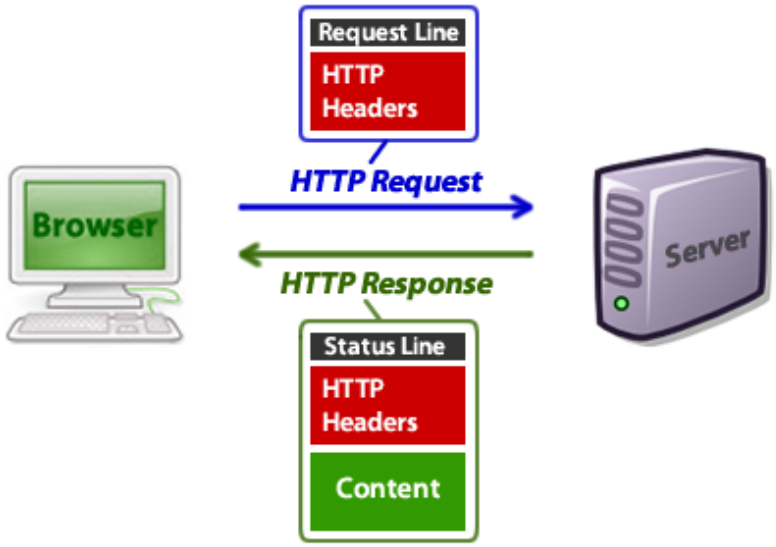
HTTP



Tecnologie Internet

a.a. 2022/2023

Network protocols

A protocol is the formal definition of external behavior for communicating entities. It defines *format*, *order* of *messages sent and received*, and *actions taken* on message transmission/reception.

- e.g., TCP, IP, HTTP, Skype, IEEE 802.11 (WiFi)

A protocol is *connection oriented* if the peer entity must be synchronized before exchanging useful data (connection set up); otherwise it is *connectionless*.

Internet standards

- IETF: Internet Engineering Task Force
  - RFC: Request for comments https://www.ietf.org/rfc/rfc2026.txt

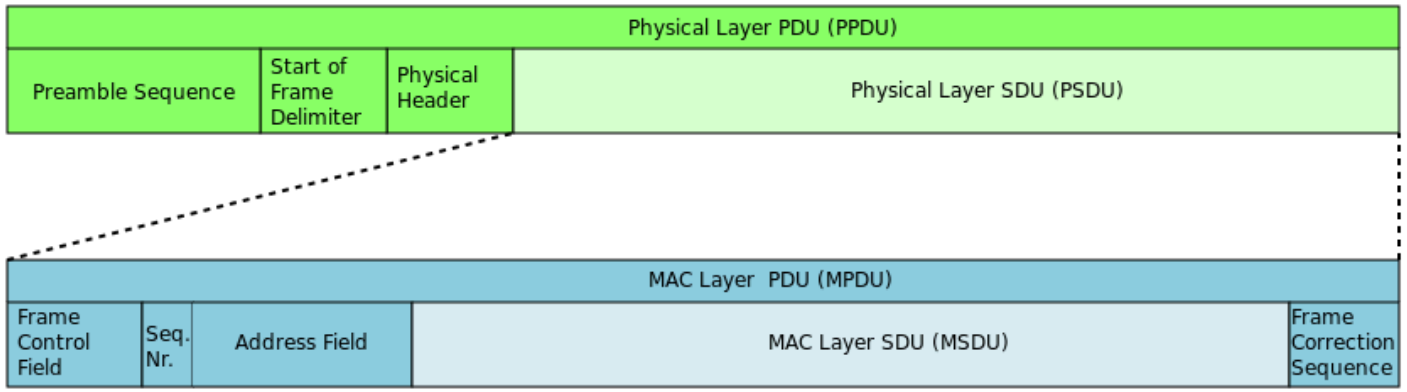
Network protocols

Human vs network protocol:

Protocol layers

Networking functions are structured as a layered model:

- layer *n* communicates with other layer *n;* the data exchanged is called *Protocol Data Unit (PDU)*
- layer *n* uses the *service* of layer *n-1* and offers a service to layer *n+1*; the interface data is called *Service Data Unit (SDU)*
- entities at the same layer are denoted as *peer entities*
- operation rules between peer entities are called *procedures*



Why layering?

Explicit structure allows identification, relationship of complex system's pieces

- layered *reference model* for discussion

Modularization eases maintenance, updating of system

- a change of implementation of a layer's service is transparent to rest of system

Internet protocol stack

Application: supporting network applications
- FTP, SMTP, HTTP

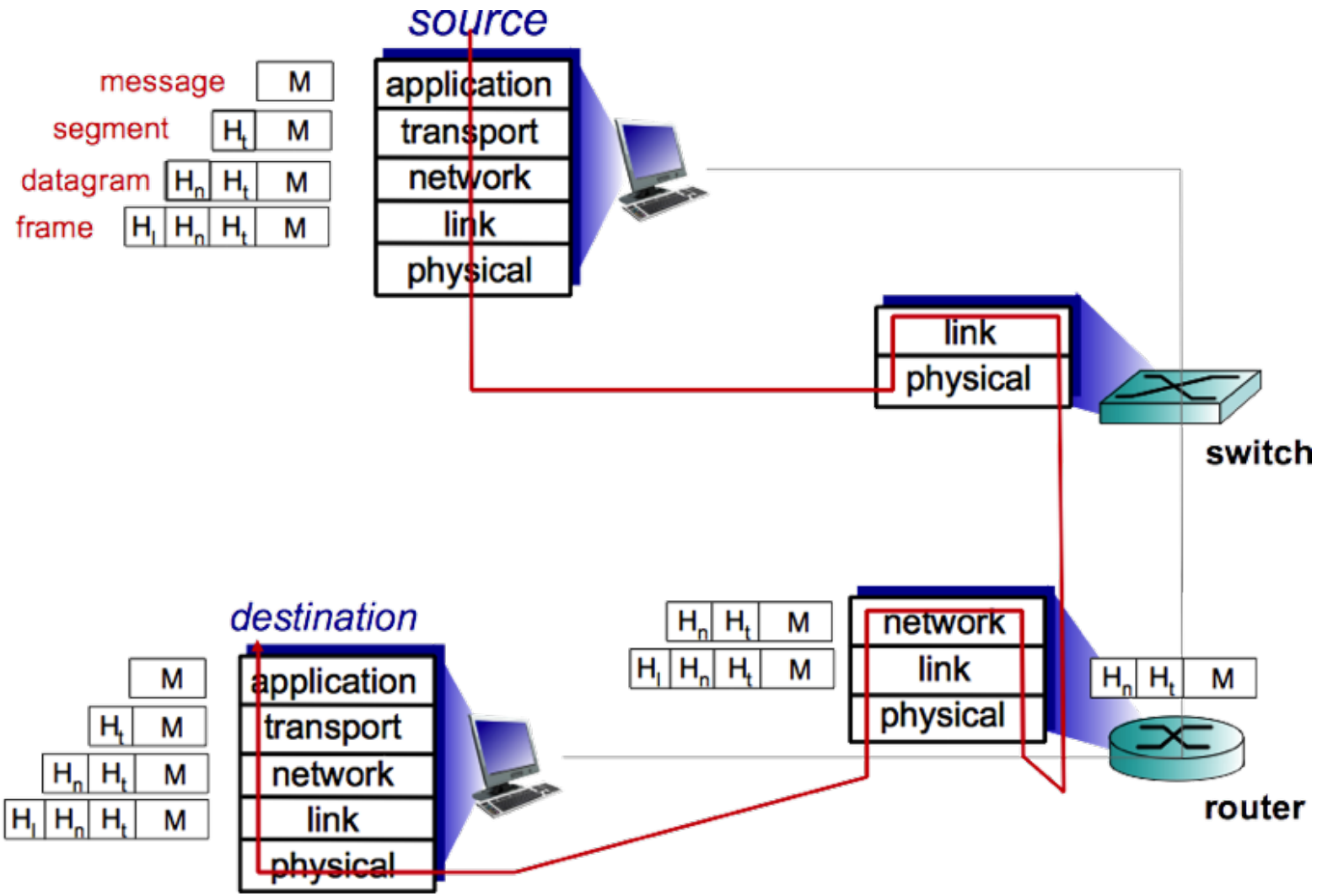
Transport: process-process data transfer
- TCP, UDP

Network: routing of datagrams from source to destination
- IP, routing protocols

Link: data transfer between neighboring network elements
- Ethernet, 802.11 (WiFi), PPP

Physical: bits "on the wire"

