Elementary Sockets

Lecture 3 (B)

Topic Outline

- Introduction to TCP and UDP sockets
- **TCP and UDP client-server programs**
- SCTP association
- I/O multiplexing and non-blocking I/O
- Socket options
- **Remote Method Invocation**
- Name and address conversions

Stream Socket vs. Datagram Socket

Stream socket	• A dedicated point-to-point channel between a client and server. • Use TCP (Transmission Control Protocol) for data transmission. • Lossless and reliable. • Sent and received in the same order.
Datagram socket	• No dedicated point-to-point channel between a client and server. • Use UDP (User Datagram Protocol) for data transmission. • May lose data and not 100% reliable. • Data may not received in the same order as sent.

?

- When to use **Stream Socket** & when to use **Datagram Socket**?

DatagramPacket

- The **`DatagramPacket`** class represents a datagram packet.
- Datagram packets are used to implement a connectionless packet delivery service.
- Each message is routed from one machine to another based solely on information contained within the packet.

java.net.DatagramPacket	
length: int	A JavaBeans property to specify the length of buffer.
address: InetAddress	A JavaBeans property to specify the address of the machine where the package is sent or received.
port: int	A JavaBeans property to specify the port of the machine where the package is sent or received.
+DatagramPacket(buf: byte[], length: int, host: InetAddress, port: int)	Constructs a datagram packet in a byte array buf of the specified length with the host and the port for which the packet is sent. This constructor is often used to construct a packet for delivery from a client.
+DatagramPacket(buf: byte[], length: int)	Constructs a datagram packet in a byte array buf of the specified length.
+getData(): byte[]	Returns the data from the package.
+setData(buf: byte[]): void	Sets the data in the package.

DatagramSocket

DatagramSocket

The **`DatagramSocket`** class represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

Create a server DatagramSocket

To create a server **`DatagramSocket`**, use the constructor **`DatagramSocket(int port)`**, which binds the socket with the specified port on the local host machine.

Create a client DatagramSocket

To create a client **`DatagramSocket`**, use the constructor **`DatagramSocket()`**, which binds the socket with any available port on the local host machine.

Sending and Receiving a DatagramSocket

Sending

To send data, you need to create a packet, fill in the contents, specify the Internet address and port number for the receiver, and invoke the **`send(packet)`** method on a **`DatagramSocket`**.

