CS 455/555
Intro to Networks and Communications

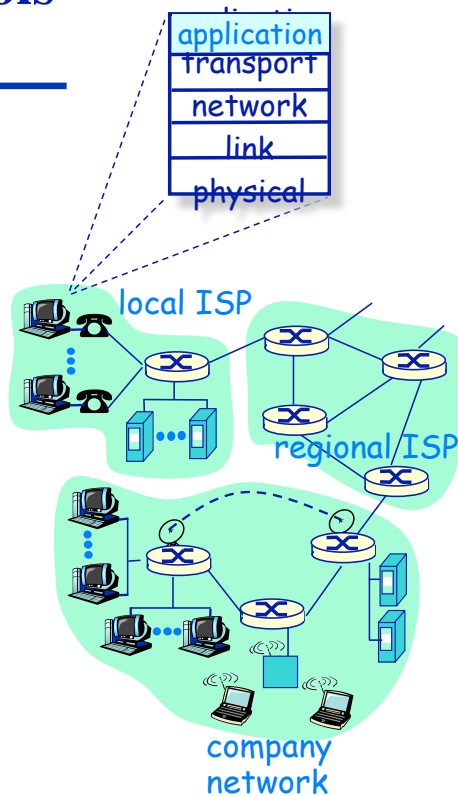# Client/Server Computing and Socket Programming

Michele Weigle

Department of Computer Science
Old Dominion University
mweigle@cs.odu.edu

http://www.cs.odu.edu/~mweigle/courses/cs455-f06/
http://www.cs.odu.edu/~mweigle/courses/cs555-f06/

1

Application-Layer Protocols
Overview

◆ Application-layer protocols define:
   » The types of messages exchanged
   » The syntax and _____ 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
transport
network
link
physical

local ISP

regional ISP

company network

2

```
           Hypertext Transfer Protocol -- HTTP/1.1
```

Abstract

   The Hypertext Transfer Protocol (HTTP) is an application-level
   protocol for distributed, collaborative, hypermedia information
   systems. It is a generic, stateless, protocol which can be used for
   many tasks beyond its use for hypertext, such as name servers and
   distributed object management systems, through extension of its
   request methods, error codes and headers [47]. A feature of HTTP is
   the typing and negotiation of data representation, allowing systems
   to be built independently of the data being transferred.

   HTTP has been in use by the World-Wide Web global information
   initiative since 1990. This specification defines the protocol
   referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

3

Application-Layer Protocols
Outline

- ◆ The architecture of distributed systems
  - » Client/Server computing
  - » P2P computing
  - » Hybrid (Client/Server and P2P) systems
- ◆ 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)
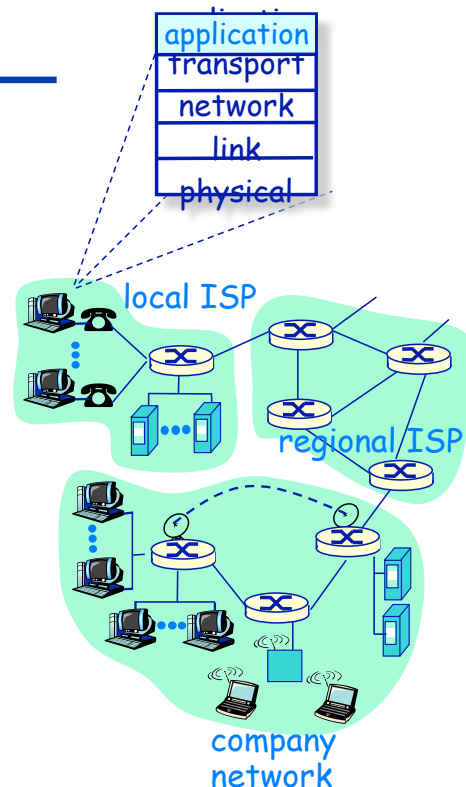
4

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 *v*. out-of-band control signaling
  » Push *v*. pull protocols
  » Persistent *v*. non-persistent connections

◆ Client/server service architectures
  » Contacted server responds versus forwards request
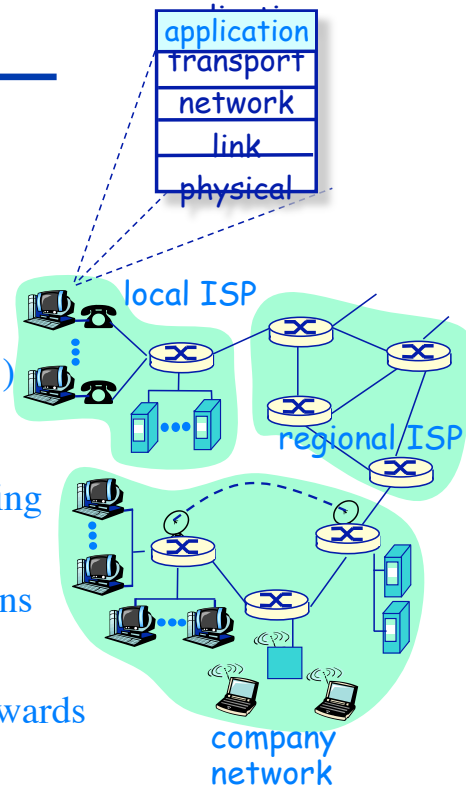
5

