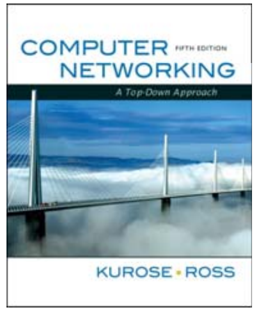
Chapter 2
Application Layer



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

Computer Networking: A Top Down Approach, 5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Chapter 2: Application layer

Chapter 2: Application Layer

- Our goals:
- conceptual, implementation aspects of network application protocols
  - transport-layer service models
  - client-server paradigm
  - peer-to-peer paradigm
- learn about protocols by examining popular application-level protocols
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- programming network applications
  - socket API

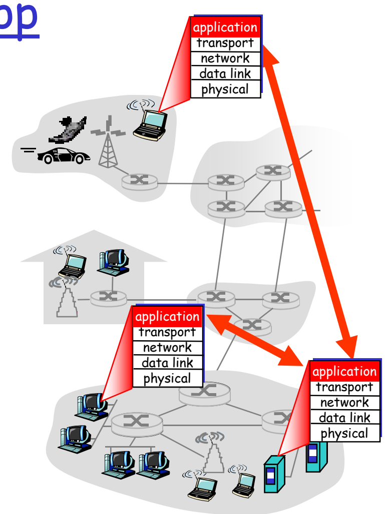
Some network apps

- e-mail
- web
- instant messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video clips
- voice over IP
- real-time video conferencing
- grid computing

Creating a network app

write programs that
- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

No need to write software for network-core devices
- Network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



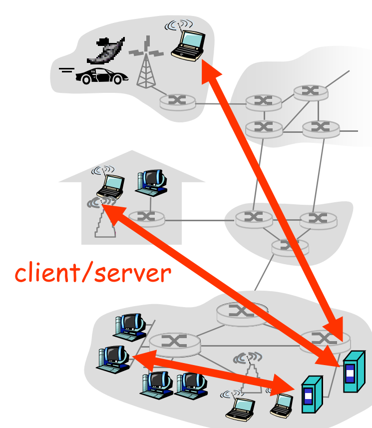
Chapter 2: Application layer

- 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

Application architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P

Client-server architecture

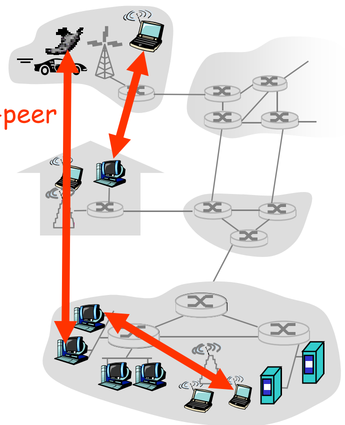


server:
- always-on host
- permanent IP address
- server farms for scaling

clients:
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Pure P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses



- Highly scalable but difficult to manage

Hybrid of client-server and P2P

Skype
- voice-over-IP P2P application
- centralized server: finding address of remote party:
- client-client connection: direct (not through server)

Instant messaging
- chatting between two users is P2P
- centralized service: client presence detection/location
  - user registers its IP address with central server when it comes online
  - user contacts central server to find IP addresses of buddies

Processes communicating

Process: program running within a host.
- within same host, two processes communicate using inter-process communication (defined by OS).
- processes in different hosts communicate by exchanging messages

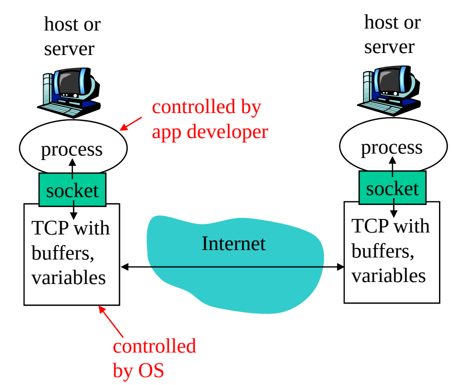
Client process: process that initiates communication

Server process: process that waits to be contacted

- Note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door which brings message to socket at receiving process



- API: (1) choice of transport protocol; (2) ability to fix a few parameters (lots more on this later)

Application Programming Interface

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host suffice for identifying the process?

Addressing processes

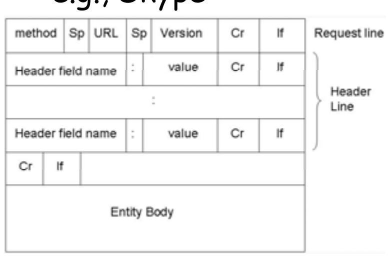
- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *Q:* does IP address of host on which process runs suffice for identifying the process?
  - *A:* No, *many* processes can be running on *same host*
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - Port number: 80
- more shortly…

App-layer protocol defines

- **Types of messages** exchanged,
  - e.g., request, response
- ***Message syntax***:
  - what fields in messages & how fields are delineated
- ***Message semantics***
  - meaning of information in fields
- **Rules** for when and how processes send & respond to messages

Public-domain protocols:
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

Proprietary protocols:
- e.g., Skype



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"

Throughput
- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

Security
- Encryption, data integrity, …

Transport service requirements of common apps

Application	Data loss	Throughput	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

TCP service:

- *connection-oriented:* setup required between client and server processes
- *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 throughput guarantees, security

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, throughput guarantee, or security

Q: why bother? Why is there a UDP?

Internet apps: application, 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	HTTP (eg Youtube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	typically UDP

RTP (Real-Time Application Protocol) ถูกแทนโดย RFC 3550
SIP (Session Initiation Protocol) RFC 3261
RTP (Real-time Transport Protocol) RFC 1889

Chapter 2: Application layer

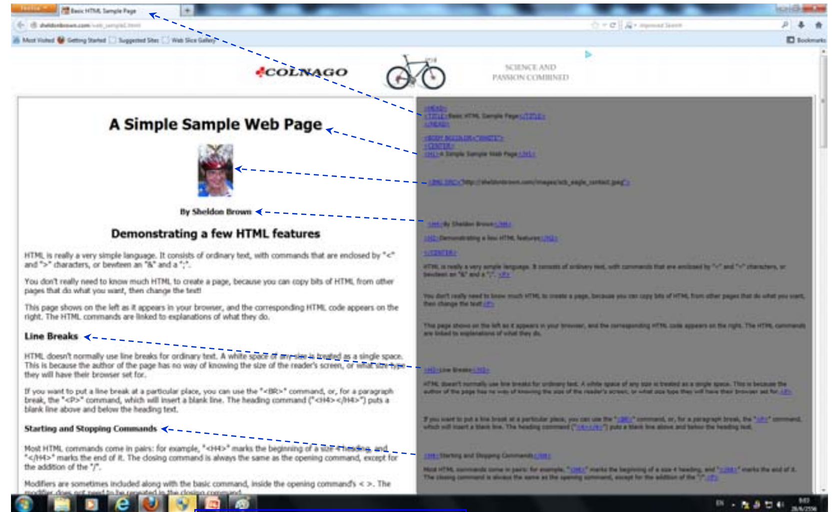
- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P applications
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP

Web and HTTP

First some jargon

- Web page consists of objects
- Object can be HTML file, JPEG image, Java applet, audio file,…
- Web page consists of base HTML-file which includes several referenced objects
- Each object is addressable by a URL (Uniform Resource Locator)
- Example URL:

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

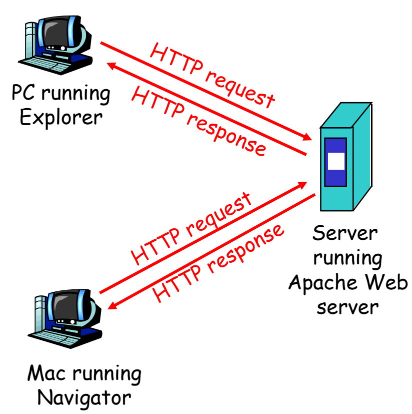



```
<html>
<head>
<title>Document Title</title>
</head>
<body >
.....................
</body>
</html>
```


HTTP overview

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



Web browser : Internet Explorer, FireFox, Google Chrome
Web Server : Apache, Internet Information Service (IIS)

HTTP overview (continued)

Uses TCP:

- 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

aside

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 connections

Nonpersistent HTTP

- At most one object is sent over a TCP connection.

Persistent HTTP

- Multiple objects can be sent over single TCP connection between client and server.

Nonpersistent HTTP

Suppose user enters URL

`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

1b. HTTP server at host www.someSchool.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. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Nonpersistent HTTP (cont.)

4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects

time

Non-Persistent HTTP: Response time

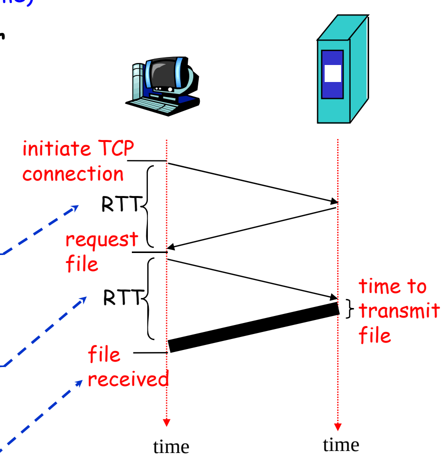
(RTT : Round Trip Time)

Definition of RTT: time for a small packet to travel from client to server and back.

Response time:

- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- file transmission time

total = 2RTT+transmit time



Persistent HTTP

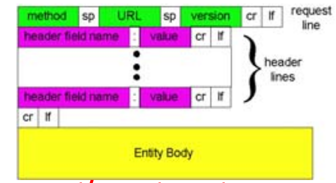
Nonpersistent HTTP issues:

- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

Persistent HTTP

- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP Request Message



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

request line (GET, POST, HEAD commands)

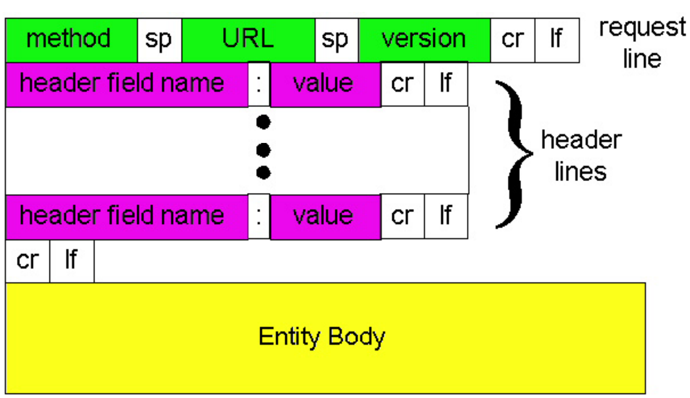
```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr
```

header lines

Carriage return, line feed indicates end of message

(extra carriage return, line feed)

HTTP Request Message: general format



Uploading form input

Post method:

- Web page often includes form input
- Input is uploaded to server in entity body

URL method:

- Uses GET method
- Input is uploaded in URL field of request line:

`www.somesite.com/animalsearch?monkeys&banana`

GET www.somsite.com/animalsearch?monkeys&banana HTTP/1.1

Method types

HTTP/1.0 (RFC 1945)

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1 (RFC (2616)

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

HTTP response message

status line (protocol status code status phrase)

```
HTTP/1.1 200 OK
```

header lines

```
Connection close
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, e.g., requested HTML file

