Computer Networks CN MID I

CHAPTER 2 IMPORTANT STUFF

Client–Server Paradigm

Server

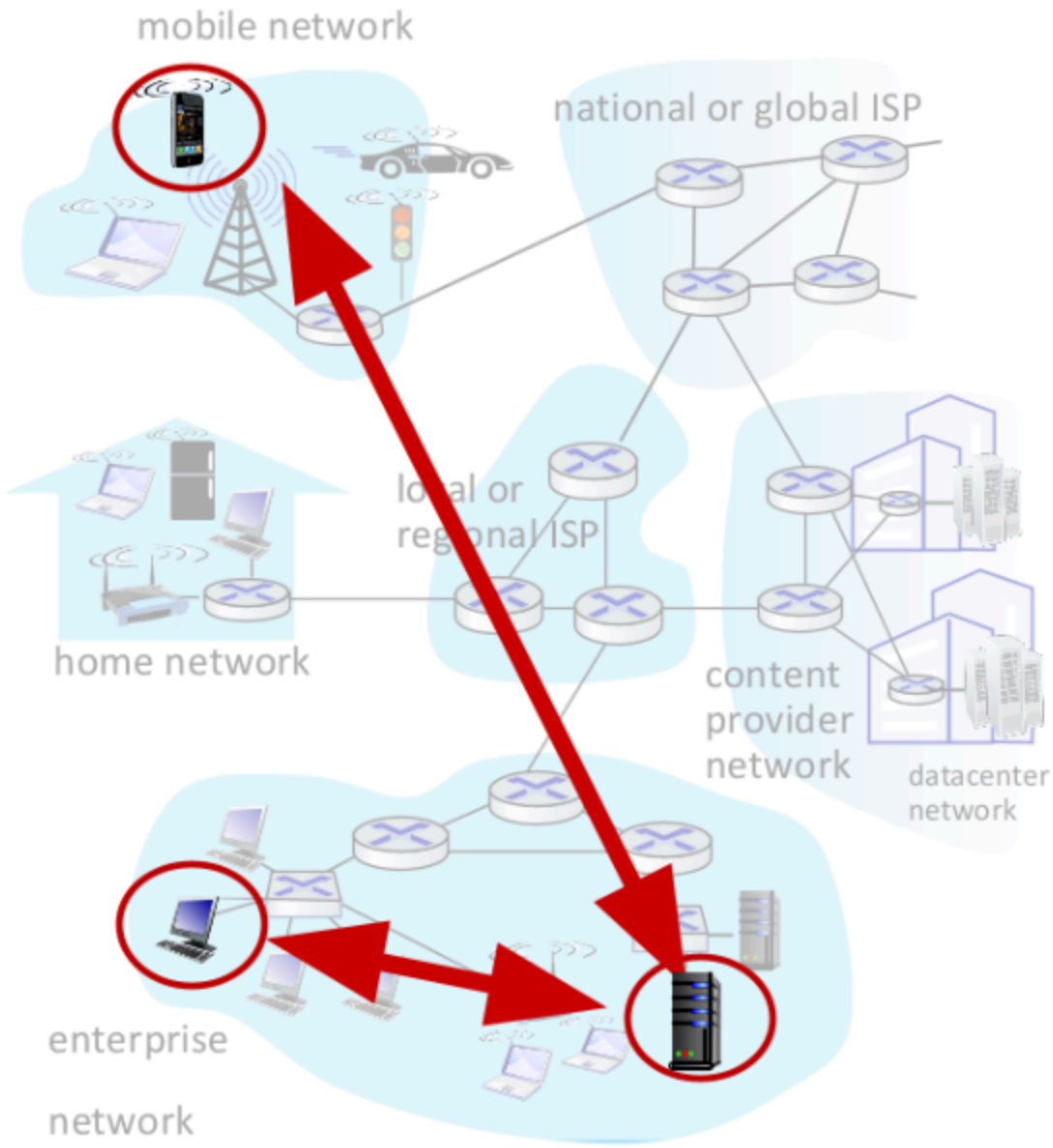
- **Always-on host** → Continuously running to provide services.
- **Permanent IP address** → Easy to locate and connect to.
- **Often located in data centers** → Ensures scalability, reliability, and better infrastructure.

