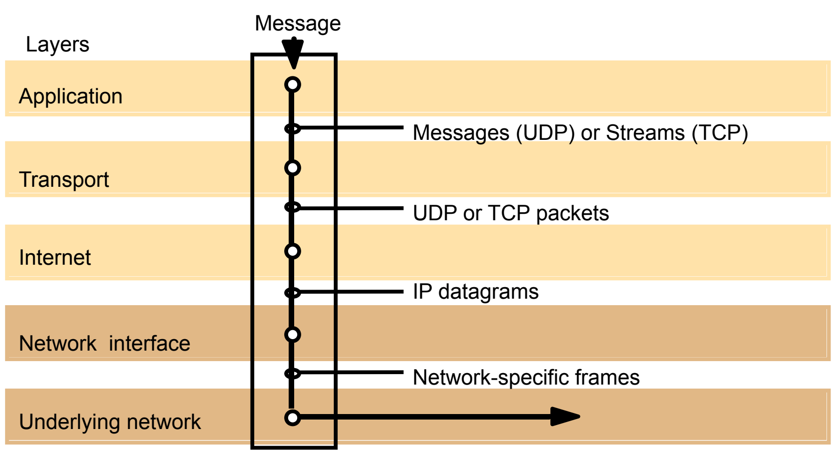
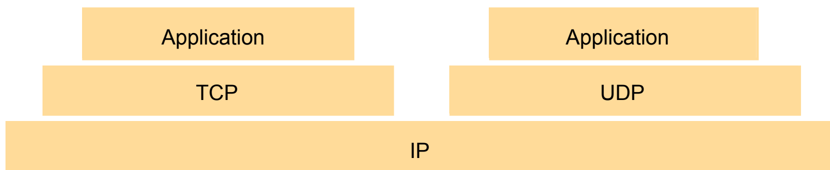
Network Programming using sockets

TCP/IP layers



2

The programmer's conceptual view of a TCP/IP Internet



3

A Programmer's View of the Internet

1. Hosts are mapped to a set of 32-bit ***IP addresses***.
   - 128.2.203.179

2. The set of IP addresses is mapped to a set of identifiers called Internet ***domain names***.
   - 128.2.203.179 is mapped to www.cs.cmu.edu

3. A process on one Internet host can communicate with a process on another Internet host over a ***connection***.

4

IP Addresses

32-bit IP addresses are stored in an *IP address struct*

- IP addresses are always stored in memory in network byte order (big-endian byte order)
- True in general for any integer transferred in a packet header from one machine to another.
  - E.g., the port number used to identify an Internet connection.

```
/* Internet address structure */
struct in_addr {
    unsigned int s_addr; /* network byte order (big-endian) */
};
```

Handy network byte-order conversion functions:

`htonl`: convert uint32_t from host to network byte order.
`htons`: convert uint16_t from host to network byte order.
`ntohl`: convert uint32_t from network to host byte order.
`ntohs`: convert uint16_t from network to host byte order.

5

Dotted Decimal Notation

By convention, each byte in a 32-bit IP address is represented by its decimal value and separated by a period

- IP address `0x8002C2F2 = 128.2.194.242`

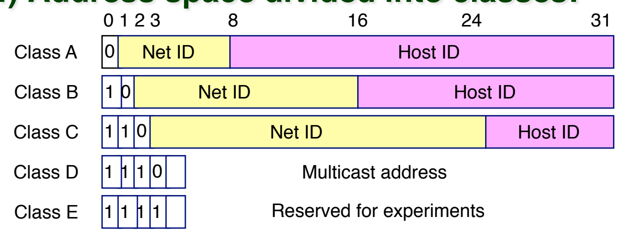
Functions for converting between binary IP addresses and dotted decimal strings:

- `inet_aton`: converts a dotted decimal string to an IP address in network byte order.
- `inet_ntoa`: converts an IP address in network byte order to its corresponding dotted decimal string.
- "n" denotes network representation. "a" denotes application representation.

