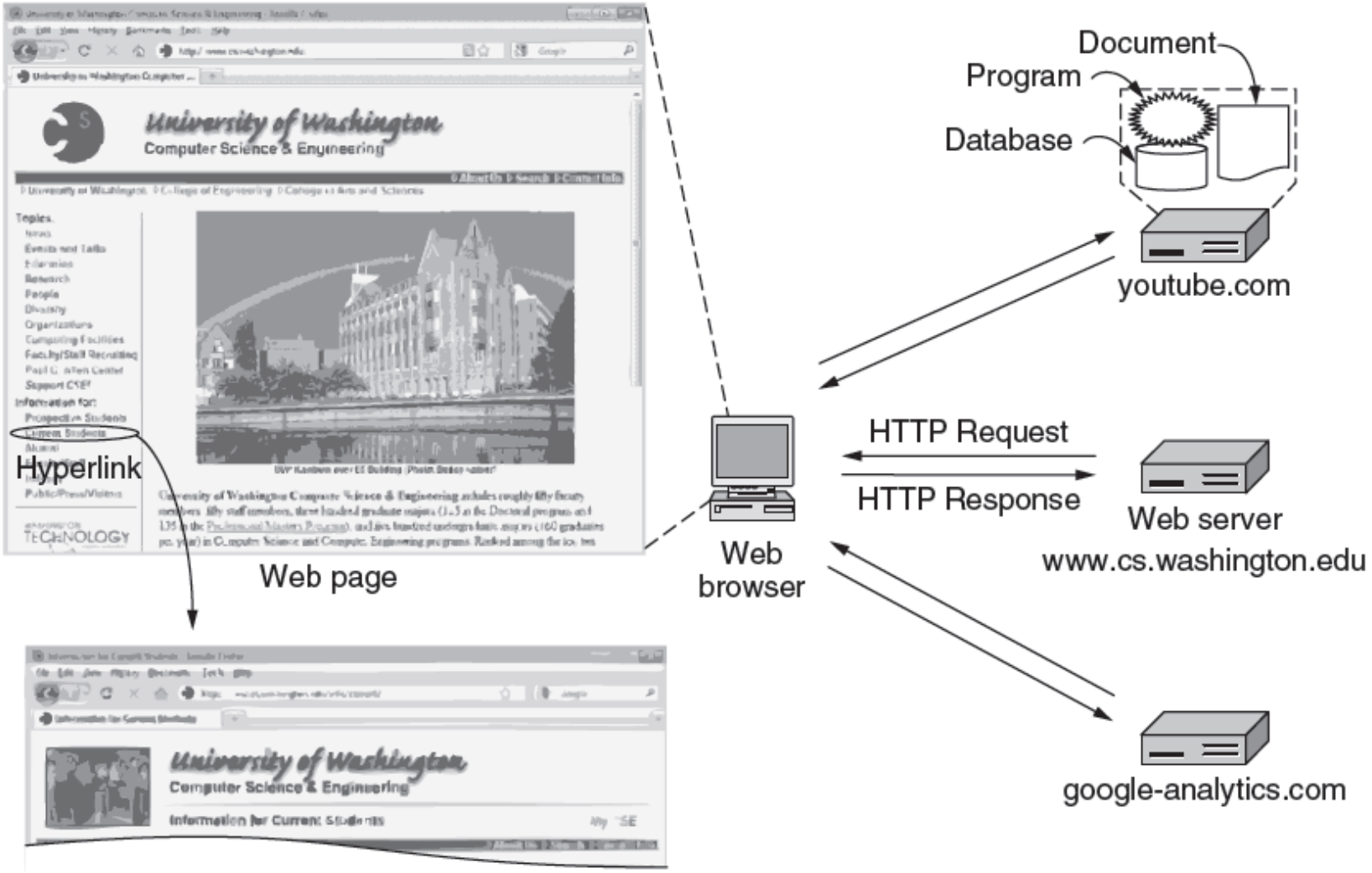
Couche application

- Présentation générale:
  - Modèle des services de la couche transport
  - Modèle client-serveur et Modèle pair-à-pair
- Socket: UDP TCP
- Protocoles
  - Web HTTP
  - FTP
  - SMTP / POP3 / IMAP
  - DNS
- Résaux Pair à pair

Http: Principes



H. Fauconnier

Web et HTTP

- *Une page web page* contient des *objets*
- Objet : fichier HTML , images JPEG, applet, fichiers audio,…
- Une page web page consiste en un fichier de base *HTML* contenant des objets référencés
- Chaque est adressable par une *URL (Uniform Resource Locator)*

`www.someschool.edu/someDept/pic.gif`

host name | path name

URL:

- Example: http://www.phdcomics.com/comics.php

Protocol Server Page on server

Our focus

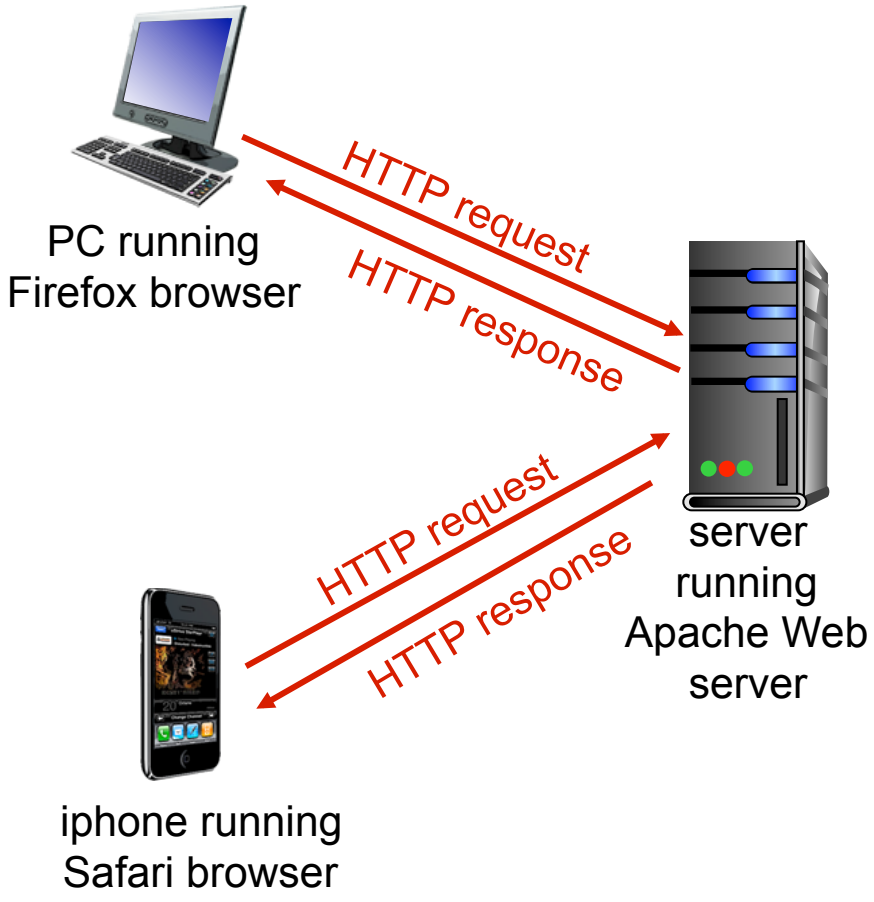
Name	Used for	Example
http	Hypertext (HTML)	http://www.ee.uwa.edu/~rob/
https	Hypertext with security	https://www.bank.com/accounts/
ftp	FTP	ftp://ftp.cs.vu.nl/pub/minix/README
file	Local file	file:///usr/suzanne/prog.c
mailto	Sending email	mailto:JohnUser@acm.org
rtsp	Streaming media	rtsp://youtube.com/montypython.mpg
sip	Multimedia calls	sip:eve@adversary.com
about	Browser information	about:plugins

Common URL protocols

HTTP overview

HTTP: hypertext transfer protocol

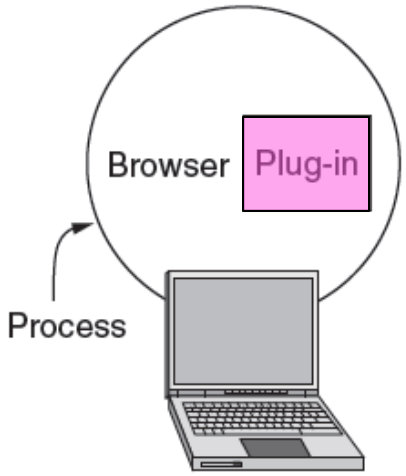
- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests

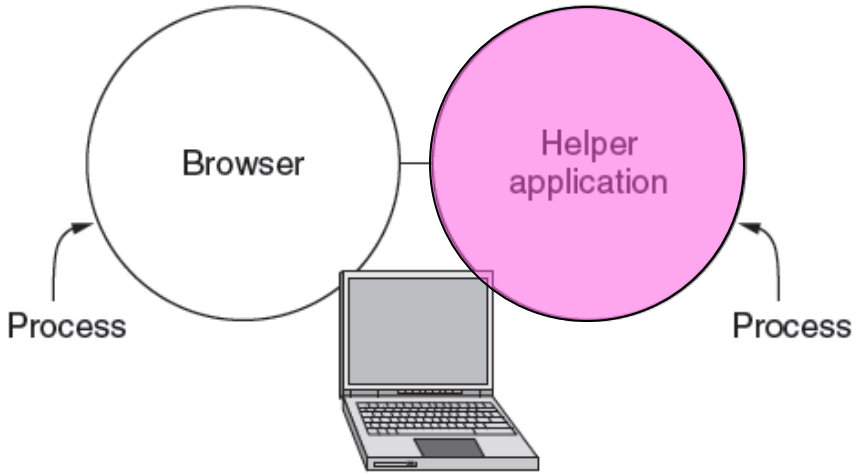


Content type

Content type is identified by MIME types

- Browser takes the appropriate action to display
- Plug-ins / helper apps extend browser for new types





HTTP overview

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

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

Overview

Steps a client (browser) takes to follow a hyperlink:

- Determine the protocol (HTTP)
- Ask DNS for the IP address of server
- Make a TCP connection to server
- Send request for the page; server sends it back
- Fetch other URLs as needed to display the page
- Close idle TCP connections

Steps a server takes to serve pages:

- Accept a TCP connection from client
- Get page request and map it to a resource (e.g., file name)
- Get the resource (e.g., file from disk)
- Send contents of the resource to the client.
- Release idle TCP connections

HTTP connections

non-persistent HTTP

- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

persistent HTTP

- multiple objects can be sent over single TCP connection between client, server

Non-persistent 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

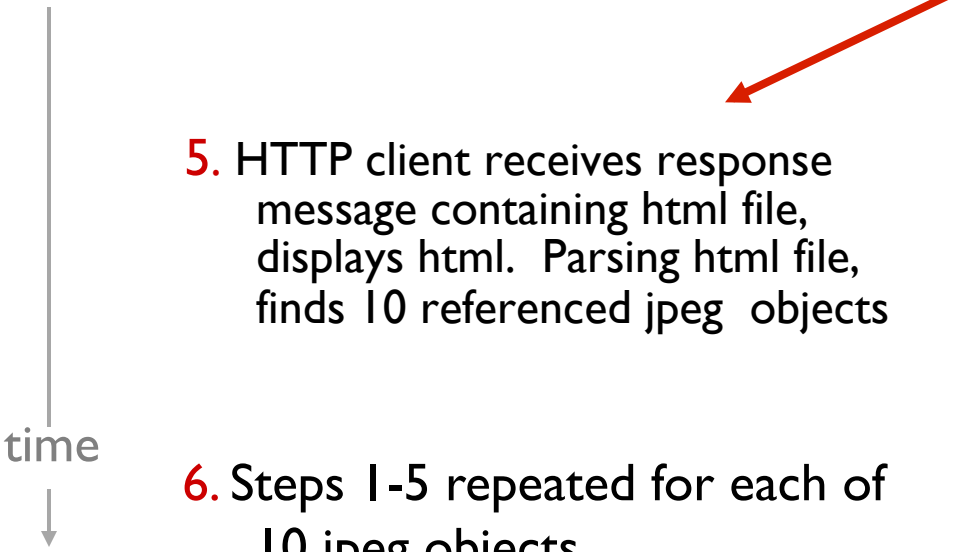
1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

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

Non-persistent 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

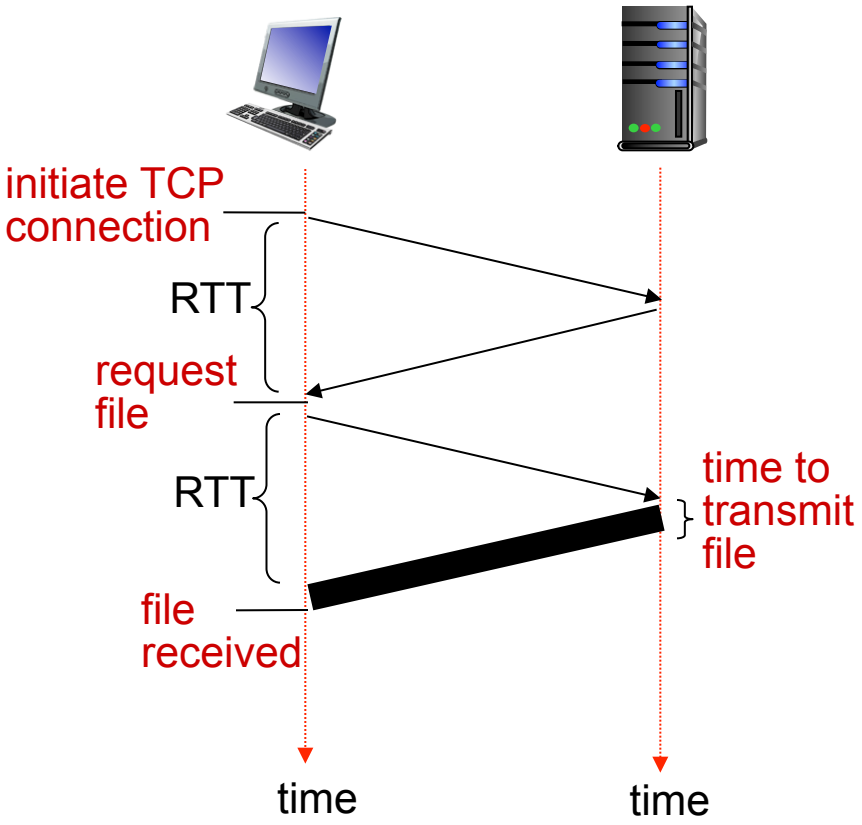
Non-persistent HTTP: response time

RTT (definition): time for a small packet to travel from client to server and back

HTTP 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
- non-persistent HTTP response time =

  2RTT+ file transmission time



Persistent HTTP

non-persistent HTTP issues:

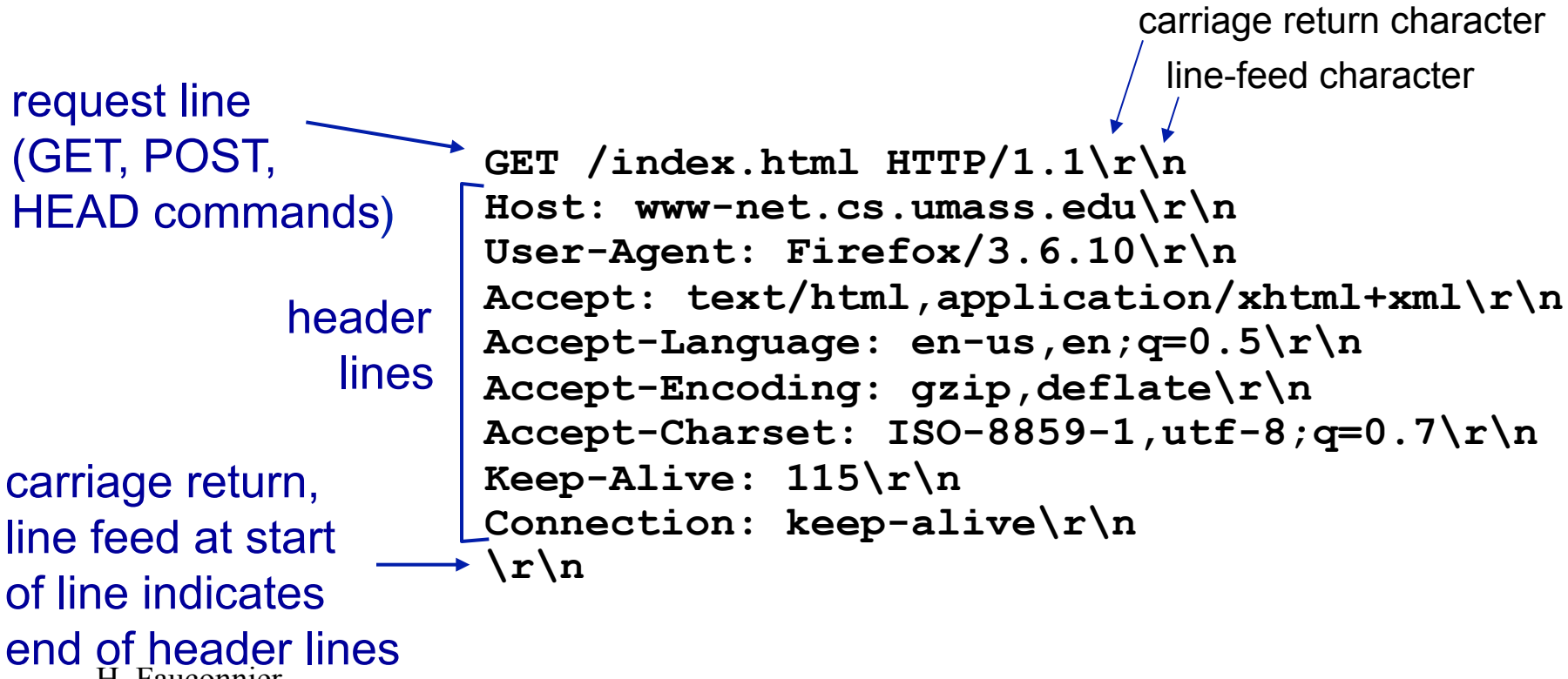
- requires 2 RTTs per object
- OS overhead for *each* TCP connection
- browsers often open parallel TCP connections to fetch referenced objects

persistent HTTP:

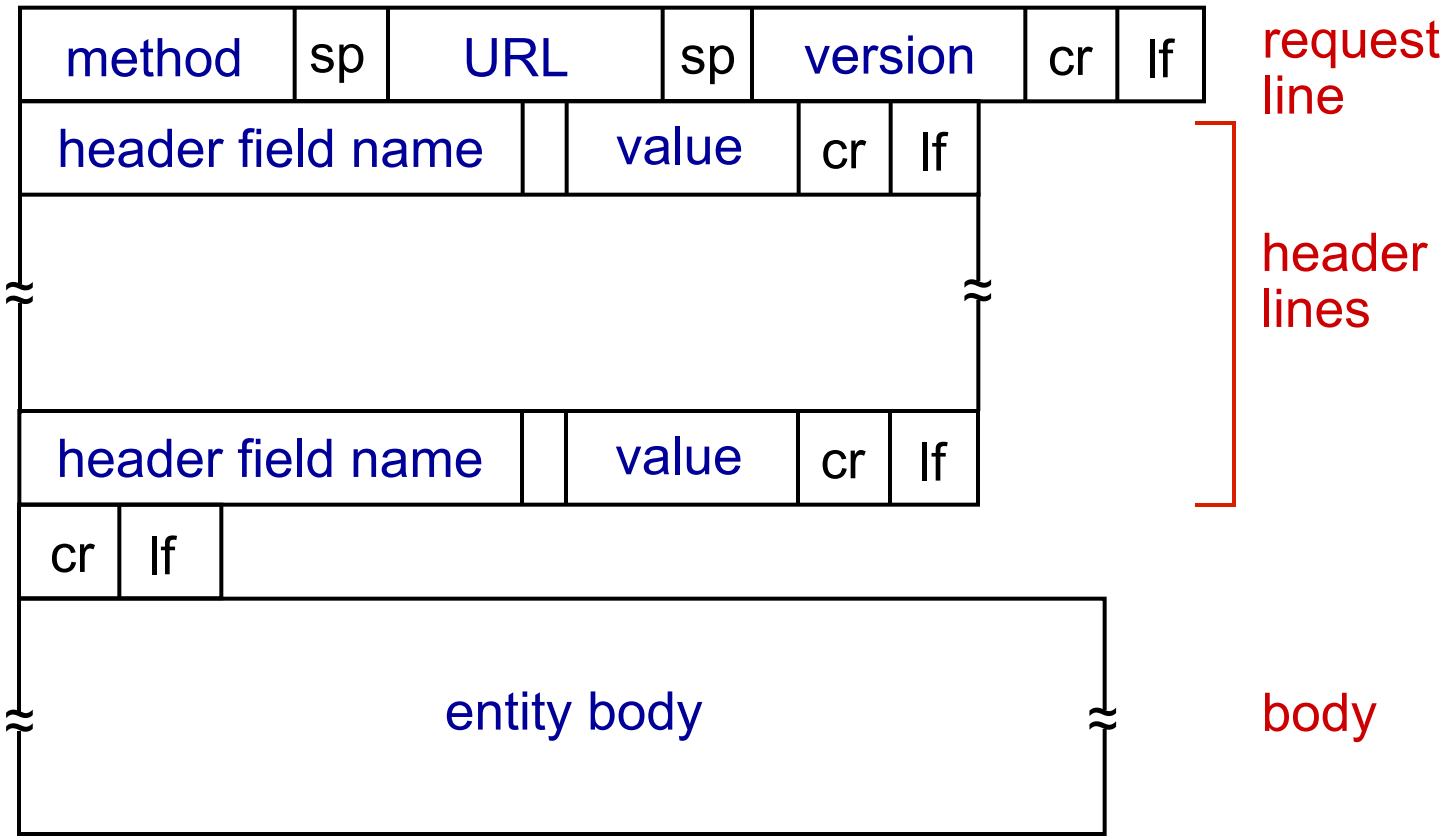
- server leaves connection open after sending response
- subsequent HTTP messages between same client/server sent over open connection
- client sends requests as soon as it encounters a referenced object
- as little as one RTT for all the referenced objects

HTTP request message

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



HTTP request message: general format



HTTP

Headers:

Function	**Example Headers**
Browser capabilities (client → server)	User-Agent, Accept, Accept-Charset, Accept-Encoding, Accept-Language
Caching related (mixed directions)	If-Modified-Since, If-None-Match, Date, Last-Modified, Expires, Cache-Control, ETag
Browser context (client → server)	Cookie, Referer, Authorization, Host
Content delivery (server → client)	Content-Encoding, Content-Length, Content-Type, Content-Language, Content-Range, Set-Cookie

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
```

Method types

HTTP/1.0:

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

HTTP/1.1:

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

HTTP

Request methods.

Fetch a page →

Used to send input data to a server program →

Method	Description
GET	Read a Web page
HEAD	Read a Web page's header
POST	Append to a Web page
PUT	Store a Web page
DELETE	Remove the Web page
TRACE	Echo the incoming request
CONNECT	Connect through a proxy
OPTIONS	Query options for a page

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
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 ...
```

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
- some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

505 HTTP Version Not Supported

HTTP

Response codes tell the client how the request fared:

Code	Meaning	Examples
1xx	Information	100 = server agrees to handle client's request
2xx	Success	200 = request succeeded; 204 = no content present
3xx	Redirection	301 = page moved; 304 = cached page still valid
4xx	Client error	403 = forbidden page; 404 = page not found
5xx	Server error	500 = internal server error; 503 = try again later

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!