Encapsulation



Network services

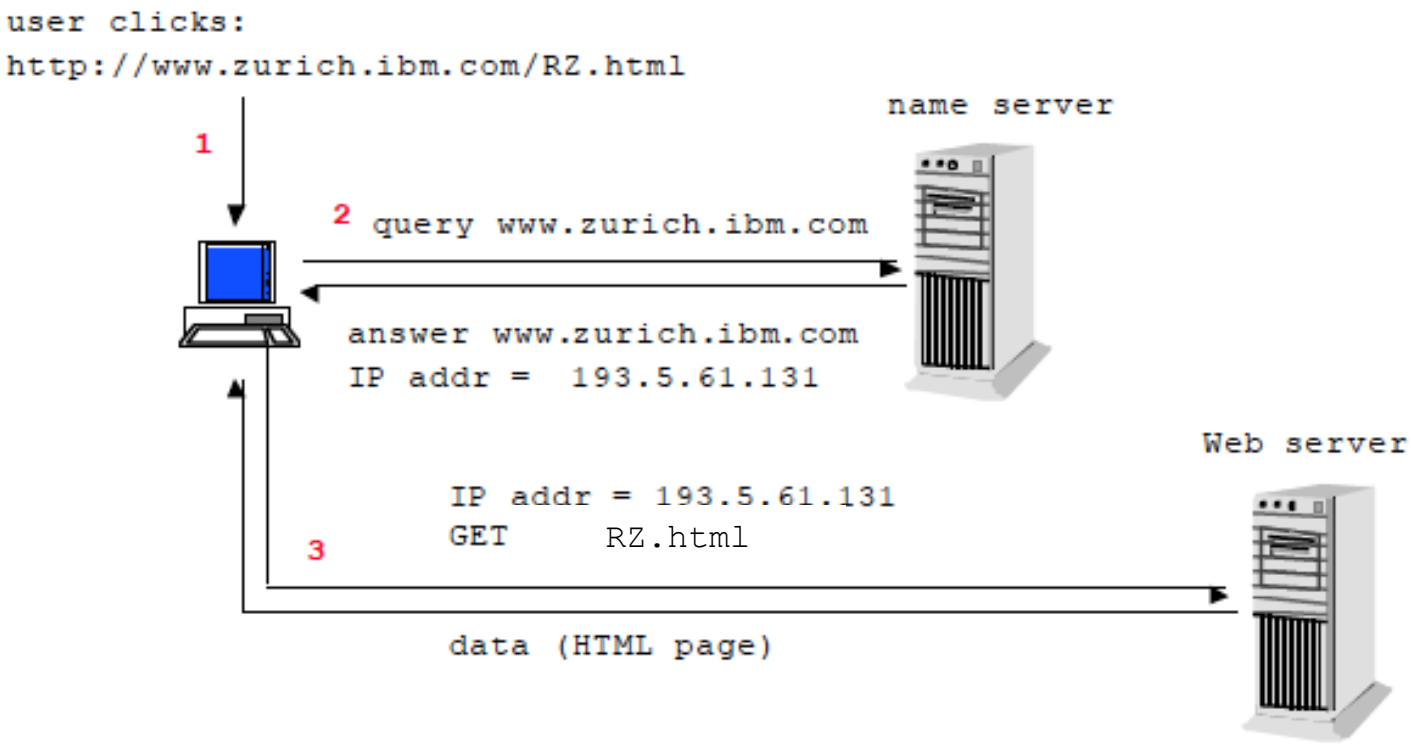
A network service is an application based on application-layer network protocols.

Examples:

- name service
- e-mail
- Web
- text messaging
- P2P file sharing (BitTorrent, eMule, ..)
- multi-user network games
- streaming stored video (YouTube, Netflix, ..)
- voice over IP, real-time video conferencing (Skype, Teams, ..)
- social networks

Network services

Example with two network services: name and Web



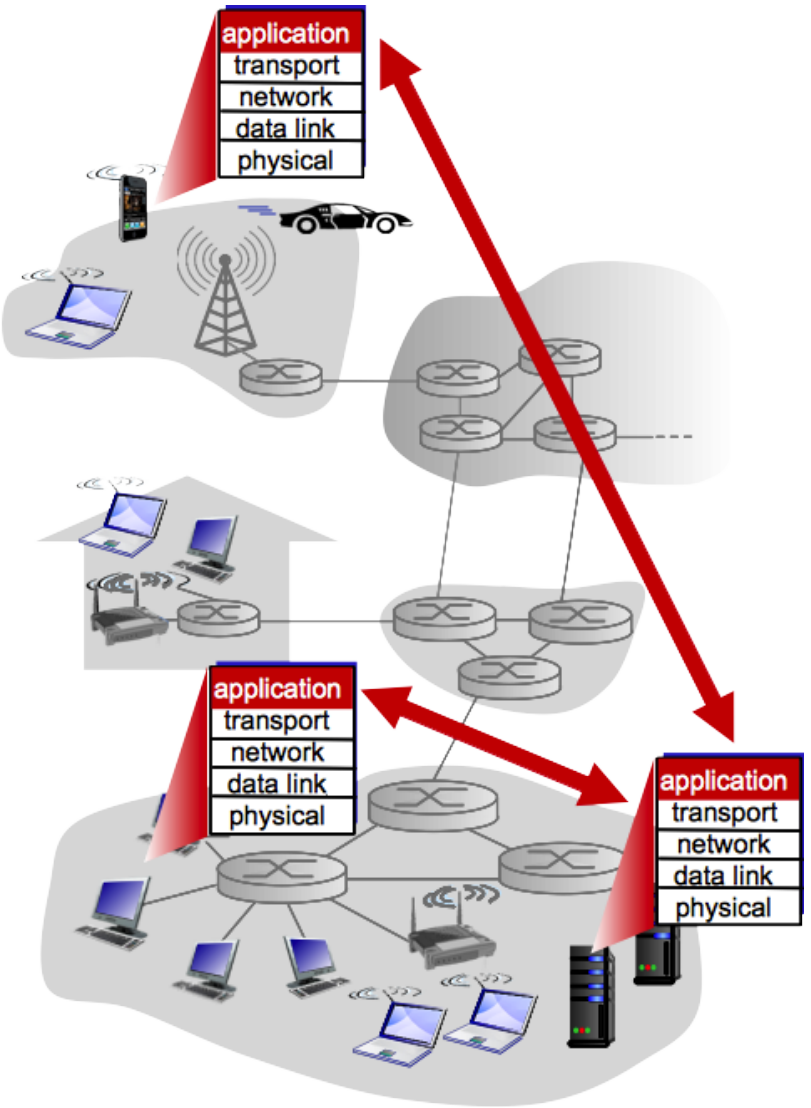
Creating a network service

Write programs that

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

End-to-end principle

- network-core devices do not run network services



Communicating processes

Process: program running within a host

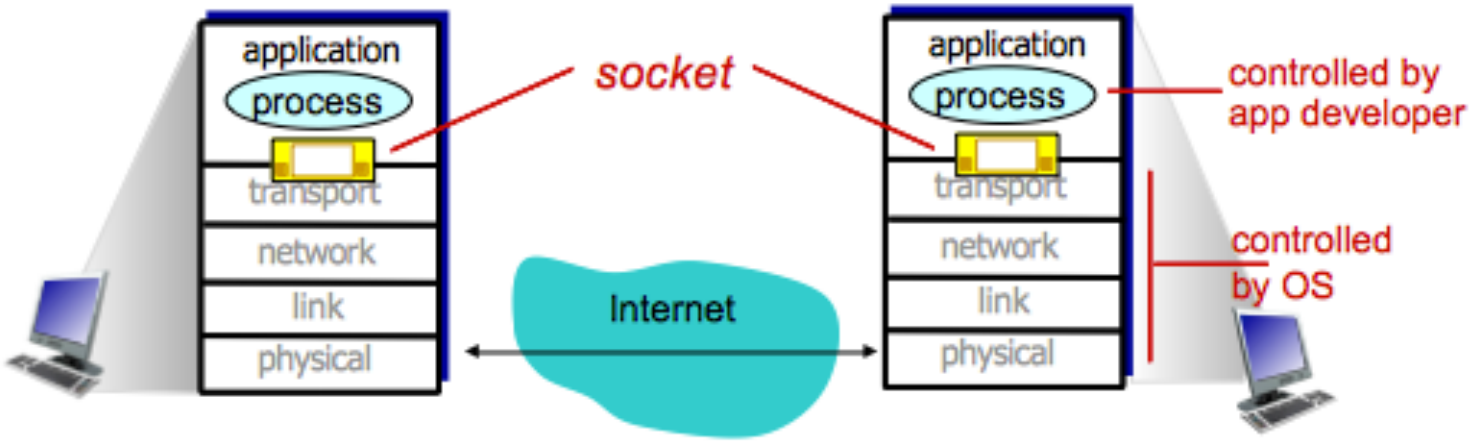
- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

Client process: process that initiates communication

Server process: process that waits to be contacted

Sockets

A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the transport layer can identify the application that data is destined to be sent to.