6

IP Address Structure

IP (V4) Address space divided into classes:



Network ID Written in form w.x.y.z/n

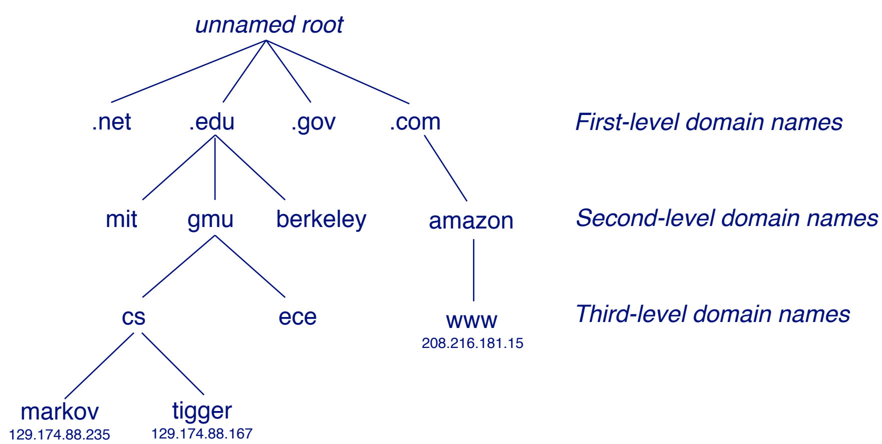
- **n = number of bits in host address**
- **E.g., GMU written as 129.174.0.0/16**
  - **Class B address**

Unrouted (private) IP addresses:

10.0.0.0/8   172.16.0.0/12   192.168.0.0/16

7

Internet Domain Names



8

Domain Naming System (DNS)

The Internet maintains a mapping between IP addresses and domain names in a huge worldwide distributed database called *DNS*.

- Conceptually, programmers can view the DNS database as a collection of millions of *host entry structures*:

```
/* DNS host entry structure */
struct hostent {
   char   *h_name;       /* official domain name of host */
   char   **h_aliases;   /* null-terminated array of domain names */
   int    h_addrtype;    /* host address type (AF_INET) */
   int    h_length;      /* length of an address, in bytes */
   char   **h_addr_list; /* null-terminated array of in_addr structs */
};
```

Functions for retrieving host entries from DNS:

- `gethostbyname`: query key is a DNS domain name.
- `gethostbyaddr`: query key is an IP address.

9

Properties of DNS Host Entries

Each host entry is an equivalence class of domain names and IP addresses.

Each host has a locally defined domain name `localhost` which always maps to the *loopback address* `127.0.0.1`

Different kinds of mappings are possible:

- Simple case: 1-1 mapping between domain name and IP addr:
  - `kittyhawk.cmcl.cs.cmu.edu` maps to `128.2.194.242`
- Multiple domain names mapped to the same IP address:
  - `eecs.mit.edu` and `cs.mit.edu` both map to `18.62.1.6`
- Multiple domain names mapped to multiple IP addresses:
  - `aol.com` and `www.aol.com` map to multiple IP addrs.
- Some valid domain names don't map to any IP address:
  - for example: `cmcl.cs.cmu.edu`

10

A Program That Queries DNS

```
int main(int argc, char **argv) { /* argv[1] is a domain name */
    char **pp;                    /* or dotted decimal IP addr */
    struct in_addr addr;
    struct hostent *hostp;

    if (inet_aton(argv[1], &addr) != 0)
        hostp = Gethostbyaddr((const char *)&addr, sizeof(addr),
                AF_INET);
    else
        hostp = Gethostbyname(argv[1]);
    printf("official hostname: %s\n", hostp->h_name);

    for (pp = hostp->h_aliases; *pp != NULL; pp++)
        printf("alias: %s\n", *pp);

    for (pp = hostp->h_addr_list; *pp != NULL; pp++) {
        addr.s_addr = ((struct in_addr *)*pp)->s_addr;
        printf("address: %s\n", inet_ntoa(addr));
    }
}
```


Querying DNS from the Command Line

Domain Information Groper (`dig`) provides a scriptable command line interface to DNS.

```
linux> dig +short kittyhawk.cmcl.cs.cmu.edu
128.2.194.242
linux> dig +short -x 128.2.194.242
KITTYHAWK.CMCL.CS.CMU.EDU.
linux> dig +short aol.com
205.188.145.215
205.188.160.121
64.12.149.24
64.12.187.25
linux> dig +short -x 64.12.187.25
aol-v5.websys.aol.com.
```

Internet Connections

Clients and servers communicate by sending streams of bytes over *connections*:

- **Point-to-point, full-duplex (2-way communication), and reliable.**

***A socket* is an endpoint of a connection**

- **Socket address is an `IPaddress:port` pair**

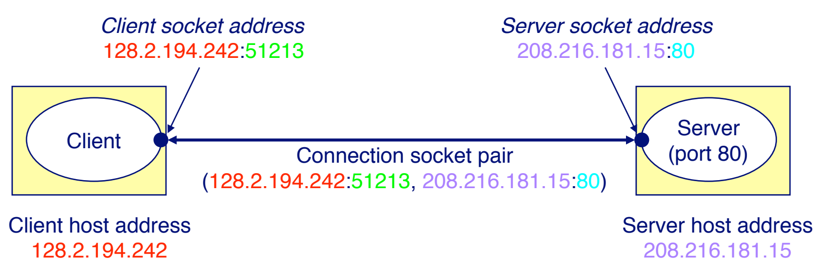
A *port* is a 16-bit integer that identifies a process:

- ***Ephemeral port*: Assigned automatically on client when client makes a connection request**
- **Well-known port: Associated with some service provided by a server (e.g., port 80 is associated with Web servers)**

A connection is uniquely identified by the socket addresses of its endpoints (*socket pair*)

- `(cliaddr:cliport, servaddr:servport)`

Putting it all Together: Anatomy of an Internet Connection



Clients

Examples of client programs

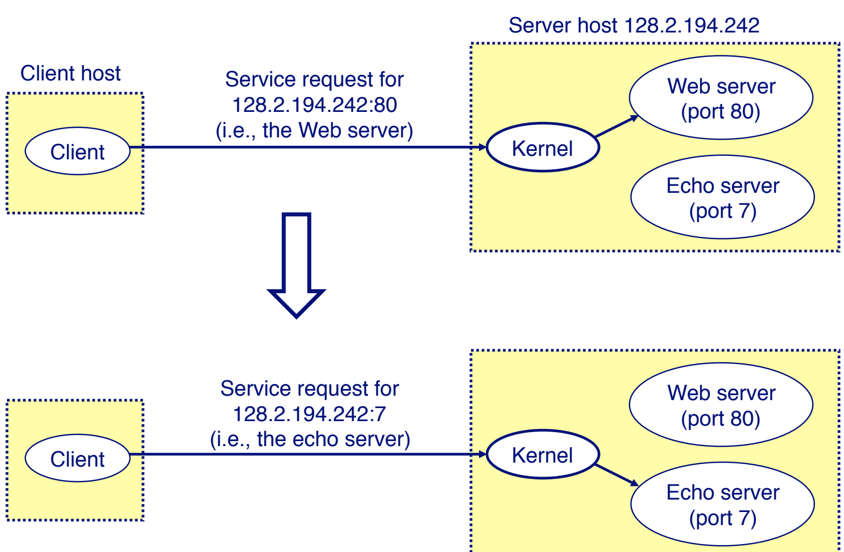
- Web browsers, `ftp`, `telnet`, `ssh`

How does a client find the server?

- The IP address in the server socket address identifies the host *(more precisely, an adapter on the host)*
- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.
- Examples of well known ports
  - Port 7: Echo server
  - Port 23: Telnet server
  - Port 25: Mail server
  - Port 80: Web server

15

Using Ports to Identify Services



16

Servers

Servers are long-running processes (daemons).

- Created at boot-time (typically) by the init process (process 1)
- Run continuously until the machine is turned off.

Each server waits for requests to arrive on a well-known port associated with a particular service.

- Port 7: echo server
- Port 23: telnet server
- Port 25: mail server
- Port 80: HTTP server

A machine that runs a server process is also often referred to as a "server."

17

Server Examples

Web server (port 80)

- Resource: files/compute cycles (CGI programs)
- Service: retrieves files and runs CGI programs on behalf of the client

FTP server (20, 21)

- Resource: files
- Service: stores and retrieve files

> See `/etc/services` for a comprehensive list of the services available on a Linux machine.

Telnet server (23)

- Resource: terminal
- Service: proxies a terminal on the server machine

Mail server (25)

- Resource: email "spool" file
- Service: stores mail messages in spool file

18

Sockets Interface

Created in the early 80's as part of the original Berkeley distribution of Unix that contained an early version of the Internet protocols.

