CS450 Computer Networks

The slides used in class are derived from the slides available on our text book companion website:
http://wps.pearsoned.com/ecs_kurose_compnetw_6/
copyright 1996-2012   J.F Kurose and K.W. Ross

CS450 Computer Networks

Lesson 3

Application Layer – Socket Programming

Knowledge is for action, action is for achievement, achievement is for fulfillment

Lesson 3: Application layer – Socket Programming

Our Goal:

Learn how to build client/server applications that communicate using sockets

- Socket programming using TCP
- Socket programming using UDP

Socket programming

Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

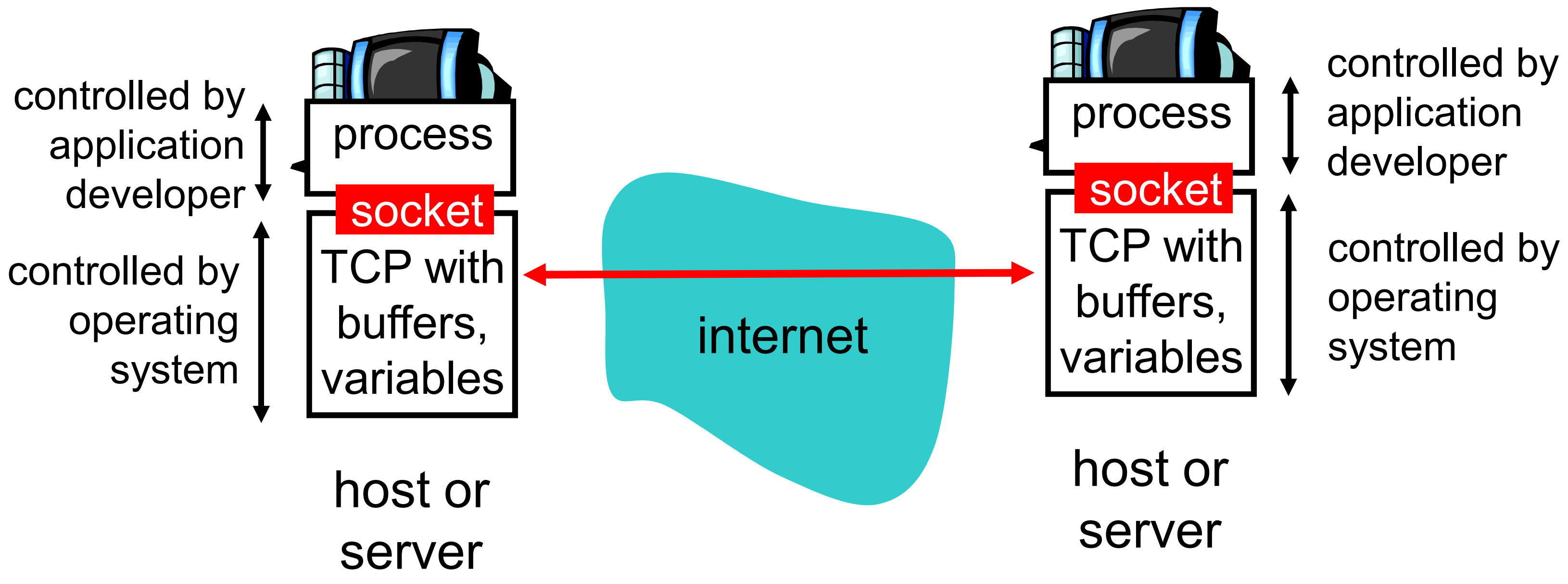
socket

a *host-local*, *application-created*, *OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of *bytes* from one process to another


controlled by application developer
controlled by operating system
process
socket
TCP with buffers, variables
host or server
internet
process
socket
TCP with buffers, variables
host or server
controlled by application developer
controlled by operating system

