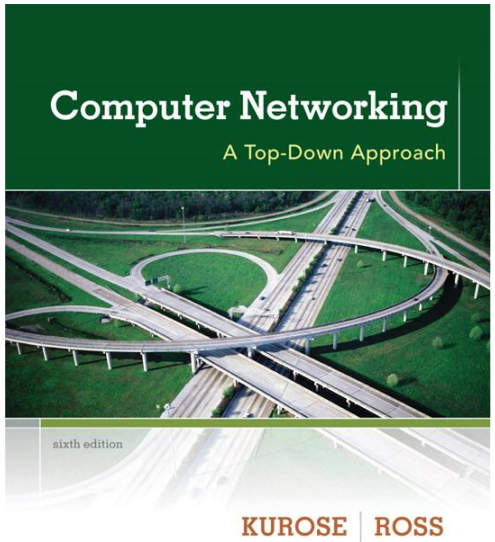
บทที่ 2: Application Layer



การใช้สไลด์ :

เนื้อหาในสไลด์เหล่านี้ถูกแปลมาจากสไลด์ต้นฉบับประกอบหนังสือของผู้แต่งชื่อ Kurose และ Ross

ผู้แปลอนุญาตให้ทุกท่านสามารถใช้สไลด์ทั้งหมดได้ ดังนั้นท่านสามารถดูภาพเคลื่อนไหว สามารถเพิ่ม ,แก้ไข และ ลบสไลด์ (นับรวมข้อความนี้) และเนื้อหาของสไลด์เพื่อให้เหมาะกับความต้องการของท่าน

สำหรับการแลกเปลี่ยน เราต้องการสิ่งต่อไปนี้เท่านั้น :

- ถ้าท่านใช้สไลด์เหล่านี้ (เป็นตัวอย่าง, ในห้องเรียน) อย่าลืมกล่าวถึงที่มาของสไลด์ (หลังจากนี้ เราต้องการให้ทุกคนอุดหนุนและใช้หนังสือของผู้แต่งด้านข้าง)
- ถ้าคุณโพสต์สไลด์ใด ๆ ในเวป, อย่าลืมกล่าวถึงว่า คุณแก้ไขจากสไลด์ต้นฉบับของเรา และ ระบุถึงลิขสิทธิ์ของเราด้วย

ขอขอบคุณและขอให้สนุก!

ณัฐนนท์ ลีลาตระกูล ผู้เรียบเรียง

© สงวนลิขสิทธิ์ 2013
เนื้อหาทั้งหมดเป็นลิขสิทธิ์ของคณะวิทยาการสารสนเทศ

Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

Chapter 2: application layer

เป้าหมาย:

- แนวคิด, การ implement โปรโตคอลแอพพลิเคชั่นด้านเครือข่าย
  - model การบริการชั้น transport
  - ชุดแนวคิด (หรือ paradigm) client-server
  - ชุดแนวคิด peer-to-peer
- เรียนรู้เกี่ยวกับโพรโตคอลโดยการศึกษาแอพพลิเคชันโพรโตคอลที่เป็นที่นิยม
  - HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- สร้างแอพพลิเคชันด้านระบบเครือข่าย
  - socket API

ตัวอย่างของแอพพลิเคชันระบบเครือข่าย

- e-mail
- web
- ส่งข้อความ (text messaging)
- login ทางไกล
- แชร์ไฟล์ผ่าน P2P
- multi-user network games
- ส่ง video ที่เก็บไว้อย่างต่อเนื่อง (streaming stored video) เช่น YouTube, Hulu, Netflix
- voice over IP (e.g., Skype)
- ส่ง video การประชุมแบบ real-time
- social networking
- บริการค้นหาข้อมูล
- ...
- ...

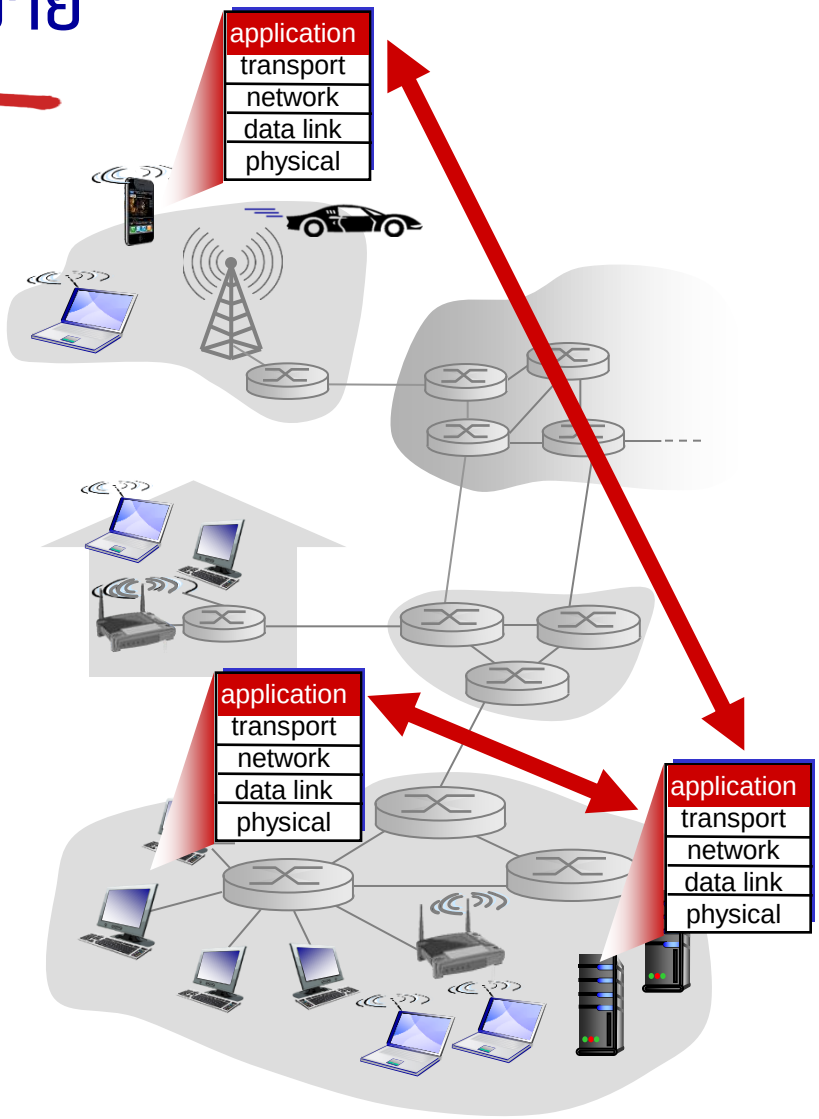
การสร้างแอพพลิเคชันระบบเครือข่าย

เขียนโปรแกรมที่:

- ทำงานอยู่บนเครื่องปลายทางที่ต่างๆกัน (ไม่ใช่ที่แกนของเครือข่าย)
- สื่อสารกันผ่านเครือข่าย
- เช่น ซอฟแวร์ของ web server จะสื่อสารกับตัว web browser

ไม่จำเป็นต้องเขียนซอฟท์แวร์ที่ทำงานที่แกนของระบบเครือข่าย

- อุปกรณ์ที่แกนระบบเครือข่ายไม่ต้องทำงานแอพพลิเคชันของผู้ใช้
- แอพพลิเคชันจะอยู่บนเครื่องปลายทางเท่านั้น ทำให้การพัฒนาแอพพลิเคชันเป็นไปอย่างรวดเร็ว

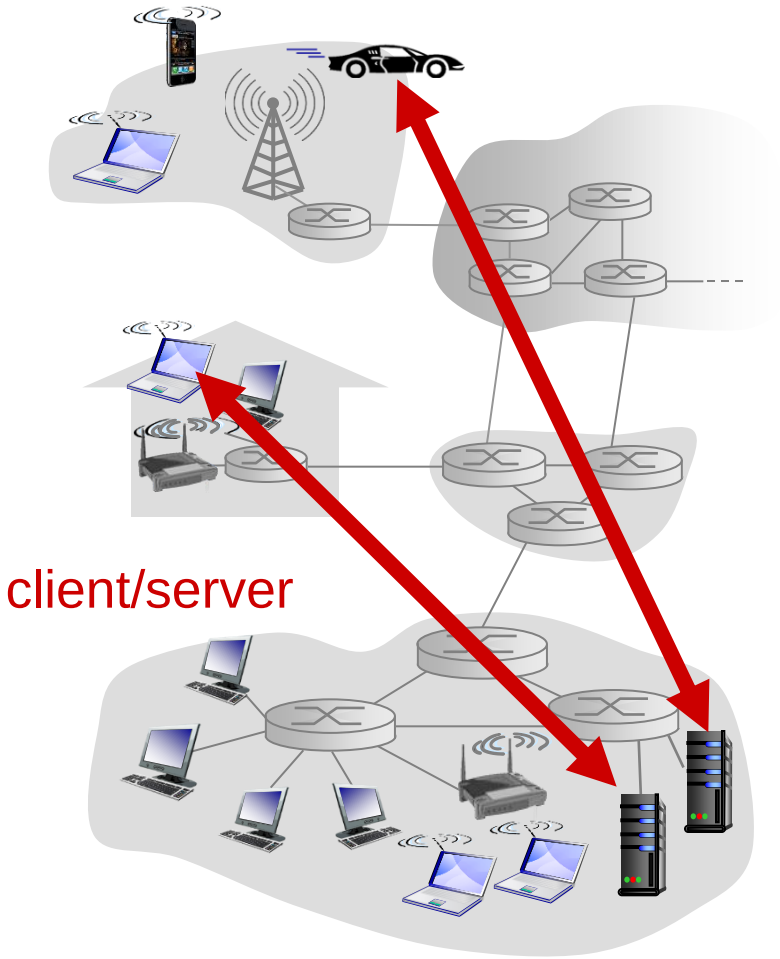


สถาปัตยกรรมของแอพพลิเคชั่น

โครงสร้างที่เป็นไปได้ของแอพพลิเคชั่น:

- client-server (ลูกข่าย – แม่ข่าย)
- peer-to-peer (P2P, เพื่อน ถึง เพื่อน)

โครงสร้างของระบบ Client-Server



server:

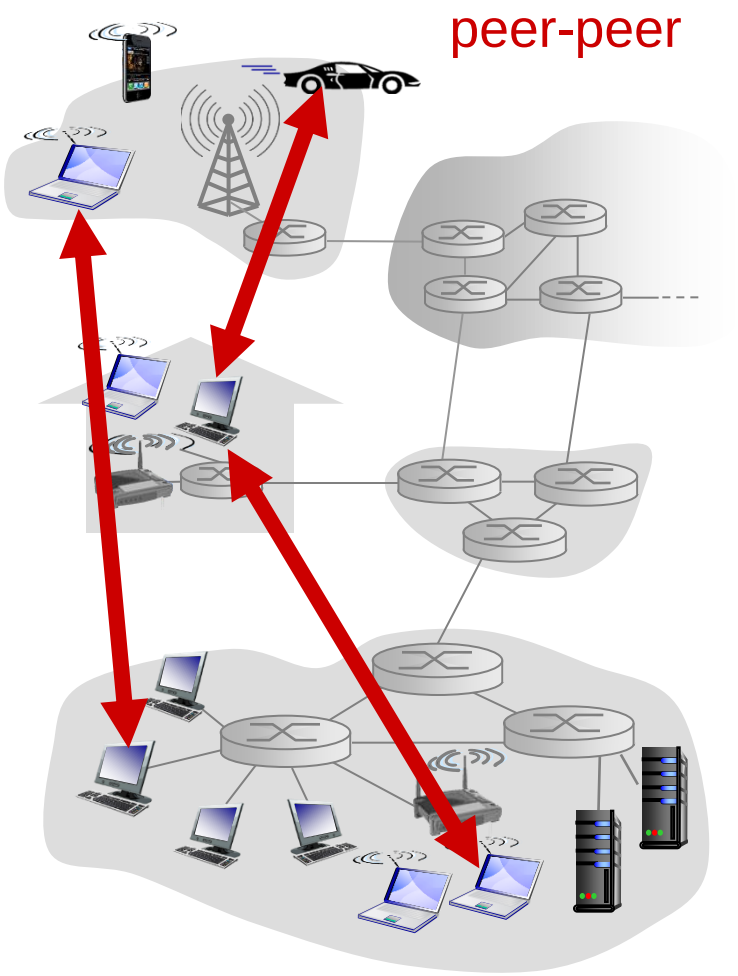
- เป็นเครื่องที่ทำงานตลอดเวลา
- มีไอพีแอดเดรสค่อนข้างถาวร
- อยู่ใน Data Center เพื่อรองรับการบริการผู้ใช้ที่เพิ่มมากขึ้น

clients:

- มีการติดต่อสื่อสารกับผู้ให้บริการ
- มีการเชื่อมต่อกับผู้ให้บริการเป็นระยะ ๆ ไม่สม่ำเสมอ
- อาจจะมีไอพีแอดเดรสที่ไม่แน่นอน
- ไม่ได้ติดต่อกับผู้รับคนอื่นโดยตรง

โครงสร้างของระบบ P2P

- ไม่มีเครื่องแม่ข่ายที่ให้บริการตลอดเวลา
- สามารถเชื่อมต่อกับลูกข่ายกันเองได้
- peer A (ซึ่งส่งคำขอไปยัง peer B) อาจเป็นผู้ให้บริการกับ peer C เองก็ได้
  - *Self-scalability: มีความสามารถรองรับจำนวนผู้ใช้ที่เพิ่มมากขึ้นได้โดยตัวโครงสร้างเอง เพราะ peer ตัวใหม่ที่เข้ามาในระบบ จะทำให้ความสามารถในการบริการเพิ่มมากขึ้น เช่นเดียวกับความต้องการในการใช้บริการที่เพิ่มขึ้น*
- การเชื่อมต่อระหว่าง peers ไม่สม่ำเสมอและไอพีแอดเดรสจะเปลี่ยนไปเรื่อยๆ
  - ทำให้การจัดการซ้ำซ้อนยุ่งยาก



Process ที่ติดต่อสื่อสารกัน

process: โปรแกรมที่กำลังทำงานอยู่บนเครื่อง

- ภายใต้เครื่องเดียวกัน process สองprocess สามารถสื่อสารกันได้ด้วยช่องทางสื่อสารระหว่าง process หรือ inter-process communication (ระบบปฏิบัติการจัดเตรียมให้)
- processes ที่อยู่คนละเครื่องสามารถติดต่อกันได้โดยการแลกเปลี่ยนข้อความกันผ่านเครือข่าย

clients, servers

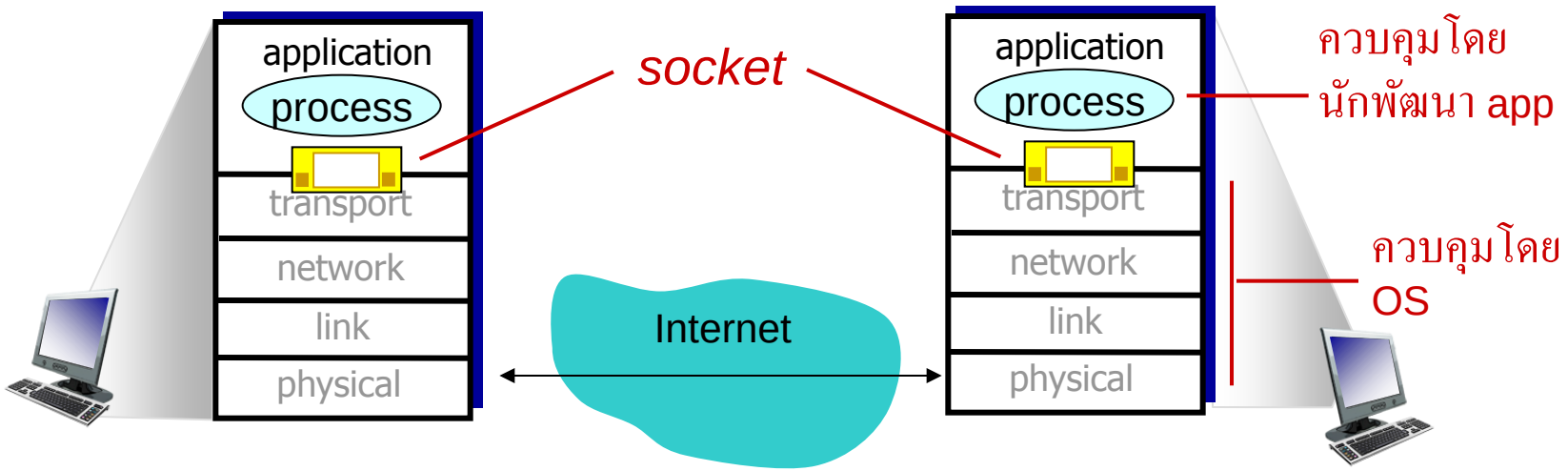
client process: ตัวโปรแกรมลูกข่ายที่เริ่มติดต่อสื่อสารขอรับบริการ

server process: โปรแกรมที่เครื่องแม่ข่ายรอรับการขอบริการจากลูกข่าย

- ถ้าเป็นแอพพลิเคชันในสถาปัตยกรรมแบบ P2P ก็จะมีทั้ง client processes และ server processes

Sockets

- process จะส่ง/รับข้อมูลผ่านทาง socket
- socket เปรียบเสมือนประตู
  - process ตัวส่งจะส่งข้อความผ่านทางประตู
  - process ตัวส่งต้องพึ่งพาระบบการขนส่งไปยังอีกด้านของประตู โดยการส่งข้อความผ่านทาง socket ไปยัง process ที่กำลังรอรับอยู่



การกำหนดที่อยู่ให้กับ processes

- process จะรับข้อความได้จะต้องมีเลขระบุ process
- แต่ละเครื่องจะมีเลขไอพีแอดเดรส (แบบ 32 บิต) ไม่ซ้ำใคร (ใช้ระบุเครื่องได้ แต่ ยังระบุ process ไม่ได้)

- *Q:* จำนวนไอพีแอดเดรสทั้งหมดเพียงพอต่อ process ในแต่ละเครื่องไหม ?
  - *A:* ไม่, *มี process จำนวนมากทำงานอยู่บนเครื่องเดียวกัน*

- *เลขระบุ process* ใช้ทั้งไอพีแอดเดรสและ port numbers ที่เกี่ยวข้องกับ process นั้น ๆ บนเครื่อง
- ตัวอย่าง port numbers:
  - HTTP server: 80
  - mail server: 25
- ส่งข้อความ HTTP ไปยัง gaia.cs.umass.edu web server ที่:
  - IP address: 128.119.245.12
  - port number: 80
- รายละเอียดจะตามมา...

protocol ในชั้น App จะนิยาม

- ชนิดของข้อความที่แลกเปลี่ยนกัน
  - เช่น ข้อความที่ส่งไปร้องขอ, ข้อความที่ตอบรับ
- รูปแบบของข้อความ:
  - จะบอกถึงสิ่งที่อยู่ในข้อความ และบอกว่าสิ่งนั้นอยู่ส่วนไหนของข้อความที่ส่งมา
- ความหมายของข้อความ
  - ความหมายของข้อมูลในข้อความที่ส่งมาในแต่ละส่วน (field)
- กฎที่กำหนดว่าเมื่อไรจะส่งข้อความหรือเมื่อไรจะตอบรับข้อความ

โพรโตคอลสำหรับสาธารณะ (โพรโตคอลเปิด):

- ถูกนิยามใน RFCs
- ทำให้เครื่อง (จากต่างผู้ผลิต) ทำงานร่วมกันได้
- เช่น HTTP, SMTP

โพรโตคอลที่มีลิขสิทธิ์:

- เช่น Skype

แอพพลิเคชันต้องการบริการอะไรจากชั้น Transport

ความสมบูรณ์ของข้อมูล

- บางแอพพลิเคชันต้องการรับส่งข้อมูลที่น่าเชื่อถือ 100% เช่น การส่งไฟล์ การติดต่อกับเว็บ
- มีบางแอพพลิเคชันที่สามารถรับส่งข้อมูลผิดพลาดได้บ้าง เช่น ส่งข้อความเสียง

throughput

- บางแอพพลิเคชันต้องการ throughput ขั้นต่ำเช่น ส่ง multi-media ทั้งภาพและเสียง
- ในบางแอพพลิเคชั่นก็ทำงานในแบบต้องการ throughput เท่าไรก็ได้ เรียกว่า elastic apps

เวลาในการส่ง

- บางแอพพลิเคชันต้องการความเร็วในการรับส่งข้อมูล เช่น คุยโทรศัพท์ผ่านอินเตอร์เน็ท เกมที่ผู้ใช้มีปฏิสัมพันธ์กับเครื่องแม่ข่ายหรือผู้เล่นเกมรายอื่นผ่านเครือข่าย

ความปลอดภัย

- เข้ารหัส, ความสมบูรณ์ ความเป็นปกติของข้อมูล, ...

บริการในชั้น Transport ที่ apps ทั่วไปต้องการ

application	data loss	throughput ขั้นต่ำ	sensitive ต่อความล่าช้า
การรับส่ง file	no loss	elastic	no
e-mail	no loss	elastic	no
เอกสาร Web	no loss	elastic	no
real-time audio/video	ทนต่อ loss	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100’s msec
audio/video ที่เก็บไว้ก่อนได้	ทนต่อ loss	same as above	yes, 2-3 secs
interactive games	ทนต่อ loss	2-3 kbps ขึ้นไป	yes, 100’s msec
text messaging	no loss	elastic	yes and no

บริการของ protocol ชั้น transport ใน Internet

บริการของ TCP:

- *การส่งข้อมูลที่น่าเชื่อได้* ระหว่าง process ผู้ส่งและ process ผู้รับ
- *การควบคุมการไหลของข้อมูล:* ผู้ส่งไม่ส่งข้อความไปท่วนผู้รับ
- *ควบคุมความคับคั่ง:* มีการควบคุมการส่งข้อมูลในขณะที่ระบบเครือข่ายคับคั่ง
- *สิ่งไม่ได้ให้:* ไม่รับประกันเรื่องเวลา แบนด์วิดท์ขั้นต่ำ ความปลอดภัยของข้อมูล
- *connection-oriented:* ต้อง setup การเชื่อมต่อก่อนการให้บริการ

บริการของ UDP:

- *ข้อมูลที่ถูกส่งเป็นการส่งแบบไม่น่าเชื่อถือ*ระหว่าง process ผู้ส่งและ process ผู้รับ
- *สิ่งที่ไม่ได้ให้บริการ: ความน่าเชื่อถือ*, การควบคุมการไหลของข้อมูล, การควบคุมความคับคั่ง, เวลาที่ใช้ในการส่ง, การรับประกันปริมาณข้อมูลmที่ส่งได้, ความปลอดภัย, หรือ การ setup การเชื่อมต่อ

Q: ทำไมถึงยังต้องมีบริการ UDP อยู่อีก?