User-server state: cookies

many Web sites use cookies

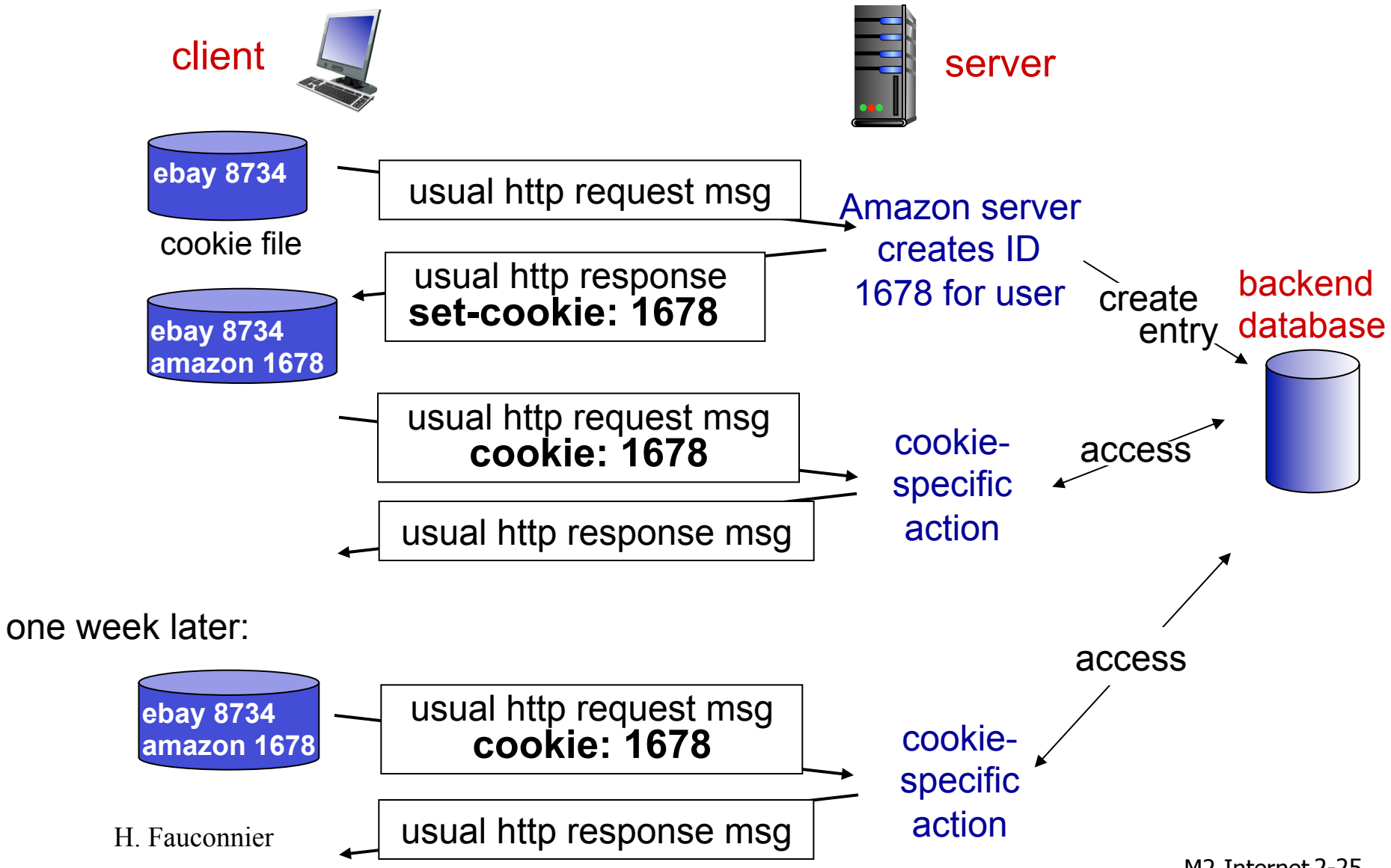
four components:

1) cookie header line of HTTP *response* message
2) cookie header line in next 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 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.)



Cookies (continued)

what cookies can be used for:

- 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

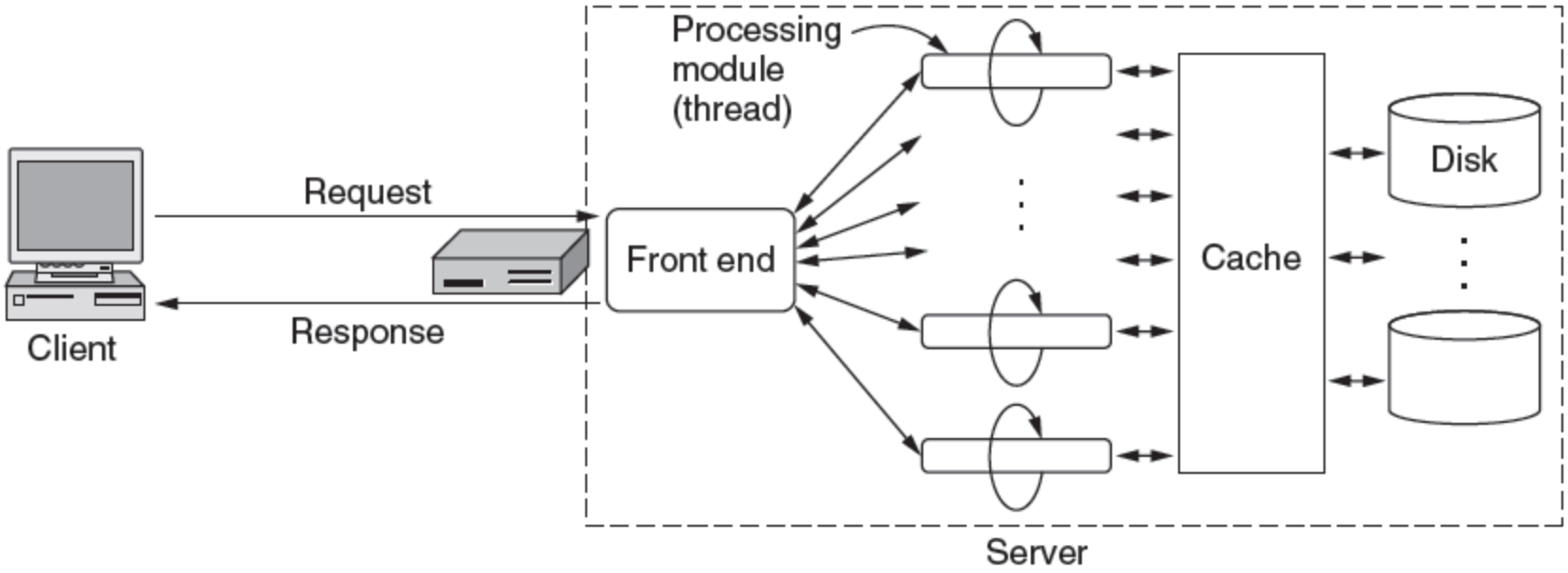
how to keep "state":

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

Caching

To scale performance, Web servers can use:

- Caching, multiple threads, and a front end



Caching…

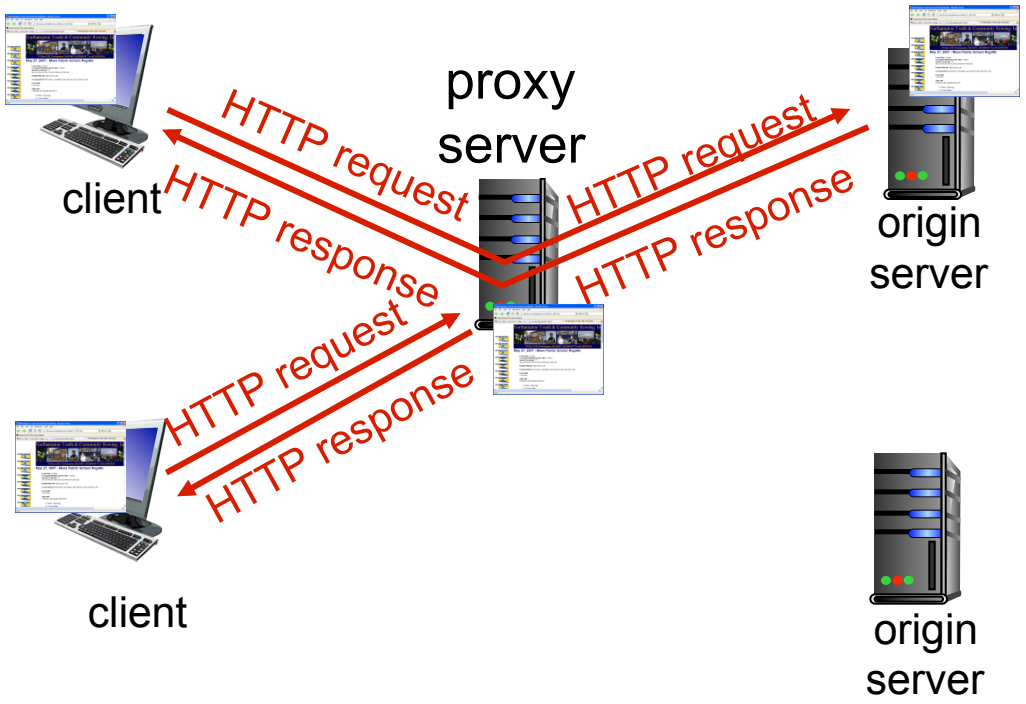
Server steps, revisited:

- Resolve name of Web page requested
- Perform access control on the Web page
- Check the cache
- Fetch requested page from disk or run program
- Determine the rest of the response
- Return the response to the client
- Make an entry in the server log

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
  - server for original requesting client
  - client to origin 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 (so too does P2P file sharing)

Caching example:

assumptions:

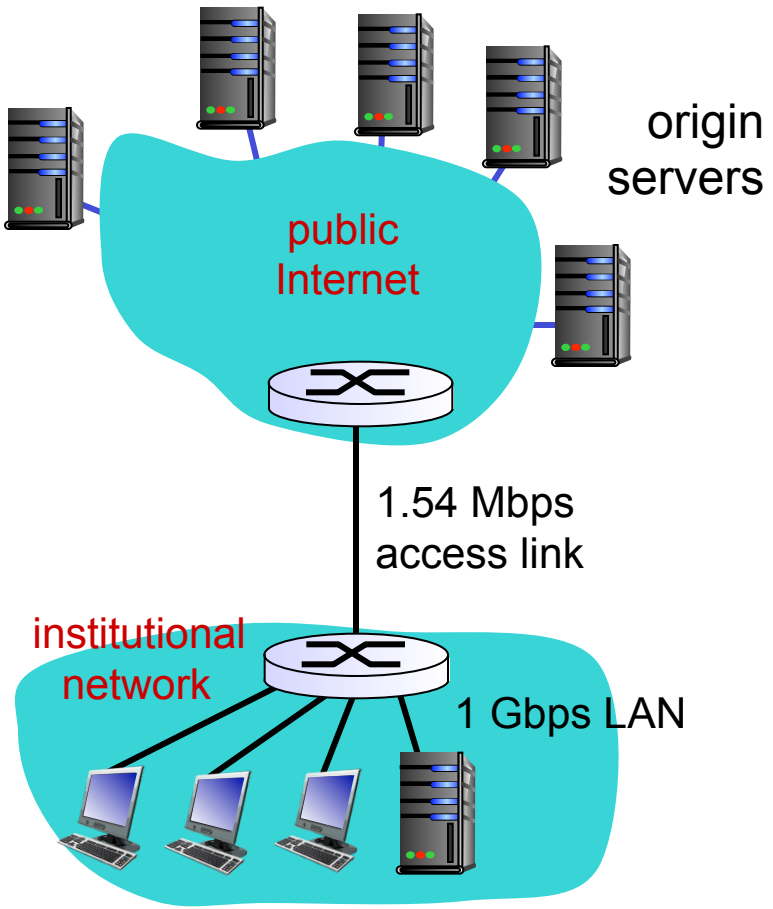
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

- LAN utilization: 15%
- access link utilization = 99%
- total delay = Internet delay + access delay + LAN delay

  = 2 sec + minutes + usecs

problem!



Caching example: fatter access link

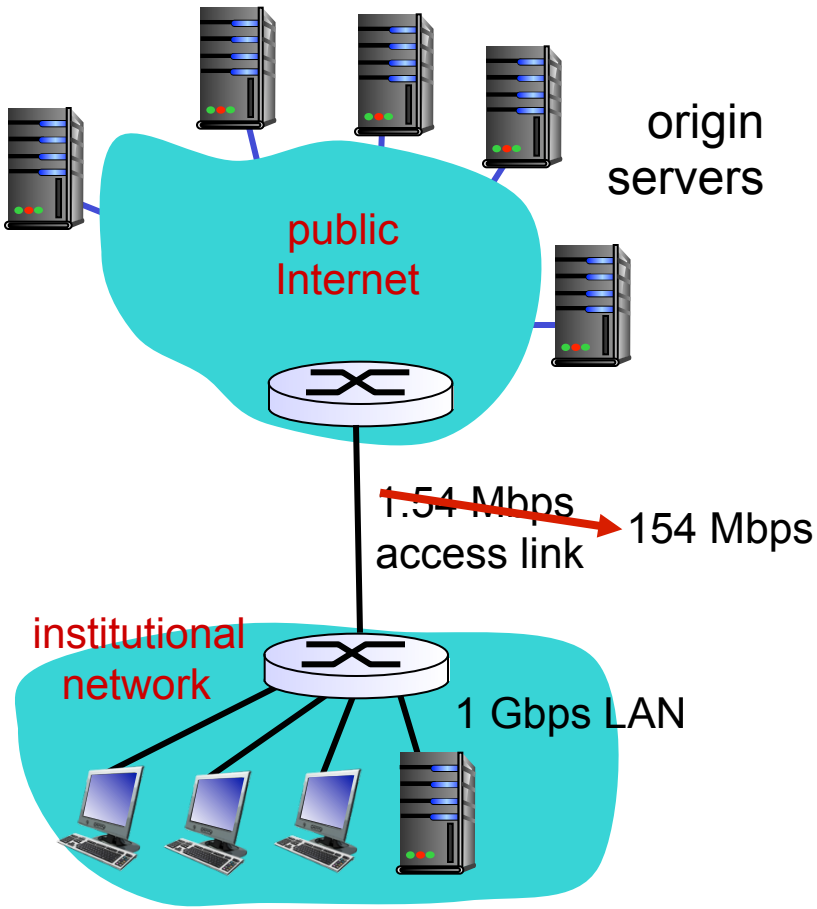
assumptions:

- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: ~~1.54 Mbps~~ 154 Mbps

consequences:

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

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



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

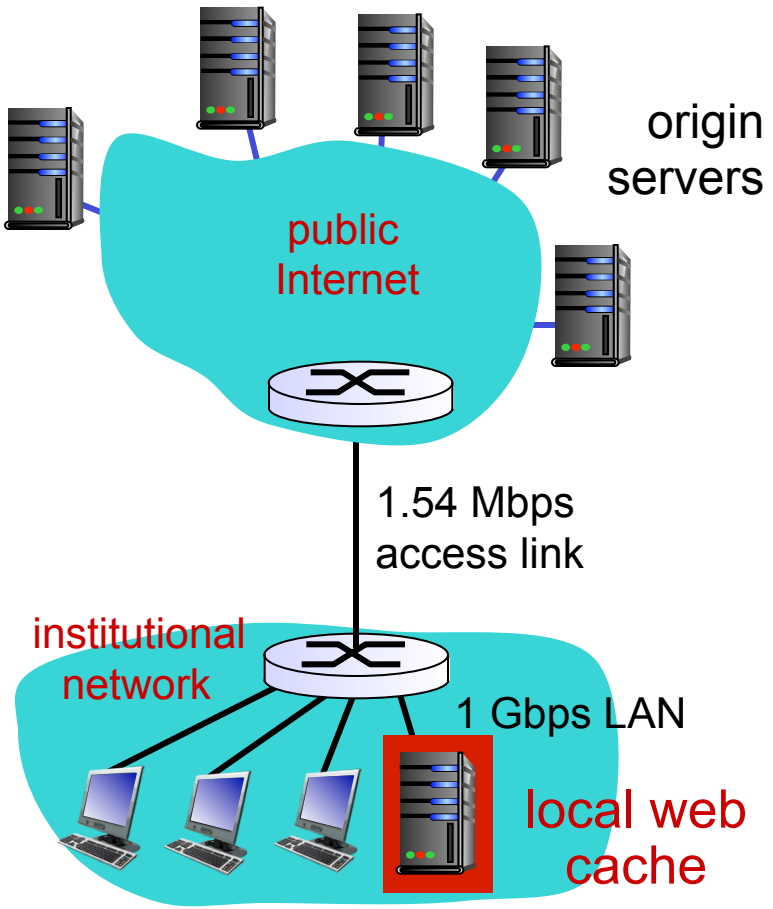
- avg object size: 100K bits
- avg request rate from browsers to origin servers:15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

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

How to compute link utilization, delay?

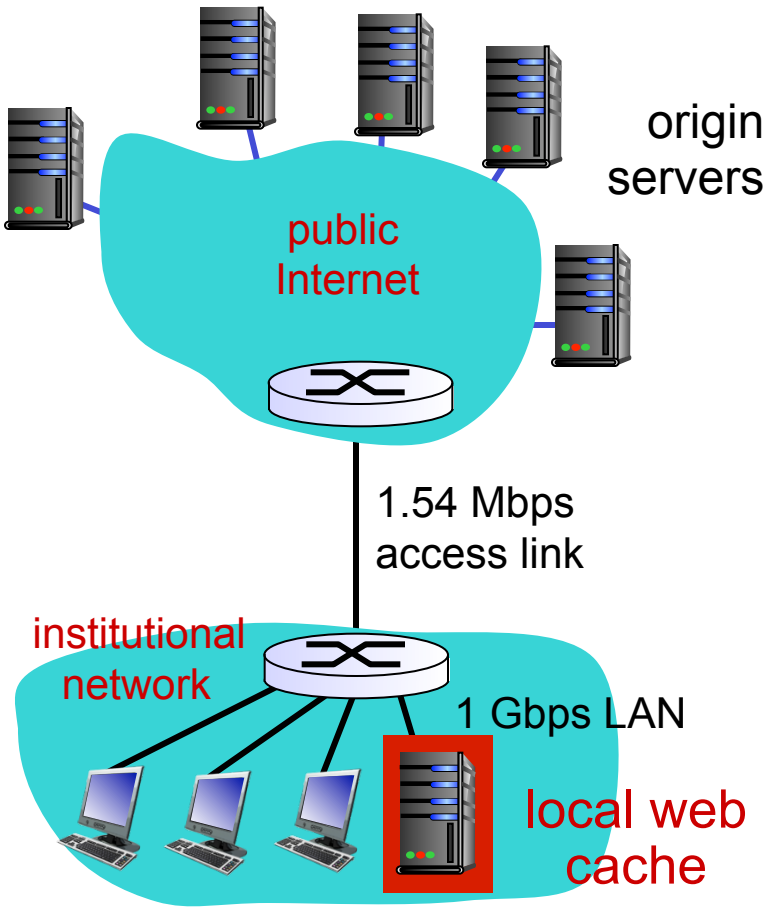
Cost: web cache (cheap!)



Caching example: install local cache

Calculating access link utilization, delay with cache:

- suppose cache hit rate is 0.4
  - 40% requests satisfied at cache, 60% requests satisfied at origin
- access link utilization:
  - 60% of requests use access link
- data rate to browsers over 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
  - less than with 154 Mbps link (and cheaper too!)