Application-Layer Protocols
Client-Server Architecture
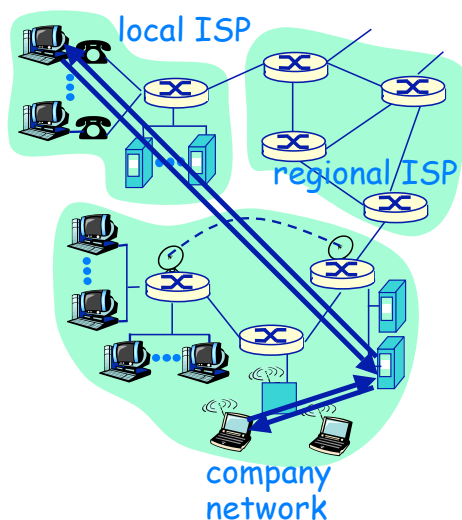
Server:
  » always-on host
  » permanent IP address
  » server farms for scaling

Clients:
  » communicate with server
  » may be intermittently connected
  » may have _____ IP addresses
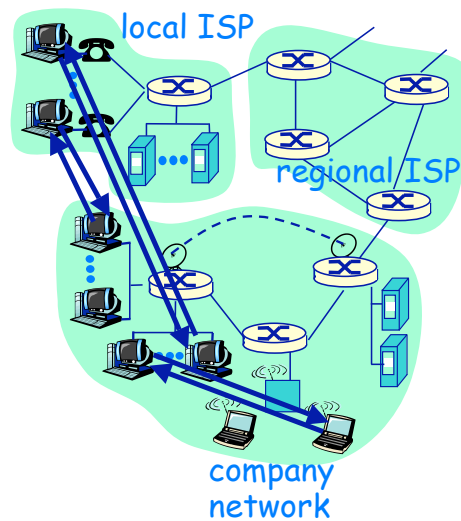 » do not communicate directly with each other

6

Application-Layer Protocols
Pure P2P Architecture

- ◆ No always-on server
- ◆ Arbitrary end systems directly communicate
- ◆ Peers are intermittently connected and change IP addresses
- ◆ Example: Gnutella

Highly scalable

But difficult to manage

local ISP

regional ISP

company network

Application-Layer Protocols
Hybrid of Client-Server and P2P

Napster
- » File transfer P2P
- » File search centralized:
  - ❖ Peers register content at central server
  - ❖ Peers query same central server to locate content

Instant messaging
- » Chatting between two users is P2P
- » Presence detection/location centralized:
  - ❖ User registers its IP address with central server when it comes online
  - ❖ User contacts central server to find IP addresses of buddies

Application-Layer Protocols
Transport Services

Data loss
- Some apps (e.g., audio) can tolerate some loss
- Other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Timing
- Some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

Bandwidth
- Some apps (e.g., multimedia) require minimum amount of bandwidth to be "effective"
- Other apps ("elastic apps") make use of whatever bandwidth they get

Internet Applications
Transport Service Requirements

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	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
instant messaging	no loss	elastic	yes and no

Internet Transport Protocols
Services Provided

◆ 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?

Internet Applications
Application and Transport Protocols

Application	Application layer protocol	Underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	proprietary (e.g. RealNetworks)	TCP or UDP
Internet telephony	proprietary (e.g., Dialpad)	typically UDP

The Application Layer
The client-server paradigm

application
transport
network
link
physical

Client

◆ Typical network application has two pieces: _____ and _____

◆ 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"

local ISP

reply

regional ISP

request

company network

application
transport
network
link
physical

Server

13
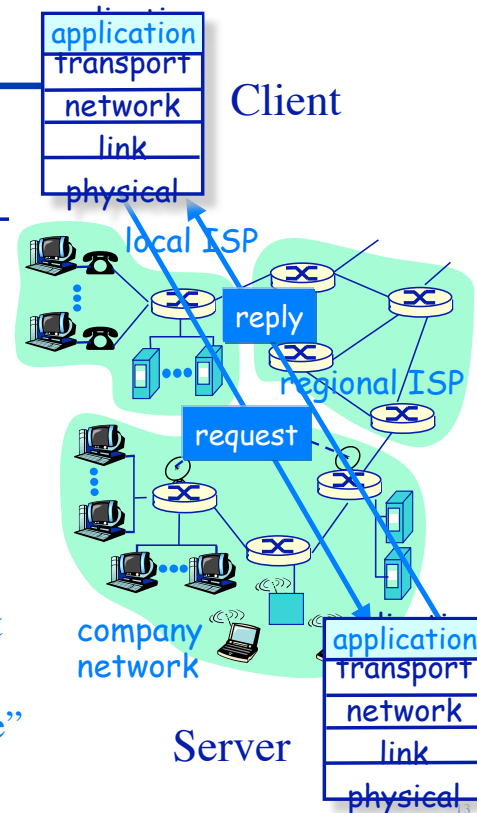
Client/Server Paradigm
Socket programming

