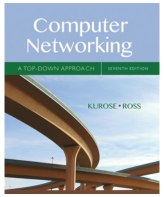
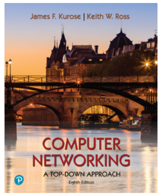
Chapter 2
Application Layer

Courtesy to the textbooks' authors and Pearson Addison-Wesley because many slides are adapted from the following textbooks and their associated slides.



Jim Kurose, Keith Ross, "Computer Networking: A Top Down Approach", 7th Edition, Pearson, 2016.



Jim Kurose, Keith Ross, "Computer Networking: A Top Down Approach", 8th Edition, Pearson, 2020.

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP
  - reading assignment

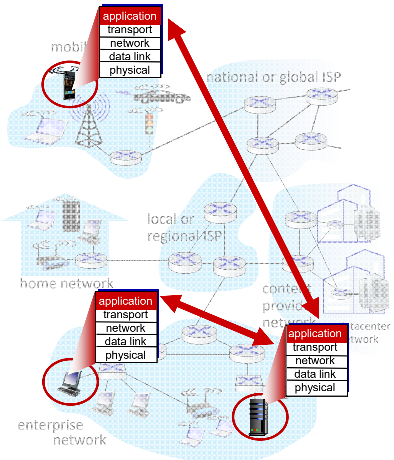
Where does a network app reside?

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



Client-server paradigm

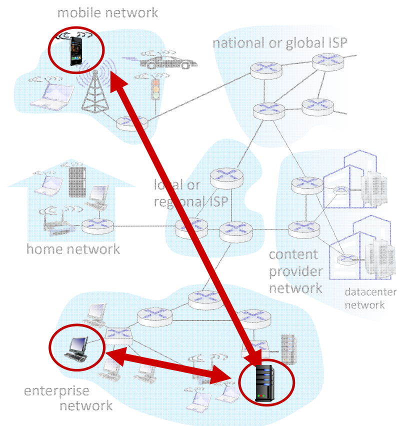
server:

- always-on host
- permanent IP address
- often in data centers, for scaling

clients:

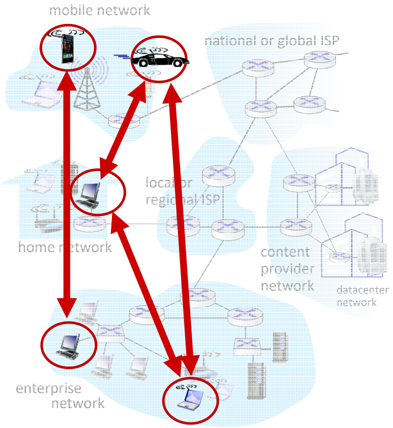
- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other

- examples: HTTP, IMAP, FTP



Peer-peer architecture

- *no* always-on server
- arbitrary end systems directly communicate
- a peer *i)* requests service from other peers and *ii)* provides 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
- example: P2P file sharing



How does two processes in hosts communicate?

process: program running within a host

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

clients, servers

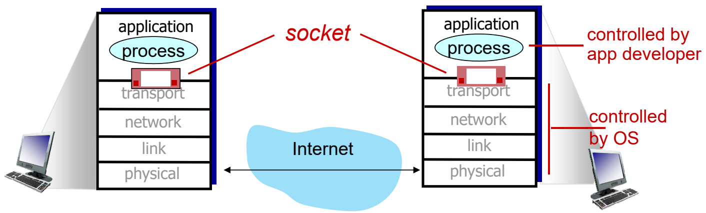
client process: process that initiates communication

server process: process that waits to be contacted

- note: an application with P2P architectures have both client process & server process

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door (or mailbox)
  - 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
  - two sockets involved: one on each side



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 number associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to www.nthu.edu.tw web server:
  - IP address: 140.114.69.135
  - port number: 80
- more shortly…

An application-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

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
- throughput
  - some apps (e.g., multimedia) require minimum amount of throughput
  - 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/download	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
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 100's msec
text messaging	no loss	elastic	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 is overloaded
- *connection-oriented:* setup required between client and server processes
- *does not provide:* timing, minimum throughput guarantee, security

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.

Why is there UDP?