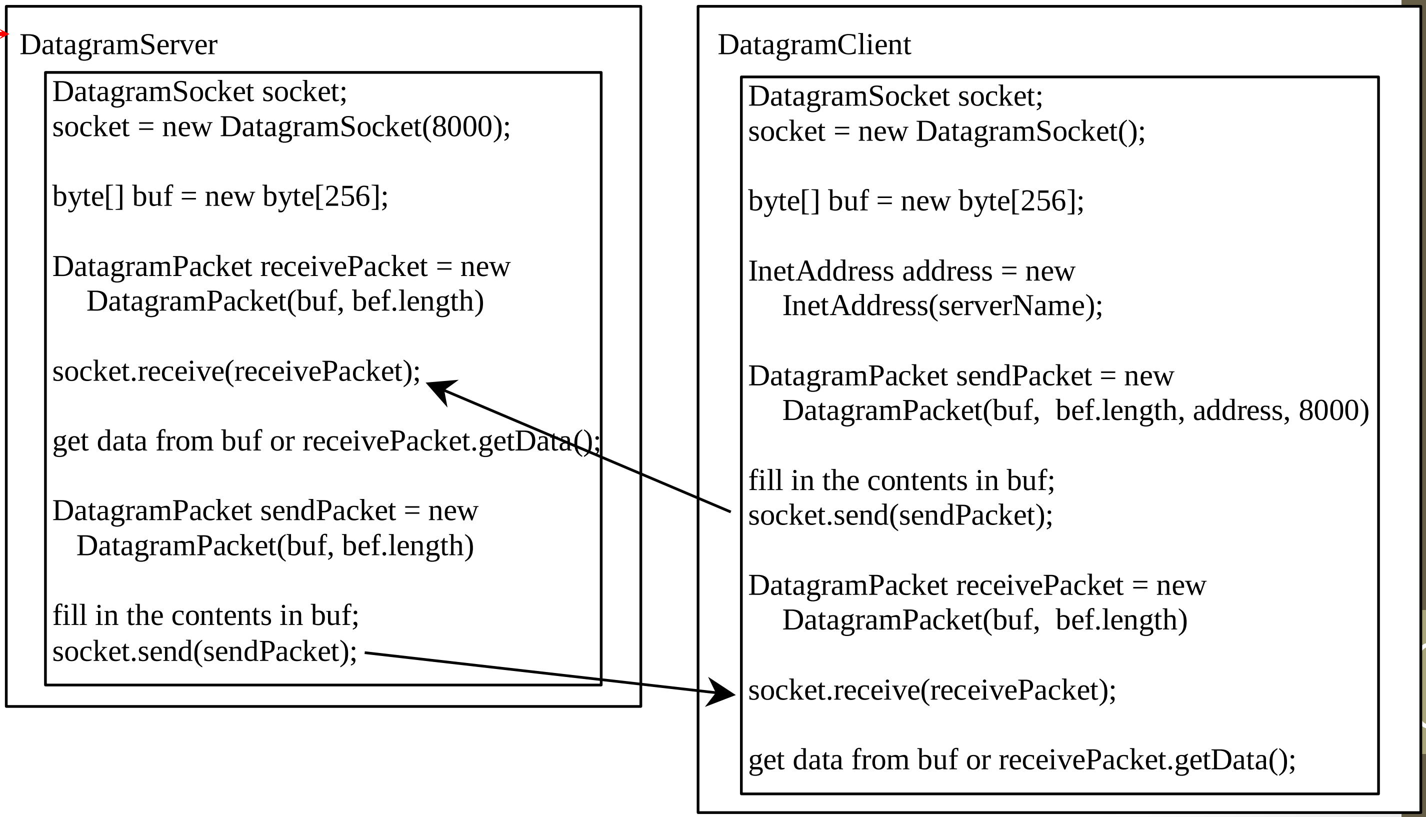
Receiving

To receive data, create an empty packet and invoke the **`receive(packet)`** method on a **`DatagramSocket`**.

Datagram Programming

Datagram programming is different from stream socket programming in the sense that there is **no concept** of a **`ServerSocket`** for datagrams. Both client and server use **`DatagramSocket`** to send and receive packets.

Designate one a server



```
DatagramSocket socket;
socket = new DatagramSocket(8000);

byte[] buf = new byte[256];

DatagramPacket receivePacket = new
    DatagramPacket(buf, bef.length)

socket.receive(receivePacket);

get data from buf or receivePacket.getData();

DatagramPacket sendPacket = new
   DatagramPacket(buf, bef.length)

fill in the contents in buf;
socket.send(sendPacket);
```

```
DatagramSocket socket;
socket = new DatagramSocket();

byte[] buf = new byte[256];

InetAddress address = new
    InetAddress(serverName);

DatagramPacket sendPacket = new
    DatagramPacket(buf,  bef.length, address, 8000)

fill in the contents in buf;
socket.send(sendPacket);

DatagramPacket receivePacket = new
    DatagramPacket(buf,  bef.length)

socket.receive(receivePacket);

get data from buf or receivePacket.getData();
```

Sample Programs

- Random Quotes
  - Server: **UDPQuoteServer.java**
  - Client: **RandomQuoteClient.java**