Socket programming *with TCP*

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

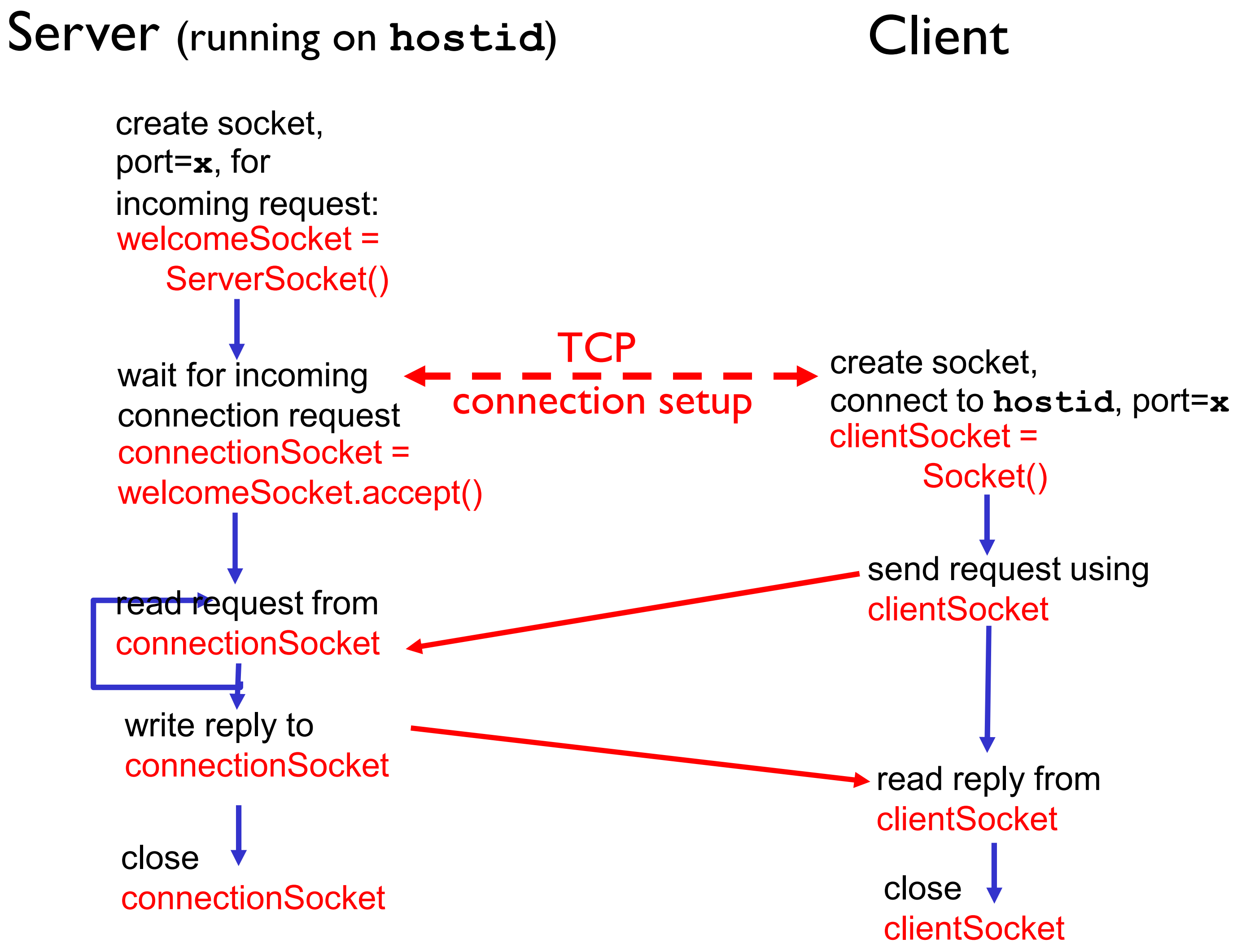
- creating client-local TCP socket
- specifying IP address, port number of server process
- when client creates socket: client TCP establishes connection to server TCP

