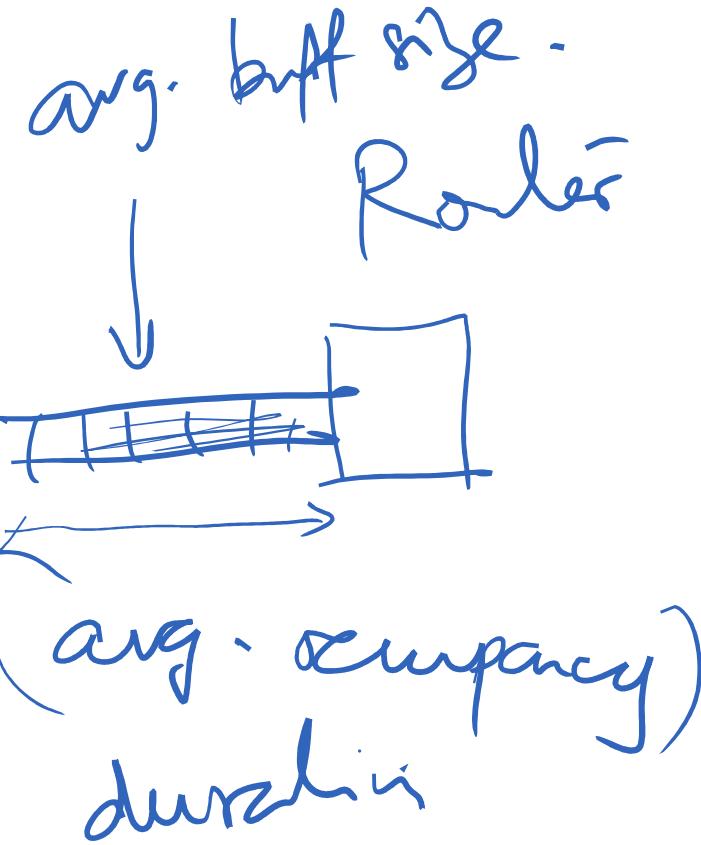
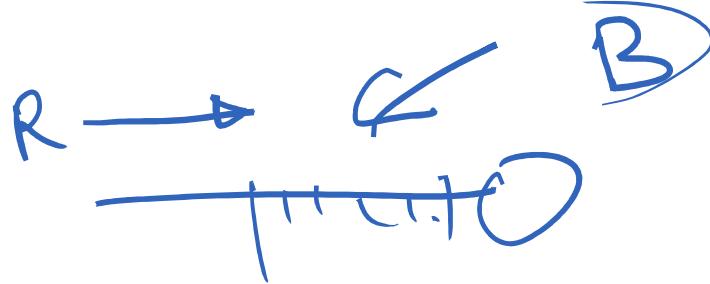


Arrival
Rate



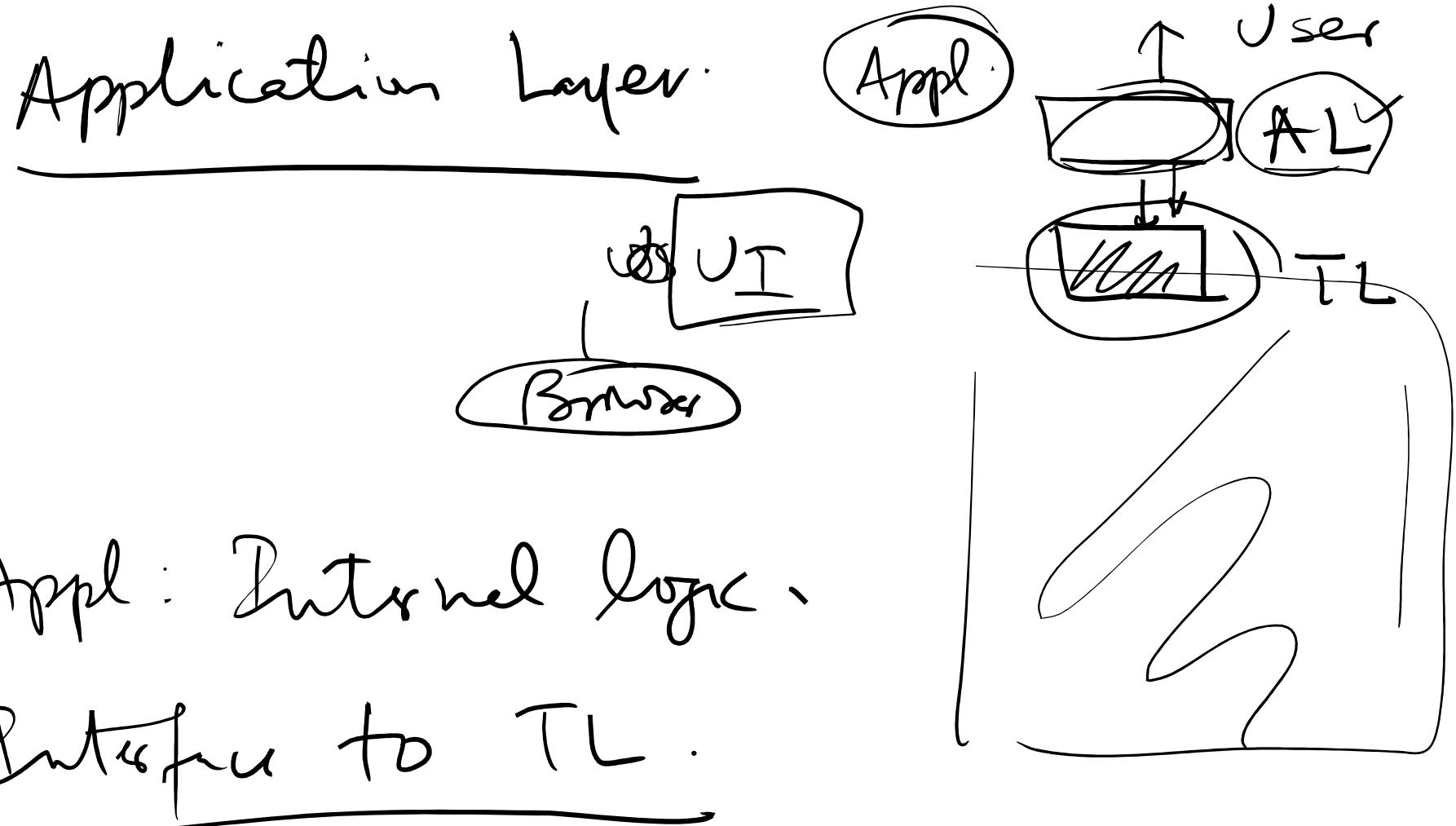


$$R \Delta = B$$

Δ *dashed*

Little's Law:

$$\bar{R} \bar{\Delta} = \bar{B}$$



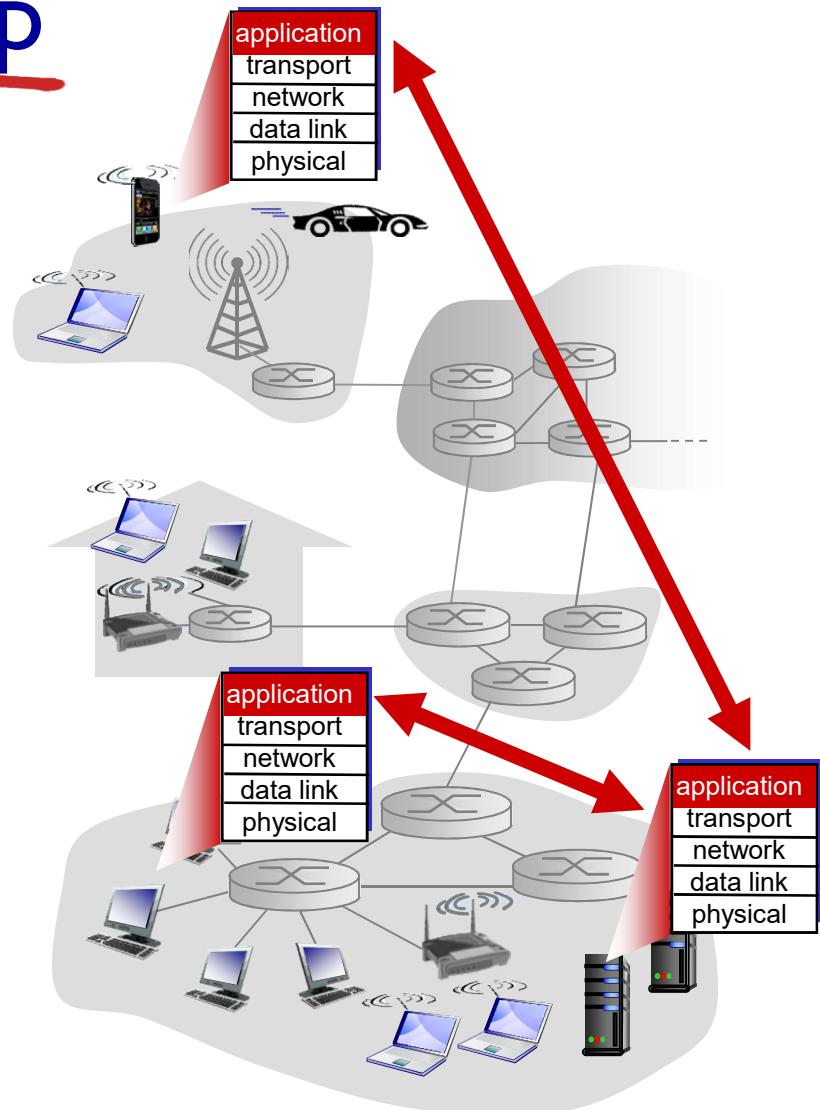
Creating a network app

write programs that:

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

no need to write software for network-core devices

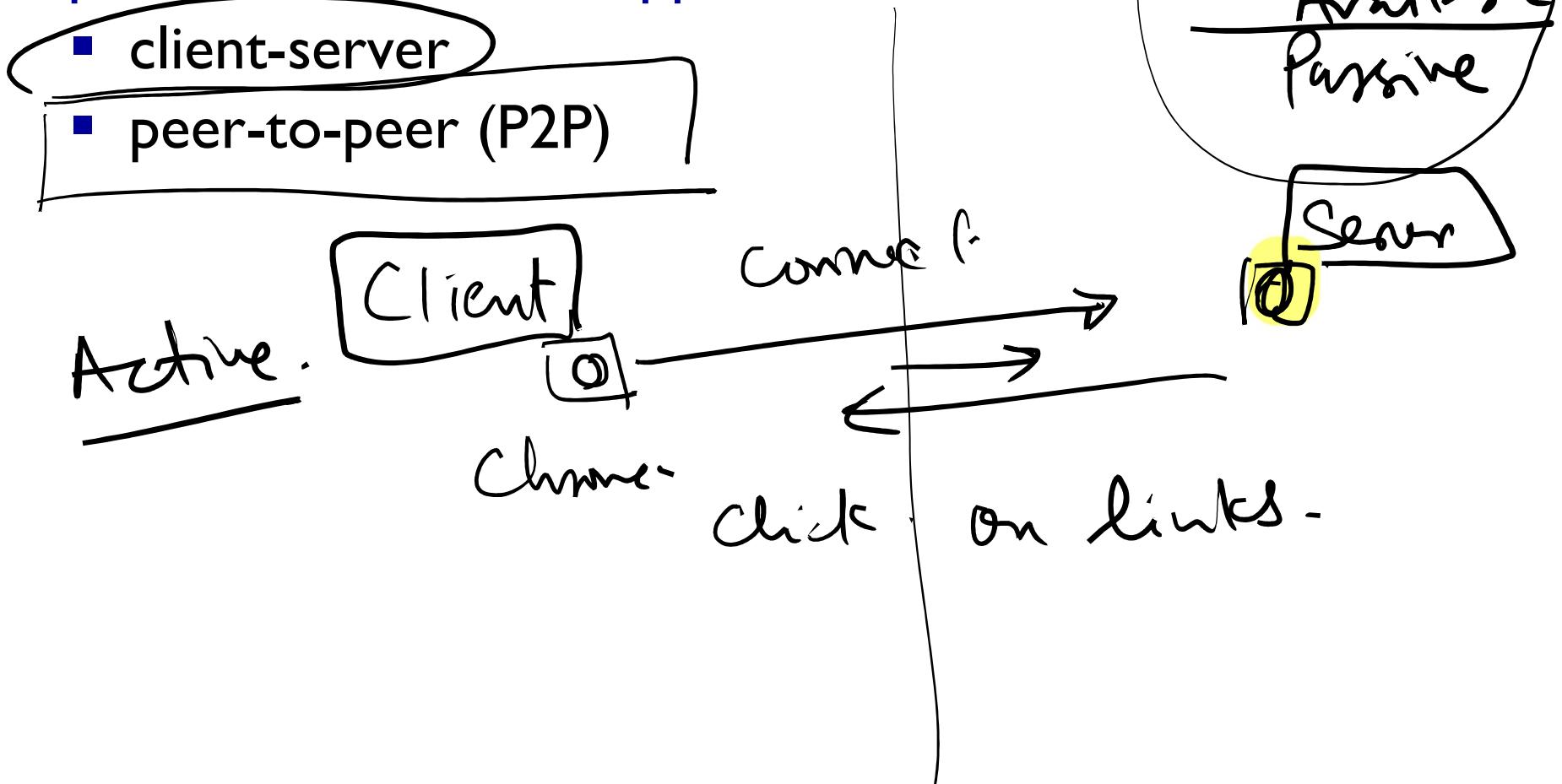
- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



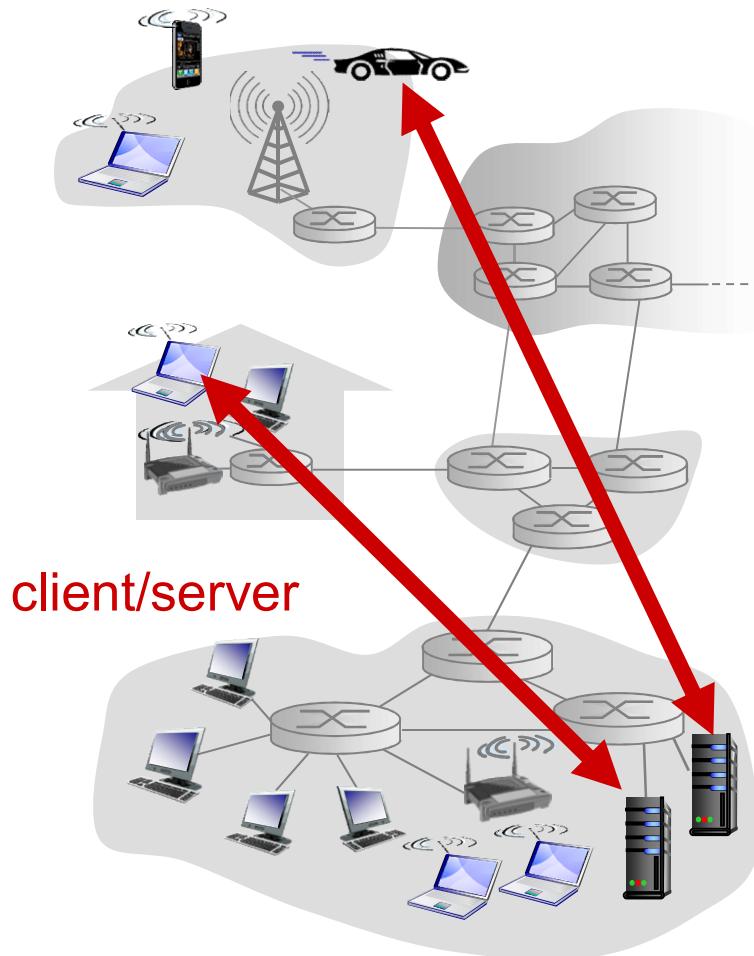
Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)



Client-server architecture



server:

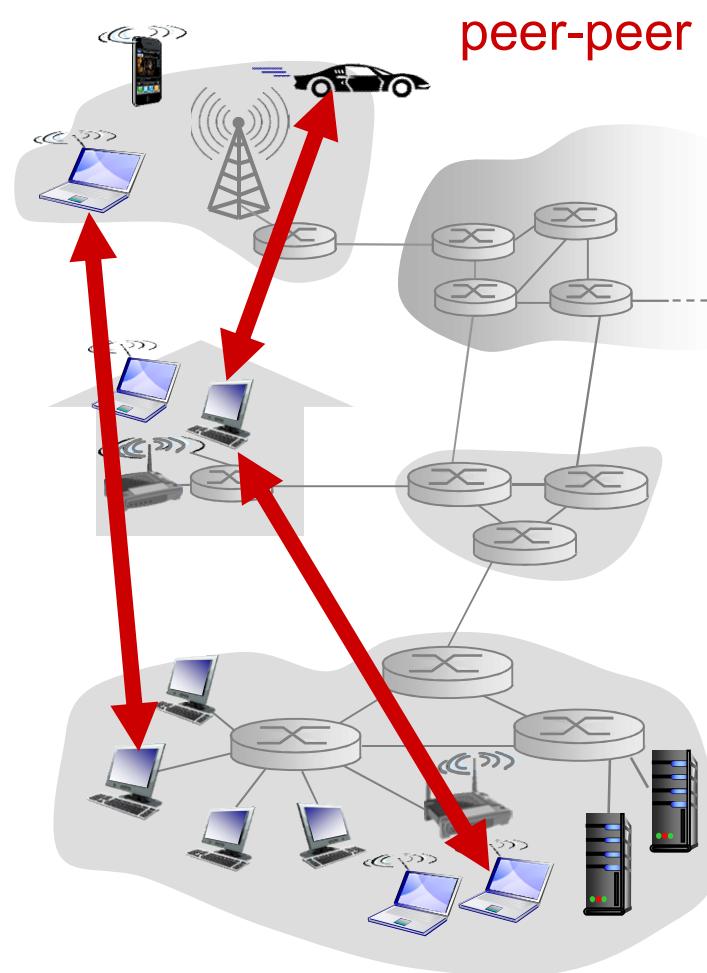
- always-on host
- permanent IP address
- data centers for scaling

clients:

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

P2P architecture

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
 - complex management



Processes communicating

process: program running within a host

- within same host, two processes communicate using **inter-process communication** (defined by OS)
- processes in different hosts communicate by exchanging **messages**

clients, servers

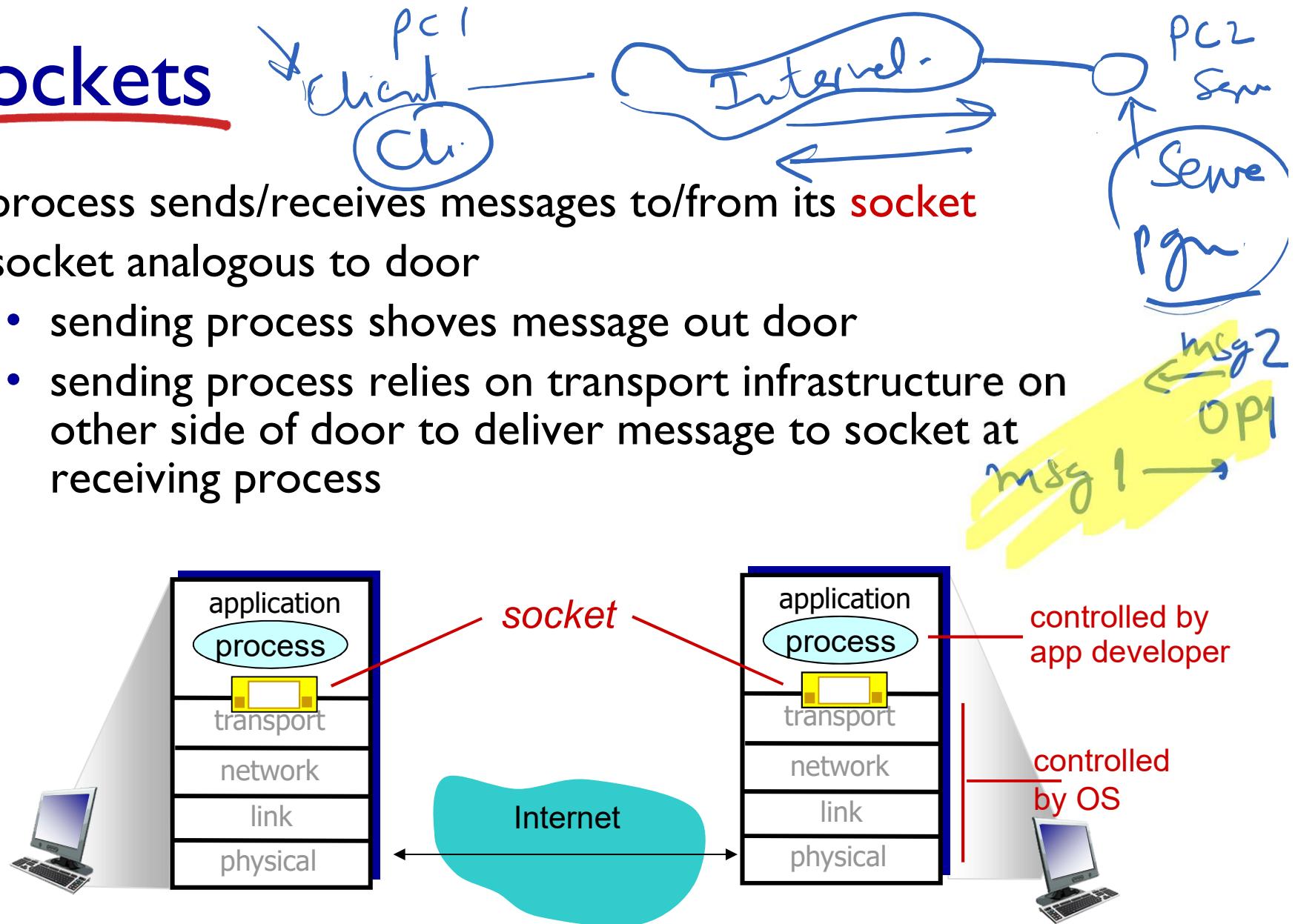
client process: process that initiates communication

server process: process that waits to be contacted

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

Sockets

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



Addressing processes

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

App-layer protocol defines

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

open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

NL → Unreliable
→ drops pkt
→ Err
→ Misrouted

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% **reliable data transfer**
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require **low delay** to be “effective”

Interactive

throughput

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

security

- encryption, data integrity, ...

Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	yes, 100's msec
stored audio/video	loss-tolerant	same as above	
interactive games	loss-tolerant	few kbps up	yes, few secs
text messaging	no loss	elastic	yes, 100's msec yes and no

Internet transport protocols services

TCP service:

- ✓ **reliable transport** between sending and receiving process
- ✓ **flow control**: sender won't overwhelm receiver
- **congestion control**: throttle sender when network overloaded
- **does not provide**: timing, minimum throughput guarantee, security
- ✗ **connection-oriented**: setup required between client and server processes

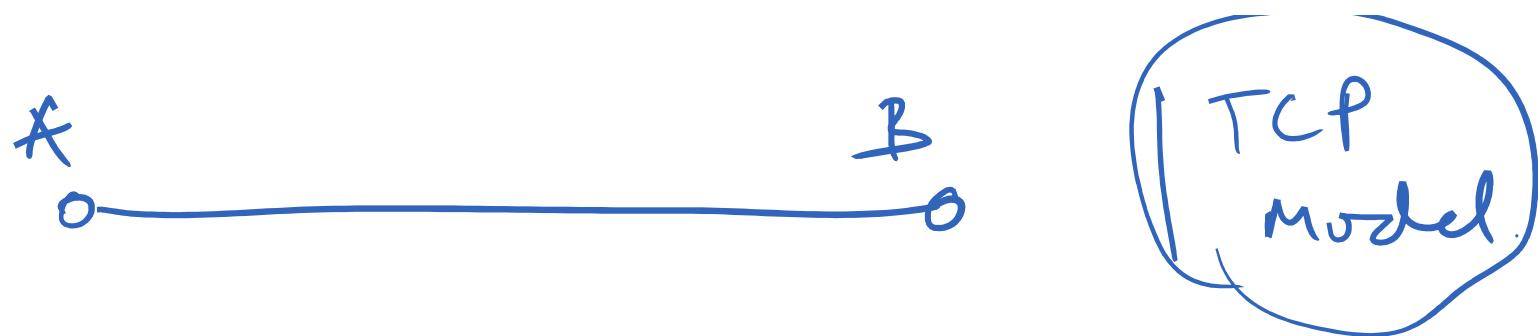
UDP service:

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

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

	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



protocol. msgs), associated

Request - Response protocol.

Q1: R we going to move +

A1 :

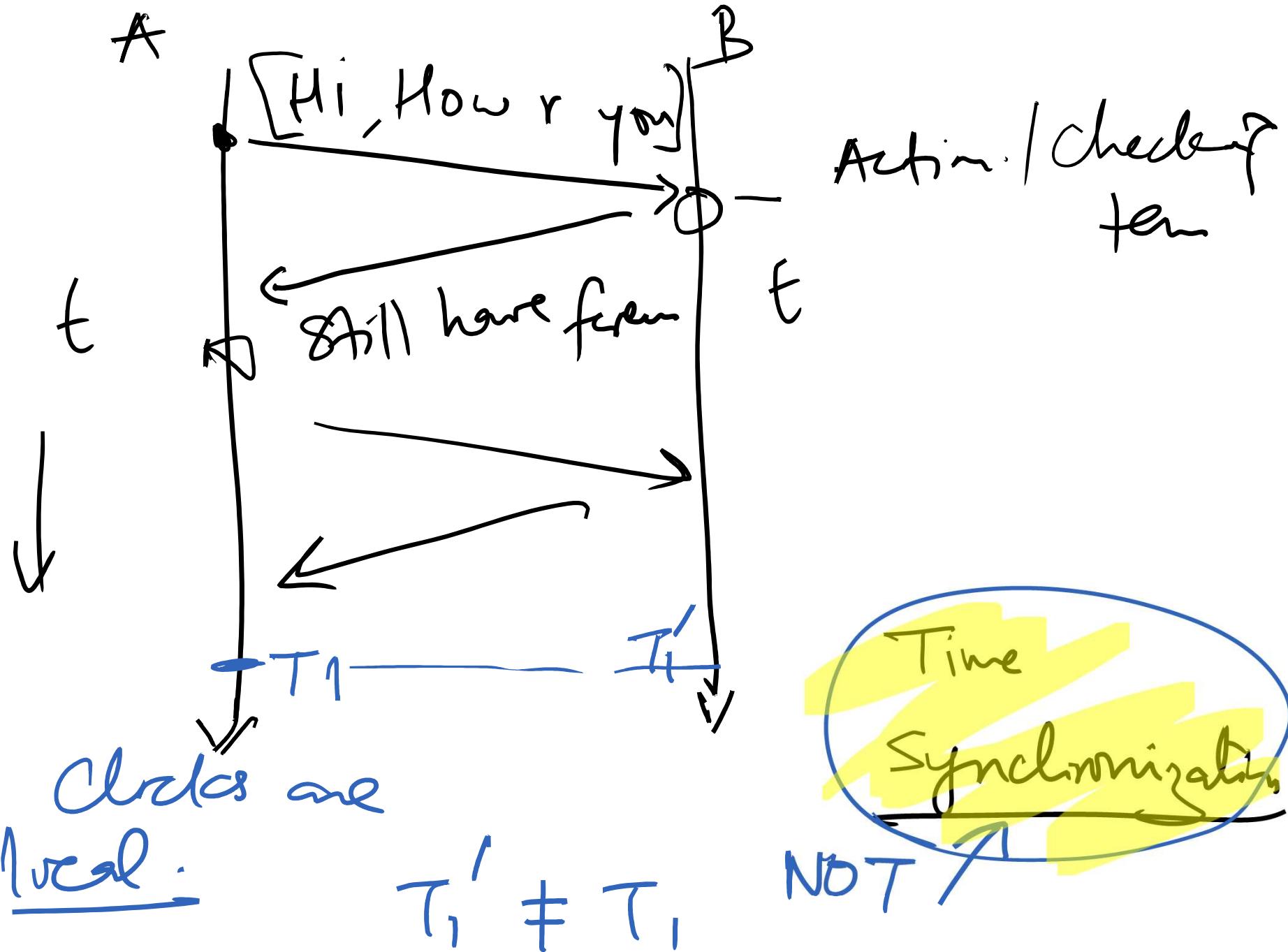
Yes.

NO

stateless

work online
~~R~~ (Chk for tickets)

stop



Msg ↗

Structure

"Simple"

parsing

decoding

Are you going to movies?

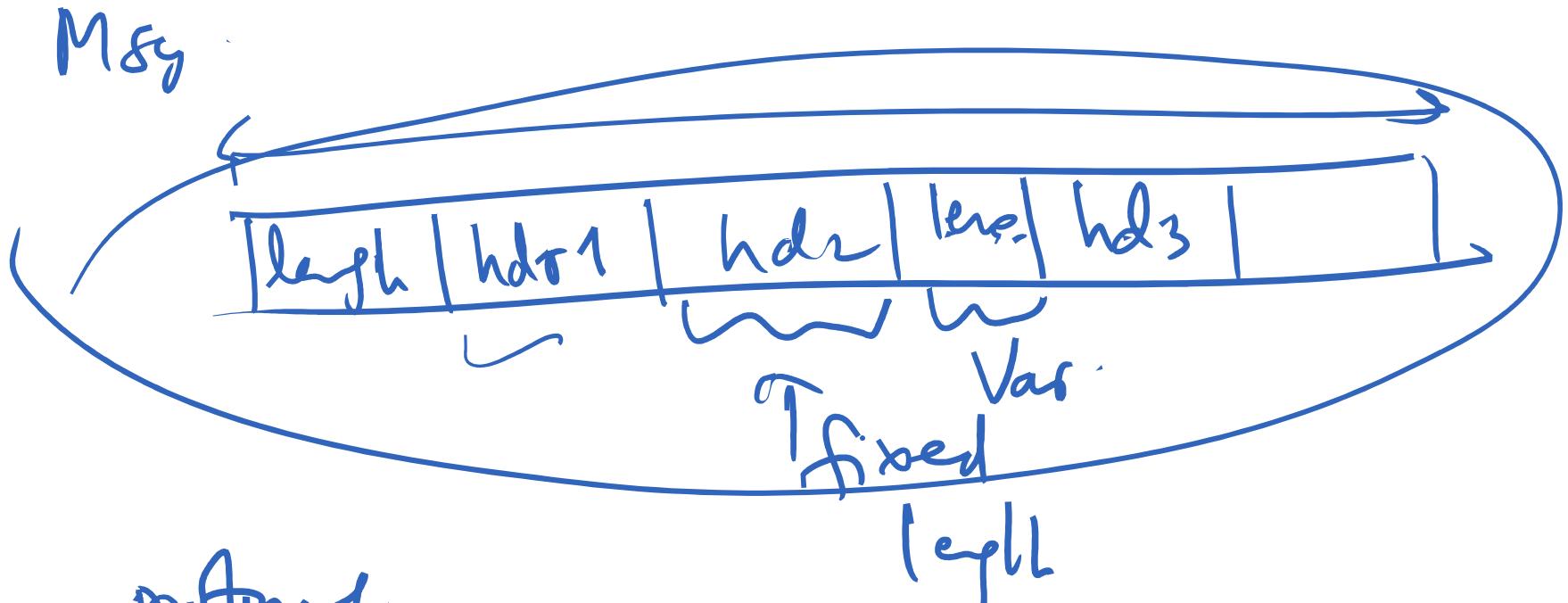
A_i

Msg ↘

t₁ t₂ t₃ - - -

sep

Length | msg



protocol:

both sides agree
to syntax / semantics.

Securing TCP

TCP & UDP

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext

SSL

- provides encrypted TCP connection
- data integrity
- end-point authentication

SSL is at app layer

- apps use SSL libraries, that “talk” to TCP

SSL socket API

- cleartext passwords sent into socket traverse Internet encrypted
- see Chapter 8

Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

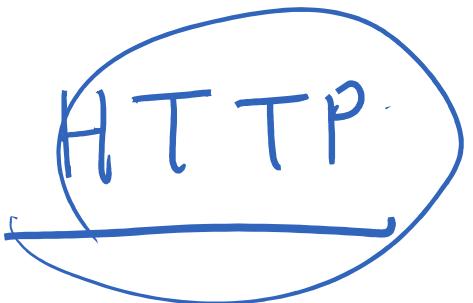
2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

Web browser



Web and HTTP

First, a review...