ตัวอย่าง app ที่ใช้ใน Internet และ protocol ในชั้น transport ของมัน

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 (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

ความปลอดภัย TCP

TCP & UDP

- ไม่มีการเข้ารหัส
- ข้อมูลพาสเวิร์ดที่ถูกส่งผ่านเครือข่ายสามารถถูกดูได้เลย

SSL

- ให้บริการการเชื่อมต่อ TCP แบบเข้ารหัส
- มีความสมบูรณ์ของข้อมูล
- มีการระบุตัวตนที่ปลายทาง

SSL อยู่ในชั้น App

- แอพพลิเคชันใช้ SSL libraries ซึ่งติดต่อกับ TCP

SSL socket API

- ข้อมูลพาสเวิร์ดที่ส่งออกไปจะถูกเข้ารหัสไว้
- ดูต่อบทที่ 7

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

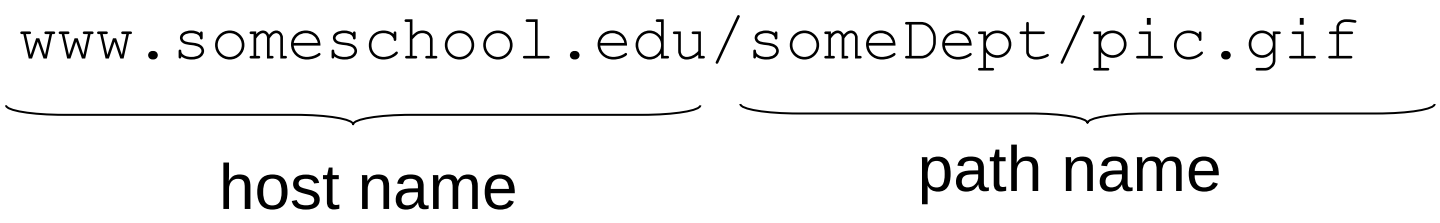
2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

Web and HTTP

ก่อนอื่นเลย, มาทบทวนกัน...

- web page ประกอบไปด้วย *file HTML หลัก* ซึ่งรวมหลายๆออบเจ็ค
- ออบเจ็คอาจจะเป็น HTML file, JPEG image, Java applet, audio file,...
- ที่อยู่ของแต่ละออบเจ็คถูกกำหนดโดย *URL* เช่น



ภาพรวมของ HTTP

HTTP: hypertext transfer protocol

- เป็นโพรโตคอลในชั้นแอพพลิเคชันที่ใช้กับเว็บ
- client/server model
  - *client:* จะมีบราวเซอร์ ทำหน้าที่ร้องขอและรับข้อมูล (โดยใช้ HTTP Protocol) จากเซิฟเวอร์มาแสดงผล
  - *server:* รอรับการร้องขอแล้วส่งข้อมูล (โดยใช้ HTTP Protocol) ที่ถูกร้องขอกลับไป

ภาพรวมของ HTTP (ต่อ)

การใช้ TCP:

- client (เครื่องลูกข่าย) เริ่มการติดต่อ (สร้าง socket) ไปยังพอร์ท 80 ของเซิฟเวอร์
- server (เครื่องแม่ข่าย) รับการเชื่อมต่อ TCP จากเครื่องลูกข่าย
- ข้อความ HTTP (application-layer protocol messages) จะถูกแลกเปลี่ยนกันระหว่างตัวบราวเซอร์กับตัวเว็บเซิฟเวอร์
- ปิดการเชื่อมต่อ TCP

HTTP ไม่มีการเก็บสถานะของผู้ใช้

- เซิฟเวอร์ไม่มีการเก็บข้อมูลการขอใช้บริการของลูกข่ายที่ผ่านมา

เกร็ดความรู้

โปรโตคอลแบบที่ต้องเก็บสถานะจะซับซ้อน

- สถานะเก่าๆของผู้ใช้จะต้องถูกเก็บไว้
- ถ้าเซิฟเวอร์หรือเครื่องผู้ใช้บริการเกิดดับไปกระทันหัน, พอเปิดเครื่องขั้นใหม่ สถานะของเครื่องผู้ใช้กับเซิฟเวอร์ก็จะมีสถานะไม่เหมือนกัน และ การทำให้เหมือนกันใหม่เป็นเรื่องยาก

การเชื่อมต่อของ HTTP

non-persistent HTTP
(การเชื่อมต่อหลายครั้ง)

- รับ/ส่งหนึ่ง object ต่อหนึ่งการเชื่อมต่อ
  - พบรับ/ส่งเสร็จ การเชื่อมต่อจะถูกปิด
- ดาวโหลดหลายๆออบเจ็คก็ต้องใช้การเชื่อมต่อหลายครั้ง

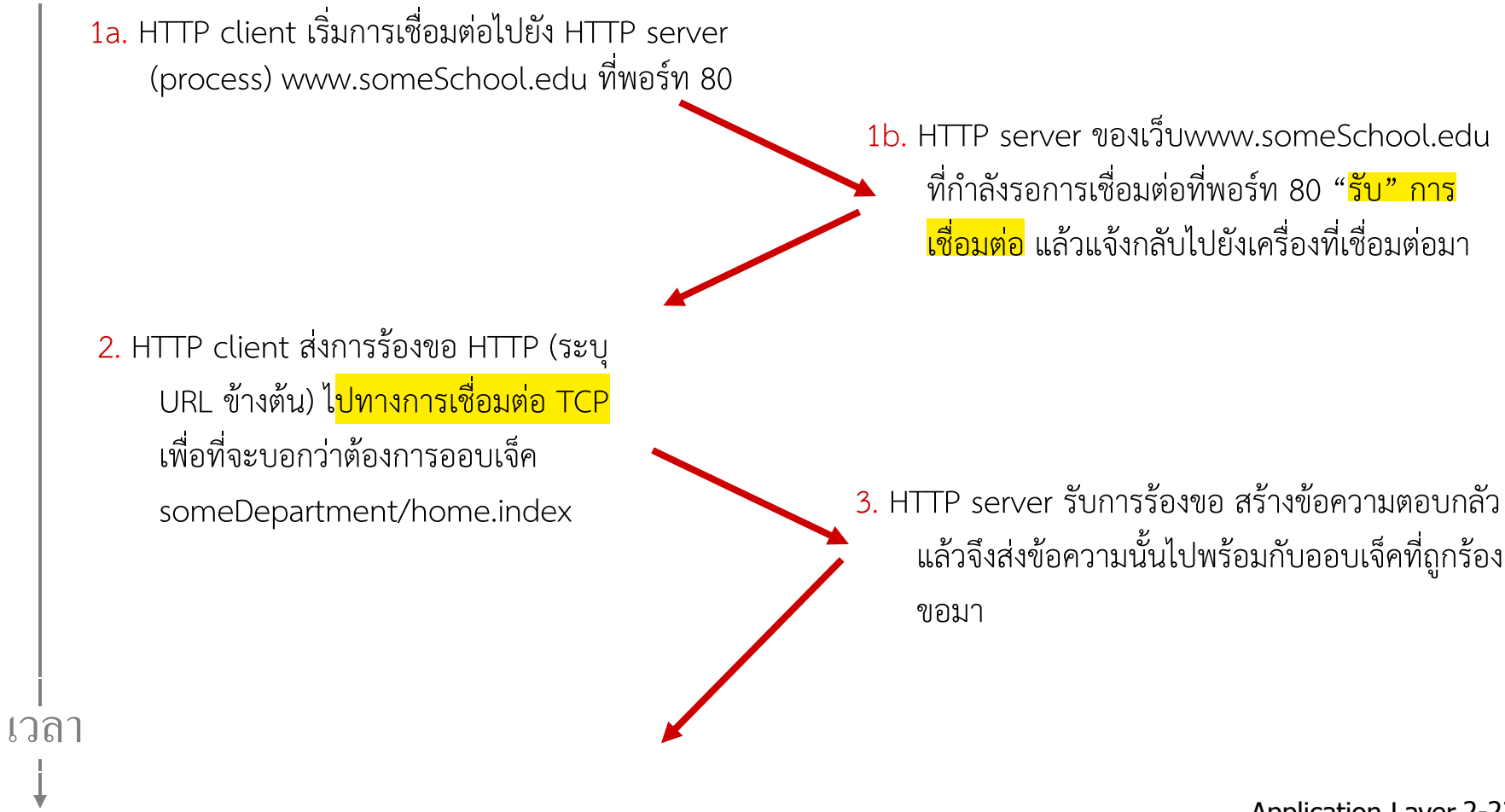
persistent HTTP
(การเชื่อมแบบคงอยู่)

- สามารถดาวโหลดหลายๆออบเจ็คด้วยการเชื่อมต่อ 1 ครั้ง

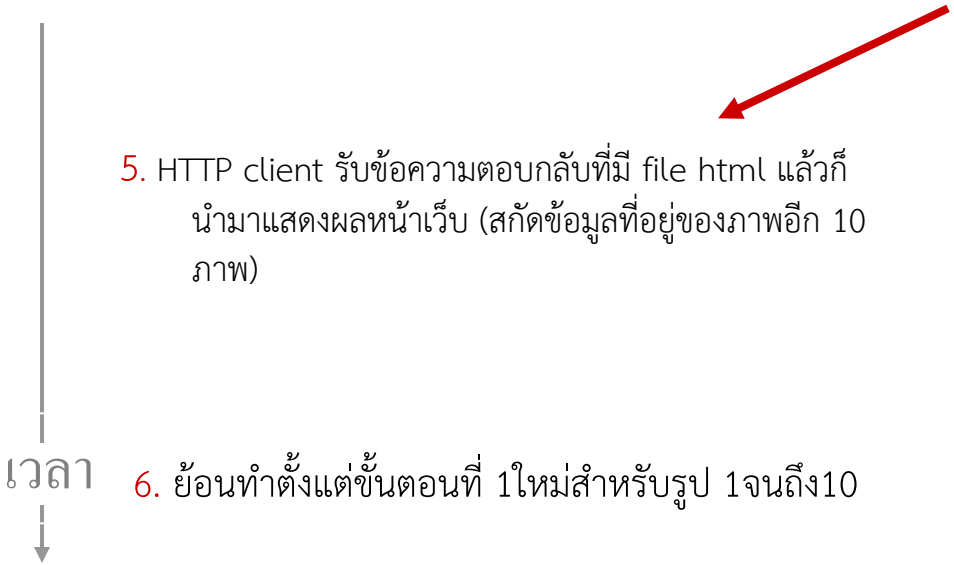
Non-persistent HTTP

สมมติว่าใส่ URL นี้ ซึ่งอ้างอิงไปยังรูปภาพ 10 รูป :

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



Non-persistent HTTP (cont.)



4. HTTP server ปิดการเชื่อมต่อ

5. HTTP client รับข้อความตอบกลับที่มี file html แล้วก็นำมาแสดงผลหน้าเว็บ (สกัดข้อมูลที่อยู่ของภาพอีก 10 ภาพ)

เวลา

6. ย้อนทำตั้งแต่ขั้นตอนที่ 1ใหม่สำหรับรูป 1จนถึง10

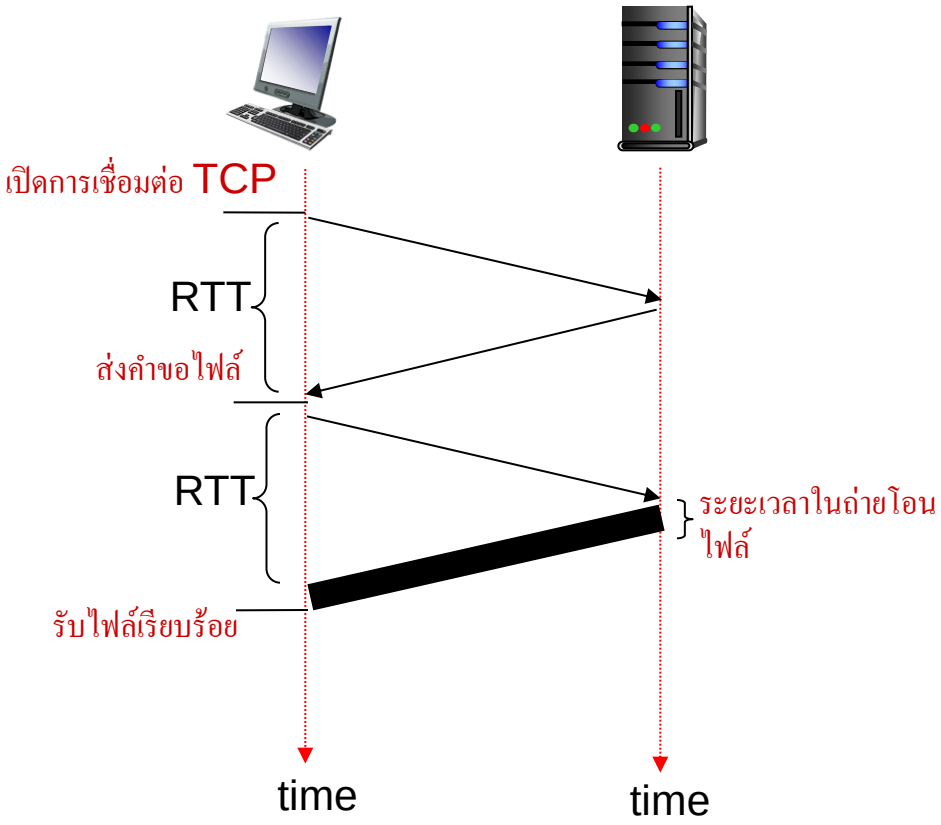
Non-persistent HTTP: เวลาตอบกลับ

RTT (Round Trip Time): เวลาใช้ในการเดินทางของแพคเกจเล็ก ๆ จากลูกข่ายไปยังเซิฟเวอร์แล้วกลับมา

ระยะเวลาตอบกลับ ของ HTTP:

- 1 RTT เพื่อเปิดการเชื่อมต่อ TCP
- 1 RTT สำหรับ request และ สำหรับ response ขนาด 2-3 bytes
- เวลาในการถ่ายโอนข้อมูล
- ระยะเวลาตอบกลับ ของ non-persistent HTTP =

  2RTT+ เวลาในการโอนถ่ายข้อมูล



Persistent HTTP

ประเด็นของ non-persistent HTTP:

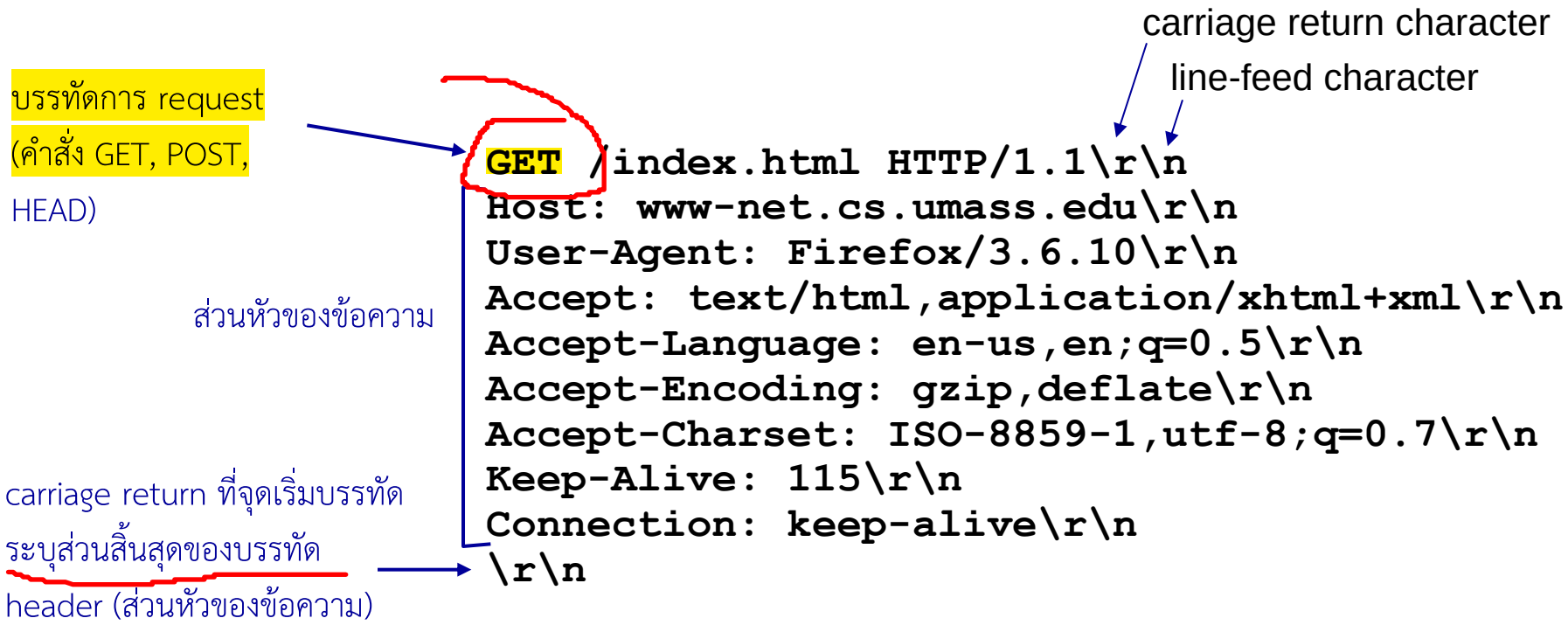
- ต้องใช้ 2 RTTs ต่อ1 ออบเจ็ค
- เสียเวลาที่ตัว OS เพื่อสร้างการเชื่อมต่อทุกครั้ง
- ส่วนใหญ่บราวเซอร์จะเปิดหลายๆการเชื่อมต่อเพื่อดึงหลาย ๆ object ในหน้าเว็บในเวลาเดียวกัน

persistent HTTP:

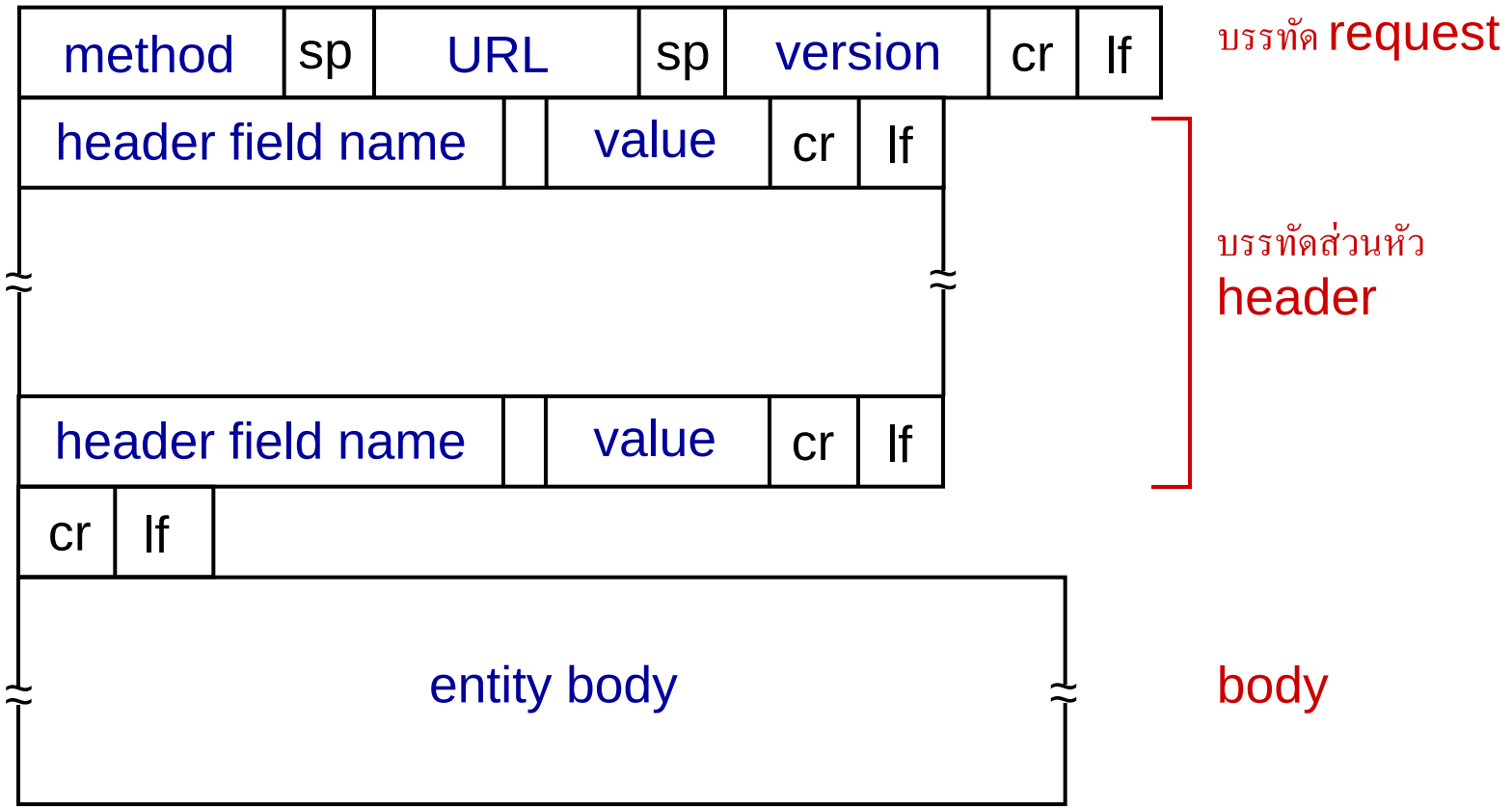
- เซิฟเวอร์จะเปิดการเชื่อมต่อค้างไว้หลังจากส่ง response
- การส่งข้อมูลต่อไประหว่าง client / server ก็จะใช้การเชื่อมต่อที่เปิดไว้นี้
- client ส่ง request ได้ทันทีเมื่ออยากจะถึง object ในหน้าเว็บ
- ใช้เพียง 1 RTT สำหรับการดึงข้อมูล object ทั้งหมด