- when contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint

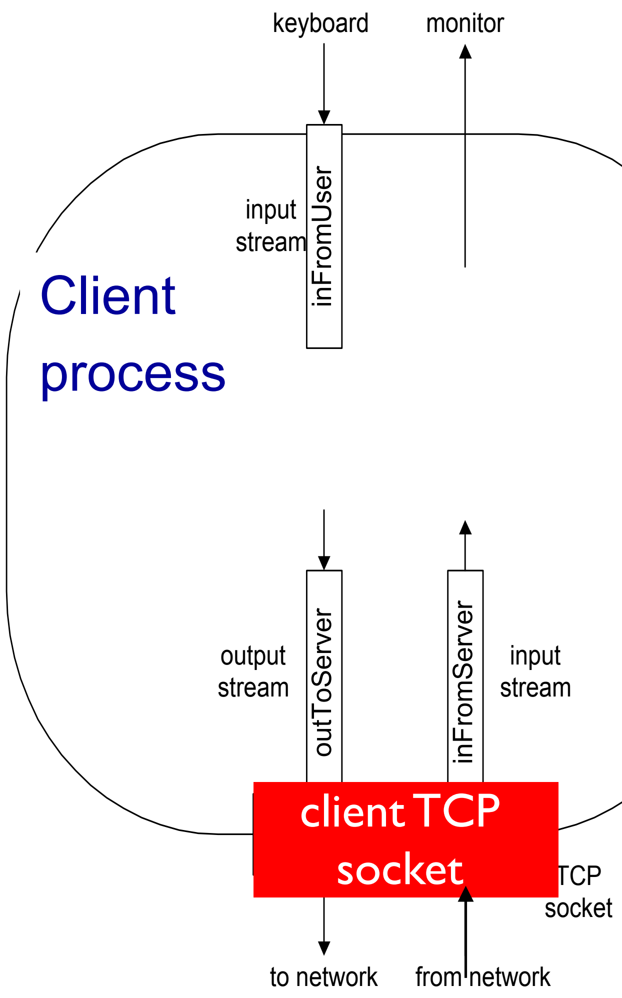
TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

Client/server socket interaction: TCP


Server (running on **hostid**)
Client
create socket, port=**x**, for incoming request:
welcomeSocket = ServerSocket()
wait for incoming connection request
connectionSocket = welcomeSocket.accept()
TCP connection setup
create socket, connect to **hostid**, port=**x**
clientSocket = Socket()
send request using clientSocket
read request from connectionSocket
write reply to connectionSocket
read reply from clientSocket
close connectionSocket
close clientSocket

Stream jargon

- stream is a sequence of characters that flow into or out of a process.
- input stream is attached to some input source for the process, e.g., keyboard or socket.
- output stream is attached to an output source, e.g., monitor or socket.



Socket programming with TCP

Example client-server app:

1) client reads line from standard input (**`inFromUser`** stream) , sends to server via socket (**`outToServer`** stream)

2) server reads line from socket

3) server converts line to uppercase, sends back to client

4) client reads, prints modified line from socket (**`inFromServer`** stream)

Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

This package defines Socket() and ServerSocket() classes

server name, e.g., www.umass.edu

server port #

create input stream

create clientSocket object of type Socket, connect to server

create output stream attached to socket

Example: Java client (TCP), cont.

create input stream attached to socket →

```
    BufferedReader inFromServer =
      new BufferedReader(new
      InputStreamReader(clientSocket.getInputStream()));
```

```
    sentence = inFromUser.readLine();
```

send line to server →

```
    outToServer.writeBytes(sentence + '\n');
```

read line from server →

```
    modifiedSentence = inFromServer.readLine();

    System.out.println("FROM SERVER: " + modifiedSentence);
```

close socket (clean up behind yourself!) →

```
    clientSocket.close();

  }
}
```

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
    {
      String clientSentence;
      String capitalizedSentence;

      ServerSocket welcomeSocket = new ServerSocket(6789);

      while(true) {

         Socket connectionSocket = welcomeSocket.accept();

           BufferedReader inFromClient =
             new BufferedReader(new
             InputStreamReader(connectionSocket.getInputStream()));
```

create welcoming socket at port 6789 → ServerSocket welcomeSocket = new ServerSocket(6789);

wait, on welcoming socket accept() method for client contact create, *new* socket on return → Socket connectionSocket = welcomeSocket.accept();

create input stream, attached to socket → BufferedReader inFromClient = …

Example: Java server (TCP), cont

create output stream, attached to socket →