```
data data data data data ...
```


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 cis.poly.edu 80
```

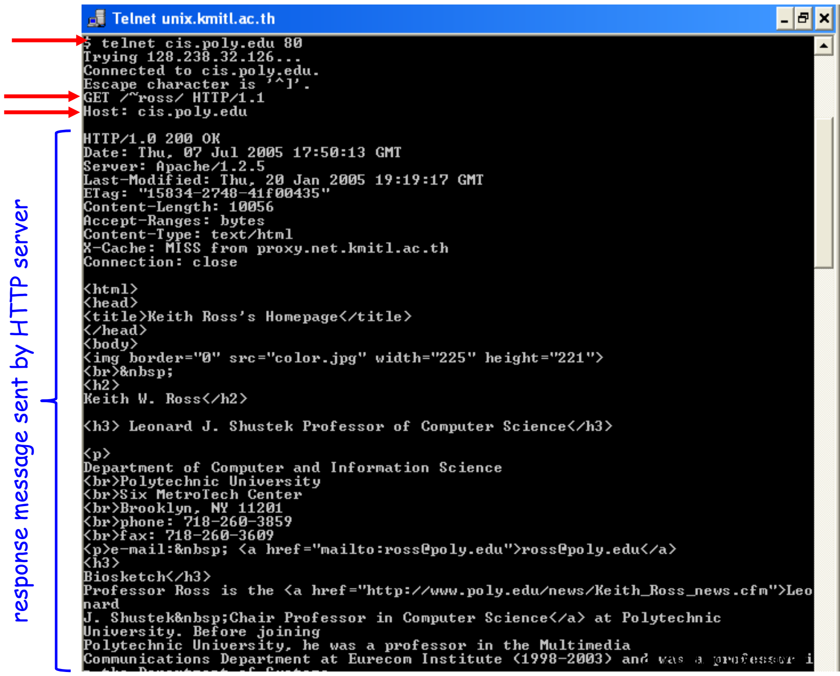
Opens TCP connection to port 80 (default HTTP server port) at cis.poly.edu. Anything typed in sent to port 80 at cis.poly.edu

2. Type in a GET HTTP request:

```
GET /~ross/ HTTP/1.1
Host: cis.poly.edu
```

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!



```
Telnet unix.kmitl.ac.th
$ telnet cis.poly.edu 80
Trying 128.238.32.126...
Connected to cis.poly.edu.
Escape character is '^]'.
GET /~ross/ HTTP/1.1
Host: cis.poly.edu

HTTP/1.0 200 OK
Date: Thu, 07 Jul 2005 17:50:13 GMT
Server: Apache/1.2.5
Last-Modified: Thu, 20 Jan 2005 19:19:17 GMT
ETag: "15834-2748-41f00435"
Content-Length: 10056
Accept-Ranges: bytes
Content-Type: text/html
X-Cache: MISS from proxy.net.kmitl.ac.th
Connection: close

<html>
<head>
<title>Keith Ross's Homepage</title>
</head>
<body>
<img border="0" src="color.jpg" width="225" height="221">
<br> 
<h2>
Keith W. Ross</h2>

<h3> Leonard J. Shustek Professor of Computer Science</h3>

<p>
Department of Computer and Information Science
<br>Polytechnic University
<br>Six MetroTech Center
<br>Brooklyn, NY 11201
<br>phone: 718-260-3859
<br>fax: 718-260-3609
<p>e-mail:  <a href="mailto:ross@poly.edu">ross@poly.edu</a>
<h3>
Biosketch</h3>
Professor Ross is the <a href="http://www.poly.edu/news/Keith_Ross_news.cfm">Leo
nard
J. Shustek Chair Professor in Computer Science</a> at Polytechnic
University. Before joining
Polytechnic University, he was a professor in the Multimedia
Communications Department at Eurecom Institute (1998-2003) and was a professor i
```


User-server state: cookies

Many major Web sites use cookies

Four components:

1) cookie header line of HTTP *response* message
2) cookie header line in HTTP *request* message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

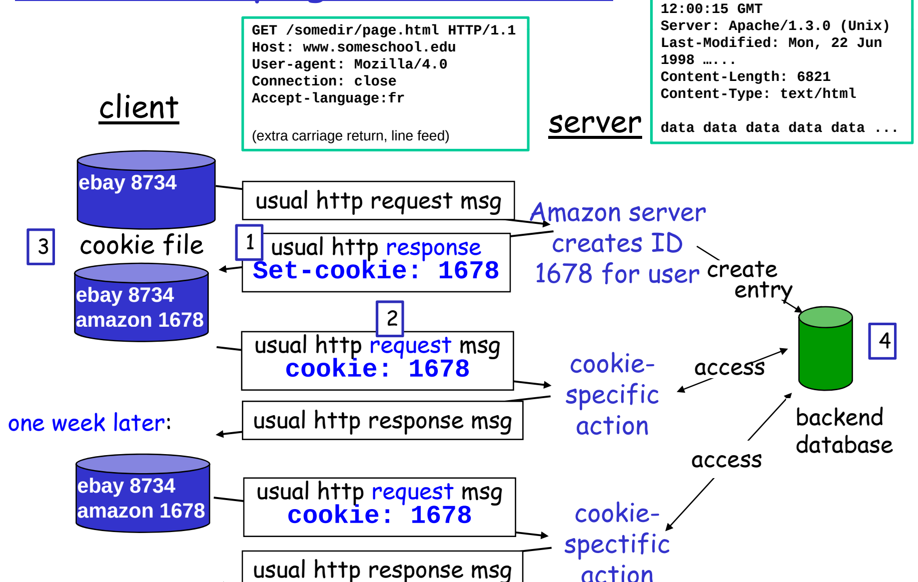
Example:

- Susan always access Internet always from PC
- visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID
  - entry in backend database for ID

Cookies: keeping "state" (cont.)

```
GET /somedir/page.html HTTP/1.1
Host: www.someschool.edu
User-agent: Mozilla/4.0
Connection: close
Accept-language:fr

(extra carriage return, line feed)
```

```
HTTP/1.1 200 OK
Connection close
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 ...
```



Cookies (continued)

What cookies can bring:

- authorization
- shopping carts
- recommendations
- user session state (Web e-mail)

aside

Cookies and privacy:

- cookies permit sites to learn a lot about you
- you may supply name and e-mail to sites

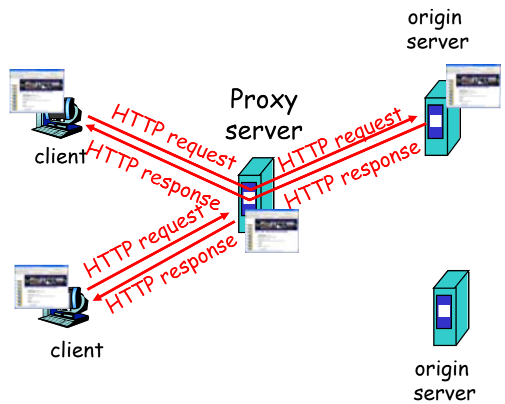
How to keep "state":

- protocol endpoints: maintain state at sender/receiver over multiple transactions
- cookies: http messages carry state

Web caches (proxy server)

Goal: satisfy client request without involving origin server

- user sets browser: Web accesses via cache
- browser sends all HTTP requests to cache
  - object in cache: cache returns object
  - else cache requests object from origin server, then returns object to client



More about Web caching

- cache acts as both client and server
- typically cache is installed by ISP (university, company, residential ISP)

Why Web caching?

- reduce response time for client request
- reduce traffic on an institution's access link.
- Internet dense with caches: enables "poor" content providers to effectively deliver content (but so does P2P file sharing)

Caching example

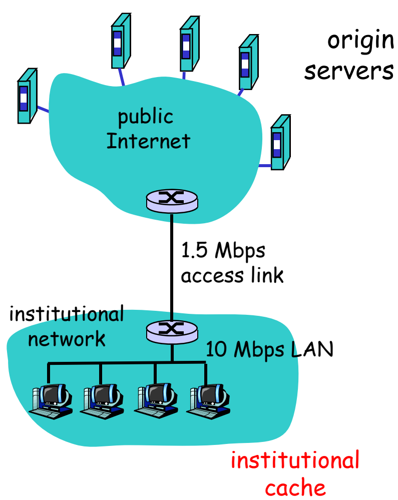
Assumptions

- average object size = 100,000 bits
- averge request rate from institution's browsers to origin servers = 15 requests/sec
- delay from institutional router to any origin server and back to router = 2 sec

Consequences

- utilization on LAN = 15%
- utilization on access link = 100%
- total delay = Internet delay + access delay + LAN delay

= 2 sec + minutes + milliseconds



Caching example (cont)

possible solution

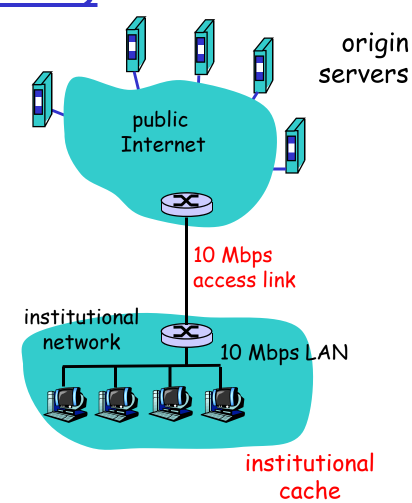
- increase bandwidth of access link to, say, 10 Mbps

consequence

- utilization on LAN = 15%
- utilization on access link = 15%
- Total delay = Internet delay + access delay + LAN delay

= 2 sec + msecs + msecs

- often a costly upgrade



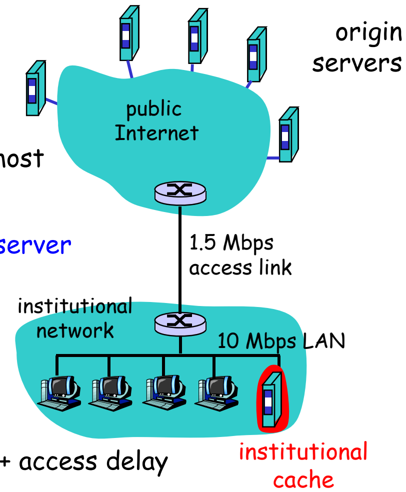
Caching example (cont)

possible solution: install cache

- suppose hit rate is 0.4

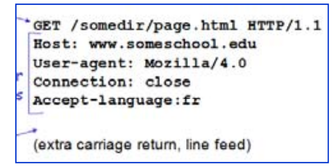
consequence

- 40% requests will be satisfied almost immediately
- 60% requests satisfied by origin server
- utilization of access link reduced to 60%, resulting in negligible delays (say 10 msec)