Addressing

To receive messages, a process must have an *identifier*

Host device has unique IP address

Q: does IP address of the host on which the process runs suffice for identifying the process?

A: no, *many* processes can be running on same host

The *identifier* includes both IP address and port number associated with process on host

Example port numbers:

HTTP server: 80

mail server: 25

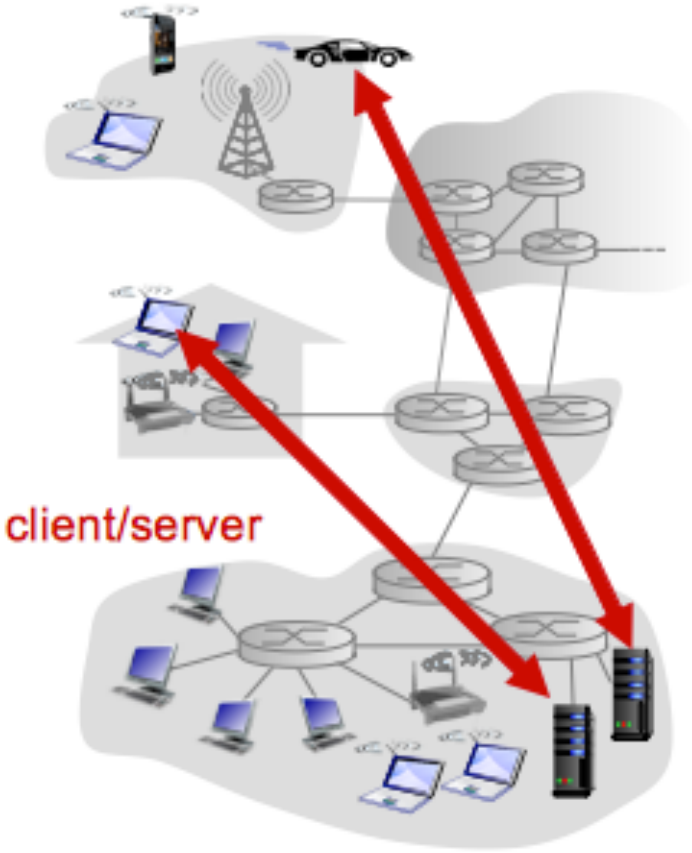
Client-server architecture

Server:

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

Client:

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



Peer-to-peer (P2P) architecture

No always-on host

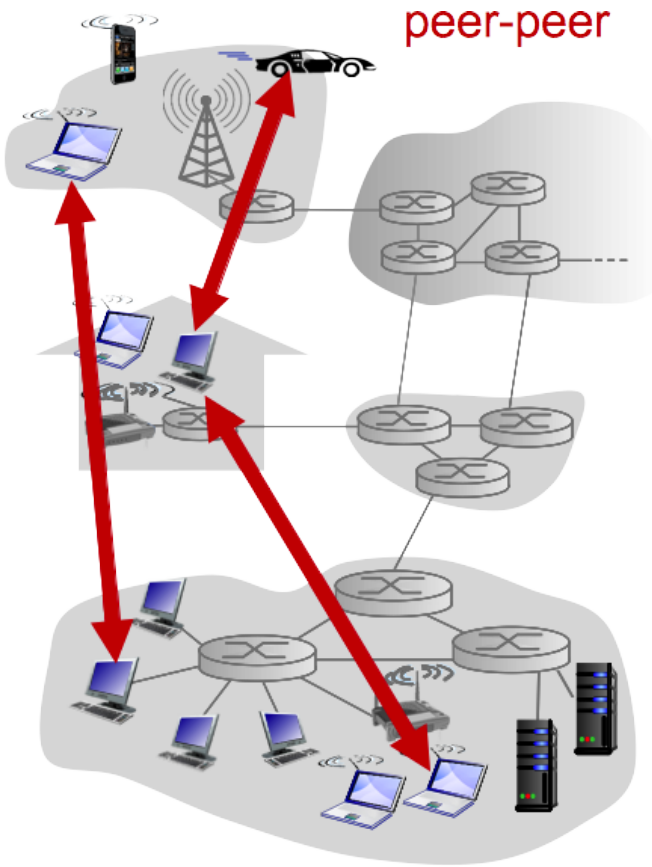
Peer = client+server

Peers request services from other peers, provide services 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



Transport requirements of network services

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, 100s ms
stored audio/video	loss-tolerant	same as above	yes, few sec.
interactive games	loss-tolerant	few kbps - 10kbps	yes, 100s ms
text messaging	no loss	elastic	yes and no

Transmission Control Protocol (TCP)

[RFC 793] invented by Kahn and Cerf in 1973

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

User Datagram Protocol (UDP) [RFC 768]

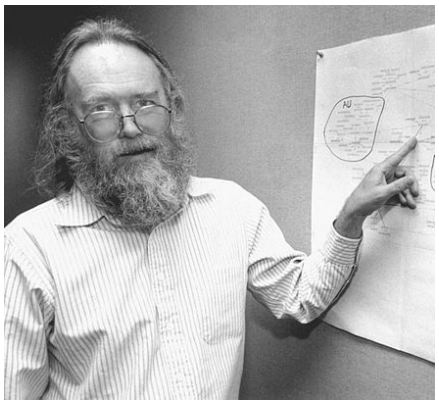
UDP (invented by Reed and Postel in 1978) is based on the concept of *datagram* (invented by Louis Pouzin in the early 1970s).

Louis Pouzin

Unreliable data transfer between sending and receiving process

Does not provide: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup

David P. Reed



Jonathan B. Postel

Network services and transport protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP (e.g., YouTube), RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP

World Wide Web

A short review:

- the WWW is an Internet-based information space where documents and other resources are identified by addresses
- a web page is a HTML document that is linked to other web pages or resources

Tim Berners-Lee, developer of the first HTTP server and browser (1989)

Uniform Resource Identifiers (URIs)

Generic Uniform Resource Identifier (URI) form [RFC3986]:

scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]

URI are used throughout HTTP as the means for identifying resources [RFC7231]. In the WWW context, URIs are informally denoted as Uniform Resource Locators (URLs).

http://www.example.com/hello.txt

http = protocol
www.example.com = host name
/hello.txt = resource

HTTP overview

HTTP: HyperText Transfer Protocol

It is an application layer protocol.

HTTP provides a uniform interface for interacting with a **resource**, regardless of its type, nature, or implementation, via the manipulation and transfer of **representations.** [RFC7231]

HTTP adopts a *client/server* model:

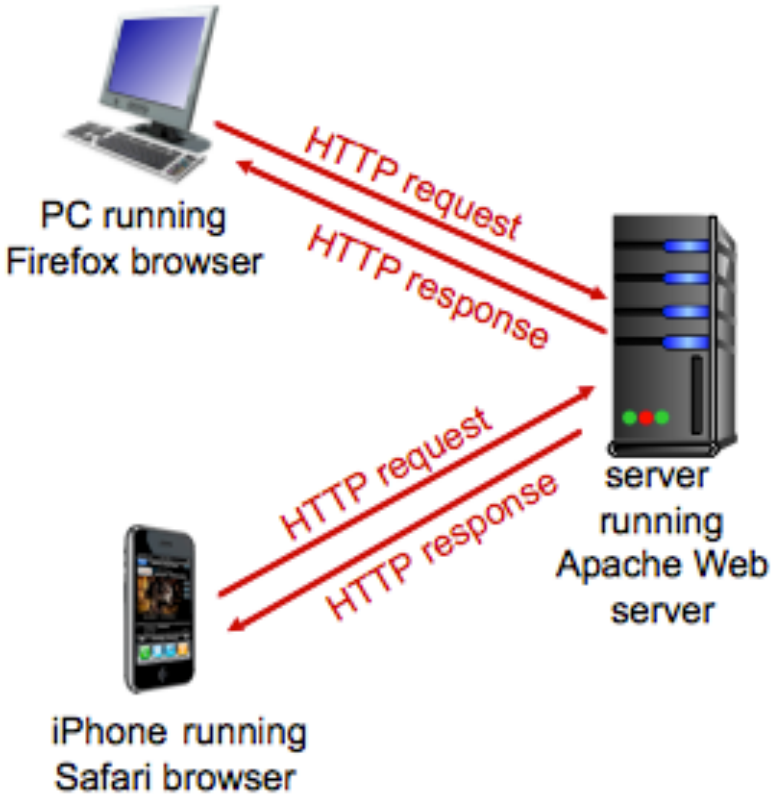
- *user agent*: client that initiates an HTTP connection and sends HTTP requests; e.g., browser, bot
- *origin server*: program that accepts (or refuses) HTTP connections and owns the requested resources

HTTP overview

local cache: local memory (both client and server may have one)

proxy: intermediary application having both client and server functionalities (may be used for several reasons, e.g., HTTP tunneling)

gateway: intermediary application acting on behalf of the server (the client is not aware of the gateway)



HTTP overview

History:

- 0.9: simple client-server protocol for requesting HTML files; only GET method
- 1.0 [RFC 1945]: stateless, generic protocol (i.e., independent of resource representation format: HTML documents, binary files, etc.); new methods are introduced
- 1.1 [RFC 2068, 2069, 2616, 2617, now obsoleted by RFC 7230-7235]: the current version, with caching mechanisms, multihoming, persistent connections

HTTP overview

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

Protocols that maintain state are complex!