HTTP request message

- ข้อความ HTTP มีสองชนิด: *request (ข้อความขอบริการ)*, *response (ข้อความตอบกลับ)*
- HTTP request message:
  - ASCII (ข้อความที่มนุษย์อ่านได้)



HTTP request message: format ทั่วไป



การส่งข้อมูลไปยังเซิฟเวอร์

วิธี POST:

- ตัวหน้าเว็บเพจส่วนมากจะมีช่องให้ใส่ข้อมูล
- ข้อมูล input ที่ป้อนจะถูกใส่ไปในส่วนของบอดี้

วิธี URL:

- ใช้วิธี GET
- ข้อมูล input จะถูกใส่ลงไปใน field URL ที่อยู่ใน header ของ request

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

Method types

HTTP/1.0:

- GET
- POST
- HEAD
  - บอกเซิฟเวอร์ว่าไม่ต้องส่งข้อมูลที่ร้องขอไปมาที่ client แล้ว

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - ใช้สำหรับอัพโหลดไฟล์ไปใน body (ไปยัง folder ของ server ที่ระบุไว้ใน field URL ที่ header ของข้อความ)
- DELETE
  - ใช้ลบไฟล์ที่ระบุใน field URL

ข้อมูล HTTP ที่ถูกตอบกลับมา

บรรทัดระบุ status (protocol status code status phrase)

header lines

ข้อมูล, เช่น file HTML ที่ขอไป

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

รหัสสถานะที่เกี่ยวกับ HTTP ที่เซิฟเวอร์อาจตอบกลับมา

- รหัสสถานะจะปรากฏในบรรทัดแรกขอข้อความจากเซิฟเวอร์ที่ตอบกลับ
- ตัวอย่างรหัสบางรหัส

200 OK

- การร้องขอสำเร็จ, สิ่งที่ร้องขออยู่ด้านท้ายของข้อความนี้

301 Moved Permanently

- สิ่งที่ร้องขอถูกย้ายไปแล้ว, ที่อยู่ใหม่อยู่ด้านท้ายของข้อความนี้ (Location:)

400 Bad Request

- เซิฟเวอร์ไม่เข้าใจข้อความร้องขอ

404 Not Found

- ไม่พบไฟล์ที่ร้องขอบนเซิฟเวอร์

505 HTTP Version Not Supported

- ร้องขอ version ของ HTTP ที่ server ไม่ได้ support

ทดลองเชื่อมต่อ HTTP ด้วยตัวเอง

1. telnet ไปยัง Web server:

```
telnet cis.poly.edu 80
```

เปิดการเชื่อมต่อ TCP ที่พอร์ท 80 ไปยัง cis.poly.edu. อะไรที่พิมพ์ลงไปจะถูกส่งไปยังพอร์ท 80 ของเซิฟเวอร์ที่ cis.poly.edu

2. พิมพ์การร้องขอ GET HTTP request:

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

ด้วยคำสั่งนี้ก็จะเป็นการส่งคำร้องขอไปยังเซิฟเวอร์

3. ดูข้อความที่เซิฟเวอร์ตอบกลับมา

(หรือใช้โปรแกรม Wireshark เพื่อที่จะดู HTTP request/response)

สถานะระหว่างผู้ใช้-server: cookies

เว็บไซต์เป็นจำนวนมากใช้ cookies

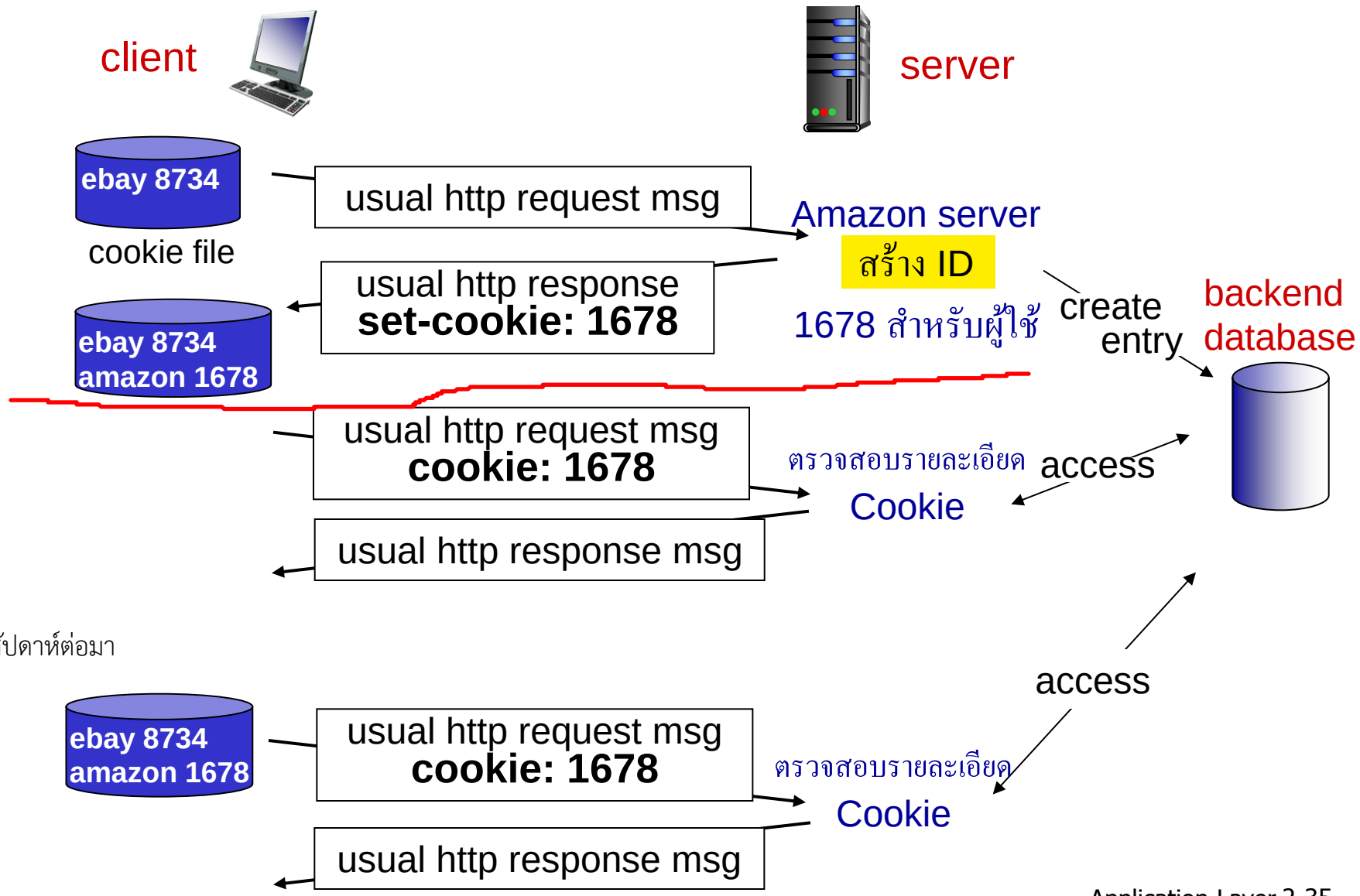
ส่วนประกอบ 4 อย่าง (ดูรูปหน้า 2-35):

1) บรรทัด set-cookie ในข้อความตอบกลับของ HTTP
2) บรรทัดยืนยัน cookie ในข้อความร้องขอหน้าเว็ปถัดไป
3) ไฟล์ cookie ที่ถูกเก็บทั้งบนเครื่องผู้ใช้ที่ถูกจัดการโดย Web browser
4) ฐานข้อมูลที่ทำงานในส่วนหลังของ server

ตัวอย่าง:

- ซูซานเล่นอินเตอร์เน็ตจากเครื่อง PC
- เข้าไปยังเว็บ e-commerce ครั้งแรก
- เมื่อการร้องขอแรกไปถึงยังเว็บไซต์ ไซต์ก็จะสร้าง
  - สร้างหมายเลข ID ที่ไม่ซ้ำใคร
  - ข้อมูลหนึ่งข้อมูลที่ฐานข้อมูลสำหรับ ID นั้น ๆ

Cookies: การเก็บ “state” (ต่อ ...)



Cookies (ต่อ)

Cookies สามารถทำอะไรได้บ้าง:

- การระบุตัวตน
- รายการรถเข็นสินค้า
- การแนะนำสินค้าหรือข้อมูล
- สถานะของช่วงการสื่อสารของผู้ใช้ (Web e-mail)

aside

cookies และความปลอดภัย:

- cookies ทำให้เว็บไซต์รู้ข้อมูลของคุณเยอะทีเดียว
- บางทีอาจรวมถึงอีเมล์ ชื่อนามสกุล

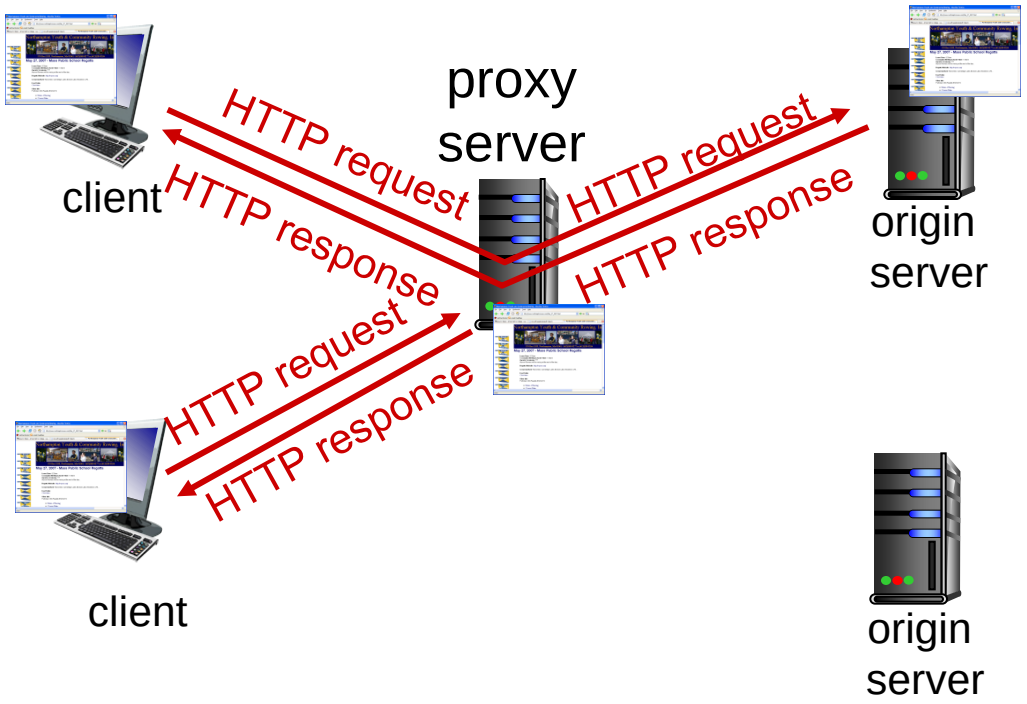
เราจะเก็บ "state" อย่างไร:

- ช่วงปลายของ protocol : เก็บสถานะที่ผู้ส่งหรือผู้รับ เมื่อมี transaction ไปหลาย ๆ ครั้งแล้ว
- cookies: ส่ง cookie (สถานะ) ไปพร้อม ๆ กับข้อความ http

Web caches (proxy server)

วัตถุประสงค์ : ตอบสนองการร้องขอจาก client โดยที่ไม่ต้องไปติดต่อกับเซิฟเวอร์จริง

- ผู้ใช้กำหนดในบราวเซอร์ให้เข้าเว็บผ่านแคช
- บราวเซอร์ส่งการร้องขอ HTTP ทั้งหมดไปยังแคช
  - ถ้าข้อมูลอยู่ในแคช: แคชส่งข้อมูลกลับ
  - ถ้าข้อมูลไม่อยู่ในแคช: ตัวแคชจะไปโหลดข้อมูลจาก server จริงมาเก็บไว้ ก่อนที่ส่งต่อให้ผู้ใช้



เรื่องเพิ่มเติมเกี่ยวกับ Web caching

- แคชทำหน้าที่เป็นทั้ง client และเซิฟเวอร์
  - เป็นเซิฟเวอร์เมื่อทำหน้าที่รับการร้องขอจาก client
  - เป็น client เมื่อทำหน้าที่ติดต่อกับเซิฟเวอร์จริง
- โดยทั่วไปแล้วแคชถูกติดตั้งโดยผู้ให้บริการ (มหาวิทยาลัย, บริษัท, ISP ที่ให้บริการตามบ้าน)

ทำไมต้องมีการเก็บแคช?

- ลดเวลาการตอบสนองจากการร้องขอของ client
- ลดปริมาณข้อมูลในลิงค์ (ที่จะไปเครือข่ายด้านนอก) ขององค์กร
- แคชช่วยให้ ผู้สร้างข้อมูลที่ไม่ค่อยมีเงินสามารถมีบริการข้อมูลอย่างมีประสิทธิภาพมาก (เหมือนที่การแชร์ไฟล์แบบ P2P ทำ)

ตัวอย่างการแคช:

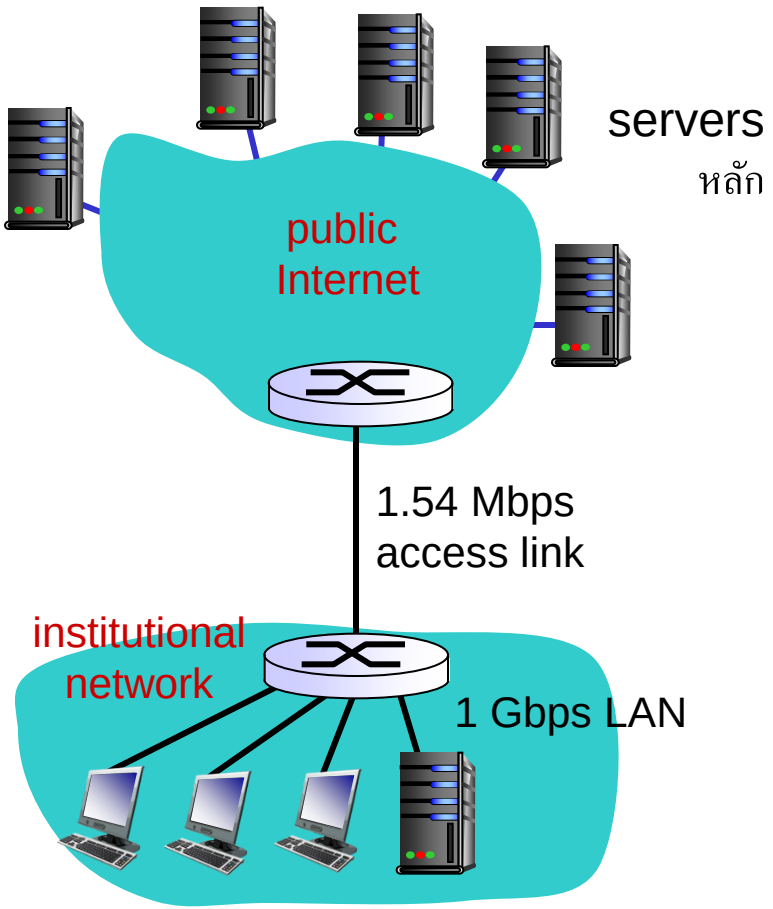
สมมติฐาน:

- ขนาดของ object โดยเฉลี่ย: 100K bits
- ค่าเฉลี่ยการร้องขอจากบราวเซอร์ไปยังเซิฟเวอร์หลัก: 15 request/วินาที
- อัตราเฉลี่ยของข้อมูลมายังบราวเซอร์ : 1.50 Mbps
- RTT จากเร้าเตอร์ไปยังเซิฟเวอร์หลัก: 2 sec
- bandwidth ของ access link : 1.54 Mbps

ผลลัพธ์:

- อัตราการใช้ LAN (LAN utilization): 15%
- อัตราการใช้ access link = 99% *ปัญหา!*
- total delay = Internet delay + access delay + LAN delay

  = 2 sec + **minutes** + usecs



ตัวอย่างการแคช: access link ที่ใหญ่ขึ้น

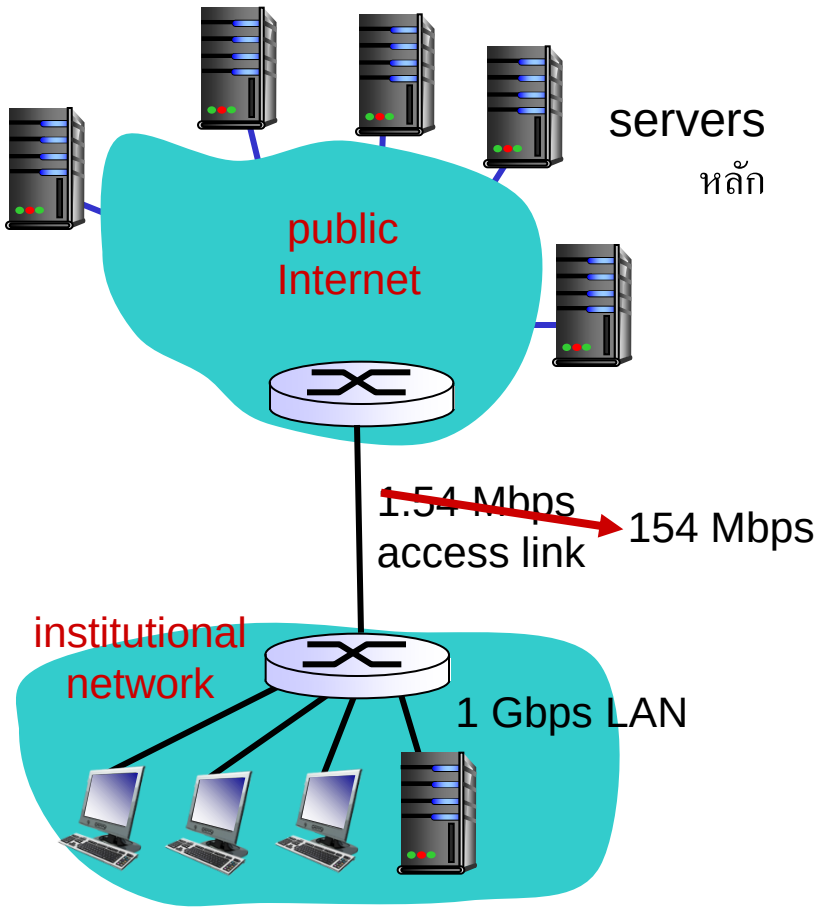
สมมติฐาน:

- ขนาดของ object โดยเฉลี่ย: 100K bits
- ค่าเฉลี่ยการร้องขอจากบราวเซอร์ไปยังเซิฟเวอร์หลัก: 15 request/วินาที
- อัตราเฉลี่ยของข้อมูลมายังบราวเซอร์ : 1.50 Mbps
- RTT จากเร้าเตอร์ไปยังเซิฟเวอร์หลัก: 2 sec
- bandwidth ของ access link : ~~1.54 Mbps~~ 154 Mbps

ผลลัพธ์:

- อัตราการใช้ LAN (LAN utilization): 15%
- อัตราการใช้ access link = ~~99%~~ 9.9%
- total delay = Internet delay + access delay + LAN delay

  = 2 sec + ~~minutes~~ msecs + usecs



ต้นทุน: ต้องเพิ่มความเร็วของ access link (ไม่ถูก!)