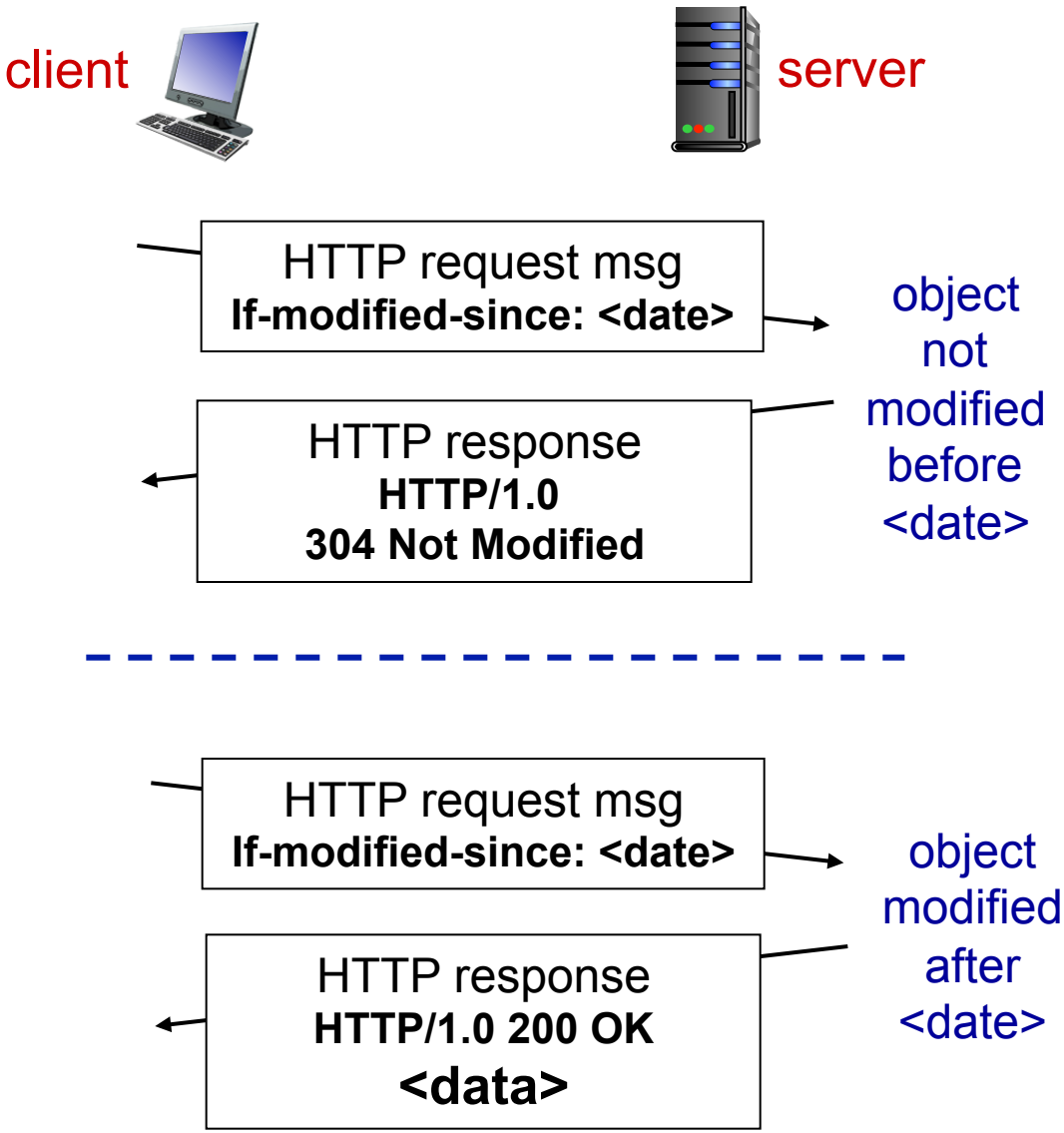


Conditional GET

- *Goal:* don't send object if cache has up-to-date cached version
  - no object transmission delay
  - lower link utilization
- *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`



Static Web Pages

Static Web pages are simply files

- Have the same contents for each viewing

Can be visually rich and interactive nonetheless:

- HTML that mixes text and images
- Forms that gather user input
- Style sheets that tailor presentation
- Vector graphics, videos, and more (over) . . .

Static Web Pages

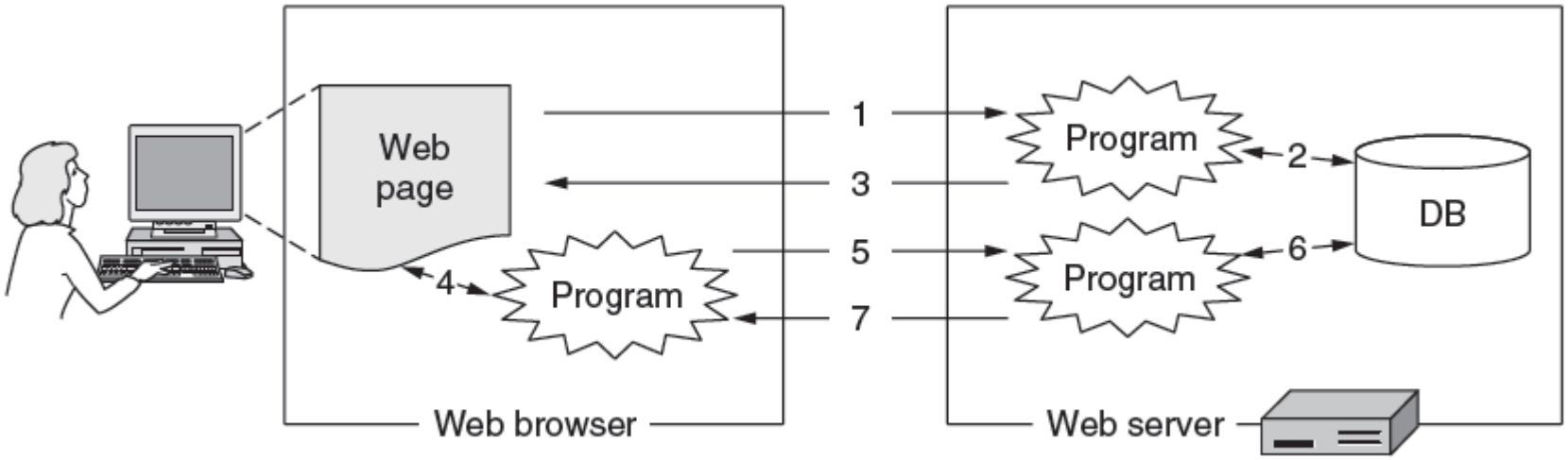
Progression of features through HTML 5.0

Item	HTML 1.0	HTML 2.0	HTML 3.0	HTML 4.0	HTML 5.0
Hyperlinks	x	x	x	x	x
Images	x	x	x	x	x
Lists	x	x	x	x	x
Active maps & images		x	x	x	x
Forms		x	x	x	x
Equations			x	x	x
Toolbars			x	x	x
Tables			x	x	x
Accessibility features				x	x
Object embedding				x	x
Style sheets				x	x
Scripting				x	x
Video and audio					x
Inline vector graphics					x
XML representation					x
Background threads					x
Browser storage					x
Drawing canvas					x

Dynamic Pages & Web Applications

Dynamic pages are generated by programs running at the server (with a database) and the client

- E.g., PHP at server, JavaScript at client
- Pages vary each time like using an application



Dynamic Pages & Web Applications

Web page that gets form input and calls a server program

```
<html>
<body>
<form action="action.php" method="post">
<p> Please enter your name: <input type="text" name="name"> </p>
<p> Please enter your age: <input type="text" name="age"> </p>
<input type="submit">
</form>
</body>
</html>
```

PHP server program that creates a custom Web page

PHP calls

```
<html>
<body>
<h1> Reply: </h1>
Hello <?php echo $name; ?>.
Prediction: next year you will be <?php echo $age + 1; ?>
</body>
</html>
```

Resulting Web page (for inputs “Barbara” and “32”)

```
<html>
<body>
<h1> Reply: </h1>
Hello Barbara.
Prediction: next year you will be 33
</body>
</html>
```

Dynamic Pages…

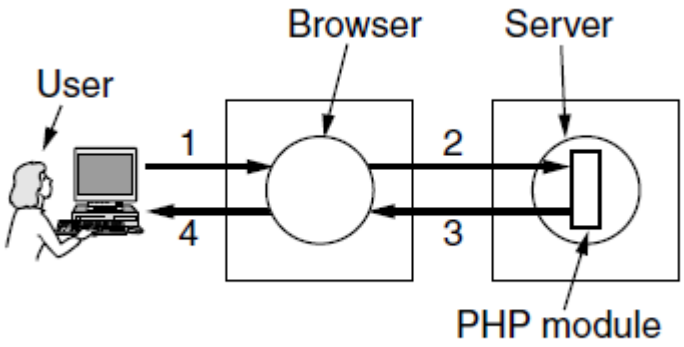
JavaScript program produces result page in the browser

First page with form, gets input and calls program above

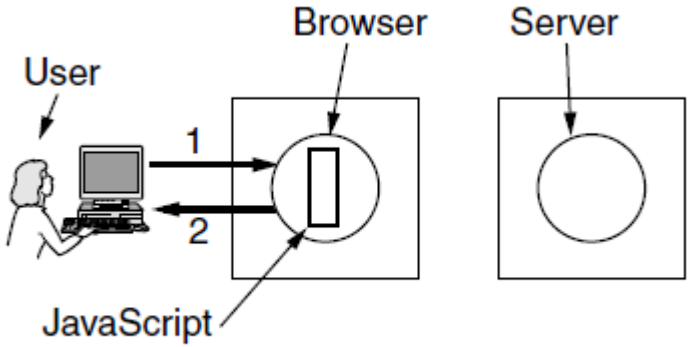
```
<html>
<head>
<script language="javascript" type="text/javascript">
function response(test_form) {
   var person = test_form.name.value;
   var years = eval(test_form.age.value) + 1;
   document.open();
   document.writeln("<html> <body>");
   document.writeln("Hello " + person + ".<br>");
   document.writeln("Prediction: next year you will be " + years + ".");
   document.writeln("</body> </html>");
   document.close();
}
</script>
</head>

<body>
<form>
Please enter your name: <input type="text" name="name">
<p>
Please enter your age: <input type="text" name="age">
<p>
<input type="button" value="submit" onclick="response(this.form)">
</form>
</body>
</html>
```

Dynamic Pages & Web Applications

The difference between server and client programs



Server-side scripting with PHP



Client-side scripting with JavaScript

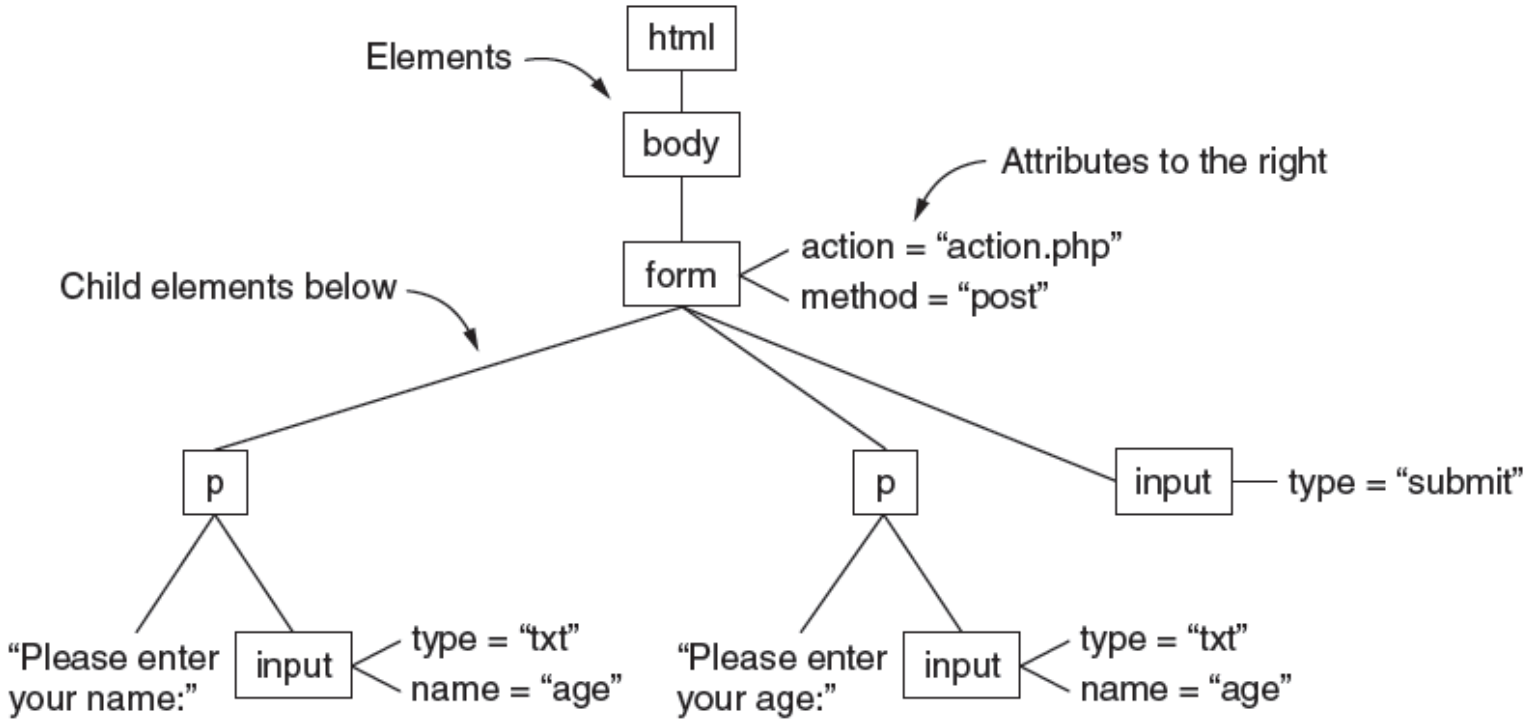
Dynamic Pages & Web Applications

Web applications use a set of technologies that work together, e.g. AJAX:

- HTML: present information as pages.
- DOM: change parts of pages while they are viewed.
- XML: let programs exchange data with the server.
- Asynchronous way to send and retrieve XML data.
- JavaScript as a language to bind all this together.

Dynamic Pages & Web Applications

The DOM (Document Object Model) tree represents Web pages as a structure that programs can alter



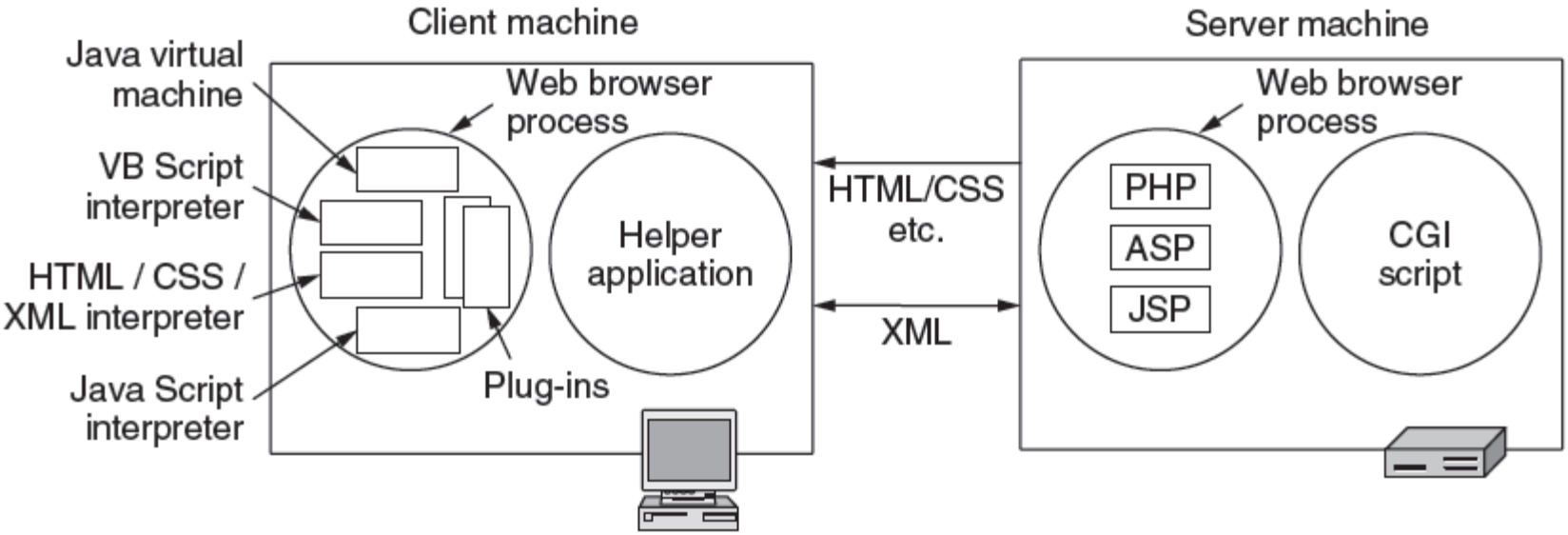
Dynamic Pages & Web Applications

XML captures document structure, not presentation like HTML. Ex:

```
<?xml version="1.0" ?>

<book_list>

<book>
   <title> Human Behavior and the Principle of Least Effort </title>
   <author> George Zipf </author>
   <year> 1949 </year>
</book>

<book>
   <title> The Mathematical Theory of Communication </title>
   <author> Claude E. Shannon </author>
   <author> Warren Weaver </author>
   <year> 1949 </year>
</book>

<book>
   <title> Nineteen Eighty-Four </title>
   <author> George Orwell </author>
   <year> 1949 </year>
</book>

</book_list>
```

Dynamic Pages & Web Applications

Web applications use a set of technologies, revisited:



The Mobile Web

Mobiles (phones, tablets) are challenging as clients:

- Relatively small screens
- Limited input capabilities, lengthy input.
- Network bandwidth is limited
- Connectivity may be intermittent.
- Computing power is limited

Strategies to handle them:

- Content: servers provide mobile-friendly versions; transcoding can also be used
- Protocols: no real need for specialized protocols; HTTP with header compression sufficient

apache

- Pour pouvoir tester les exemples on utilisera apache
  - logiciel libre disponible sur la plupart des plateformes
  - Le serveur le plus fréquent
  - Prise en charge de nombreux modules, (perl php, python, ruby…) cgi
  - Serveurs virtuels

Principes…

- Le serveur reçoit des requêtes http et renvoie des pages html dans des réponses http
  - Interpréter les requêtes
  - Lancer sur le côté serveur les applications concernées
  - Récupérer les résultats et les transmettre au client
- Configuration: httpd.conf (en général dans /etc (et par catalogue .htaccess)

Principes

- Correspondance entre url et fichiers locaux
  - DocumentRoot:
    - Si DocumentRoot = /var/www/html
    - http://www.exemple.com/un/deux.html sera converti en
    - /var/www/html/un/deux.html
  - Pages des uitlisateurs
    - UserDir: www
    - http://www.example.com/~user/file.html sera
    - /home/user/public_html/file.html
  - En plus des alias et des redirections

Pour voir…

- Sur cette machine:
  - /etc/apache2/httpd.conf
  - ServerRoot: /usr
  - DocumentRoot: /Library/WebServer/Documents
  - + fichier de configurations dans « extra »
    - UserDir: Sites

CGI

- Common Gateway Interface
- exécuter du code du côté serveur
- Passage de paramètre par la méthode POST ou la méthode GET
- Variables d'environnement

Pour Apache

- Les executables cgi (dépendant)
  - ScriptAlias /cgi-bin/ /usr/local/apache2/cgi-bin/
  - Pour http://www.example.com/cgi-bin/test.pl
  - /usr/local/apache2/cgi-bin/test.pl sera exécuté
- "Paramètres"
  - POST: transmis sur l'entrée standard (STDIN)
  - GET: variable de l'environnement QUERY_STRING
- STDOUT pour la réponse
  - (au moins un MIME type header
    - Content-type: text/html et deux newline)

Exemple

- en shell: date.cgi

```
#!/bin/sh
tmp=`/bin/date`
echo "Content-type: text/html\n
<HTML><HEAD><TITLE>Script Cgi</TITLE></
   HEAD><BODY>
<CENTER>
<H1>La date courante sur le serveur est</H1> $tmp
</CENTER> </BODY> </HTML>"
```

l'URL affichera la date

http://localhost/cgi-bin/date.cgi

Avec un formulaire:

```
<HTML><HEAD><TITLE>Formulaire simple</TITLE></HEAD>
<BODY>
<H2>Répondez aux questions suivantes</H2>
<FORM ACTION="http://www.monsite.com/cgi-bin/treat.pl"
   METHOD=GET>
Prénom : <INPUT TYPE="text" NAME=prenom SIZE=20><BR>
Nom : <INPUT TYPE="text" NAME=nom SIZE=20><BR>
Age : <SELECT NAME=age>
        <OPTION>- de 18 ans
        <OPTION>19 à 40 ans
        <OPTION>41 à 60 ans
        <OPTION>+ de 60 ans
</SELECT><BR>
<INPUT TYPE=submit VALUE="Envoyer"> <INPUT TYPE=reset
   VALUE="Remettre
à zéro">
</FORM>
</BODY>
```

http://localhost/~hf/formulaire.html

Résultat

- par la méthode get codage des paramètres:
- prenom=Hugues&nom=Fauconnier&age=41+%E0+60+ans
- le navigateur génère l'url:

```
http://www.monsite.com/cgi-bin/treat.pl?
  prenom=Hugues&nom=Fauconnier&age=41+%E0+60+ans
```

- Avec la méthode POST

```
http://www.monsite.com/cgi-bin/treat.pl
prenom=Hugues&nom=Fauconnier&age=41
```

Traitement en perl

- fichier perl

Paramètres

- Les paramètres sont accessibles par l'intermédiaire de la variable d'environnement QUERY_STRING

Variables d'environnement

- SERVER_SOFTWARE
  - Le nom et la version du serveur HTTP répondant à la requête. (Format : nom/version)
- SERVER_NAME
  - Le nom d'hôte, alias DNS ou adresse IP du serveur.
- GATEWAY_INTERFACE
  - La révision de la spécification CGI que le serveur utilise. (Format : CGI/révision)

Variables…

- SERVER_PROTOCOL
  - Le nom et la révision du protocole dans lequel la requête a été faite (Format : protocole/révision)
- SERVER_PORT
  - Le numéro de port sur lequel la requête a été envoyée.
- REQUEST_METHOD
  - La méthode utilisée pour faire la requête. Pour HTTP, elle contient généralement « GET » ou « POST ».
- PATH_INFO
  - Le chemin supplémentaire du script tel que donné par le client. Par exemple, si le serveur héberge le script « /cgi-bin/monscript.cgi » et que le client demande l'url « http://serveur.org/cgi-bin/monscript.cgi/marecherche », alors PATH_INFO contiendra « marecherche ».
- PATH_TRANSLATED
  - Contient le chemin demandé par le client après que les conversions virtuel → physique aient été faites par le serveur.