```
DataOutputStream  outToClient =
  new DataOutputStream(connectionSocket.getOutputStream());
```

read in line from socket →

```
clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '\n';
```

write out line to socket →

```
outToClient.writeBytes(capitalizedSentence);
   }
  }
 }
```

end of while loop, loop back and wait for another client connection

Socket programming *with UDP*

UDP: no "connection" between client and server

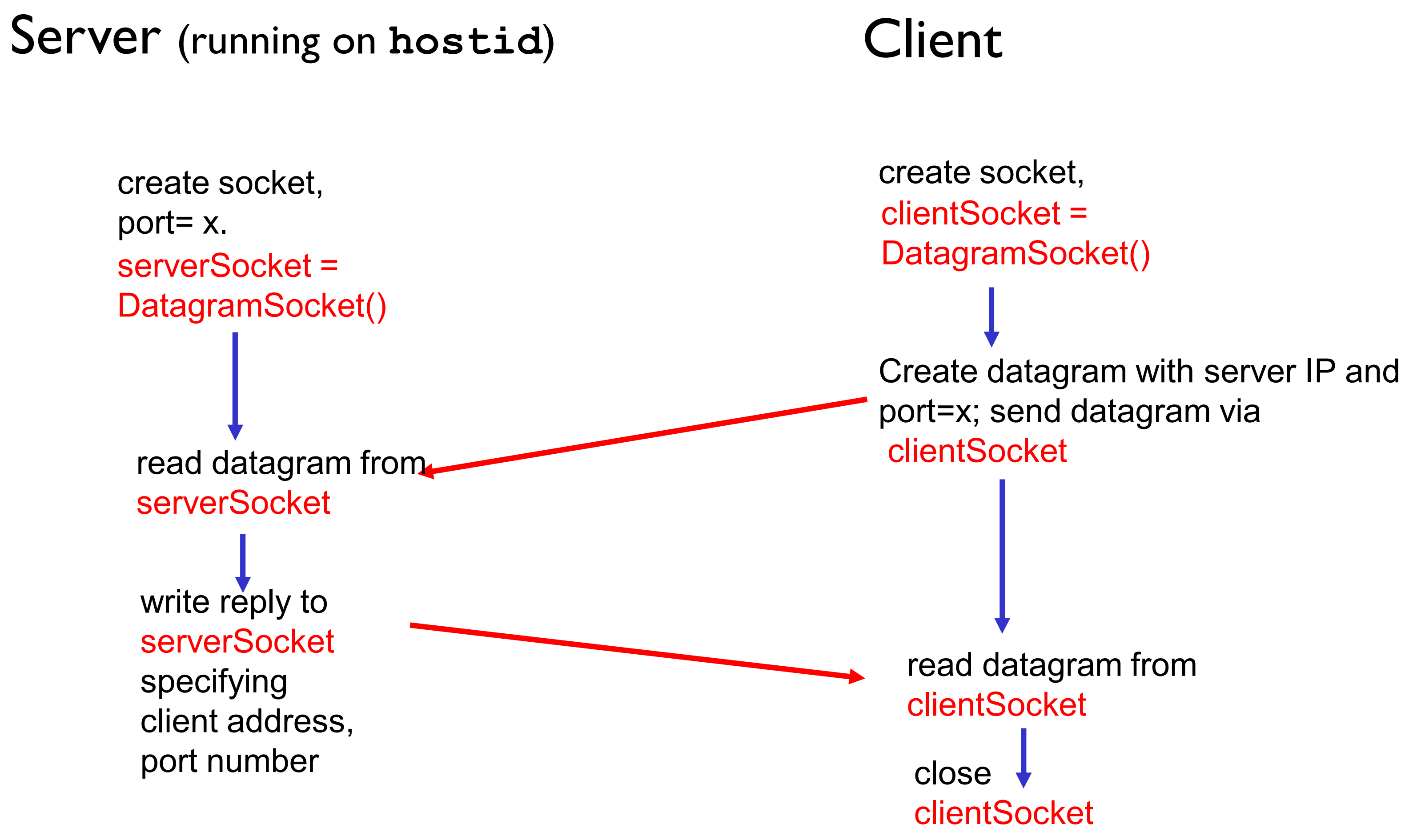
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

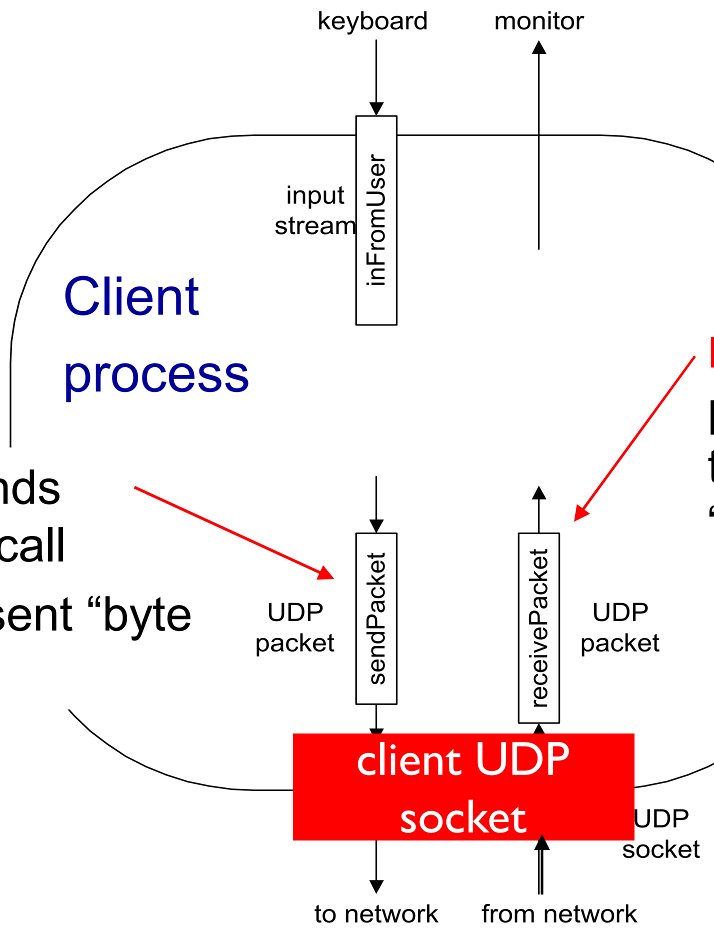
application viewpoint:

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP


Server (running on **hostid**)
Client
create socket, port= x. serverSocket = DatagramSocket()
read datagram from serverSocket
write reply to serverSocket specifying client address, port number
create socket, clientSocket = DatagramSocket()
Create datagram with server IP and port=x; send datagram via clientSocket
read datagram from clientSocket
close clientSocket

Example: Java client (UDP)



Output: sends packet (recall that TCP sent "byte stream")

Input: receives packet (recall thatTCP received "byte stream")

Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception
    {

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

      byte[] sendData = new byte[1024];
      byte[] receiveData = new byte[1024];

      String sentence = inFromUser.readLine();
      sendData = sentence.getBytes();
```

create input stream

create client socket

translate hostname to IP address using DNS

Example: Java client (UDP), cont.

create datagram with data-to-send, length, IP addr, port

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
```

send datagram to server

```
clientSocket.send(sendPacket);

DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
    new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}

}
```

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {

      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {

          DatagramPacket receivePacket =
             new DatagramPacket(receiveData, receiveData.length);
           serverSocket.receive(receivePacket);
```

create datagram socket at port 9876

create space for received datagram

receive datagram

Example: Java server (UDP), cont

```
          String sentence = new String(receivePacket.getData());

          InetAddress IPAddress = receivePacket.getAddress();

          int port = receivePacket.getPort();

                  String capitalizedSentence = sentence.toUpperCase();

          sendData = capitalizedSentence.getBytes();

          DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress,
                               port);

          serverSocket.send(sendPacket);
        }
     }
  }
```

get IP addr port #, of sender

create datagram to send to client

write out datagram to socket

end of while loop, loop back and wait for another datagram

Lesson 3: Summary

- *TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*
- *UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server*