- is no-frills and lightweight
- provides minimal services

Internet applications, and transport protocols

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP 1.1 [RFC 2616]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7320], DASH	TCP

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP

Web and HTTP

web page (or webpage)

- web page consists of *objects*
  - object is a file such as a HTML file, image, java applet, video clip, …
  - objects can be stored on different web servers
- most web pages consist of a *base HTML-file* and several *referenced objects*
  - *each* addressable by a *URL* (Uniform Resource Locator), e.g.,

```
www.someschool.edu/somefolder/index.html
```

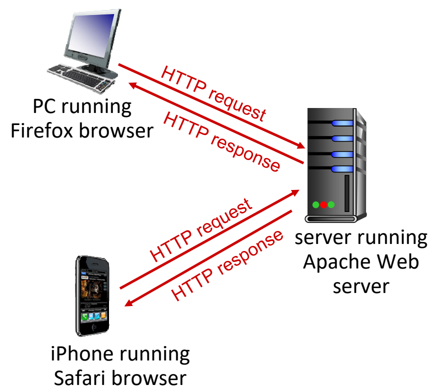
host name   path name

```
www.someschool.edu/somefolder/pic.jpg
```


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)

HTTP 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
- other mechanisms take care of "state"

HTTP connections: two types

Non-persistent HTTP

1. TCP connection opened
2. at most one object sent over TCP connection
3. TCP connection closed

downloading multiple objects required multiple connections

Persistent HTTP

- TCP connection opened to a server
- multiple objects can be sent over *single* TCP connection between client, and that server
- TCP connection closed

Non-persistent HTTP: example

User enters URL: **`www.someSchool.edu/someDepartment/home.html`**
(containing 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.html

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

time

Non-persistent HTTP: example (cont.)

User enters URL: **`www.someSchool.edu/someDepartment/homepage.html`**
(containing text, references to 10 jpeg images)

4. HTTP server closes TCP connection.

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

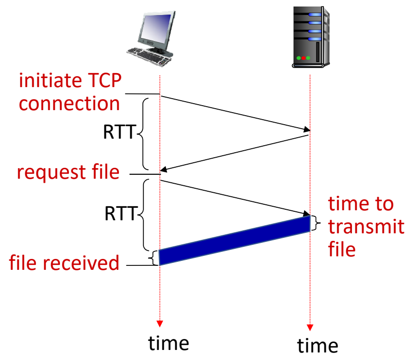
6. Steps 1-5 repeated for each of 10 jpeg objects

time

Non-persistent HTTP: response time

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

HTTP response time (per object):
- one RTT to initiate TCP connection
- one RTT for HTTP request and first few bytes of HTTP response to return
- object/file transmission time



Non-persistent HTTP response time = 2RTT+ file transmission time

Persistent HTTP (HTTP 1.1)

Non-persistent HTTP issues:
- requires 2 RTTs per object
- browsers often open multiple parallel TCP connections to fetch referenced objects in parallel
  - OS overhead for *each* TCP connection

Persistent HTTP (HTTP1.1):
- 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 (cutting response time in half)

HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP1.1: introduced multiple, pipelined GETs over single TCP connection
- server responds *in-order* (FCFS: first-come-first-served scheduling) to GET requests
- with FCFS, small object may have to wait for transmission (head-of-line [HOL] blocking) behind large object(s)
- loss recovery (retransmitting lost TCP segments) stalls object transmission

multiple parallel TCP connections: one for a single object
- drawback: high overhead (# of sockets maintained at servers), unfair