Variables

- SCRIPT_NAME
  - Le chemin virtuel vers le script étant exécuté. Exemple : « /cgi-bin/script.cgi »
- QUERY_STRING
  - Contient tout ce qui suit le « ? » dans l'URL envoyée par le client. Toutes les variables provenant d'un formulaire envoyé avec la méthode « GET » sera contenue dans le QUERY_STRING sous la forme « var1=val1&var2=val2&... ».
- REMOTE_HOST
  - Le nom d'hôte du client. Si le serveur ne possède pas cette information (par exemple, lorsque la résolution DNS inverse est désactivée), REMOTE_HOST sera vide.
- REMOTE_ADDR
  - L'adresse IP du client.
- AUTH_TYPE
  - Le type d'identification utilisé pour protéger le script (s'il est protégé et si le serveur supporte l'identification).

Variables

- AUTH_TYPE
  - Le type d'identification utilisé pour protéger le script (s'il est protégé et si le serveur supporte l'identification).
- REMOTE_USER
  - Le nom d'utilisateur du client, si le script est protégé et si le serveur supporte l'identification.
- REMOTE_IDENT
  - Nom d'utilisateur (distant) du client faisant la requête. Le serveur doit supporter l'identification RFC 931. Cette variable devraient être utilisée à des fins de journaux seulement.
- CONTENT_TYPE
  - Le type de contenu attaché à la requête, si des données sont attachées (comme lorsqu'un formulaire est envoyé avec la méthode « POST »).
- CONTENT_LENGTH
  - La longueur du contenu envoyé par le client.

Variables

- HTTP_ACCEPT
  - Les types de données MIME que le client accepte de recevoir.
  - Exemple : text/*, image/jpeg, image/png, image/*, */*
- HTTP_ACCEPT_LANGUAGE
  - Les langages dans lequel le client accepte de recevoir la réponse.
  - Exemple : fr_CA, fr
- HTTP_USER_AGENT
  - Le navigateur utilisé par le client.
  - Exemple : Mozilla/5.0 (compatible; Konqueror/3; Linux)

Compléments Javascript

- Code qui s'exécute du côté du client
  - calcul local
  - contrôle d'une zone de saisie
  - affichage d'alerte
  - fenêtres menus etc..
- Balise :

  <SCRIPT languge="JavaScript1.2">

  le code...

  </SCRIPT>

Exemple: bonjour

```
<HTML><HEAD>
<TITLE>Très facile</TITLE>
</HEAD>

<SCRIPT language="JavaScript1.2">
 function bonjour()
 {
    alert ("Bonjour madame, bonjour monsieur");
 }
</SCRIPT>

<BODY bgcolor="WHITE" onLoad="bonjour();">
     <H1>Bonjour</H1>

</BODY></HTML>
```

http://localhost/~hf/BjrJvs.html

Un peu plus: minicalcul

```
<HTML>
<HEAD>
<TITLE>Petit calcul</TITLE>
</HEAD>
<BODY bgcolor='WHITE'>

<script language='JavaScript1.2' src='calcul.js'></script>
<script language='JavaScript1.2' src='fenetre.js'></script>
<script language='JavaScript1.2' src='ctrl.js'></script>

<CENTER><H1>Calcul</H1></CENTER>

Un petit exemple de formulaire.
<P>
Création d'une
<A href='#A' onClick='afficheDoc();'>fenêtre avec JavaScript</A>
```

Suite

```
<FORM ACTION='Simul.html' METHOD='POST' NAME='Simul'>
<CENTER>
<TABLE BORDER=3>
<TR><TD>Argument 1
   <TD> <INPUT TYPE='TEXT' SIZE=20 NAME='arg1' onChange='calcul();'></TR>
<TR><TD>* Argument 2
   <TD> <INPUT TYPE='TEXT' SIZE=20
            NAME='arg2' onChange='calcul();'>
</TR>

<TR><TD>Résultat=
   <TD> <INPUT TYPE='TEXT' SIZE=20
            NAME='res' >
 </TR>
</TABLE>
 <INPUT TYPE='BUTTON' VALUE='Vérifier' onClick='ctrl();'>
 <INPUT TYPE='RESET' VALUE='Effacer tout'
       onClick=' if (!confirm("Vraiment vous voulez effacer ?")) exit;'>
</CENTER>
</FORM>
</BODY>
</HTML>
```

http://localhost/~hf/Simul.html

Fichiers js

- Ctrl.js calcul

```
function ctrl()
{
  if (isNaN(window.document.Simul.res.value ))
  {
    alert ("Valeur incorrecte : "  +
            document.Simul.res.value + "?");
    document.forms[0].res.focus();
  }
}

function calcul()
{
 v1=document.forms[0].arg1.value;
 v2=document.forms[0].arg2.value;
 document.forms[0].res.value = v2*v1 ;
}
```

suite et fin

- exemples/fenetre.js

```
function afficheDoc()
{
  options = "width=300,height=200";
  fenetre = window.open('','MU',options);

  fenetre.document.open();
  manuel = "<HTML><HEAD><TITLE>Documentation</TITLE></
   HEAD>"
           + "<BODY bgcolor='white'>"
           + "Il n'y a pas besoin d'aide "
           + " c'est facile."
           + " Bonne chance !</BODY></HTML>";
  fenetre.document.write(manuel);
  fenetre.document.close();
}
```

Compléments: php

- php est un langage de script pour les serveurs webs
- de nombreuses fonctions permettent de traiter les requêtes http (en particulier des requêtes concernant des bases de données)
- ici on est du côté du serveur…

Exemple simple

```
<HTML> <HEAD>
<TITLE>Exemple très simple</TITLE>
</HEAD>
<BODY>
<H1>Exemple</H1>
le <?php   echo Date ("j/m/Y à H:i:s");  ?>
<P>

<?php
  echo "Client :" .  $_SERVER['HTTP_USER_AGENT'] . "<BR>";
  echo "Adresse IP client:".$_SERVER['REMOTE_ADDR']."<BR>";
  echo "Server: " . $_SERVER['SERVER_NAME'];
?>

</BODY></HTML>
```

http://localhost/cgi/~hf/ExempleSimple.php

Résultat

Exemple

le 8/11/2006 à 15:54:29 Client :Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; InfoPath.1)

Adresse IP client:127.0.0.1
Server: localhost

Reçu par le client

<HTML> <HEAD>
<TITLE>Exemple très simple</TITLE>
</HEAD>
<BODY>

<H1>Exemple</H1>

le 8/11/2006 à 15:54:29
<P>

Client :Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; .NET CLR 1.1.4322; InfoPath.1)
Adresse IP client: 127.0.0.1
Server: localhost
</BODY></HTML>

Php

- On est ici côté serveur:
  - les balises <?php> <?> sont interprétées par le serveur (apache par exemple) et servent à générer la page html reçue par le client
- Mais surtout php permet
  - d'accéder aux variables d'environnement
  - d'utiliser de nombreuses fonctionnalités
    - sessions, paramètres etc.
- Php sert souvent d'interface pour MySql serveur simple de bases de données

Php

- pas de typage ni de déclaration des variables
- $v est remplacé par la valeur de v (et permet aussi l'affectation)
- echo "$v";
- constantes define("PI, 3.1415);
- types des variables
  - numériques
    - $i=1;
    - $v=3.14;
  - chaînes de caractères (expressions régulières)
    - $nom="Hugues";
    - ',",{}

php

- Variables
  - Locales (à une fonction)
  - Globales
  - Super globales (disponibles dans tout contexte)
  - Static (garde sa valeur)
  - Variables dynamiques (le nom de la variable est une variable)
    - $a='bonjour'
    - $$a='monde'
    - echo "$a ${$a}"
    - echo "$a $bonjour"

php

- tableaux
  - indicés
    - $tab[0]="un";
    - $tab=array("un","deux","trois");
  - associatifs
    - $m=array("un"=>"one",
      "deux"=>"two");
    - $m["trois"]="three";
  - next() prev() key() current

```
do
{echo "Clé=key($m).Valeur= current($m)"}
while(next($mes));
foreach($m as $cle =>$val)
{echo "Clé=$cle.Valeur=$val";}
```

php

- Mais aussi (php4 et php5)
  - Programmation orientée objets
    - Classes et Objets
    - Liaison dynamique
    - Constructeurs
    - …
  - Exceptions

Php

- structures de contrôles
  - if
  - if else
  - while
  - do while
  - for
  - foreach
    - break, continue

fonctions

```
function Nom([$arg1, $arg2, ...])
{
     corps
}
```

passage par valeur (et par référence &)

exemples

```
function Add($i,$j){
    $somme= $i + $j;
    return $somme;
}
function Add($i,$j,&$somme){
    $somme= $i + $j;
}
```

Pour le serveur…

- tableaux associatifs prédéfinis
  - $_SERVER: environnement serveur
    - REQUEST_METHOD
    - QUERY_STRING
    - CONTENT_LENGTH
    - SERVER_NAME
    - PATH_INFO
    - HTTP_USER_AGENT
    - REMOTE_ADDR
    - REMOTE_HOST
    - REMOTE_USER
    - REMOTE_PASSWORD

Suite

- Autres tableaux
  - $_ENV : environnement système
  - $_COOKIE
  - $_GET
  - $_POST
  - $_FILES
  - $_REQUEST (variables des 4 précédents)
  - $_SESSION
  - $GLOBALS les variables globales du script

Cookies et php

```
<?php
// Est-ce que le Cookie existe ?
if (isSet($_COOKIE['compteur']))
{
    $message = "Vous êtes déjà venu {$_COOKIE['compteur']} fois "
      . "me rendre visite<BR>\n";
    // On incrémente le compteur
    $valeur = $_COOKIE['compteur'] + 1;
}
else
{
    // Il faut créer le cookie avec la valeur 1
    $message = "Bonjour, je vous envoie un cookie<BR>\n";
    $valeur = 1;
}
// Envoi du cookie
SetCookie ("compteur", $valeur);
?>
```

Cookies et php (fin)

```
<HTML><HEAD>
  <TITLE>Les cookies</TITLE>

</HEAD>
<BODY>

<H1>Un compteur d'accès au site avec cookie</H1>

<?php echo $message; ?>

</BODY></HTML>
```

http://localhost/~hf/SetCookie.php

En utilisant les sessions

```
<?php
  // La fonction session_start fait tout le travail
  session_start();
?>
<HTML><HEAD>
  <TITLE>Les cookies</TITLE>
</HEAD>
<BODY>

<H1>Un compteur d'accès au site avec Session</H1>
```

Fin

```
<?php
if (!isSet($_SESSION['cp']))
{
   $_SESSION['cp']=1;
        echo "C'estlapremière fois,votre id est:" .
                                session_id()."<BR>";
}
else{
        $_SESSION['cp']++;
        echo "C'est votre ".$_SESSION['cp']." n-ième connexion";
   if($_SESSION['cp']>10){
       echo "on vous a trop vu"."<BR>";
              session_destroy();
   }
}
?>
</BODY></HTML>
```

http://localhost/~hf/SessionPHP.php

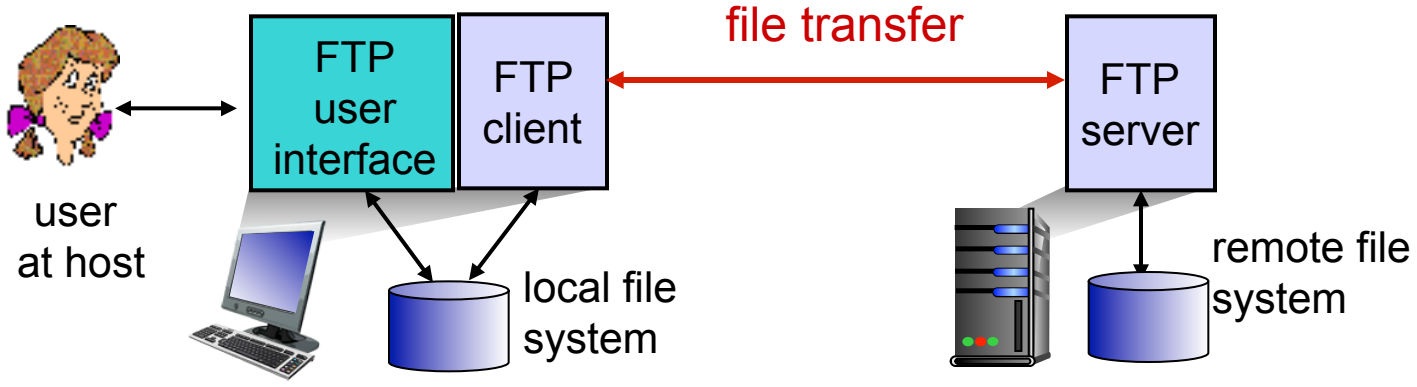
session

- session_start()
- session_destroy()
- session_id()
  - on peut associer des variables à la session par le tableau associatif $_SESSION
  - elle sera accessible à chaque session_start() jusqu'au session_destroy() pour toute connexion qui fournit le session_id().

Couche application

FTP

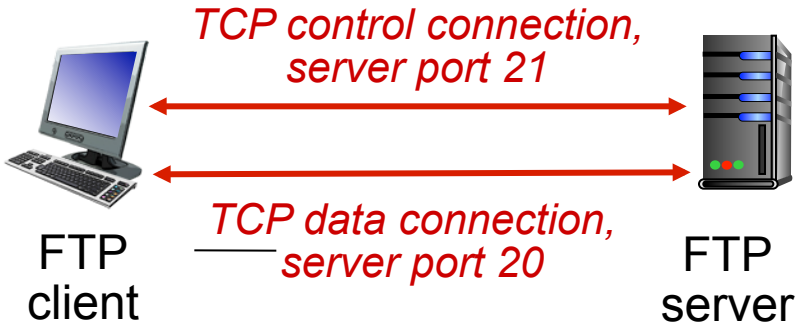
FTP: the file transfer protocol



- 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, using TCP
- client authorized over control connection
- client browses remote directory, sends commands over control connection
- when server receives file transfer command, *server* opens 2nd TCP data 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`**

Couche application

DNS

DNS: domain name system

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

Domain Name System:

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol:* hosts, name servers communicate to *resolve* names (address/name translation)
  - note: core Internet function, implemented as application-layer protocol
  - complexity at network's "edge"

DNS: services, structure

DNS services

- hostname to IP address translation
- host aliasing
  - canonical, alias names
- mail server aliasing
- load distribution
  - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

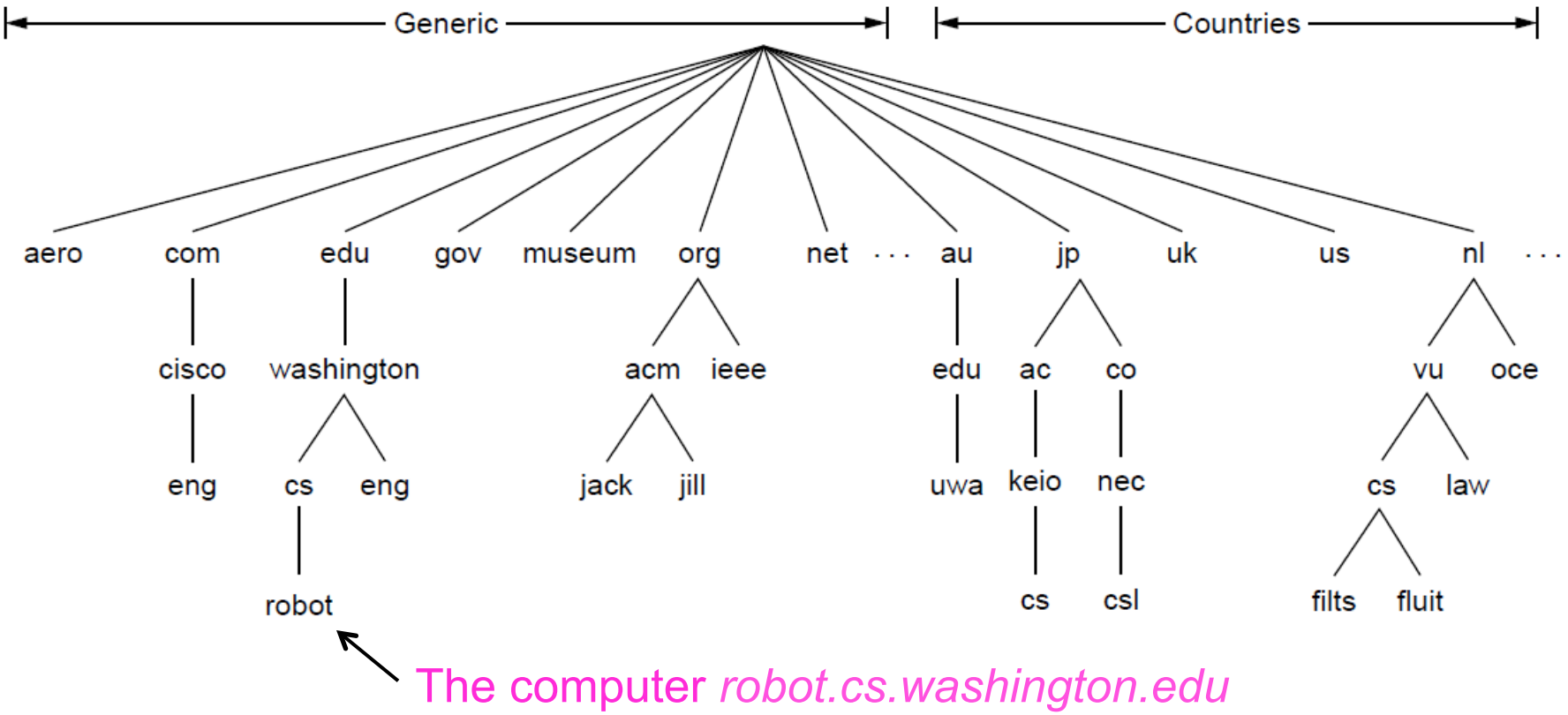
- single point of failure
- traffic volume
- distant centralized database
- maintenance

A: doesn't scale!