◆ _____ are the fundamental building block for client/server systems

◆ Sockets are created and managed by applications

» Strong analogies with files

socket

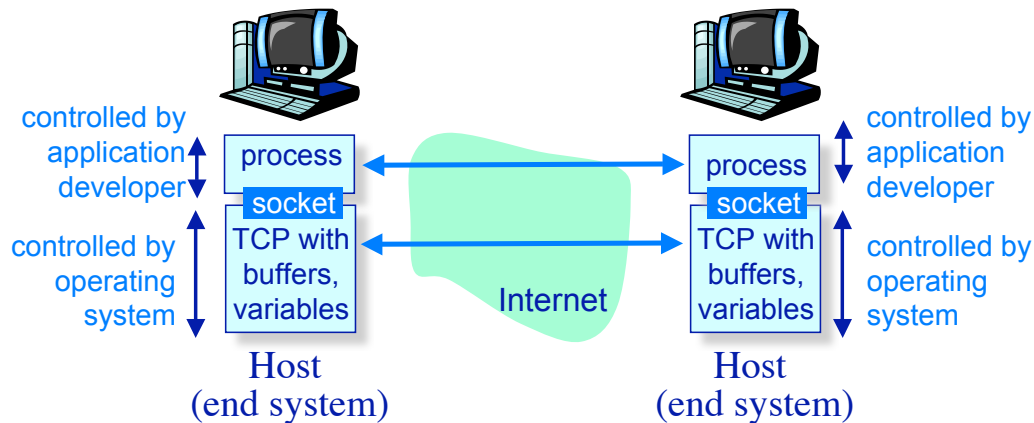a *host-local*, *application created/released*, *OS-controlled* interface into which an application process can *both send and receive* messages to/from another (remote or local) application process

◆ Two types of transport services are available via the socket API:

» _____ sockets: unreliable, datagram-oriented communications

» _____ 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

controlled by
application
developer

process

socket

controlled by
operating
system

TCP with
buffers,
variables

Internet

process

socket

TCP with
buffers,
variables

controlled by
application
developer

controlled by
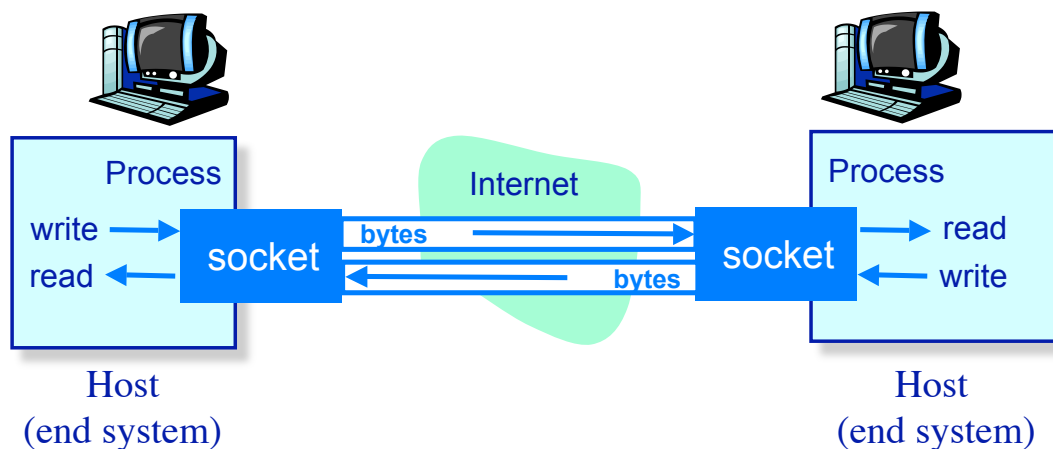operating
system

Host
(end system)

Host
(end system)

Socket Programming using TCP
TCP socket programming model

◆ A TCP socket provides a reliable _____ communications channel from one process to another
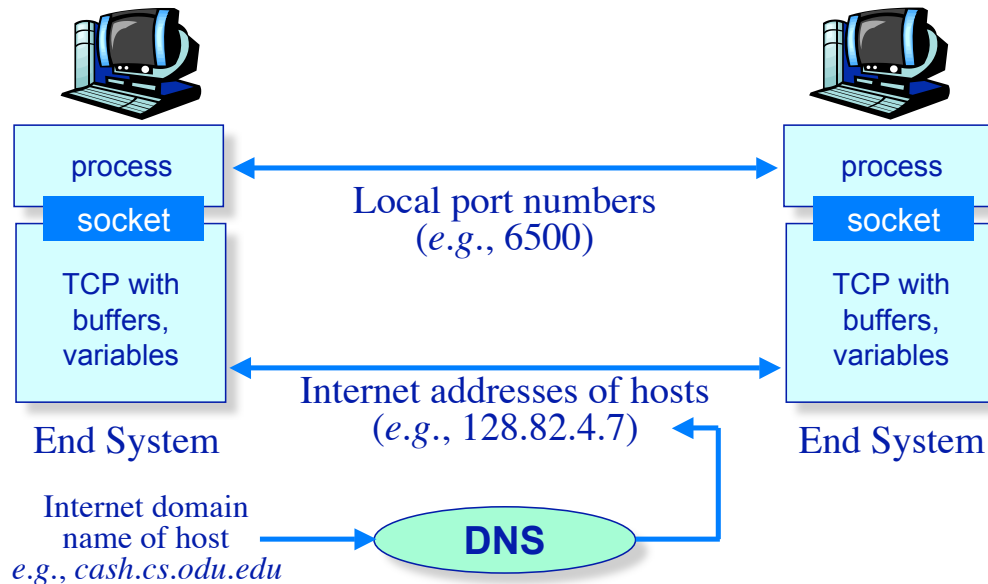» A "pair of pipes" abstraction