- web page consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,...
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL*, e.g.,

www.someschool.edu/someDept/pic.gif

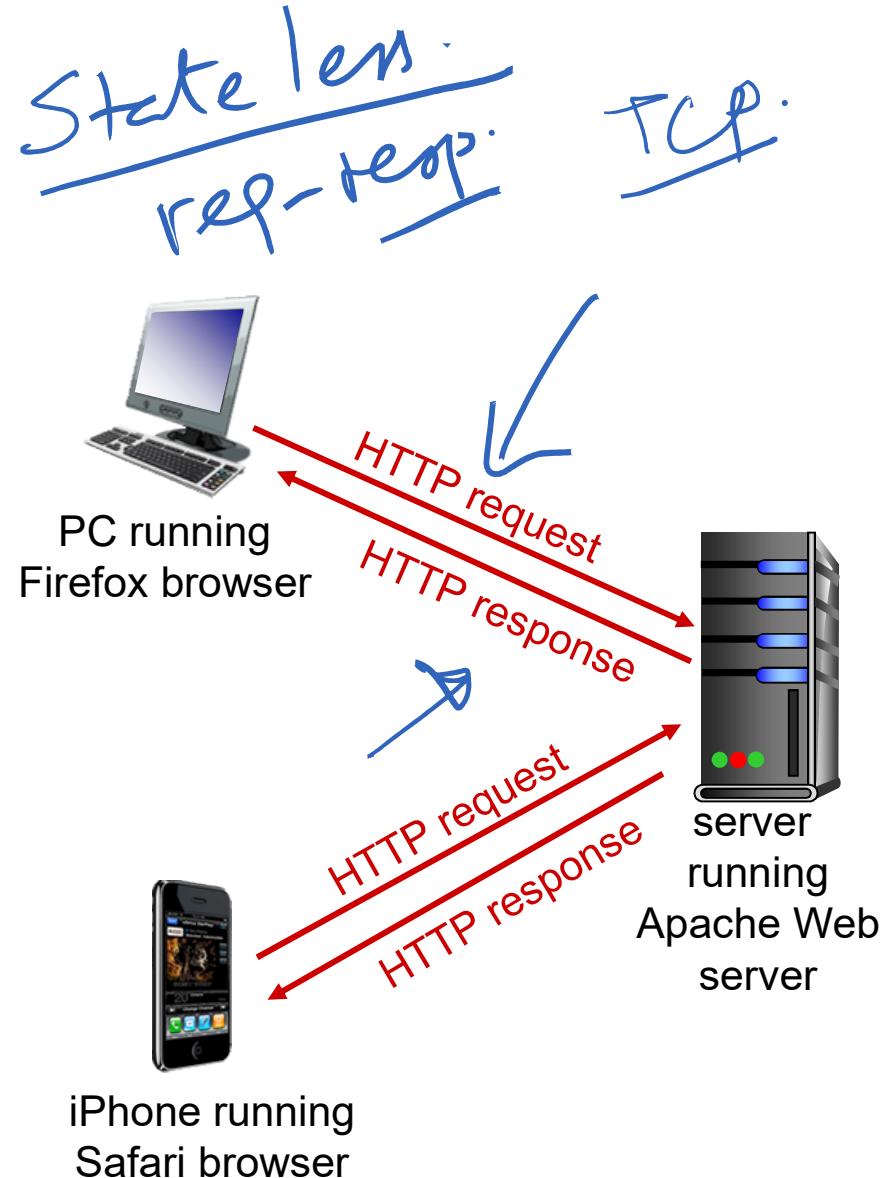
host name

path name

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



HTTP overview (continued)

uses *TCP*:

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

TCP not used
M media ←
real-time ←
*Many to many
appl.*

HTTP is “stateless”

- server maintains no information about past client requests

aside

protocols that maintain “state” are complex!

- past history (state) must be maintained
- if server/client crashes, their views of “state” may be inconsistent, must be reconciled

Server:

HTTP connections

non-persistent HTTP

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

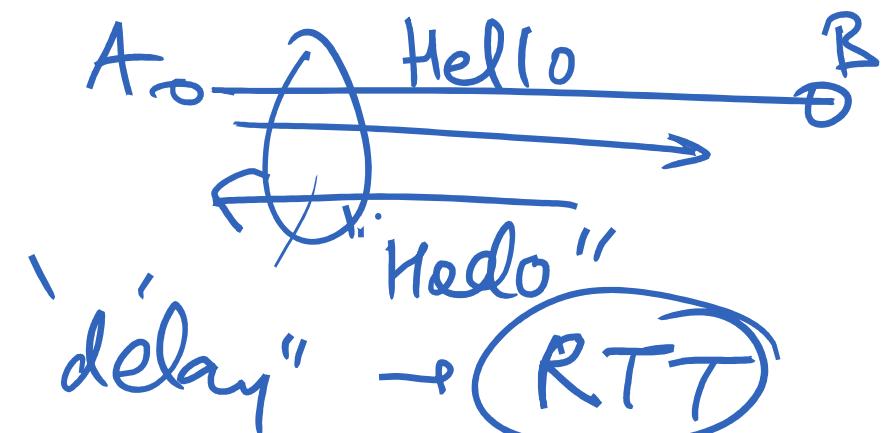
$2 \times RTT +$

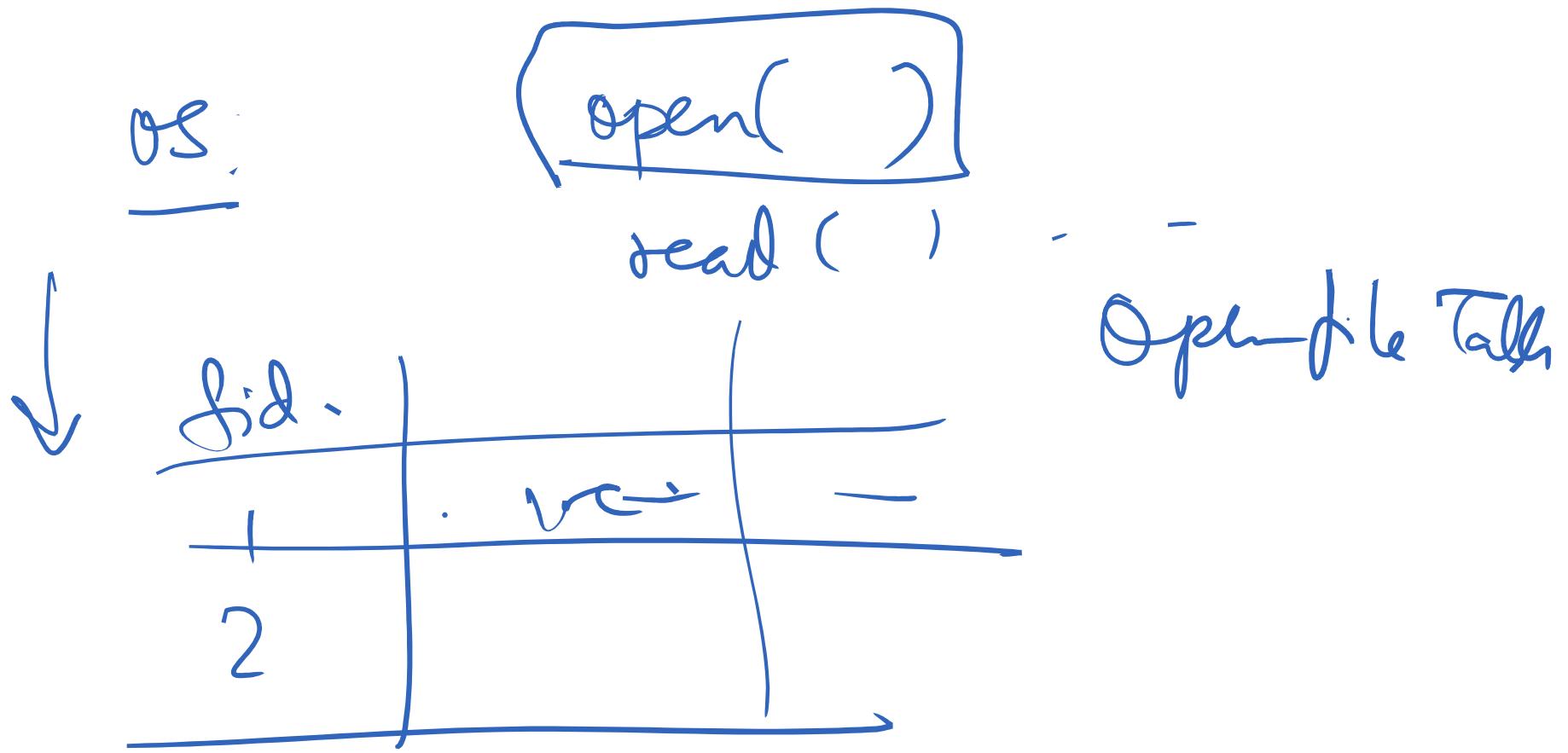
[Obj transmission time]

persistent HTTP

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

TCP : Conn · miented





Non-persistent HTTP

suppose user enters URL:

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

(contains text,
references to 10
jpeg images)

-
- time ↓
- 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

Non-persistent HTTP (cont.)

time
↓

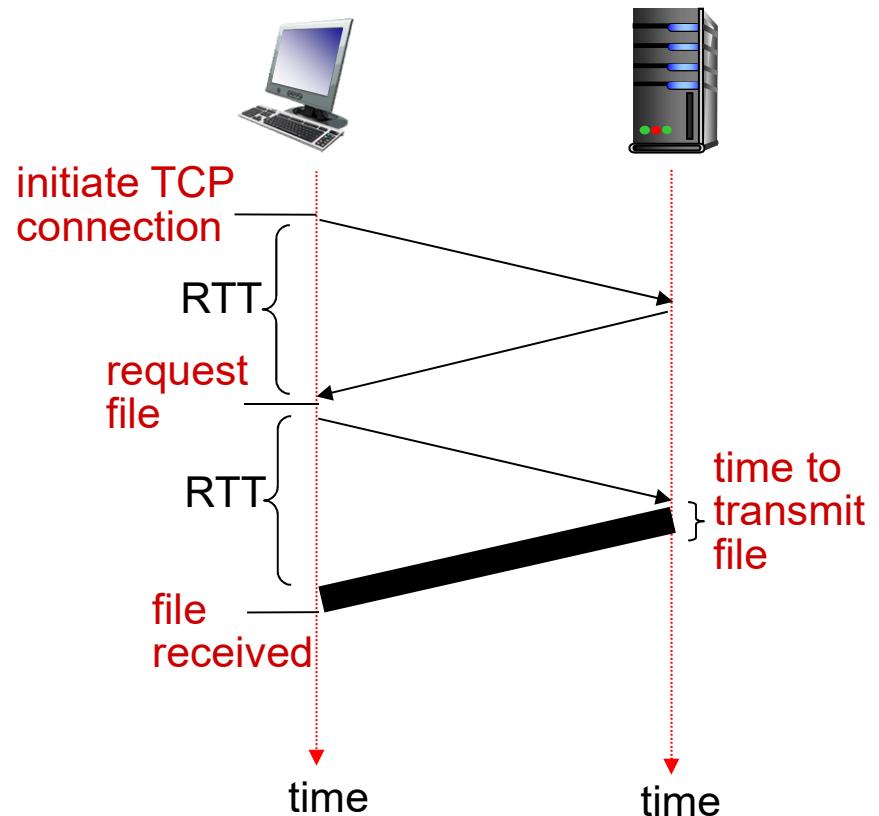
4. HTTP server closes TCP connection.
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects
6. Steps 1-5 repeated for each of 10 jpeg objects

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 = $2\text{RTT} + \text{file transmission time}$



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)

request line
(GET, POST,
HEAD commands)

header lines

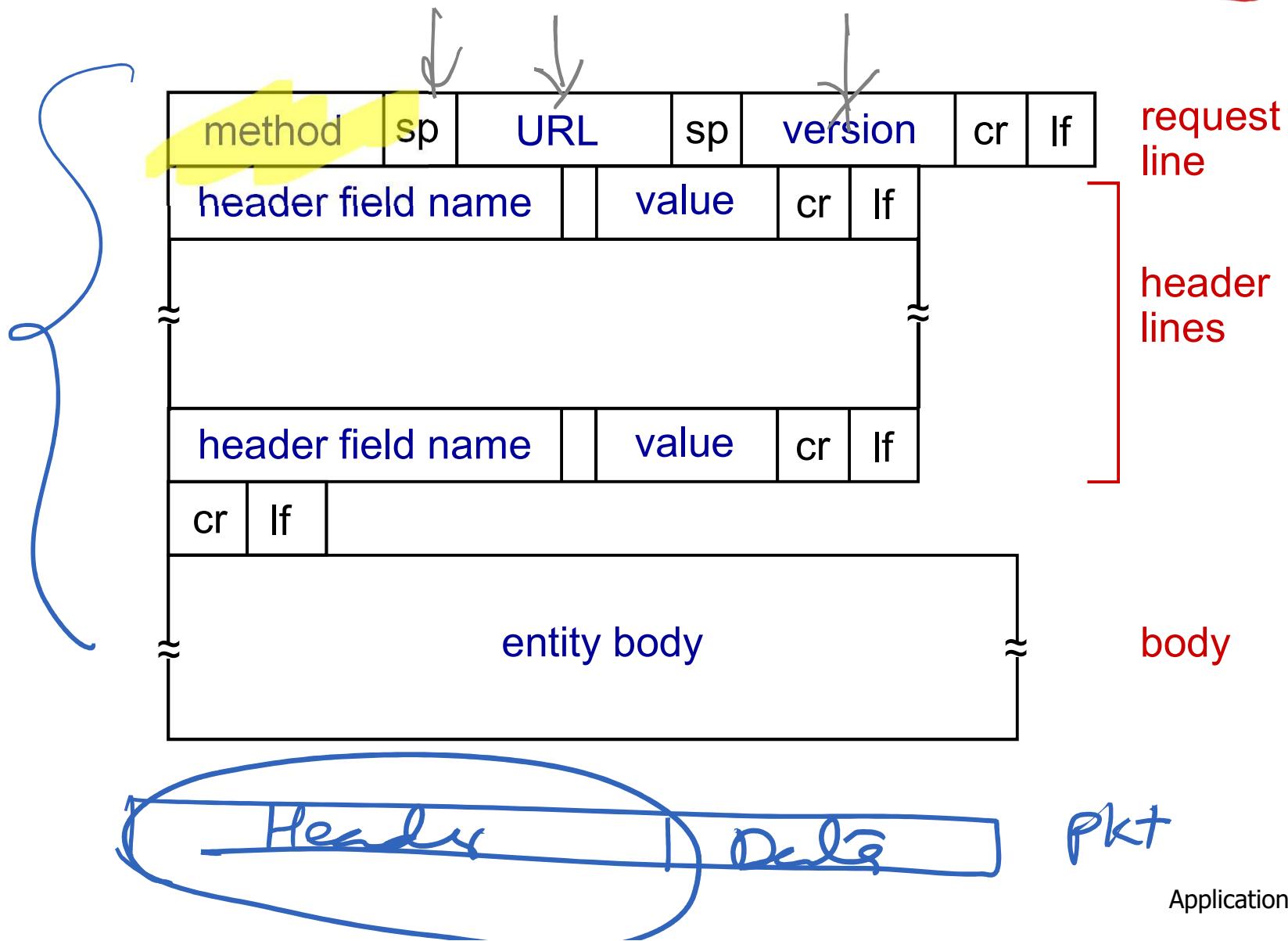
carriage return,
line feed at start
of line indicates
end of header lines

The diagram illustrates an HTTP request message. On the left, handwritten annotations explain the components: 'request line (GET, POST, HEAD commands)' points to the first line, 'header lines' points to the following lines, and 'carriage return, line feed at start of line indicates end of header lines' points to the final '\r\n'. The request message itself is shown in yellow and green boxes. The yellow box contains the request line: 'GET /index.html HTTP/1.1\r\n'. The green box contains the header lines: 'Host: www-net.cs.umass.edu\r\n', 'User-Agent: Firefox/3.6.10\r\n', 'Accept: text/html,application/xhtml+xml\r\n', 'Accept-Language: en-us,en;q=0.5\r\n', 'Accept-Encoding: gzip,deflate\r\n', 'Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n', 'Keep-Alive: 115\r\n', and 'Connection: keep-alive\r\n'. Arrows from the right side point to the '\r\n' character in the request line and the '\r\n' at the end of the header lines, labeled 'carriage return character' and 'line-feed character' respectively.

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP request message: general format



Uploading form input

POST method:

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

URL method:

- uses GET method
- input is uploaded in URL field of request line:

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

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 response message

status line
(protocol
status code
status phrase)

header
lines

data, e.g.,
requested
HTML file

```
HTTP/1.1 200 OK\r\nDate: Sun, 26 Sep 2010 20:09:20 GMT\r\nServer: Apache/2.0.52 (CentOS) \r\nLast-Modified: Tue, 30 Oct 2007 17:00:02  
GMT\r\nETag: "17dc6-a5c-bf716880"\r\nAccept-Ranges: bytes\r\nContent-Length: 2652\r\nKeep-Alive: timeout=10, max=100\r\nConnection: Keep-Alive\r\nContent-Type: text/html; charset=ISO-8859-  
1\r\n\r\ndata data data data data ...
```

* Check out the online interactive exercises for more
examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

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

Trying out HTTP (client side) for yourself

I. Telnet to your favorite Web server:

```
telnet gaia.cs.umass.edu 80
```

{ opens TCP connection to port 80
(default HTTP server port)
at gaia.cs.umass.edu.
anything typed in will be sent
to port 80 at gaia.cs.umass.edu

2. type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
```

```
Host: gaia.cs.umass.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!