Clients

- **Contact and communicate with the server** → Request services or resources.
- **May be intermittently connected** → Not always online (e.g., laptops, mobile devices).
- **May have dynamic IP addresses** → Assigned temporarily by ISPs.
- **Do not directly communicate with each other** → All interaction goes through the server.

Examples of Client–Server Applications

- **HTTP** → Web browsing.
- **IMAP** → Email retrieval.
- **FTP** → File transfer.



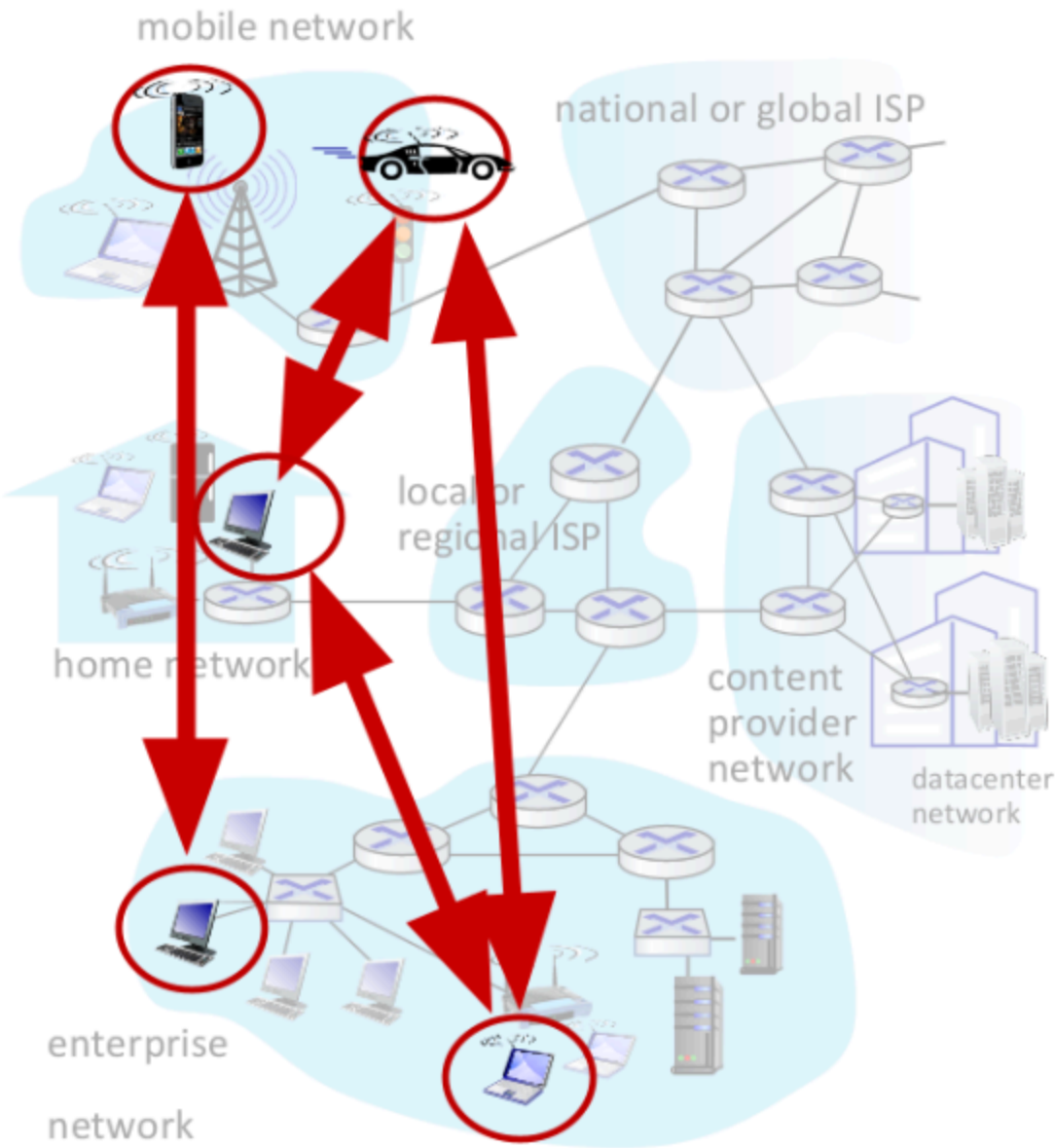
Peer-to-Peer (P2P) Architecture

Characteristics

- **No always-on server** → No central server needed.
- **Arbitrary end systems directly communicate** → Peers connect directly with each other.
- **Peers act as both clients and servers** → Each peer can **request** services and also **provide** services.
- **Self-scalability** → As more peers join:
  - They add **service demand**, but also
  - Contribute **service capacity** (resources, bandwidth).
- **Dynamic connectivity** → Peers may be intermittently connected and often have changing IP addresses.
- **Complex management** → Coordination, reliability, and security are more difficult than in client-server.

Example

