CSEE 4119 Computer Networks

Chapter 2
Application (5/5)
Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management

3.6 Principles of congestion control
3.7 TCP congestion control
Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
Transport vs. network layer

- **network layer**: logical communication between hosts
- **transport layer**: logical communication between processes
  - relies on, enhances, network layer services

**Household analogy:**
- 12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management
3.6 Principles of congestion control
3.7 TCP congestion control
Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments
to correct socket

Multiplexing at send host:
gathering data from multiple
sockets, enveloping data with
header (later used for
demultiplexing)

= socket  = process

<table>
<thead>
<tr>
<th>application P3</th>
<th>P1 application</th>
<th>P2 application</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td>transport</td>
<td>transport</td>
</tr>
<tr>
<td>network</td>
<td>network</td>
<td>network</td>
</tr>
<tr>
<td>link</td>
<td>link</td>
<td>link</td>
</tr>
<tr>
<td>physical</td>
<td>physical</td>
<td>physical</td>
</tr>
</tbody>
</table>

host 1  host 2  host 3
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number

- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
</tbody>
</table>
Connectionless demultiplexing

- **recall:** create sockets with host-local port numbers:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```

- **recall:** when creating datagram to send into UDP socket, must specify 
  
  (dest IP address, dest port number)

- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux (cont)
Connection-oriented demux: Threaded Web Server
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   - SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
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, OS-controlled* interface (a “door”) into which application process can both send and receive messages to/from another application process
Two essential types of sockets

- **C: SOCK_STREAM**
  - JAVA: Socket
  - a.k.a. TCP
  - reliable delivery
  - in-order guaranteed
  - connection-oriented
  - bidirectional

- **C: SOCK_DGRAM**
  - JAVA: DatagramSocket
  - a.k.a. UDP
  - unreliable delivery
  - no order guarantees
  - no notion of "connection" - app includes dest. in packets
  - can send or receive

Q: why have type SOCK_DGRAM?
A Socket-eye view of the Internet

- Each host machine has an IP address
- When a packet arrives at a host
The Bare minimum

- To code a socket, you will need at least
  - ACCEPT: *block and wait* for CONNECT PKT
  - CONNECT: *establish* a connection
  - RECEIVE: *block and wait* for a SEND PKT
  - SEND: *actually sending* a PKT on the channel
  - DISCONNECT: *putting an end*

- These are the functions you’ll see
  - C, JAVA, for any connection-oriented transport
A first example

- How does it work
  - Server LISTEN, wait for CONNECT PKT
  - Client send a CONNECT message, and then block until received the answer from server
  - Once server received CONNECT message, it becomes unblocked, send an answer, and becomes blocked again in READ
  - Once the client received the answer, it becomes unblocked, SENDS a request message, and block again in READ
  - The server finally answer with data, and close
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

host or server

TCP with buffers, variables

socket

process

controlled by application developer

controlled by operating system

host or server

TCP with buffers, variables

socket

process

internet
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)

create socket,
port=x, for
incoming request:
welcomeSocket =
ServerSocket()

wait for incoming
connection request
connectionSocket =
welcomeSocket.accept()

read request from
connectionSocket

write reply to
connectionSocket

close
connectionSocket

TCP connection setup

Client

create socket,
connect to hostid, port=x
clientSocket =
Socket()

send request using
clientSocket

read reply from
clientSocket

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)

```java
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());

        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        String sentence = inFromUser.readLine();
        String modifiedSentence = sentence.toLowerCase();

        outToServer.writeBytes(modifiedSentence);
        outToServer.close();
        clientSocket.close();
    }
}
```
Example: Java client (TCP), cont.

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

            String clientSentence = inFromClient.readLine();
            String capitalizedSentence = clientSentence.toUpperCase();
            System.out.println(capitalizedSentence);
        }
    }
}
```
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
```
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   - SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
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)

- create socket, port= x.
  - serverSocket = DatagramSocket()
- read datagram from serverSocket
- write reply to serverSocket specifying client address, port number

Client

- 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 that TCP received “byte stream”)

Client process

client UDP socket
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();
Example: Java client (UDP), cont.