(or use Wireshark to look at captured HTTP request/response)

State-less protocol → server scaling ↑

login:

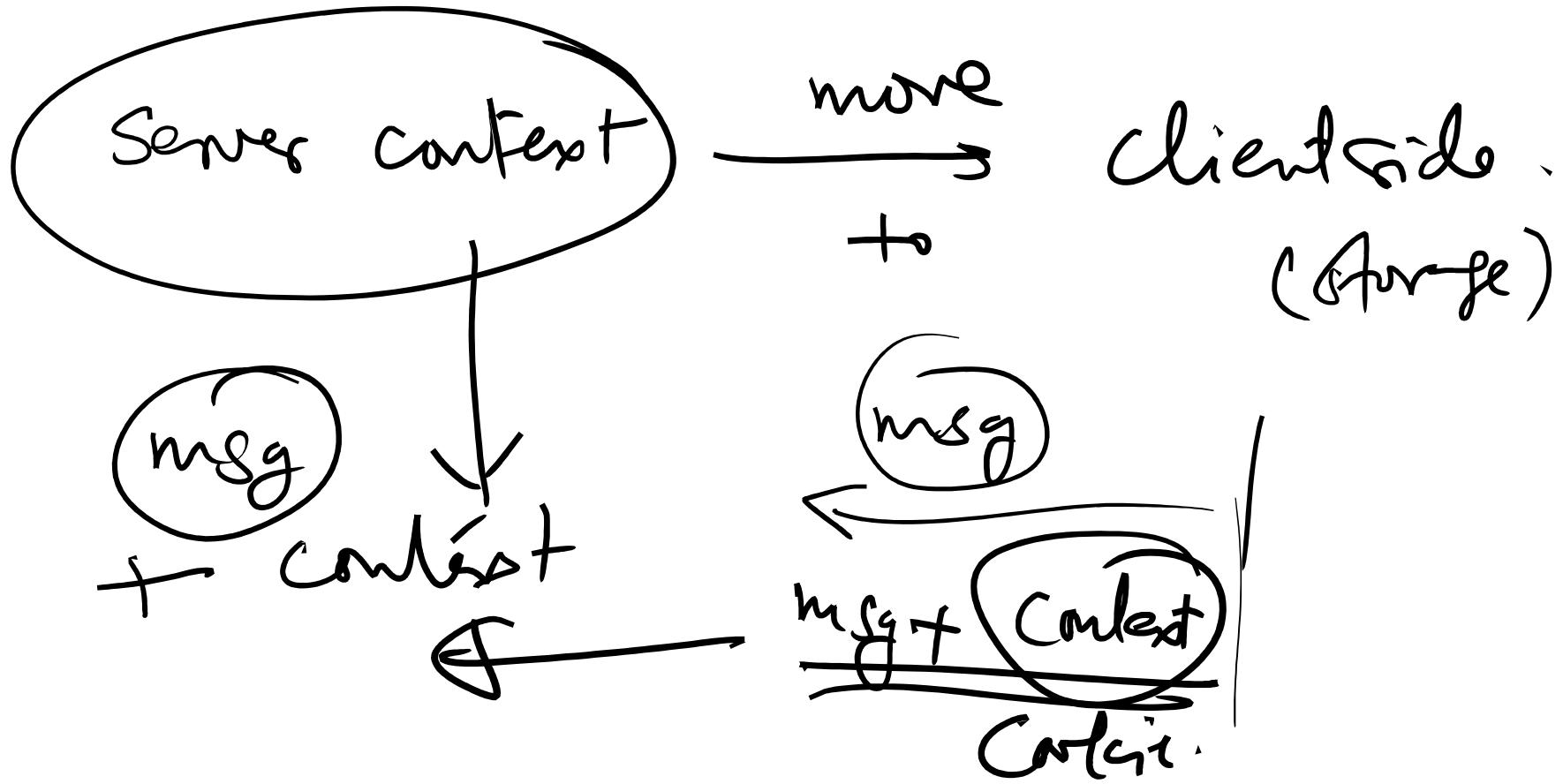
login id.

password

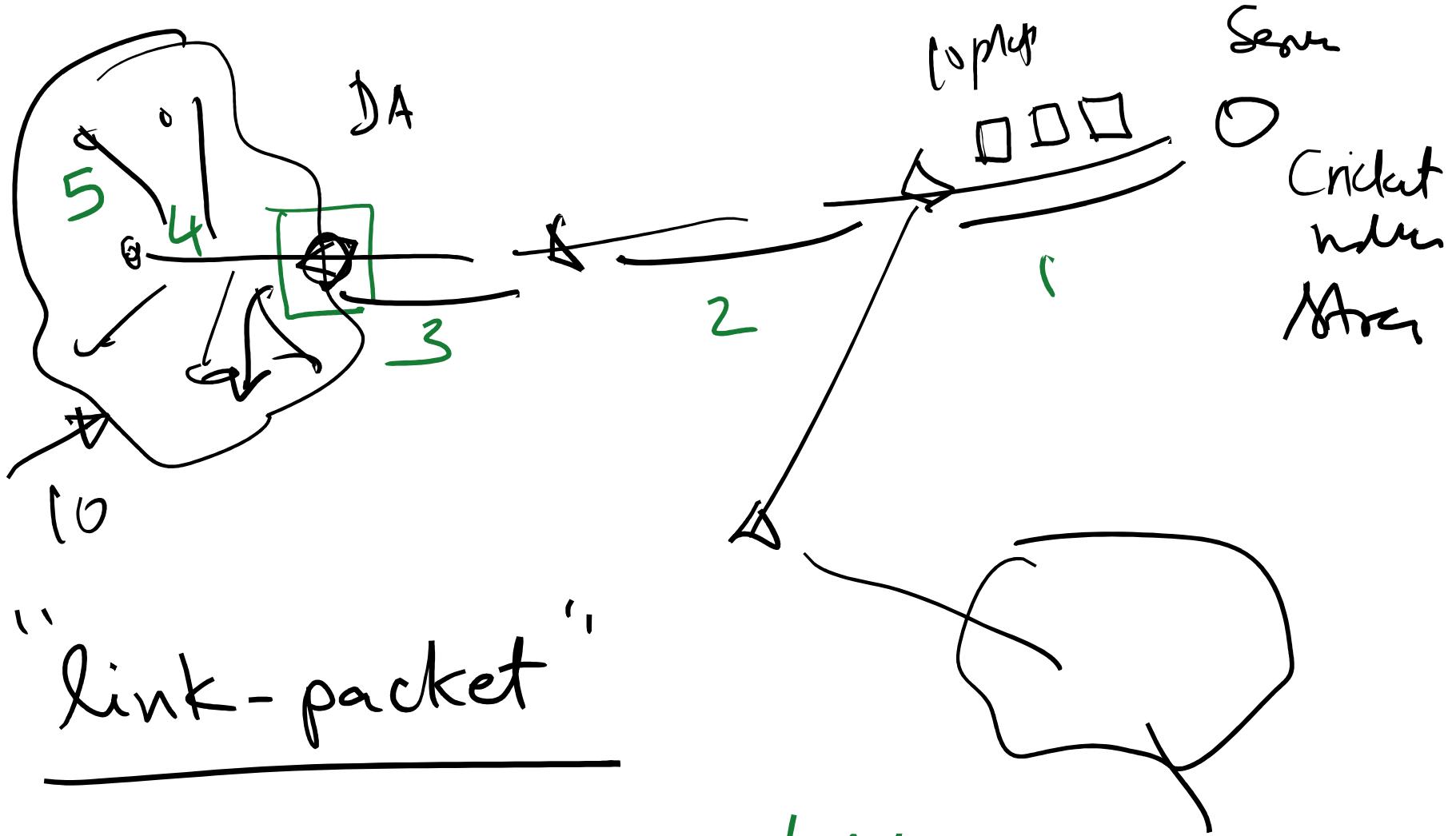
ok

} valid.

[get file.]



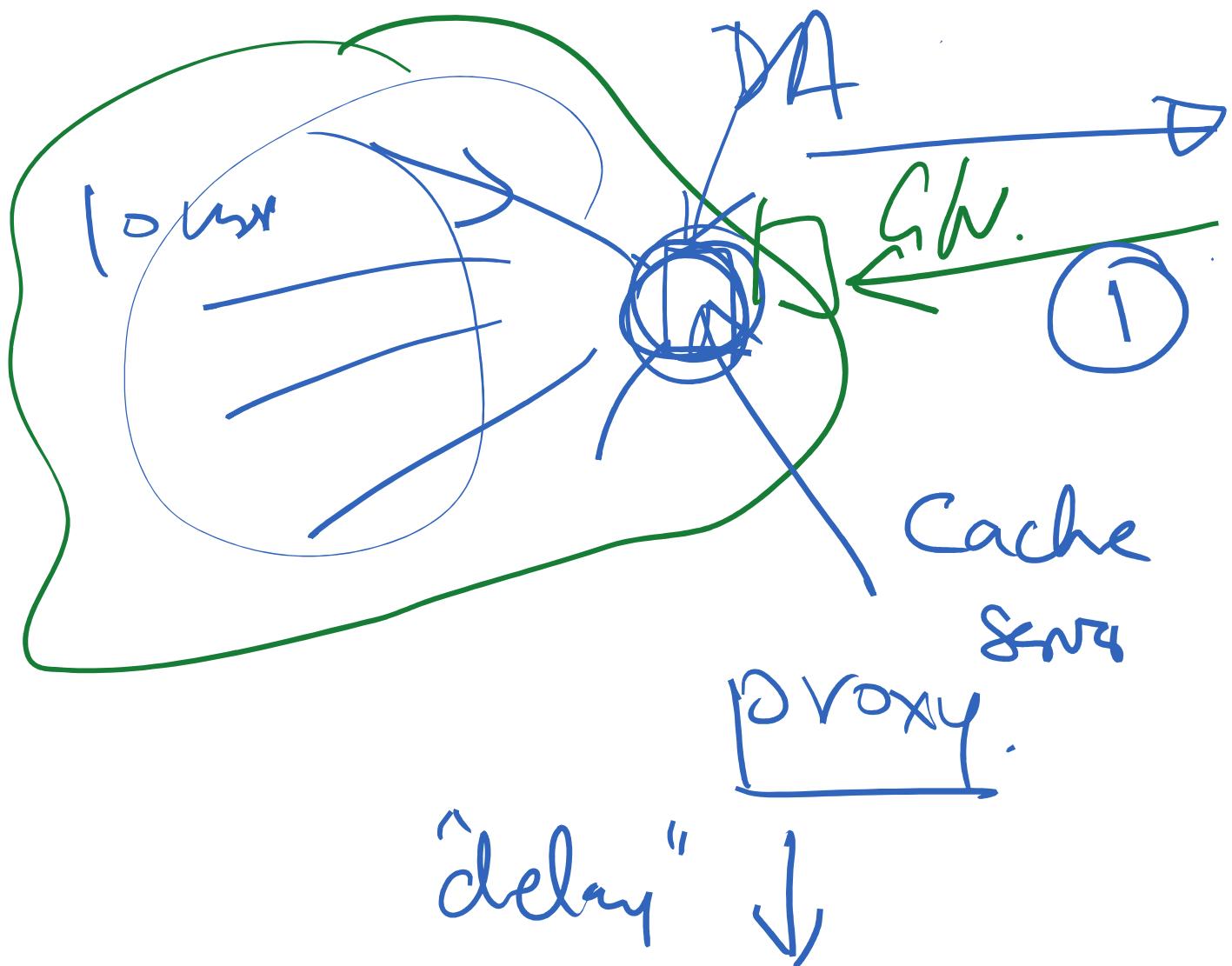
client



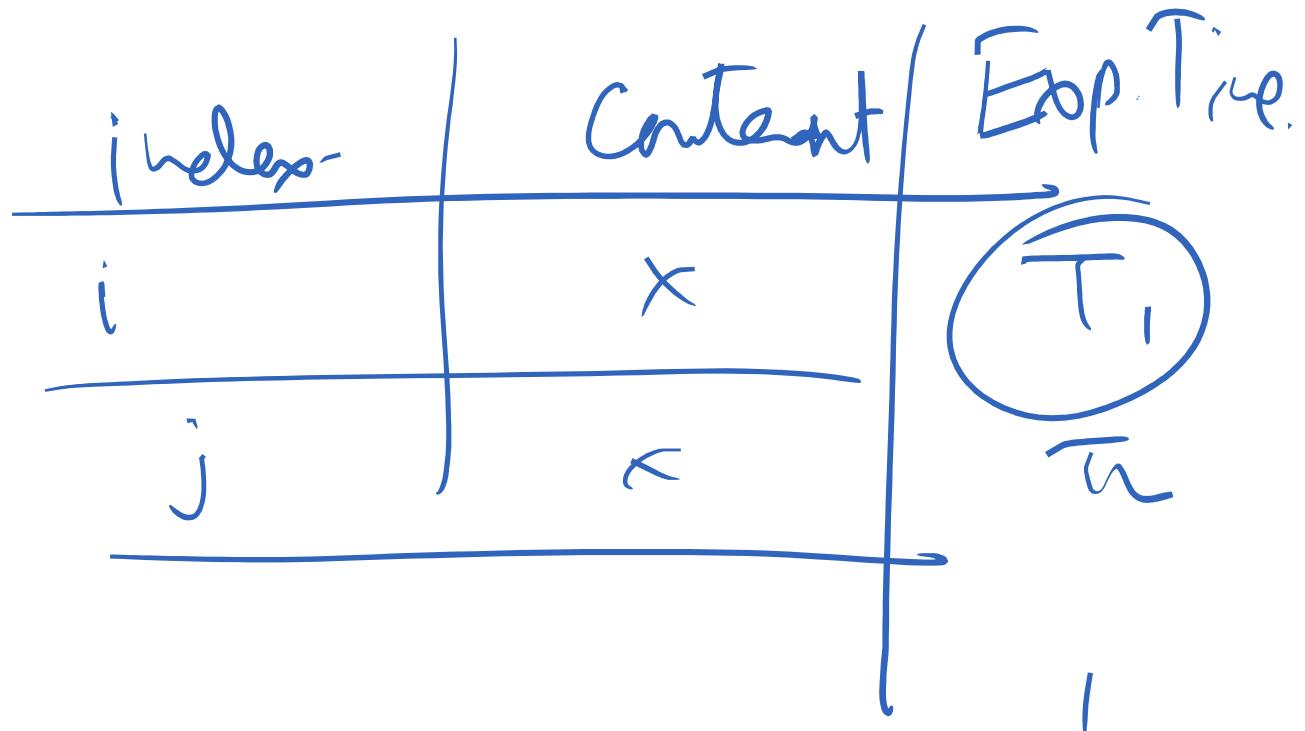
"link-packet"

5 links \times 10 copies/pkt

"50 link pkts."/ms



Cache.



HTTP :

GET.

CONDITIONAL

GET.

$$T > T_0$$

User-server state: **cookies**



many Web sites use cookies

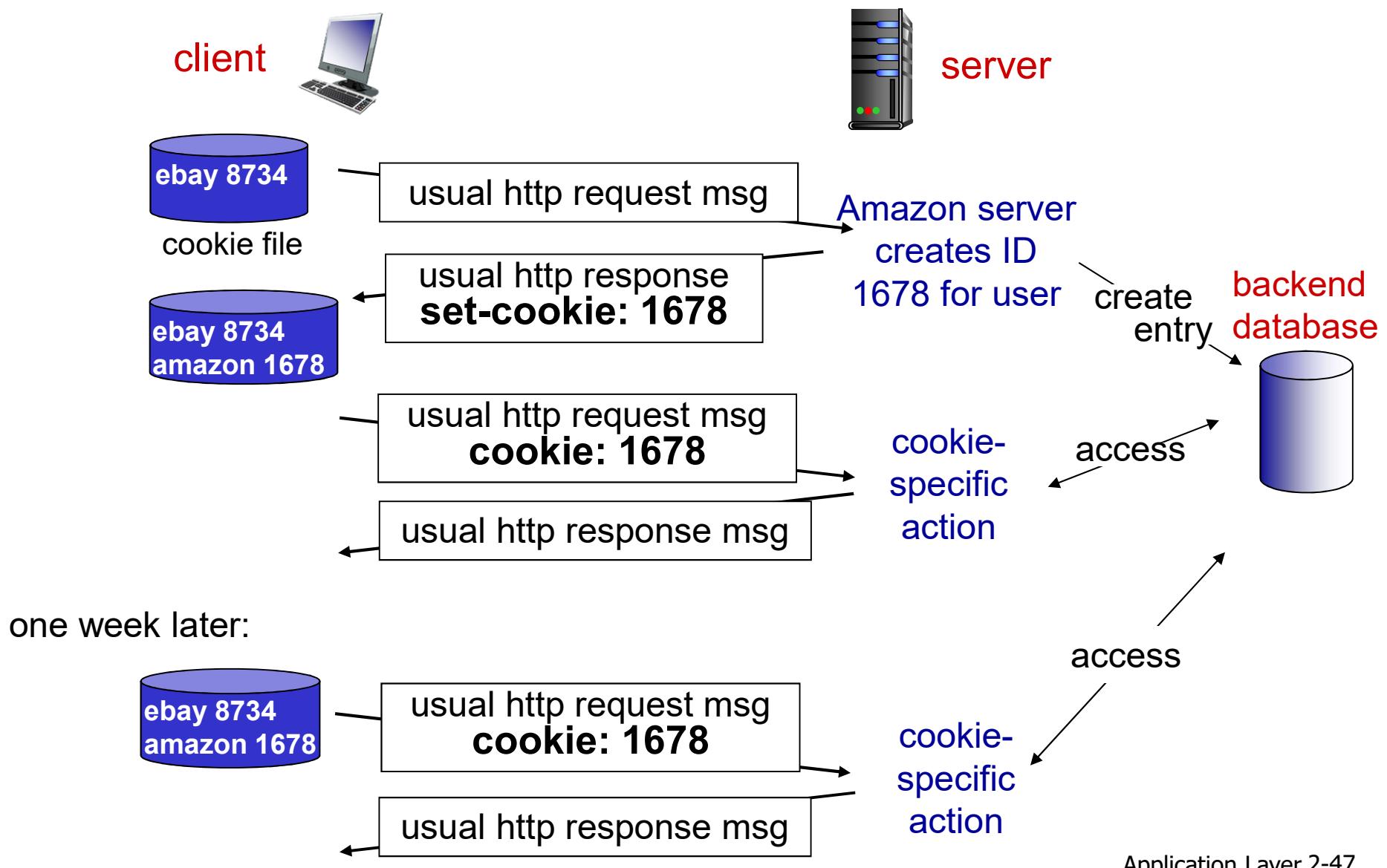
four components:

- 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)

how to keep “state”:

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

aside

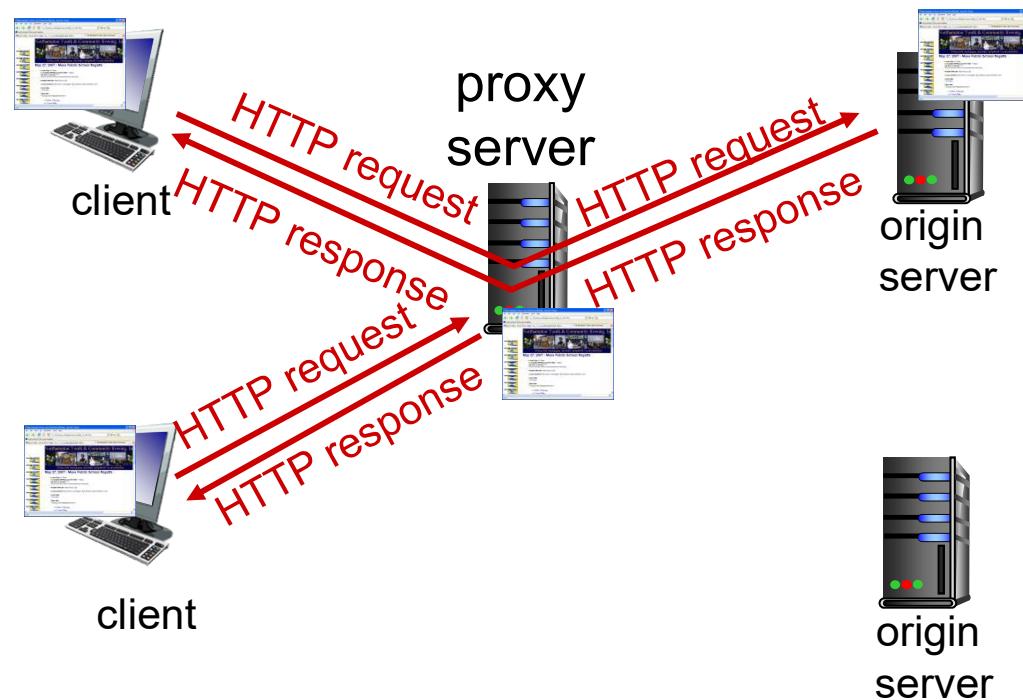
cookies and privacy:

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

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)

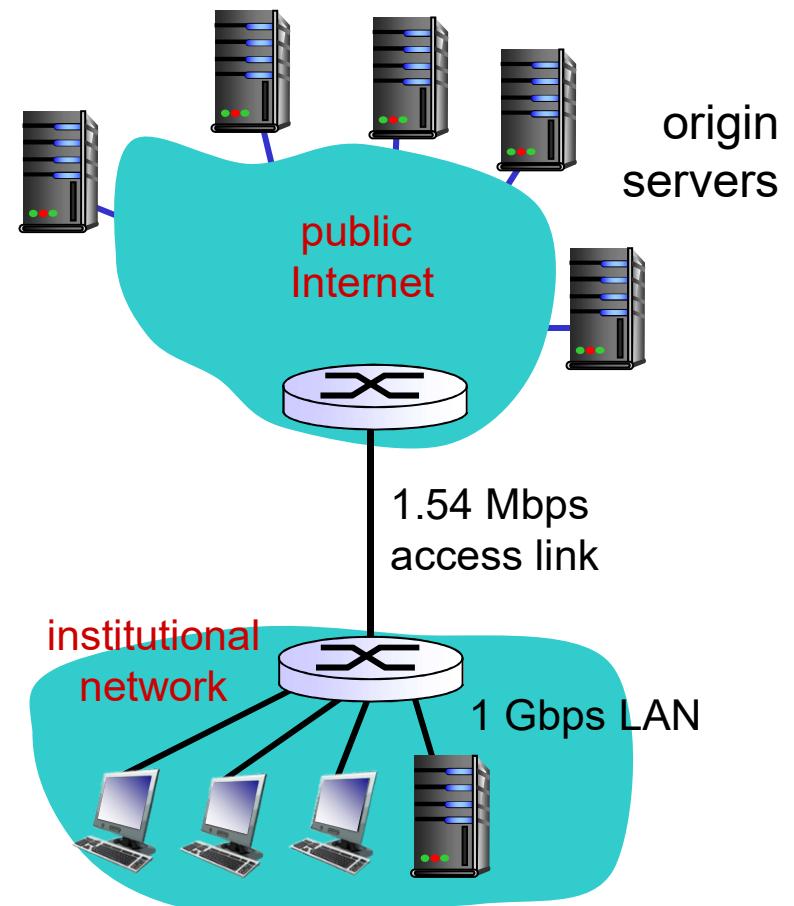
Caching example:

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% *problem!*
- access link utilization = **99%**
- total delay = Internet delay + access delay + LAN delay
= 2 sec + minutes + usecs



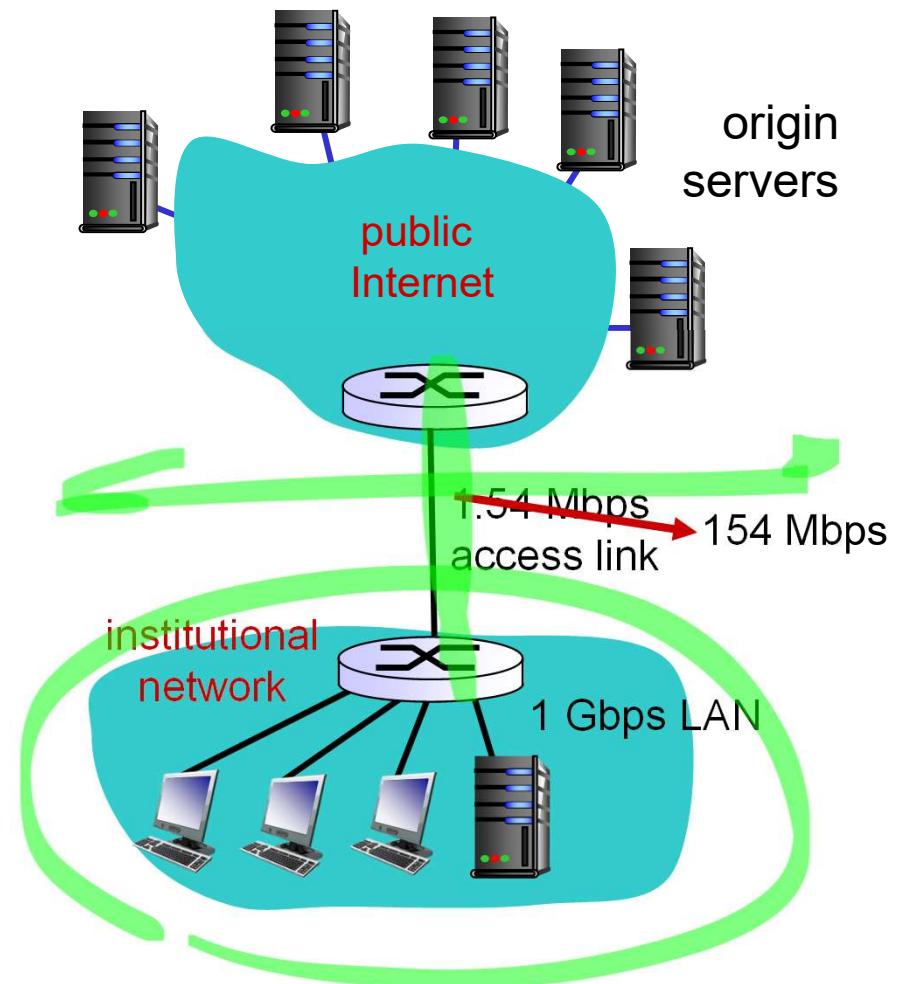
Caching example: fatter access link

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
 $= 2 \text{ sec} + \cancel{\text{minutes}} + \cancel{\text{usecs}}$
msecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

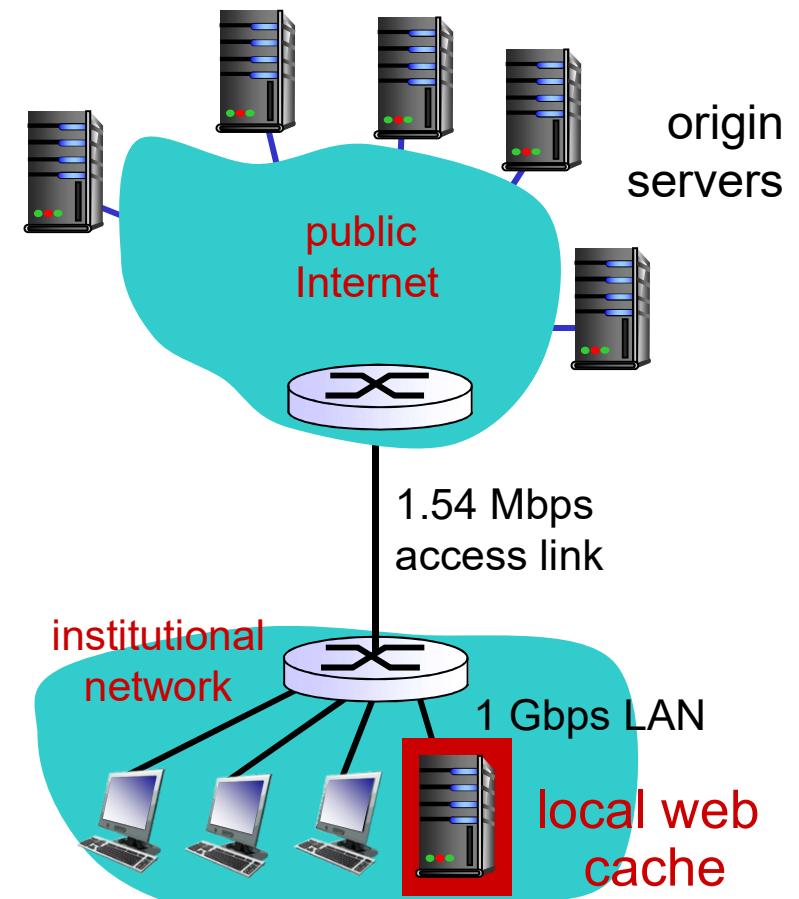
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

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

How to compute link utilization, delay?

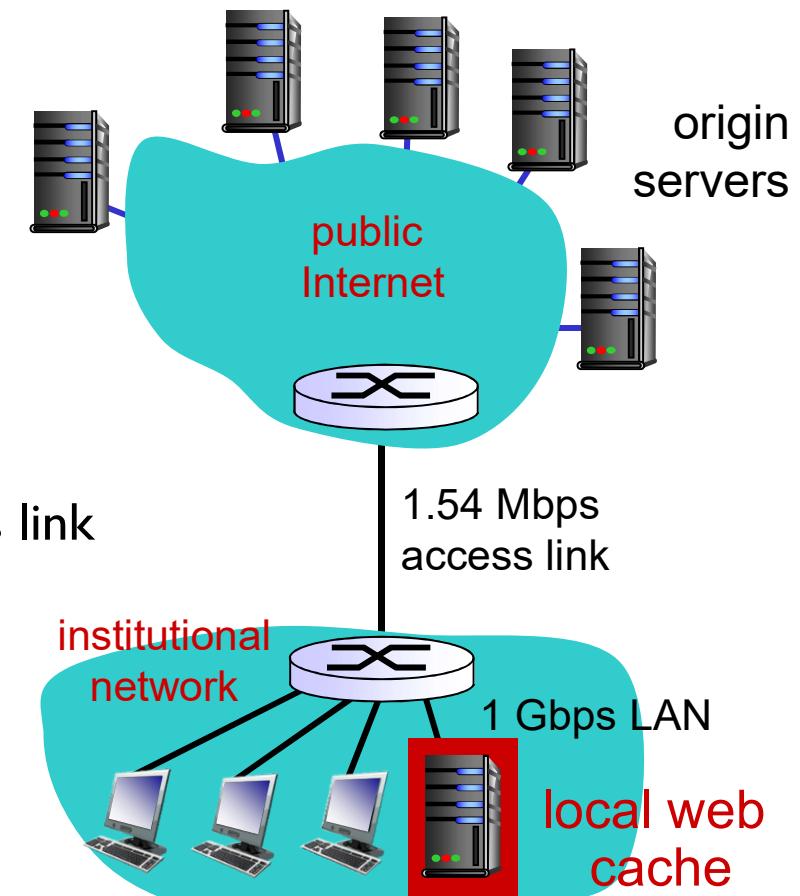
Cost: web cache (cheap!)



Caching example: install local cache

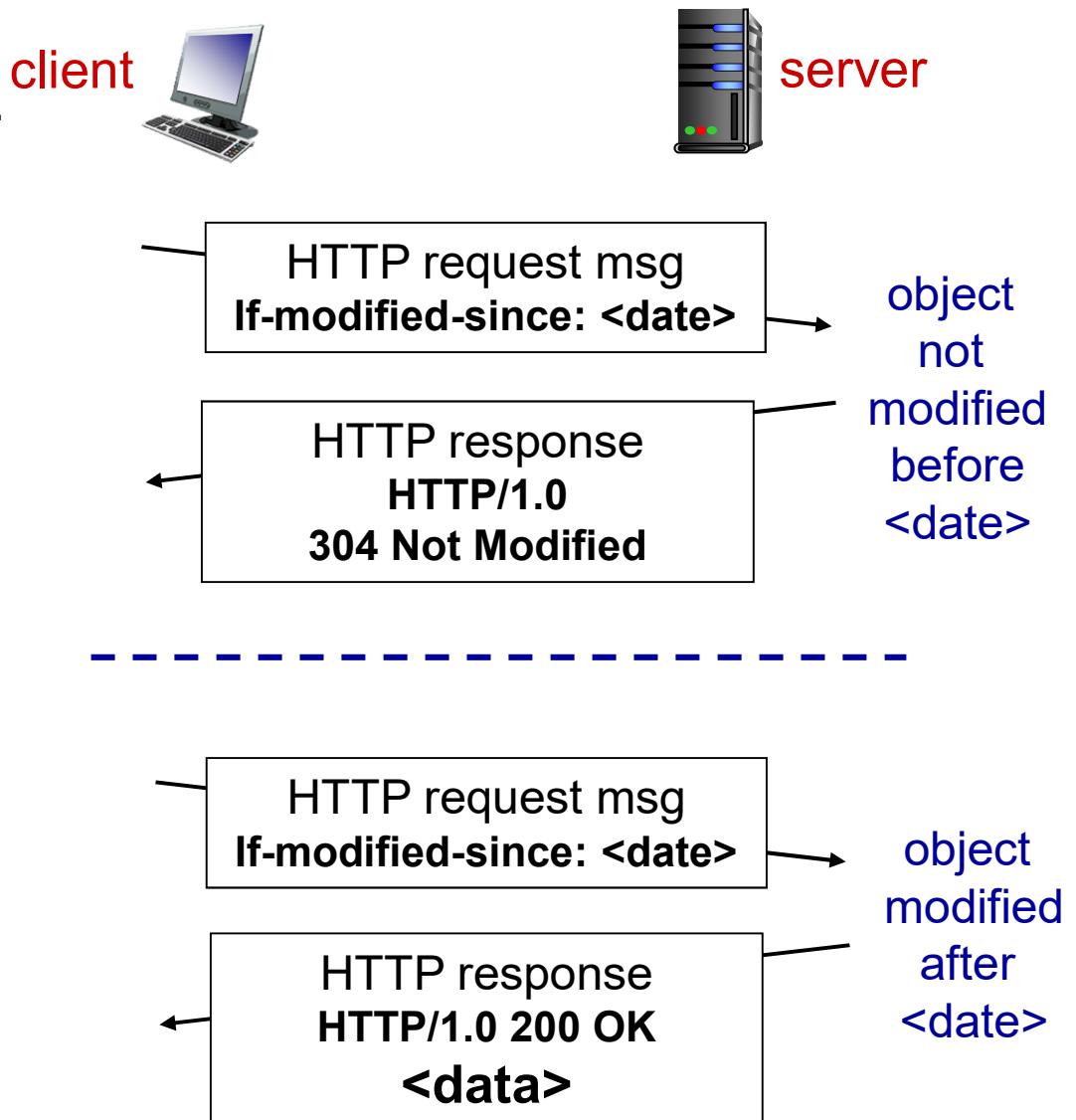
Calculating access link utilization, delay with cache:

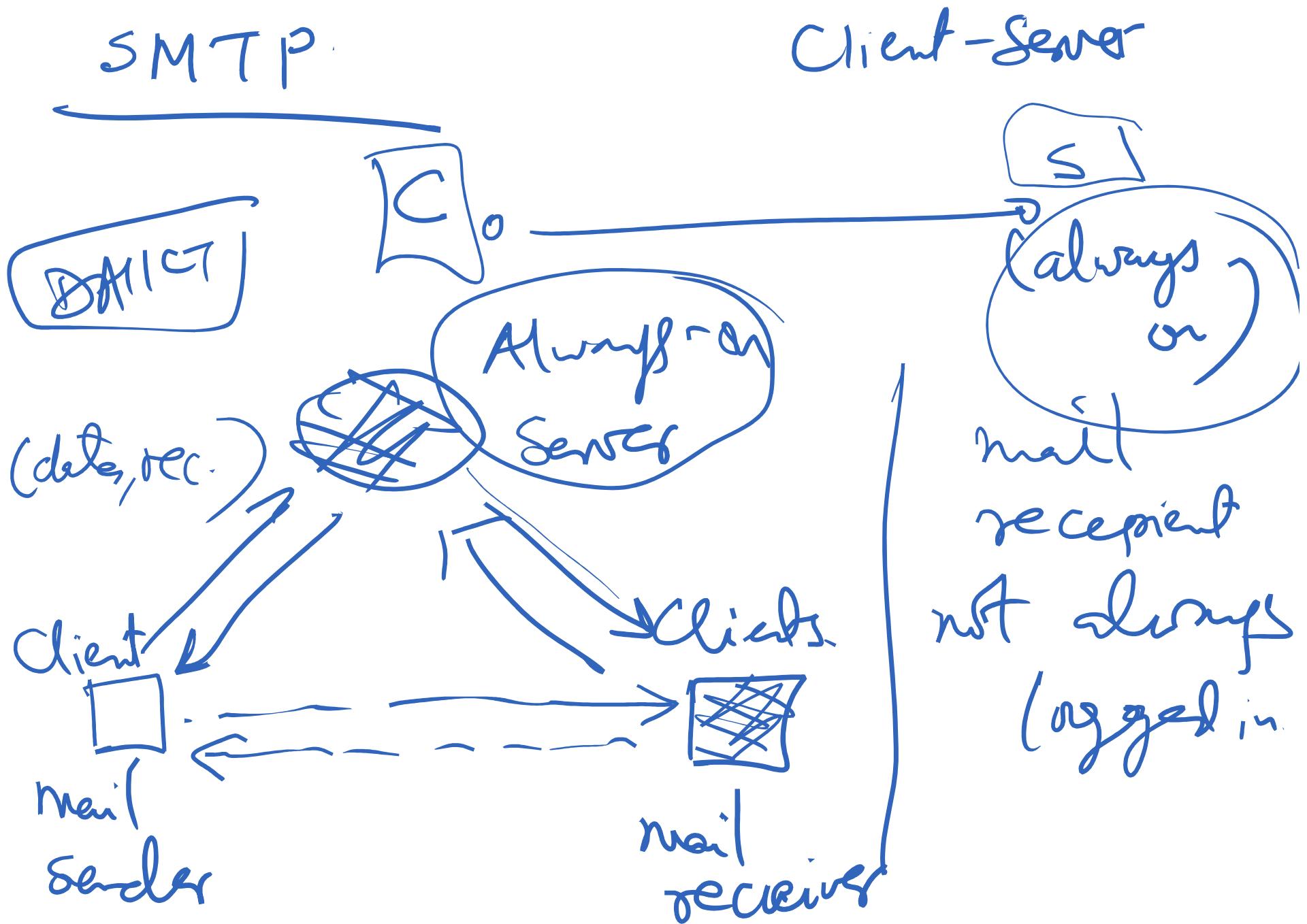
- suppose cache hit rate is 0.4
 - 40% requests satisfied at cache,
60% requests satisfied at origin
- access link utilization:
 - 60% of requests use access link
- data rate to browsers over access link
 $= 0.6 * 1.50 \text{ Mbps} = .9 \text{ Mbps}$
 - utilization = $0.9 / 1.54 = .58$
- total delay
 - $= 0.6 * (\text{delay from origin servers}) + 0.4 * (\text{delay when satisfied at cache})$
 - $= 0.6 (2.01) + 0.4 (\sim \text{msecs}) = \sim 1.2 \text{ secs}$
 - less than with 154 Mbps link (and cheaper too!)



Conditional GET

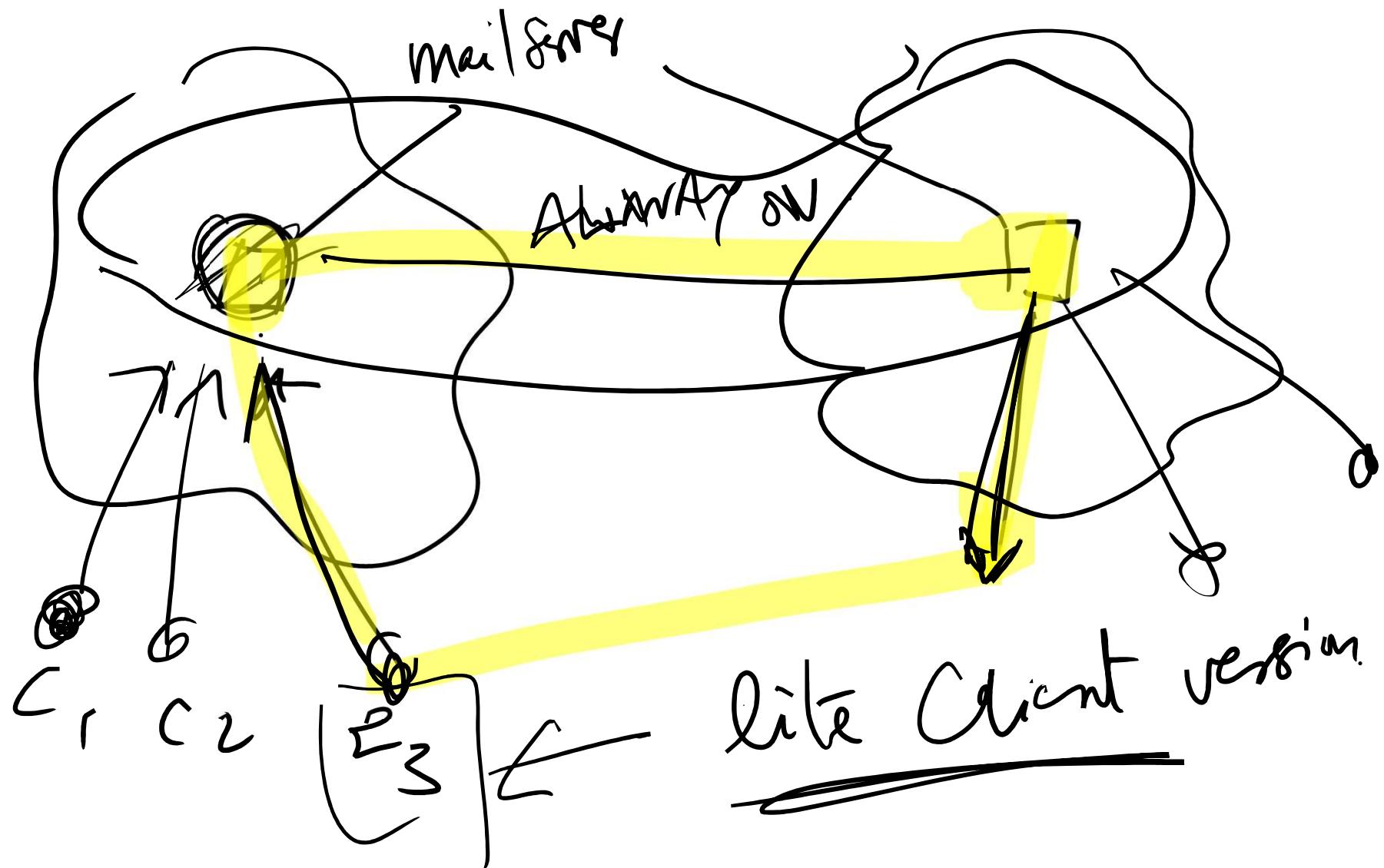
- **Goal:** don't send object if cache has up-to-date cached version
 - no object transmission delay
 - lower link utilization
- **cache:** specify date of cached copy in HTTP request
If-modified-since: <date>
- **server:** response contains no object if cached copy is up-to-date:
HTTP/1.0 304 Not Modified





DHIC.

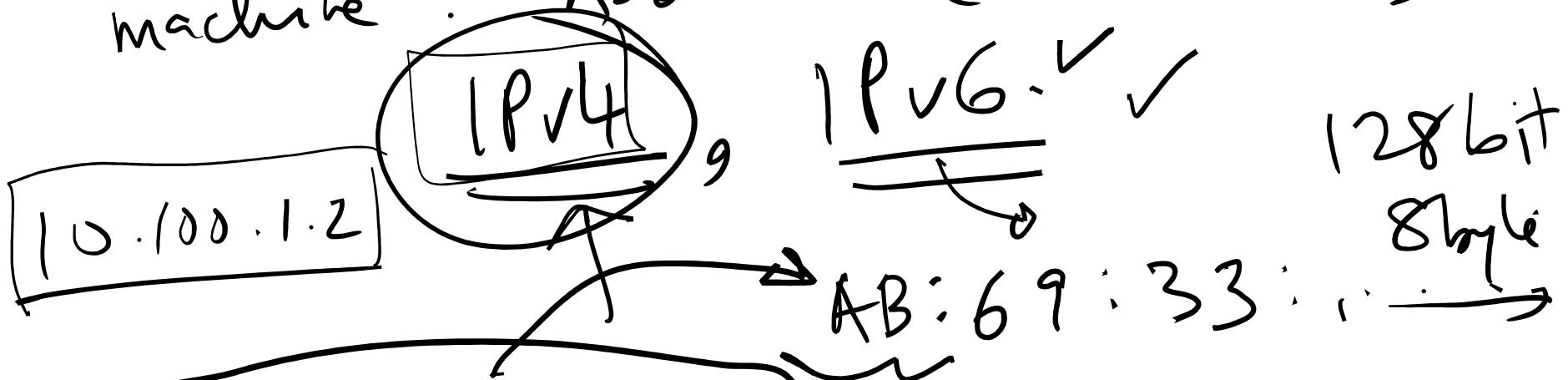
IITB



DNS

Domain Name Service

machine : Address (IP address)



www.daiict.ac.in Hex

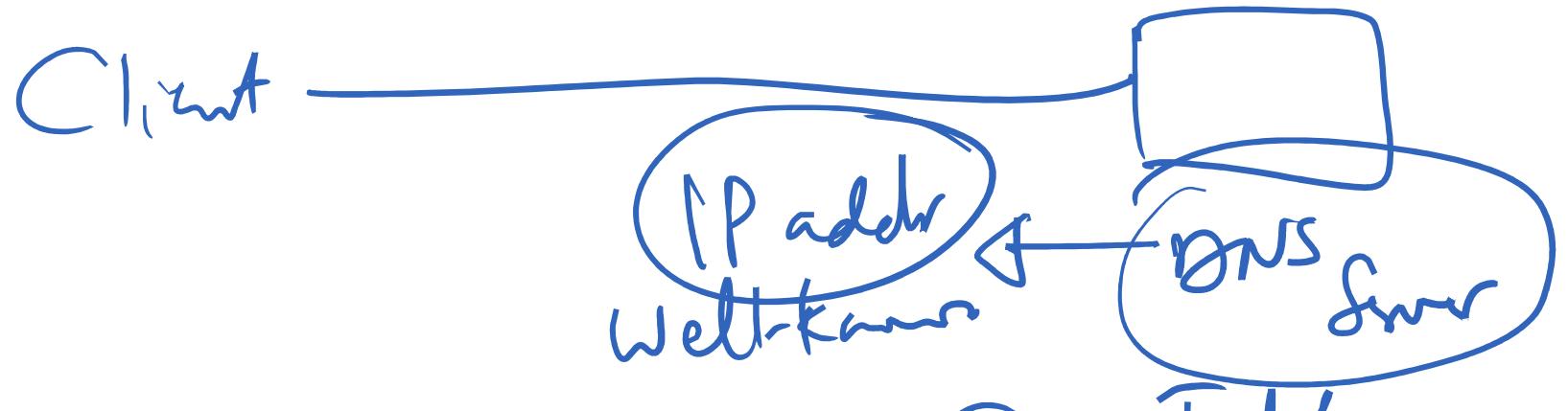
dns addr ip addr
map

use

Table

DNS Addr	IP addr
daict.ac.in	220.x.x.4,
'	'
{	'
	'

-
- tables will - millions of entries.
 - ✓ dynamic : Syncd.

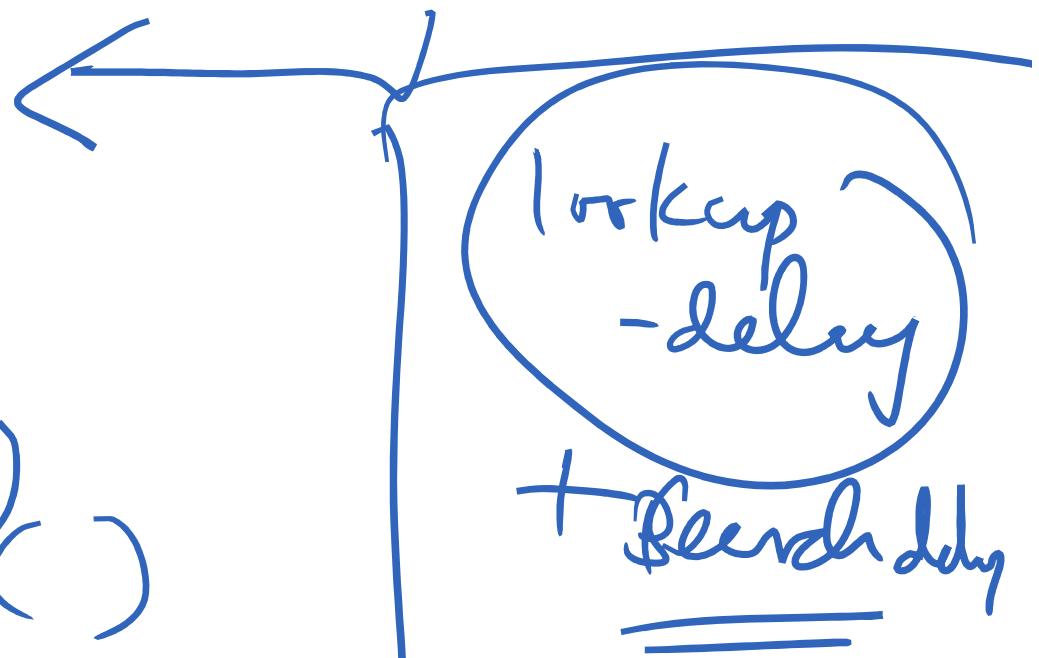


✓ `lookup(DNS, ipaddr = ??)`

~~rep(~~

✓ `add_entry()`

✓ `del_entry()`
+ ~~`update_entry()`~~



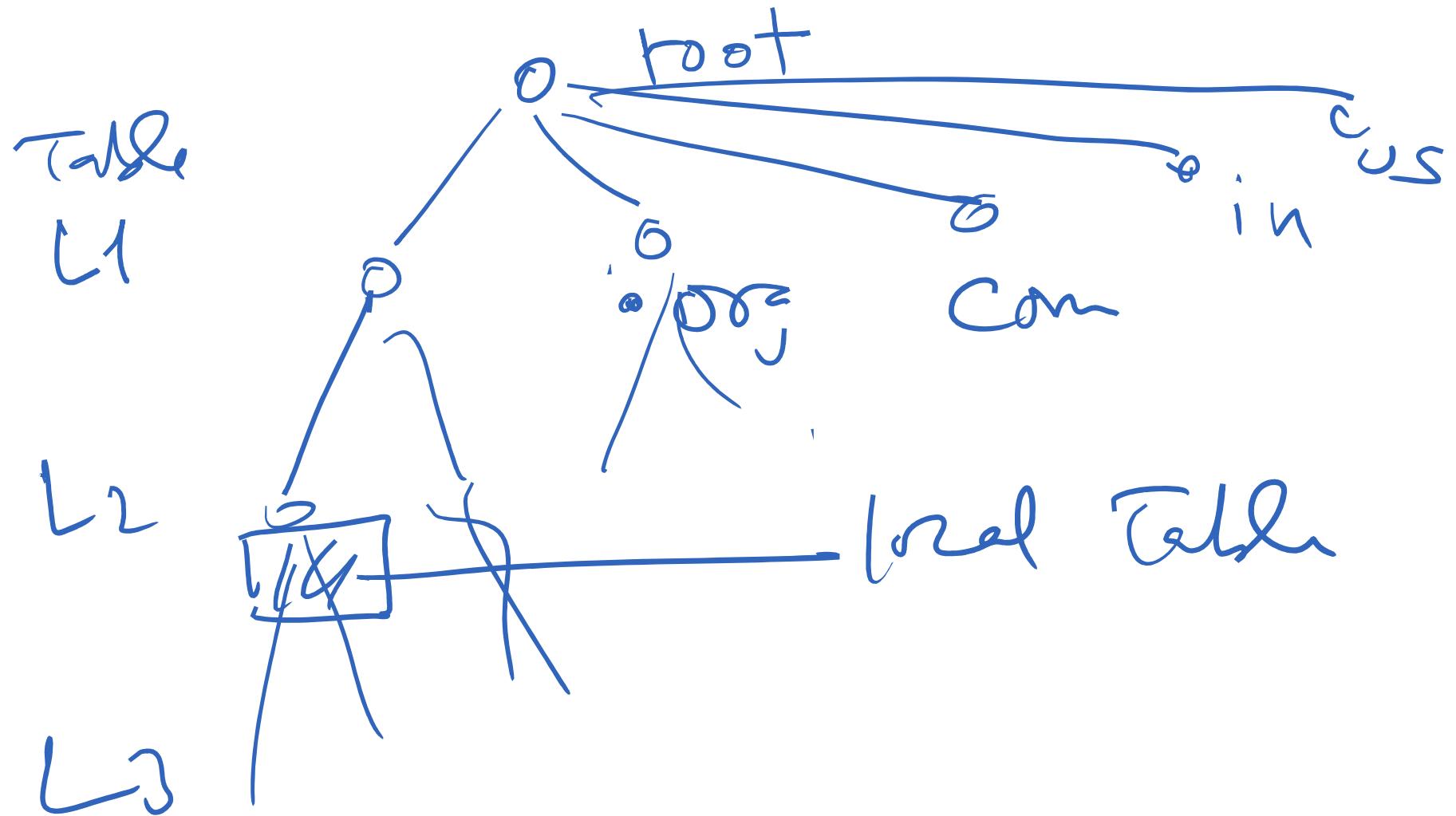
Naming conflict



"global"
distorted

www.dairict.ac.in





distributed Table.