Process

write

read

socket

Internet

bytes

bytes

Process

read

write

socket

Host
(end system)

Host
(end system)

Socket Programming using TCP
Network addressing for sockets

◆ Sockets are addressed using an _____ and _____

process

socket

TCP with
buffers,
variables

End System

process

socket

TCP with
buffers,
variables

End System

Local port numbers
(*e.g.*, 6500)

Internet addresses of hosts
(*e.g.*, 128.82.4.7)

Internet domain
name of host
e.g., cash.cs.odu.edu

DNS

Socket Programming using TCP
Socket programming in general

Client

write

read

socket

Internet

bytes

bytes

Server

read

write
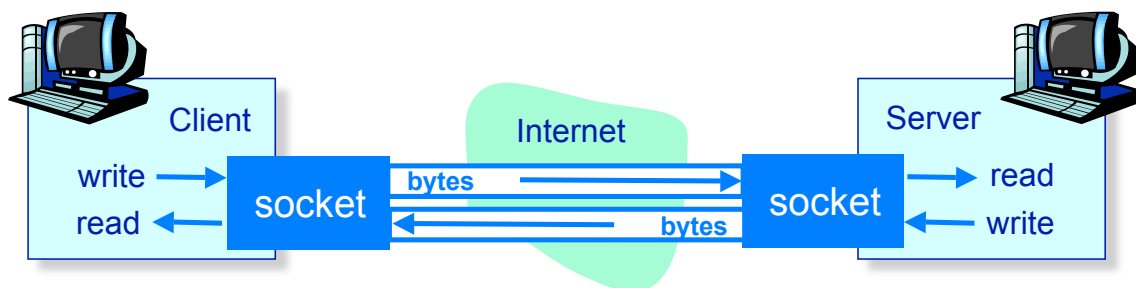
socket

◆ Client creates a local TCP socket specifying the IP and port number of server process
  » if necessary, client resolves IP address from hostname
◆ 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

