CS 348: Computer Networks

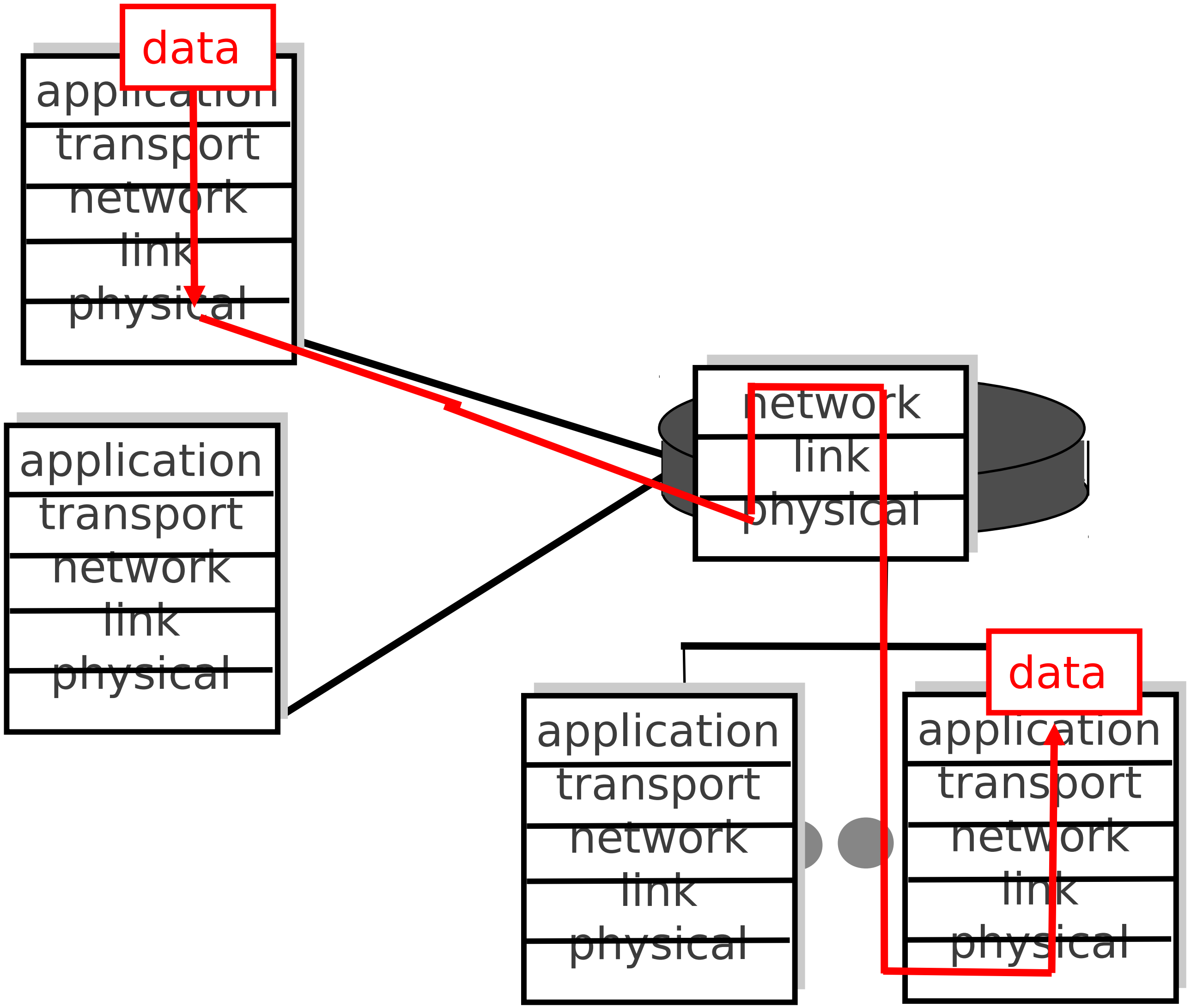
- Sockets; 8th – 11th Oct 2012

Instructor: Sridhar Iyer
IIT Bombay

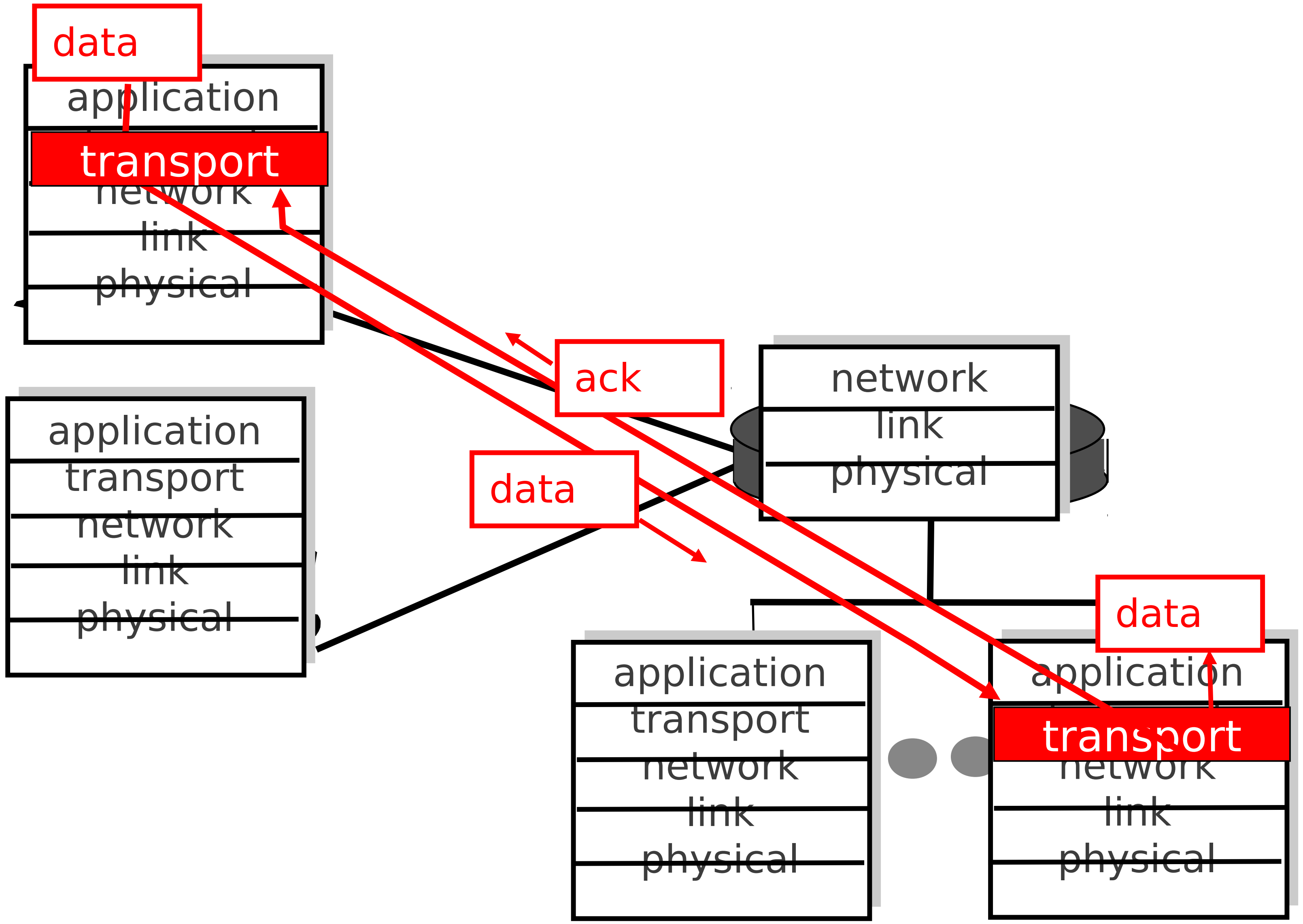
TCP/IP layers

- Physical Layer:
  - deals with interfaces to the physical transmission medium
- Data Link Layer:
  - deals with framing, error detection/correction and multiple access
- Network Layer:
  - deals with addressing, routing and congestion control
- Transport Layer:
  - deals with retransmissions, sequencing and congestion control
- Application Layer:
  - providing services to application developers