ตัวอย่างการแคช: ติดตั้งแคชท้องถิ่น

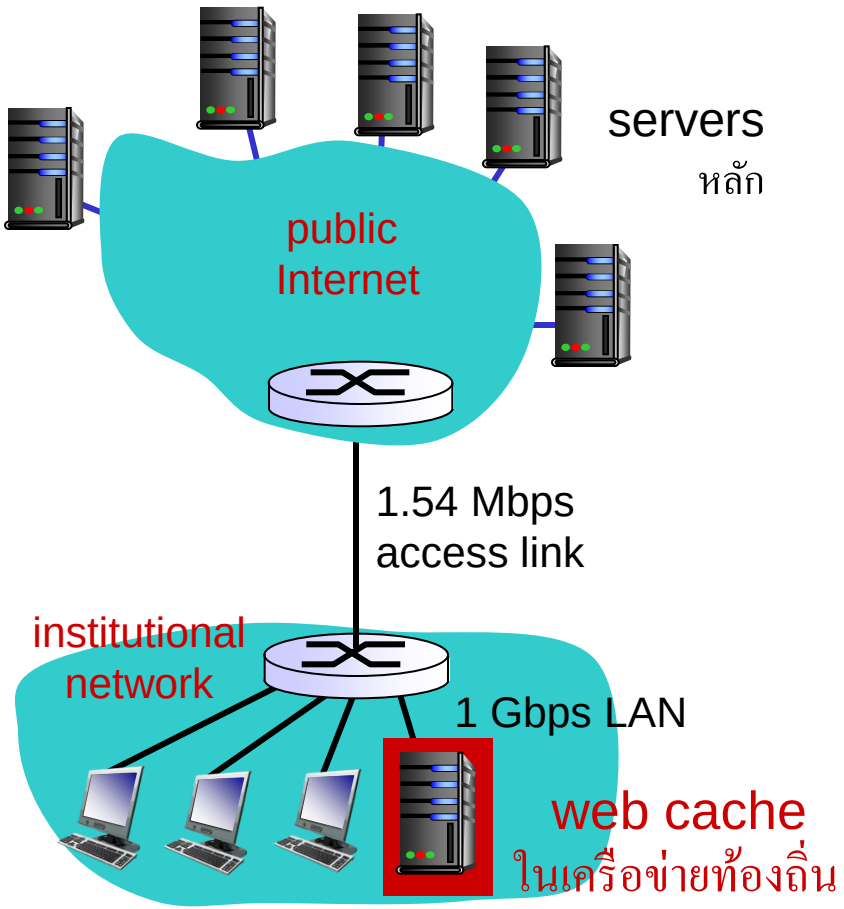
สมมติฐาน:

- ขนาดของ object โดยเฉลี่ย: 100K bits
- ค่าเฉลี่ยการร้องขอจากบราวเซอร์ไปยังเซิฟเวอร์หลัก: 15 request/วินาที
- อัตราเฉลี่ยของข้อมูลมายังบราวเซอร์ : 1.50 Mbps
- RTT จากเร้าเตอร์ไปยังเซิฟเวอร์หลัก: 2 sec
- bandwidth ของ access link : 1.54 Mbps

ผลลัพธ์:

- อัตราการใช้ LAN (LAN utilization): 15%
- อัตราการใช้ access link = ?
- total delay = ?

How to compute link utilization, delay?

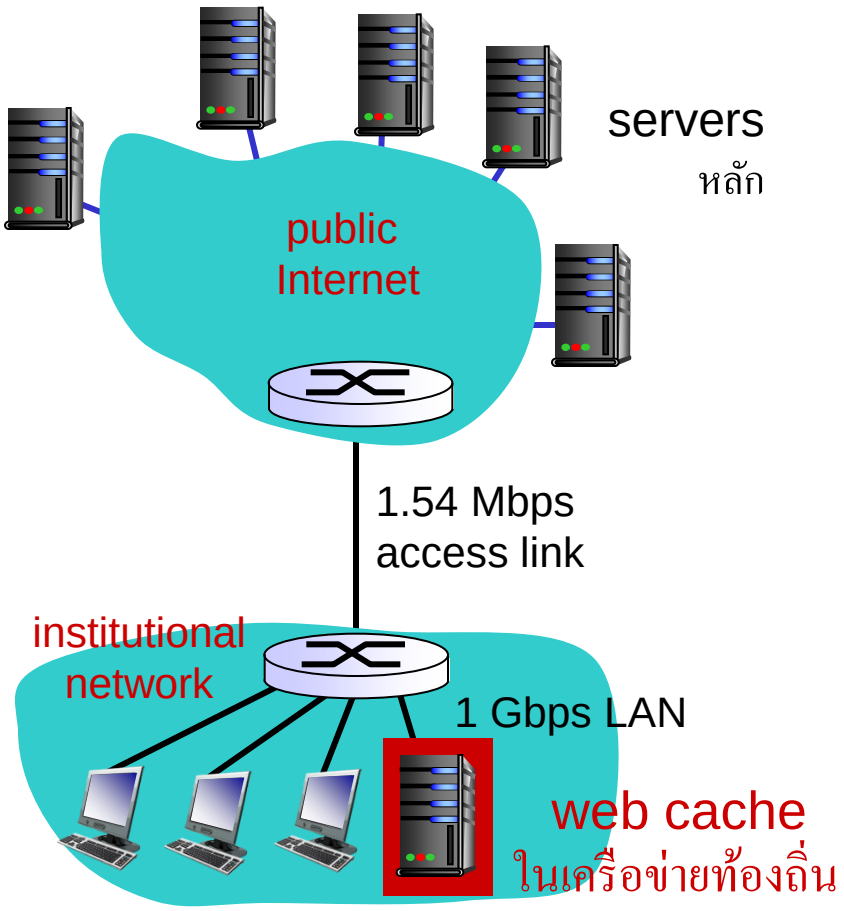


ต้นทุน: web cache (ถูก!)

ตัวอย่างการแคช: ติดตั้งแคชท้องถิ่น

คำนวณ utilization และ delay access link เมื่อมีการติดตั้ง cache:

- สมมติว่า hit rate ของ cache คือ 0.4
  - 40% ของ requests จะมีอยู่ใน cache, 60% ของ requests อยู่ที่ server หลัก
- utilization ของ access link:
  - 60% ของ requests ใช้ access link
- อัตราข้อมูลที่จะมาที่ browsers ผ่าน access link = 0.6*1.50 Mbps = .9 Mbps
  - utilization = 0.9/1.54 = .58
- total delay
  - = 0.6 * (delay from origin servers) +0.4 * (delay when satisfied at cache)
  - = 0.6 (2.01) + 0.4 (~msecs)
  - = ~ 1.2 secs
  - delay น้อยลงเมื่อใช้ 154 Mbps link กับ แคช (ถูกกว่าด้วย!)

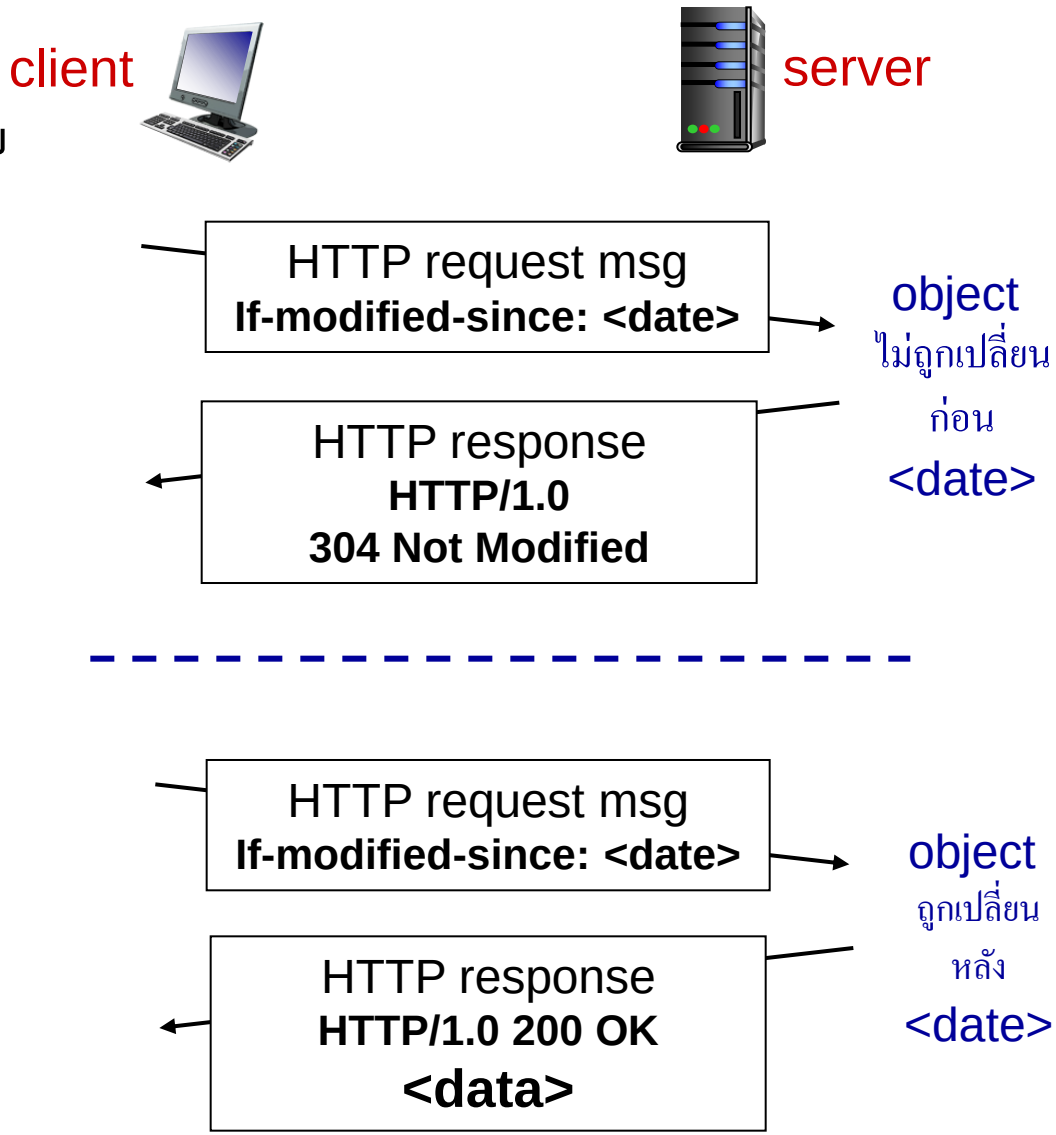


การ GET แบบมีเงื่อนไข

- *จุดประสงค์:* server ไม่ส่งออปเจ็คกลับมาหากข้อมูลไม่เปลี่ยนแปลง
  - ไม่มีความล่าช้าจากการโอนถ่ายข้อมูล
  - ลดอัตราการใช้งานของ link ลง
- *cache:* ระบุวันที่เก็บแคชไว้ใน HTTP request

  If-modified-since: <date>
- *server:* จะไม่ส่งออบเจ็คกลับมาหากแคชที่ client มีข้อมูลเหมือนกับทาง server:

  HTTP/1.0 304 Not Modified



Chapter 2: outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

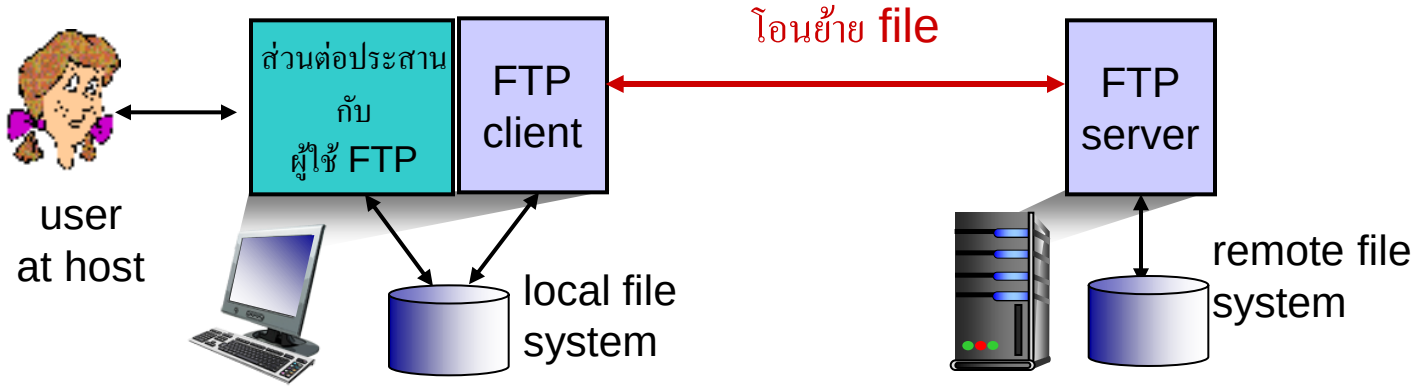
- SMTP, POP3, IMAP

2.5 DNS

2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

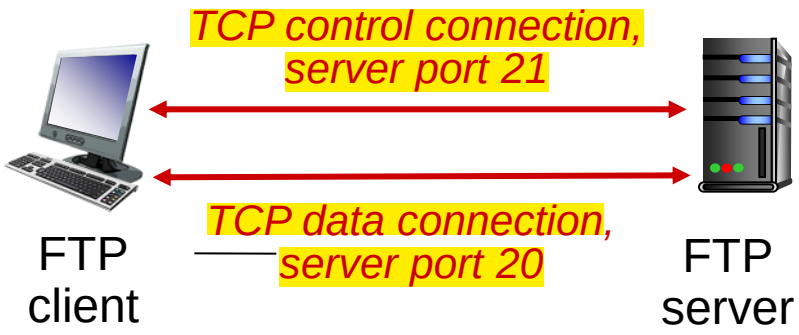
FTP: the file transfer protocol



- คัดลอกไฟล์ จาก/ไปที่ เครื่องที่อยู่ไกล
- รูปแบบ ไคลเอนต์/เซิร์ฟเวอร์
  - ไคลเอนต์ : ด้านที่เริ่มต้นส่งข้อมูล (อาจจะส่งไป /รับจากเครื่องที่อยู่ไกล)
  - เซิร์ฟเวอร์ : เครื่องที่อยู่ไกล
- ftp: RFC 959
- ftp เซิร์ฟเวอร์ใช้ port 21

FTP: connection สำหรับควบคุม และสำหรับข้อมูล

- FTP client ติดต่อ FTP เซิฟเวอร์ ที่พอร์ท 21 ผ่าน TCP
- ไคลเอนต์จะทำการพิสูจน์ตัวตนและสิทธิในการเข้าใช้งานบนเซิฟเวอร์ผ่าน connection สำหรับการควบคุม
- ไคลเอนต์สามารถดูข้อมูลในไดเรกทอรี่ ส่งคำสั่งผ่าน connection สำหรับการควบคุม
- เมื่อ server ได้รับคำสั่งให้โอนย้าย file server จะเปิดคอนเน็คชั่น TCP อีกคอนเน็คชั่นนึงสำหรับการถ่ายโอนไฟล์
- หลังจากถ่ายโอนข้อมูล 1 ไฟล์สิ้นสุด server จะทำการปิดคอนเน็คชั่นที่พอร์ต 20 ทันที



- การทำงานที่มีการแยกคอนเน็คชั่นเป็น 2 คอนเน็คชั่นคือ คอนเน็คชั่นควบคุมและ คอนเน็คชั่นข้อมูลนั้น เป็นการทำงานที่เรียกว่า : *"out of band"*
- FTP เซิฟเวอร์จะทำการเก็บ"สถานะ"การใช้งานของไคลเอนต์ ทำให้รู้ว่าผู้ใช้กำลังดู directory ใดอยู่ และ ยังช่วยการระบุตัวตนในตอนแรกเริ่ม

FTP commands, responses

ตัวอย่างคำสั่ง:

- ส่งคำสั่งเป็นอักขระ ASCII ผ่าน connection สำหรับการควบคุม
- **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

ตัวอย่างรหัสที่ server ตอบ

- รหัสสถานะและคำอธิบาย (เหมือนของ HTTP)
- **331 Username OK, password required**
- **125 data connection already open; transfer starting**
- **425 Can't open data connection**
- **452 Error writing file**

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

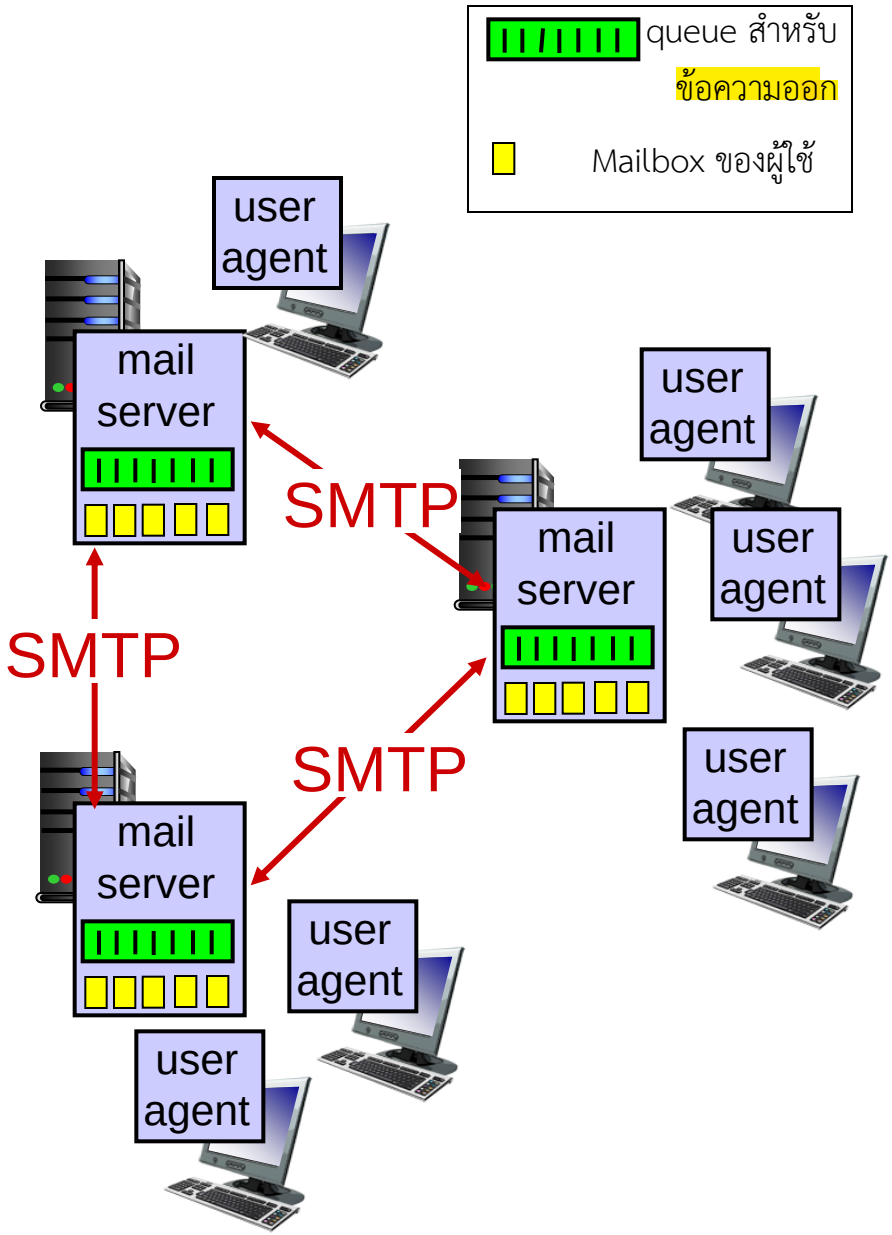
Electronic mail

3 ส่วนประกอบหลัก:

- user agents
- mail servers
- simple mail transfer protocol: SMTP

User Agent

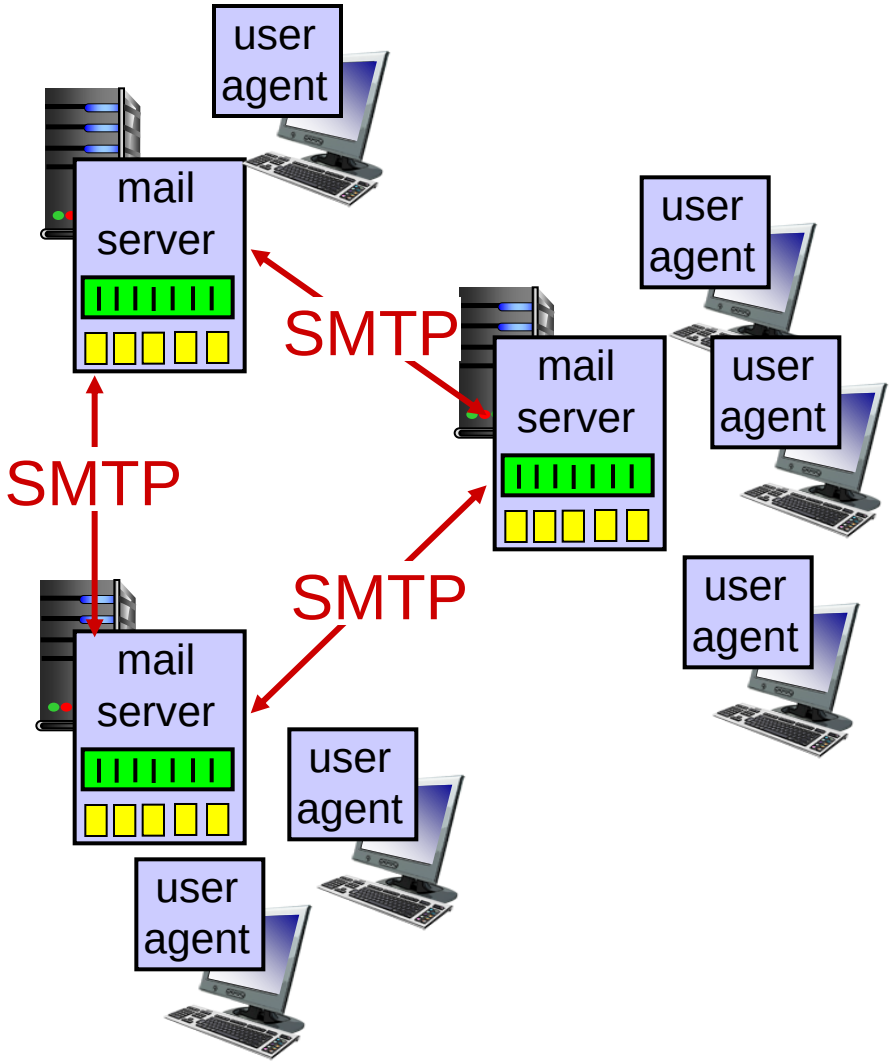
- หรือที่เรียกว่า “mail reader”
- สร้าง, แก้ไข, อ่านข้อความ mail
- เช่น Outlook, Thunderbird, iPhone mail client
- ข้อความที่จะออก และ ที่จะเข้าม ถูกเก็บใน server



Electronic mail: mail servers

mail servers:

- *mailbox* เป็นส่วนที่จัดเก็บข้อความที่มีมาถึงผู้ใช้
- ส่วนที่จัดเก็บข้อความของอีเมลลงในคิวก่อนที่จะถูกส่งออกไป
- การส่งข้อความในอีเมลระหว่างเครื่องเมลเซิร์ฟเวอร์ด้วย โปรโตคอล *SMTP*
  - เครื่องผู้ใช้ : ส่งอีเมลไปยังเครื่องเมลเซิร์ฟเวอร์
  - “เครื่องเซิร์ฟเวอร์” : รับอีเมลจากเครื่องเมลเซิร์ฟเวอร์



Electronic Mail: SMTP [RFC 2821]

- ใช้โปรโตคอล TCP เพื่อให้แน่ใจว่าข้อความถูกส่งอย่างถูกต้องจาก client ไปยังเซิฟเวอร์ผ่านพอร์ท 25
- การถ่ายโอนโดยตรง : เป็นการถ่ายโอนจากเครื่องเซริฟเวอร์ผู้ส่งไปยังเครื่องเซริฟเวอร์ผู้รับ
- การถ่ายโอนสามขั้นตอน
  - ขั้น handshaking (server ทักทายกัน, ตกลงค่าของการส่ง)
  - ขั้นถ่ายโอนข้อความ
  - ขั้นการปิด
- คำสั่งและการตอบกลับ (เหมือนกับ HTTP, FTP)
  - คำสั่ง: อักขระ ASCII
  - คำตอบ: รหัสสถานะและคำอธิบาย
- ข้อความต้องอยู่ในรูปแบบ 7-bit ASCI

สถานการณ์: Alice ส่งจดหมายไปหา Bob

1) Alice ใช้ UA เพื่อเขียนจดหมาย “ส่งถึง” bob@someschool.edu

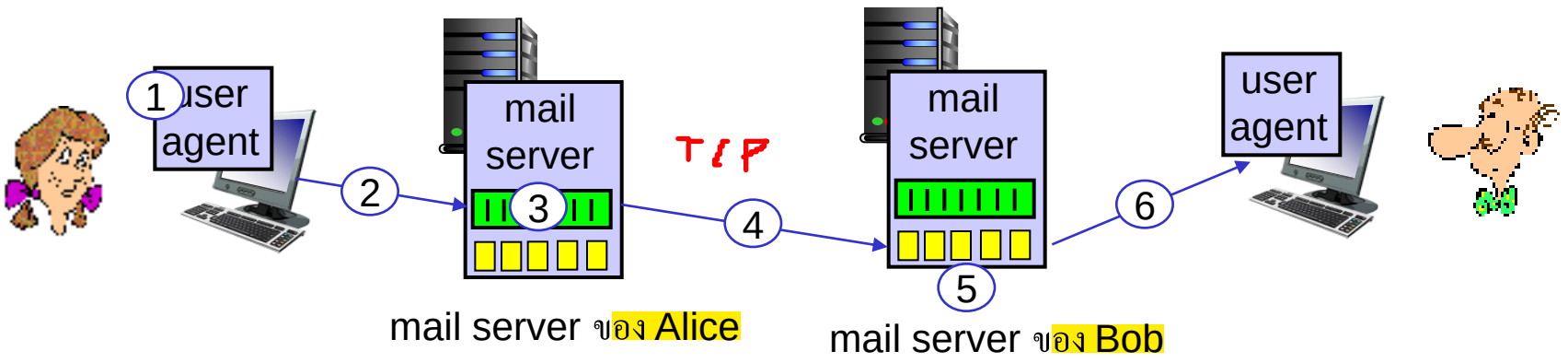
2) Alice’s UA ส่งจดหมายไปที่ mail server ของเธอ ; จดหมายถูกวางไว้ที่message queue

3) SMTP ฝั่ง client ที่อยู่ใน mail server ของ Alice เปิดการเชื่อมต่อ TCP กับ mail server ของ Bob

4) SMTP client ส่งจดหมายของ Alice บนการเชื่อมต่อ TCP

5) mail server ของ Bob วางจดหมายไว้ใน mailbox ของ Bob

6) Bob ใช้ user agent อ่านจดหมาย



ตัวอย่างการติดต่อกันของ SMTP

```
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
```

ลองติดต่อกับ mail server โดยใช้ SMTP ด้วยตัวเอง:

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

จากข้อความดังกล่าวข้างต้นจะช่วยให้คุณส่งอีเมล์โดยไม่ต้องใช้โปรแกรมอีเมล์

SMTP: final words

- SMTP ใช้ connection ที่คงอยู่ (persistent connection)
- SMTP กำหนดให้ ส่วนหัวอีเมล์ กับ ตัว body ถูกเขียนด้วยอักขระ ASCII 7-bit
- SMTP serverจะใช้CRLF.CRLF เพื่อกำหนดจุดสิ้นสุดข้อความ

เทียบกับ HTTP:

- HTTP: ดึง web page มา
- SMTP: ผลักอีเมล์ไป
- ทั้งคู่ติดต่อกันโดยส่งคำสั่ง/คำตอบเป็นอักขระ ASCII และใช้รหัสสถานะ
- HTTP: แต่ละ object อยู่ในข้อความตอบกลับเฉพาะของแต่ละ object เอง
- SMTP: object ทั้งหมดถูกรวมอยู่ในเมล์

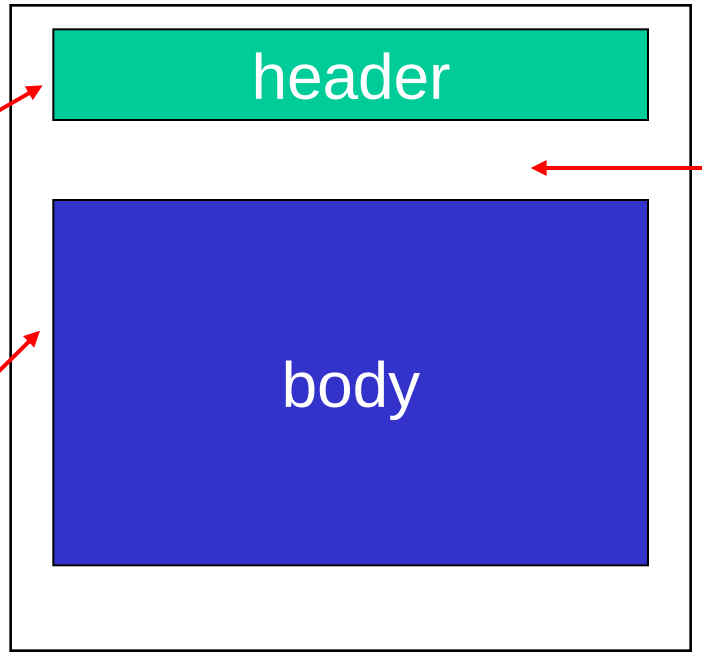
Mail message format

SMTP: protocol สำหรับการแลกเปลี่ยนอีเมลล์

RFC 822: มาตรฐานสำหรับรูปแบบข้อความ:

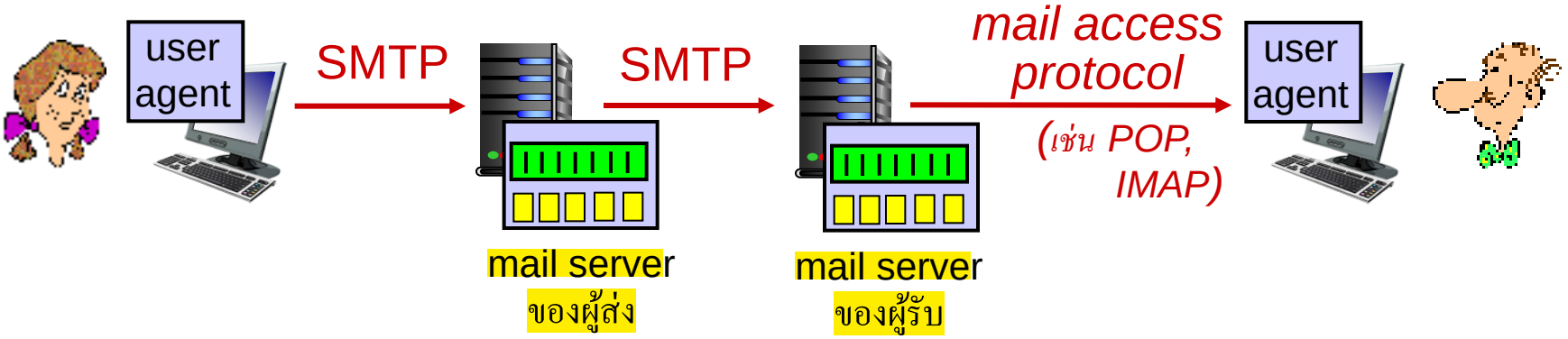
- บรรทัดส่วนหัว เช่น,
  - To:
  - From:
  - Subject:

  ต่างจากคำสั่ง MAIL FROM, RCPT TO ของ SMTP!
- Body: ส่วน “ข้อความอีเมล์”
  - เป็นอักขระ ASCII เท่านั้น



Mail access protocols

(โปรโตคอลสำหรับการอ่านเมล์)



- SMTP: จัดส่ง/จัดเก็บไปยังเซิร์ฟเวอร์ของผู้รับ
- protocol สำหรับการอ่านเมล์: ดึงข้อมูลจากเซิร์ฟเวอร์
  - POP: Post Office Protocol [RFC1939]: authorization (การระบุตัวตน), download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: มีคุณสมบัติมากขึ้น เช่น การจัดการการเก็บข้อความบนเซิร์ฟเวอร์
  - HTTP: gmail, Hotmail, Yahoo! Mail, ฯลฯ

POP3 protocol

ขั้นตอน authorization

- client commands:
  - **user:** ระบุ username
  - **pass:** รหัสผ่าน
- server responses
  - +OK
  - -ERR

ขั้นตอนการติดต่อทำงาน (transaction phase), client:

- **list:** แสดงรายการหมายเลขของข้อความ
- **retr:** ดึงข้อความโดยใช้หมายเลข
- **dele:** ลบ
- 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 (เพิ่มเติม) และ IMAP

เรื่องเพิ่มเติมเกี่ยวกับ POP3

- ตัวอย่างการใช้งาน POP3 ใน slide ที่แล้วอยู่ใน mode "download และ delete"
  - Bob ไม่สามารถอ่าน e-mail ซ้ำได้ หากมีการเปลี่ยน client
- POP3 "download-and-keep": copy ข้อความไว้ใน client ได้หลายเครื่อง
- POP3 ไม่มีการเก็บ state ของ sessions (ช่วงเวลาการติดต่อ)

IMAP

- เก็บข้อความ ทั้งหมดไว้ที่เดียว: ที่ server
- ยินยอมให้ user สามารถจัดการข้อความใน folder ได้
- เก็บสถานะของผู้ใช้แม้จะต่าง sessions กัน:
  - ตั้งชื่อ folder และจัดกลุ่มข้อความตาม Folder ได้ (โดยจะจัดคู่ id ของข้อความกับชื่อ folder)

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

DNS: domain name system

คน: สามารถระบุตัวตนได้หลายวิธี:

- เลขประชาชน, ชื่อ, เลข passport

Internet hosts, routers:

- IP address (32 bit) – ใช้สำหรับระบุที่อยู่
- "ชื่อ", เช่น, www.yahoo.com ซึ่งชื่อเหล่านี้ง่ายที่มนุษย์จะจดจำ

คำถาม: การจับคู่ของ name กับ IP address มีวิธีการอย่างไร แล้วในทางกลับกันล่ะ?

Domain Name System (ระบบชื่อโดเมน):

- *ฐานข้อมูลแบบกระจาย* ทำงานโดยใช้โครงสร้างแบบลำดับชั้นประกอบด้วย *name server* จำนวนมาก
- *protocol ในชั้น application:* hosts สื่อสารกับ name servers เพื่อ *resolve* ชื่อโดเมน (การแปลงข้อมูลระหว่าง address และ name)
  - ข้อสังเกต: เป็นการทำงานส่วนที่เป็นแกนของ Internet ที่ถูก implement ที่ชั้น application ด้วย
  - ดังนั้น ความซับซ้อนอยู่ที่ขอบของเครือข่าย (network' s "edge")

DNS: บริการ, โครงสร้าง

บริการของ DNS

- แปล hostname ให้เป็น IP address
- ช่วยให้ host มีชื่อแฝงหลายๆชื่อได้
  - ชื่อจริงที่เป็นทางการ (canonical), ชื่อนามแฝง (alias)
- mail server มีชื่อแฝงได้หลายชื่อ
- ช่วยกระจายการโหลด
- Load (request จาก client) จะถูกกระจายไปยัง Web Server ที่เก็บข้อมูลเหมือน ๆ กันได้ โดยอาศัย DNS: server หลายเครื่อง (ที่อยู่ IP ต่างกัน) ที่ให้บริการข้อมูลแบบเดียวกัน จะถูกกำหนดไว้กับชื่อโดเมนเดียวได้

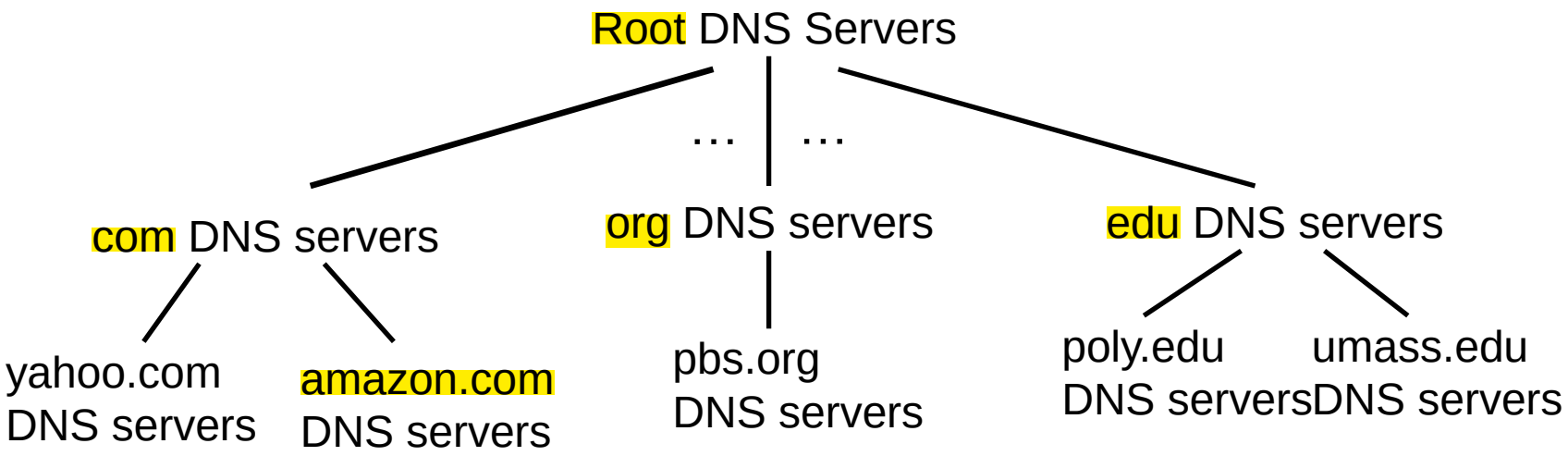
เหตุผลที่ไม่ใช้ DNS แบบรวมศูนย์

- จุดบอดจุดเดียว
- ปริมาณของการโหลดข้อมูลเยอะเกินไป
- ฐานข้อมูลส่วนกลางอยู่ไกลไป
- ดูแลรักษาได้ยาก

คำตอบ: ไม่ scale!

→ *Input มากขึ้น ตอบสนองได้ไม่ดี*

DNS: ฐานข้อมูลที่เป็นลำดับชั้นและกระจาย

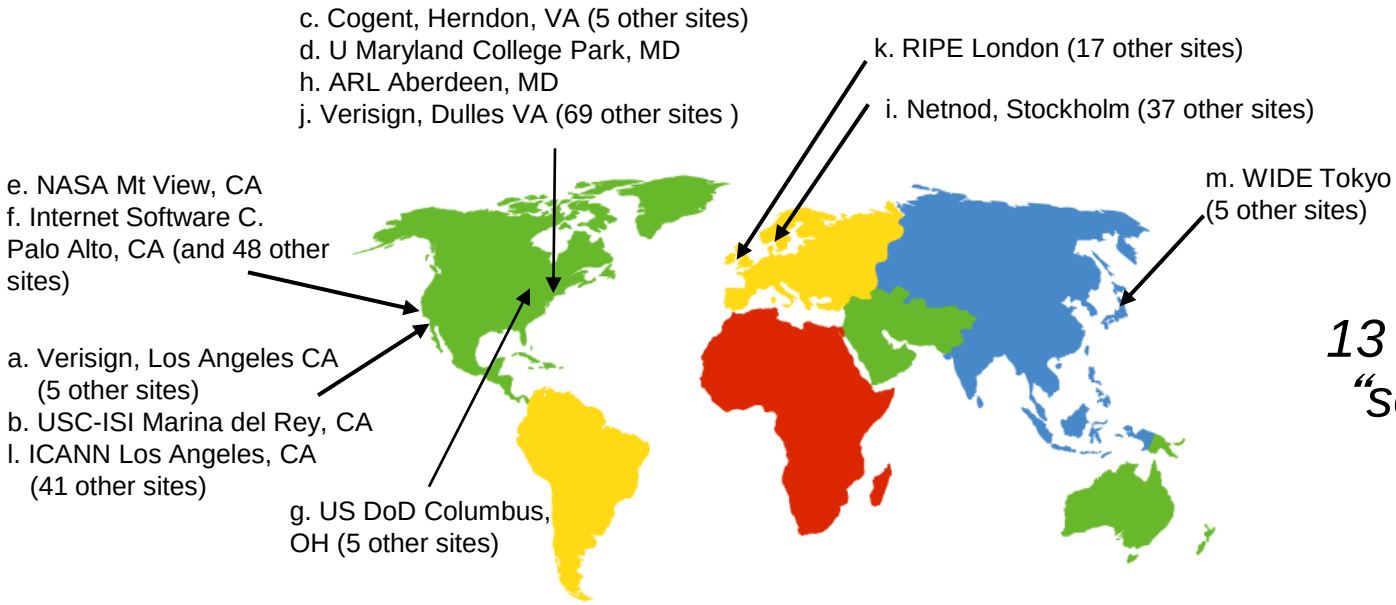


เมื่อลูกข่ายต้องการรู้ถึง IP ของ www.amazon.com การแปลชื่อโดเมนมีขั้นตอนต่อไปนี้:

- ลูกข่ายสอบถามไปยัง root server เพื่อที่จะไปค้นหา .com DNS server
- ลูกข่ายสอบถามไปยัง .com DNS server เพื่อที่จะไปค้นหา amazon.com DNS server
- ลูกข่ายสอบถามไปยัง amazon.com DNS server เพื่อขอ IP ของ www.amazon.com

DNS: root name servers

- name server ท้องถิ่นติดต่อ root name server เมื่อมันไม่สามารถแปลชื่อโดเมนได้
- root name server:
  - ถ้าไม่สามารถแปลชื่อโดเมนได้ root ก็จะไปติดต่อกับ authoritative name server
  - เมื่อได้รับข้อมูลคู่ของชื่อโดเมนและที่อยู่ IP แล้ว
  - root ส่งข้อมูลคู่ดังกล่าวกลับไปที่ name server ท้องถิ่น



13 root name "servers" ทั่วโลก

TLD, authoritative servers

top-level domain (TLD) servers (domain name server ระดับบน):

- ดูแลโดเมน com, org, net, edu, aero, jobs, museums, และรวมถึง top-level country domains, e.g.: uk, fr, ca, jp ทั้งหมด
- บริษัท Network Solutions ดูแล TLD servers สำหรับ .com
- บริษัท Educause ดูแล TLD servers สำหรับ .edu

authoritative DNS servers:

- เป็น DNS server(s) ที่มีองค์กรเป็นเจ้าของ, ให้บริการแปลง authoritative hostname เป็นไอพีของเครื่องขององค์กร
- ถูกดูแลโดยองค์กรเองหรือผู้ให้บริการ name server โดยเฉพาะ