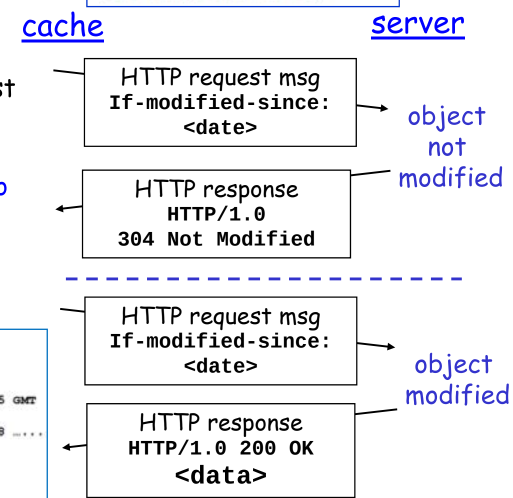


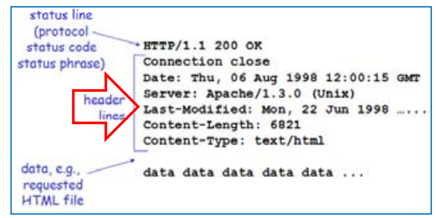
- total avg delay = Internet delay + access delay + LAN delay = 0.6*(2.01) secs + 0.4*milliseconds < 1.4 secs

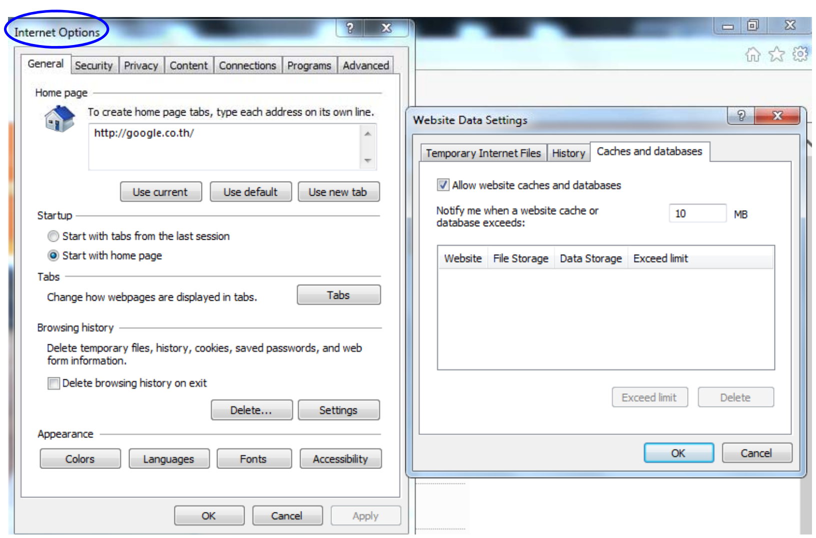
Conditional GET

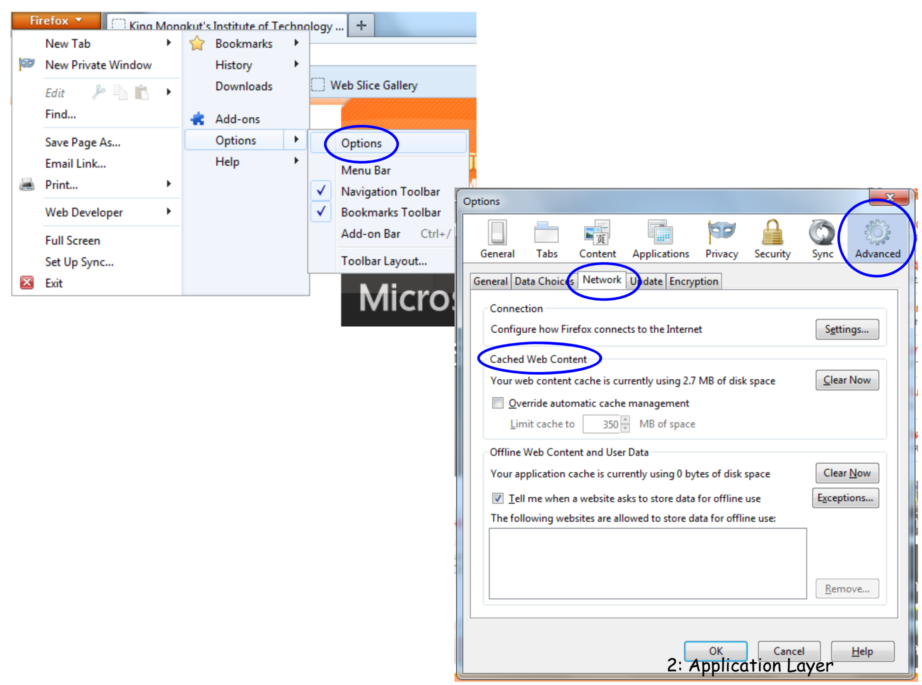


- Goal: don't send object if cache has up-to-date cached version
- cache: specify date of cached copy in HTTP request
  - `If-modified-since: <date>`
- server: response contains no object if cached copy is up-to-date:
  - `HTTP/1.0 304 Not Modified`





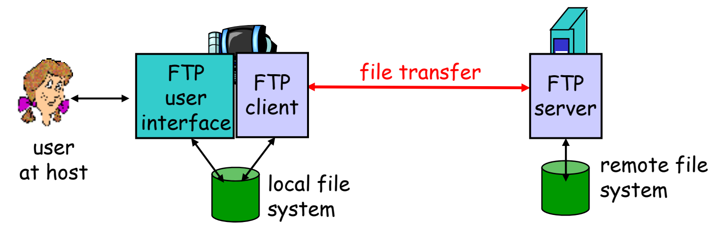




Chapter 2: Application layer

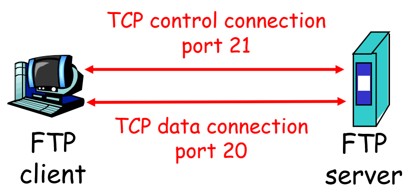
- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P applications
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

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, TCP is transport protocol
- client authorized over control connection
- client browses remote directory by sending commands over control connection.
- when server receives file transfer command, server opens *2nd* TCP connection (for file) to client
- after transferring one file, server closes data connection.
- server opens another TCP data connection to transfer another file.
- control connection: "out of band"
- FTP server maintains "state": current directory, earlier authentication

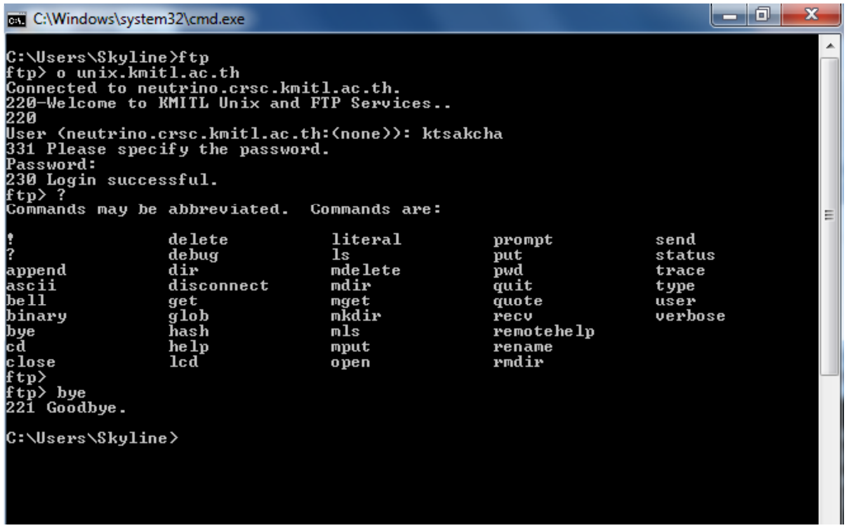
FTP commands, responses

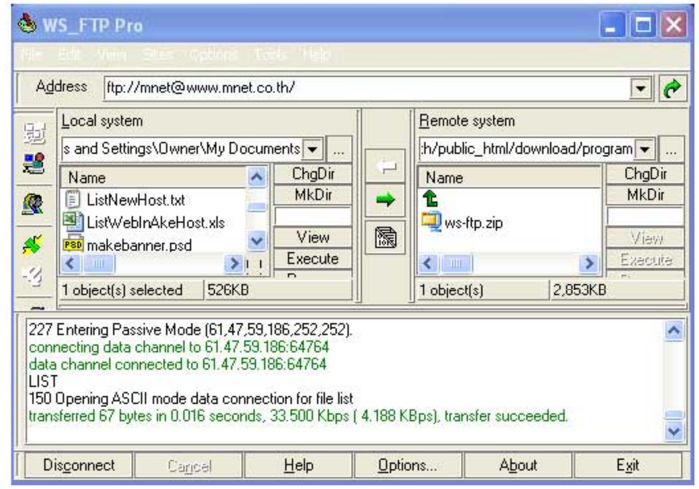
Sample commands:

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

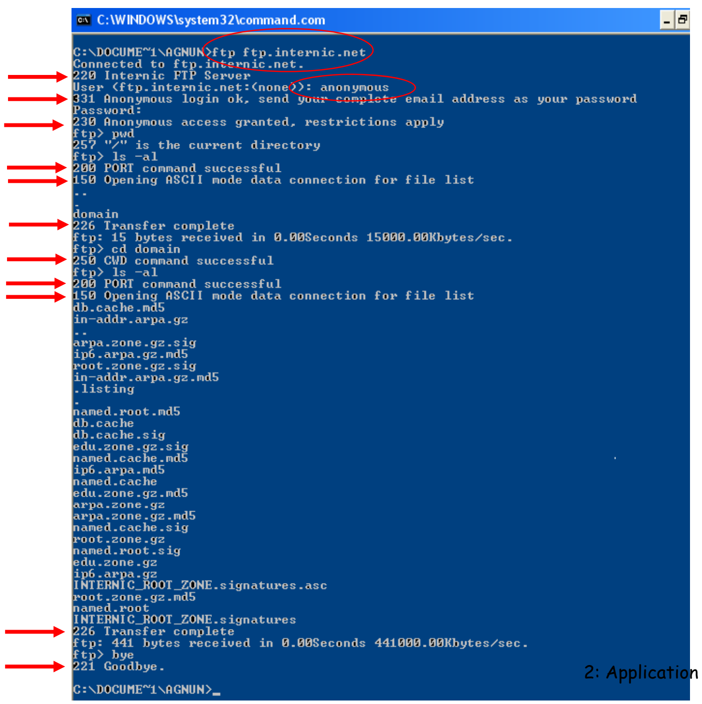
Sample return codes

- 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**

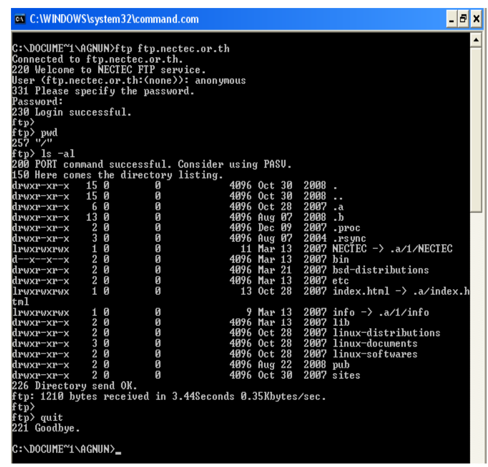




ดาวน์โหลดโปรแกรม WS_FTP Pro

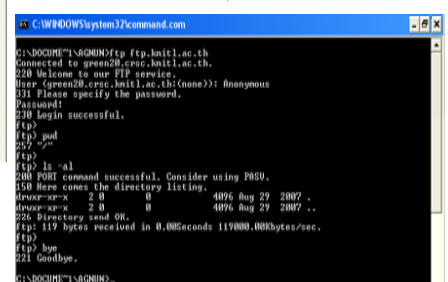


Example Anonymous FTP Servers



ftp.nectec.or.th

ftp.kmitl.ac.th



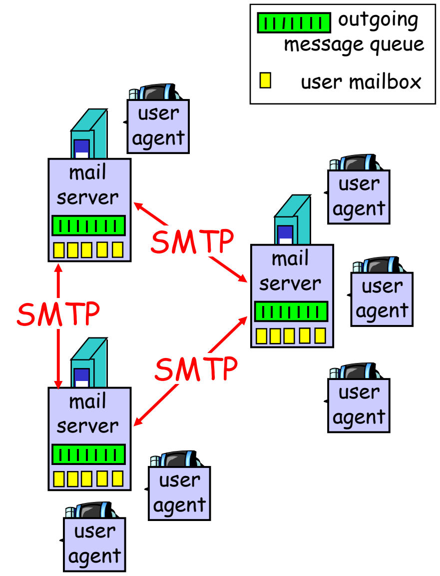
Chapter 2: Application layer

- 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

Electronic Mail

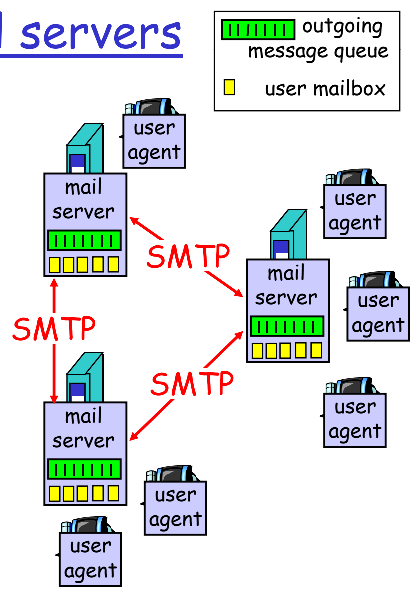
Three major components:
- User Agents
- Mail Servers
- Simple Mail Transfer Protocol: SMTP (RFC 2821)

User Agent
- a.k.a. "Mail Reader"
- Composing, Editing, Reading Mail Messages
- e.g., Eudora, Outlook, elm, Mozilla Thunderbird
- outgoing, incoming messages stored on server



Electronic Mail: mail servers

Mail Servers
- mailbox contains incoming messages 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

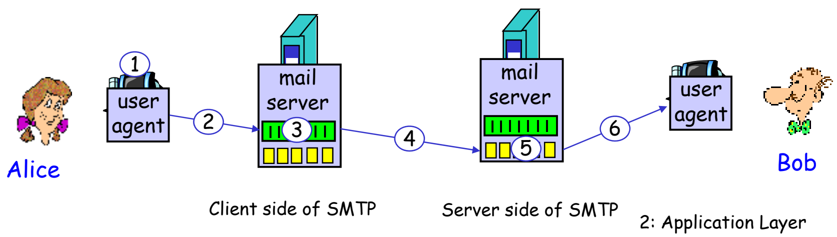


Electronic Mail: SMTP [RFC 2821]

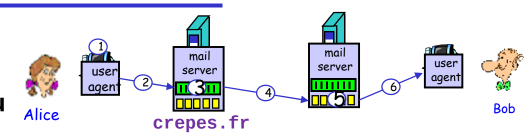
- **uses TCP** to reliably transfer email message from client to server, port 25
- **direct transfer**: sending server to receiving server
- **three phases of transfer**
  - handshaking (greeting)
  - transfer of messages
  - closure
- **command/response interaction**
  - commands: ASCII text
  - response: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

Scenario: Alice sends message to Bob

1) Alice uses user agent (UA) to compose message and "to" bob@someschool.edu
2) Alice's UA sends message to her mail server; message placed in message queue
3) Client side of SMTP opens TCP connection with Bob's mail server
4) SMTP client sends Alice's message over the TCP connection
5) Bob's mail server places the message in Bob's mailbox
6) Bob invokes his user agent to read message



Sample SMTP interaction



crepes.fr hamburger.edu

```
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
```

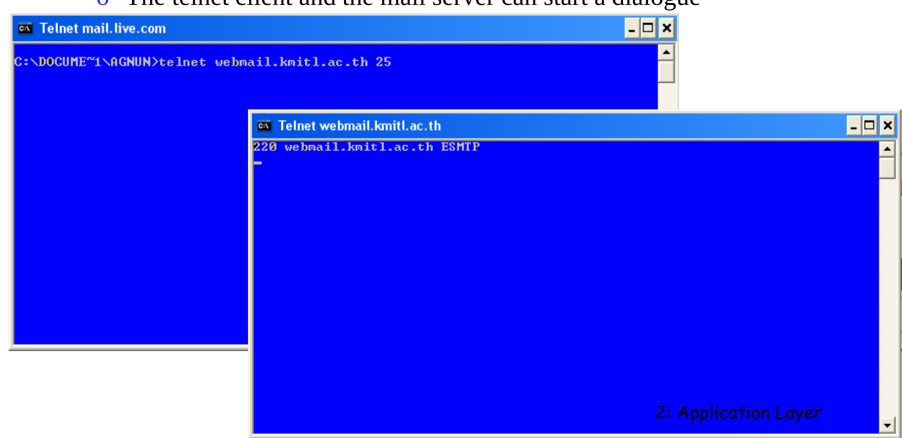

Try SMTP interaction for yourself:

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

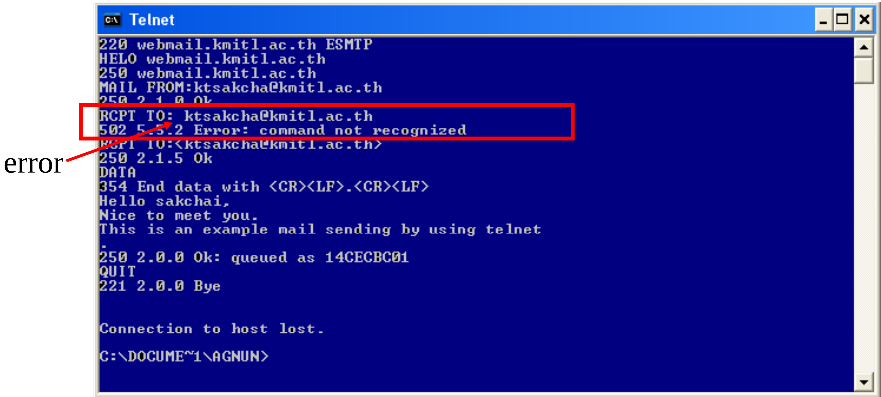
above lets you send email without using email client (reader)

An Example SMTP Session

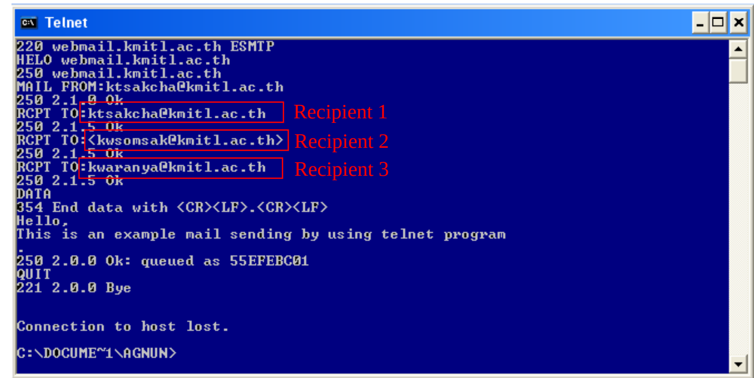
- How to connect to an SMTP server?
  - telnet servername 25
  - A TCP connection gets established over port number 25
  - The telnet client and the mail server can start a dialogue



An Example SMTP Session (cont.)



An Example SMTP Session (cont.)



Sending email to several recipients

SMTP: final words

- SMTP uses persistent connections
- SMTP requires message (header & body) to be in 7-bit ASCII
- SMTP server uses **CRLF.CRLF** to determine end of message

Comparison with HTTP:

- HTTP: pull
- SMTP: push
- both have ASCII command/response interaction, status codes
- HTTP: each object encapsulated in its own response message
- SMTP: multiple objects sent in multipart message

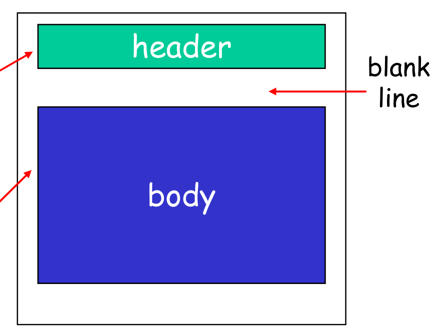
Mail Message Format

SMTP (RFC 2821) : protocol for exchanging email messages

RFC 822: standard for text message format:

- **header lines**, e.g.,
  - To:
  - From:
  - Subject:

  different from SMTP commands!
- **body**
  - the "message", ASCII characters only



SMTP Command : HELO, MAIL FROM, RCPT TO, DATA, QUIT

Mail Message Format

MIME – Multipurpose Internet Mail Extensions (RFC 2045, 2056)

MIME = ไม่ มี, มิมี

- Multimedia mail extensions
- SMTP requires all data to be *7-bit ASCII characters*
  - All non-ASCII data must be encoded as ASCII strings
- Additional lines in the message header declare MIME content type

MIME version

Method used to encode data

Multimedia data type, subtype, parameter declaration

ASCII encoded data

```
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 and subtypes

Type	Subtype	Description
Text	Plain html	Unformatted Text Html format
Image	Gif Jpeg bmp	Still picture in GIF format Still picture in JPEG format Still picture in BMP format
Audio	Basic X-wav	Audible sound (au) Wave file
Video	Mpeg Quicktime	Movie in MPEG format Mov.
Multipart	Mixed	Independent parts in the specified order

MIME Types : *Multipart Types*

MIME version

Header/Body separators

```
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--
```


Received Message

- Receiving Server ที่ได้รับ message ด้วย RFC 822 และ MIME header line จะทำการเพิ่ม
- Received : header line ในบรรทัดบนสุดของ message

Received: from crepes.fr by hamburger.edu; 12 Oct 98 15:27:39 GMT
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

Received Message

- บางครั้ง บรรทัด Received: มีหลายบรรทัด เช่น