Layering: physical communication



Layering: logical communication



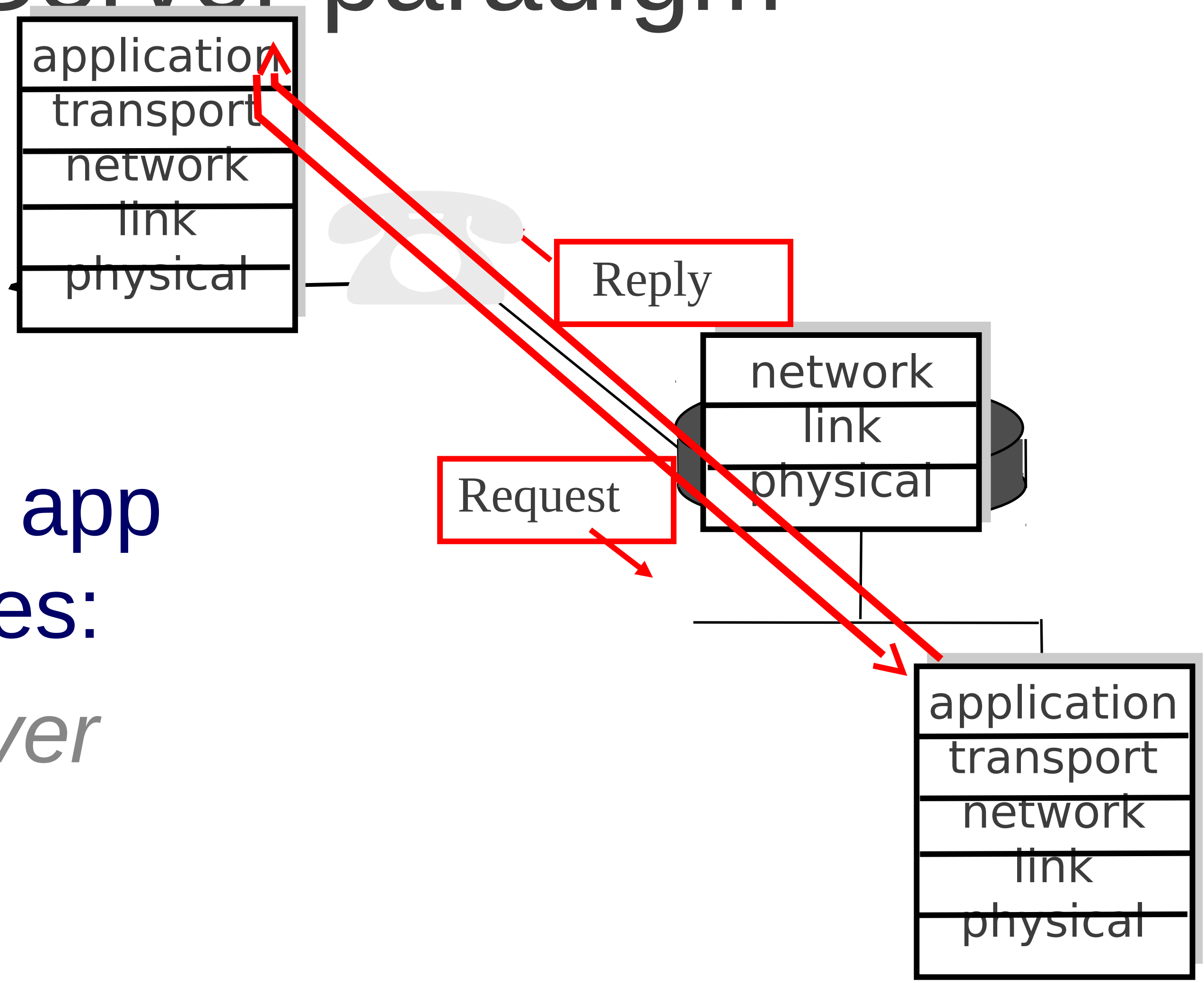
Application layer

- Application:
  - communicating, distributed processes
  - running in network hosts in “user space”
  - exchange messages to implement application
  - e.g., email, file transfer, the Web

Application layer protocols

- one “piece” of an application
- define messages exchanged by application components and actions taken
- uses services provided by lower layer protocols