If the server/client crashes, its views of "state" may be inconsistent, thus it must be reconciled.

HTTP connections

non-persistent HTTP

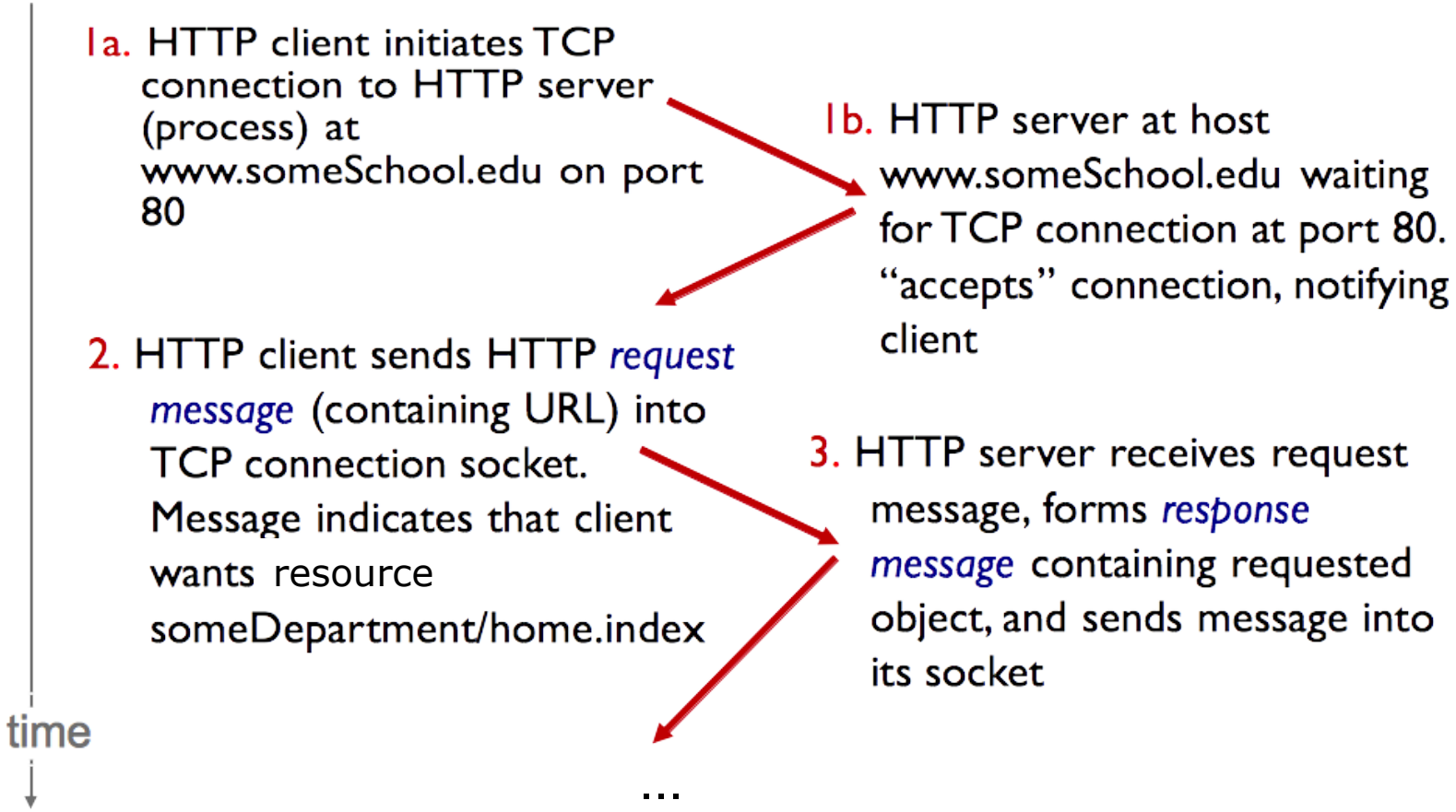
- at most one resource sent over TCP connection
- connection then closed
- downloading multiple resource representations requires multiple connections

persistent HTTP

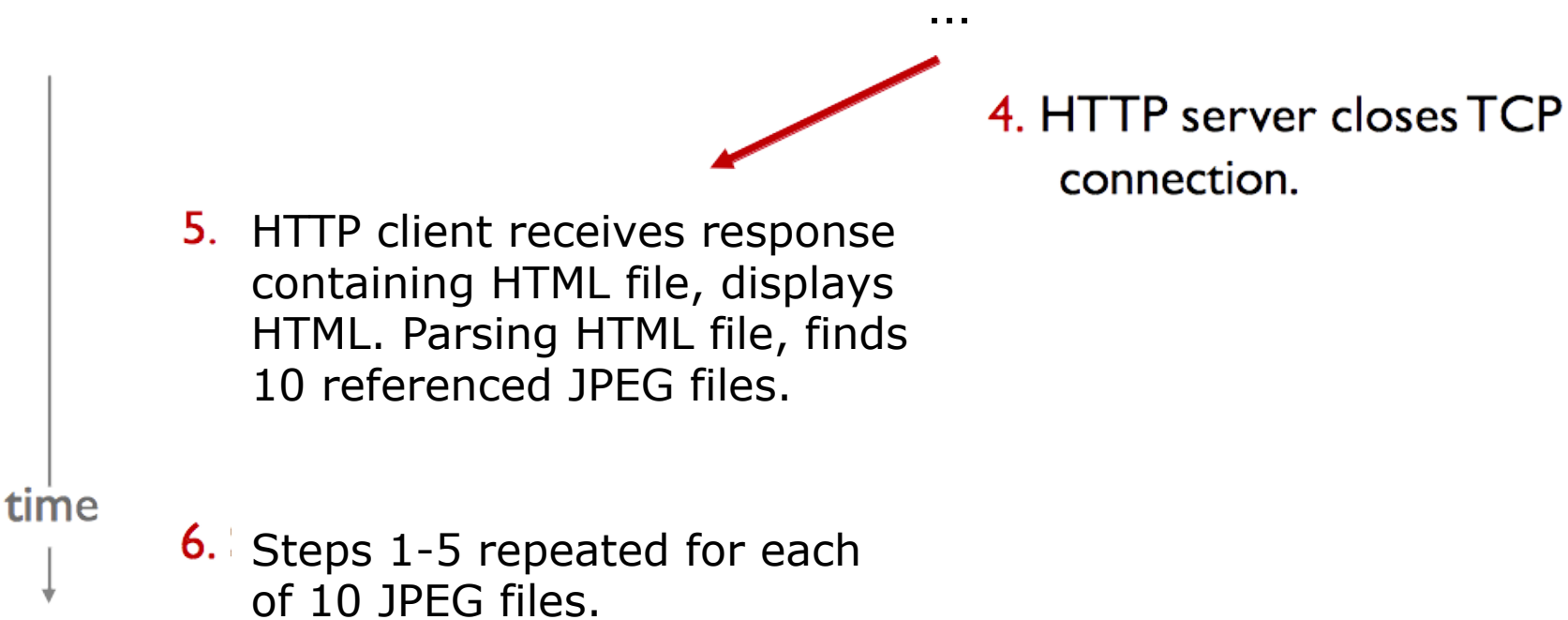
- multiple resource representations can be sent over single TCP connection between client and server

Non-persistent HTTP

Suppose the user enters the following URI:
www.someSchool.edu/someDepartment/home.index