```
Received: from hamburger.edu by sushi.jp; 3 Jul 01 15:30:01 GMT
Received: from crepes.fr by hamburger.edu; 3 Jul 01 15:17:39 GMT
```

- เกิดจาก user มีการ forward mail ไปยัง SMTP Server อื่นๆ

Base64 Encoding Table

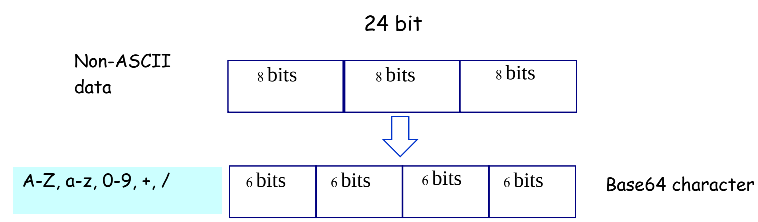
Base64 Encoding Table

Value	Binary	Char	Value	Binary	Char	Value	Binary	Char	Value	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
0	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

Base64 Encoding

1. ทำการเข้ารหัสครั้งละ 3 ตัวอักขระ โดยทำการแปลงแต่ละตัวอักขระเป็นเลขฐานสอง ขนาด 8 บิต โดยเทียบจากตาราง ASCII (ให้เติม 0 ตรงตำแหน่งบิตซ้ายสุด ให้เป็น 8 บิต)
2. นำเลขฐานสองของแต่ละตัวอักขระมาเขียนเรียงกัน (24 บิต)
3. ทำการแบ่งออกเป็นชุดๆ ละ 6 บิต (ได้ทั้งสิ้น 4 ชุด)
4. ทำการแปลงแต่ละชุดให้เป็นตัวอักขระ โดยเทียบจากตาราง Base64
5. กลับไปทำขั้นตอนที่ 1 ใหม่ จนกระทั่ง เหลือ<u>**ข้อมูลกลุ่มสุดท้าย**</u> ซึ่งมีได้ 3 กรณีดังนี้
   - กรณีที่ 1 ข้อมูลมี 3 ตัวอักขระพอดี (ทำขั้นตอนที่ 1 ถึง 4)
   - กรณีที่ 2 ข้อมูลมี จำนวน 2 ตัวอักขระ
   - กรณีที่ 3 ข้อมูลมี จำนวน 1 ตัวอักขระ
6. ทำการแปลงตัวอักขระที่เหลือ (2 ตัวหรือ 1 ตัว) ให้เป็นเลขฐานสอง
7. **เติม 0 ให้ครบ 24 บิต**
8. ทำการแบ่งออกเป็นชุดๆ ละ 6 บิต (ได้ทั้งสิ้น 4 ชุด)
9. ทำการแปลงแต่ละชุดให้เป็นตัวอักขระ โดยเทียบจากตาราง Base64 **ถ้าชุดใดเป็น 0 ทั้งหมดให้แทนด้วย “=”**

Base64 Encoding



<u>**ข้อมูลกลุ่มสุดท้าย**</u> ซึ่งมีได้ 3 กรณีดังนี้

- กรณีที่ 1 ข้อมูลมี 3 ไบต์พอดี (ทำขั้นตอนที่ 1 ถึง 3)
- กรณีที่ 2 ข้อมูลมี จำนวน 2 ไบต์ (ขาด 1 ไบต์ ครบ 3 ไบต์)
- กรณีที่ 3 ข้อมูลมี จำนวน 1 ไบต์ (ขาด 2 ไบต์ ครบ 3 ไบต์)

- กรณีที่ 2 และ 3 **เติม 0 ให้ครบ 24 บิต**
- ทำการแบ่งออกเป็นชุดๆ ละ 6 บิต (ได้ทั้งสิ้น 4 ชุด)
- ทำการแปลงแต่ละชุดให้เป็นตัวอักขระ โดยเทียบจากตาราง Base64 **ถ้าชุดใดเป็น 0 ทั้งหมดให้แทนด้วย “=” (กรณีที่ 2 จะมี “=” กรณีที่ 3 จะมี “==”)**

กรณีที่ 1 ครบ 3 ไบต์ หรือ 24 บิต

- จงแปลง ABC ให้อยู่ในรูปของ Base64
- $A=41_{16}$, $B=42_{16}$, $C=43_{16}$
- 0100 0001 0100 0010 0100 0011
- 0100 0001 0100 0010 0100 0011
- Q U J D

- ABC → QUJD

กรณี 2 เหลือ 2 ไบต์

- จงแปลง ABCDE ให้อยู่ในรูปของ Base64
- $A=41_{16}$, $B=42_{16}$, $C=43_{16}$, $D=44_{16}$, $E=45_{16}$
- 0100 0001 0100 0010 0100 0011 01000100 01000101
- 0100 0001 0100 0010 0100 0011 01000100 01000101 00000000
- Q U J D R E U =

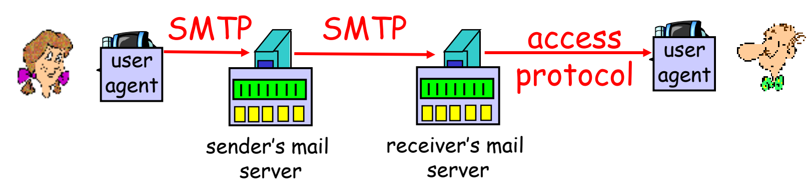
- ABCDE → QUJDREU=

กรณี 3 เหลือ 1 ไบต์

- จงแปลง ABCD ให้อยู่ในรูปของ Base64
- $A=41_{16}$, $B=42_{16}$, $C=43_{16}$,$D=44_{16}$
- 0100 0001 0100 0010 0100 0011 01000100
- 0100 0001 0100 0010 0100 0011 01000100 0000000000000000
- Q U J D R A = =

- ABCD → QUJDRA==

Mail access protocols



- SMTP: delivery/storage to receiver's server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939] Port No. = 110
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730] Port No. = 143
    - more features (more complex)
    - manipulation of stored messags on server
  - HTTP: gmail, 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 bob
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
```


POP3 (more) and IMAP

More about POP3

- Previous example uses "download and delete" mode.
- Bob cannot re-read e-mail if he changes client
- "Download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

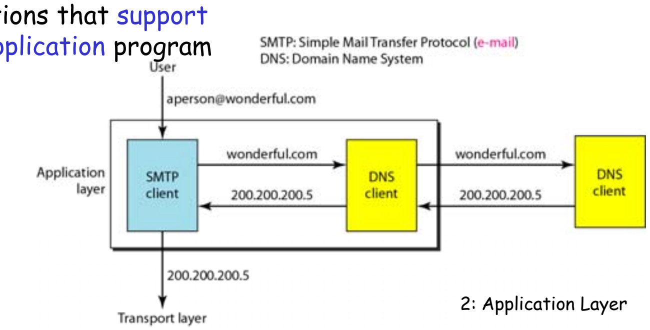
- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name

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

DNS: Domain Name System

- Client/server applications can be divided into two categories
  - Applications that can be directly used by user, such as e-mail
  - Applications that support other application program
- Domain Name System (DNS) is a supporting program that us used by other programs



DNS: Domain Name System

Social security Number (SSN)

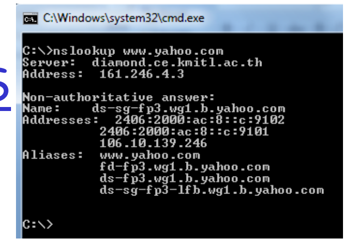
People: many identifiers:
- SSN, name, passport #

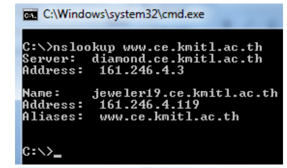
Internet hosts, routers:
- IP address (32 bit) - used for addressing datagrams
- "name", e.g., www.yahoo.com - used by humans

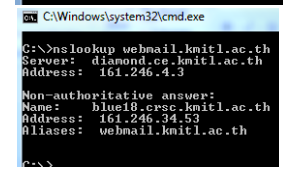
Q: map between IP addresses and name ?

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





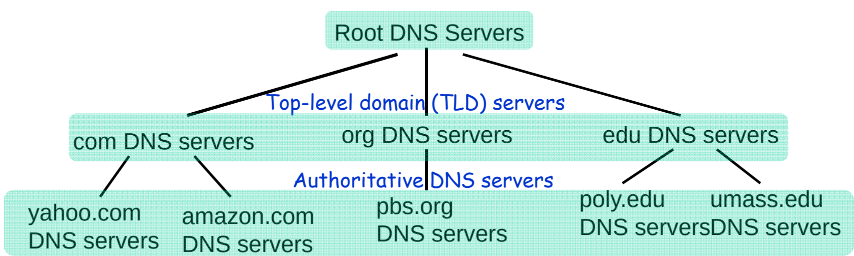


DNS services
- hostname to IP address translation
- host aliasing
  - Canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: set of IP addresses for one canonical name

Why not centralize DNS?
- single point of failure
- traffic volume
- distant centralized database
- maintenance

doesn't *scale!*

Distributed, Hierarchical Database



Client wants IP for www.amazon.com; 1st approx:
- client queries a root server to find com DNS server
- client queries com DNS server to get amazon.com DNS server
- client queries amazon.com DNS server to get IP address for www.amazon.com

DNS: Root name servers

- contacted by Local Name Server that can not resolve name
- root name server:
  - contacts authoritative name server if name mapping is known
  - gets mapping
  - returns mapping to local name server

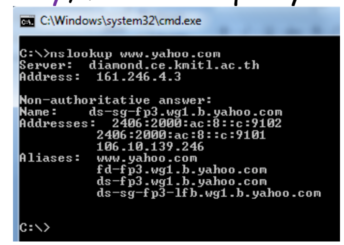
13 root name servers worldwide

TLD and Authoritative Servers

- Top-level domain (TLD) servers:
  - responsible for com, org, net, edu, etc, and all top-level country domains uk, fr, ca, jp.
  - Network Solutions maintains servers for com TLD
  - Educause for edu TLD
- Authoritative DNS servers:
  - organization's DNS servers, providing authoritative hostname to IP mappings for organization's servers (e.g., Web, mail).
  - can be maintained by organization or service provider

Local Name Server

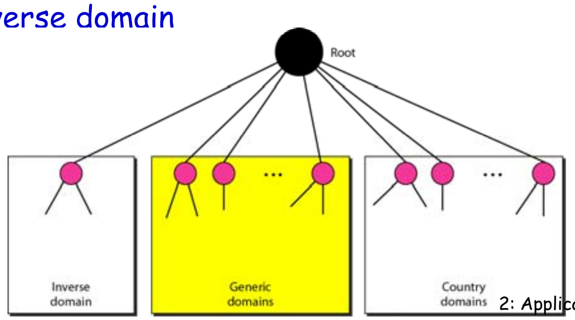
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one.
  - also called "**default name server**"
- when host makes DNS query, query is sent to its local DNS server
  - acts as proxy, forwards query into hierarchy



```
C:\Windows\system32\cmd.exe

C:\>nslookup www.yahoo.com
Server:  diamond.ce.kmitl.ac.th
Address:  161.246.4.3

Non-authoritative answer:
Name:    ds-sg-fp3.wg1.b.yahoo.com
Addresses:  2406:2000:ac:8::c:9102
          2406:2000:ac:8::c:9101
          106.10.139.246
Aliases:  www.yahoo.com
          fd-fp3.wg1.b.yahoo.com
          ds-fp3.wg1.b.yahoo.com
          ds-sg-fp3-lfb.wg1.b.yahoo.com

C:\>
```

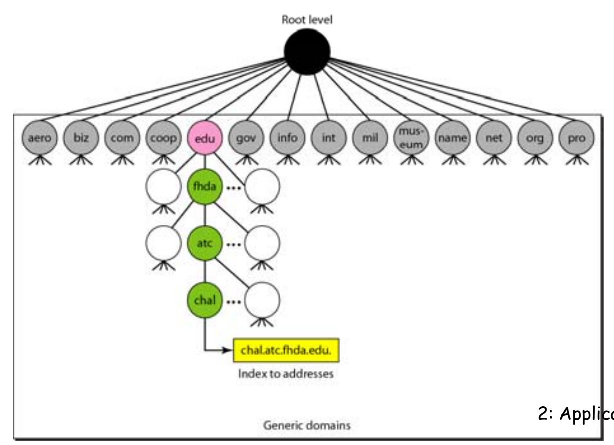

DNS in Internet

- DNS is a protocol that can be used in different platforms
- In Internet, domain name space (tree) is divided into three different sections:
  - Generic domain
  - Country domains
  - Inverse domain



