Application Layer

Shivkumar Kalyanaraman
shivkuma@ecse.rpi.edu

Biplab Sikdar
sikdab@rpi.edu

Adapted in part from J. Kurose
(Umass)
Overview

• Conceptual + implementation aspects of network application protocols
  – client server paradigm
  – service models
• Specific protocols:
  – http, ftp, smtp, pop, dns
• Sockets
Recap: Reference Models

<table>
<thead>
<tr>
<th>TCP/IP Model</th>
<th>TCP/IP Protocols</th>
<th>OSI Ref Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>FTP</td>
<td>Application</td>
</tr>
<tr>
<td>Transport</td>
<td>Telnet</td>
<td>Presentation</td>
</tr>
<tr>
<td>Internetwork</td>
<td>HTTP</td>
<td>Session</td>
</tr>
<tr>
<td>Host to Network</td>
<td>TCP</td>
<td>Transport</td>
</tr>
<tr>
<td></td>
<td>UDP</td>
<td>Network</td>
</tr>
<tr>
<td></td>
<td>IP</td>
<td>Datalink</td>
</tr>
<tr>
<td></td>
<td>Ethernet</td>
<td>Physical</td>
</tr>
<tr>
<td></td>
<td>Packet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Telnet Radio</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Point-to-Point</td>
<td></td>
</tr>
</tbody>
</table>

“Top-down” approach means we will first learn the application layer and then learn about lower layers.
Applications and app-layer protocols

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

Application-layer protocols
- one “piece” of an app
- define messages exchanged by apps and actions taken
- user services provided by lower layer protocols
Network applications: some jargon

• A process is a program that is running within a host.

• Within the same host, two processes communicate with interprocess communication defined by the OS.

• Processes running in different hosts communicate with an application-layer protocol

• A user agent is an interface between the user and the network application.
  – Web: browser
  – E-mail: mail reader
  – Streaming audio/video: media player
Client-server paradigm

Typical network app has two pieces: \textit{client} and \textit{server}

Client:

- initiates contact with server ("speaks first")
- typically \textbf{requests} service from server,
- for Web, client is implemented in browser; for e-mail, in mail reader
Typical network app has two pieces: *client* and *server*

**Server:**
- provides requested service to client, via replies
- e.g., Web server sends requested Web page, mail server delivers e-mail
Application-layer protocols

**API: application programming interface**

- defines interface between application and transport layer
- **socket**: Internet API
  - two processes communicate by sending data into socket, reading data out of socket

**Q:** how does a process “identify” the other process with which it wants to communicate?
  - IP address of *host* running other process
  - “port number” - allows receiving host to determine to which local process the message should be delivered
What Transport Service does an App need?

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
## Transport service requirements of common apps

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Bandwidth</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>loss-tolerant</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5Kb-1Mb</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video: 10Kb-5Mb</td>
<td></td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few Kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>financial apps</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
## Services provided by Internet transport protocols

### 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 bandwidth guarantee

**Q:** why bother? Why is there a UDP?
# Internet apps: their protocols and transport protocols

<table>
<thead>
<tr>
<th>Application</th>
<th>Application layer protocol</th>
<th>Underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>smtp [RFC 821]</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>telnet [RFC 854]</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>http [RFC 2068]</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>ftp [RFC 959]</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>proprietary</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>(e.g. RealNetworks)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remote file server</td>
<td>NSF</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary</td>
<td>typically UDP</td>
</tr>
<tr>
<td>(e.g., Vocaltec)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Web: the *http* protocol

http: hypertext transfer protocol

- Web’s application layer protocol
- client/server model
  - **client**: browser that requests, receives, “displays” Web objects
  - **server**: Web server sends objects in response to requests

- http1.0: RFC 1945
- http1.1: RFC 2068
The http protocol: more

http: TCP transport service:
• client initiates TCP connection (creates socket) to server, port 80
• server accepts TCP connection from client
• http messages (application-layer protocol messages) exchanged between browser (http client) and Web server (http server)
• TCP connection closed

http is “stateless”
• server maintains no information about past client requests

Protocols that maintain “state” are complex!
• past history (state) must be maintained
• if server/client crashes, their views of “state” may be inconsistent, must be reconciled
http example

Suppose user enters URL www.School.edu/Department/home.index (contains text, references to 10 jpeg images)


1b. http server at host www.School.edu waiting for TCP connection at port 80. “accepts” connection, notifying client

2. http client sends http *request message* (containing URL) into TCP connection socket

3. http server receives request message, forms *response message* containing requested object (Department/home.index), sends message into socket
http example (cont.)