Non-persistent HTTP



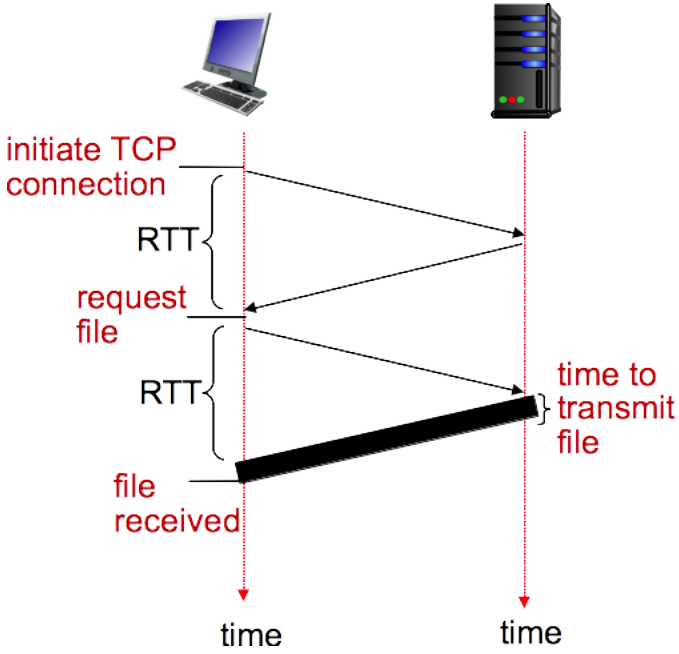
Non-persistent HTTP

Round Trip Time (RTT):

time for a small packet to travel from client to server and back

Response time:

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



Non-persistent HTTP issues:

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

Persistent HTTP

The server leaves the connection open after sending response.

Subsequent HTTP messages between same client/server are sent over the open connection.

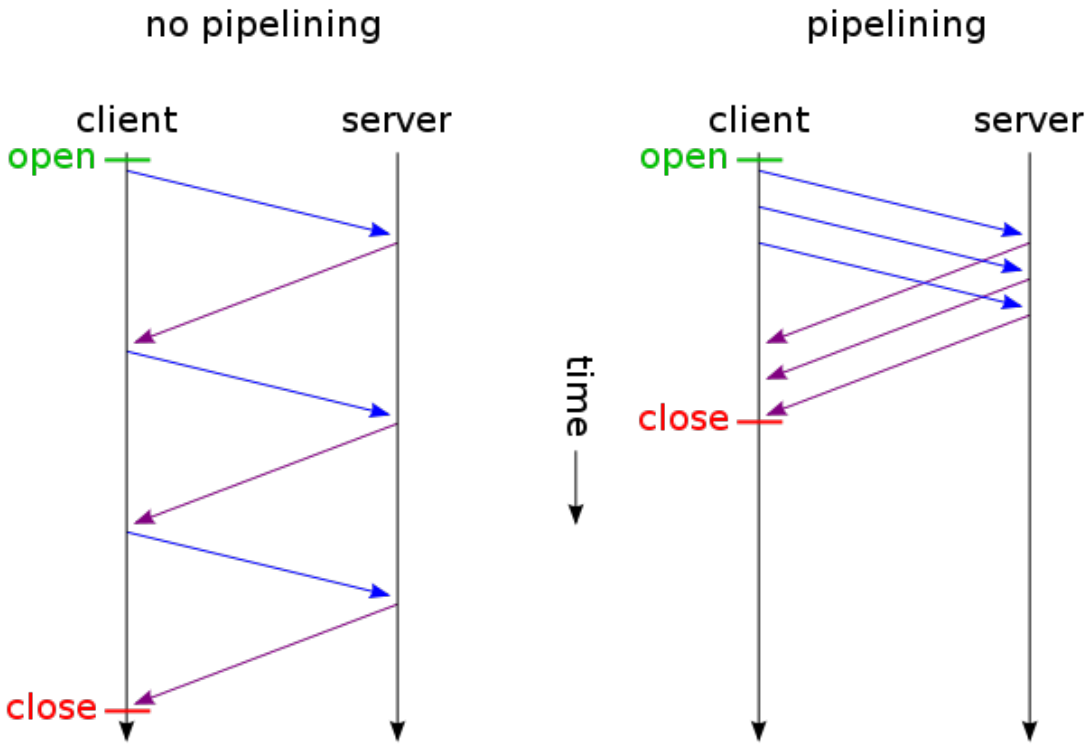
The client sends requests as soon as it encounters a referenced resource.

As little as **1 RTT for the connection initiation, then 1 RTT per resource**.

Allows for *pipelining*, which further reduces the response time.

Pipelining

HTTP pipelining is a technique in which multiple HTTP requests are sent on a single TCP connection without waiting for the corresponding responses.



HTTP message

The general format is:

```
HTTP-message = start-line
*( header-field CRLF )
CRLF
[ message-body ]
```

```
start-line = request-line / status-line

request-line = method SP request-target SP HTTP-version CRLF

status-line = HTTP-version SP status-code SP reason-phrase CRLF
```

where CRLF = \r\n and SP = space

HTTP header

Headers are in MIME format [RFC 822] and specify:

- general transmission features
- features of the transmitted entity
- request features
- response features

Date: date and time of the transmission
MIME-Version: always 1.0
Transfer-Encoding: encoding format used for the transmission (e.g., "chunked")
Cache-Control: requested or suggested caching method for the resource representation
Connection: to be kept alive after the response, or to be closed, etc.
Via: used by proxies and gateways

HTTP header

Headers are in MIME format [RFC 822] and specify:

- general transmission features
- features of the transmitted resource
- request features
- response features

Content-type: MIME type of the attached resource representation (mandatory for messages with a body)
Content-Length: length of the body, in bytes
Content-Encoding, Content-Language, Content-Location, Content-MD5, Content-Range: encoding, language, URL, MD5 digest value and required range of the enclosed resource representation
Expires: a date after which the enclosed resource representation is no more valid
Last-Modified: date and time of the last modification to the enclosed resource representation

HTTP request message

Example:

request line

header lines

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

```
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
\r\n
```

Methods

[RFC7231]

Method	Description	Sec.
GET	Transfer a current representation of the target resource.	4.3.1
HEAD	Same as GET, but only transfer the status line and header section.	4.3.2
POST	Perform resource-specific processing on the request payload.	4.3.3
PUT	Replace all current representations of the target resource with the request payload.	4.3.4
DELETE	Remove all current representations of the target resource.	4.3.5
CONNECT	Establish a tunnel to the server identified by the target resource.	4.3.6
OPTIONS	Describe the communication options for the target resource.	4.3.7
TRACE	Perform a message loop-back test along the path to the target resource.	4.3.8

Idempotent and safe/unsafe HTTP methods

An **idempotent** method is a HTTP method that can be called many times without different outcomes. It would not matter if the method is called only once, or ten times over. The result should be the same. Only idempotent methods can be pipelined.

A **safe** method is a HTTP method that does not modify resources. For instance, using GET or HEAD on a resource URI, should NEVER change the resource.

Methods

The **GET** method (which is both safe and idempotent) is used to request a resource. A representation of the resource is returned in the response message.

The **POST** method (which is neither safe nor idempotent) is used to create/update a resource (in particular, *subordinate* resources).

The **PUT** method (which is idempotent but not safe) is used to update/create a resource.

The difference between POST and PUT is that PUT requests are idempotent. That is, calling the same PUT request multiple times will always produce the same result. In contrast, calling a POST request repeatedly may have side effects like creating multiple copies of the same resource.

Methods

Resource creation: POST to a resource collection --> the server takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc.

It is best practice to return the status of “201 Created” and the location of the newly created resource, since its location was unknown at the time of submission. This allows the client to access the new resource later, if it needs to.

```
POST /accounts HTTP/1.1
{
        …
}
```

Response:

```
201 Created
Location: https://api.stormpath.com/accounts/abcdef1234
```

Methods

Resource creation: PUT allows the client to specify the resource identifier of the resource to be created.

```
PUT /accounts/abcdef1234 HTTP/1.1

{
      “givenName”: “John”,
      “surname”: “Smith”,
      “status”: “enabled”
}
```

Methods

For resource update: POST allows the client to send either all available values or just a subset of available values:

```
POST /accounts/abcdef1234 HTTP/1.1
{
      “status”: “disabled”
}
```

For this case, it would be better using the PATCH method [RFC 5789], if supported by the server.