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

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

String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```
Example: Java server (UDP)

```java
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);

            DatagramPacket sendPacket =
                new DatagramPacket(sendData, sendData.length);
            serverSocket.send(sendPacket);
        }
    }
}
```
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);

}
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   ▪ SMTP, POP3, IMAP
2.5 DNS
2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP
+ (Bonus) Same with C
The Bare minimum

- To code a socket, you will need at least
  - ACCEPT: block and wait for CONNECT PKT
  - CONNECT: establish a connection
  - RECEIVE: block and wait for a SEND PKT
  - SEND: actually sending a PKT on the channel
  - DISCONNECT: putting an end

- These are the functions you’ll see
  - C, JAVA, etc.
Socket functions overview (C)

- For TCP with C, the primitives are:
  - **SOCKET**
  - **BIND**
  - **LISTEN**:
    - **ACCEPT**: *block and wait* for CONNECT PKT
    - **CONNECT**: *establish* a connection
    - **RECEIVE**: *block and wait* for a SEND PKT
    - **SEND**: *actually sending* a PKT on the channel
    - **DISCONNECT**: *putting an end*
Socket Creation in C: socket

- int s = socket(domain, type, protocol);
  - s: socket descriptor, an integer
  - domain: integer, communication domain
    - e.g., PF_INET (IPv4 protocol) - typically used
  - type: communication type
    - SOCK_STREAM: reliable, 2-way, connection-based service
    - SOCK_DGRAM: unreliable, connectionless,
      - other values: need root permission, rarely used, or obsolete
  - protocol: specifies protocol - usually set to 0
- NOTE: socket call does not specify where data will be coming from, nor where it will be going to - it just creates the interface!
The bind function

- associates and (can exclusively) reserves a port for use by the socket

```c
int status = bind(sockid, &addrport, size);
```

- **status**: error status, = -1 if bind failed
- **sockid**: integer, socket descriptor
- **addrport**: struct sockaddr, the (IP) address and port of the machine (address usually set to INADDR_ANY - chooses a local address)
- **size**: the size (in bytes) of the addrport structure

- bind can be skipped for both types of sockets. When and why?
**Skipping the bind**

- **SOCK_DGRAM:**
  - if only sending, no need to bind. The OS finds a port each time the socket sends a pkt
  - if receiving, need to bind

- **SOCK_STREAM:**
  - At the client - determined during conn. setup
  - don’t need to know port sending from (during connection setup, receiving end is informed of port)
Connection Setup (SOCK_STREAM)

- Recall: no connection setup for SOCK_DGRAM
- A connection occurs between two kinds of participants
  - passive: waits for an active participant to request connection
  - active: initiates connection request to passive side
- Once connection is established, passive and active participants are “similar”
  - both can send & receive data
  - either can terminate the connection
Connection setup cont’d

- Passive participant
  - step 1: listen (for incoming requests)
  - step 3: accept (a request)
  - step 4: data transfer
- The accepted connection is on a new socket
- The old socket continues to listen for other active participants
- Why?

- Active participant
  - step 2: request & establish connection
  - step 4: data transfer
Connection setup: listen & accept

- Called by passive participant
- int status = listen(sock, queuelen);
  - status: 0 if listening, -1 if error
  - sock: integer, socket descriptor
  - queuelen: integer, # of active participants that can “wait” for a connection
  - listen is non-blocking: returns immediately
- int s = accept(sock, &name, &namelen);
  - s: integer, the new socket (used for data-transfer)
  - sock: integer, the orig. socket (being listened on)
  - name: struct sockaddr, address of the active participant
  - namelen: sizeof(name): value/result parameter
    - must be set appropriately before call
    - adjusted by OS upon return
  - accept is blocking: waits for connection before returning
connect call

- int status = connect(sock, &name, namelen);
  - status: 0 if successful connect, -1 otherwise
  - sock: integer, socket to be used in connection
  - name: struct sockaddr: address of passive participant
  - namelen: integer, sizeof(name)

- connect is **blocking**
Sending / Receiving Data

- With a connection (SOCK_STREAM):
  - `int count = send(sock, &buf, len, flags);`
    - `count`: # bytes transmitted (-1 if error)
    - `buf`: char[], buffer to be transmitted
    - `len`: integer, length of buffer (in bytes) to transmit
    - `flags`: integer, special options, usually just 0
  - `int count = recv(sock, &buf, len, flags);`
    - `count`: # bytes received (-1 if error)
    - `buf`: void[], stores received bytes
    - `len`: # bytes received
    - `flags`: integer, special options, usually just 0
  - Calls are **blocking** [returns only after data is sent (to socket buf) / received]
Sending / Receiving Data (cont’d)

