**Application-Layer Protocols**

**Overview**

- Application-layer protocols define:
  - The types of messages exchanged
  - The syntax and semantics of messages
  - The rules for when and how messages are sent

- Public protocols (defined in RFCs)
  - HTTP, FTP, SMTP, POP, IMAP, DNS

- Proprietary protocols
  - RealAudio, RealVideo
  - IP telephony
  - …
**Application-Layer Protocols**

**Outline**

- The architecture of distributed systems
  - Client/Server computing
- The programming model used in constructing distributed systems
  - Socket programming
- Example client/server systems and their application-level protocols
  - The World-Wide Web (HTTP)
  - Reliable file transfer (FTP)
  - E-mail (SMTP & POP)
  - Internet Domain Name System (DNS)

**Application-Layer Protocols**

**Outline**

- Example client/server systems and their application-level protocols
  - The World-Wide Web (HTTP)
  - Reliable file transfer (FTP)
  - E-mail (SMTP & POP)
  - Internet Domain Name System (DNS)
- Protocol design issues:
  - In-band vs. out-of-band control signaling
  - Push vs. pull protocols
  - Persistent vs. non-persistent connections
- Client/server service architectures
  - Contacted server responds vs. forwards request
**The Application Layer**

The client-server paradigm

- Typical network application has two pieces: *client* and *server*
- **Client:**
  - Initiates contact with server ("speaks first")
  - Requests service from server
  - For Web, client is implemented in browser; for e-mail, in mail reader
- **Server:**
  - Provides requested service to client
  - "Always" running
  - May also include a "client interface"

**Client/Server Paradigm**

Socket programming

- Sockets are the fundamental building block for client/server systems
- Sockets are created and managed by applications
  - Strong analogies with files
- Two types of transport services are available via the socket API:
  - UDP sockets: unreliable, datagram-oriented communications
  - TCP sockets: reliable, stream-oriented communications
**Client/Server Paradigm**

Socket-programming using TCP

- A socket is an application created, OS-controlled interface into which an application can both send and receive messages to and from another application.
  - A "door" between application processes and end-to-end transport protocols.

**Socket-programming using TCP**

TCP socket programming model

- A TCP socket provides a reliable, bi-directional, byte-stream communications channel from one process to another.
  - A "pair of pipes" abstraction.
Socket-programming using TCP

Network addressing for sockets

- Sockets are addressed using an IP address and port number

Socket-programming using TCP

Socket programming in Java

- Client creates a local TCP socket specifying the host and port number of server process
  - Java resolves host names to IP addresses using DNS
- Client contacts server
  - Server process must be running
  - Server must have created socket that "welcomes" client’s contact
- When the client creates a socket, the client’s TCP establishes connection to server’s TCP
- When contacted by a client, server creates a new socket for server process to communicate with client
  - This allows the server to talk with multiple clients
- The client reads a line of text from standard input and sends the text to the server via a socket.
- The server receives the line of text from the client and converts the line of characters to all uppercase.
- The server sends the converted line back to the client.
- The client receives the converted text and writes it to standard output.
Socket programming with TCP Example

**Client structure**

- Client reads from standard input (inFromUser stream), writes to server via a socket (outToServer stream)
- Server reads line from a socket
- Server converts line to uppercase and writes back to client
- Client reads from socket, (inFromServer stream) prints modified line to standard output

**Client/Server TCP socket interaction in Java**

Server (running on swan.cs.unc.edu)

1. create socket for incoming request (port=6789)
2. welcomeSocket = new ServerSocket(...)

Client (running on classroom.cs...)