DNS Name space

DNS namespace is hierarchical from the root down

- Different parts delegated to different organizations



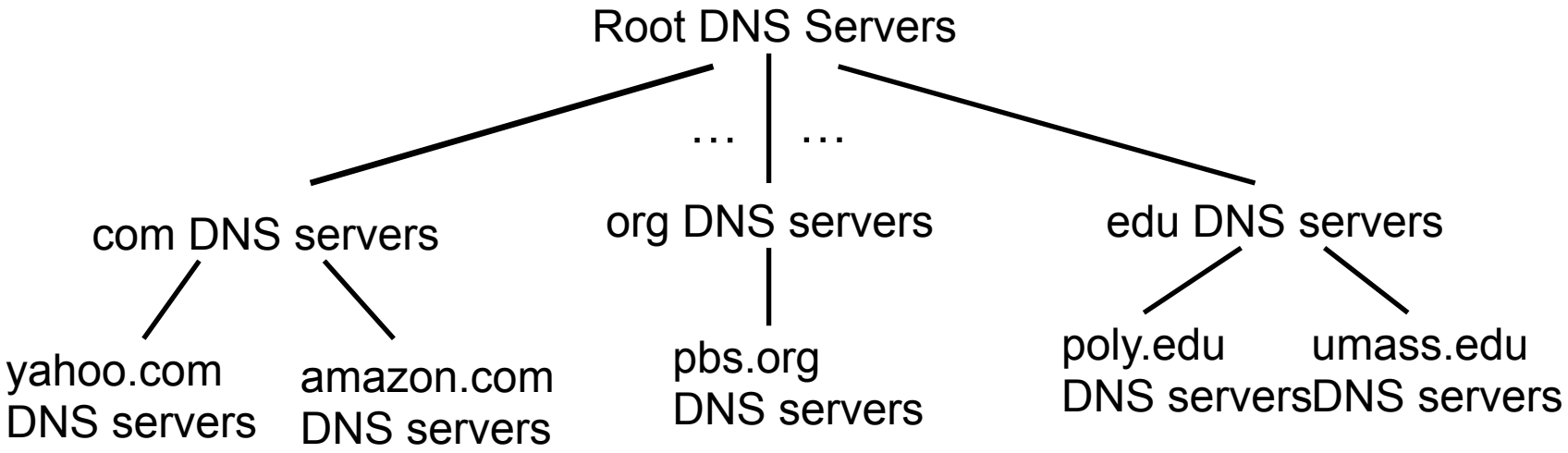
DNS Name Space

Generic top-level domains are controlled by ICANN who appoints registrars to run them

Domain	Intended use	Start date	Restricted?
com	Commercial	1985	No
edu	Educational institutions	1985	Yes
gov	Government	1985	Yes
int	International organizations	1988	Yes
mil	Military	1985	Yes
net	Network providers	1985	No
org	Non-profit organizations	1985	No
aero	Air transport	2001	Yes
biz	Businesses	2001	No
coop	Cooperatives	2001	Yes
info	Informational	2002	No
museum	Museums	2002	Yes
name	People	2002	No
pro	Professionals	2002	Yes
cat	Catalan	2005	Yes
jobs	Employment	2005	Yes
mobi	Mobile devices	2005	Yes
tel	Contact details	2005	Yes
travel	Travel industry	2005	Yes
xxx	Sex industry	2010	No

This one was controversial

DNS: a distributed, hierarchical database

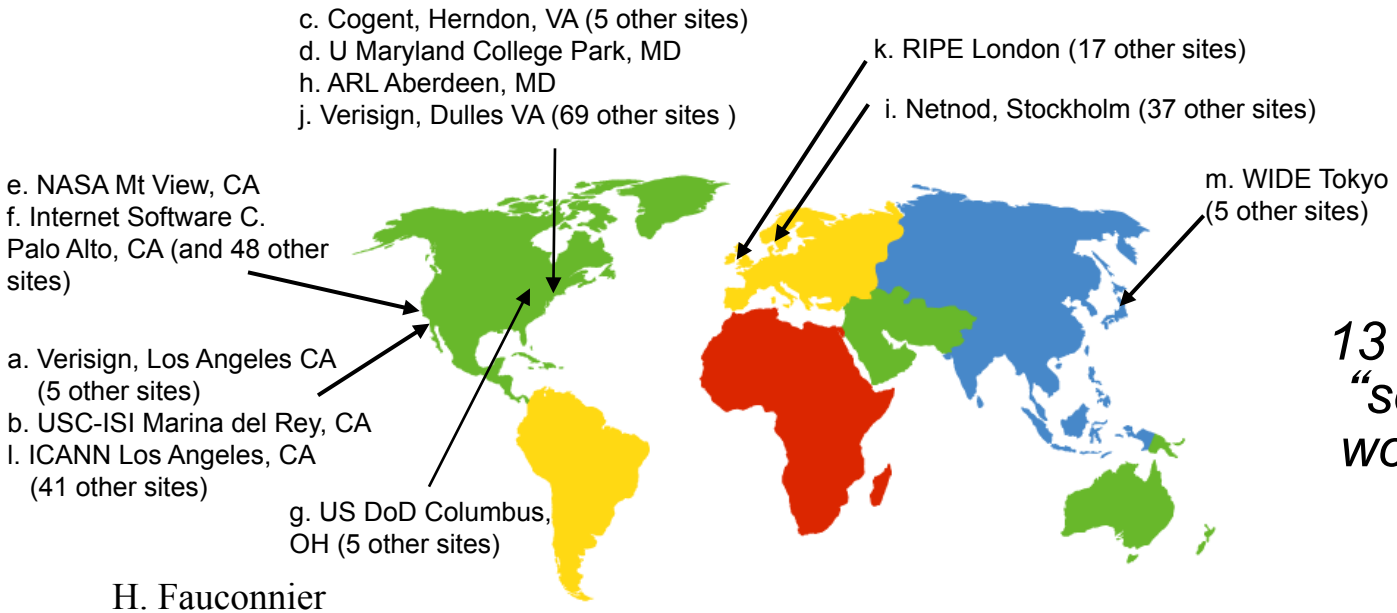


client wants IP for www.amazon.com; 1st approx:

- client queries 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 not known
  - gets mapping
  - returns mapping to local name server



13 root name "servers" worldwide

H. Fauconnier

TLD, authoritative servers

top-level domain (TLD) servers:

- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

authoritative DNS servers:

- organization's own DNS server(s), providing authoritative hostname to IP mappings for organization's named hosts
- can be maintained by organization or service provider

Local DNS name server

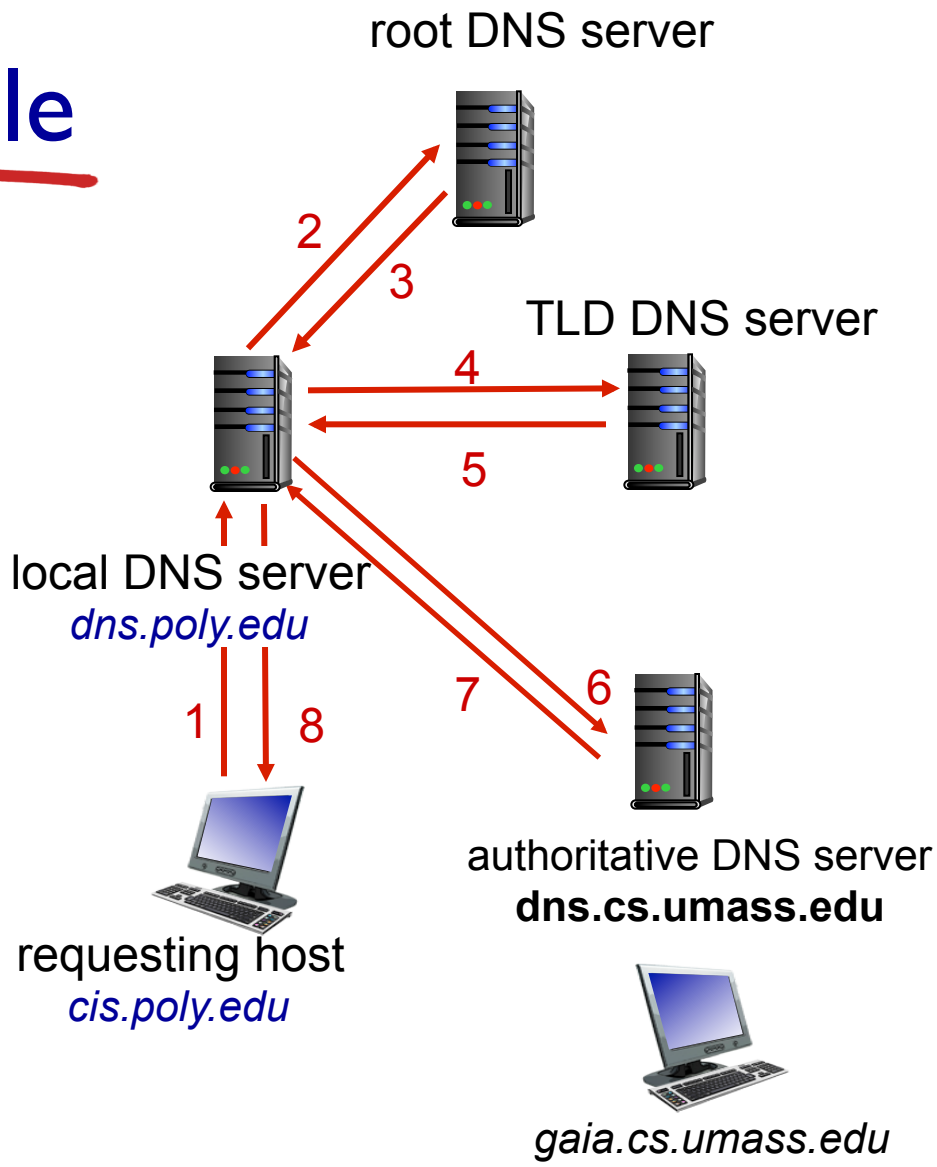
- does not strictly belong to hierarchy
- each ISP (residential ISP, company, university) has one
  - also called "default name server"
- when host makes DNS query, query is sent to its local DNS server
  - has local cache of recent name-to-address translation pairs (but may be out of date!)
  - acts as proxy, forwards query into hierarchy

DNS name resolution example

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

iterated query:

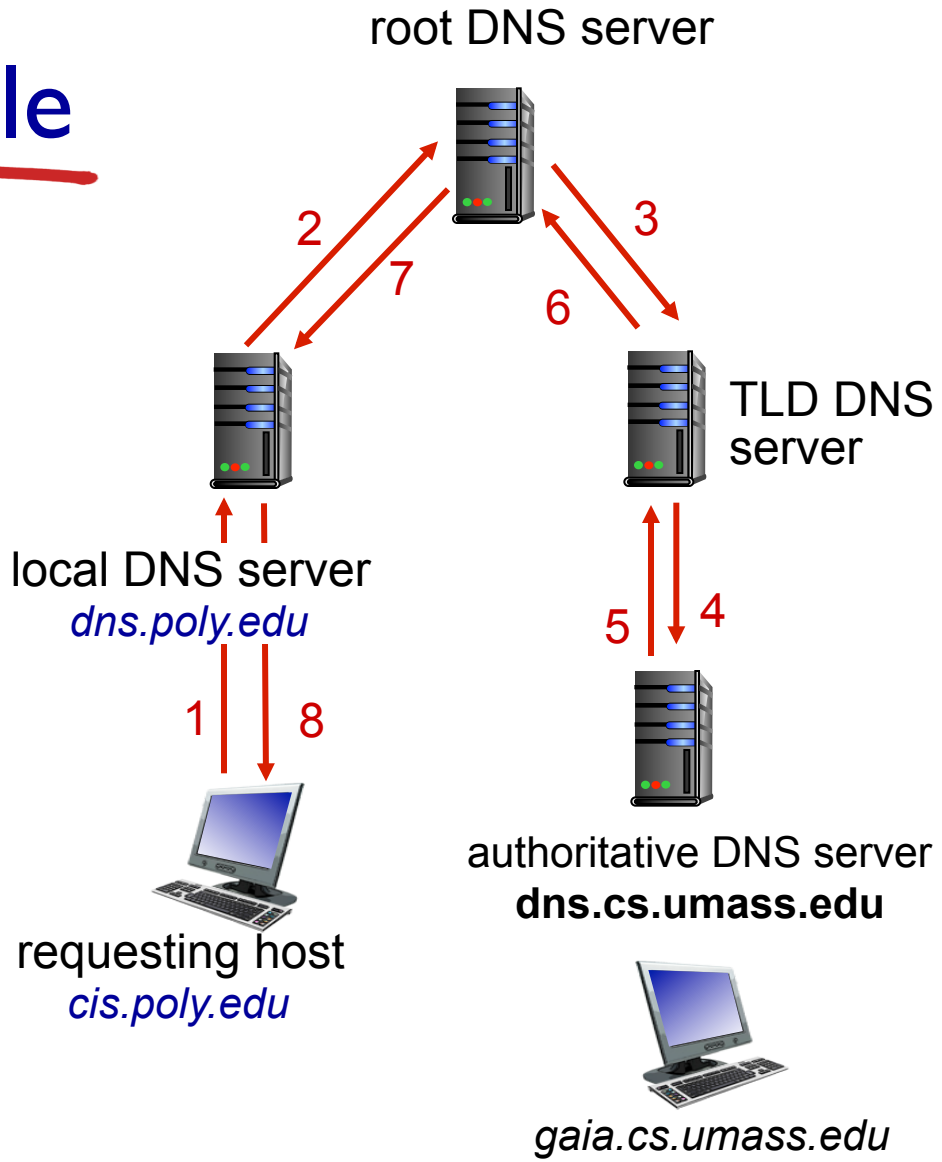
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



DNS name resolution example

recursive query:

- puts burden of name resolution on contacted name server
- heavy load at upper levels of hierarchy?



DNS: caching, updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time (TTL)
  - TLD servers typically cached in local name servers
    - thus root name servers not often visited
- cached entries may be *out-of-date* (best effort name-to-address translation!)
  - if name host changes IP address, may not be known Internet-wide until all TTLs expire
- update/notify mechanisms proposed IETF standard
  - RFC 2136

DNS records

DNS: distributed db storing resource records (RR)

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

type=A

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

type=NS

- **`name`** is domain (e.g., foo.com)
- **`value`** is hostname of authoritative name server for this domain

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

type=MX

- **`value`** is name of mailserver associated with **`name`**

Domain Resource Records

The key resource records in the namespace are IP addresses (A/AAAA) and name servers (NS), but there are others too (e.g., MX)

Type	Meaning	Value
SOA	Start of authority	Parameters for this zone
A	IPv4 address of a host	32-Bit integer
AAAA	IPv6 address of a host	128-Bit integer
MX	Mail exchange	Priority, domain willing to accept email
NS	Name server	Name of a server for this domain
CNAME	Canonical name	Domain name
PTR	Pointer	Alias for an IP address
SPF	Sender policy framework	Text encoding of mail sending policy
SRV	Service	Host that provides it
TXT	Text	Descriptive ASCII text

Domain Resource Records

```
; Authoritative data for cs.vu.nl
cs.vu.nl.       86400   IN   SOA     star boss (9527,7200,7200,241920,86400)
cs.vu.nl.       86400   IN   MX      1 zephyr
cs.vu.nl.       86400   IN   MX      2 top
cs.vu.nl.       86400   IN   NS      star

star            86400   IN   A       130.37.56.205
zephyr          86400   IN   A       130.37.20.10
top             86400   IN   A       130.37.20.11
www             86400   IN   CNAME   star.cs.vu.nl
ftp             86400   IN   CNAME   zephyr.cs.vu.nl

flits           86400   IN   A       130.37.16.112
flits           86400   IN   A       192.31.231.165
flits           86400   IN   MX      1 flits
flits           86400   IN   MX      2 zephyr
flits           86400   IN   MX      3 top

rowboat                 IN   A       130.37.56.201
                        IN   MX      1 rowboat
                        IN   MX      2 zephyr

little-sister           IN   A       130.37.62.23

laserjet                IN   A       192.31.231.216
```

Name server

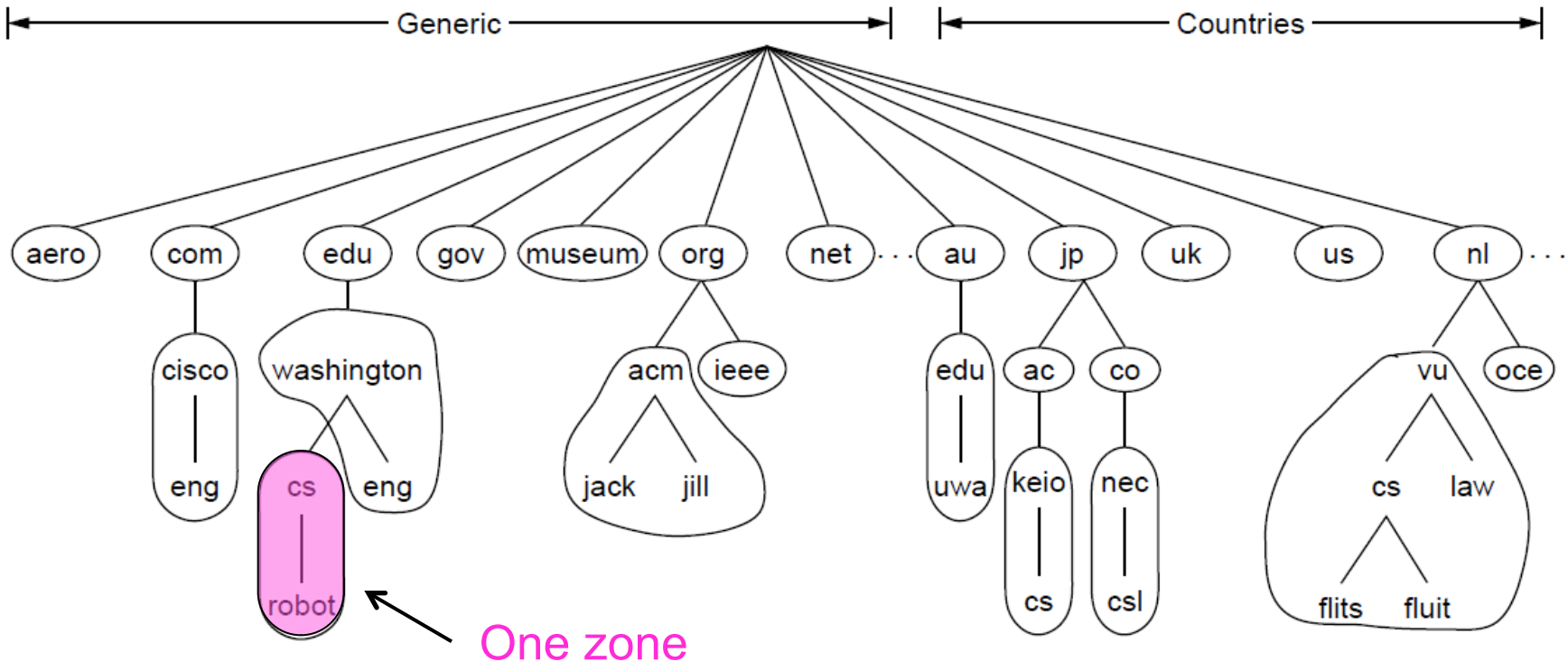
IP addresses of computers

Mail gateways

- A portion of a possible DNS database for cs.vu.nl.

Name Servers

Name servers contain data for portions of the name space called zones (circled).

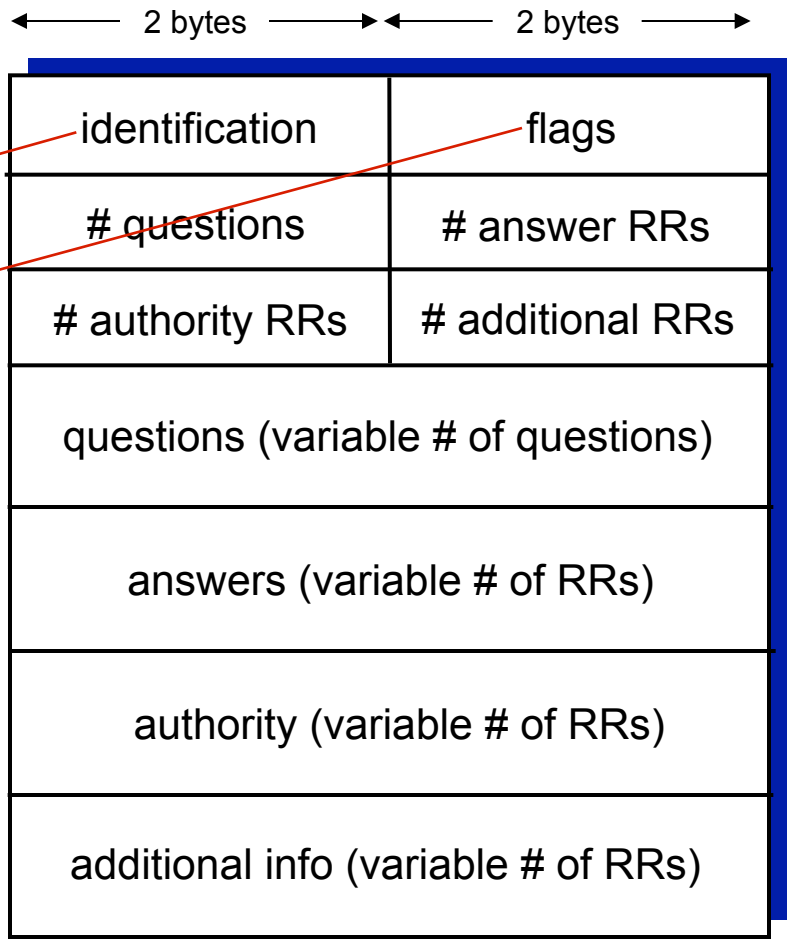


DNS protocol, messages

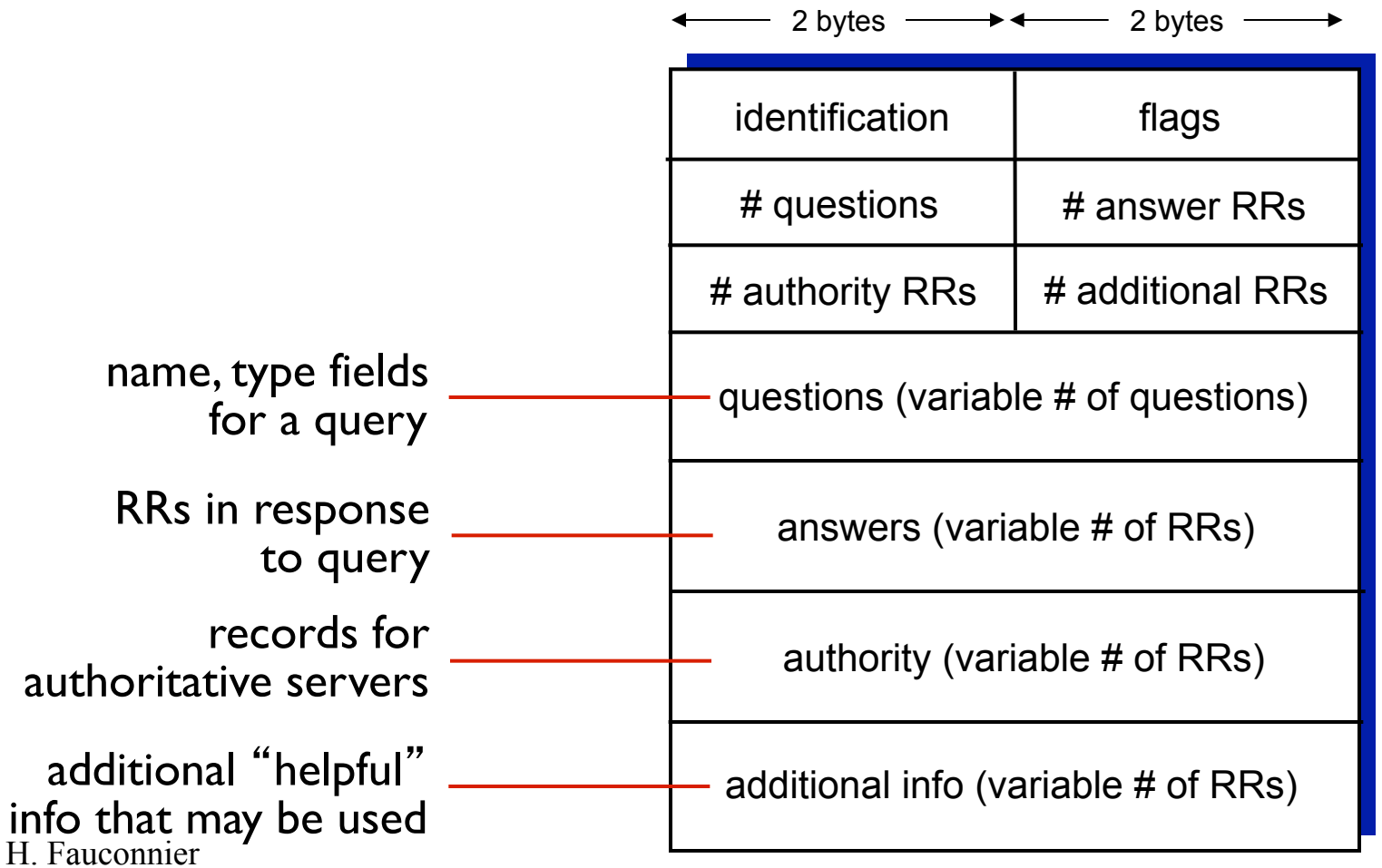
- *query* and *reply* messages, both with same *message format*