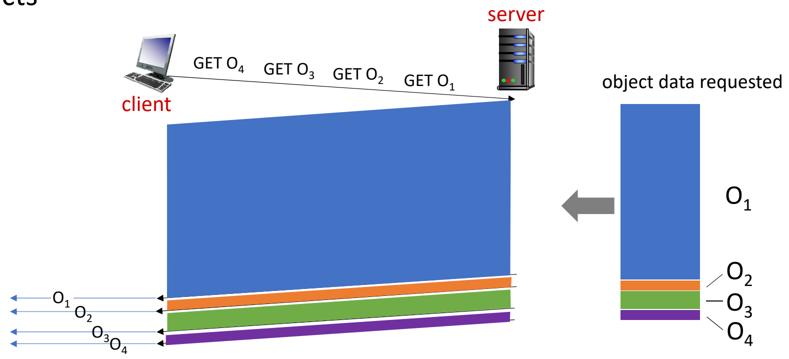
HTTP/2

Key goal: decreased delay in multi-object HTTP requests

HTTP/2: [RFC 7540, 2015] increased flexibility at *server* in sending objects to client:
- methods, status codes, most header fields unchanged from HTTP 1.1
- transmission order of requested objects based on client-specified object priority (not necessarily FCFS)
- *push* unrequested objects to client
- divide objects into frames, schedule frames to mitigate HOL blocking

HTTP/2: mitigating HOL blocking

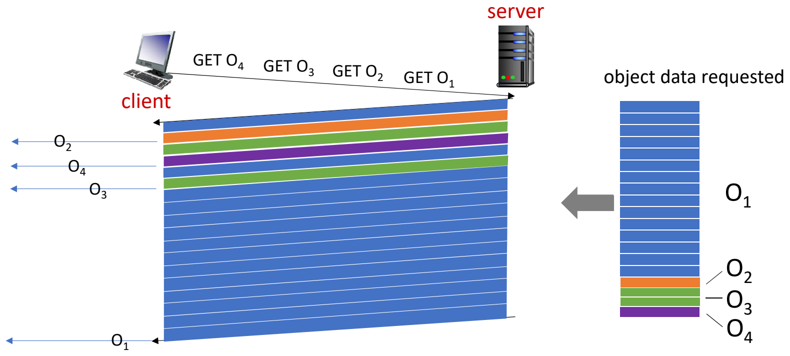
HTTP 1.1: client requests 1 large object (e.g., video file) and 3 smaller objects



objects delivered in order requested: O_2, O_3, O_4 wait behind O_1

HTTP/2: mitigating HOL blocking

HTTP/2: objects divided into frames, frame transmission interleaved



O_2, O_3, O_4 delivered quickly, O_1 slightly delayed

HTTP request message

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

request line (GET, POST, HEAD commands)

carriage return (CR)
line-feed (LF) character

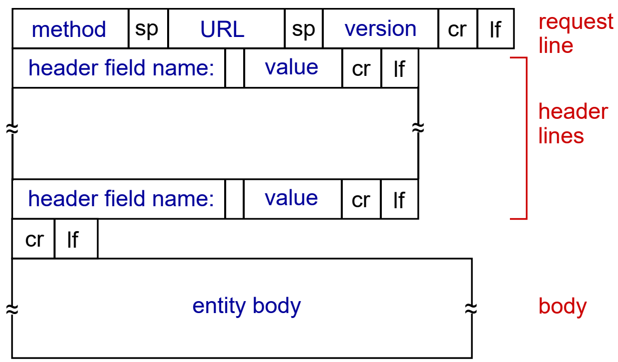
header lines

```
GET /somefolder/index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X
   10.15; rv:80.0) Gecko/20100101 Firefox/80.0 \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
Connection: keep-alive\r\n
\r\n
```

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

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

HTTP request message: general format



Other HTTP request messages

POST method:

- web page might include form input or allow to upload files
- user input (sent from client to server) is put in entity body of HTTP POST request message

GET method (for sending data to server):

- include information in URL field of HTTP GET request message (following a '?'):

`www.google.com/search?q=http+get&lr=lang_en`

- entity body ignored or rejected

HEAD method:

- requests headers (only) that would be returned *if* specified URL were requested with an HTTP GET method.
- often for debugging

PUT method:

- create or update a file to server
  - with content in entity body
  - at specific URL

HTTP response message

protocol | status code | status phrase

status line → `HTTP/1.1 200 OK`

header lines:

```
Date: Tue, 08 Sep 2020 00:53:20 GMT
Server: Apache/2.4.6 (CentOS)
   OpenSSL/1.0.2k-fips PHP/7.4.9
   mod_perl/2.0.11 Perl/v5.16.3
Last-Modified: Tue, 01 Mar 2016 18:57:50 GMT
Accept-Ranges: bytes
Content-Length: 2651
Content-Type: text/html; charset=UTF-8
\r\n
```

data, e.g., requested HTML file → `data 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 message