Provides a user-level interface to the network.

Underlying basis for all Internet applications.

Based on client/server programming model.

19

Sockets

What is a socket?

- To the kernel, a socket is an endpoint of communication.
- To an application, a socket is a file descriptor that lets the application read/write from/to the network.
  - All Unix I/O devices, including networks, are modeled as files.

Clients and servers communicate with each other by reading from and writing to socket descriptors.

The main distinction between regular file I/O and socket I/O is how the application "opens" the socket descriptors.

20

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API

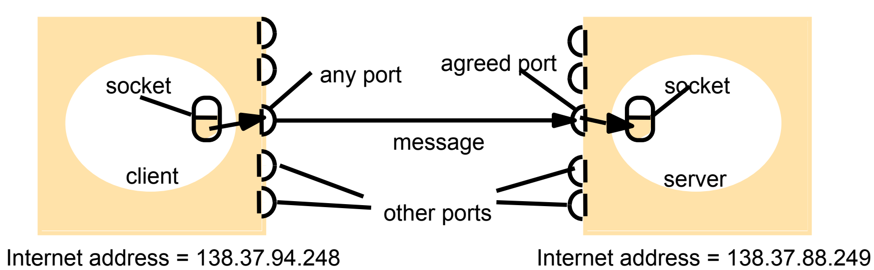
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

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

21

Sockets and ports



22

Berkeley Sockets (1)

Socket primitives for TCP/IP.

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

23

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

application viewpoint

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

24

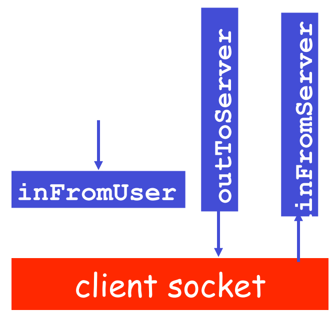
Socket programming with TCP

Example client-server app:

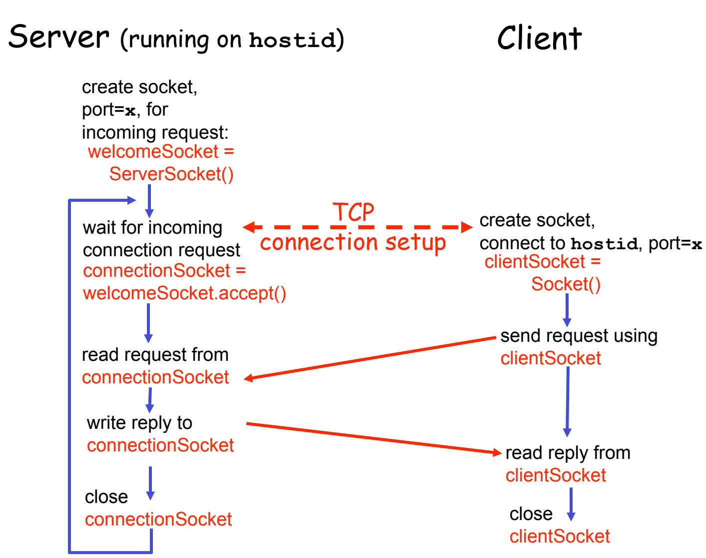
- client reads line from standard input (**inFromUser** stream) , sends to server via socket (**outToServer** stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (**inFromServer** stream)

Input stream: sequence of bytes into process

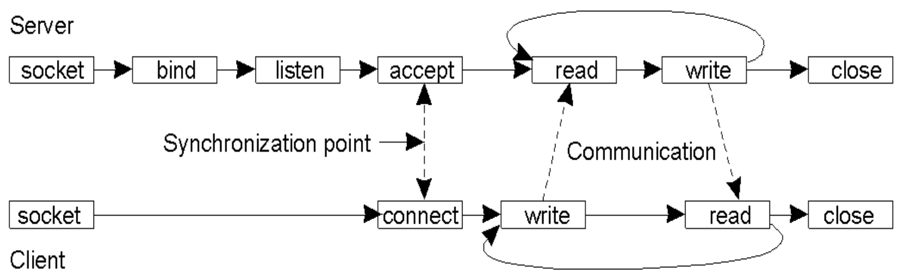
Output stream: sequence of bytes out of process



Client/server socket interaction: TCP



Berkeley Sockets (2)



Connection-oriented communication pattern using sockets.

