                msg += new String(buffer, 0, len);
                size -= len;
            }

            forward(msg);

        }
    }

    private void forward(String msg){
        synchronized (socketList) {
            for (Socket socket : socketList) {
                try {
                    DataOutputStream out = new DataOutputStream(
                            socket.getOutputStream());

                    out.writeInt(msg.length());
                    out.write(msg.getBytes(), 0, msg.length());
                } catch (IOException ex) {
                    print("Unable to forward message to %s:%d\n",
                        socket.getInetAddress().getHostName(),
                            socket.getPort());
                }
            }
        }
    }

    ...
}
```

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) {
  ...
}
```

Appendix

Checking IP Address and Network Address

If you want to know what IP address is used by your computer currently, you can use the following command in the command prompt (Windows) or terminal (macOS):

*Windows: **ipconfig***

Sample output:

```
Ethernet adapter Ethernet 3:

   Connection-specific DNS Suffix  . : comp.hkbu.edu.hk
   Link-local IPv6 Address . . . . . : fe90::74ba:7d29:dbaf:94df%12
   IPv4 Address. . . . . . . . . . . : 158.182.9.128
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 158.182.9.2
```

*macOS**: ifconfig en0***

Sample output:

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
 ether 38:f9:d3:96:e8:57
 inet6 fe80::c1b:ef2d:9aa1:d2bd%en0 prefixlen 64 secured scopeid 0xa
 inet 192.168.1.16 netmask 0xffffff00 broadcast 192.168.1.255
 nd6 options=201<PERFORMNUD,DAD>
 media: autoselect
 status: active
```

In the sample outputs, the IP addresses and subnet masks (netmasks) are highlighted in yellow and cyan respectively. The subnet marks of both sample outputs are the same, but they are represented in two different formats – 4 octets (**255.255.255.0**) and hexadecimal (**0x ff ff ff 00**).

255.255.255.0 ⬄ 0x ff ff ff 00 ⬄ /24

In the first output, the IP address is **158.182.9.128** and the subnet mask is **255.255.255.0**. Because the first three octets of the subnet mask are *255*, it means that the first three octets of the IP address are used for representing the network address. Therefore, the network address of that computer is **158.182.9.0**.

In the second output, the IP address is **192.168.1.16** and the netmask is **0 x ff ff ff 0** (255.255.255.0). Same as the first output, the first octets of the IP address are used for representing the network address. The network address of that computer is **192.168.1.0**.

Due to the network addresses of these two computers are different (**158.182.9.0** and **192.168.1.0**), **they are NOT in the same subnet**. Therefore, the broadcast packets cannot be transmitted from one to another.