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



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:
    `(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

Attacking 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

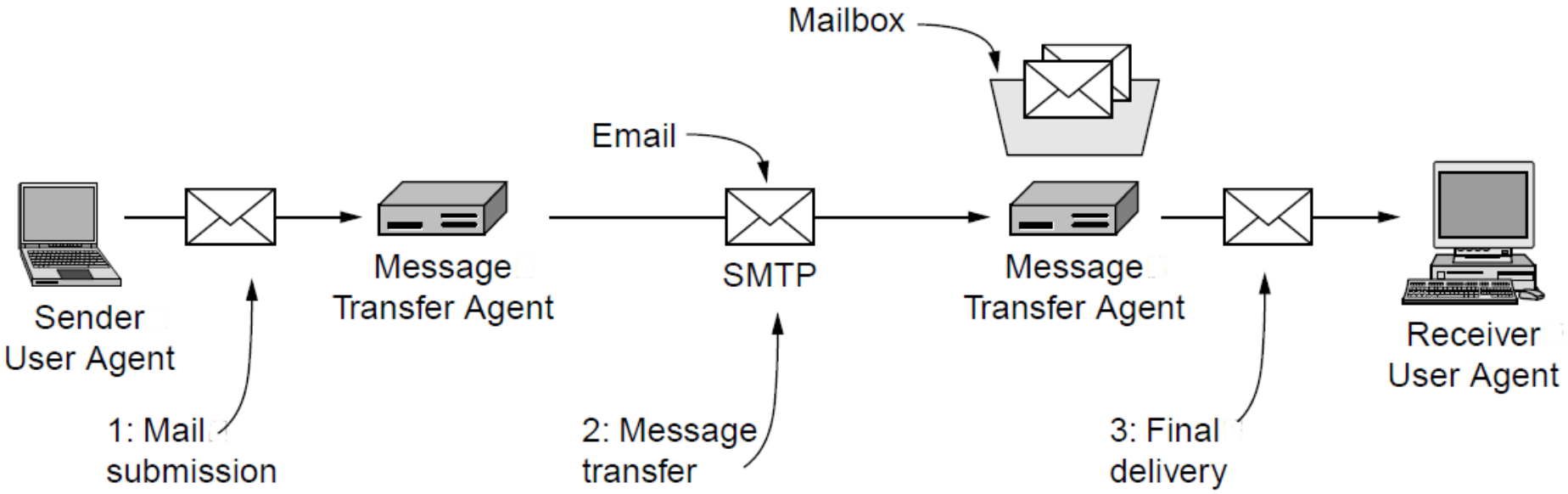
Couche application

electronic mail

- SMTP, POP3, IMAP

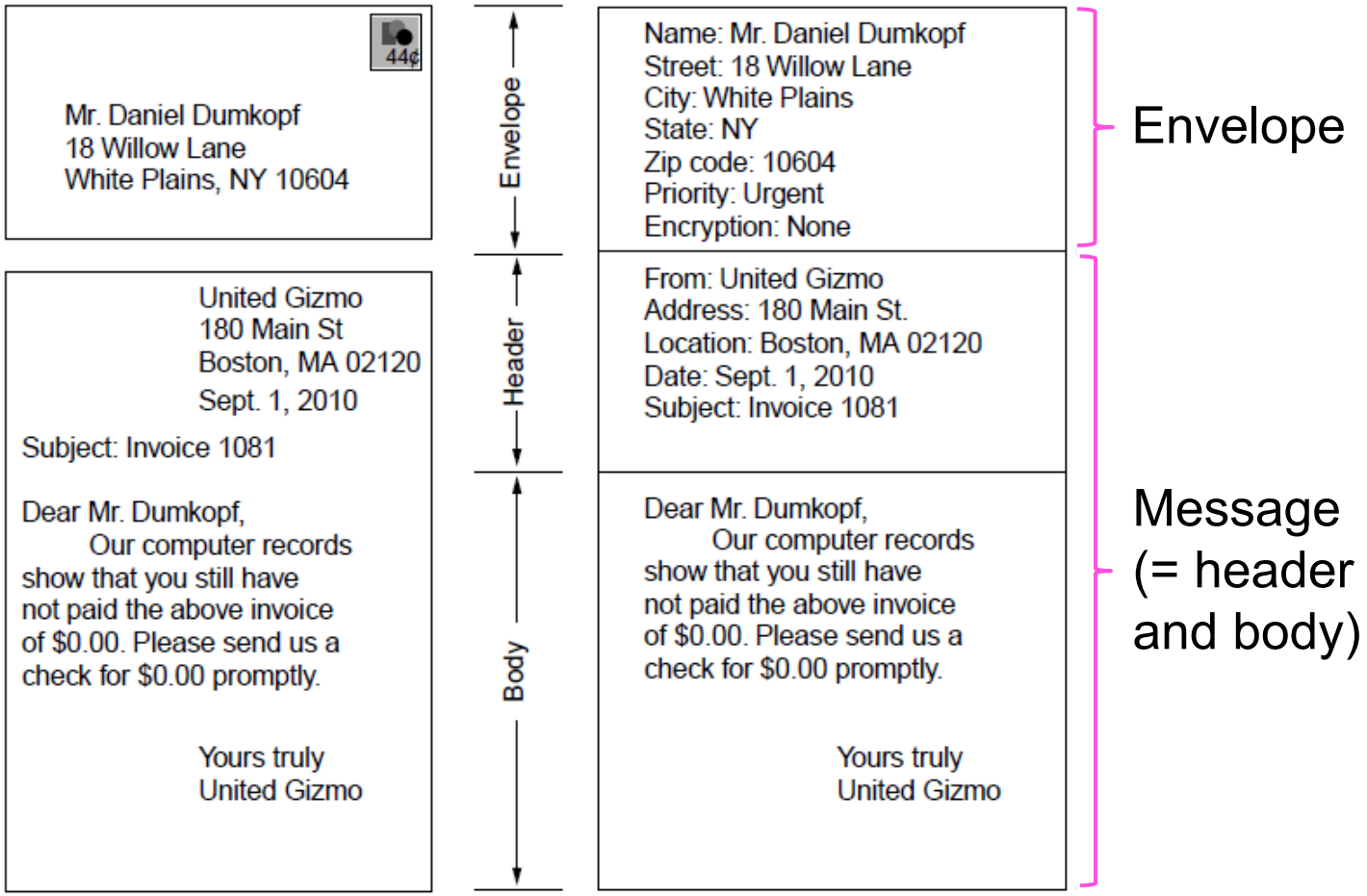
Architecture et Services

The key components and steps (numbered) to send email



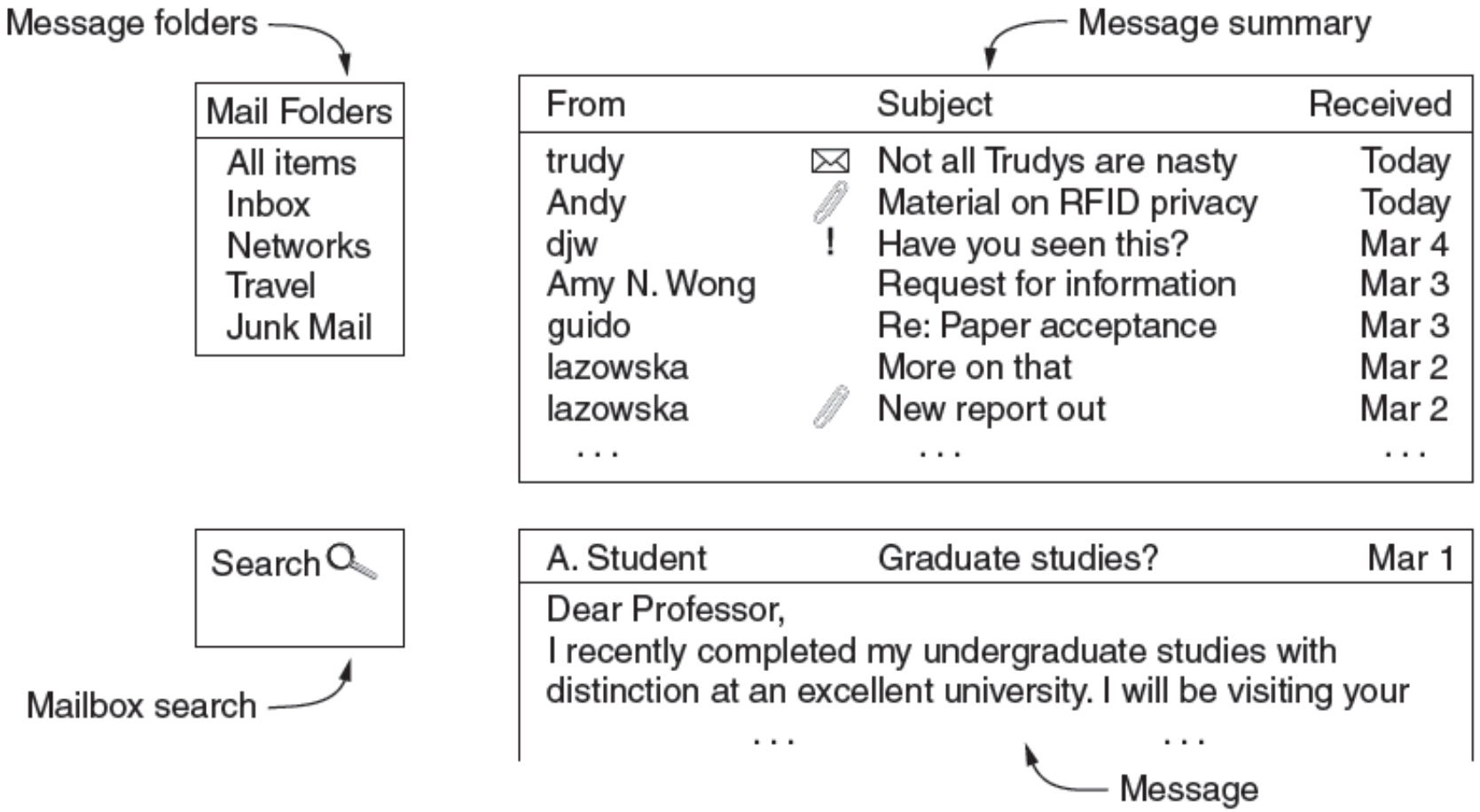
Architecture of the email system

Architecture and Services (2)



The User Agent

What users see – interface elements of a typical user agent



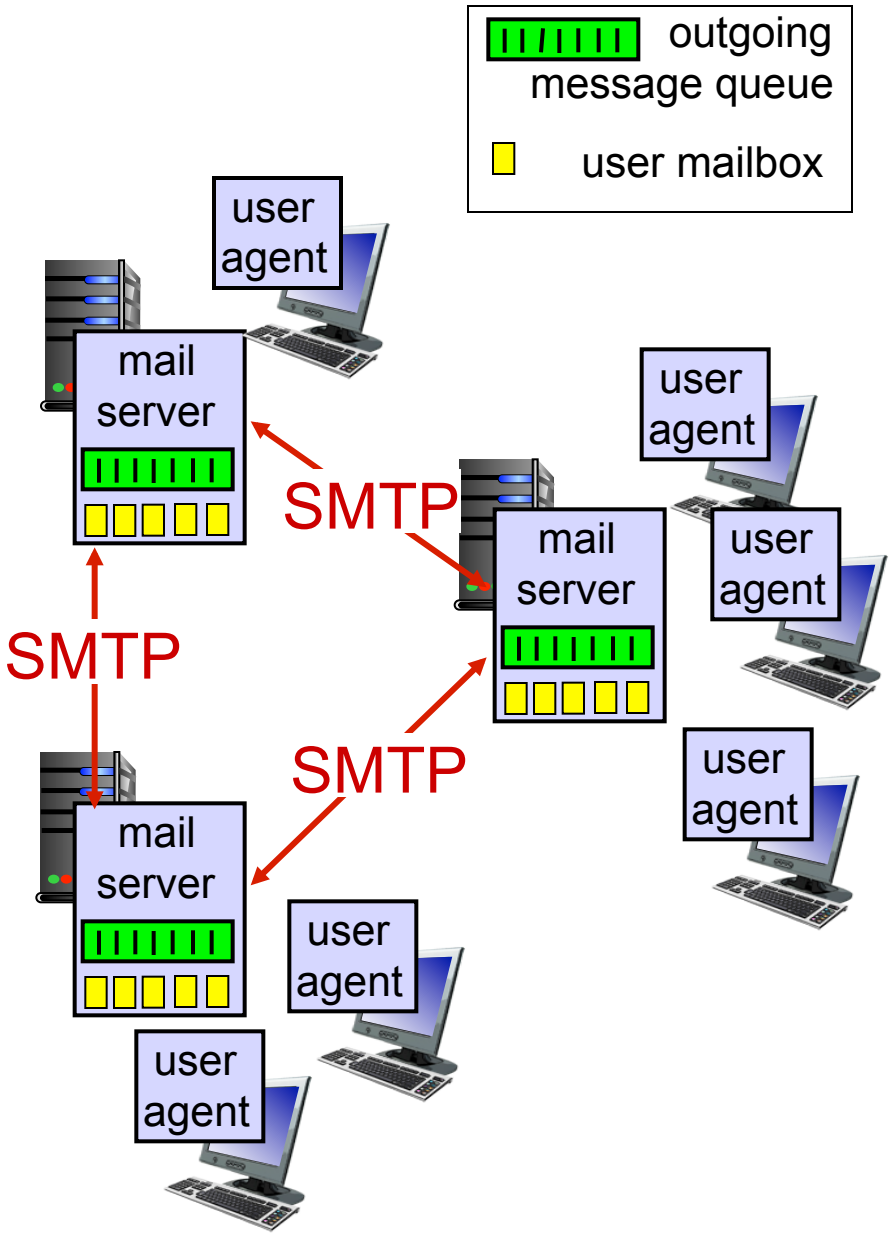
Electronic mail

Three major components:

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

User Agent

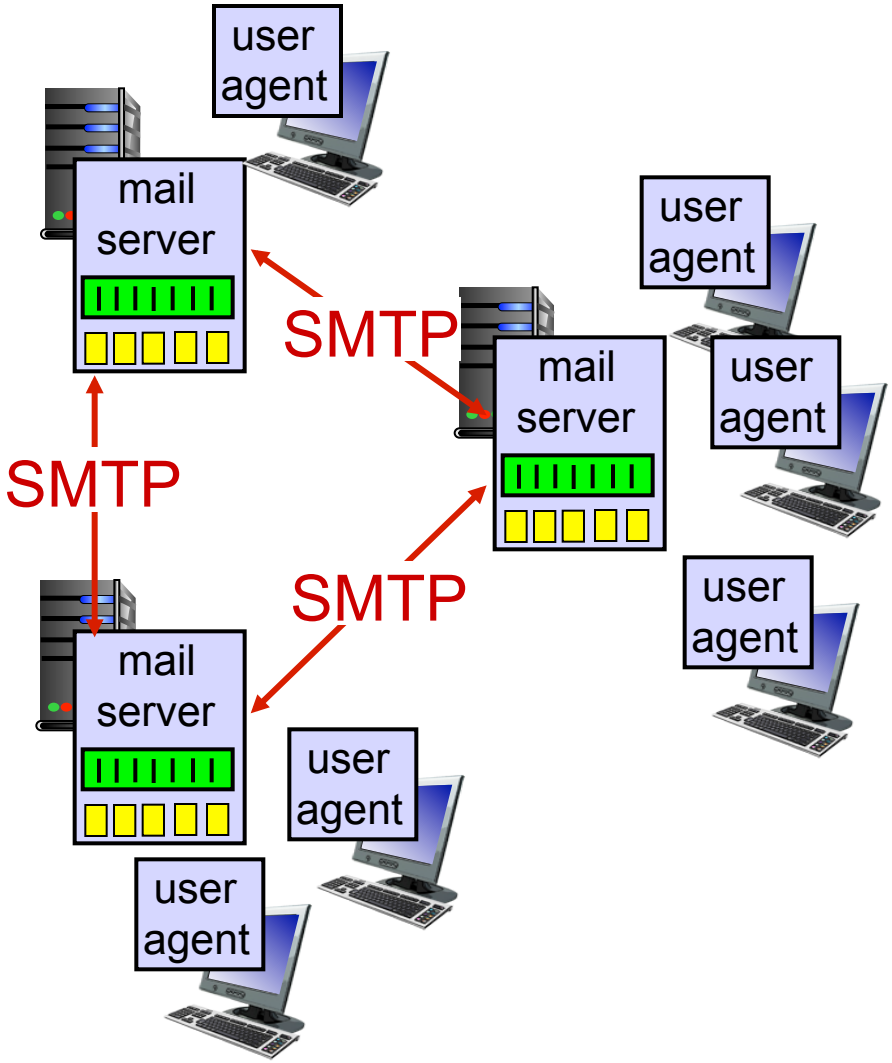
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Outlook, Thunderbird, iPhone mail client
- outgoing, incoming messages stored on server



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

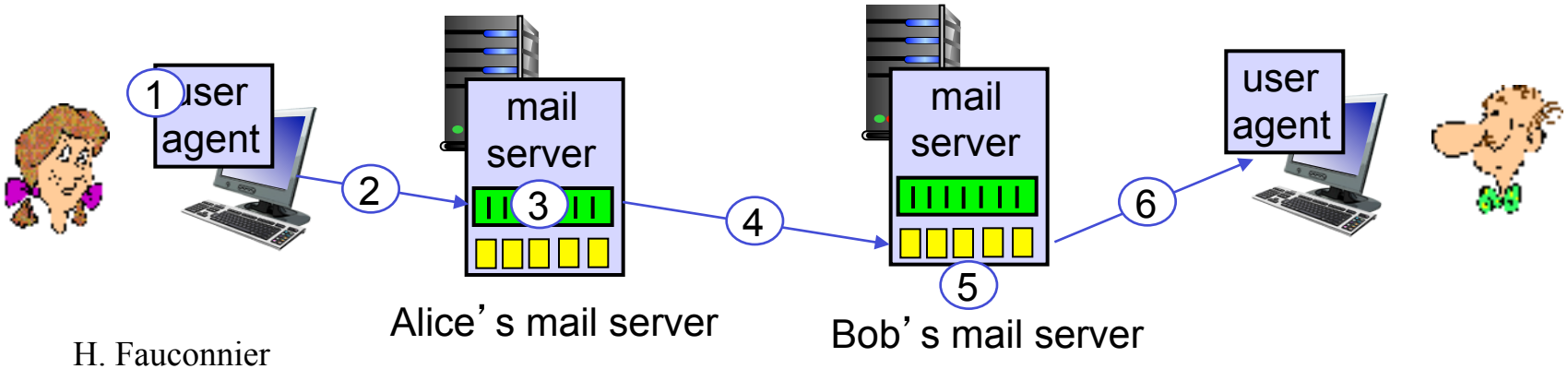


Electronic Mail: SMTP [RFC 2821]

- 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 (like HTTP, FTP)
  - commands: ASCII text
  - response: status code and phrase
- messages must be in 7-bit ASCI

Scenario: Alice sends message to Bob

1) Alice uses UA to compose message "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

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

Try SMTP interaction for yourself:

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

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

SMTP: final words

- SMTP uses persistent connections
- SMTP requires 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 msg
- SMTP: multiple objects sent in multipart msg

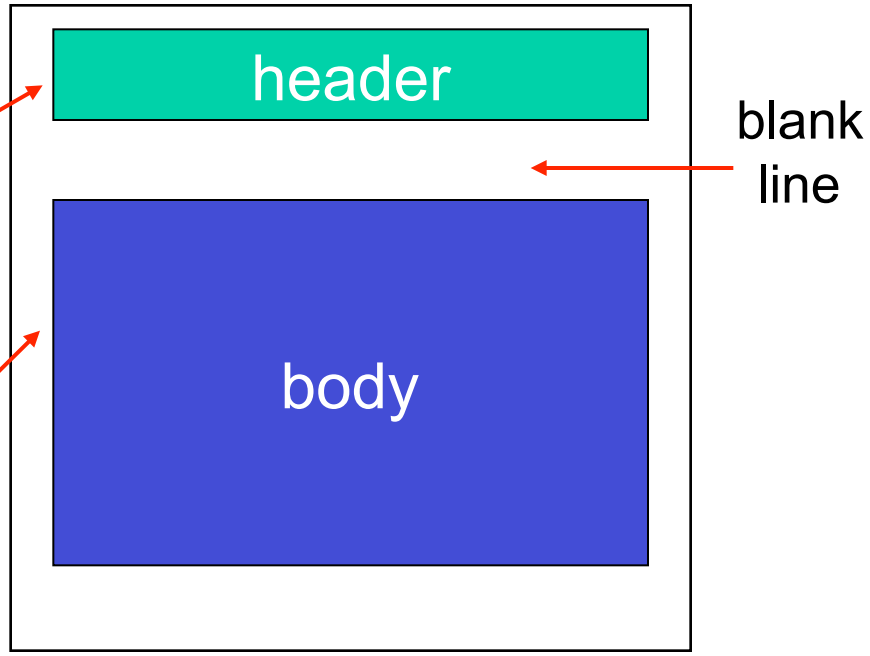
Mail message format

SMTP: protocol for exchanging email msgs

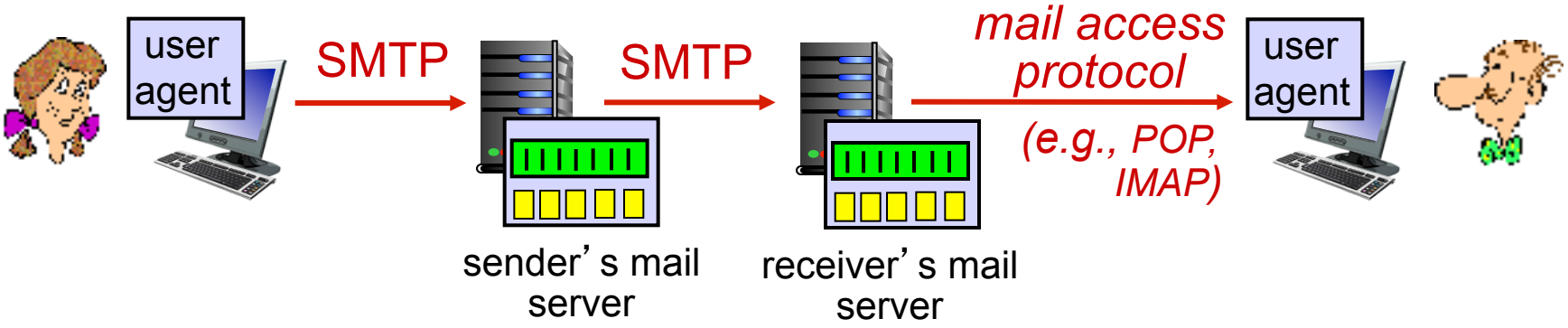
RFC 822: standard for text message format:

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

  different from SMTP MAIL FROM, RCPT TO: commands!
- Body: the "message"
  - ASCII characters only



Mail access protocols



- SMTP: delivery/storage to receiver's server
- mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]: authorization, download
  - IMAP: Internet Mail Access Protocol [RFC 1730]: more features, including manipulation of stored msgs 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 POP3 "download and delete" mode
  - Bob cannot re-read e-mail if he changes client
- POP3 "download-and-keep": copies of messages on different clients
- POP3 is stateless across sessions

IMAP

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

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

2.6 P2P applications

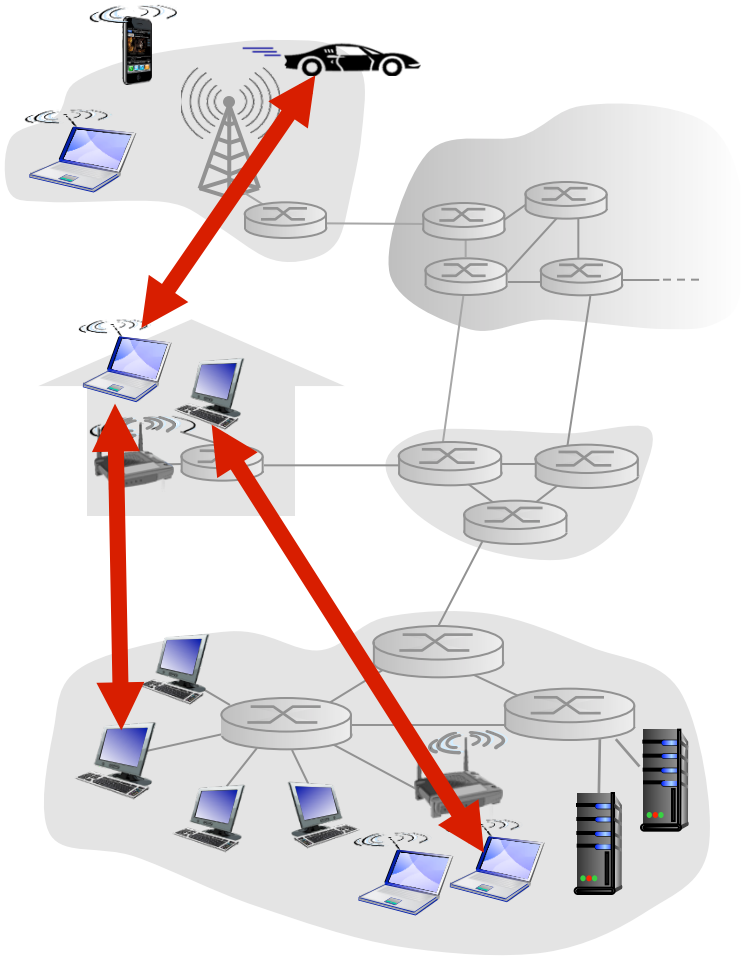
2.7 socket programming with UDP and TCP

Pure P2P architecture

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

examples:

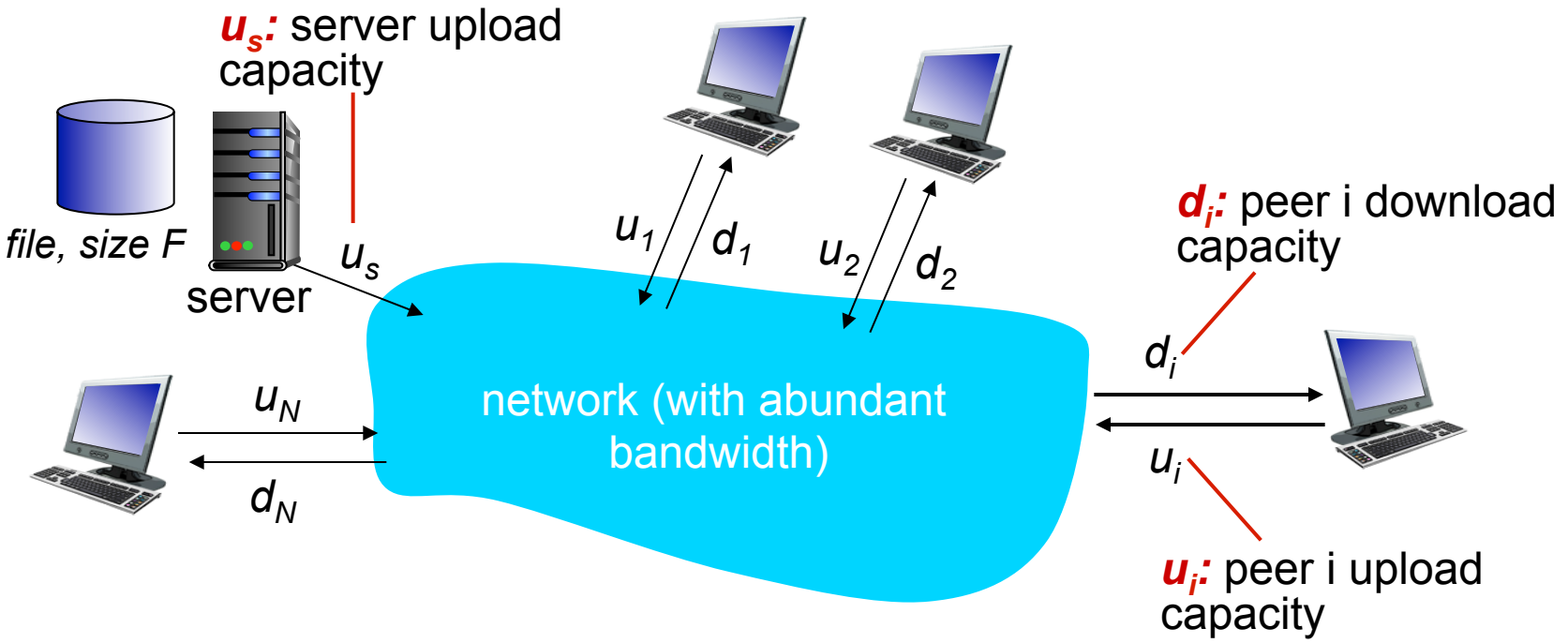
- file distribution (BitTorrent)
- Streaming (KanKan)
- VoIP (Skype)



File distribution: client-server vs P2P

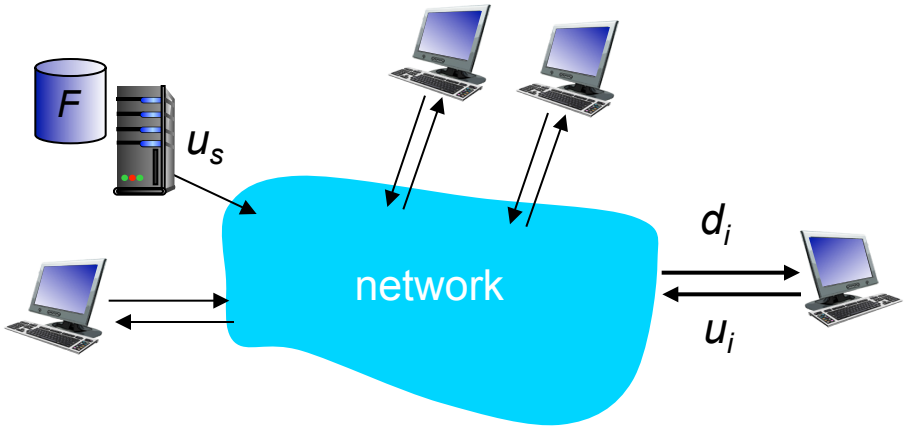
Question: how much time to distribute file (size *F*) from one server to *N* peers?

- 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}



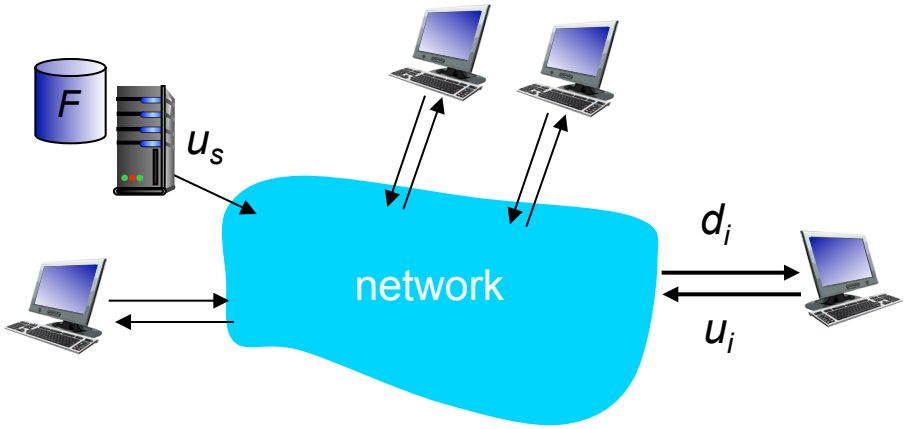
time to distribute F to N clients using client-server approach

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

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 $u_s + Su_i$



time to distribute F to N clients using P2P approach

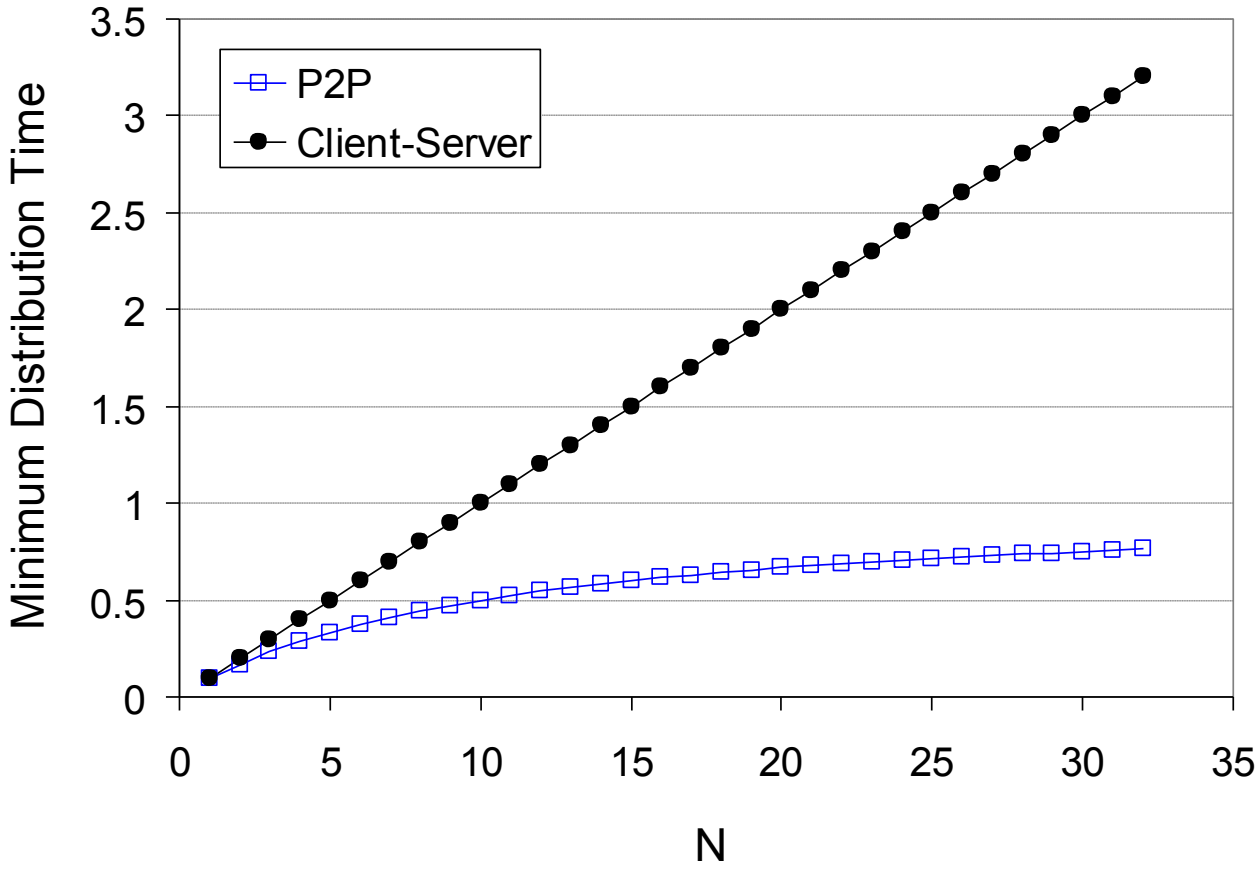
$$D_{P2P} \geq max\{F/u_s, F/d_{min}, NF/(u_s + Su_i)\}$$

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, $u_s = 10u$, $d_{min} \geq u_s$

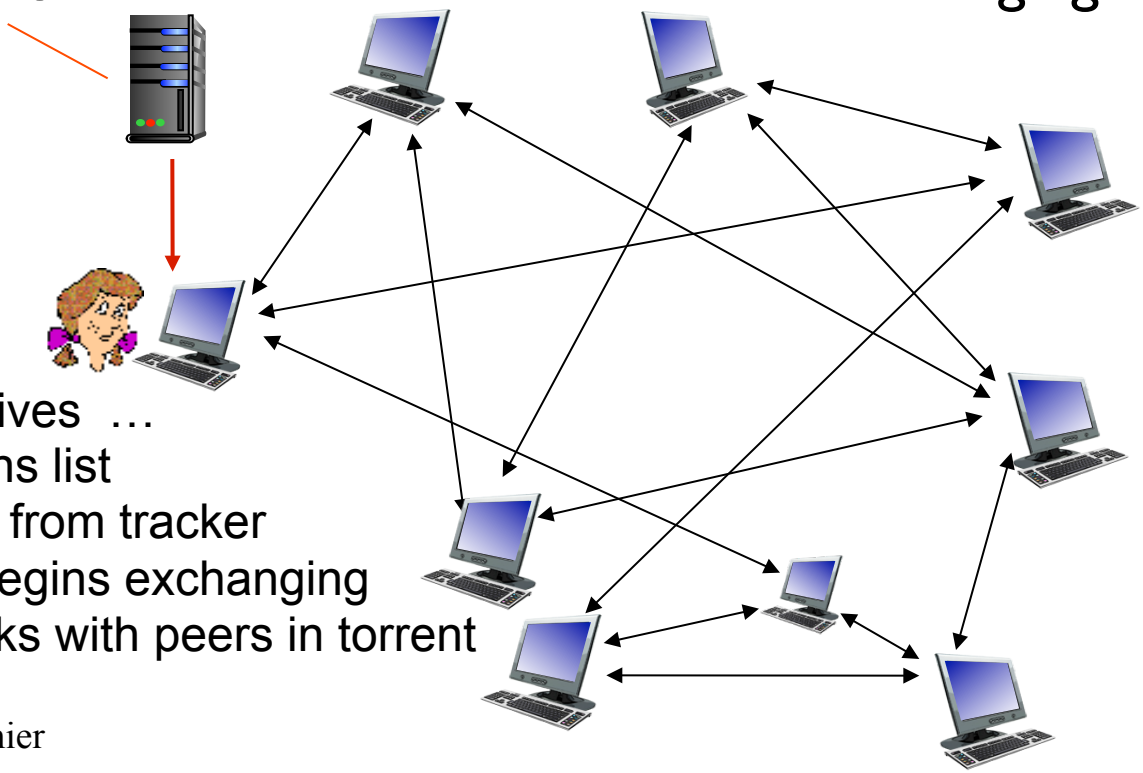


P2P file distribution: BitTorrent

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

tracker: tracks peers participating in torrent

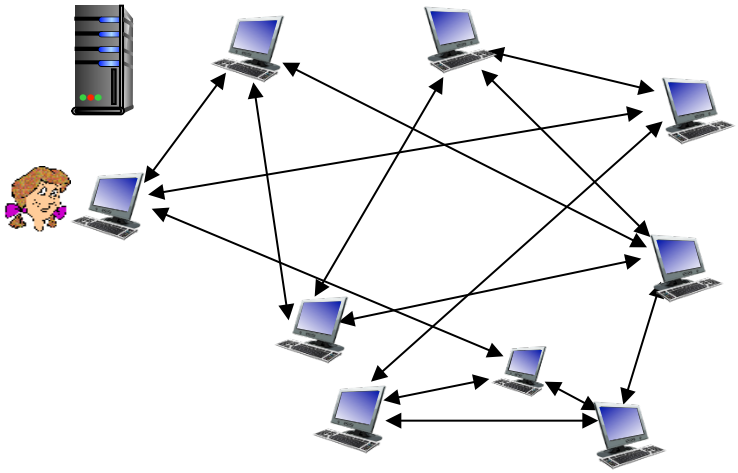
torrent: group of peers exchanging chunks of a file



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

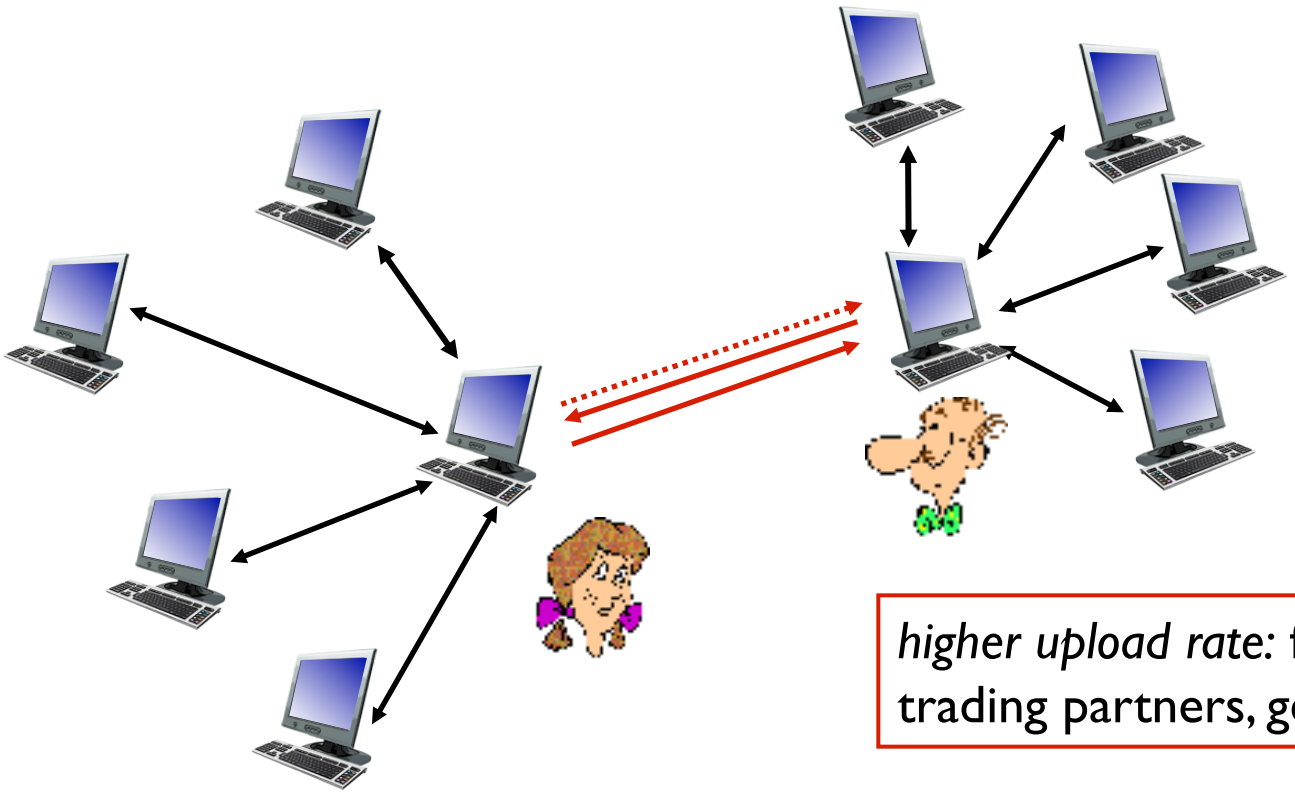
- Alice sends chunks to those four peers currently sending her chunks *at highest rate*
  - other peers are choked by Alice (do not receive chunks from her)
  - re-evaluate top 4 every10 secs
- every 30 secs: randomly select another peer, starts sending chunks
  - "optimistically unchoke" this peer
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

H. Fauconnier

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

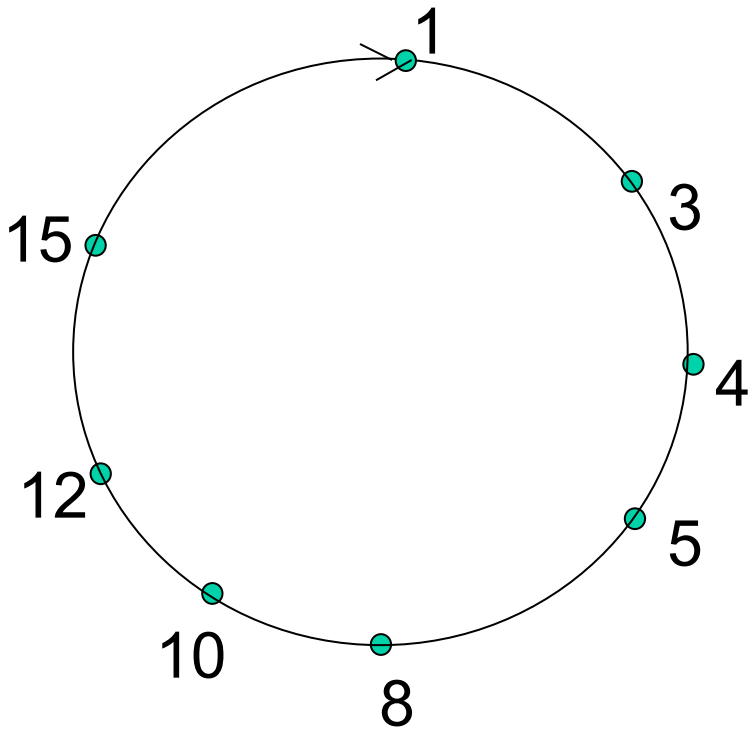
- assign integer identifier to each peer in range $[0,2^n-1]$ for some *n*.
  - each identifier represented by *n* bits.