Generic Domains

- Generic domains define registered hosts according to their generic behavior
- Each node in tree defines a domain, which is an index to domain name space database

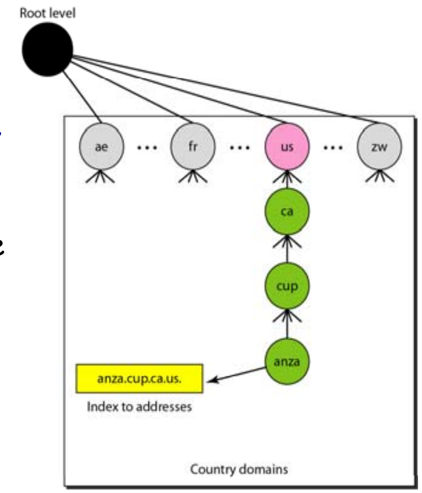


Generic domain labels

Label	Description
aero	Airlines and aerospace companies
biz	Businesses or firms (similar to "com")
com	Commercial organizations
coop	Cooperative business organizations
edu	Educational institutions
gov	Government institutions
info	Information service providers
int	International organizations
mil	Military groups
museum	Museums and other nonprofit organizations
name	Personal names (individuals)
net	Network support centers
org	Nonprofit organizations
pro	Professional individual organizations

Country Domains

- Country domains section uses two-character country abbreviations
- Seconds labels can be organization, or they can be more specific, national designations
- Ex. Anza.cup.ca.us can be translated to De Anza College in Cupertino, California, in the United State



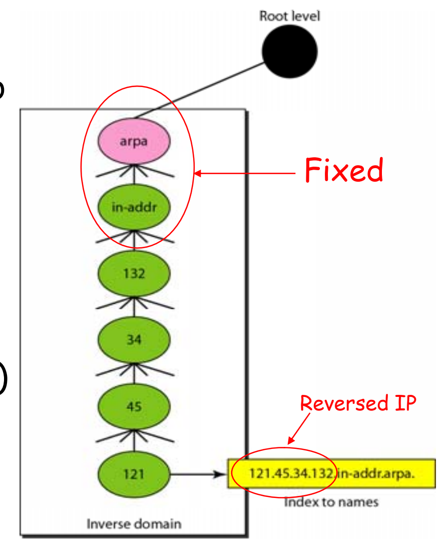
.ae – United Arab Emirates
.fr – France
.us – United States
.zw – Zimbabwe
.ch - Switzerland

Inverse Domain

- Inverse domain is used to ***map IP Address to Domain Name***
- For example, when server has received a request from client to do a task
- Although server has file that contains a list of authorized clients, only IP address of client (extracted from received IP packet) is listed
- Server asks its resolver to send a query to DNS server to map address to name to determine if client is on authorized list
- This type of query is called an **inverse or pointer (PTR) query**

Inverse Domain (cont.)

- To handle a pointer query, inverse domain is added to domain name space with the first-level node called **arpa** (for historical reasons)
- The second level is also one single node named **in-addr** (for inverse address)
- The rest of the domain defines **IP addresses**



Reversed IP.in-addr.arpa

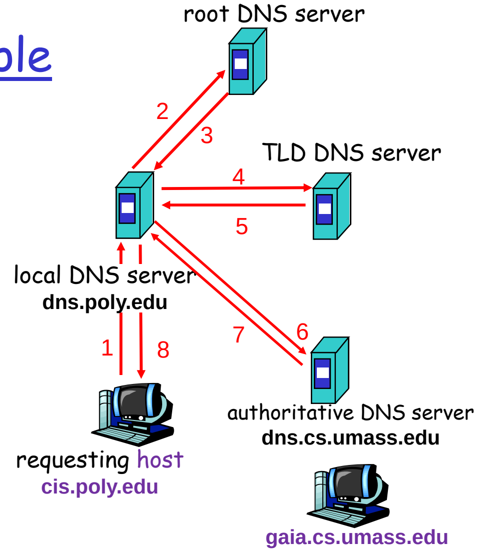
IP address = 132.34.45.121

DNS name resolution example

- Host at cis.poly.edu wants IP address for gaia.cs.umass.edu

Iterated Query:

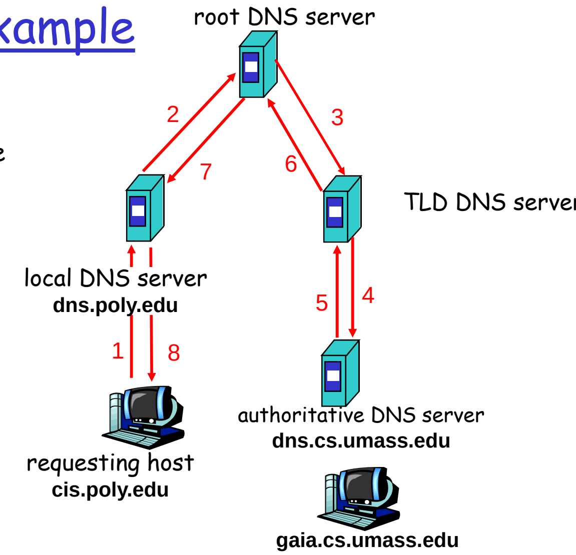
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

Recursive Query:

- puts burden of name resolution on contacted name server



DNS records

TTL : time to live of RR;
It is determines when a resource should be removed from cache

DNS: distributed db storing Resource Records (RR)

RR format: `(name, value, type, TTL)`

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

  (relay1.bar.foo.com, 145.37.93.126, A)
- Type=NS
  - **name** is domain (e.g. foo.com)
  - **value** is hostname of authoritative name server for this domain

  (foo.com, dns.bar.foo.com, NS)
- Type=CNAME
  - **name** is alias name for some "canonical" (the real) name `www.ibm.com` is really `servereast.backup2.ibm.com`
  - **value** is canonical name

  (foo.com, relay1.bar.foo.com, CNAME)
- Type=MX
  - **value** is canonical name of mail server associated with **name**

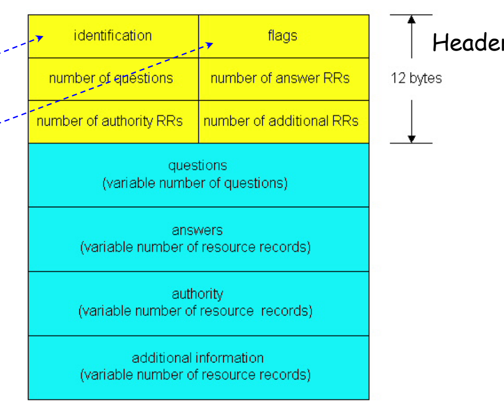
  (foo.com, mail.bar.foo.com, MX)

DNS protocol, messages

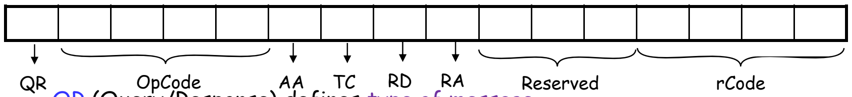
DNS protocol : *query* and *reply* messages, both with same *message format*

message header

- identification: 16 bits # for query, reply to query uses same #
- flags: (16 bits)
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



Flags Field



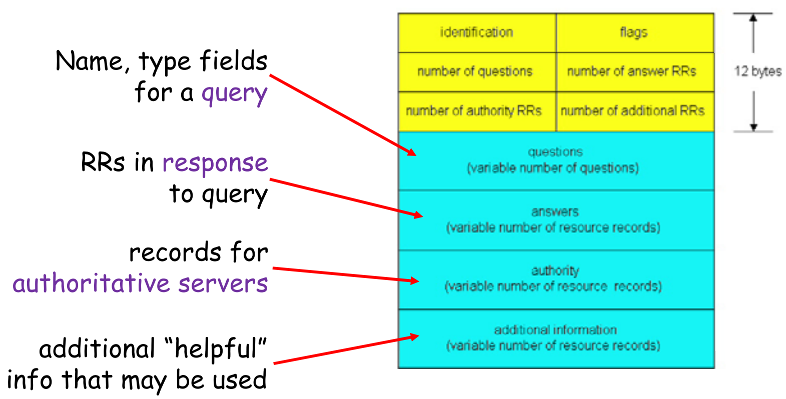
- QR (Query/Response) defines type of message
  - If it is 0, the message is a query
  - If it is 1, the message is a response
- OpCode : defines type of query or response
  0 if standard query, 1 if inverse query, 2 if a server status request
- AA (authoritative answer)
  - When it is set (value of 1) it means that name server is an authoritative server
  - It is used only in response message
- TC (truncated)
  - When it is set (value of 1), it means that response was more than 512 bytes and truncated to 512
  - It is used when DNS uses services of UDP
- RD (recursion desired)
  - When it is set (value of 1), it means client desires recursive answer
  - It is set in query message and repeated in response message
- RA (recursion available), this bit is valid in response and denotes whether recursive query support is available
- Reserved
  - 3-bit subfield is set to 000
- rCode shows status of error in response

Flags Field

Value	Meaning
0	No error
1	Format error
2	Problem at name server
3	Domain reference problem
4	Query type not supported
5	Administratively prohibited
6-15	Reserved

Value of rCode

DNS protocol, messages



Ex.
- Answer field in reply to MX query contains a RR providing canonical name of mail server
- Additional section contains a Type A record providing IP address for canonical hostname of mail server

Inserting records into DNS

- example: new startup "Network Utopia"
- register name networkuptopia.com at *DNS registrar* (e.g., Network Solutions)
  - provide names, IP addresses of authoritative name server (primary and secondary)
  - registrar inserts two RRs into com TLD server:
    hostname of authoritative name server
  - `(networkutopia.com, dns1.networkutopia.com, NS)`
  - `(dns1.networkutopia.com, 212.212.212.1, A)`
- create authoritative server
  - Type A record for www.networkuptopia.com;
  - Type MX record for networkutopia.com
- How do people get IP address of your Web site?

Encapsulation

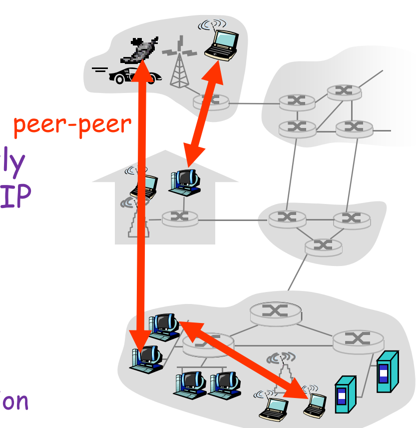
- DNS can use either UDP or TCP.
- In both cases the well-known port used by the server is port 53.
- UDP is used when the size of response message is less than 512 bytes because most UDP packages have a 512-byte packet size limit.
- If the size of response message is more than 512 bytes, a TCP connection is used.