With PUT, it must be a full resource update: the client has to send *all* attribute values in a PUT request to guarantee idempotency.

```
PUT /accounts/abcdef1234 HTTP/1.1
{
      //FULL RESOURCE UPDATE
      “givenName”: “J”,
      “surname”: “Smith”,
      “status”: “Enabled”
}
```

Methods

Generally, in practice, always use

- **POST for resource creation**

- **PUT for resource update**

GET /device-management/devices : Get all devices
POST /device-management/devices : Create a new device

GET /device-management/devices/{id} : Get the device information identified by "id"
PUT /device-management/devices/{id} : Update the device information identified by "id"

Methods

The **DELETE** method (which is idempotent but not safe) deletes the specified resource.

The **HEAD** method (which is both safe and idempotent) is similar to the GET method, but the server must respond with the header only, not the body of the message.
It is used to check

- the validity of a URI
- the accessibility of a URI
- the cache coherence of a URI

Request features in the HTTP header

User-Agent

- a string which decribes the client that originated the request; usually: type, version and o.s. of the client

Referer

- URL of the element (e.g., web page) from which the current request originated
- must be empty if the requested URL was typed in the browser by the user or selected from the bookmarks
- important for user profiling and advertising
- spelling error for historical reasons (it should be Referrer)

Host

- domain name and port to which the request is made
- why Host? because the URI placed in the request is just the part that is local to the server

Request features in the HTTP header

From

- e-mail address of the requester
- requires user agreement, thus no one uses this header field

Range

- range (in bytes) of the request
- used to resume downloads

Accept, Accept-Charset, Accept-Encoding, Accept-Language

- format negotiation
- the client specifies what it is able to accept, the server finds the best match

If-Modified-Since, If-Unmodified-Since

- used in conditional requests

Authorization, Proxy-Authorization

Conditional GET

Goal: don't send resource representation if client has up-to-date cached version

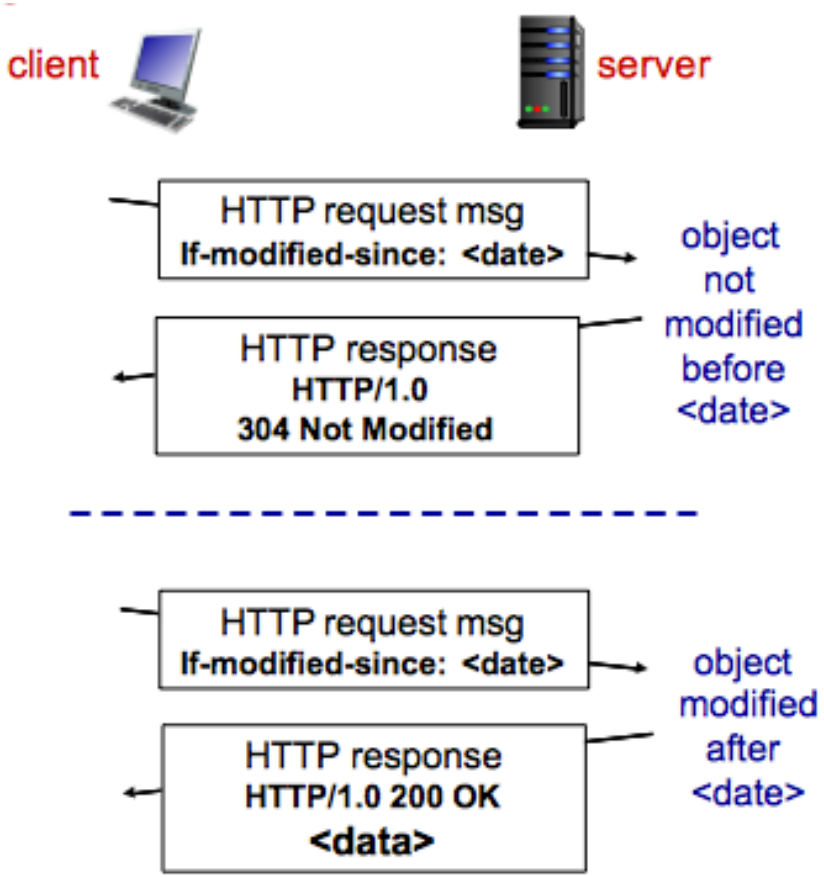
- no transmission delay
- lower link utilization

client: specify date of cached copy in HTTP request

If-Modified-Since: <date>

server: response contains no resource representation if cached copy is up-to-date:

HTTP/1.0 304 Not Modified



Example GET with telnet

```
spumante:~ ardarico$ telnet checkip.dyndns.org 80
Trying 91.198.22.70...
Connected to checkip.dyndns.com.
Escape character is '^]'.
GET / HTTP/1.1
HOST: checkip.dyndns.org

HTTP/1.1 200 OK
Content-Type: text/html
Server: DynDNS-CheckIP/1.0
Connection: close
Cache-Control: no-cache
Pragma: no-cache
Content-Length: 105

<html><head><title>Current IP Check</title></head><body>Current IP Address:
160.78.28.223</body></html>
Connection closed by foreign host.
```

Hit carriage return twice to send the request.

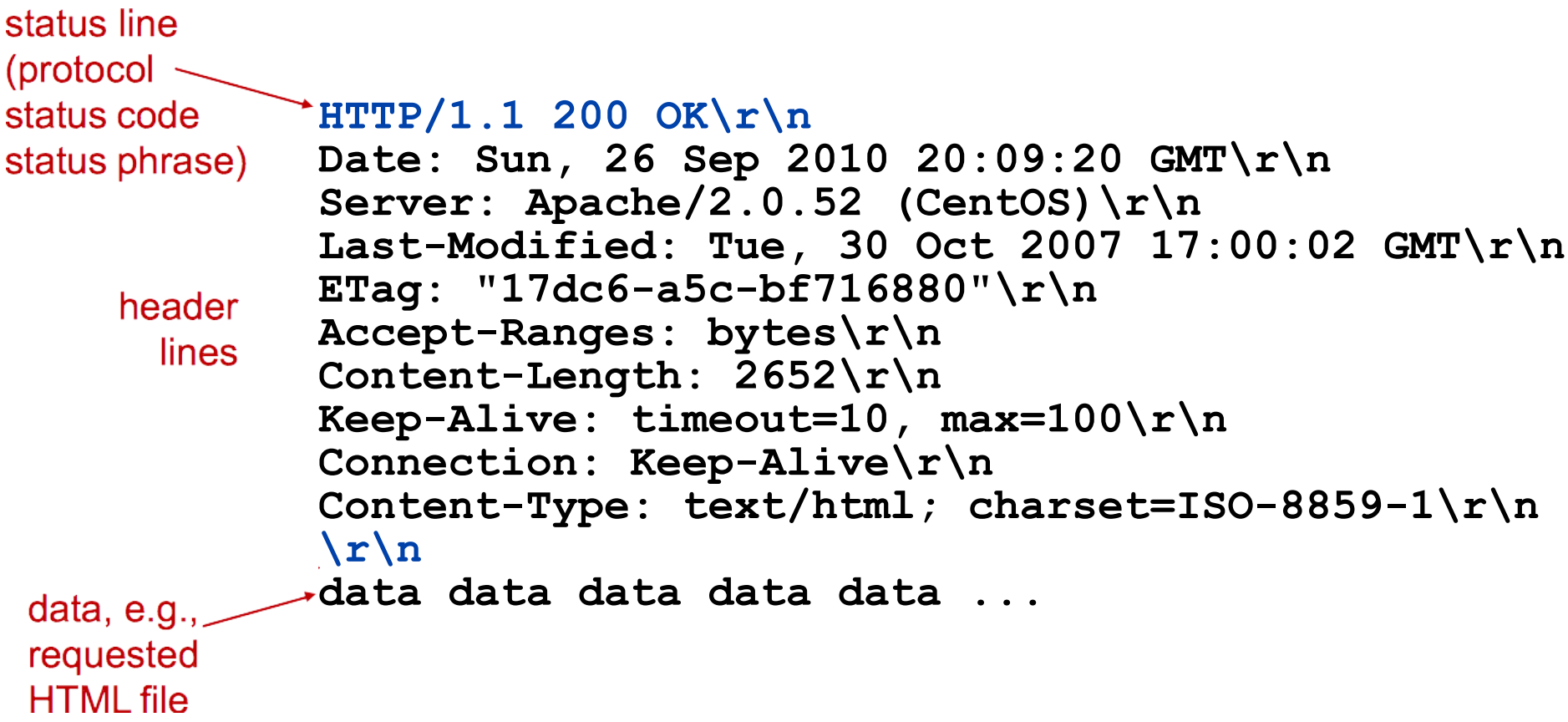
Example OPTIONS with curl

```
spumante:dci ardarico$ curl -v -X OPTIONS http://www.example.com
* Adding handle: conn: 0x7f96b900b600
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x7f96b900b600) send_pipe: 1, recv_pipe: 0
* About to connect() to www.example.com port 80 (#0)
*   Trying 93.184.216.34...
* Connected to www.example.com (93.184.216.34) port 80 (#0)
> OPTIONS / HTTP/1.1
> User-Agent: curl/7.30.0
> Host: www.example.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Allow: OPTIONS, GET, HEAD, POST
< Cache-Control: max-age=604800
< Date: Tue, 06 Oct 2015 09:11:29 GMT
< Expires: Tue, 13 Oct 2015 09:11:29 GMT
* Server EOS (lax004/2816) is not blacklisted
< Server: EOS (lax004/2816)
< x-ec-custom-error: 1
< Content-Length: 0
<
* Connection #0 to host www.example.com left intact
```

curl is a command line tool for getting or sending files using URL syntax. It can be used to send any HTTP request, as well.