27

Sockets used for streams

Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM,0)
•
bind(s, ServerAddress);
listen(s,5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

ServerAddress and *ClientAddress* are socket addresses

28

Example: Java client (TCP)

Create input stream → `BufferedReader inFromUser = ...`
Create client socket, connect to server → `Socket clientSocket = ...`
Create output stream attached to socket → `DataOutputStream outToServer = ...`

```
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());
```

29

Example: Java client (TCP), cont.

Create input stream attached to socket → `BufferedReader inFromServer = ...`
Send line to server → `outToServer.writeBytes(...)`
Read line from server → `modifiedSentence = inFromServer.readLine();`

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

    sentence = inFromUser.readLine();

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

    modifiedSentence = inFromServer.readLine();

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

    clientSocket.close();

  }
}
```

30

Example: Java server (TCP)

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

```
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()));
```

31

Example: Java server (TCP), cont

Create output stream, attached to socket

Read in line from socket

Write out line to socket

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

clientSentence = inFromClient.readLine();

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

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

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

32

Socket programming with UDP

UDP: no "connection" between client and server

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

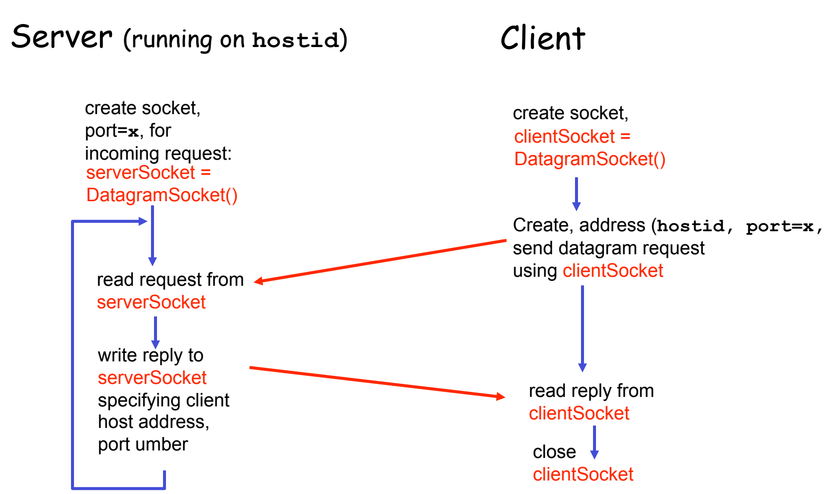
UDP: transmitted data may be received out of order, or lost

application viewpoint

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

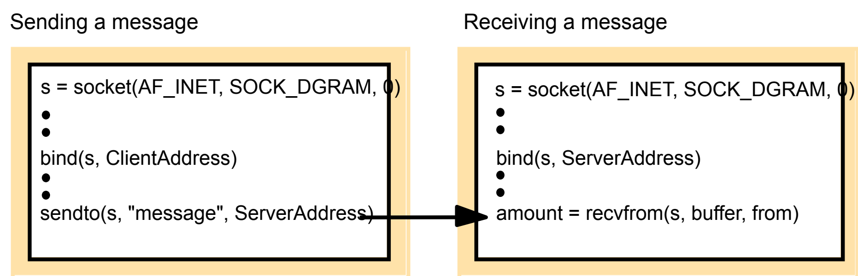
33

Client/server socket interaction: UDP



34

Sockets used for datagrams



ServerAddress and *ClientAddress* are socket addresses

35

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 → BufferedReader inFromUser

Create client socket → DatagramSocket clientSocket

Translate hostname to IP address using DNS → InetAddress IPAddress

36

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();
}
}
```

37

Example: Java server (UDP)

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

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

Create datagram socket at port 9876

```
     DatagramSocket serverSocket = new DatagramSocket(9876);

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

     while(true)
       {
```

Create space for received datagram

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

Receive datagram

```
         serverSocket.receive(receivePacket);
```

38

Example: Java server (UDP), cont

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

Get IP addr port #, of sender

```
InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

        String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();
```

Create datagram to send to client

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

Write out datagram to socket

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

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

39

Next Class

- Using sockets in C programs
  - Follow approach described in Bryant & O'Halloran

40