Client-Server paradigm

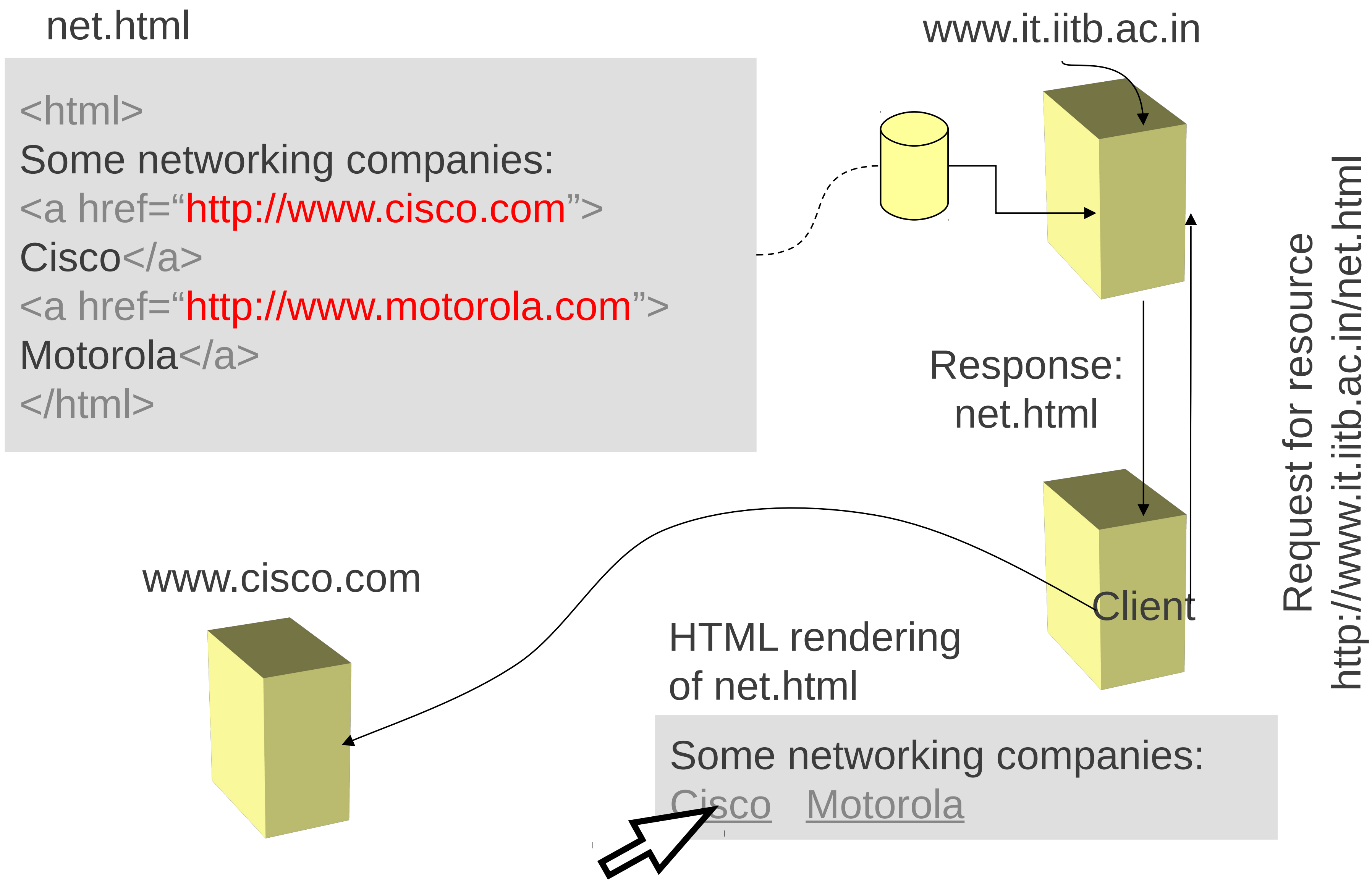


Typical network app has two pieces: *client* and *server*

Actions

- Client
  - initiates contact with server (“speaks first”)
  - typically requests service from server
  - e.g.: sends request for Web page
- Server
  - provides requested service to client
  - e.g., sends requested Web page

Example: Web access (HTTP)



Transport service requirements

- Data loss
  - some apps (e.g., audio) can tolerate some loss; others (e.g., ftp) require 100% reliability
- Timing
  - some apps (e.g., interactive games) require low delay to be "effective"
- Bandwidth
  - some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"

Transport service requirements

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	loss-tolerant	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kb-1Mb video:10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps up	yes, 100's msec