- Without a connection (SOCK_DGRAM):
  - int count = sendto(sock, &buf, len, flags, &addr, addrlen);
    - count, sock, buf, len, flags: same as send
    - addr: struct sockaddr, address of the destination
    - addrlen: sizeof(addr)
  - int count = recvfrom(sock, &buf, len, flags, &addr, &addrlen);
    - count, sock, buf, len, flags: same as recv
    - addr: struct sockaddr, address of the source
    - addrlen: sizeof(addr): value/result parameter

- Calls are **blocking** [returns only after data is sent (to socket buf) / received]
**close**

- When finished using a socket, the socket should be closed:

  ```c
  status = close(s);
  ```

  - `status`: 0 if successful, -1 if error
  - `s`: the file descriptor (socket being closed)

- **Closing a socket**
  - closes a connection (for SOCK_STREAM)
  - frees up the port used by the socket
The **struct sockaddr**

- **The generic:**
  ```c
  struct sockaddr {
    u_short sa_family;
    char sa_data[14];
  };
  ```

  - **sa_family**
    - specifies which address family is being used
    - determines how the remaining 14 bytes are used

- **The Internet-specific:**
  ```c
  struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
  };
  ```

  - **sin_family** = AF_INET
  - **sin_port**: port # (0-65535)
  - **sin_addr**: IP-address
  - **sin_zero**: unused
## TCP - Serial Model

<table>
<thead>
<tr>
<th>Client Side</th>
<th>Server Side</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sd=socket(type)</code></td>
<td><code>sd=socket(type)</code></td>
</tr>
<tr>
<td></td>
<td><code>bind(sd, port)</code></td>
</tr>
<tr>
<td></td>
<td><code>listen(sd, len)</code></td>
</tr>
<tr>
<td><code>connect(sd, dest)</code></td>
<td><code>new_sd=accept(sd)</code></td>
</tr>
<tr>
<td><code>write(sd, ...) / send(sd, ...)</code></td>
<td><code>read(new_sd, ...) / recv(new_sd)</code></td>
</tr>
<tr>
<td><code>read(sd, ...) / recv(sd, ...)</code></td>
<td><code>write(new_sd, ...) / send(new_sd, ...)</code></td>
</tr>
<tr>
<td><code>close(sd)</code></td>
<td><code>close(new_sd)</code></td>
</tr>
</tbody>
</table>
## TCP - Parallel Model

<table>
<thead>
<tr>
<th>Client Side</th>
<th>Server Side</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sd=socket(type)</code></td>
<td><code>sd=socket(type)</code></td>
</tr>
<tr>
<td></td>
<td><code>bind(sd,port)</code></td>
</tr>
<tr>
<td></td>
<td><code>listen(sd,len)</code></td>
</tr>
<tr>
<td><code>connect(sd,dest)</code></td>
<td><code>new_sd=accept(sd)</code></td>
</tr>
<tr>
<td></td>
<td><code>Create another process (e.g., fork)</code></td>
</tr>
<tr>
<td></td>
<td>`close(sd)</td>
</tr>
<tr>
<td><code>write(sd,....)</code></td>
<td><code>read(new_sd,....)</code></td>
</tr>
<tr>
<td><code>read(sd,....)</code></td>
<td><code>write(new_sd,....)</code></td>
</tr>
<tr>
<td><code>close(sd)</code></td>
<td><code>close(new_sd)</code></td>
</tr>
<tr>
<td></td>
<td><code>exit()</code></td>
</tr>
</tbody>
</table>
# UDP - Serial Model