1. create socket, connect to swan.cs.unc.edu, port=6789
2. clientSocket = new Socket(...)
3. connectionSocket = clientSocket.accept()
4. read request from connectionSocket
5. write reply to connectionSocket
6. close connectionSocket

**Diagram**

[Diagram showing the interaction between the client and server.]
Socket programming with TCP Example
Java client

```java
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String[] args) throws Exception {
        String sentence;
        String modifiedSentence;

        // Create (buffered) input stream using standard input
        BufferedReader inFromUser = new BufferedReader(
            new InputStreamReader(System.in));

        System.out.println("Client ready for input");

        // While loop to read and handle multiple input lines
        while ((sentence = inFromUser.readLine()) != null) {
            // Create client socket with connection to server at port 6789
            Socket clientSocket = new Socket("swan.cs.unc.edu", 6789);

            // Create output stream attached to socket
            DataOutputStream outToServer = new DataOutputStream(
                clientSocket.getOutputStream());

            // Create (buffered) input stream attached to socket
            BufferedReader inFromServer = new BufferedReader(
                new InputStreamReader(
                    clientSocket.getInputStream()));

            // Write line to server
            outToServer.writeBytes(sentence + 
                System.getProperty("line.separator");

            // Read line from server
            String modifiedSentence = inFromServer.readLine();

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

            // end while, loop to accept more lines from user
        }

        // end main
    }
}
```

Socket programming with TCP Example
Java client II

```java
// Create output stream attached to socket
DataOutputStream outToServer = new DataOutputStream(
    clientSocket.getOutputStream());

// Create (buffered) input stream attached to socket
BufferedReader inFromServer = new BufferedReader(
    new InputStreamReader(
        clientSocket.getInputStream()));

// Write line to server
outToServer.writeBytes(sentence + 
    System.getProperty("line.separator");

// Read line from server
String modifiedSentence = inFromServer.readLine();

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

// end while, loop to accept more lines from user

// end main
```

// end class
Socket programming with TCP Example

Java server

```java
import java.io.*;
import java.net.*;
public class TCPServer {
    public static void main(String[] args) throws Exception {
        String clientSentence;
        String capitalizedSentence;

        // Create "welcoming" socket using port 6789
        ServerSocket welcomeSocket = new ServerSocket(6789);

        System.out.println("Server Ready for Connection");

        // While loop to handle arbitrary sequence of clients making requests
        while (true) {

            // Waits for a client to connect and creates new socket for connection
            Socket connectionSocket = welcomeSocket.accept();

            System.out.println("Client Made Connection");

            // Create (buffered) input stream attached to connection socket
            BufferedReader inFromClient = new BufferedReader(new InputStreamReader(connectionSocket.getInputStream()));

            // Create output stream attached to connection socket
            DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());

            // Read input line from socket
            String clientSentence = inFromClient.readLine();

            System.out.println("Client sent: " + clientSentence);
            capitalizedSentence = clientSentence.toUpperCase() + "\n";

            // Write output line to socket
            outToClient.writeBytes(capitalizedSentence);

            connectionSocket.close();
        }
    }
}
```
Socket-programming using UDP

UDP socket programming model

- A UDP socket provides an unreliable bi-directional communication channel from one process to another
  - A “datagram” abstraction

Socket programming with UDP Example

Client/server UDP socket interaction in Java

Server (running on 152.2.131.245)

- create socket for incoming request (port=9676)
- serverSocket = new DatagramSocket()
- read request from serverSocket
- write reply to serverSocket specifying client IP address and port number
- close clientSocket

Client

- create socket, clientSocket = new DatagramSocket()
- create address (152.2.131.245, port = 9676) and send data using clientSocket
- read reply from clientSocket
- close clientSocket
Socket Programming
Services provided by Internet transport protocols

- **TCP service:**
  - connection-oriented: setup required between client, server
  - reliable transport between sending and receiving process
  - flow control: sender won’t overwhelm receiver
  - congestion control: throttle sender when network overloaded
  - does not provide: timing, minimum bandwidth guarantees

- **UDP service:**
  - unreliable data transfer between sending and receiving process
  - does not provide: connection setup, reliability, flow control, congestion control, timing, or minimum bandwidth guarantees

**Why bother? Why is there a UDP?**