Socket Programming using TCP
Socket creation in the client-server model

socket

Client

3-way handshake

process

client socket

socket

TCP 3-way handshake

socket "welcoming" socket

bytes

socket

bytes

connection socket

Internet
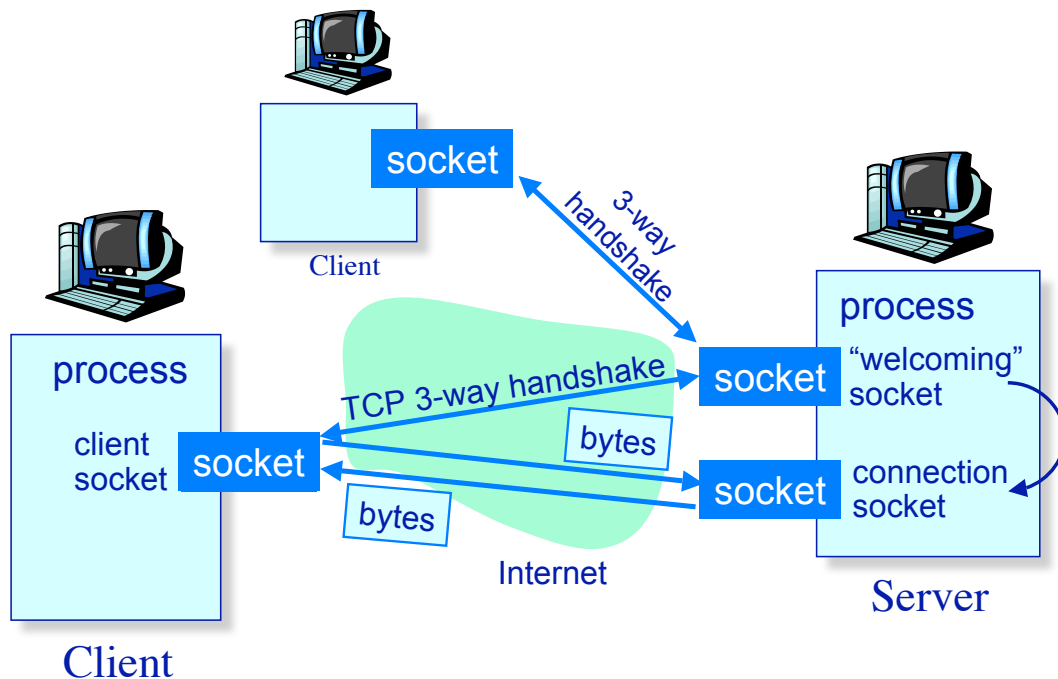
Server

Client

Socket Programming using TCP
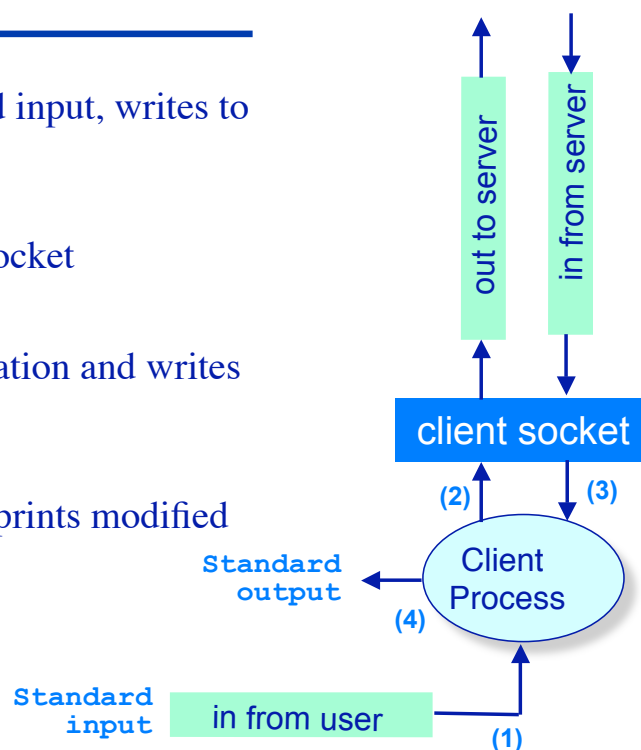## Client structure

◆ Client reads from standard input, writes to server via a socket

◆ Server reads line from a socket

◆ Server does some computation and writes back to client

◆ Client reads from socket, prints modified line to standard output

out to server

in from server

client socket

(2)   (3)

Standard output

Client Process

(4)

Standard input   in from user   (1)

Socket Programming using TCP
Client/server TCP socket interaction

Server (running on *cash.cs.odu.edu*)

create socket for incoming
request (port=**6789**)

Client (running on *tango.cs...*)

wait for incoming
connection request
on welcoming socket

⟵ TCP connection setup ⟶

create socket,
resolve **cash.cs.odu.edu** to **128.82.4.7**
connect to **128.82.42.7**, port=**6789**

read request from the
newly-created socket

write request using the socket

write reply to the client
socket

read reply from
the socket

close the client socket

close the socket

program flow →

data flow →
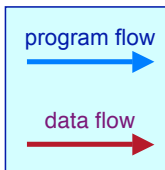
Socket Programming using UDP
UDP socket programming model

◆ A UDP socket provides an _____ bi-directional
communication channel from one process to another
  » A "datagram" abstraction

Process

write →
read ←

socket

Internet

bytes ⟶
bytes

socket

→ read
← write

Process

Host
(end system)

Host
(end system)

Socket Programming using UDP
Client/server UDP socket interaction

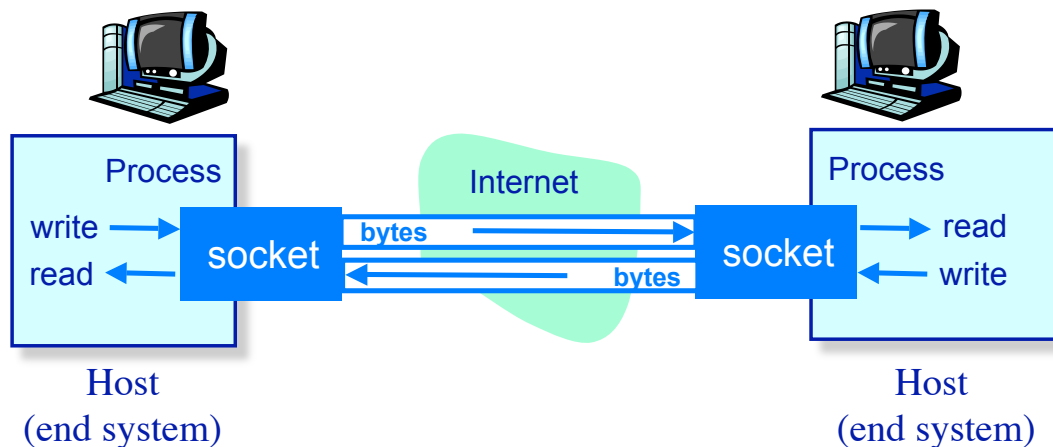Server (running on *cash.cs.odu.edu*)   Client

create socket for incoming
request (port=**9876**)

create socket

create address, resolve
hostname to IP address
(128.82.42.7, port = 9876) and
send datagram using the socket