HTTP response message

Example:



Status codes

The status code appears in the first line, in every server-to-client response message.

It is made of three digits: one for the class, two for the specific request.

1xx - Informational: a temporary response, while the request is being handled

2xx - Successful: the server has received, understood and accepted the request

3xx - Redirection: the server has received and understood the request, but more actions from the client are necessary to completely fulfill the request

4xx - Client error: syntax error or unauthorized request

5xx - Server error: the server is unable to fulfill the request

Status codes

100 Continue (if the client has not yet sent the body)

200 Ok (successful GET)
201 Created (succesful POST/PUT)

301 Moved permanently (unvalid URI, the server knows the new location)

400 Bad request (syntax error in the request)
401 Unauthorized (missing authorization)
403 Forbidden (the request cannot be authorized)
404 Not found (wrong URI)

500 Internal server error (typical from web applications)
501 Not implemented (the method is unknown to the server)

Response features in the HTTP header

Server

- a string that describes the server: type, version, o.s.

WWW-Authenticate

- contains a challenge (i.e., a special code) for the client, in case of 401 status code (Unauthorized); the client will use the challenge to generate an authorization code for next request

Accept-Ranges

- specifies the types of ranges that can be accepted (bytes or none)

Cookies

Many Web sites use cookies

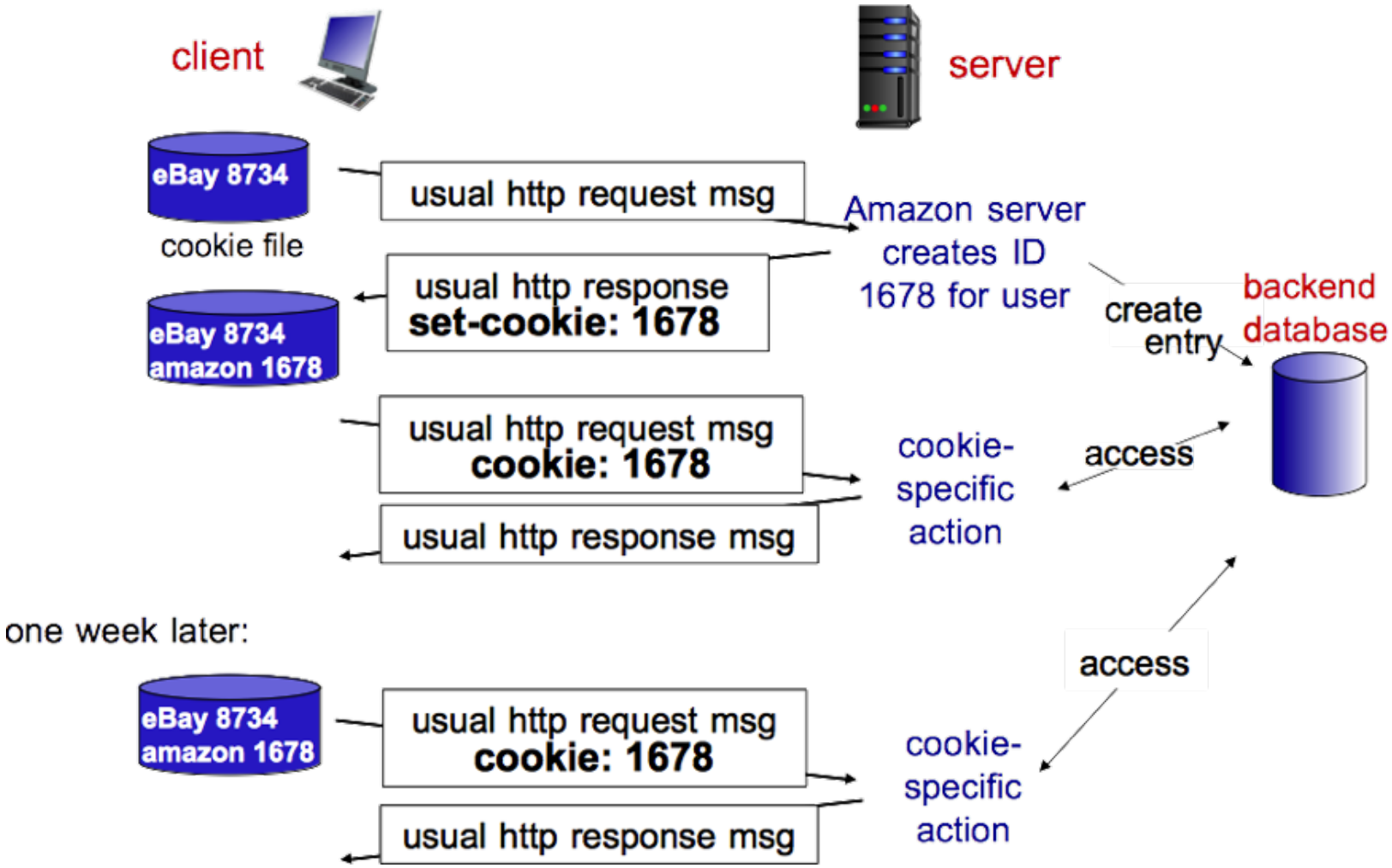
Four components:

- cookie header line of first HTTP *response* message
- cookie header line in next HTTP *request* messages
- cookie file kept on user's host, managed by user's browser
- back-end database at Web site

What cookies can be used for:

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

Cookies


client
server
eBay 8734
cookie file
usual http request msg
Amazon server creates ID 1678 for user
usual http response
set-cookie: 1678
eBay 8734
amazon 1678
create entry
backend database
usual http request msg
cookie: 1678
cookie-specific action
access
usual http response msg
one week later:
access
eBay 8734
amazon 1678
usual http request msg
cookie: 1678
cookie-specific action
usual http response msg

Proxy

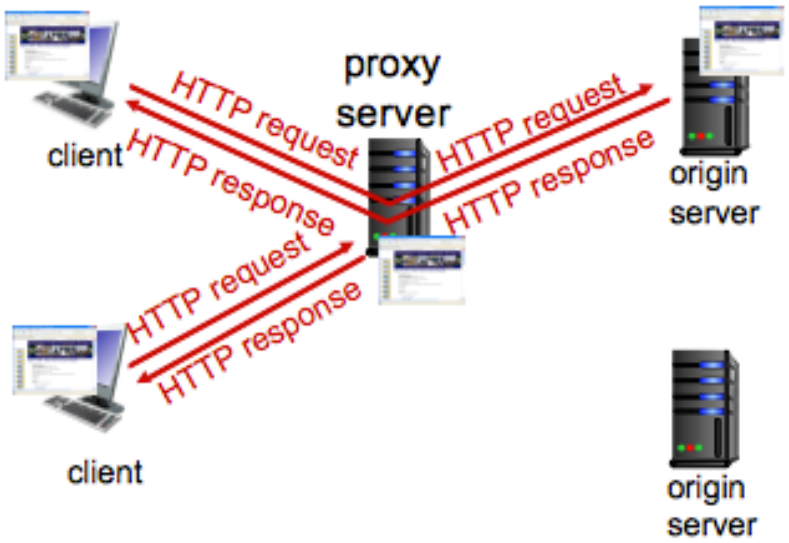
A proxy intermediary application having both client and server functionalities.

A **transparent proxy** is a proxy that does not modify the request or response beyond what is required for proxy authentication and identification. Example: HTTP tunneling.

A **non-transparent proxy** is a proxy that modifies the request or response in order to provide some added service to the user agent, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering.