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

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

1. telnet to your favorite Web server:

```
% telnet www.cs.nthu.edu.tw 80
```

- opens TCP connection to port 80 (default HTTP server port) at www.cs.nthu.edu.tw.
- anything typed in will be sent to port 80 at www.cs.nthu.edu.tw

2. type in a GET HTTP request:

```
GET /~jungchuk/test.html HTTP/1.1
Host: www.cs.nthu.edu.tw
```

- 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)

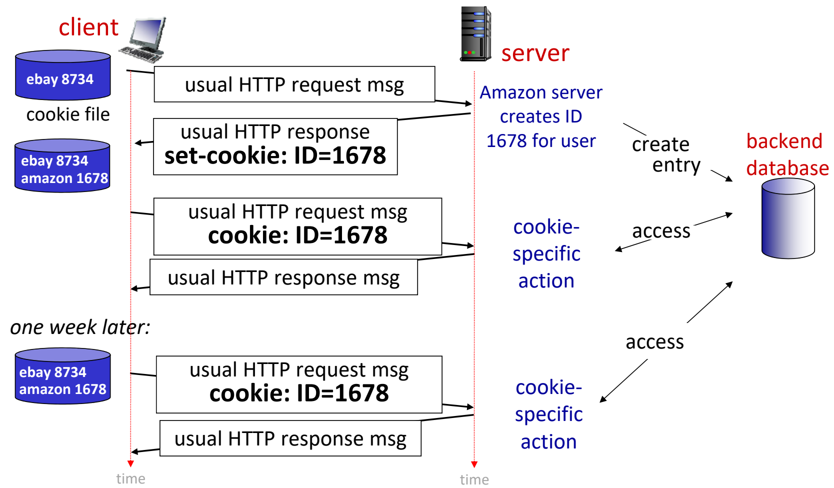
Maintaining user/server state: cookie

- *state* is very helpful for:
  - identification/authorization
  - shopping carts
  - recommendations

- Challenge: HTTP itself is *stateless*

- Where to keep state?
  - at protocol endpoints
    - client side: browser
    - server side: backend database
  - in messages
    - HTTP messages carry state
- How exactly?
  - *cookie*
  - local storage
  - IndexedDB data
  - Session storage
  - …

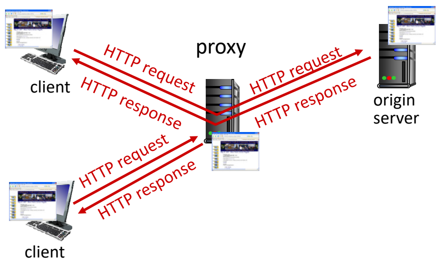
Maintaining user/server state: cookie



Web cache (or called proxy)

- *Goal:* satisfy client requests on behalf of origin server
  - user configures browser to point to a (local) *Web cache*
- *Why* web caching?
  - reduce response time for client request
    - cache is closer to client
  - reduce traffic on ISP's access link
  - enables "poor" content providers to more effectively deliver content
  - anonymity

- browser/client sends all HTTP requests to proxy
  - *if* object not in proxy: proxy requests object from origin server, caches received object, then returns object to client
  - *else* proxy returns object to client



Check freshness by conditional GET

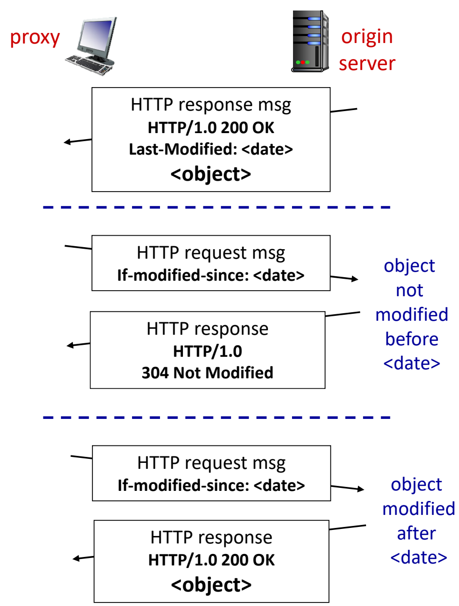
Goal: don't send object if proxy has cached up-to-date version
- "If-modified-since" header
- "Expires" or "max-age" header