financial apps	no loss	elastic	yes and no

Some application protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	smtp [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
Web	http [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	RTP [RFC 3550]	TCP or UDP
remote file server	NFS	TCP or UDP
Internet telephony	VoIP [RFC 3261]	typically UDP

Traditional distributed applications

- Application logic
- Transport interface code:
  - Makes the appropriate network calls to send and receive the messages
  - Usually divided into transport-independent and transport-dependent parts
- Middleware provides transparency of the transport interface code

Middleware

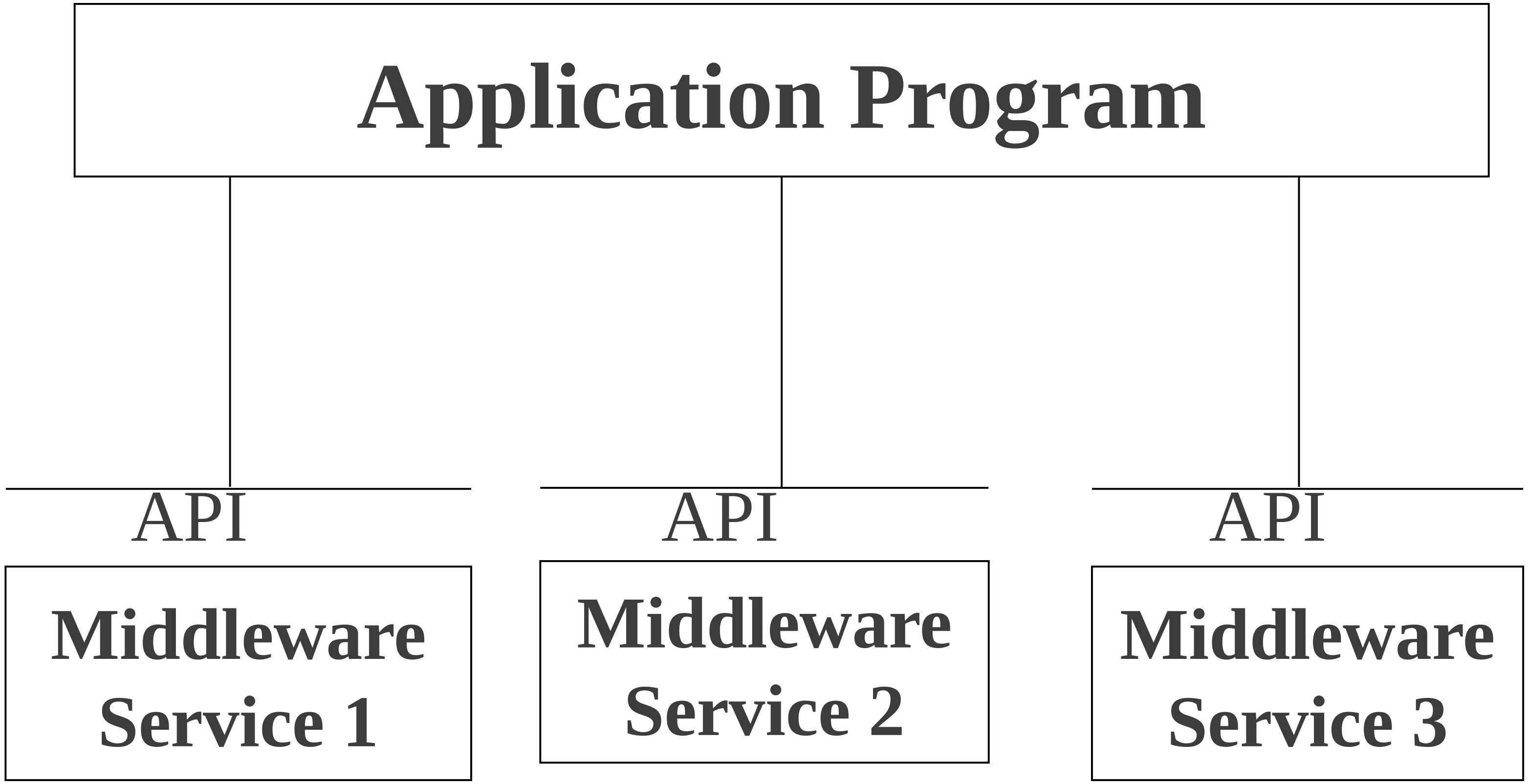
- Software between application programs and OS/network