Search:

www.mit.cs multi.edu

Search for IP addr.

expensive.

only one / session

cache:

Trade-off

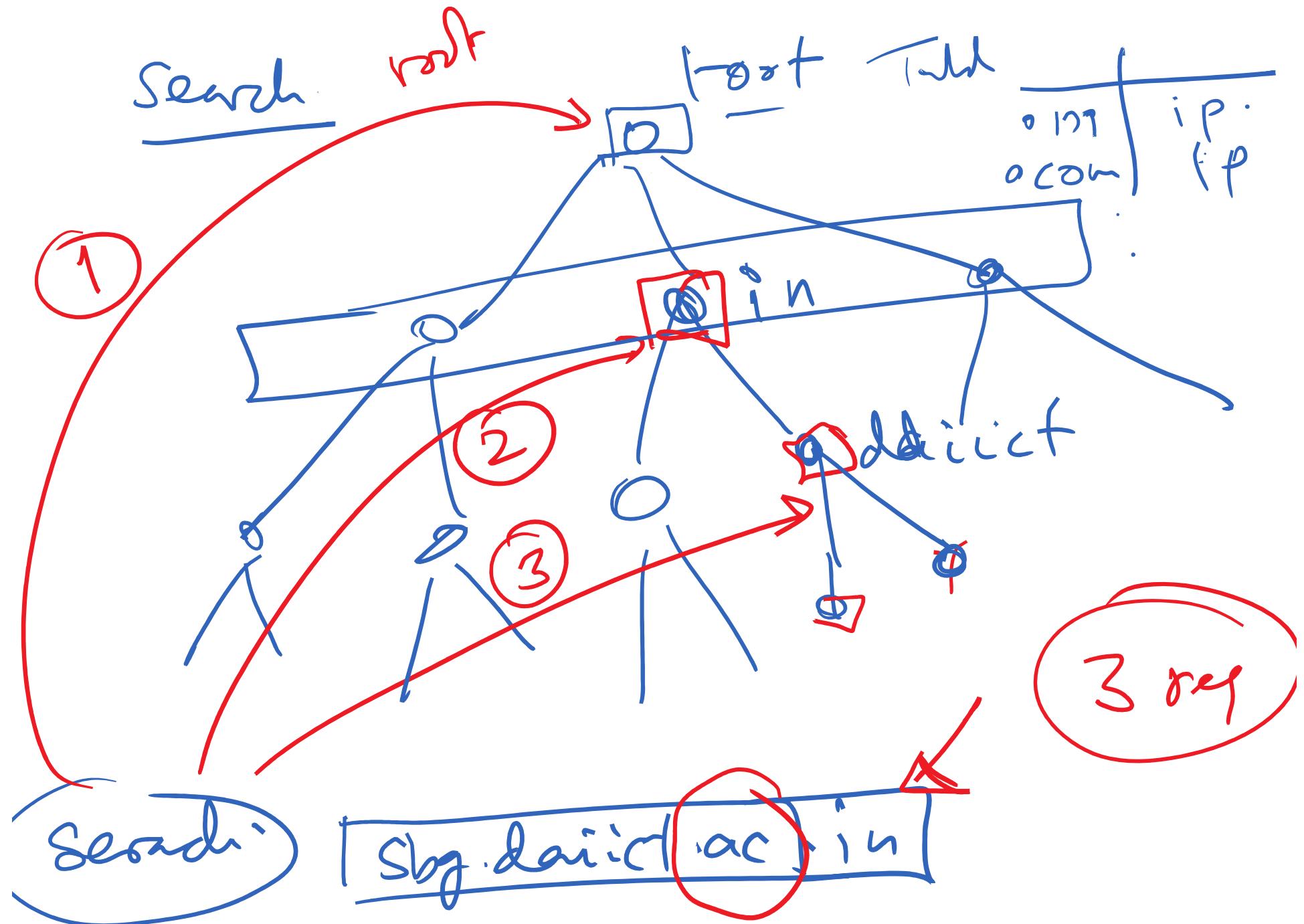
DNS

Hierarchy of

name \leftrightarrow ip Table

- Maintenance at each level.

- Search \rightarrow leading to appropriate level.



Chapter 2: outline

2.1 principles of network applications

2.2 Web and HTTP

2.3 electronic mail

- SMTP, POP3, IMAP

2.4 DNS

2.5 P2P applications

2.6 video streaming and content distribution networks

2.7 socket programming with UDP and TCP

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

Q: how to map between IP address and name, and vice versa ?

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

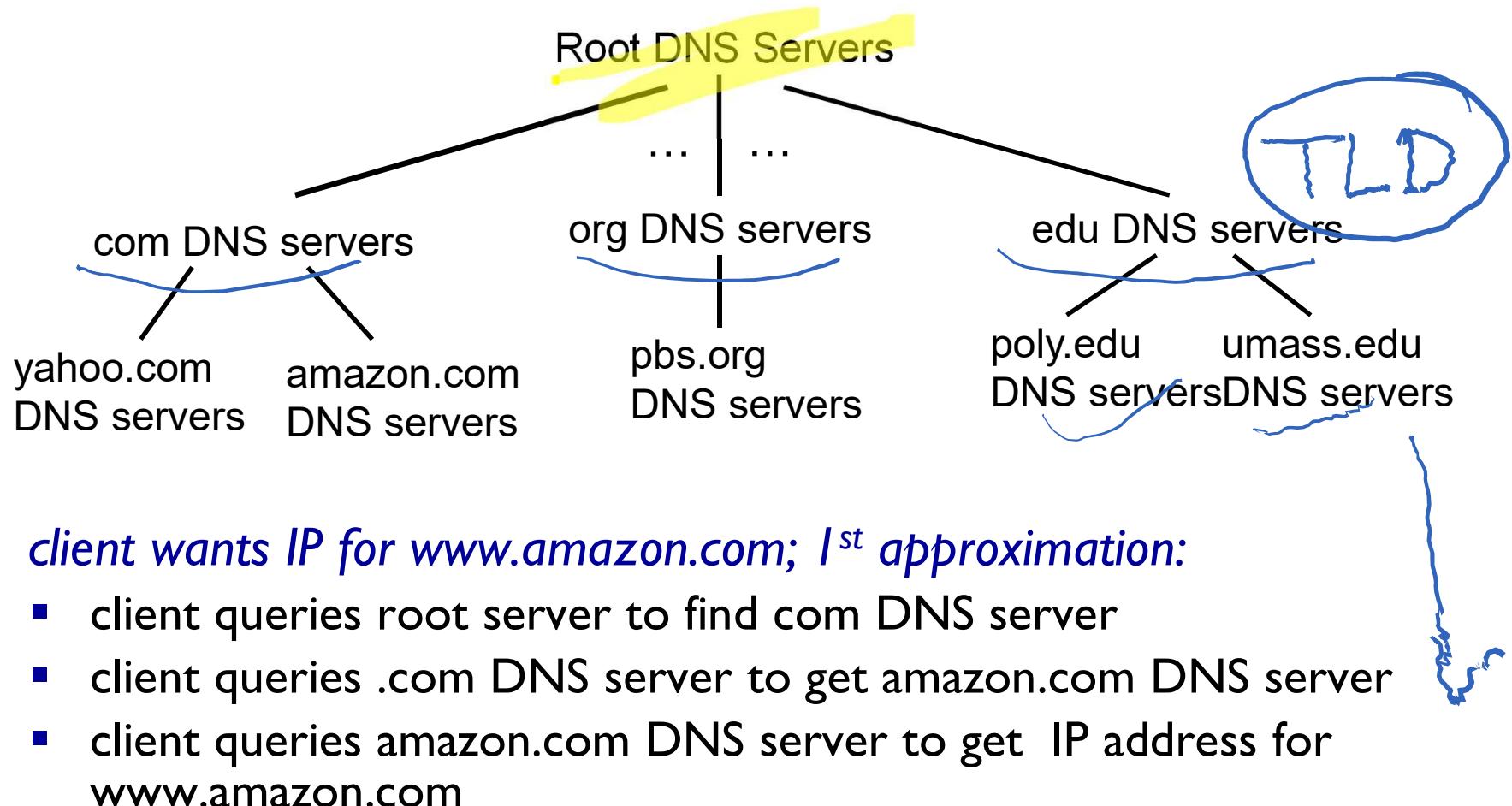
- hostname to IP address translation
- host aliasing
 - canonical, alias names
- mail server aliasing
- load distribution
 - replicated Web servers: many IP addresses correspond to one name

why not centralize DNS?

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

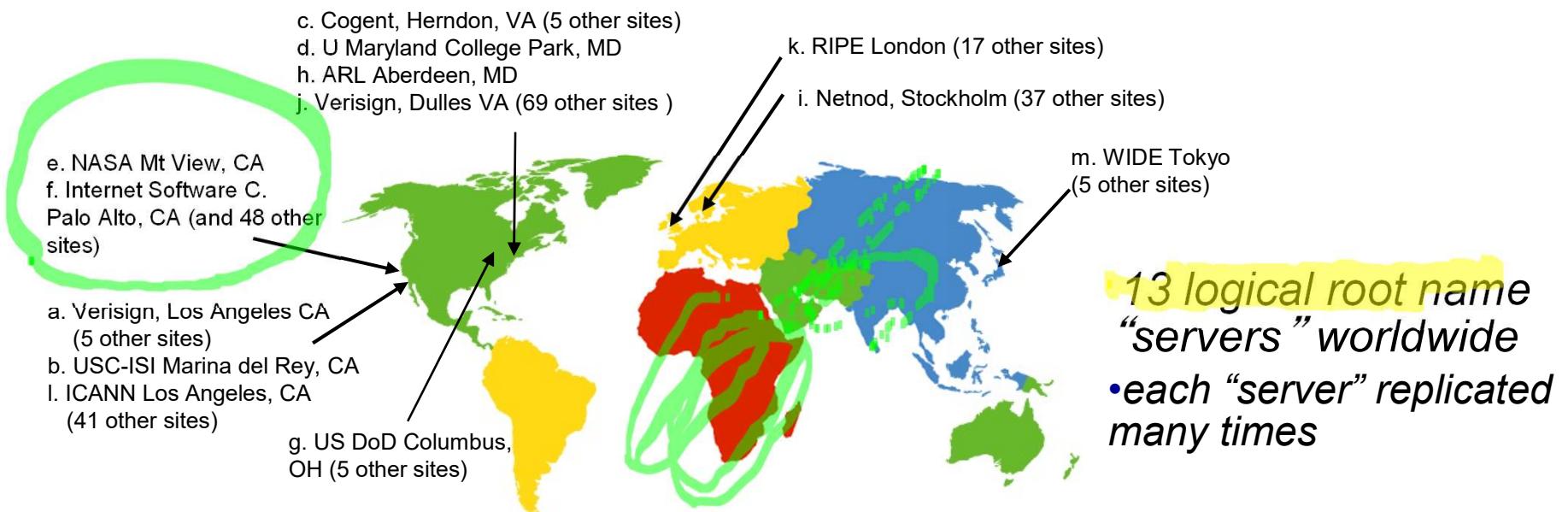
A: *doesn't scale!*

DNS: a distributed, hierarchical database



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



TLD, authoritative servers

top-level domain (TLD) servers:

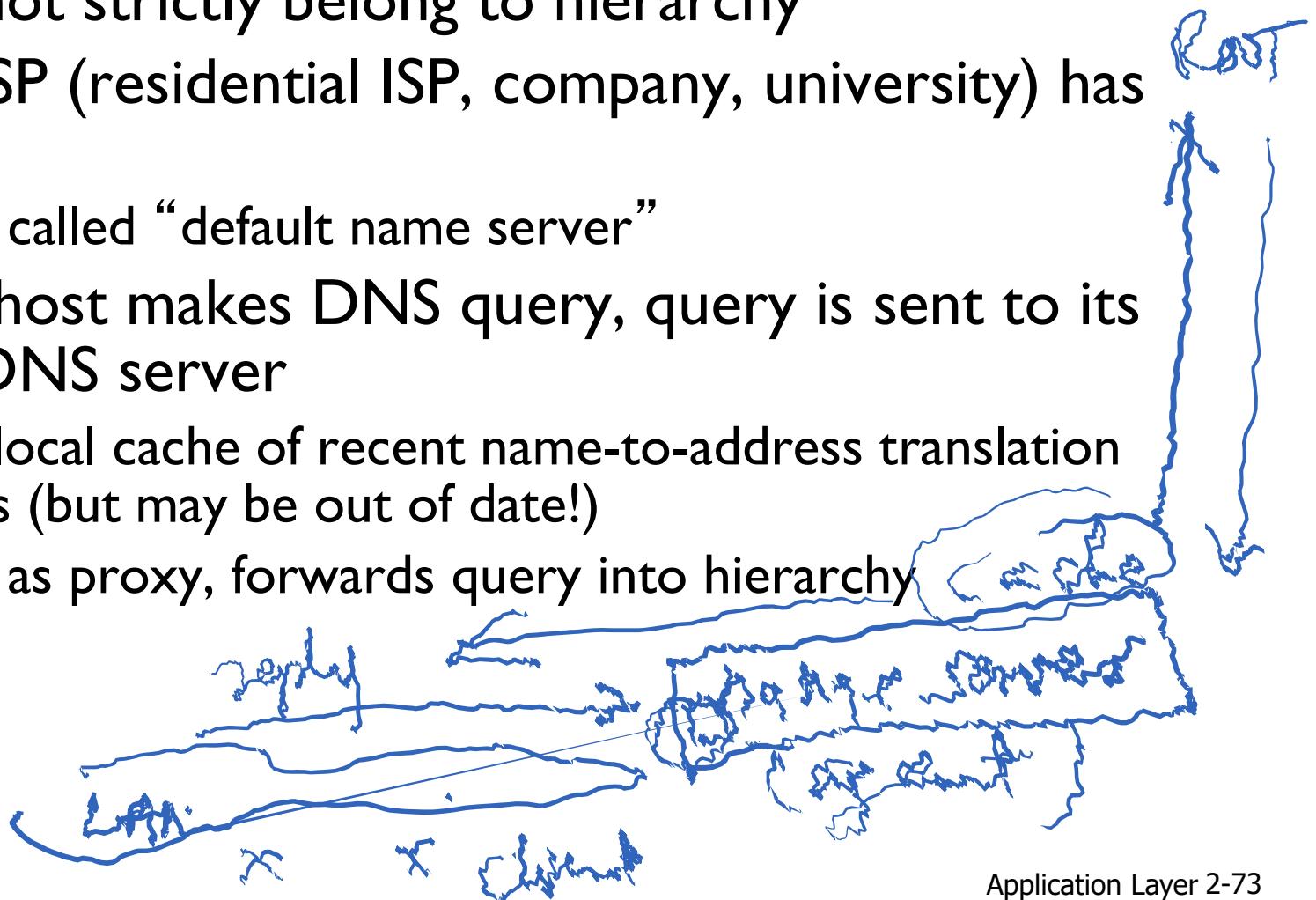
- responsible for com, org, net, edu, aero, jobs, museums, and all top-level country domains, e.g.: uk, fr, ca, jp
- Network Solutions maintains servers for .com TLD
- Educause for .edu TLD