DNS name server ท้องถิ่น

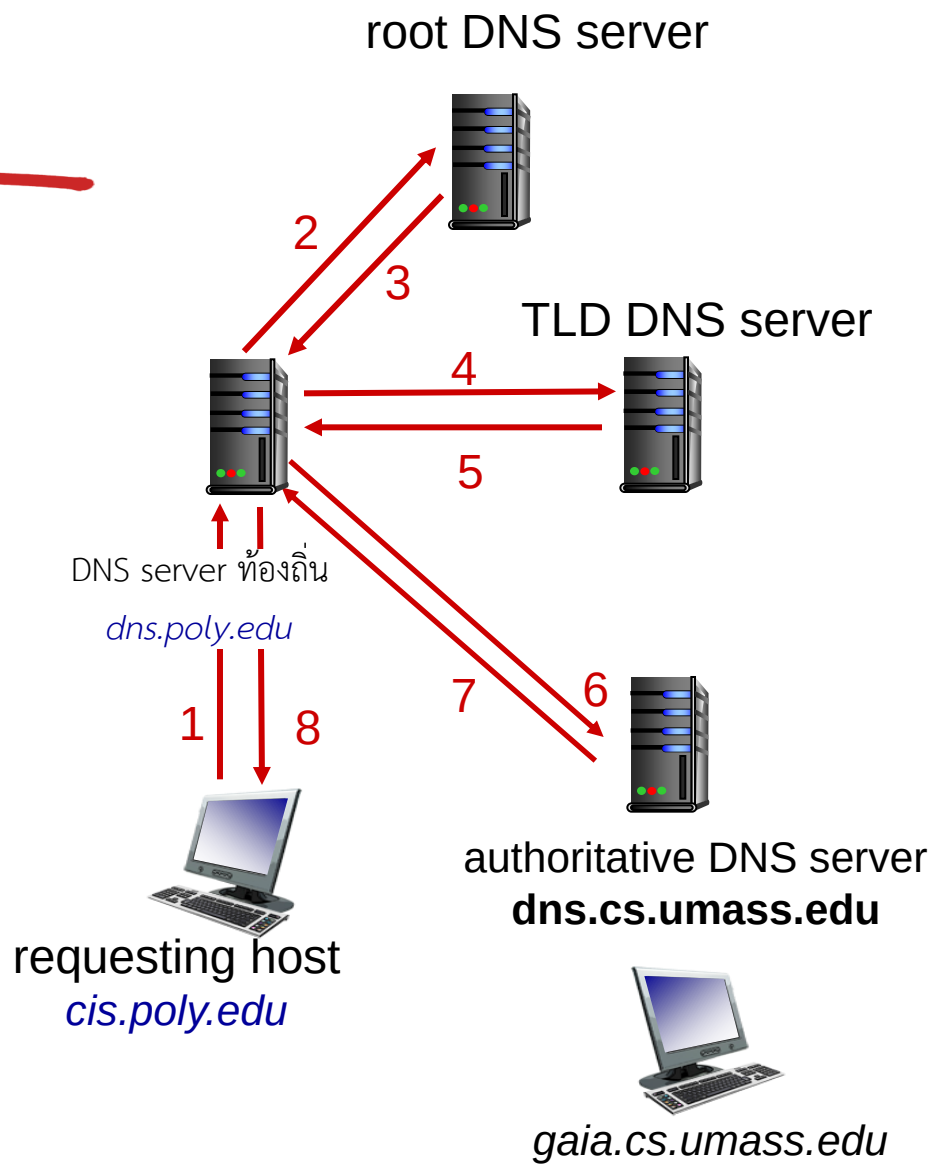
- ไม่ได้อยู่ในโครงสร้างลำดับขั้นของ DNS ซะทีเดียว
- แต่ละ ISP (ISP ตามบ้านเรือน, องค์กร, มหาวิทยาลัย) ต่างก็มี name sever -> เรียกกันว่า “default name server”
- เมื่อเครื่องต้นทางสร้างคำขอบริการ DNS, คำขอจะถูกส่งต่อไปที่ DNS server ท้องถิ่น
  - จำการจับคู่ล่าสุดซึ่งจะแปลง name เป็น address (แต่ข้อมูลการจับคู่ก็อาจหมดอายุได้)
  - ทำหน้าที่เหมือน proxy, ส่งต่อคำขอเข้าไปในระบบ domain name ที่เป็นลำดับชั้น

ตัวอย่างการแปลชื่อโดเมน

- เครื่อง cis.poly.edu ต้องการหมายเลขไอพีแอดเดรสของเครื่อง gaia.cs.umass.edu

iterated query (ส่งแบบซ้ำ):

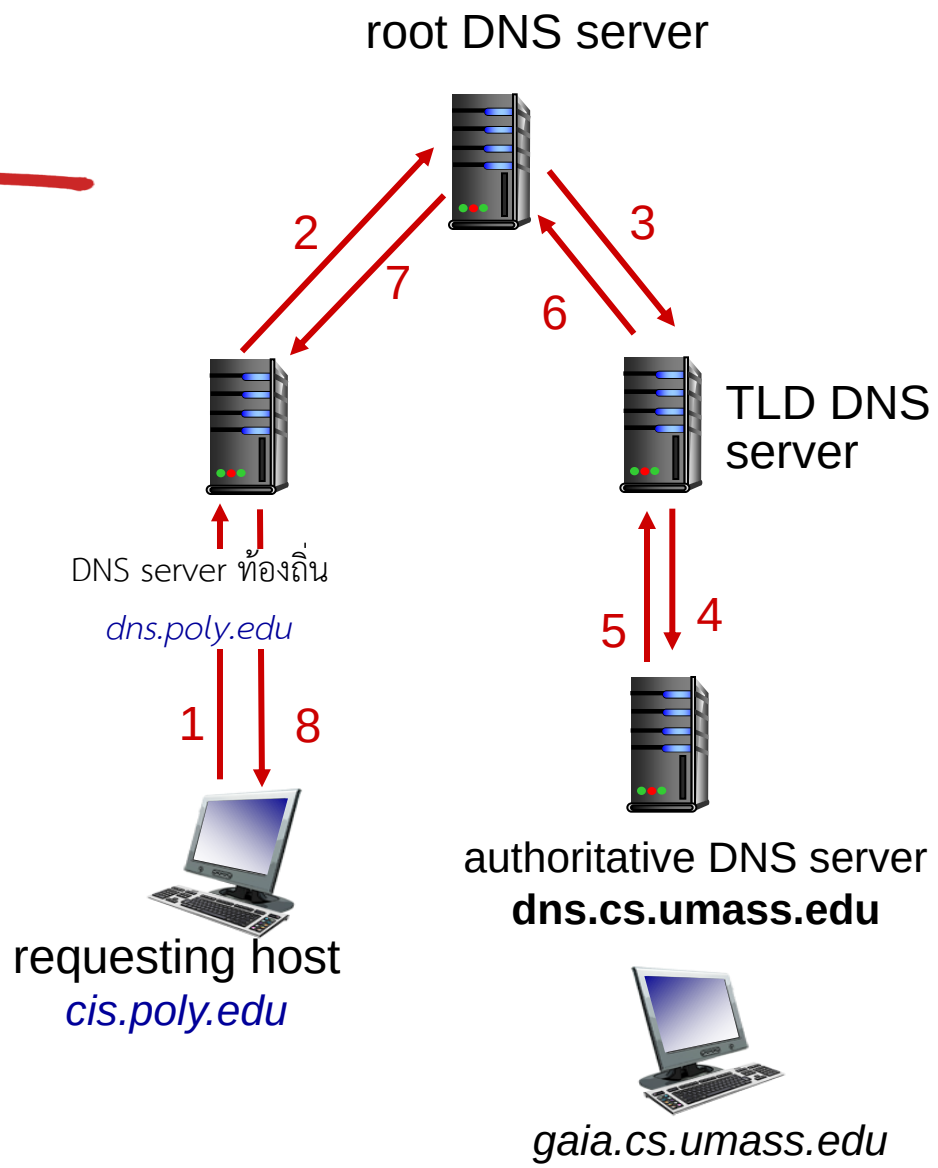
- เซิฟเวอร์ที่ถูกติดต่อตอบชื่อเซิฟเวอร์ที่ต้องไปหาต่อกลับไป
- “ฉันไม่รู้จักชื่อนี้ลองไปถามเซิฟเวอร์นี้ซิ”



ตัวอย่างการแปลชื่อโดเมน

recursive query (ส่งแบบเวียนเกิด):

- ผลักภาระการหาชื่อที่อยู่ไปยัง name server ที่ถูกติดต่ออยู่
- เซิฟเวอร์ลำดับบนจะทำงานหนัก



DNS: การเก็บและการอัพเดทคู่ชื่อโดแมนกับที่อยู่ IP

- เมื่อ name server ได้รับข้อมูลการจับคู่ก็จะเก็บใส่แคชเอาไว้
  - แคชที่หมดอายุจะถูกลบโดยอัตโนมัติ (TTL)
  - TLD servers โดยปกติแล้วจะถูกเก็บใน local name servers
    - ดังนั้น ก็จะไม่ค่อยมีการติดต่อไปยัง root name servers
- หน่วยข้อมูลการจับคู่ที่แคชเก็บไว้อาจจะเก่าเกินไป (best effort name-to-address translation!)
  - เพราะ เครื่องอาจถูกเปลี่ยนเลขไอพีแต่ใช้ชื่อโดเมนเดิมได้ ซึ่ง name server ท้องถิ่นจะยังไม่รู้ถึงการเปลี่ยนแปลงนี้ จนกว่าหน่วยข้อมูลที่แคชที่เก็บไว้จะหมดอายุ
- การแจ้ง/การ update หน่วยข้อมูลถูกเสนอใน IETF standard
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

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

type=A

- **`name`** is ชื่อเครื่อง
- **`value`** is IP address

type=NS

- **`name`** is โดเมน (เช่น, foo.com)
- **`value`** is ชื่อเครื่องของ authoritative name server สำหรับโดเมนนี้

type=CNAME

- **`name`** is นามแฝง (alias) ที่จะใช้สำหรับ ชื่อที่เป็นทางการ (canonical)
- **`www.ibm.com`** จริง ๆ ก็เป็นนามแฝงของ **`servereast.backup2.ibm.com`**
- **`value`** is ชื่อเครื่องที่เป็นทางการ

type=MX

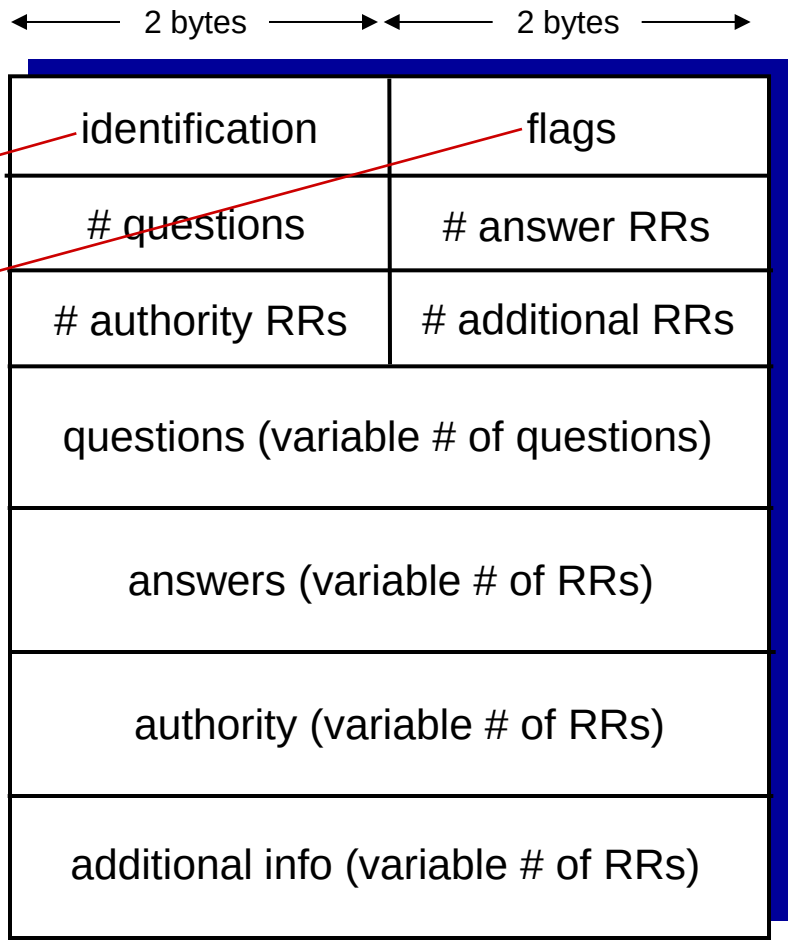
- **`value`** is ชื่อของ mailserver ที่เกี่ยวข้อง **`name`**

DNS protocol, messages

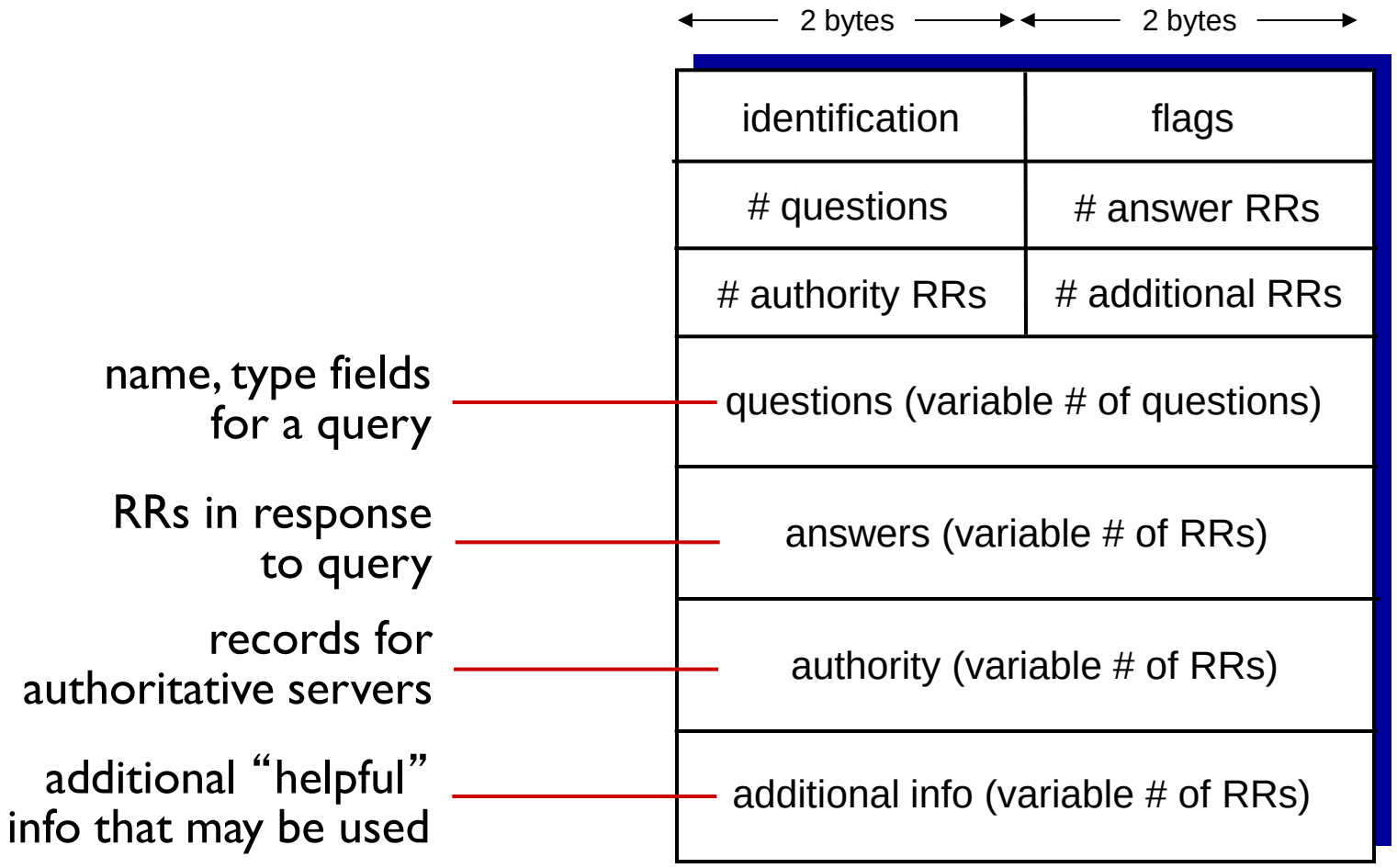
- *query* and *reply* messages, both with same *message format*

msg header

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



DNS protocol, messages



การใส่ records เข้าไปใน DNS

- ตัวอย่าง: บริษัทเพิ่งตั้งใหม่ชื่อ “Network Utopia”
- ลงทะเบียนชื่อโดเมน networkuptopia.com กับ *นายทะเบียน DNS* (เช่น Network Solutions)
  - ระบุชื่อโดเมน, ที่อยู่ IP ของ authoritative name server (ตัวหลัก และ ตัวสำรอง)
  - นายทะเบียนใส่ records 2 record เข้าไปใน .com TLD server:

    (networkutopia.com, dns1.networkutopia.com, NS)

    (dns1.networkutopia.com, 212.212.212.1, A)
- สร้าง record type A สำหรับ www.networkuptopia.com; record type MX สำหรับ networkutopia.com

การจู่โจม DNS

DDoS attacks

- Bombard root servers with traffic
  - Not successful to date
  - Traffic Filtering
  - Local DNS servers cache IPs of TLD servers, allowing root server bypass
- Bombard TLD servers
  - Potentially more dangerous

Redirect attacks

- Man-in-middle
  - Intercept queries
- DNS poisoning
  - Send bogus relies to DNS server, which caches

Exploit DNS for DDoS

- Send queries with spoofed source address: target IP
- Requires amplification

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

2.6 แอพพลิเคชัน P2P

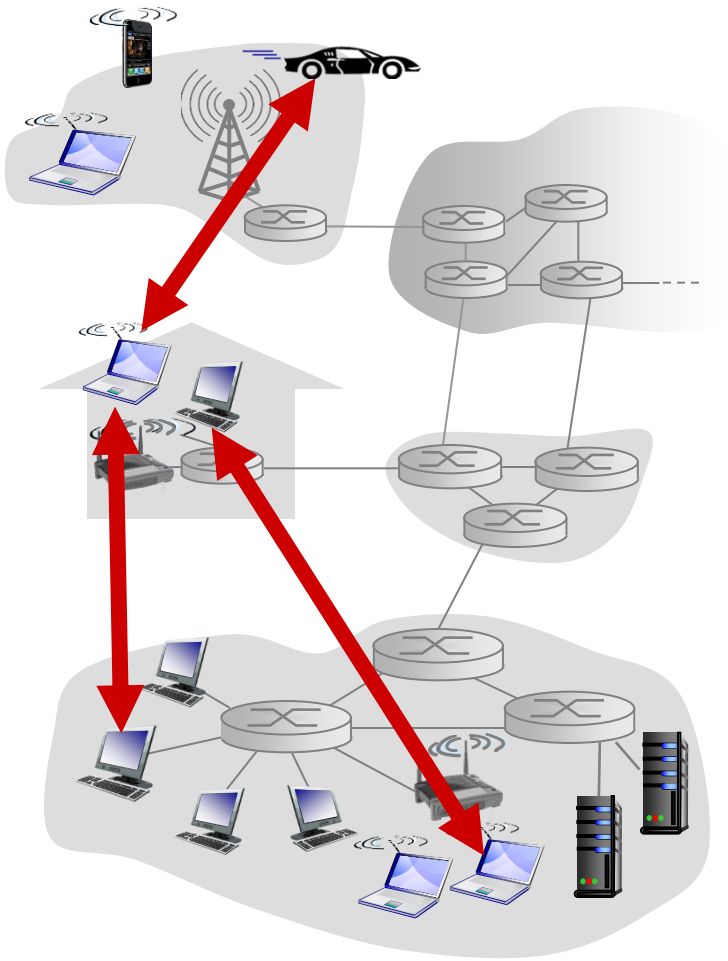
2.7 socket programming กับ UDP และ TCP

Pure P2P architecture

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

examples:

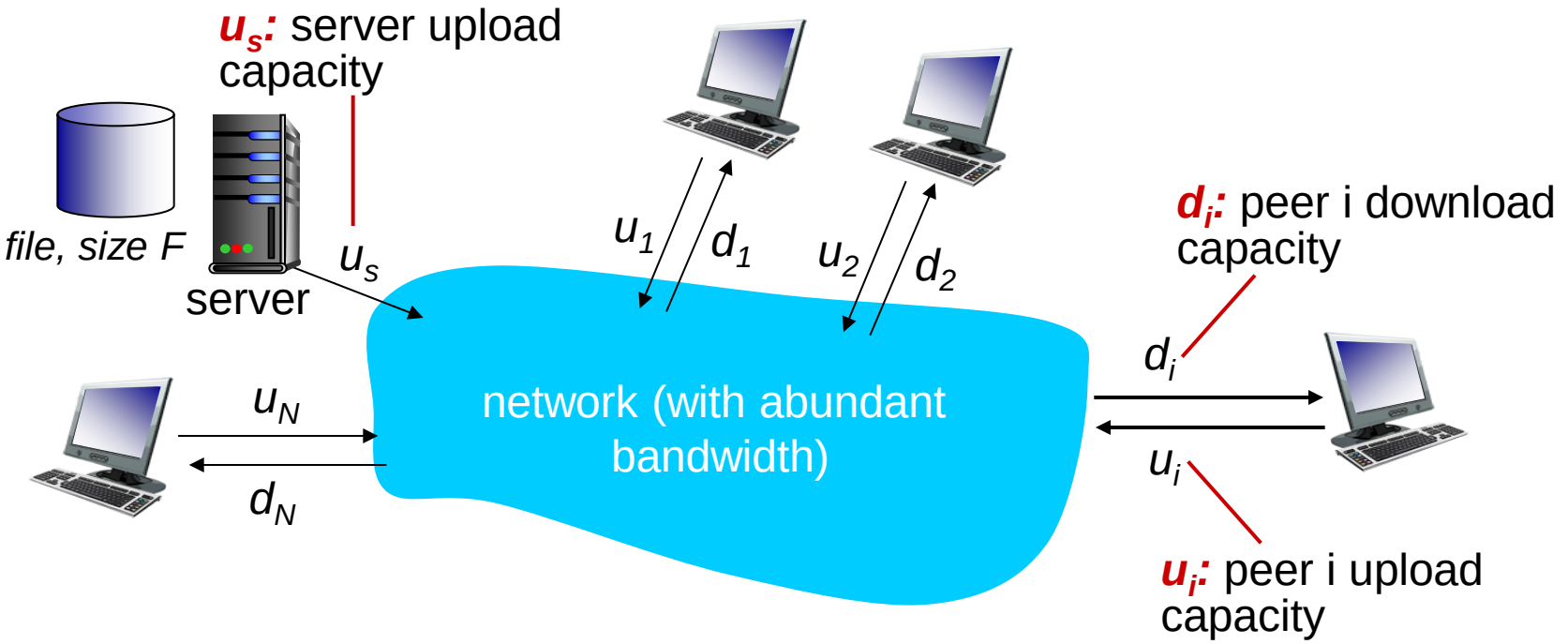
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

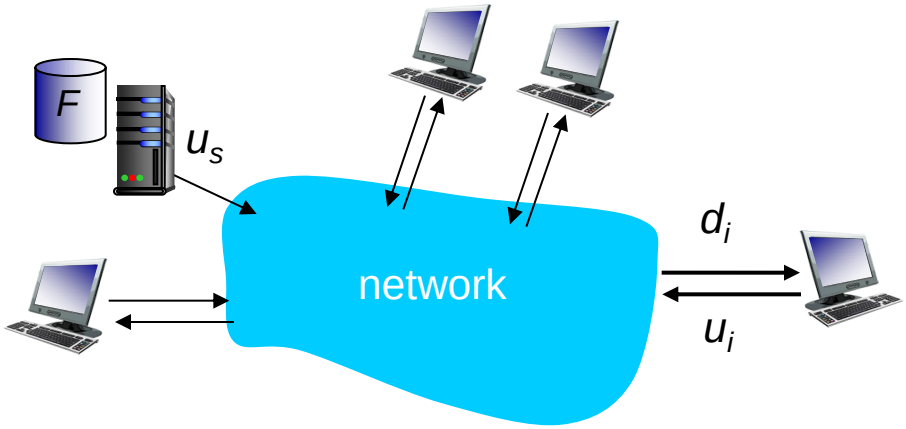
Question: how much time to distribute file (size *F*) from one server to *N* peers?