- Provides a set of higher-level distributed computing capabilities and a set of standards-based interfaces

Middleware

- Interfaces allow applications to be distributed and to take advantage of other services provided over the network
- Middleware is a set of services that are accessible to application programmers through an API
- Example: Sockets, RPC, CORBA

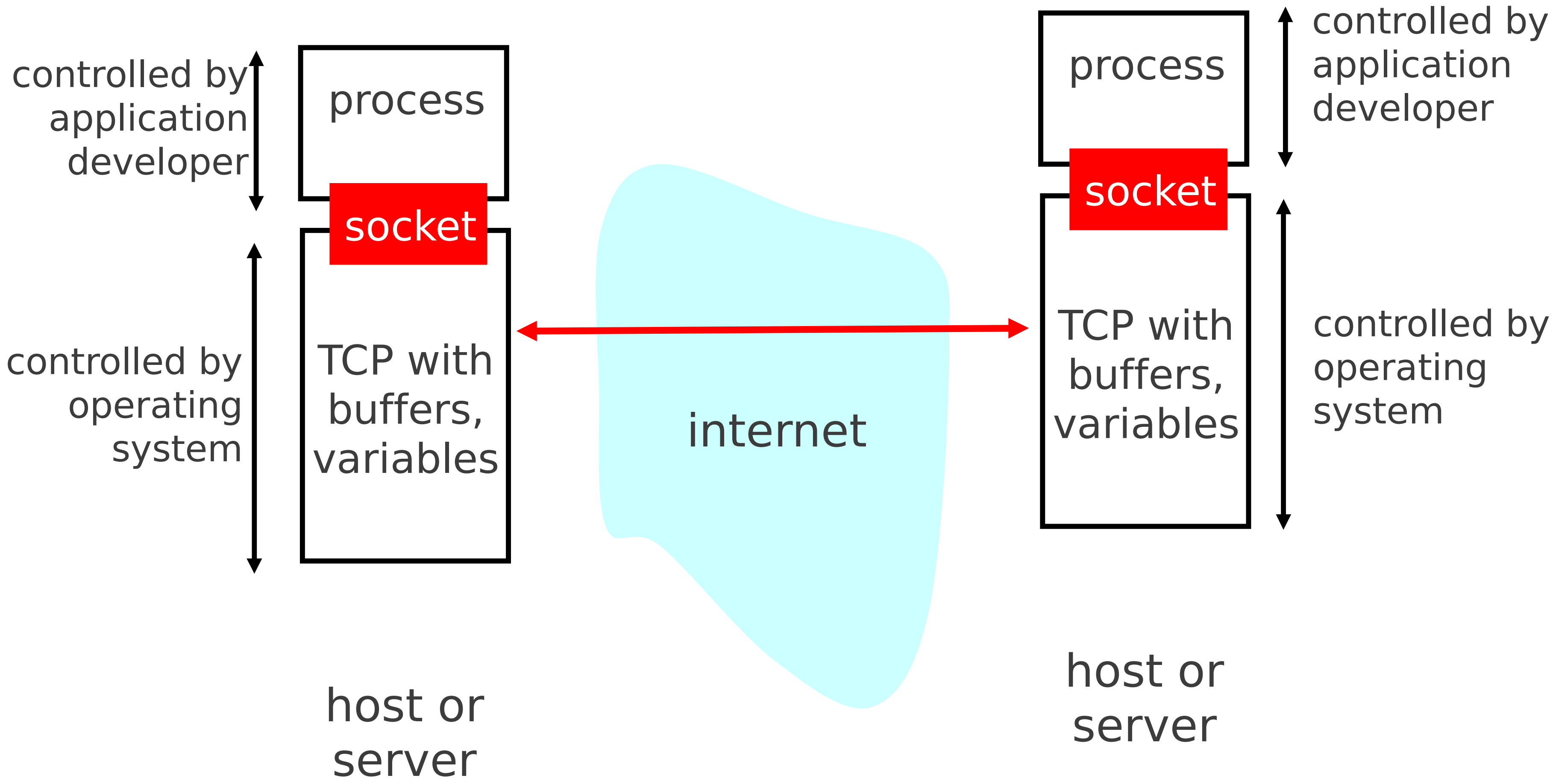
Middleware & API



Sockets API

- Interface between application and transport layer
  - two processes communicate by
  - sending data into a socket
  - reading data out of a socket
- Client “identifies” Server process using <IP address ; port number>