6. Steps 1-5 repeated for each of 10 jpeg objects
Non-persistent and persistent connections

Non-persistent

- HTTP/1.0
- server parses request, responds, and closes TCP connection
- 2 RTTs to fetch each object
- Each object transfer suffers from slow start

Persistent

- default for HTTP/1.1
- on same TCP connection: server, parses request, responds, parses new request,..
- Client sends requests for all referenced objects as soon as it receives base HTML.
- Fewer RTTs and less slow start.

But most 1.0 browsers use parallel TCP connections.
http message format: request

- two types of http messages: **request**, **response**
- http request message:
  - ASCII (human-readable format)

```
GET /somedir/page.html HTTP/1.0
User-agent: Mozilla/4.0
Accept: text/html, image/gif, image/jpeg
Accept-language: fr
```

(extra carriage return, line feed)

Carriage return, line feed indicates end of message
http request message: general format

<table>
<thead>
<tr>
<th>method</th>
<th>URL</th>
<th>version</th>
<th>cr</th>
<th>if</th>
</tr>
</thead>
<tbody>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>if</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>header field name</td>
<td>:</td>
<td>value</td>
<td>cr</td>
<td>if</td>
</tr>
<tr>
<td>cr</td>
<td>if</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Entity Body
http message format: *response*

status line:
(protocol status code status phrase)

HTTP/1.0 200 OK
Date: Thu, 06 Aug 1998 12:00:15 GMT
Server: Apache/1.3.0 (Unix)
Last-Modified: Mon, 22 Jun 1998 ...
Content-Length: 6821
Content-Type: text/html

data data data data data data ...

data, e.g., requested html file

header lines
http response status codes

In first line in server->client response message.

A few sample codes:

200 OK
 - request succeeded, requested object later in this message

301 Moved Permanently
 - requested object moved, new location specified later in this message (Location:)

400 Bad Request
 - request message not understood by server

404 Not Found
 - requested document not found on this server

505 HTTP Version Not Supported
Trying out http (client side) for yourself

1. Telnet to your favorite Web server:
   
   telnet www.eurecom.fr 80

   Opens TCP connection to port 80 (default http server port) at www.eurecom.fr. Anything typed in sent to port 80 at www.eurecom.fr

2. Type in a GET http request:

   GET /~ross/index.html HTTP/1.0

   By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to http server

3. Look at response message sent by http server!
User-server interaction: authentication

Authentication goal: control access to server

- **stateless**: client presents authorization in each request
- authorization: typically name, password
  - authorization: header line in request
  - Sends header line
    - WWW authenticate: if unauthorized

Browser caches name & password so that user does not have to repeatedly enter it.
User-server interaction: cookies

- server sends "cookie" to client in response msg
  
  \texttt{Set-cookie: 1678453}

- client presents cookie in later requests
  
  \texttt{cookie: 1678453}

- server matches presented-cookie with server-stored info
  - authentication
  - remembering user preferences

\begin{itemize}
  \item \texttt{usual http request msg}
  \item \texttt{usual http response + Set-cookie: #}
  \item \texttt{usual http request msg}
  \item \texttt{cookie: #}
  \item \texttt{usual http response msg}
  \item \texttt{cookie-specific action}
  \item \texttt{usual http request msg}
  \item \texttt{cookie: #}
  \item \texttt{usual http response msg}
  \item \texttt{cookie-specific action}
\end{itemize}
User-server interaction: conditional GET

- **Goal:** don’t send object if client has up-to-date stored (cached) version

- **client:** specify date of cached copy in http request
  
  \[\text{If-modified-since: } <\text{date}>\]

- **server:** response contains no object if cached copy up-to-date:

  \[
  \text{HTTP/1.0 304 Not Modified}
  \]

- server response contains object if cached copy modified:

  \[
  \text{HTTP/1.1 200 OK}
  
  \text{...}
  
  <\text{data}>
  \]
Web Caches (proxy server)

**Goal:** satisfy client request without involving origin server

- user sets browser: Web accesses via web cache
- client sends all http requests to web cache
  - if object at web cache, web cache immediately returns object in http response
  - else requests object from origin server, then returns http response to client
Why Web Caching?