- *proxy:* specify date of cached copy in HTTP request

  If-modified-since: <date>

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

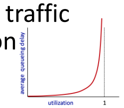
  HTTP/1.0 304 Not Modified



Caching example

Scenario:

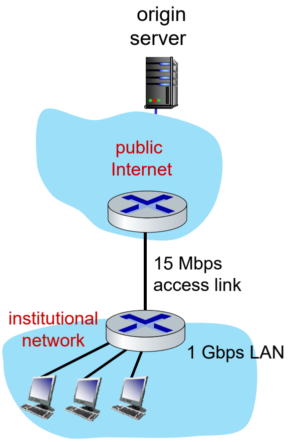
- access link rate: 15 Mbps
- rate of demand over access link/LAN = ?
  - web object size: 0.1 Mb
  - average request rate: 150 requests/sec
- one-way delay in public Internet: 2 seconds
- link/LAN delay is 10 ms at low or medium traffic
  is minutes at high utilization



Performance:

- access link utilization = 15/15 = 1
- LAN utilization: 15M/1G = 0.015
- end-to-end delay
  = Internet delay + access link delay + LAN delay
  ≈ 2 sec + minutes + 10 ms

access link is the bottleneck



Option 1: buy a faster access link

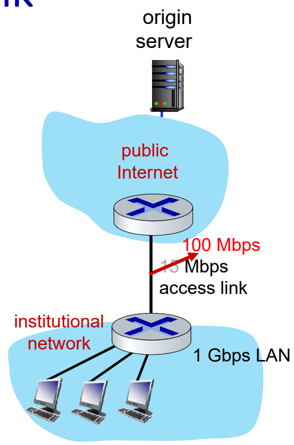
Scenario:

- access link rate: ~~15~~ 100 Mbps
- rate of demand over access link/LAN = 15 Mbps
  - web object size: 0.1 Mb
  - average request rate: 150 requests/sec
- one-way delay in public Internet: 2 seconds
- link/LAN delay is 10 ms at low or medium traffic
  is minutes at high utilization

Performance:

- access link utilization = ~~1~~ 15/100 = 0.15
- LAN utilization: 0.015
- end-to-end delay
  = Internet delay + access link delay + LAN delay
  ≈ 2 sec + ~~minutes~~ 10 ms + 10 ms

Cost: faster access link (expensive!)



Option 2: install a proxy

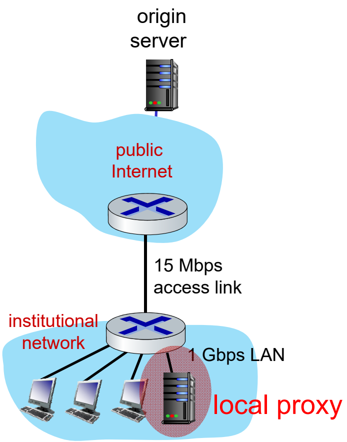
Scenario:

- access link rate: 15 Mbps
- rate of demand over access link/LAN = 15 Mbps
  - web object size: 0.1 Mb
  - average request rate: 150 requests/sec
- one-way delay in public Internet: 2 seconds
- link/LAN delay is 10 ms at low or medium traffic
  is minutes at high utilization

Cost: proxy (cheap!)

Performance:

- LAN utilization: .?
- access link utilization = ?
- average end-end delay = ?



Calculating end-to-end delay with proxy

suppose cache hit rate is 0.4:

- each request satisfied by proxy takes
  ≈ 10 ms = 0.01 s
- each request served by origin server takes
  - access link utilization = 0.6 * 15 / 15 = 0.6
  - access link delay ≈ 10 ms
  - end-to-end delay
    ≈ 2 s + 10 ms + 10 ms ≈ 2.02 s
- average end-end delay:
  = 0.6 * (delay from origin server)
  + 0.4 * (delay when satisfied by proxy)
  ≈ 0.6 * 2.02 s + 0.4 * 0.01 s
  ≈ 1.2 secs

  lower avg end-end delay than with 100 Mbps link (and cheaper too!)