Web caching

What: using a proxy server to satisfy client requests without involving the origin server



The user tells the proxy address to the browser, to enable cache-aware Web access

The browser sends all HTTP requests to the proxy:

- if the requested resource representation is in cache, then the proxy returns it
- else the proxy requests the resource representation from the origin server, then returns it to client

Web caching

The proxy acts as both client and server:

- server for original requesting client
- client to origin server

Typically, caches are installed by ISPs (university, company, residential ISP)

Why Web caching?

- reduces response time for client requests
- reduces traffic on an institution's access link
- enables "poor" content providers to effectively deliver content

Web caching

To measure the average occupancy of a server or resource during a specified period of time (usually a "busy hour"), we use

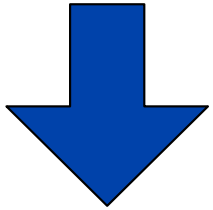
Traffic Intensity: aL/R

where

a = average arrival rate of packets [packets/s]
L = average packet length [bits]
R = transmission rate [bits/s]

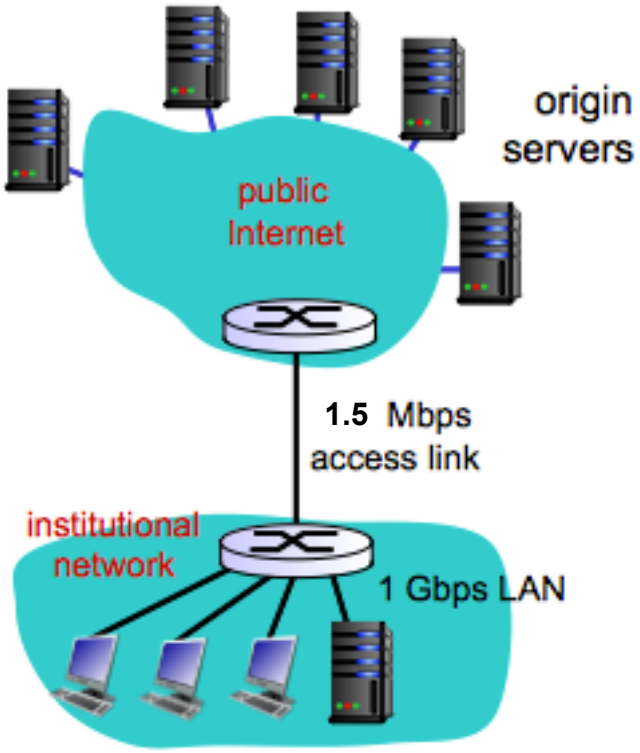
Web caching - example

- avg resource representation size: 100 Kbit
- avg request rate from browsers to origin servers: 15/sec
- LAN rate: 1 Gbps
- access link rate: 1.5 Mbps



- traffic intensity on the LAN: 0.0015
- traffic intensity on the access link: 1

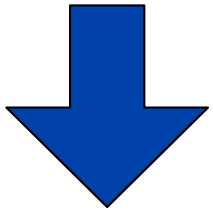
The LAN delay can be neglected, but the delay on the access link becomes very large - RTT on the order of minutes.



Web caching - example

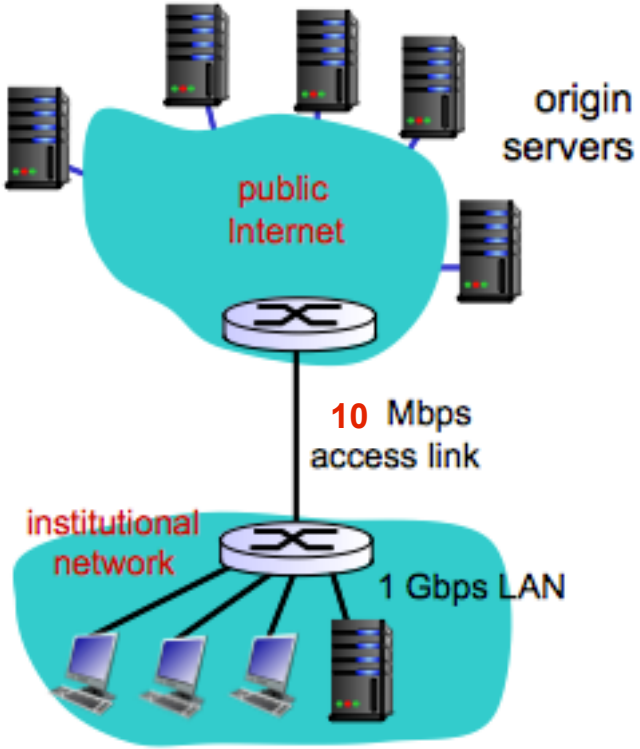
Expensive solution:

- avg resource representation size: 100 Kbit
- avg request rate from browsers to origin servers: 15/sec
- LAN rate: 1 Gbps
- access link rate: 10 Mbps



- traffic intensity on the LAN: 0.0015
- traffic intensity on the access link: 0.15

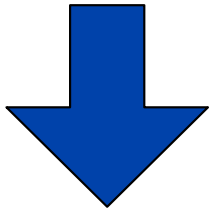
Now also the delay between the two routers is negligible, but this solution may be very costly for the institution.



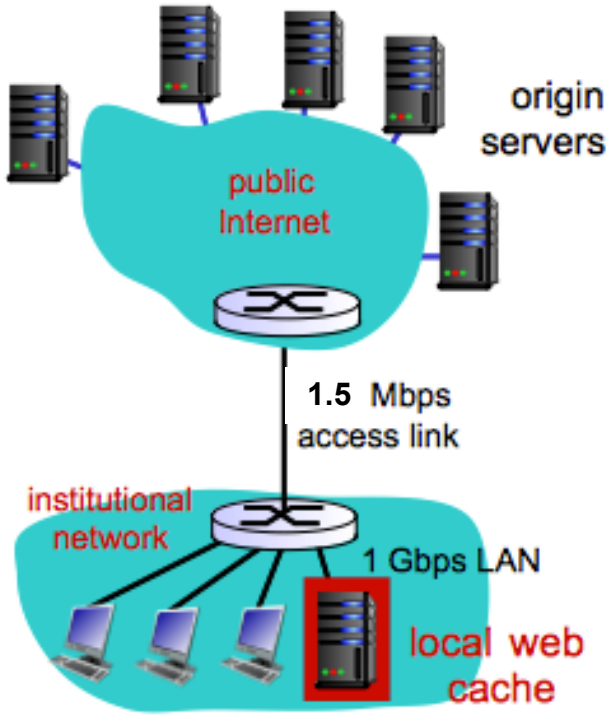
Web caching - example

Wiser solution:

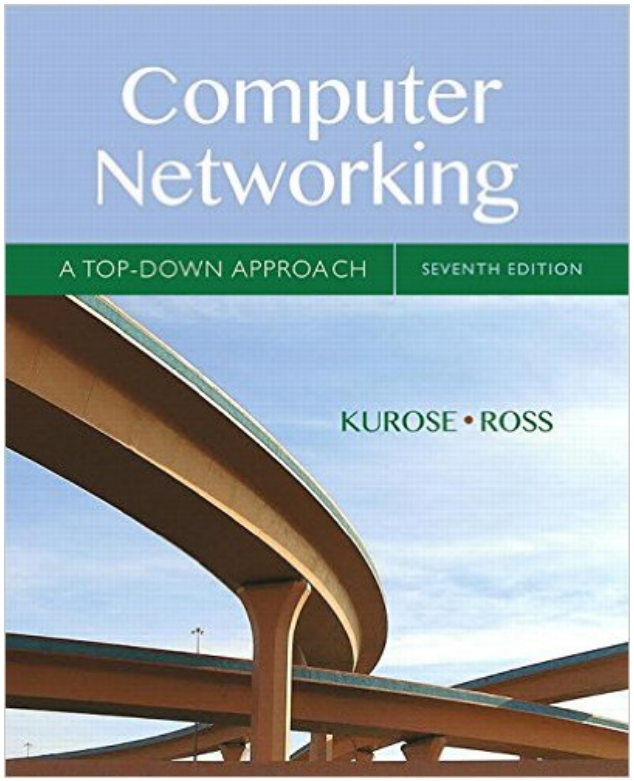
- avg resource representation size: 100 Kbit
- avg request rate from browsers to origin servers: 15/sec
- LAN rate: 1 Gbps
- access link rate: 1.5 Mbps
- institutional cache



- traffic intensity on the LAN: 0.0015
- traffic intensity on the access link: depends on the **hit rate** of the cache (hr = 0.4 --> traffic intensity = 0.6)



References



Jim Kurose, Keith Ross
Computer Networking: A Top Down Approach
Addison-Wesley, May 2016

RFC 7230 ... 7235