Assume: cache is “close” to client (e.g., in same network)

- smaller response time: cache “closer” to client
- decrease traffic to distant servers
  - link out of institutional/local ISP network often bottleneck
Content Delivery: motivation
Content Delivery: congestion

Browsers

Routers

Networks

Web Servers
Content Delivery: idea

- Reduces load on server
- Avoids network congestion
- “Inverse” Web Caching
CDN: Architectural Layout

- Publisher informs RR of Content Availability.
- Content Pushed to Distribution System.
- Client Requests Content, Requested redirected to RR.
- RR finds the most suitable Surrogate
- Surrogate services client request.
ftp: the file transfer protocol

- transfer file to/from remote host
- client/server model
  - **client**: side that initiates transfer (either to/from remote)
  - **server**: remote host
- **ftp**: RFC 959
- **ftp server**: port 21
**ftp: separate control, data connections**

- **ftp client** contacts **ftp server** at port 21, specifying TCP as transport protocol
- **two parallel TCP connections** opened:
  - **control**: exchange commands, responses between client, server. "out of band control"
  - **data**: file data to/from server
- **ftp server** maintains "**state**": current directory, earlier authentication
ftp commands, responses

Sample commands:
- sent as ASCII text over control channel
- USER *username*
- PASS *password*
- LIST returns list of file in current directory
- RETR *filename* retrieves (gets) file
- STOR *filename* stores (puts) file onto remote host

Sample return codes:
- status code and phrase (as in http)
- 331 Username OK, password required
- 125 data connection already open; transfer starting
- 425 Can’t open data connection
- 452 Error writing file
Electronic Mail

Three major components:
• user agents
• mail servers
• simple mail transfer protocol: smtp

User Agent
• a.k.a. “mail reader”
• composing, editing, reading mail messages
• e.g., Eudora, Outlook, elm, Netscape Messenger
• outgoing, incoming messages stored on server
Electronic Mail: mail servers

Mail Servers

- **mailbox** contains incoming messages (yet to be read) for user
- **message** queue of outgoing (to be sent) mail messages
- **smtp protocol** between mail servers to send email messages
  - client: sending mail server
  - “server”: receiving mail server
Electronic Mail: smtp [RFC 821]

- uses tcp to reliably transfer email msg from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - **commands**: ASCII text
  - **response**: status code and phrase
- messages must be in 7-bit ASCII
Sample smtp interaction

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
Try smtp interaction for yourself:

- `telnet servername 25`
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands

above lets you send email without using email client (reader)
smtp: final words

- smtp uses persistent connections
- smtp requires that message (header & body) be in 7-bit ascii
- certain character strings are not permitted in message (e.g., CRLF . CRLF). Thus message has to be encoded (usually into either base-64 or quoted printable)
- smtp server uses CRLF . CRLF to determine end of message

Comparison with http

- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: each object is encapsulated in its own response message
- smtp: multiple objects message sent in a multipart message
Mail message format

**smtp:** protocol for exchanging email msgs

**RFC 822:** standard for text message format:

- header lines, e.g.,
  - To:
  - From:
  - Subject: different from smtp commands!