<table>
<thead>
<tr>
<th>Client Side</th>
<th>Server Side</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sd=socket(type)</code></td>
<td><code>sd=socket(type)</code></td>
</tr>
<tr>
<td><code>bind(sd,port)</code></td>
<td><code>bind(sd,port)</code></td>
</tr>
<tr>
<td><code>connect(sd,dest)</code></td>
<td><code>recvfrom(sd,...)</code></td>
</tr>
<tr>
<td><code>write(sd,...)</code></td>
<td><code>recvfrom(sd,...)</code></td>
</tr>
<tr>
<td><code>read(sd,...)</code></td>
<td><code>sendto(sd,...)</code></td>
</tr>
<tr>
<td><code>close(sd)</code></td>
<td><code>close(sd)</code></td>
</tr>
</tbody>
</table>
Chapter 2: Application layer

2.1 Principles of network applications
2.2 Web and HTTP
2.3 FTP
2.4 Electronic Mail
   ▪ SMTP, POP3, IMAP
2.5 DNS

2.6 P2P applications
2.7 Socket programming with TCP
2.8 Socket programming with UDP

+ (Bonus) Same with C
+ (Bonus) A few more functions
Address and port byte-ordering

- Address and port are stored as integers
  - `u_short sin_port;` (16 bit)
  - `in_addr sin_addr;` (32 bit)

- Problem:
  - Different machines / OS’s use different word orderings
    - little-endian: lower bytes first
    - big-endian: higher bytes first
  - These machines may communicate with one another over the network

```
struct in_addr {
    u_long s_addr;
};
```
Solution: Network Byte-Ordering

- **Defs:**
  - Host Byte-Ordering: the byte ordering used by a host (big or little)
  - Network Byte-Ordering: the byte ordering used by the network - always big-endian

- Any words sent through the network should be converted to Network Byte-Order prior to transmission (and back to Host Byte-Order once received)

- Q: should the socket perform the conversion automatically?

- Q: Given big-endian machines don’t need conversion routines and little-endian machines do, how do we avoid writing two versions of code?
UNIX’s byte-ordering funcs

- `u_long htonl(u_long x);`
- `u_short htons(u_short x);`
- `u_long ntohl(u_long x);`
- `u_short ntohs(u_short x);`

- On big-endian machines, these routines do nothing
- On little-endian machines, they reverse the byte order

- Same code would have worked regardless of endian-ness of the two machines
Dealing with blocking calls

- Many of the functions we saw block until a certain event
  - accept: until a connection comes in
  - connect: until the connection is established
  - recv, recvfrom: until a packet (of data) is received
  - send, sendto: until data is pushed into socket’s buffer
    - Q: why not until received?
- For simple programs, blocking is convenient
- What about more complex programs?
  - multiple connections
  - simultaneous sends and receives
  - simultaneously doing non-networking processing
Dealing w/ blocking (cont’d)

- **Options:**
  - create multi-process or multi-threaded code
  - turn off the blocking feature (e.g., using the `fcntl` file-descriptor control function)
  - use the `select` function call.
Other useful functions

- `bzero(char* c, int n):` 0's n bytes starting at c
- `gethostname(char *name, int len):` gets the name of the current host
- `gethostbyaddr(char *addr, int len, int type):` converts IP hostname to structure containing long integer
- `inet_addr(const char *cp):` converts dotted-decimal char-string to long integer
- `inet_ntoa(const struct in_addr in):` converts long to dotted-decimal notation
- `read(), write()`
- Warning: check function assumptions about byte-ordering (host or network). Often, they assume parameters / return solutions in network byte-order
Release of ports

- Sometimes, a “rough” exit from a program (e.g., ctrl-c) does not properly free up a port
- Eventually (after a few minutes), the port will be freed
- To reduce the likelihood of this problem, include the following code:
  ```c
  #include <signal.h>
  void cleanExit(){exit(0);}
  ```
  - in socket code:
    signal(SIGTERM, cleanExit);
    signal(SIGINT, cleanExit);
Final Thoughts

- Make sure to `#include` the header files that define used functions

- Additional info:
  - Ross and Kurose, *Computer Networking A Top-Down Approach*
  - Comer, *Internetworking with TCP/IP, ch. 21*
  - Comer and Stevens, *Internetworking with TCP/IP – Vol. 3*
  - man-pages
Chapter 2: Summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
  - hybrid

- application service requirements:
  - reliability, bandwidth, delay

- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP

- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, Skype

- socket programming
Chapter 2: Summary

**most importantly: learned about protocols**

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated

**Important themes:**
- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- “complexity at network edge”