Chapter 2
Application Layer

A note on the use of these ppt slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we’d like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

All material copyright 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved
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
Circular DHT (2)

O(N) messages on avg to resolve query, when there are N peers.

Who’s resp for key 1110?

Define closest as closest successor.
Peer Churn

- peer 5 abruptly leaves
- Peer 4 detects; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8’s immediate successor its second successor.

- What if peer 13 wants to join?

- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.
Circular DHT with Shortcuts

- each peer keeps track of IP addresses of predecessor, successor, shortcuts.
- reduced from 6 to 2 messages.
- possible to design shortcuts so $O(\log N)$ neighbors, $O(\log N)$ messages in query.

Who’s resp for key 1110?
P2P Case study: Skype

- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with SNs
- Index maps usernames to IP addresses; distributed over SNs
Peers as relays

- problem when both Alice and Bob are behind “NATs”.
  - NAT prevents an outside peer from initiating a call to insider peer

- solution:
  - using Alice’s and Bob’s SNs, relay is chosen
  - each peer initiates session with relay.
  - peers can now communicate through NATs via relay
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</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>application</th>
<th>transport</th>
<th>network</th>
<th>link</th>
<th>physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td></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</td>
<td></td>
</tr>
<tr>
<td>(message)</td>
<td></td>
</tr>
</tbody>
</table>

32 bits
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

**soorma.cs.columbia.edu**
(128.59.22.237)

**cluster.cs.columbia.edu**
(128.59.21.14, 128.59.16.7, 128.59.16.5, 128.59.16.4)

**newworld.cs.umass.edu**
(128.119.245.93)
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
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