- 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)

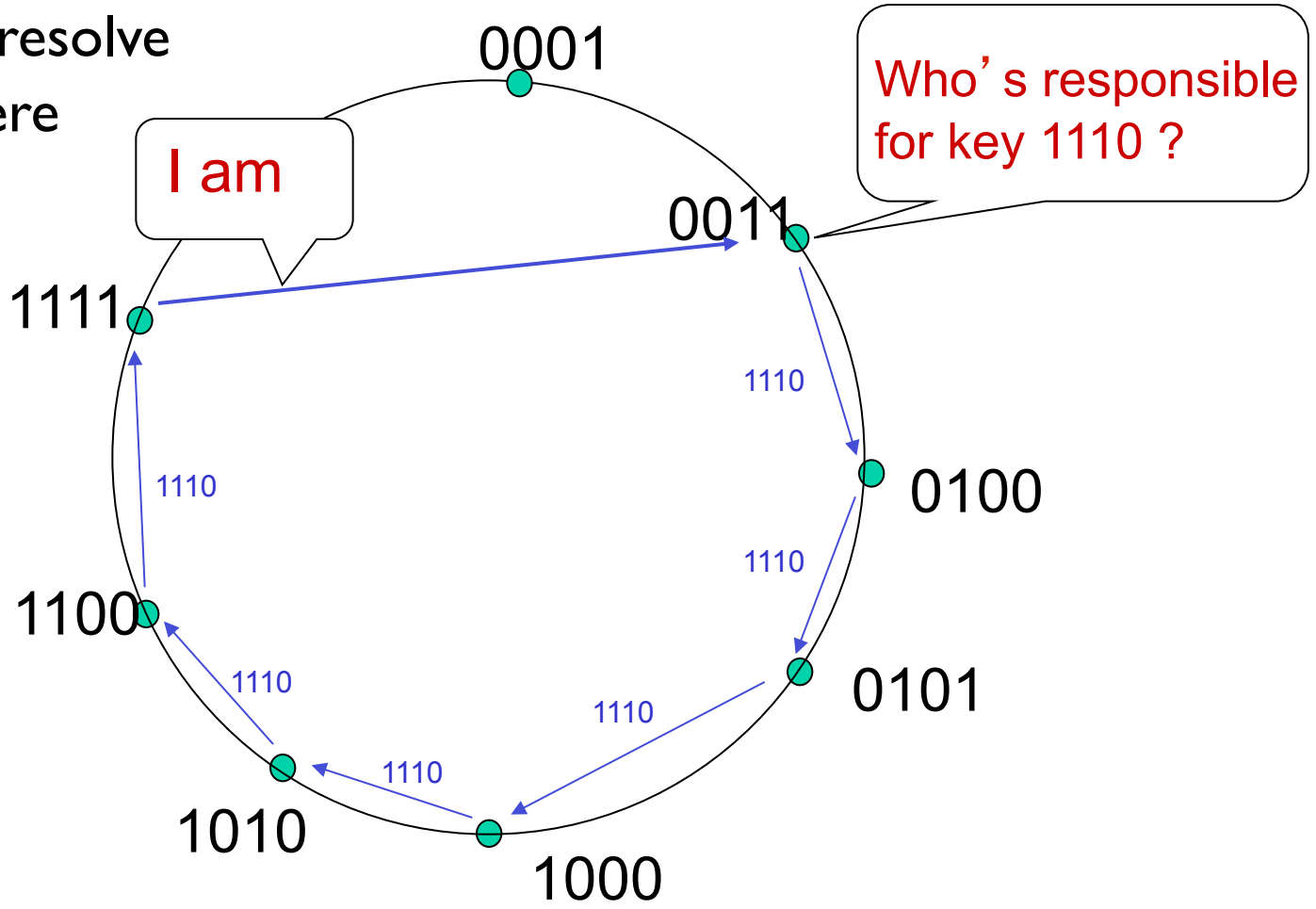


- each peer *only* aware of immediate successor and predecessor.
- "overlay network"

H. Fauconnier

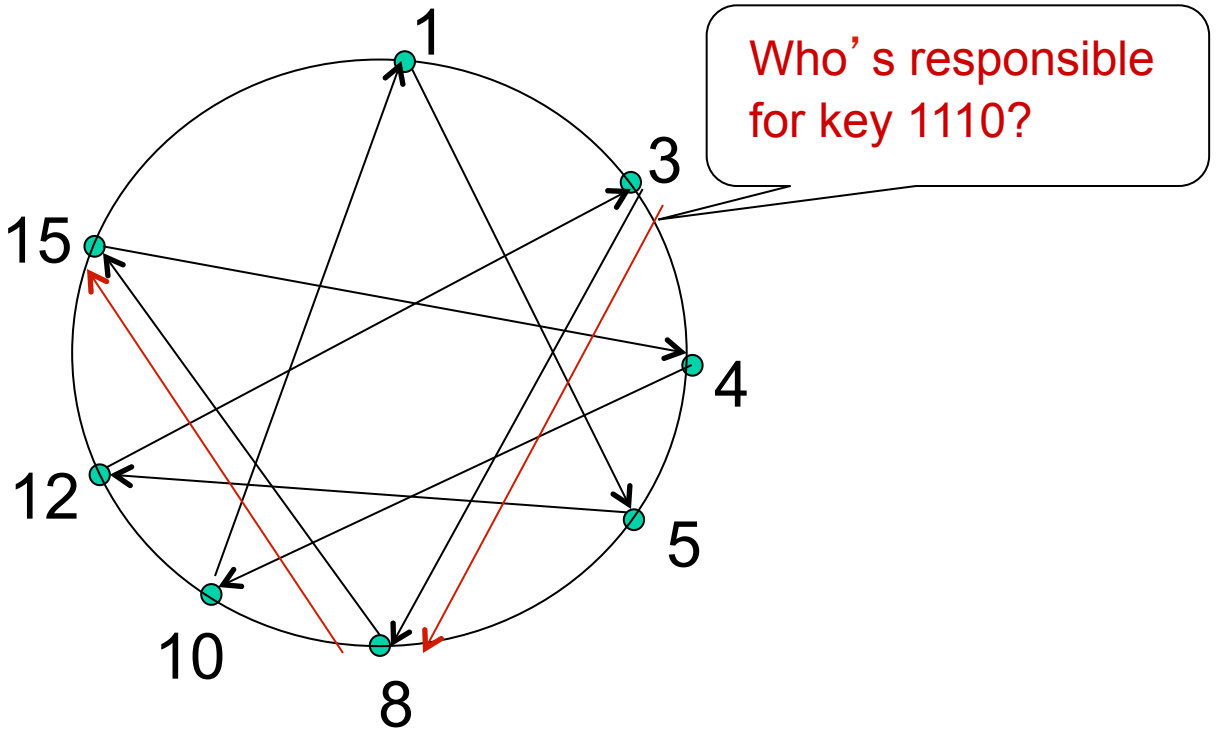
Circular DHT (1)

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



Define closest as closest successor

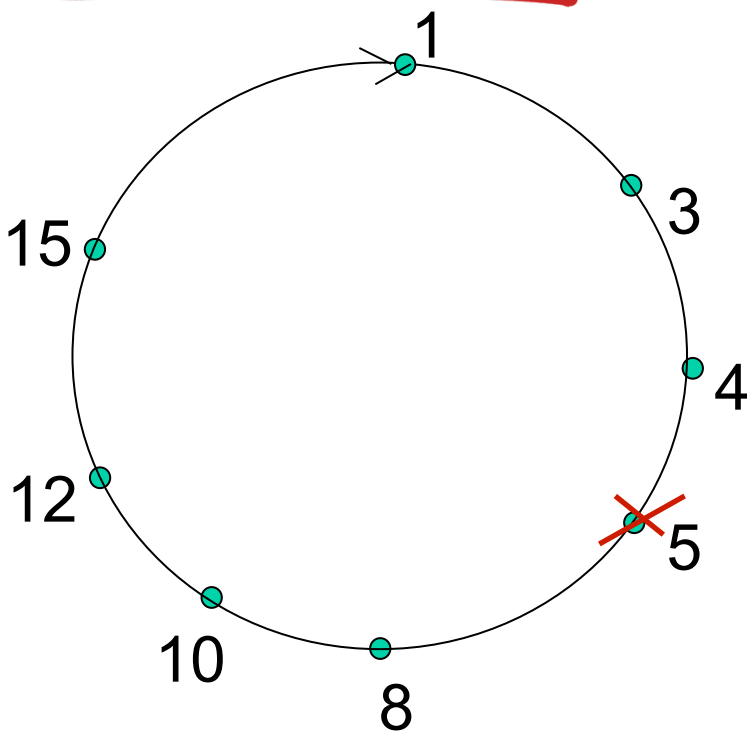
Circular DHT with shortcuts



- 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

H. Fauconnier

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?

H. Fauconnier

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

2.2 Web and HTTP

2.3 FTP

2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS

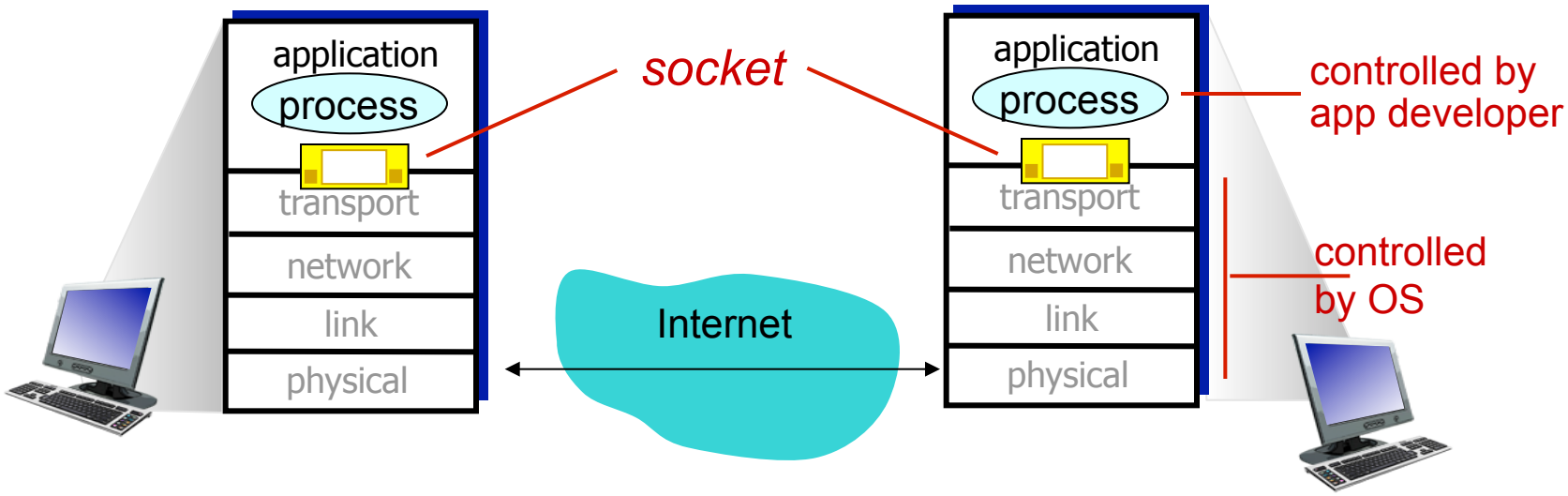
2.6 P2P applications

2.7 socket programming with UDP and 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

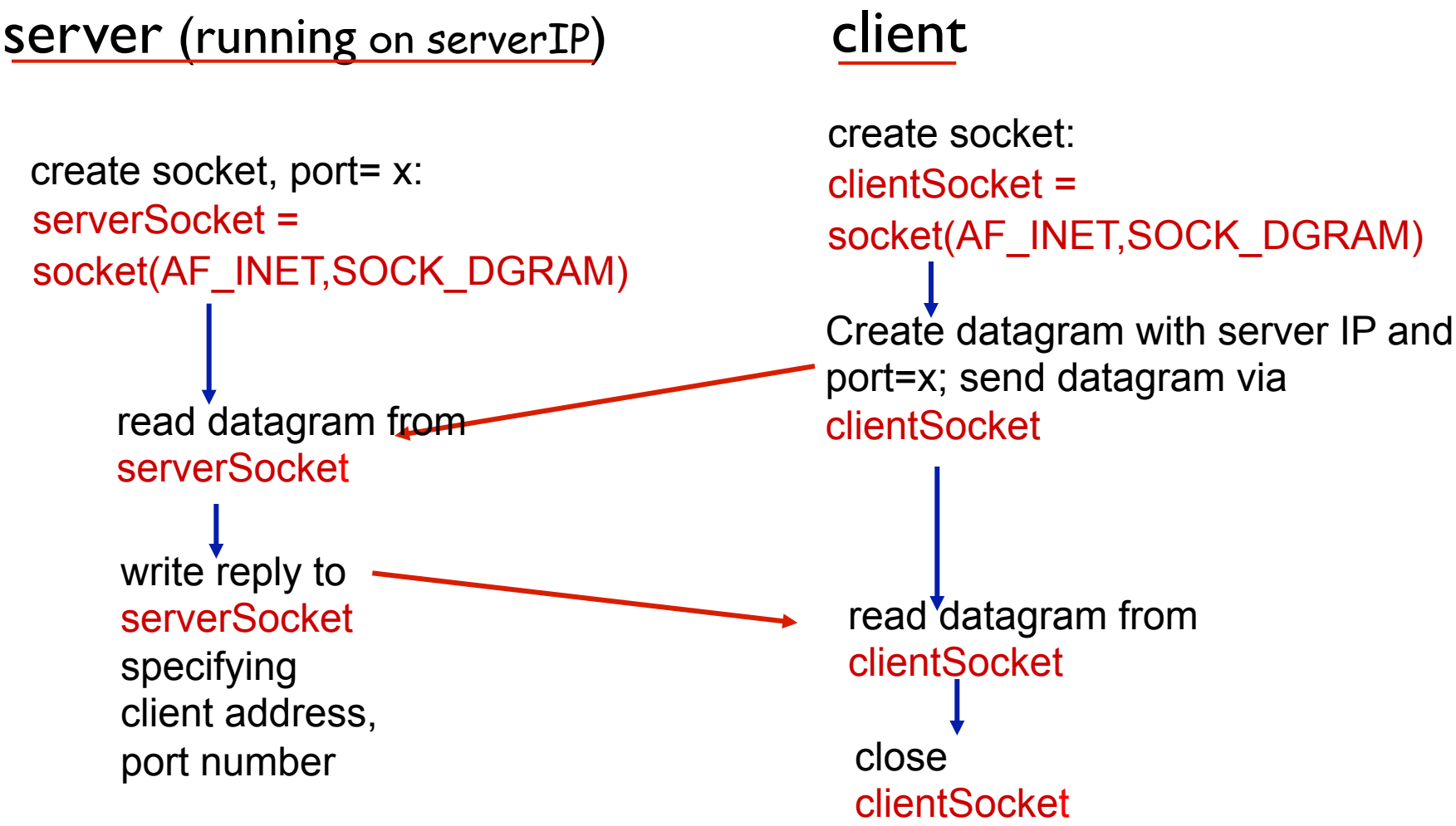
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr extracts sender IP address and port# from received packet

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 app: UDP client

Python UDPClient

include Python's socket library

```
from socket import *
serverName = 'hostname'
serverPort = 12000
```

create UDP socket for server

```
clientSocket = socket(socket.AF_INET,
                      socket.SOCK_DGRAM)
```

get user keyboard input

```
message = raw_input('Input lowercase sentence:')
```

Attach server name, port to message; send into socket

```
clientSocket.sendto(message,(serverName, serverPort))
```

read reply characters from socket into string

```
modifiedMessage, serverAddress =
                      clientSocket.recvfrom(2048)
```

print out received string and close socket

```
print modifiedMessage
clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client's address (client IP and port)

send upper case string back to this client

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:

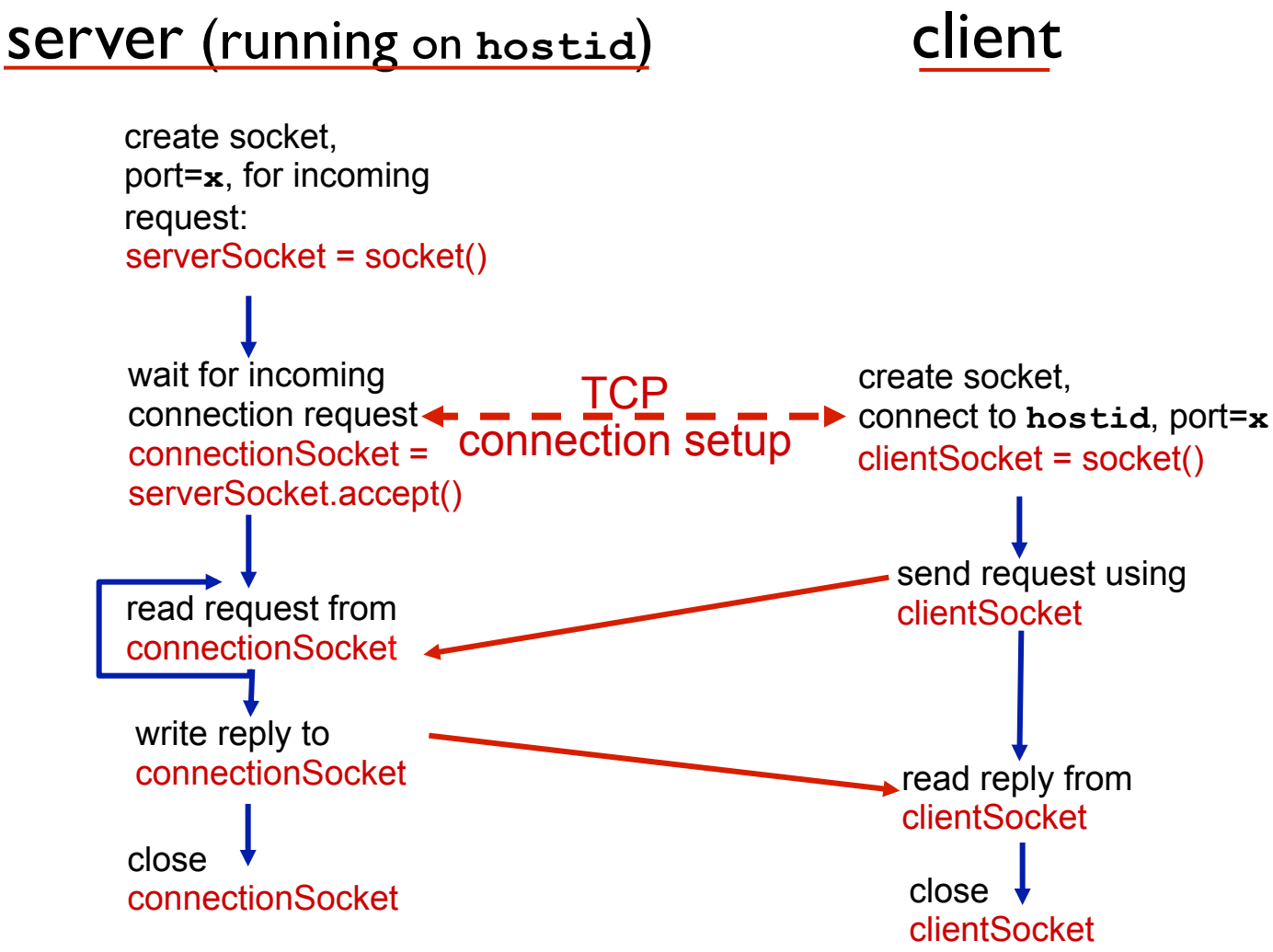
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

create TCP socket for server, remote port 12000

No need to attach server name, port

Example app: TCP server

Python TCPServer

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

Chapter 2: summary

our study of network apps now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
  - P2P: BitTorrent, DHT
- socket programming: TCP, UDP sockets

Chapter 2: summary

most importantly: learned about protocols!

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

important themes:

- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"