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

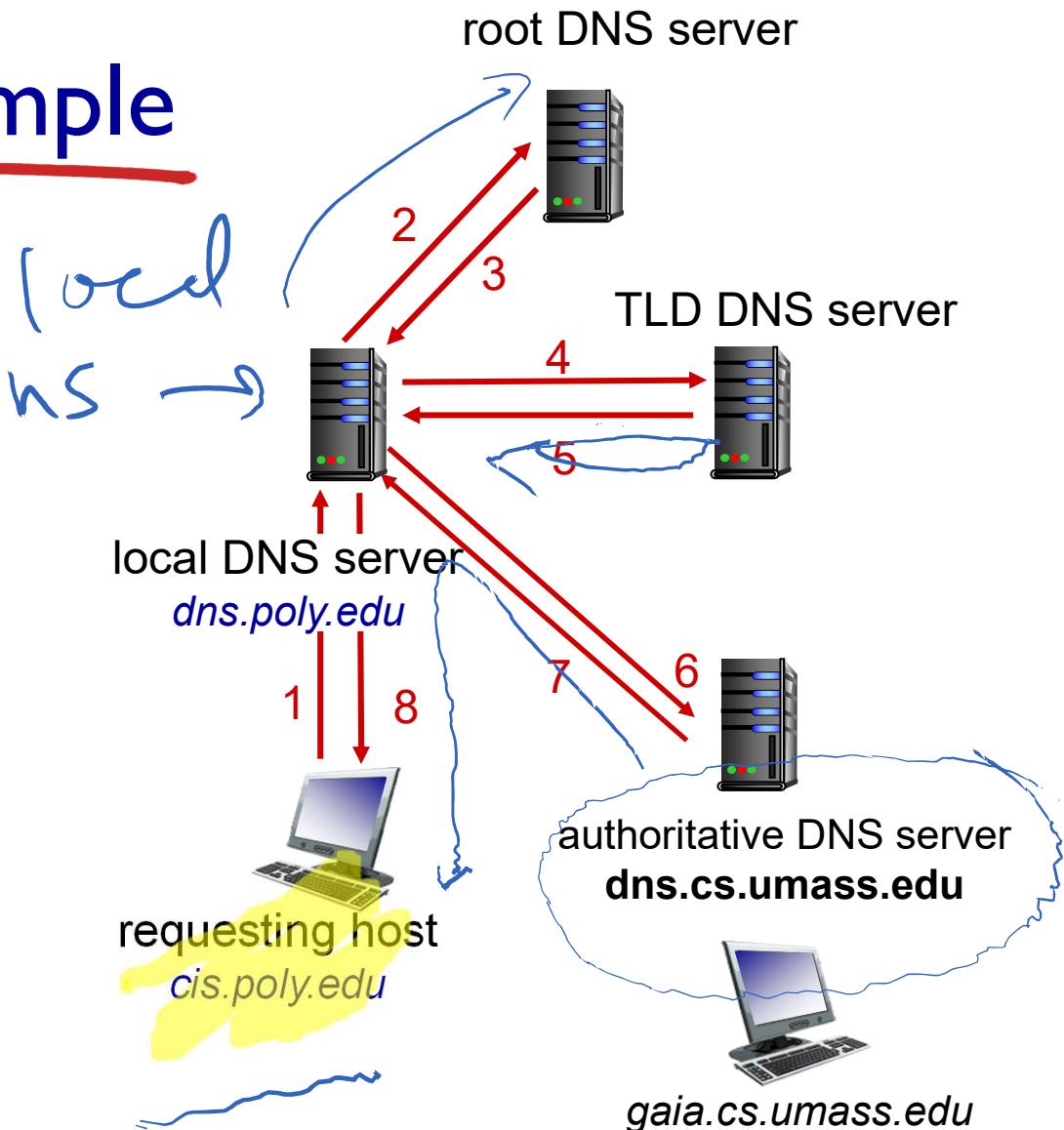


DNS name resolution example

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

iterated query:

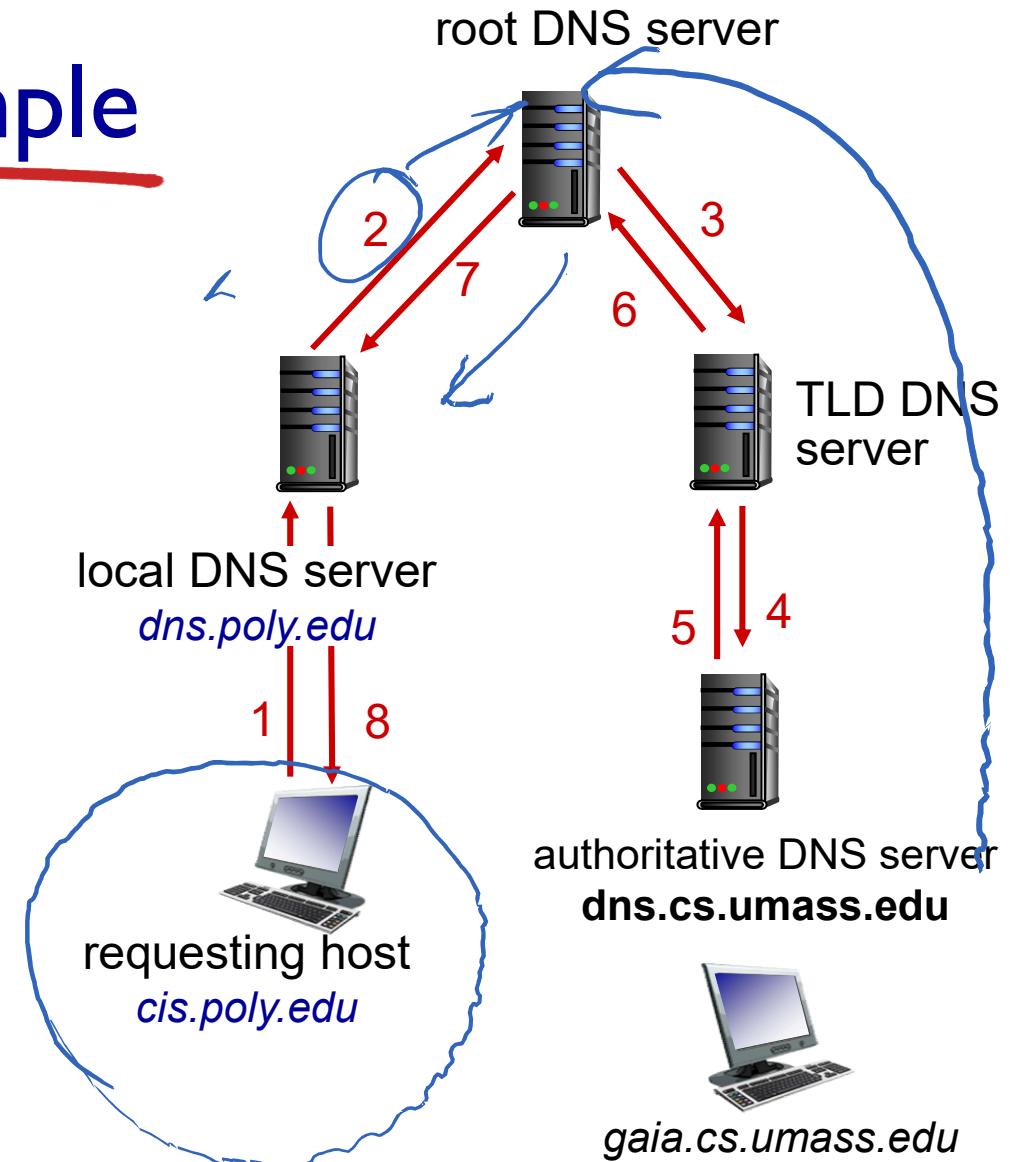
- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



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

list of names,
(one: CNAME)

DNS: distributed database 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

other names

type=MX

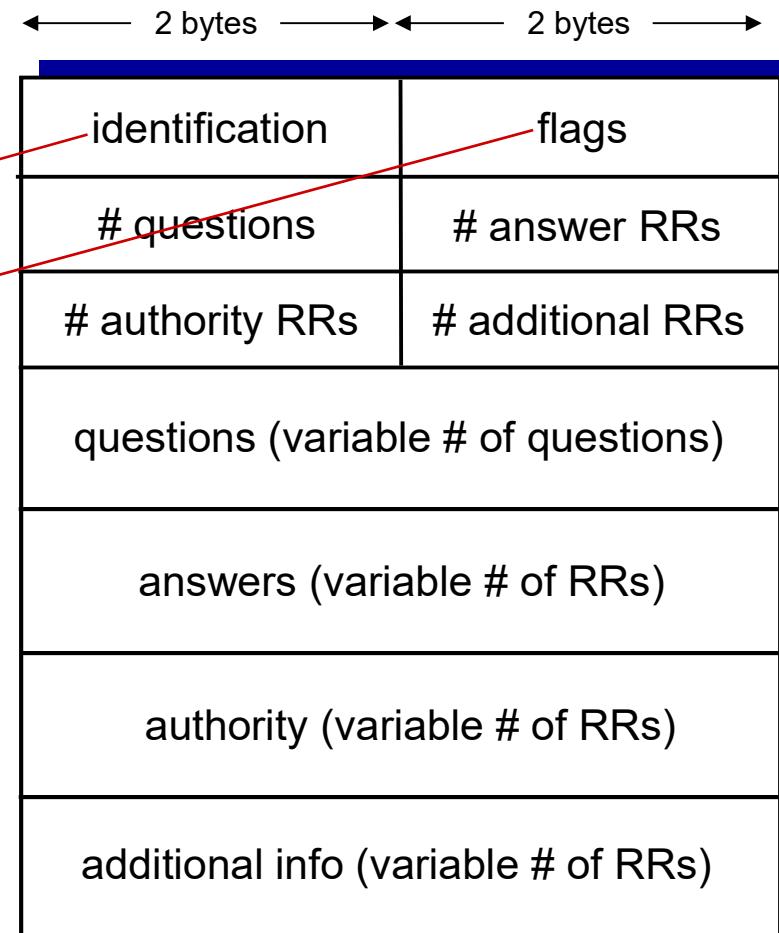
- **value** is name of mailserver associated with **name**

DNS protocol, messages

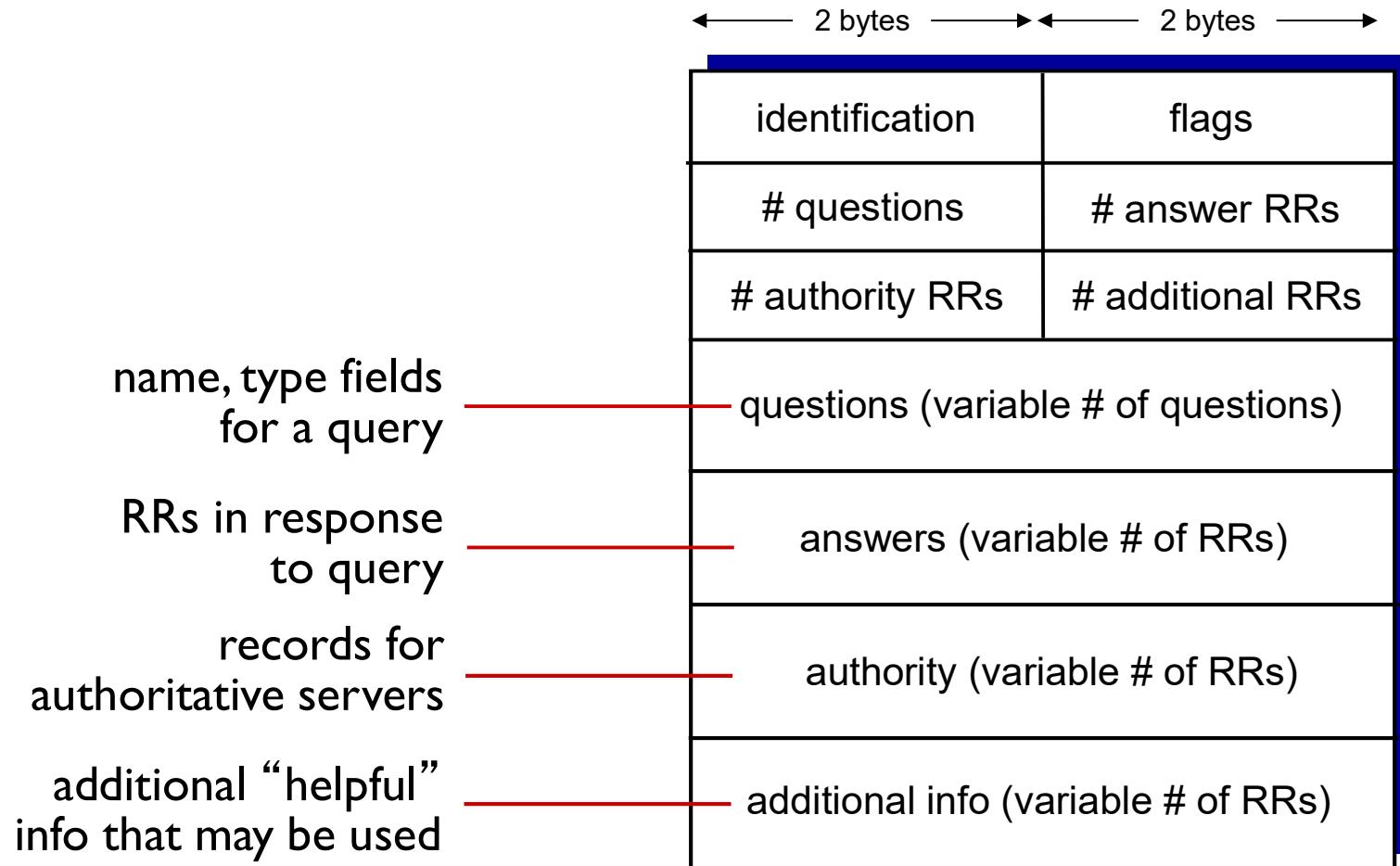
- *query* and *reply* messages, both with same *message format*

message 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 replies to DNS server, which caches

exploit DNS for DDoS

- send queries with spoofed source address: target IP
- requires amplification

DNS + Browser delay :

↓
N queries : RTT_1, \dots, RTT_n

Browser delay : RTT_0 .

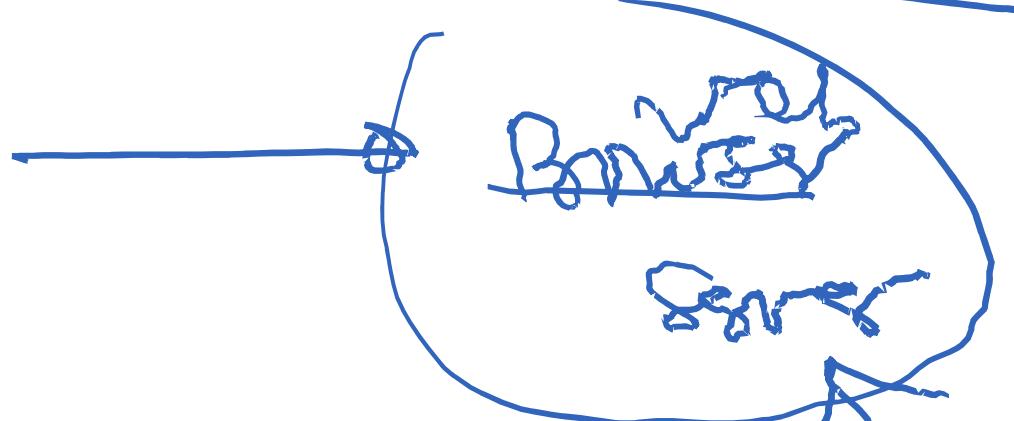
Total delay :

- ① $RTT_1 + \dots + RTT_n$
- ② $RTT_1 + \dots + RTT_n + RTT_0$
- ③ $RTT_1 + \dots + RTT_n + 3RTT_0$
- ④ $RTT_1 + \dots + RTT_n + 2RTT_0$

DNS delay =

$$RTT_0 + RTT_1 + \dots + RTT_n$$

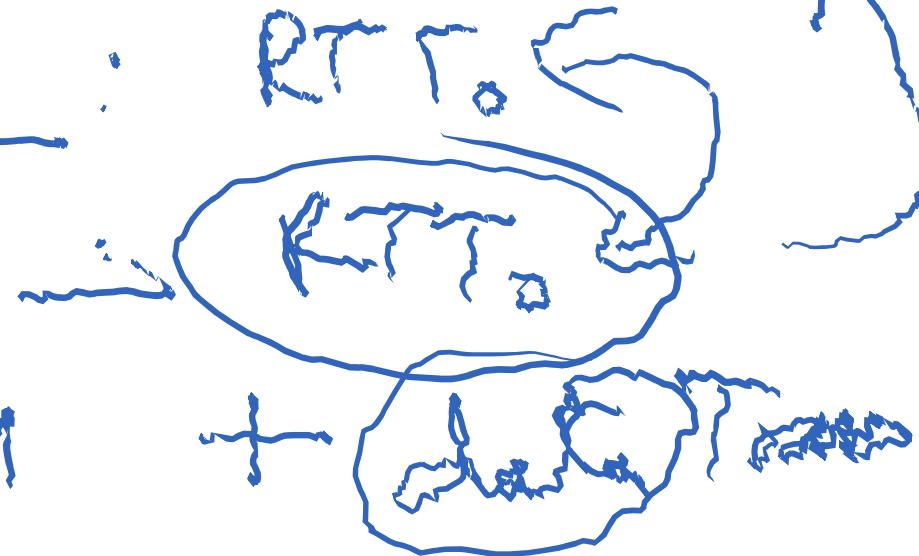
→ IP addr



TCP connection :

$$RTT_0$$

(Query + Resp)
RTT delay



UDP berende Appl. protokol

Rep: How are you?

Resp: Fine / NAT fire

src addr	dest addr	seqno	msg.
-------------	--------------	-------	------