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 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

Pure P2P architecture

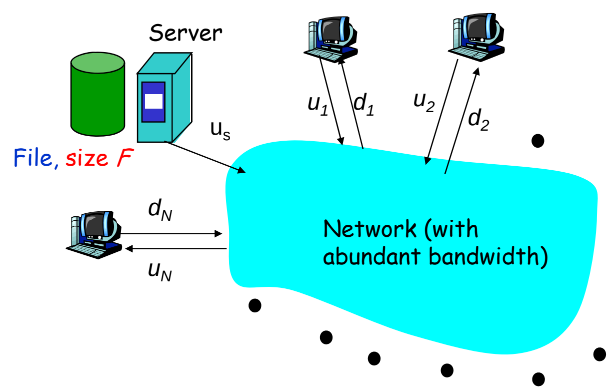
- *no* always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses



- Three topics:
  - File distribution
  - Searching for information (database distribution)
  - Case Study: Skype

File Distribution: Server-Client vs P2P

Question : How much time to distribute file from one server to *N* *peers*?



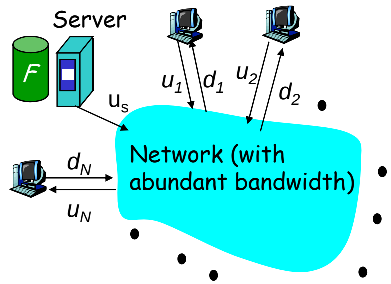
u_s: server upload bandwidth

u_i: peer i upload bandwidth (i=1,2,...,N)

d_i: peer i download bandwidth (i=1,2,...,N)

File distribution time: server-client

- server sequentially sends N copies:
  - NF/u_s time
- client i takes F/d_i time to download



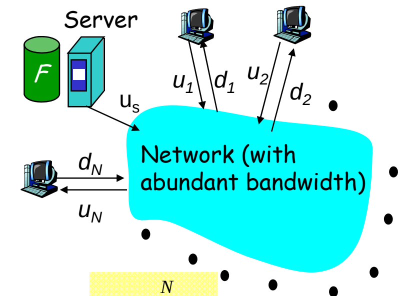
Time to distribute F to N clients using client/server approach

$$D_{CS} = \max\left\{\frac{NF}{u_s}, \frac{F}{d_{\min}}\right\}$$

increases linearly in N (for large N)

File distribution time: Peer to Peer (P2P)

- server must send one copy:
  - F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \sum_{i=1}^{N} u_i$

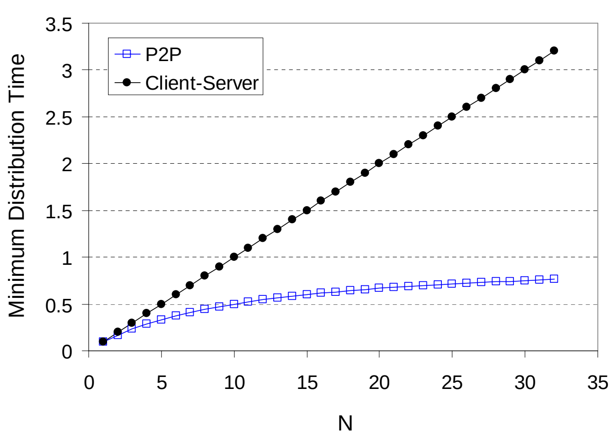


$$D_{P2P} = \max\left\{\frac{F}{u_s}, \frac{F}{d_{\min}}, \frac{NF}{u_s + \sum_{i=1}^{N} u_i}\right\}$$

Total upload capacity of system as a whole = Upload rate of server plus upload rate of each of individual peer

Server-client vs. P2P: example

Client upload rate = u, F/u = 1 hour, u_s = 10u, $d_{min} \geq u_s$



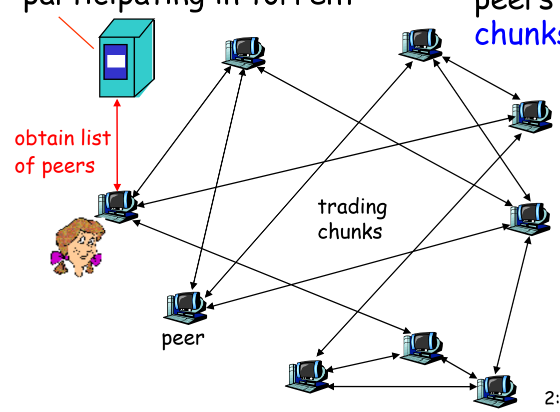
File distribution: **BitTorrent**

- P2P file distribution

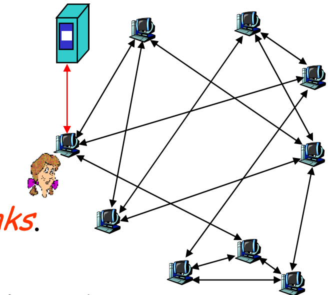
Collection of all peers participating in distribution of particular file

tracker: tracks peers participating in torrent

torrent: group of peers exchanging chunks of a file



BitTorrent (1)



- file divided into 256KBytes *chunks*.
- peer joining torrent:
  - has no chunks, but will accumulate them over time
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")
- while downloading, peer uploads chunks to other peers.
- peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain

เห็นแก่ผู้อื่น/เสียสละ

BitTorrent (2)

- Chunk ไหนควร request ก่อน
- Peer เพื่อนบ้านไหนที่ควรส่ง request chunk

Pulling Chunks

- at any given time, different peers have different subsets of file chunks
- periodically, peer (Alice) asks each neighbor for list of chunks that they have
- Alice sends requests for her missing chunks
  - **rarest first**

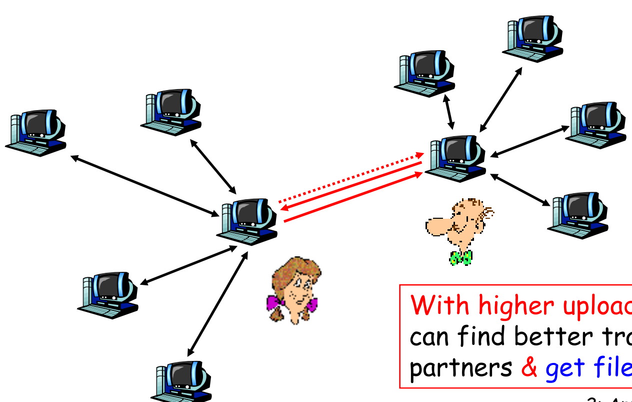
Chunks that have fewest repeated copies among neighbors and then request those rarest chunks first

Sending Chunks: tit-for-tat

- Alice sends chunks to four neighbors currently sending her chunks *at highest rate*
  - re-evaluate top 4 every 10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - newly chosen peer may join top 4
  - "optimistically unchoke"

BitTorrent: Tit-for-tat (การตอบแทน)

(1) Alice "optimistically unchokes" Bob การแลกเปลี่ยนกัน
(2) Alice becomes one of Bob's top-four providers; Bob reciprocates
(3) Bob becomes one of Alice's top-four providers



With higher upload rate, can find better trading partners & get file faster!

Distributed Hash Table (DHT)

(Indexing and Searching technique)

- DHT = distributed P2P database
- Database has (key, value) pairs;
  - key: social security (ss) number; value: human name
    - Ex. Key-value pair is (156-45-7081, Johnny Wu)
  - key: content type; value: IP address
    - Ex. Content type : Name of movies, albums, and software
    - Ex. Key-value pair is (Led Zeppelin IV, 203.17.123.38))
    - Ex. Key-value pair is (สุภาพบุรุษจุฑาเทพ, 200.15.100.22)
- Peers query Database with key
  - DB returns values that match the key
- Peers can also insert (key, value) pairs into our database

DHT Identifiers

An approach to designing P2P database

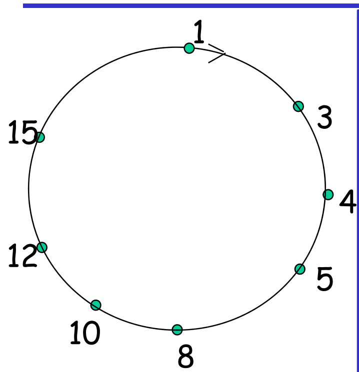
- Assign integer identifier to each peer in range $[0,2^n-1]$.
  - Each identifier can be represented by n bits
- Require each key to be an integer in same range using hash function.
- To get integer keys, hash original key.
  - eg, key = h("Led Zeppelin IV")
  - This is why they call it a distributed "hash" table

How to assign keys to peers?

- Central issue:
  - Assigning (key, value) pairs to peers.
- Rule: assign key to the peer that has the closest ID.
- Convention in lecture: defining closest peer as the immediate successor of the key.
- Ex: n=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

If key is larger than all peer identifiers, we use modulo-2^n convention storing (key-value) pair in peer with smallest ID

Circular DHT (1)



Building a database can be with client-server architecture
- all (key,value) pairs are stored in one central server such as Napster.

If peer keep track of all peers in system (peer IDs and IP address), peer could locally determine closest peer.

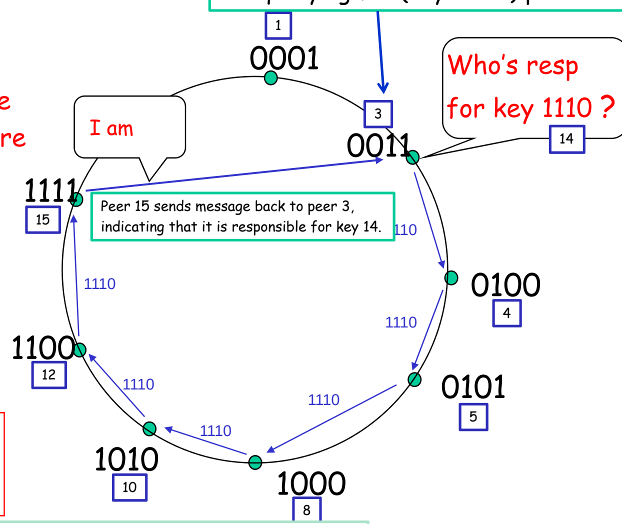
But such approach requires each peer to keep track of all other peers in DHT- which is completely impractical for large-scale system with millions of peers

- Each peer *only* aware of immediate successor and predecessor.
- "Overlay network"
  - peers form abstract logical network
  - Links are not physical links, but are simply virtual liaisons between pairs of peers

Circle DHT (2)

Peer 3 wants to determine which peer in DHT is responsible for key 14
-either for inserting or
querying for (key-value) pair

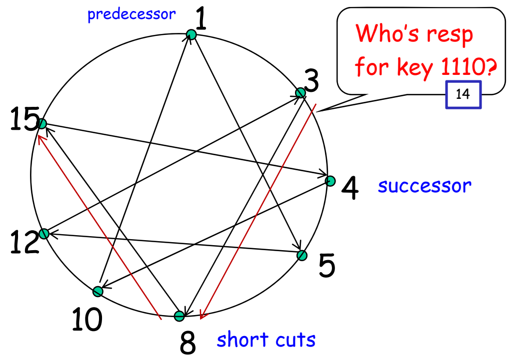
O(N) messages on avg to resolve query, when there are N peers