read request from
the socket

write reply to the
socket specifying
client IP address and
port number

read reply from the socket

close the socket

program flow

data flow

23
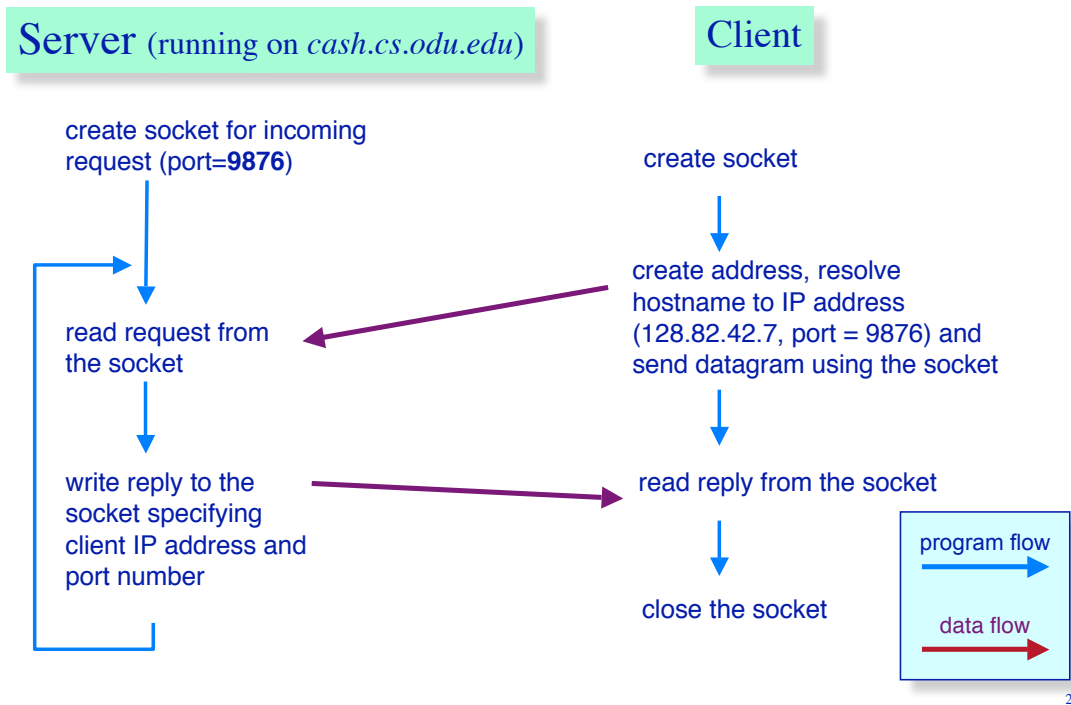
Socket Programming in C
Recommended Reading
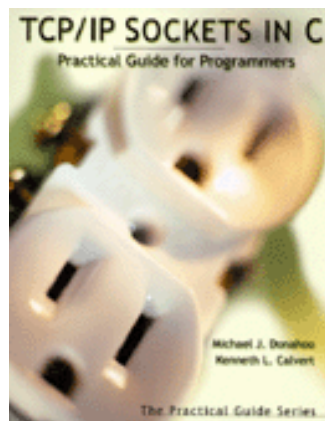
*TCP/IP Sockets in C: Practical
Guide for Programmers*
Donahoo and Calvert
$25.95

24

Socket Programming in C
Include Files

```
#include <stdio.h>      /* printf() and fprintf() */
#include <sys/types.h   /* Socket data types */
#include <sys/socket.h> /* socket(), connect(), send(), recv() */
#include <netinet/in.h> /* IP Socket data types */
#include <arpa/inet.h>  /* sockaddr_in, inet_addr() */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */
```

Socket Programming in C
TCP/IP Sockets

◆ `mySock = socket(family, type, protocol);`

TCP/IP-specific sockets

	Family	Type	Protocol
TCP	PF_INET	SOCK_STREAM	IPPROTO_TCP
UDP		SOCK_DGRAM	IPPROTO_UDP

◆ Socket reference
  » File (socket) descriptor in UNIX

```
struct sockaddr
{
     unsigned short sa_family;    /* Address family (e.g., AF_INET) */
     char sa_data[14];            /* Protocol-specific address information */
};
```

```
struct sockaddr_in
{
     unsigned short sin_family;  /* Internet protocol (AF_INET) */
     unsigned short sin_port;    /* Port (16-bits) */
     struct in_addr sin_addr;    /* Internet address (32-bits) */
     char sin_zero[8];           /* Not used */
};
struct in_addr
{
      unsigned long s_addr;      /* Internet address (32-bits) */
};
```

	Family	Blob		
sockaddr				
	2 bytes	2 bytes	4 bytes	8 bytes
sockaddr_in	Family	Port	Internet address	Not used