Sockets interface



Socket

- host-local, application-owned, OS-controlled, communication interface
- two processes communicate by sending data into socket, reading data out of socket
- door between application process and transport protocol

Socket types

- Socket identification:
  - “IP address” of client and server hosts
  - “port number” of client and server applications
- Socket types:
  - reliable, byte stream-oriented (TCP)
  - Unreliable, connection-less datagram (UDP)

Client actions

- Create a socket (*socket*)
- Map server name to IP address (*gethostbyname*)
- Connect to a given port on the server address (*connect*)

- Client must contact server first!

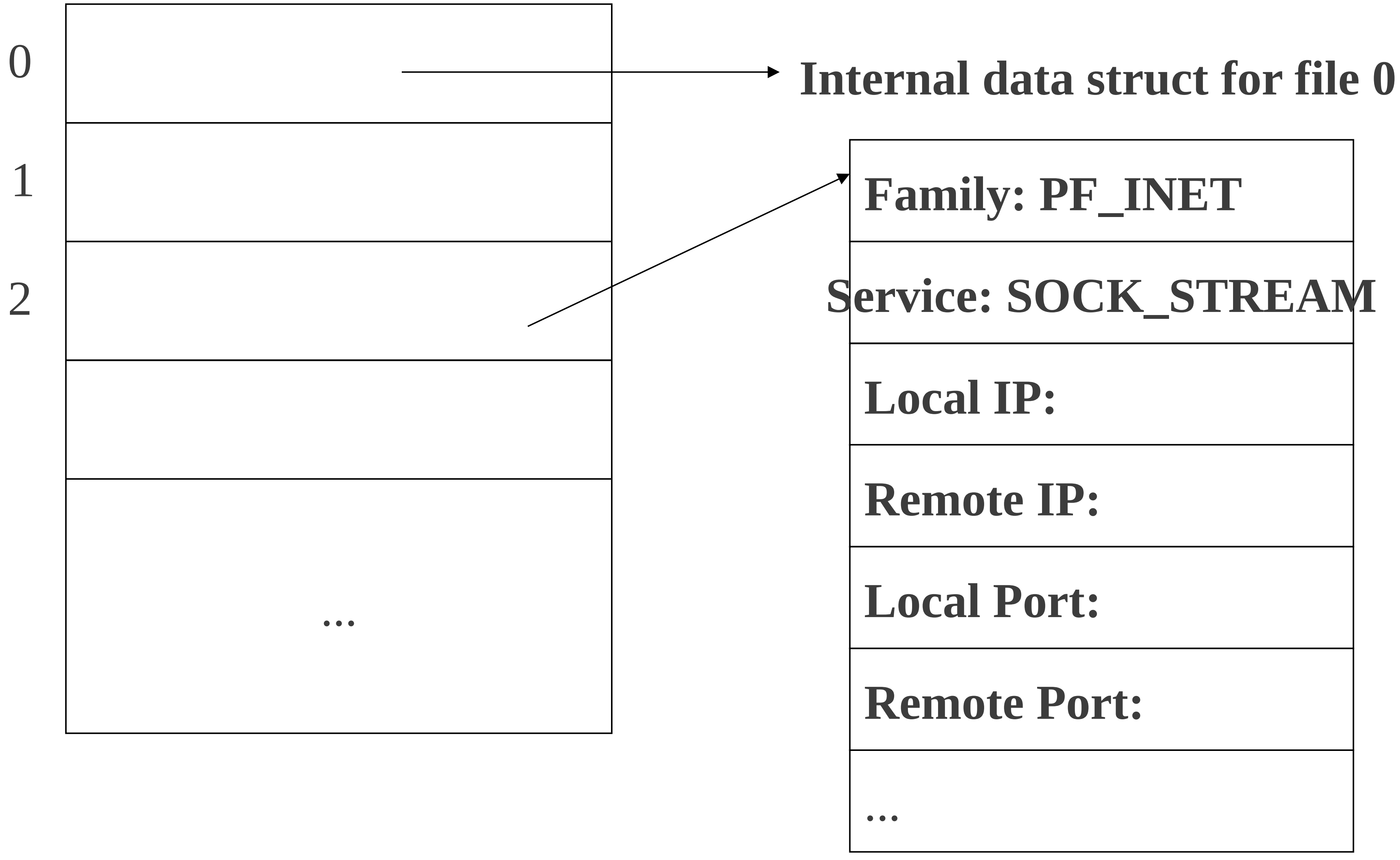
Server actions

- Create a socket (*socket*)
- Bind to one or more port numbers (*bind*)
- Listen on the socket (*listen*)
- Accept client connections (*accept*)
- Server process must be running!