- **P2P file sharing** → *BitTorrent* is a well-known system where files are distributed among peers.



What is a Process?

- A **process** is a program running within a host.

Communication Types

1. **Within the Same Host**
    - Processes communicate using **Inter-Process Communication (IPC)**.

- IPC mechanisms are defined by the **Operating System** (e.g., pipes, shared memory, sockets).

2. **Across Different Hosts**
    - Processes communicate by **exchanging messages** over the network.

Sockets

Definition

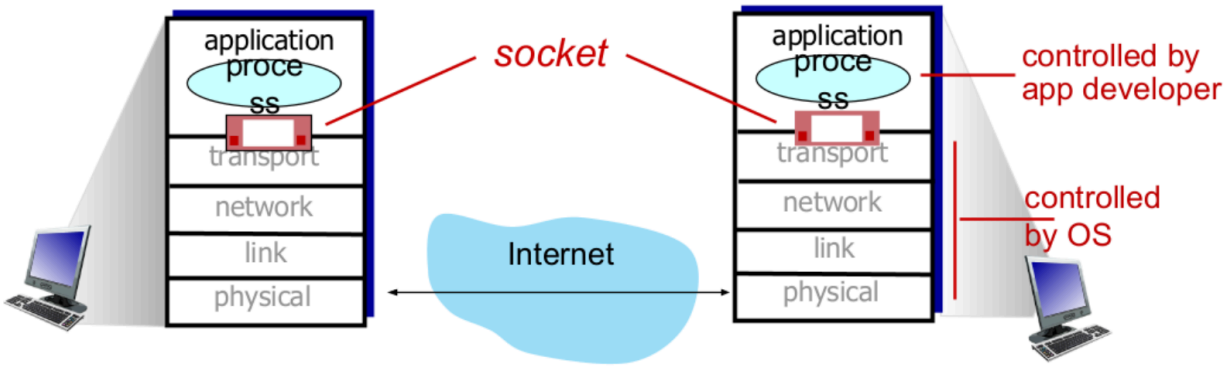
- A **socket** is the interface between a process and the network.
- It allows processes to **send** and **receive** messages.

Analogy

- A socket is like a **door**:
    - The **sending process** pushes a message “out the door.”
    - The **network/transport layer infrastructure** (on the other side of the door) is responsible for carrying the message to the destination.
    - The **receiving process** gets the message “in through its door.”

Key Points

- **Two sockets are involved** in communication:
    - One at the **sender’s side**.
    - One at the **receiver’s side**.
- Processes rely on the underlying **transport protocols** (e.g., TCP, UDP) to ensure message delivery.