- body
  - the “message”, ASCII characters only
Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data ......
...........................
......base64 encoded data
```
MIME types

Content-Type: type/subtype; parameters

Text
• example subtypes: plain, html

Image
• example subtypes: jpeg, gif

Audio
• example subtypes: basic (8-bit mu-law encoded), 32kadpcm (32 kbps coding)

Video
• example subtypes: mpeg, quicktime

Application
• other data that must be processed by reader before “viewable”
• example subtypes: msword, octet-stream
Multipart Type

From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789

--98766789
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain

Dear Bob,
Please find a picture of a crepe.

--98766789
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data ....
........................
......base64 encoded data
--98766789--
Mail access protocols

- **SMTP**: delivery/storage to receiver’s server
- Mail access protocol: retrieval from server
  - **POP**: Post Office Protocol [RFC 1939]
    - authorization (agent ↔ server) and download
  - **IMAP**: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
  - **HTTP**: Hotmail, Yahoo! Mail, etc.
POP3 protocol

authorization phase
• client commands:
  – user: declare username
  – pass: password
• server responses
  – +OK
  – -ERR

transaction phase, client:
• list: list message numbers
• retr: retrieve message by number
• dele: delete
• quit

S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
DNS: Domain Name System

People: many identifiers:
  – SSN, name, Passport #

Internet hosts, routers:
  – IP address (32 bit) - used for addressing datagrams
  – “name”, e.g., gaia.cs.umass.edu - used by humans

Q: map between IP addresses and name?

Domain Name System:
  • distributed database implemented in hierarchy of many name servers
  • application-layer protocol host, routers, name servers to communicate to resolve names (address/name translation)
    – note: core Internet function implemented as application-layer protocol
    – complexity at network’s “edge”
DNS name servers

Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn’t *scale*!

• no server has all name-to-IP address mappings

**local name servers:**
- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

**authoritative name server:**
- for a host: stores that host’s IP address, name
- can perform name/address translation for that host’s name

Rensselaer Polytechnic Institute
Shivkumar Kalvanaraman, Biplab Sikdar
DNS: Root name servers

- contacted by local name server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
- ~ dozen root name servers worldwide
Simple DNS example

host `surf.eurecom.fr` wants IP address of `gaia.cs.umass.edu`

1. Contacts its local DNS server, `dns.eurecom.fr`
2. `dns.eurecom.fr` contacts root name server, if necessary
3. root name server contacts authoritative name server, `dns.umass.edu`, if necessary
DNS example

Root name server:
• may not know authoritative name server
• may know **intermediate name server:** who to contact to find authoritative name server

Requesting host: surf.eurecom.fr
Local name server: dns.eurecom.fr
Intermediate name server: dns.umass.edu
Authoritative name server: dns.cs.umass.edu
Gaia.cs.umass.edu
DNS: iterated queries

**recursive query:**
- puts burden of name resolution on contacted name server
- heavy load?

**iterated query:**
- contacted server replies with name of server to contact
- “I don’t know this name, but ask this server”
DNS: caching and updating records

- once (any) name server learns mapping, it caches mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136
DNS records

**DNS:** distributed db storing resource records (RR)

**RR format:** (name, value, type, ttl)

- **Type=A**
  - name is hostname
  - value is IP address

- **Type=NS**
  - name is domain (e.g. foo.com)
  - value is IP address of authoritative name server for this domain

- **Type=CNAME**
  - name is an alias name for some “cannonical” (the real) name
  - value is cannonical name

- **Type=MX**
  - value is hostname of mailserver associated with name
DNS protocol, messages

**DNS protocol**: *query* and *reply* messages, both with same *message format*

**msg header**
- **identification**: 16 bit # for query, reply to query uses same #
- **flags**:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative

<table>
<thead>
<tr>
<th>identification</th>
<th>flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of questions</td>
<td>number of answer RRs</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>number of additional RRs</td>
</tr>
<tr>
<td>questions (variable number of questions)</td>
<td></td>
</tr>
<tr>
<td>answers (variable number of resource records)</td>
<td></td>
</tr>
<tr>
<td>authority (variable number of resource records)</td>
<td></td>
</tr>
<tr>
<td>additional information (variable number of resource records)</td>
<td></td>
</tr>
</tbody>
</table>
### DNS protocol, messages

- **Name, type fields** for a query
- **RRs in response to query**
- **Records for authoritative servers**
- **Additional “helpful” info that may be used**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identification</td>
<td>Unique identifier for each query.</td>
</tr>
<tr>
<td>flags</td>
<td>Flags that indicate the query type and other attributes.</td>
</tr>
<tr>
<td>number of questions</td>
<td>Number of questions in the query.</td>
</tr>
<tr>
<td>number of answer RRs</td>
<td>Number of answer records in response.</td>
</tr>
<tr>
<td>number of authority RRs</td>
<td>Number of authority records in response.</td>
</tr>
<tr>
<td>number of additional RRs</td>
<td>Number of additional records in response.</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>section</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>questions</td>
<td>(variable number of questions)</td>
</tr>
<tr>
<td>answers</td>
<td>(variable number of resource records)</td>
</tr>
<tr>
<td>authority</td>
<td>(variable number of resource records)</td>
</tr>
<tr>
<td>additional information</td>
<td>(variable number of resource records)</td>
</tr>
</tbody>
</table>

**Diagram:**

12 bytes
Sockets

**Goal:** learn models of 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.
TCP sockets

**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

Diagram:
- **socket**: controlled by application developer
- **TCP with buffers, variables**: controlled by operating system
- **process**: controlled by application developer
- **host or server**: controlled by operating system
- **internet**: bidirectional connection
TCP sockets

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

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

Example client-server app:

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

Input stream: sequence of bytes into process

Output stream: sequence of bytes out of process
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
UDP Sockets

**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*
Client/server socket interaction: UDP

Server (running on hostid)

create socket, port=x, for incoming request:
s = DatagramSocket()

read request from s

write reply to s
specifying client host address, port number

Client

create socket, clientSocket = DatagramSocket()

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

read reply from clientSocket

close clientSocket
Our study of network apps now complete!

- **application service requirements:**
  - reliability, bandwidth, delay
- **client-server paradigm**
- **Internet transport service model**
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- **specific protocols:**
  - http
  - ftp
  - smtp, pop3
  - dns
- **sockets**
  - client/server implementation
  - using tcp, udp sockets
Application Layer: 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
- control vs. data msgs
  - in-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- “complexity at network edge”
- security: authentication