Sockets API: History

- introduced in BSD4.1 UNIX, 1981
- extended the conventional UNIX I/O facilities to use file descriptors for network communication
- extended the *read* and *write* system calls so they work with the new network descriptors.

Sockets data structure



Socket functions

- When a socket is created it does not contain information about how it will be used
  - A *passive* socket is used by a server to wait for an incoming connection
  - An *active* socket is used by a client to initiate a connection

Sockets example

- 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 from socket (**inFromServer** stream)
- Client prints line to standard output (**outToUser** stream)

Socket function calls

- socket: create a descriptor for use in network communication
  - socket (family, type, protocol)
  - returns integer descriptor for socket or –1
- close: terminate communication and de-allocate a socket descriptor
  - close (s)
  - returns 0 or -1

Socket function calls

- connect: connect to a remote peer
  - connect (s, address, len)
  - used to specify the remote end point address
  - used by client primarily
  - used with TCP and UDP
  - returns 0 or –1

Socket function calls

- gethostbyname (name)
  - translates host name to an IP address
  - returns pointer to *a “hostent”* structure, or 0 (error)
- getprotobyname (name)
  - translates protocol’s name to its official integer value
  - returns pointer to a “protoent” structure, or 0 (error)
- getservbyname (name, protocol)
  - used to map a service name to a protocol port number
  - returns pointer to a “servent” structure, or 0 (error)

Socket function calls

- bind: bind a local IP address and protocol number to a socket
  - bind (s, address, len)
  - used by servers primarily
  - returns 0 (success), or -1 (error)

Socket function calls

- listen: place the socket in passive mode
  - listen (s, Qlen)
  - puts the socket in a receiving mode to accept incoming requests
  - sets a limit on the queue size for incoming TCP connection requests
  - returns 0 or –1

Socket function calls

- accept: accept the next incoming connection
    - accept (s, address, len)
    - used only by servers
    - returns socket descriptor of the new socket
    - used only with TCP

Socket function calls

- write: send outgoing data across a connection
  - write (s, buffer, len)
  - send (s, msg, len, flags)

  – returns the number of bytes sent, or -1
  - sendto (s, msg, len, flags, to, tolen)

  – send a message using the destination structure "*to*"

Socket function calls

- read: acquire incoming data
  - read (s, buffer, len)
  - recv (s, buffer, len, flags)
    - » returns the number of bytes, or –1 (error)
  - recvfrom (s, buffer, len, flags, from, fromlen)
    - » gets the next message that arrives at a socket and records the sender's address

Socket function calls

- select (numfds, refds, wrfds, exfds, time)
  - provides asynchronous I/O by permitting a single process to wait for the first of a set of file descriptors to become ready
  - caller can also specify a maximum timeout for the wait
  - returns the number of ready file descriptors, 0 if time limit reached, or -1

Socket parameter description

- **s, from to**: socket descriptor
- **address**: pointer to the struct sockaddr
- **len, fromlen, tolen**: size of sockaddr
- **name**: character string
- **protocol**: char string
- **Qlen**: integer
- **buffer**: character array
- **flags**: integer, control bits
- **numfds**: number of file descriptors
- **refds**: address of fds for input
- **wrfds**: address of fds for output
- **exfds**: address of fds for exceptions