- peer upload/download capacity is limited resource



File distribution time: client-server

- *server transmission:* must sequentially send (upload) *N* file copies:
  - time to send one copy: F/u_s
  - time to send N copies: NF/u_s
- *client:* each client must download file copy
  - d_{min} = min client download rate
  - min client download time: F/d_{min}



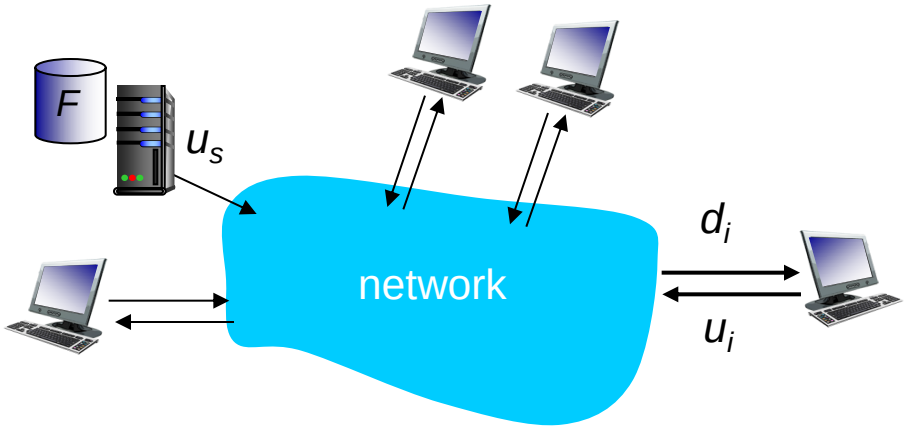
time to distribute F to N clients using client-server approach

$$D_{c\text{-}s} \geq max\{NF/u_s, F/d_{min}\}$$

increases linearly in N

File distribution time: P2P

- *server transmission:* must upload at least one copy
  - time to send one copy: F/u_s
- *client:* each client must download file copy
  - min client download time: F/d_{min}
- *clients:* as ~~aggregate~~ must download NF bits
  - max upload rate (limting max download rate) is $u_s + \Sigma u_i$



time to distribute F to N clients using P2P approach

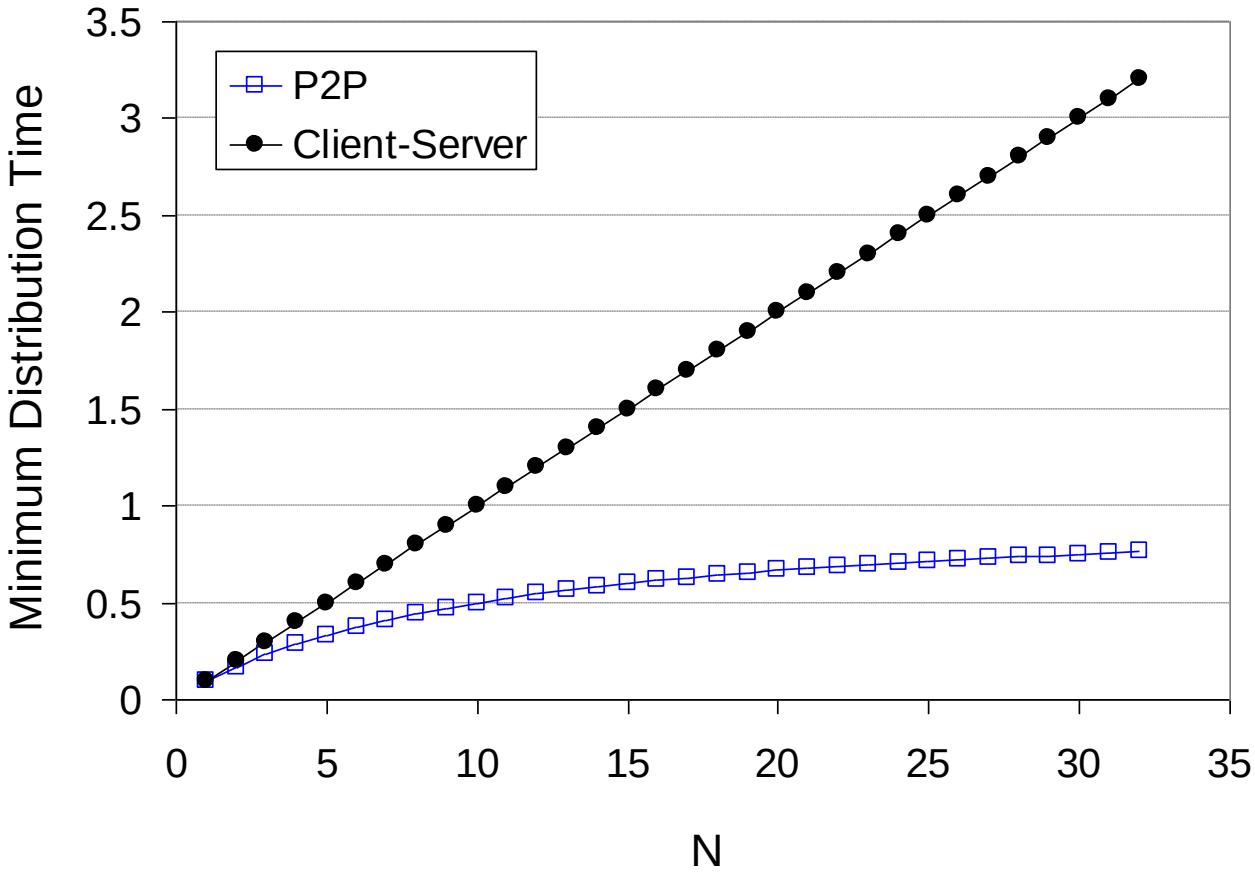
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + \Sigma u_i)\}$$

increases linearly in N …

… but so does this, as each peer brings service capacity

Client-server vs. P2P: example

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

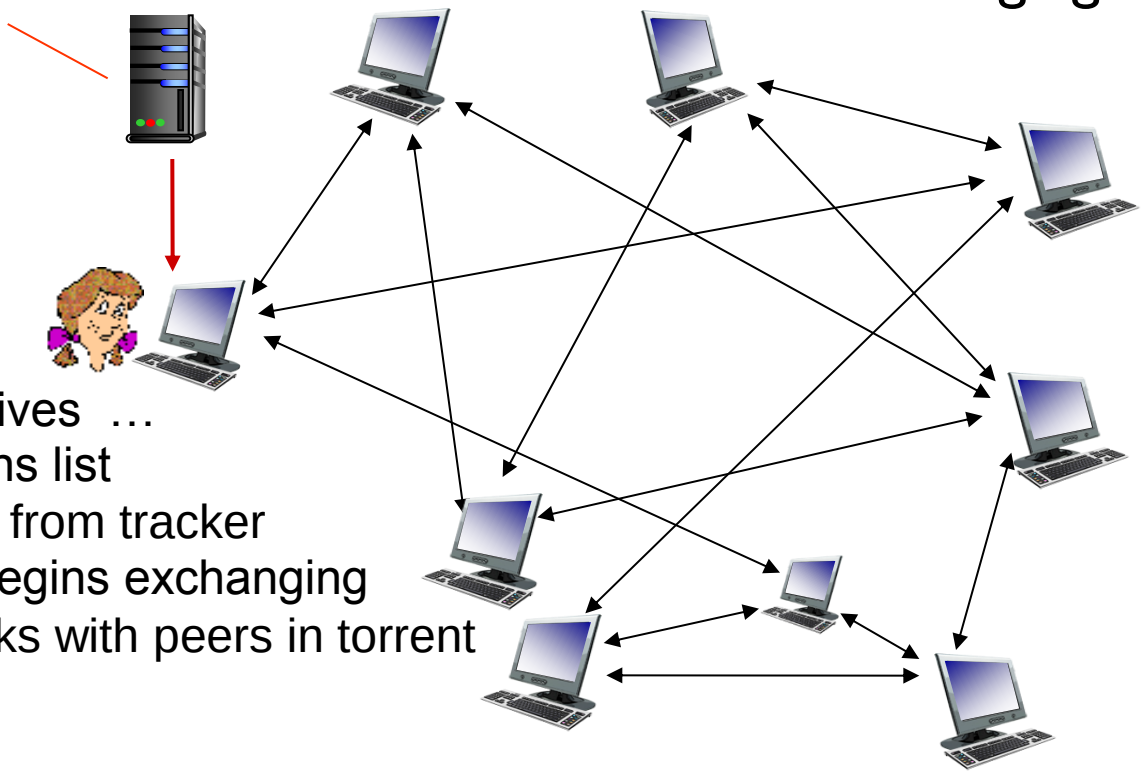


P2P file distribution: BitTorrent

- file divided into 256Kb chunks
- peers in torrent send/receive file chunks

tracker: tracks peers participating in torrent

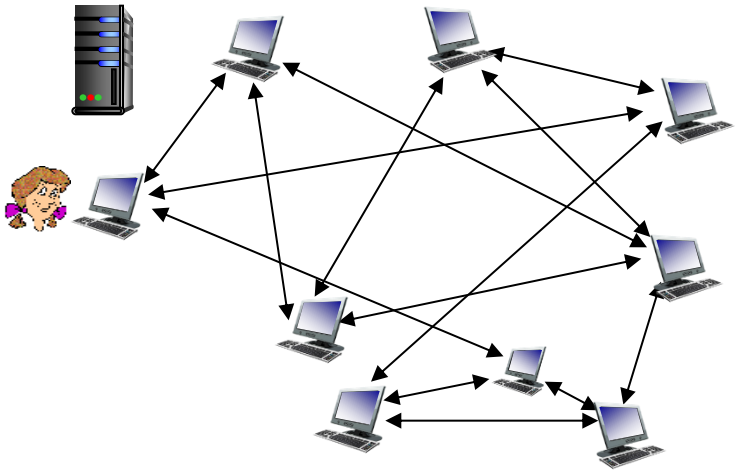
torrent: group of peers exchanging chunks of a file



Alice arrives …
… obtains list
of peers from tracker
… and begins exchanging
file chunks with peers in torrent

P2P file distribution: BitTorrent

- peer joining torrent:
  - has no chunks, but will accumulate them over time from other peers
  - registers with tracker to get list of peers, connects to subset of peers ("neighbors")



- while downloading, peer uploads chunks to other peers
- peer may change peers with whom it exchanges chunks
- *churn:* peers may come and go
- once peer has entire file, it may (selfishly) leave or (altruistically) remain in torrent

BitTorrent: requesting, sending file chunks

requesting chunks:

- at any given time, different peers have different subsets of file chunks
- periodically, Alice asks each peer for list of chunks that they have
- Alice requests missing chunks from peers, rarest first

sending chunks: tit-for-tat

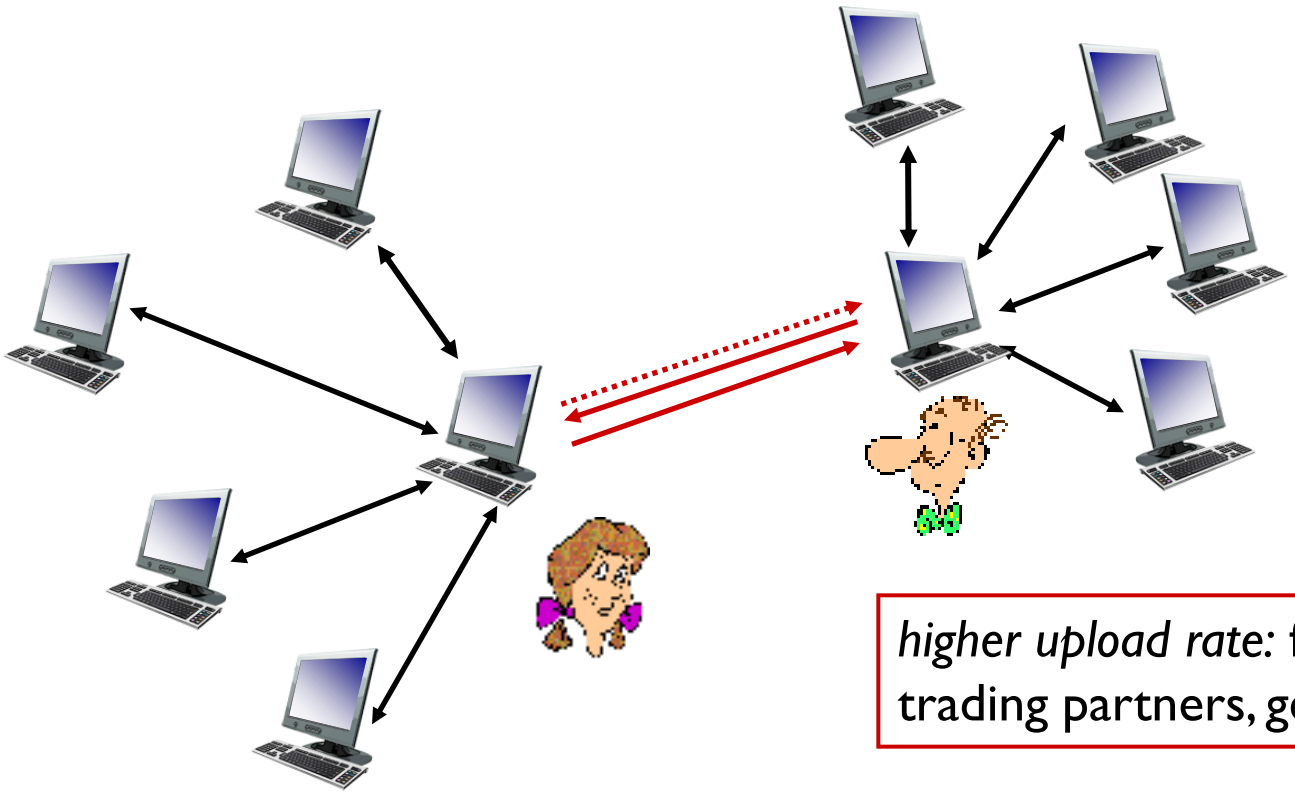
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
  - newly chosen peer may join top 4

BitTorrent: tit-for-tat

(1) Alice "optimistically unchokes" Bob

(2) Alice becomes one of Bob's top-four providers; Bob reciprocates

(3) Bob becomes one of Alice's top-four providers



higher upload rate: find better trading partners, get file faster !

Distributed Hash Table (DHT)

- DHT: a *distributed P2P database*
- database has (key, value) pairs; examples:
  - key: ss number; value: human name
  - key: movie title; value: IP address
- Distribute the (key, value) pairs over the (millions of peers)
- a peer queries DHT with key
  - DHT returns values that match the key
- peers can also insert (key, value) pairs

Q: how to assign keys to peers?

- central issue:
  - assigning (key, value) pairs to peers.
- basic idea:
  - convert each key to an integer
  - Assign integer to each peer
  - put (key,value) pair in the peer that is closest to the key

DHT identifiers

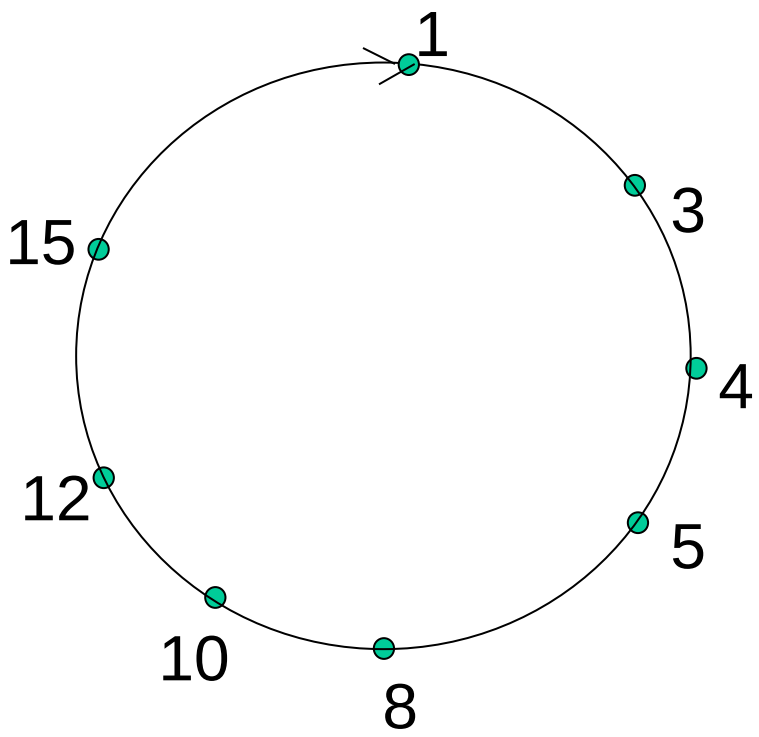
- assign integer identifier to each peer in range $[0,2^n-1]$ for some *n*.
  - each identifier represented by *n* bits.

- require each key to be an integer in same range
- to get integer key, hash original key
  - e.g., key = hash("Led Zeppelin IV")
  - this is why its is referred to as a *distributed "hash" table*

Assign keys to peers

- rule: assign key to the peer that has the *closest* ID.
- convention in lecture: closest is the *immediate successor* of the key.
- e.g., *n*=4; peers: 1,3,4,5,8,10,12,14;
  - key = 13, then successor peer = 14
  - key = 15, then successor peer = 1

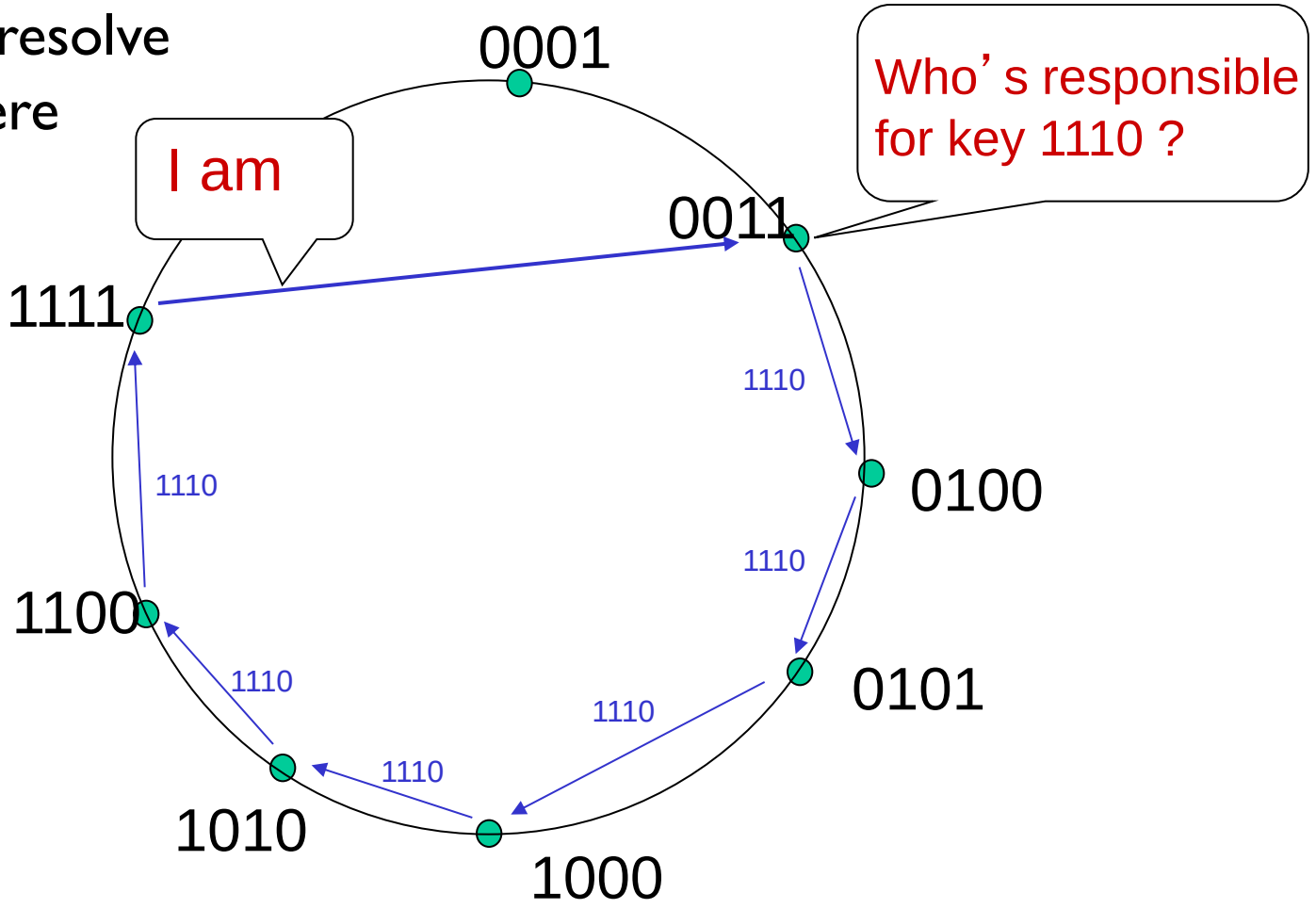
Circular DHT (1)



- each peer *only* aware of immediate successor and predecessor.
- "overlay network"