Define closest as closest successor

In circular DHT, N/2 msgs are sent on average for query

Circular DHT with Shortcuts



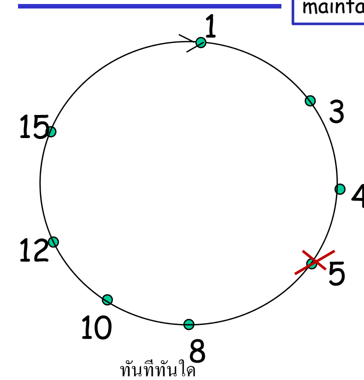
When peer receives message that is querying for key, it forwards message to neighbor (successor neighbor or one of short-cut neighbors which is the closet to key)

- Each peer keeps track of IP addresses of predecessor, successor, short cuts (used to expedite routing of query message).
- Reduced from 6 to 2 messages.
- Possible to design shortcuts so O(log N) neighbors, O(log N) messages in query

Short cuts can significantly reduce number of message used to process a query

Peer Churn

In P2P system, peer can come or go without warning. When designing DHT, we must be concerned about maintaining DHT overlay in the presence of such peer churn

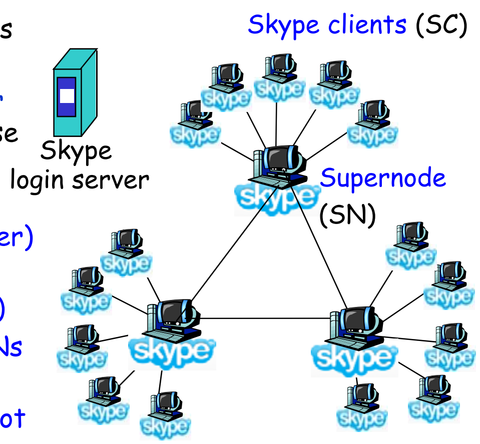


- To handle peer churn, require each peer to know the IP address of its two successors.
- Each peer periodically pings its two successors to see if they are still alive.

- 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 (peer 10).
- What if peer 13 wants to join?

P2P Case study: Skype

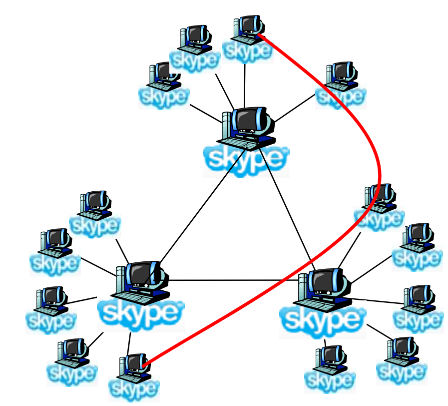
- inherently P2P: pairs of users communicate.
- proprietary application-layer protocol (inferred via reverse engineering)
- hierarchical overlay with Supernodes (SNs) (super peer)
- Index maps usernames to IP addresses (and port number)
- Index is distributed over SNs
- Because Skype protocol is proprietary, it is currently not clear how index mappings are organized across supernodes



Peers as relays

Network Address Translator (NAT)

- 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 2: Application layer

Socket Programming

Goal: learn how to build client/server application that communicate using sockets

Socket API (Application Programming Inteface)

Berkeley Software Distribution

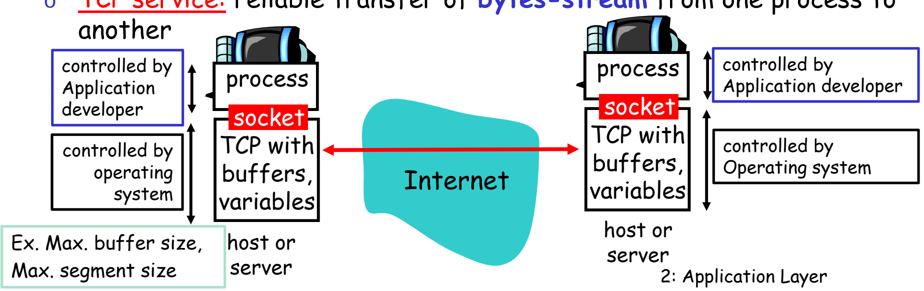
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by applications
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram : User Datagram Protocol (UDP)
  - reliable, byte stream-oriented : Transmission Control Protocol (TCP)

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

Socket-programming using TCP

- Network Apps. consist of client program and server program.
- When these two programs are executed, client process and server process are created.
- Processes running on different machines communicate with each other by sending messages into socket.
- Main task of developer's is to write code for both client and server program.
- Socket: a door between application process and end-end-transport protocol (UDP or TCP)
- TCP service: reliable transfer of **bytes-stream** from one process to another



Socket programming *with TCP*

In order for server to be able to react to client's initial contact, server has to be ready.

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**) | Client

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

When client creates socket: client TCP establishes connection to server TCP.
Three-way handshake takes place at transport layer which is completely transparent to client and server program

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

TCP
connection setup

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

IP address of server process

port number of server process

send request using
clientSocket

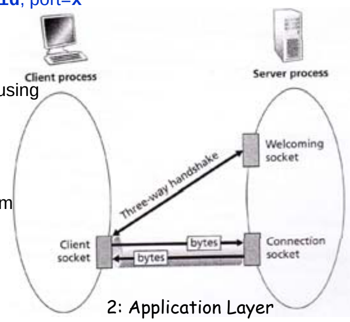
read request from
connectionSocket

write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

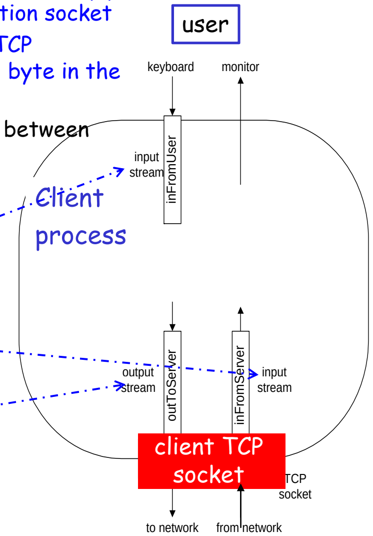


2: Application Layer

Stream jargon

- From app's perspective, TCP connection is direct virtual pipe between client's socket and server's communication socket
- Client process can send bytes into socket, and TCP guarantees that sever process will receive each byte in the order sent
- TCP thus provides reliable byte-stream service between client and server process

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, e.g., keyboard or socket.
- An output stream is attached to an output source, e.g., monitor or socket.

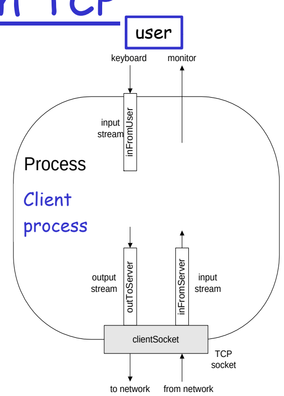


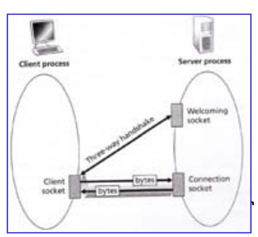
2: Application Layer

Socket programming with TCP

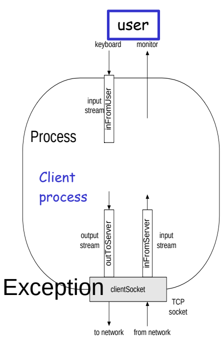
Example client-server app:

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)




```
import java.io.*;
import java.net.*;
class TCPClient {

    public static void main(String argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;

        BufferedReader inFromUser =
          new BufferedReader(new InputStreamReader(System.in));

        Socket clientSocket = new Socket("hostname", 6789);

        DataOutputStream outToServer =
          new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream → BufferedReader inFromUser

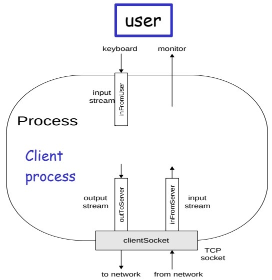
Create client socket, connect to server → Socket clientSocket

Host name of server — Port number

Create output stream attached to socket → DataOutputStream outToServer

2: Application Layer

Example: Java client (TCP) cont.



Create input stream attached to socket

```
BufferedReader inFromServer =
  new BufferedReader(new
  InputStreamReader(clientSocket.getInputStream()));
```

```
sentence = inFromUser.readLine();
```

Send line to server

```
outToServer.writeBytes(sentence + '\n');
```

Read line from server

```
modifiedSentence = inFromServer.readLine();

System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();

  }
}
```


Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
  {
    String clientSentence;
    String capitalizedSentence;
```

Create welcoming socket at port 6789

```
    ServerSocket welcomeSocket = new ServerSocket(6789);

    while(true) {
```

Wait, on welcoming socket for contact by client

```
        Socket connectionSocket = welcomeSocket.accept();
```

Create input stream, attached to socket

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


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

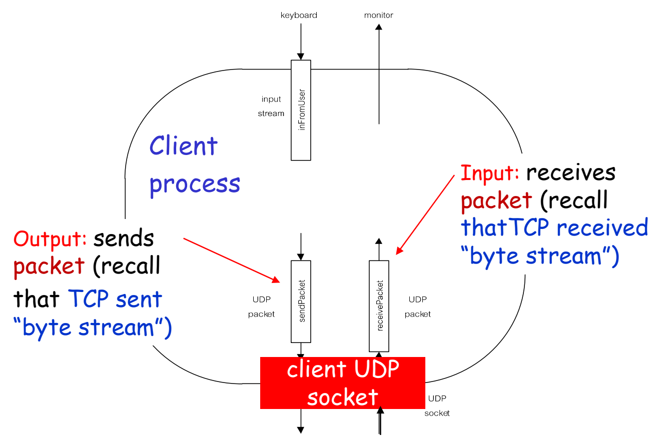
create socket, clientSocket = DatagramSocket()

Create datagram with server IP and port=x; send datagram via clientSocket

read datagram from clientSocket

close clientSocket

Example: Java client (UDP)



Example: Java client (UDP)

```
import java.io.*;
import java.net.*;  ประกอบด้วย DatagramSocket, InetAddress

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();
```

Create input stream

Input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

Create datagram with data-to-send, length, IP addr, port

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

Send datagram to server

```
clientSocket.send(sendPacket);

DatagramPacket receivePacket =
  new DatagramPacket(receiveData, receiveData.length);
```

Read datagram from server

```
clientSocket.receive(receivePacket);

String modifiedSentence =
   new String(receivePacket.getData());

System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
}
}
```


Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {
```

Create datagram socket at port 9876

```
   DatagramSocket serverSocket = new DatagramSocket(9876);

   byte[] receiveData = new byte[1024];
   byte[] sendData  = new byte[1024];

   while(true)
     {
```

Create space for received datagram

```
      DatagramPacket receivePacket =
        new DatagramPacket(receiveData, receiveData.length);
```

Receive datagram

```
      serverSocket.receive(receivePacket);
```


Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());
```

Get IP addr port #, of sender

```
InetAddress IPAddress = receivePacket.getAddress();

int port = receivePacket.getPort();

      String capitalizedSentence = sentence.toUpperCase();

sendData = capitalizedSentence.getBytes();
```

Create datagram to send to client

```
DatagramPacket sendPacket =
  new DatagramPacket(sendData, sendData.length, IPAddress,
                     port);
```

Write out datagram to socket

```
serverSocket.send(sendPacket);
  }
 }
}
```

End of while loop, loop back and wait for another datagram