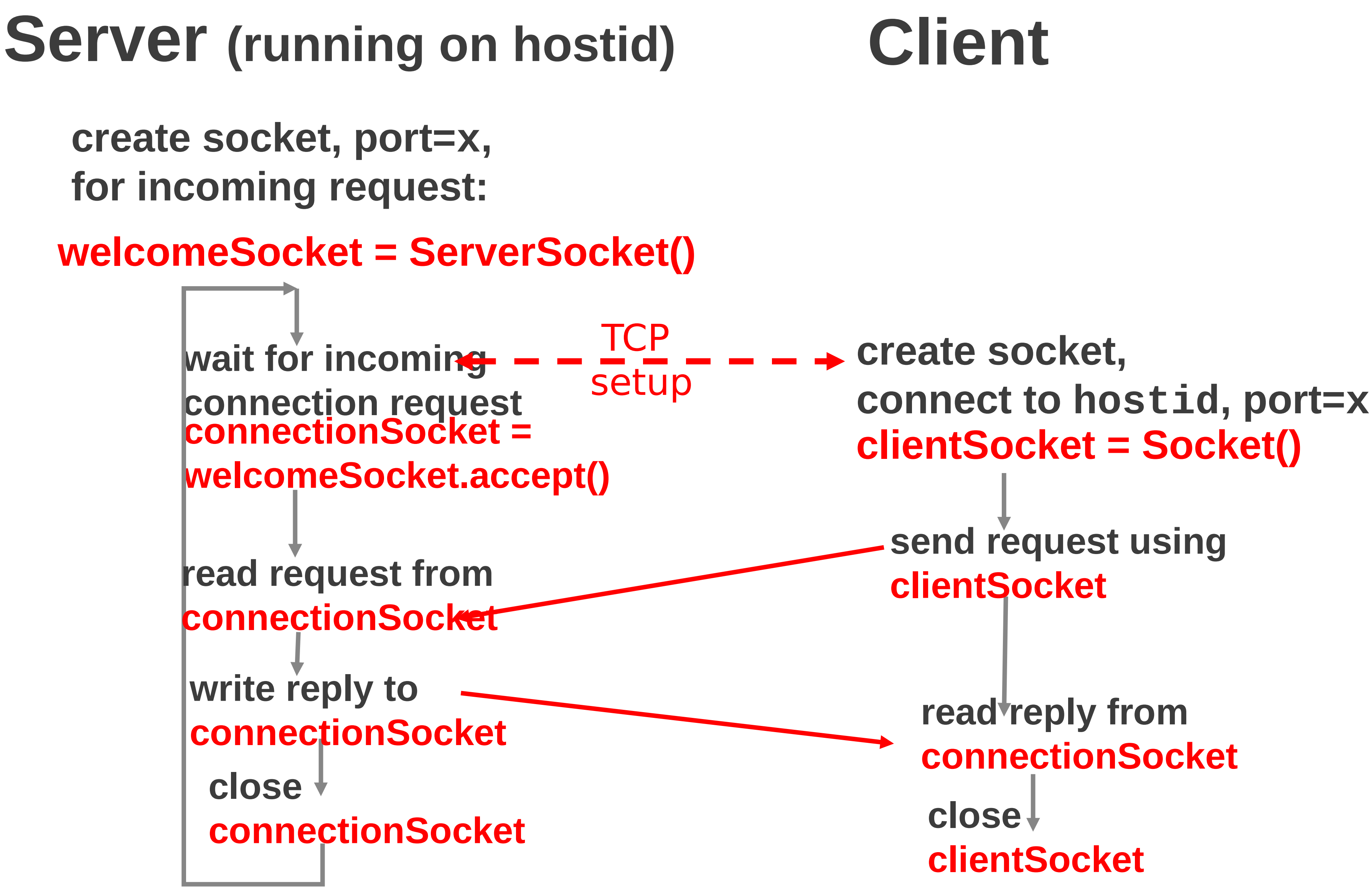
Socket programming with TCP

- Client:
  - must contact server first
  - creates client-local TCP socket
  - specifying IP address, port number of server process
  - client TCP establishes connection to server TCP

Socket programming with TCP

- Server:
  - server process must first be running
  - server must have created socket to accept client's contact
  - When contacted by client, server TCP creates new socket for server process to communicate with client
    - allows server to talk with multiple clients

TCP socket interaction



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());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Java client (TCP), contd.

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);

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

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Java server (TCP), contd.

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

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

UDP socket interaction

Server (running on **hostid**)

create socket,port=x,
For incoming request:
serverSocket = DatagramSocket()

read request from serverSocket

write reply to serverSocket specifying client host address, port umber

Client

create socket,
clientSocket = DatagramSocket()

Create, address (`hostid, port=x,`
send datagram request using clientSocket

read reply from clientSocket

close clientSocket

TCP sockets

- Reliable, connection-oriented sockets are useful when
  - Remote procedures are not idempotent
  - Reliability is a must
  - Messages exceed UDP packet size
  - Server is stateful

UDP sockets

- Unreliable, connectionless sockets are useful when
  - Remote procedures are idempotent
  - Reliability is not very important
  - Server and client messsages fit completely within a packet
  - Server is stateless

Client architecture

- Simpler than servers
  - Typically do not interact with multiple servers concurrently
  - Typically do not require special ports
- Most client software executes as a conventional program
- Clients, unlike servers, do not require special privileged ports
- Most clients rely on OS for security

Server architecture

- Can be quite complex
- Depends on requirements for
  - Type of connection
  - Server state
  - Servicing of requests

Type of connection

- Connection-Oriented:
  - reliable but needs OS resources
- Connection-less:
  - needs less resources but application has to handle loss of messages

Server state

- Stateless:
  - each transaction is independent, crash transparent
- Stateful:
  - server maintains state, faster but expensive for server

Servicing of requests

- Iterative:
  - accept requests one at a time
- Concurrent:
  - fork a new process for each client
  - can service multiple clients
  - needs more resources

Super server process: inetd

- Common services have dedicated port numbers
- inetd binds to all ports required
- Selects and accepts incoming client calls
- Forks program that provides port-specific service and continues

inetd (Internet daemon)

Lines from /etc/services.conf

Client	Server	Port
Mail	smtpd	25
Telnet	telnetd	23
FTP	ftpd	20, 21
Browser	httpd	80
SNMP	snmpd	161
NFS	nfsd	2049

Lines from /etc/inetd.conf

```
 stream      tcp   nowait      root  /usr/sbin/tcpd in.ftpd -l -a
et      stream      tcp   nowait      root  /usr/sbin/tcpd in.telne
```

Sockets Summary