ENSF 607

5 – Java Sockets

Java Sockets

Introduction

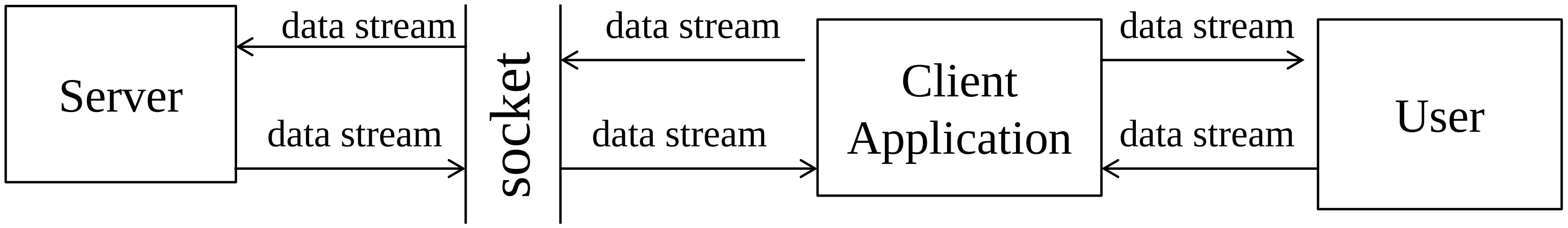
- Java `URLs` and `URLConnections` classes provide a **high-level mechanism** for accessing resources on the Internet.
- `Sockets` provide **lower-level network communication**.
  - Example: client-server application.

Client-Server Applications

- Simple examples of **client-server** applications are those used for processing database queries such as Airline Ticket Reservation Systems
  - The **clients** use the **services** provided by the **server**.
- **TCP (Transmission Control Protocol)**, provides a **communication channel** that client-server applications can communicate over the Internet.

Introduction

- **Client program** and a **server program** should establish a **connection** by binding a **socket** to its end.
  - The client and the server each reads from or writes to the socket



What is a Socket?

- **Sockets** are a one end-point of a two-way communication link between two programs.
- The **Communication Point** is identified by a combination of an **IP address** and a **port number**.
  - Multiple connections can be established between host and the server.
- **`java.net package`** provides classes **`Socket`** and **`ServerSocket`** for this purpose.
  - For Web applications, the URL classes are more appropriate. However, those classes use sockets for their implementations.

Steps to Build the Client Side

Build the Client-Side

To build the client-side, first you need to create a class (say Client), and take the following steps:

- Step 1:
  - Create a **Socket object**, indicating the host name, and the port number:

    ```
    Socket(String host, int port);
    ```

- Step 2
  - Obtain socket's input/output handle:

    ```
        Outputstream getOutputStream();
        Inputstream getOutputStream();
    ```

Build the Client-Side (cont'd)

- Step 3
  - Open streams on the socket, by passing socket I/O handles to:
    - BufferedReader (for reading from socket)
    - PrintWriter (for writing to socket)
- Step 4
  - Obtain standard input/output stream, if necessary to communicate with the user:
    - System.out.println (for writing messages to the user)
    - BufferedReader (for reading from keyboard)

Build the Client-Side (cont'd)

- Step 5
  - Start communicating between user, and server, using a while loop, as long as needed.
- Step 6
  - When no longer any communication is needed, **close all stream**.

Steps to Build the Server Side

Build the Server-Side

To build the server-side first you need to create a class (say Sever), and you need to take the following steps:

- Step 1:
  - Create a ServerSocket object, indicating a port number that is not used by other servers -- and the maximum number of clients:
    - Well-known port numbers: 80 for Web, 25 for email
    - Maximum by default is set to 50
    - Maximum is backlog

```
ServerSocket(int port) throws IOException;
ServerSocket(int port, int backlog) throws IOException;
```

Build the Server-Side (cont'd)

- Step 2
  - Accept a connection to a client by using the method:

    Socket accept();
- Step 3
  - Open streams on the socket, by passing socket I/O handles to:
    - BufferedReader (for reading from socket)
    - PrintWriter (for writing to socket)

Build the Server-Side (cont'd)

- Step 4
  - Start communicating with client(s), using a while loop, as long as needed.
- Step 5
  - When no longer any communication is neede,d **close all stream.**

Lets Look at Some Code

Example

Develop a client that:

- As long as the user wants:
  - reads user's input (say a word).
  - write the input to the socket
  - reads server's result
  - displays the result on the screen

Develop a server that:

- As long as needed:
  - reads from input (say a word).
  - capitalizes the input
  - writes the server's result on the socket

Solution posted on D2L

Serving Multiple Clients

- Real world servers need to handle multiple client.
  - Client connection requests are queued at the port, so the server must accept the connections sequentially.
- Or, The server can service them simultaneously through the use of threads.
  - one thread per each client.

Serving Multiple Clients

- Flow of logic to serve multiple client is:

```
while (true) {
     accept a new client (new connection);
     create a new thread ;
  }
```

- A server may also need to transfer data from one client to one or more clients.

.