Socket Programming in C
TCP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
/* Create socket for incoming connections */
if ((servSock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
    perror ("socket() failed");
     exit (-1);
}
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
servAddr.sin_family = AF_INET;                  /* Internet address family */
servAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
servAddr.sin_port = htons(servPort);          /* Local port */

if (bind(servSock, (struct sockaddr *)&servAddr, sizeof(servAddr)) < 0) {
     perror("bind() failed");
      exit(-1);
}
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
/* Mark the socket so it will listen for incoming connections */
  if (listen(servSock, MAXPENDING) < 0) {
     perror ("listen() failed");
     exit (-1);
  }
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clientAddrLen = sizeof(clientAddr);

    if ((clntSock=accept(servSock,(struct sockaddr *) &clientAddr,
                &clientAddrLen)) < 0) {
        perror("accept() failed");
         exit(-1);
}
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

Server is now blocked waiting for connection from a client

Later, a client decides to talk to the server…

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
/* Create a reliable, stream socket using TCP */
 if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
     perror ("socket() failed");
      exit(-1);
 }
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
servAddr.sin_family      = AF_INET;         /* Internet address family */
servAddr.sin_addr.s_addr = inet_addr(servIP);  /* Server IP address */
servAddr.sin_port        = htons(servPort);    /* Server port */

if (connect(sock, (struct sockaddr *) &servAddr, sizeof(servAddr)) < 0) {
        perror("connect() failed");
         exit (-1);
}
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
if ((clientSock=accept(servSock,(struct sockaddr *) &clientAddr,
                    &clientAddrLen)) < 0) {
        perror("accept() failed");
         exit(-1);
}
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C

TCP Client/Server Interaction

```
stringLen = strlen(mytring);   /* Determine input length */

/* Send the string to the server */
if (send(sock, myString, stringLen, 0) != stringLen) {
     perror("send() sent a different number of bytes than expected");
     exit (-1);
}
```

Client	Server
1. Create a TCP socket	1. Create a TCP socket
2. Establish connection	2. Bind socket to a port
3. Communicate	3. Set socket to listen
4. Close the connection	4. Repeatedly:
	a) Accept new connection
	b) Communicate
	c) Close the connection

Socket Programming in C

TCP Client/Server Interaction

```
/* Receive message from client */
if ((recvMsgSize = recv(clientSocket, buffer, RCVBUFSIZE, 0)) < 0) {
     perror("recv() failed");
     exit (-1);
 }
```

Client	Server
1. Create a TCP socket	1. Create a TCP socket
2. Establish connection	2. Bind socket to a port
3. Communicate	3. Set socket to listen
4. Close the connection	4. Repeatedly:
	a) Accept new connection
	b) Communicate
	c) Close the connection

Socket Programming in C
TCP Client/Server Interaction

```
close(sock);                          close(clientSocket);
```

Client

1. Create a TCP socket
2. Establish connection
3. Communicate
4. Close the connection

Server

1. Create a TCP socket
2. Bind socket to a port
3. Set socket to listen
4. Repeatedly:
   a) Accept new connection
   b) Communicate
   c) Close the connection

Socket Programming in C
TCP Tidbits

◆ Client must know the server's address and port

◆ Server only needs to know its own port

◆ No correlation between send() and recv()

Client	Server
send("Hello Bob")	
	recv() -> "Hello "
	recv() -> "Bob"
	send("Hi ")
	send("Jane")
recv() -> "Hi Jane"	

Socket Programming in C
Closing a Connection

- ◆ close() used to delimit communication
- ◆ Analogous to EOF

Echo Client	Echo Server
send(*string*)	recv(*buffer*)
while (not received entire string) 　　　recv(*buffer*) 　　　print(*buffer*)	while(client has not closed connection) 　　　send(*buffer*) 　　　recv(*buffer*)
close(*socket*)	close(*client socket*)

Socket Programming in C
Compiling and Running

```
% g++ TCPEchoClient.c -o TCPEchoClient -lsocket -lnsl
```

`-o executable` - name of the executable (instead of `a.out`)
`-lsocket -lnsl` - includes libraries necessary for socket communication

Running:

```
% ./TCPEchoClient 130.127.48.60 50000
```

Socket Programming in C
UDP Sockets

◆ C programs written using TCP or UDP sockets can look very similar

◆ How UDP sockets differ:

» no connect (*we'll come back to this later*)

» use `sendto()` and `recvfrom()` instead of `send()` and `recv()`

» UDP preserves message boundaries

Generic

```
struct sockaddr
{
        unsigned short sa_family;      /* Address family (e.g., AF_INET) */
        char sa_data[14];              /* Protocol-specific address information */
};
```

IP Specific

```
struct sockaddr_in
{
        unsigned short sin_family;     /* Internet protocol (AF_INET) */
        unsigned short sin_port;       /* Port (16-bits) */
        struct in_addr sin_addr;       /* Internet address (32-bits) */
        char sin_zero[8];              /* Not used */
};
struct in_addr
{
        unsigned long s_addr;          /* Internet address (32-bits) */
};
```

sockaddr	Family	Blob		
	2 bytes	2 bytes	4 bytes	8 bytes
sockaddr_in	Family	Port	Internet address	Not used