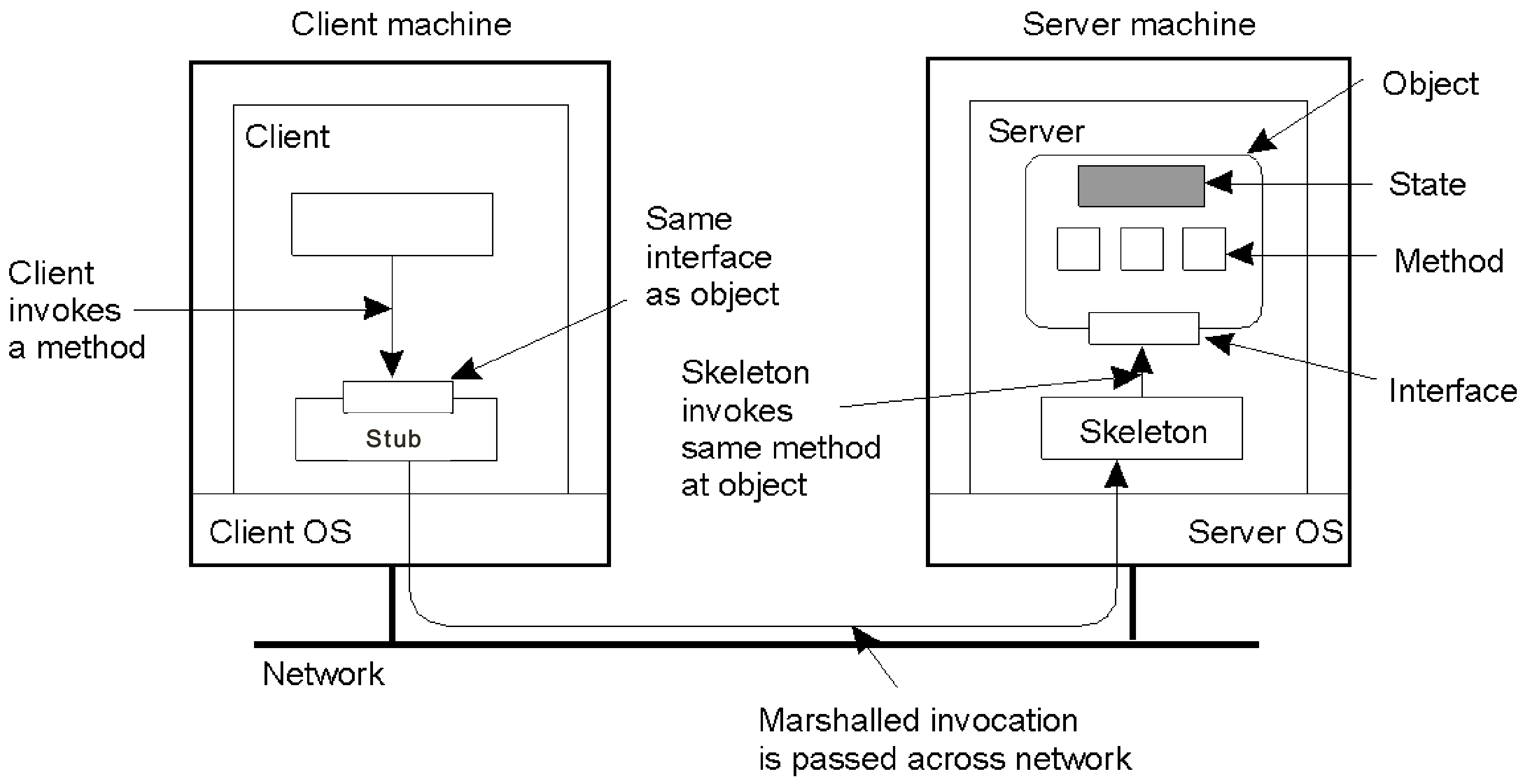
- Geometry (Mid-point of 2 lines)
  - Server: **UDPGeometryServer.java**
  - Client: **UDPGeometryClient.java**
  - Class files: **Point.java**, **Line.java**

Remote Method Invocation (RMI)

- Provides support for **distributed objects** in Java
  - Allows object to invoke methods of remote objects
  - Calling objects use the exact same syntax as for local invocations
  - Also an object-oriented equivalent for RPC

- 2 general requirements of Java RMI model
  - RMI model shall be simple & easy to use
  - Model shall fit into the Java language in a natural way

Distributed Objects



Stub

- A client-side object that represents a single server object inside the client's JVM.
- Implements the same methods as the server object
- Maintains a socket connection to the server object's JVM automatically
- Responsible for marshalling and demarshalling data on the client side

Skeleton

- A server-side object responsible for maintaining network connections
- Also responsible for marshalling and demarshalling data on the server side.

RMI Client-Server Communication

1. The client obtains an instance of the stub class. The stub class is automatically pre-generated from the target server class and implements all the methods that the server class implements.

2. The client calls a method on the stub. The method call is actually the same method call the client would make on the server object if both objects resided in the same JVM.

RMI Client-Server Communication (continued)

3. Internally, the stub either creates a socket connection to the skeleton on the server or reuses a pre-existing connection. It marshalls all the information associated to the method call, including the name of the method and the arguments, and sends this information over the socket connection to the skeleton.

RMI Client-Server Communication (continued)

4. The skeleton demarshalls the data and makes the method call on the actual server object. It gets a return value back from the actual server object, marshalls the return value, and sends it over the wire to the stub.

5. The stub demarshalls the return value and returns it to the client code.

Steps in Writing Java RMI Programs

- Create the Interface to the server
- Create the Server
- Create the Client
- Compile the Interface (javac)
- Compile the Server (javac)
- Compile the Client (javac)
- Start the RMI Registry (rmiregistry)
- Start the RMI Server
- Start the RMI Client

Sample Program

- **Interface**
  - LineService.java
- **RMI Server**
  - LineRMIServer.java
- **RMI Client**
  - MidPointCalculator.java
- **Required classes**
  - Line.java
  - Point.java

Running the RMI Server Program

- The following class files are required:
  - Interface file (**LineService.class**)
  - RMI Server program **(LineRMIServer.class)**
  - Required classes **(Line.class**, **Point.class)**

- Steps to run RMI Server Program
  - Start the RMI Registry
    - **start rmiregistry**

  - Run the program
    - **java LineRMIServer**

Running the RMI Client Program

- The following class files are required
  - Interface file (**LineService.class**)
  - RMI Client program **(MidPointCalculator.class** & **any inner classes of this class)**
  - Required classes **(Point.class)**

- Security policy file
  - See **rmi.sec** in sample source

- Running the RMI client:
  - **java -Djava.security.policy=rmi.sec MidPointCalculator localhost**