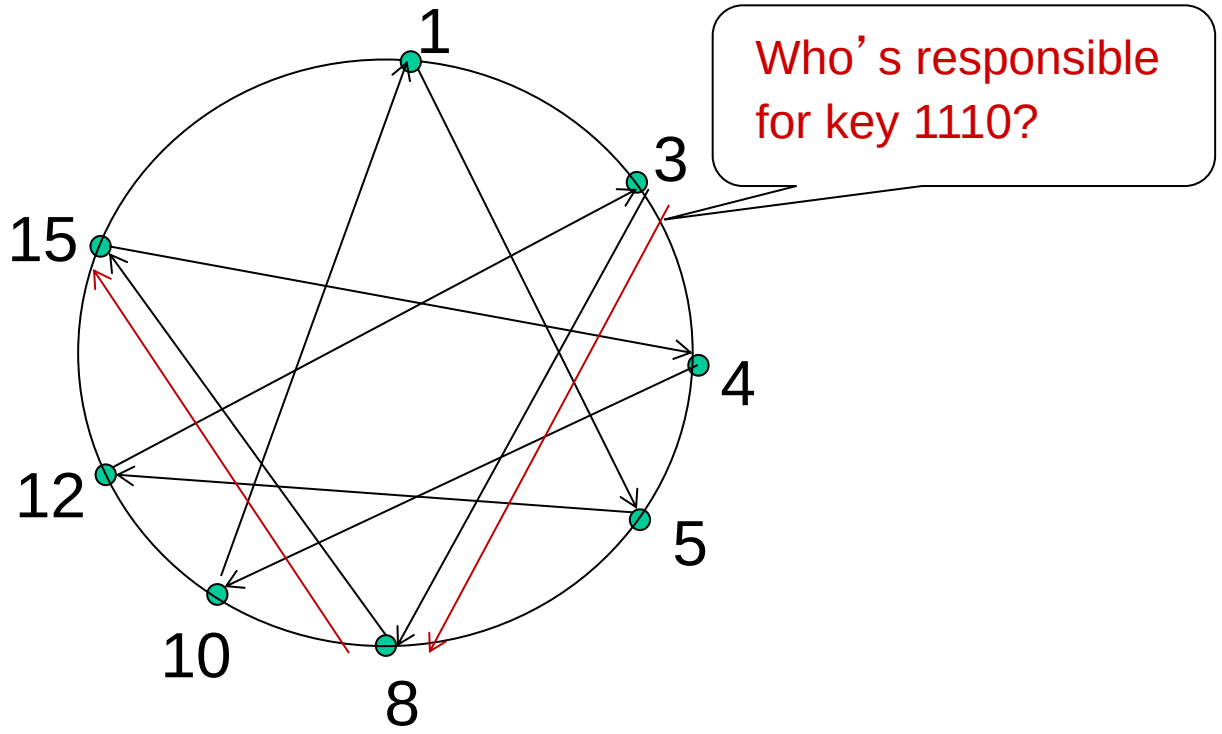
Circular DHT (1)

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



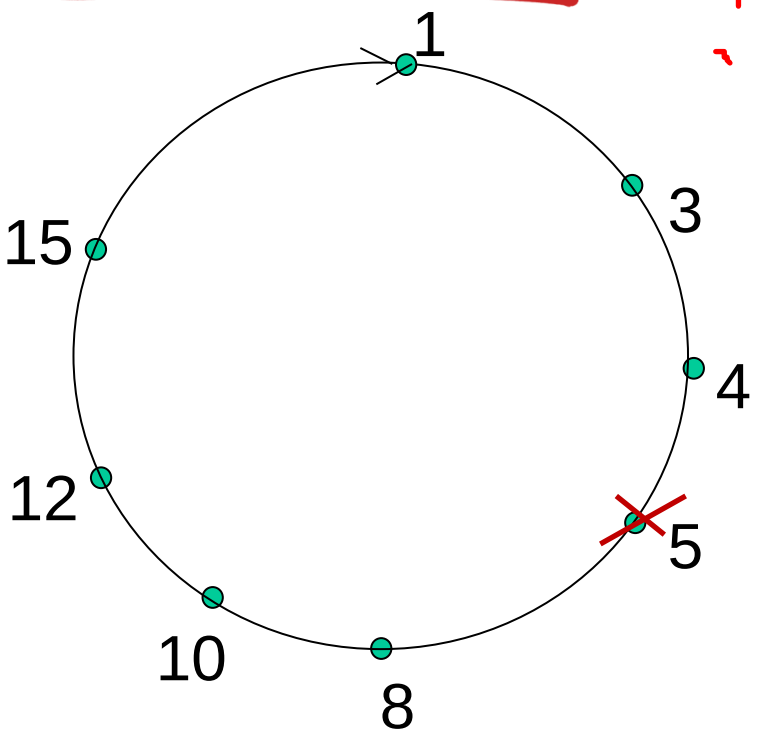
Define closest as closest successor

Circular DHT with shortcuts



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

Peer churn



handling peer churn:

- peers may come and go (churn)
- each peer knows address of its two successors
- each peer periodically pings its two successors to check aliveness
- if immediate successor leaves, choose next successor as new immediate successor

example: peer 5 abruptly leaves

- peer 4 detects peer 5 departure; makes 8 its immediate successor; asks 8 who its immediate successor is; makes 8's immediate successor its second successor.
- what if peer 13 wants to join?

บทที่ 2: Outline

2.1 หลักการของแอพพลิเคชันด้านระบบเครือข่าย

2.2 Web และ HTTP

2.3 FTP

2.4 อีเมล์

- SMTP, POP3, IMAP

2.5 DNS

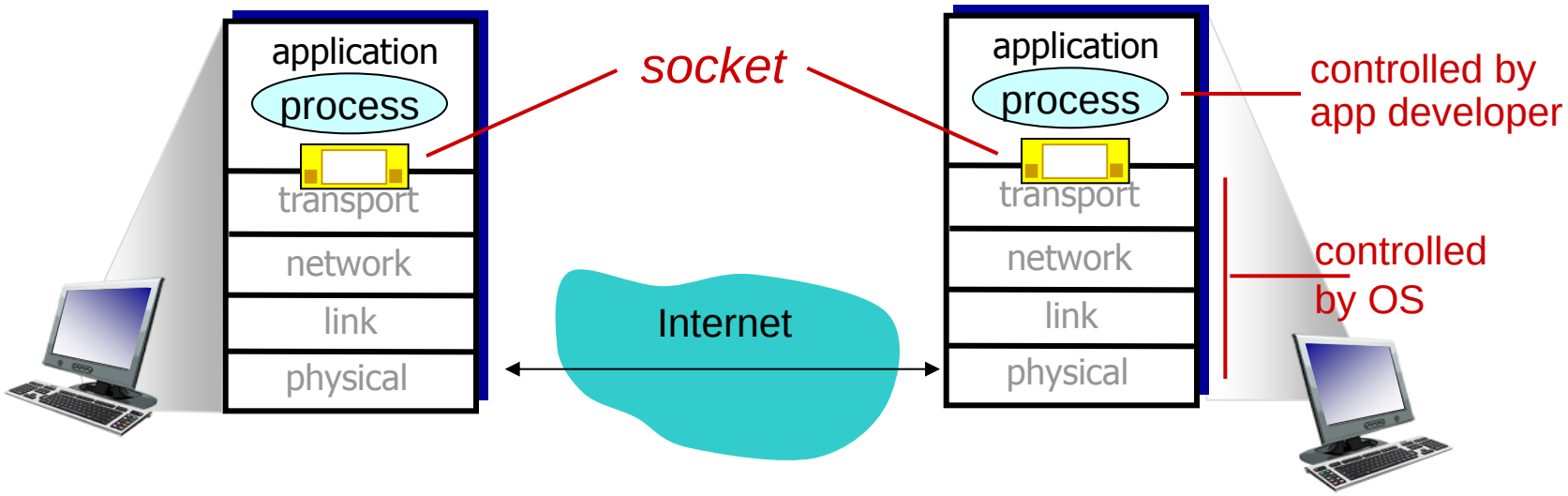
2.6 แอพพลิเคชัน P2P

2.7 socket programming กับ UDP และ TCP

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP:* unreliable datagram
- *TCP:* reliable, byte stream-oriented

Application Example:

1. Client reads a line of characters (data) from its keyboard and sends the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

Socket programming *with UDP*

UDP: no "connection" between client & server

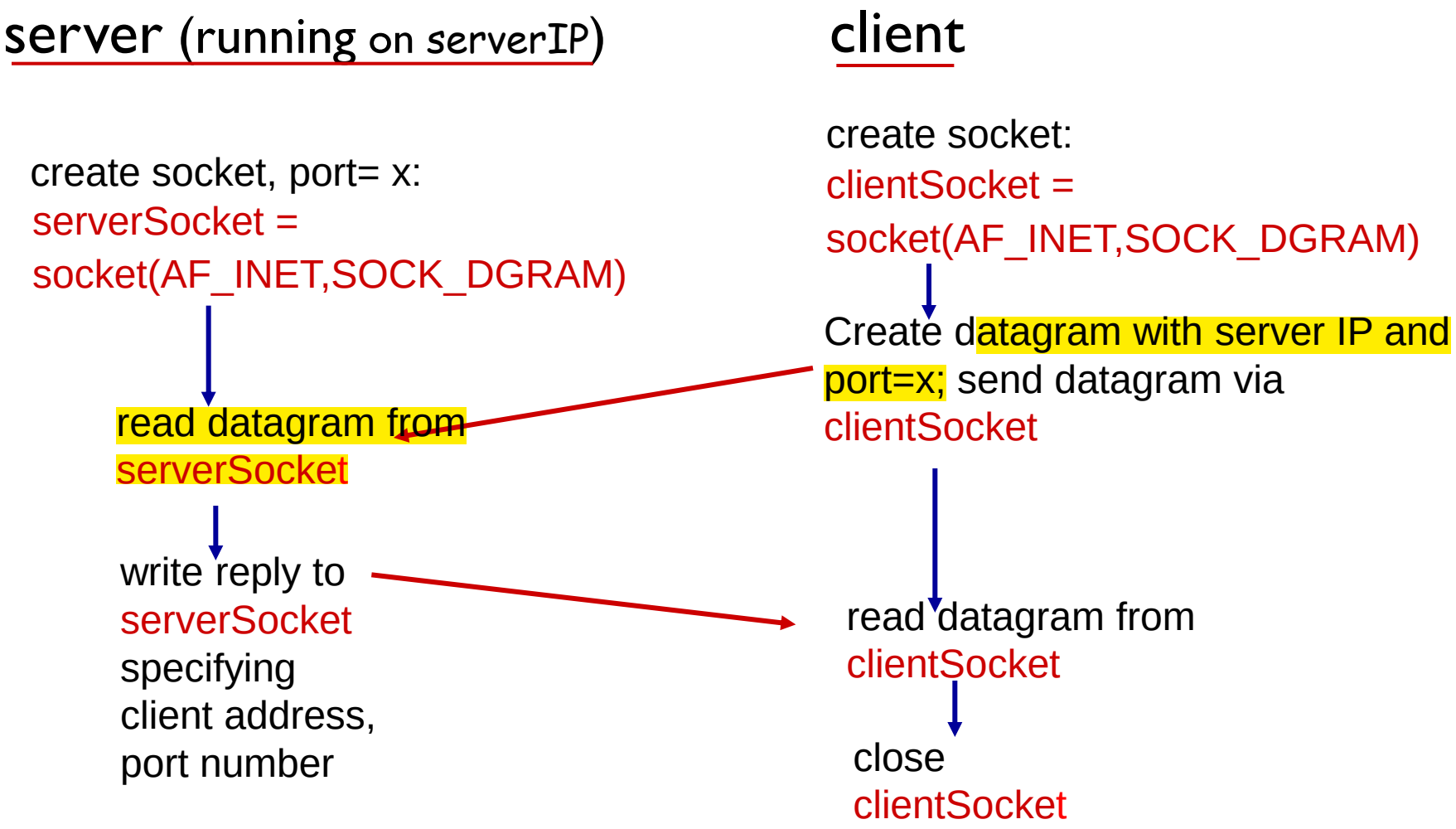
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example: Java client (UDP)

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

class UDPClient {
   public static void main(String args[]) throws Exception
   {
      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      DatagramSocket clientSocket = new DatagramSocket();

      InetAddress IPAddress = InetAddress.getByName("hostname");

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

      String sentence = inFromUser.readLine();

      sendData = sentence.getBytes();
```

Create 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

Send datagram to server

Read datagram from server

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

                clientSocket.send(sendPacket);

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

                clientSocket.receive(receivePacket);

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

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

            }
```

Example: Java server (UDP)

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

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

      DatagramSocket serverSocket = new DatagramSocket(9876);

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

      while(true)
       {

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

        serverSocket.receive(receivePacket);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

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

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

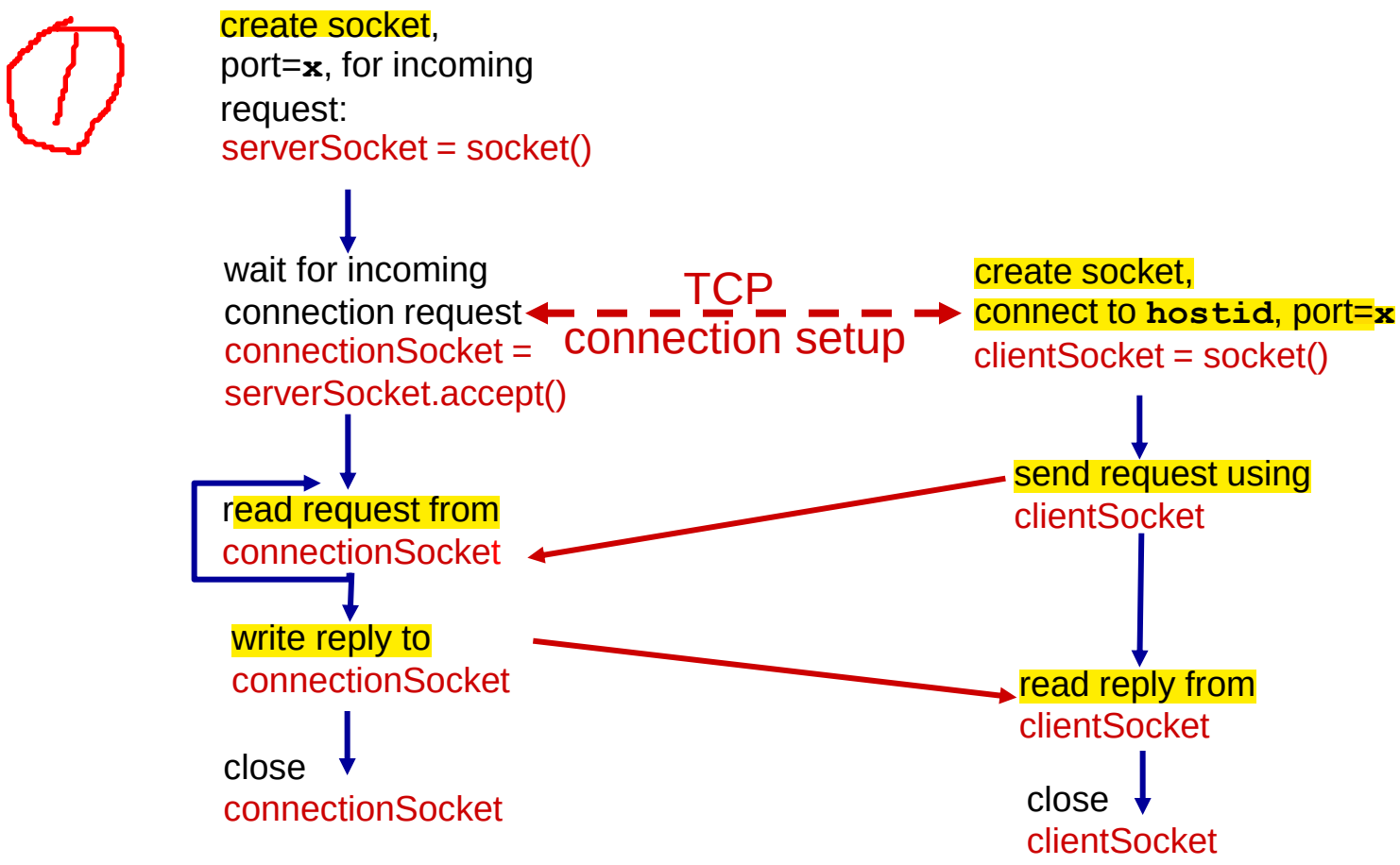
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint:

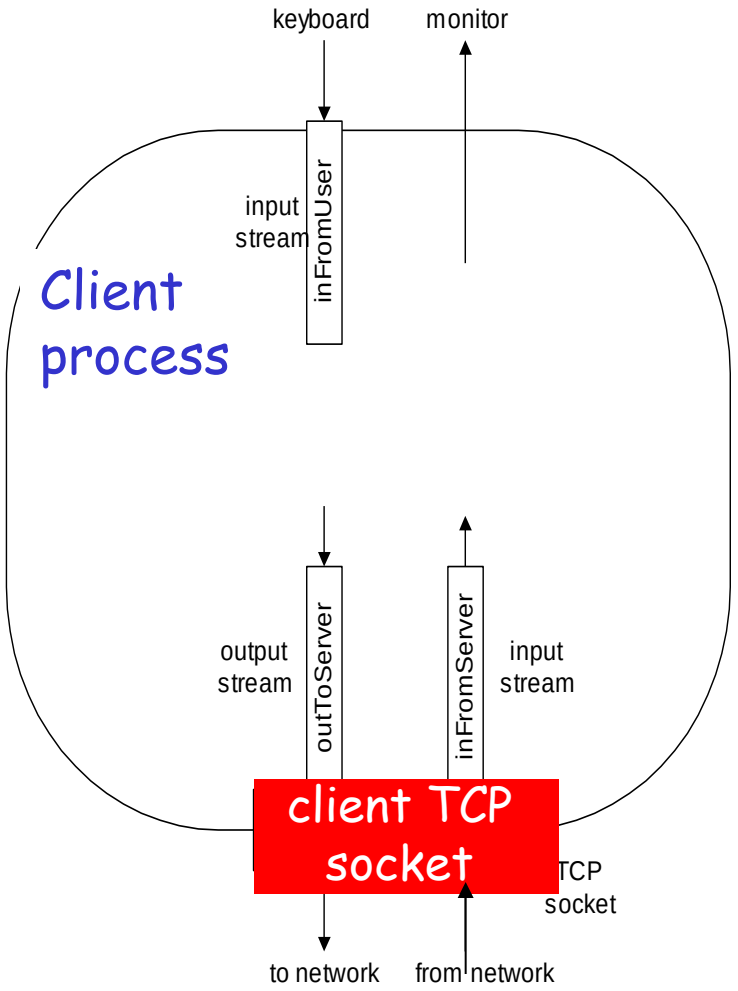
TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



Stream jargon

- 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.



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

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

Create input stream attached to socket

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

sentence = inFromUser.readLine();

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

modifiedSentence = inFromServer.readLine();

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

clientSocket.close();

    }
}
```

Send line to server

Read line from server

Example: Java server (TCP)

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

class TCPServer {

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

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

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

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

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: summary

ตอนนี้เราก็เรียนเกี่ยวกับ network apps สมบูรณ์แล้ว!

- สถาปัตยกรรมของ application
  - client-server
  - P2P
- บริการที่ application ต้องการ:
  - reliability, bandwidth, delay
- โมเดลการให้บริการส่งข้อมูลของ Internet (Internet transport service model)
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- ตัวอย่าง Protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- socket programming: TCP, UDP sockets

Chapter 2: summary

สำคัญ: เรียนรู้เกี่ยวกับ protocols!

- การแลกเปลี่ยนข้อความร้องขอ/ตอบกลับ (request/reply) ทั่ว ๆ ไป:
  - client ร้องขอข้อมูลหรือบริการ
  - server ตอบกลับด้วยข้อมูล หรือ รหัสสถานะ (status code)
- formats ของข้อความ:
  - headers (ส่วนหัว): fields (ส่วนของข้อมูล) ให้ข้อมูลที่อธิบายข้อมูลที่ app ส่งจริง ๆ
  - data: ข้อมูลที่ app จะต้องการส่งถึงกัน

หัวข้อที่สำคัญ:

- ข้อความที่ใช้ควบคุม vs ข้อความที่เป็นข้อมูล
  - in-band (ถูกส่งไปด้วยกัน), out-of-band (ถูกส่งไปคนละการเชื่อมต่อกัน)
- centralized (รวมศูนย์) vs. decentralized (กระจาย)
- stateless (ไม่เก็บสถานะ) vs. stateful (เก็บสถานะ)
- reliable vs. unreliable msg transfer
- ความซับซ้อนที่ขอบของเครือข่าย

Credit ผู้แปล

1	56910040	MS.VANNAK SOTH
2	56920001	นางสาวเจฬุรีย์ ล้ำเลิศ
3	56920003	นายธนศักดิ์ วุฒิวโรภาส
4	56920004	นายนพปฎล เฉยศิริ
5	56920005	นายปรเมศวร์ รัตนผล
6	56920006	พันตรีพรภิรมย์ มั่นฤกษ์
7	56920007	นายวันปิยะ รัตตะมณี
8	56920336	นายฉัตรชัย เสกประเสริฐ
9	56920337	นายธนพนธ์ เดชจิระกุล
10	56920338	นายธนินทร์ เมธิโยธิน
11	56920340	นางสาวพรพรรณ ขวัญกิจบรรจง
12	56920341	นายพัสกร ปัญญวรากิจ
13	56920343	นางโพธิรัตน์ หิรัญรุ่ง
14	56920344	นายรักชาติ เหมะสิขัณฑกะ
15	56920345	นายสมบูรณ์ เฉลิมรัตนาพร
16	56920347	นายเอกพล อ่อนปาน