Socket Programming in C
UDP Client/Server Interaction

Server starts by getting ready to receive client connections…

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
/* Create socket for data from a client */
 if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
     perror("socket() failed");
     exit (-1);
 }
```

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
servAddr.sin_family = AF_INET;              /* Internet address family */
servAddr.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
servAddr.sin_port = htons(servPort);        /* Local port */

if (bind(sock, (struct sockaddr *)&echoServAddr, sizeof(echoServAddr)) < 0) {
     perror("bind() failed");
     exit(-1);
}
```

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
for (;;) /* Run forever */
{
     clientAddrLen = sizeof(clientAddr);

     /* Receive message from client */
     if ((recvMsgSize = recvfrom(sock, buffer, RCVBUFSIZE, 0,
                 (struct sockaddr*) &clientAddr, &clientAddrLen)) < 0) {
         perror ("recvfrom() failed");
         exit (-1);
     }
```

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

Server is now blocked waiting for data from a client

Later, a client decides to talk to the server…

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
/* Create a datagram socket using UDP */
 if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
     perror("socket() failed");
     exit (-1);
 }
```

Client

1. Create a UDP socket
2. Communicate
3. Close the socket

Server

1. Create a UDP socket
2. Bind socket to a port
3. Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
servAddr.sin_family      = AF_INET;              /* Internet address family */
servAddr.sin_addr.s_addr = inet_addr(servIP);    /* Server IP address */
servAddr.sin_port        = htons(servPort);      /* Server port */

stringLen = strlen(myString);     /* Determine input length */

/* Send the string to the server */
if (sendto(sock, myString, stringLen, 0, (struct sockaddr*)
        &servAddr, sizeof(servAddr)) != stringLen) {
    perror("sendto() sent a different number of bytes than expected");
    exit (-1);
}
```

Client		Server	
1.	Create a UDP socket	1.	Create a UDP socket
2.	Communicate	2.	Bind socket to a port
3.	Close the socket	3.	Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
for (;;) /* Run forever */
{
    clientAddrLen = sizeof(clientAddr);

    /* Receive message from client */
    if ((recvMsgSize = recvfrom(sock, buffer, RCVBUFSIZE, 0,
                (struct sockaddr*) &clientAddr, &clientAddrLen)) < 0) {
        perror("recvfrom() failed");
        exit(-1);
    }
```

Client		Server	
1.	Create a UDP socket	1.	Create a UDP socket
2.	Communicate	2.	Bind socket to a port
3.	Close the socket	3.	Repeatedly communicate

Socket Programming in C
UDP Client/Server Interaction

```
close(sock);
```

Client	**Server**
1. Create a UDP socket	1. Create a UDP socket
2. Communicate	2. Bind socket to a port
3. Close the socket	3. Repeatedly communicate

UDP Sockets
connect and UDP

◆ You can use `connect()` on UDP sockets!

◆ When used at the client, it saves state about the server's IP address and port

◆ From then on, you can use `send()` instead of `sendto()` to send messages to the server.

◆ You will *not* be able to receive messages from any place other than the "connected" server on the socket.

◆ This call does not affect the server at all. Nothing is sent to the server to indicate that the socket is "connected".

Socket Programming in C
Name Resolution

◆ We have to do name resolution manually in C
» use `gethostbyname()` to determine the IP address
» display the hostname, any aliases, and list of addresses

```
struct hostent {
    char *h_name;        /* official name of host */
    char **h_aliases;    /* list of aliases */
    int h_addrtype;      /* type of address (AF_INET) */
    int h_length;        /* address length */
    char **h_addr_list;  /* list of addresses (nbo) */
}
```

```
#include <netdb.h>    /* for gethostbyname() */

struct hostent* gethostbyname (const char* hostName);
```

Name Resolution
Example

```
struct sockaddr_in servAddr;
struct hostent* host;
char *servIP;   // input from user

// parse input parameters, create socket, etc.

if ((host=gethostbyname(servIP)) == NULL) {
    DieWithError ("gethostbyname failed");
}

// construct servAddr structure
memset (&servAddr, 0, sizeof(servAddr));
servAddr.sin_family = AF_INET;
servAddr.sin_addr.s_addr =
        *((unsigned long*) host->h_addr_list[0]);
servAddr.sin_port = htons(port);
```

Message Boundaries
TCP Behavior

◆ TCP does not preserve message boundaries
  » one side sends separate messages back-to-back
  » the receiver could receive both messages with one `recv` call

◆ Options for dealing with this
  » read from TCP socket one byte at a time until a delimiter (such as '`\n`') is reached
  » save state so that parts of a partially-received message don't get lost

Message Boundaries
UDP Behavior

◆ UDP preserves message boundaries

◆ Each call to `recvfrom()` returns data from at most one `sendto()` call

◆ Different calls to `recvfrom()` will not return data from the same `sendto()` call

Message Boundaries
Examples

- ◆ UDPMsgClient.c
    - » sends the user-input message and then sends "EXTRA MSG"
- ◆ TCPMsgClient.c
    - » sends the user-input message and then sends "EXTRA MSG"
- ◆ TCPMsgServer.c
    - » prints out messages received from client
- ◆ TCPShortServer.c
    - » reads only 5 bytes at a time
- ◆ UDPShortServer.c
    - » reads only 5 bytes at a time