Addressing Processes

Why Addressing is Needed

- For a process to **receive messages**, it must have a **unique identifier**.

Identifiers

- Each **host device** has a **unique 32-bit IP address** (IPv4).
- But...

  Q: Does the IP address of the host suffice to identify a process?

  A: No.

  - Many processes can run on the same host.
  - Therefore, we need both:
    - **IP Address** → identifies the host.
    - **Port Number** → identifies the specific process on that host.

Examples of Well-Known Port Numbers

- **HTTP server** → Port **80**
- **Mail server (SMTP)** → Port **25**

Example

To send an HTTP message to **gaia.cs.umass.edu** web server:

- **IP Address:** `128.119.245.12`
- **Port Number:** `80`

Application-Layer Protocol

What it Defines

1. Types of Messages Exchanged

- Example: **Request, Response**

2. Message Syntax

- What fields are included in a message.
- How fields are **structured and delineated**.

3. Message Semantics

- The **meaning** of the information in each field.

4. Rules of Communication

- When and how processes **send** and **respond** to messages.

Types of Protocols

1. Open Protocols

- Defined in **RFCs (Request for Comments)**.
- Publicly available, allowing **interoperability** across different systems.
- Examples: **HTTP, SMTP**

2. Proprietary Protocols

- Owned and controlled by a **company/organization**.
- Not publicly defined, so only official clients/servers can use them.
- Example: **Zoom**

What Transport Service Does an Application Need?

1. Data Integrity

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

2. Timing

- Some apps are **time-sensitive**:
  - Internet telephony, interactive games → need **low delay** to be effective.

3. Throughput

- Some apps require a **minimum throughput** (e.g., multimedia streaming).
- Other apps ("elastic apps") can use **whatever throughput is available**.

4. Security

- May require **encryption, integrity checks**, and other security measures.

Transport Service Requirements: Common Applications

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: 5 Kbps – 1 Mbps	Yes, tens of ms
Streaming audio/video	Loss-tolerant	Video: 10 Kbps – 5 Mbps	Yes, few seconds
Interactive games	Loss-tolerant	Elastic	Yes, tens of ms
Text messaging	No loss	Elastic	Yes/No

TCP (Transmission Control Protocol)

- **Reliable transport** between sending and receiving processes.
- **Flow control** → prevents sender from overwhelming receiver.
- **Congestion control** → throttles sender if network is overloaded.
- **Connection-oriented** → setup required between client and server.
- **Does NOT provide**: timing guarantees, minimum throughput, or security.

UDP (User Datagram Protocol)

- **Unreliable transport** between sending and receiving processes.
- **No flow control, congestion control, connection setup, timing, throughput guarantees, or security**.

Q: Why bother using UDP?

- For **time-sensitive apps** where speed is more important than perfect reliability (e.g., real-time audio/video, online gaming).
- Less **overhead** than TCP → lower latency.

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 [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

Problem with Vanilla TCP & UDP

- No encryption by default.
- **Cleartext passwords** and sensitive data are sent directly over the network.
- Data can be intercepted as it **traverses the Internet**.

Transport Layer Security (TLS)

- Provides **encrypted TCP connections**.
- Ensures **data integrity** → detects tampering.
- Provides **end-point authentication** → confirms identity of communicating parties.

How TLS Works

- TLS is implemented at the **application layer**.
- Applications use **TLS libraries**, which in turn use TCP.
- **Cleartext data** from the app is sent to the TLS library → TLS encrypts it → encrypted data traverses the Internet → decrypted at the receiving end.

Web and HTTP

Quick Review

- A **web page** consists of **objects**, which may be stored on **different Web servers**.
- An **object** can be:
  - HTML file
  - JPEG image
  - Java applet

- Audio file
- ...and more

Structure of a Web Page

- A web page has a **base HTML file** that references several **objects**.
- Each object is **addressable by a URL**, for example:

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

- **Host name:** `www.someschool.edu`
- **Path name:** `/someDept/pic.gif`

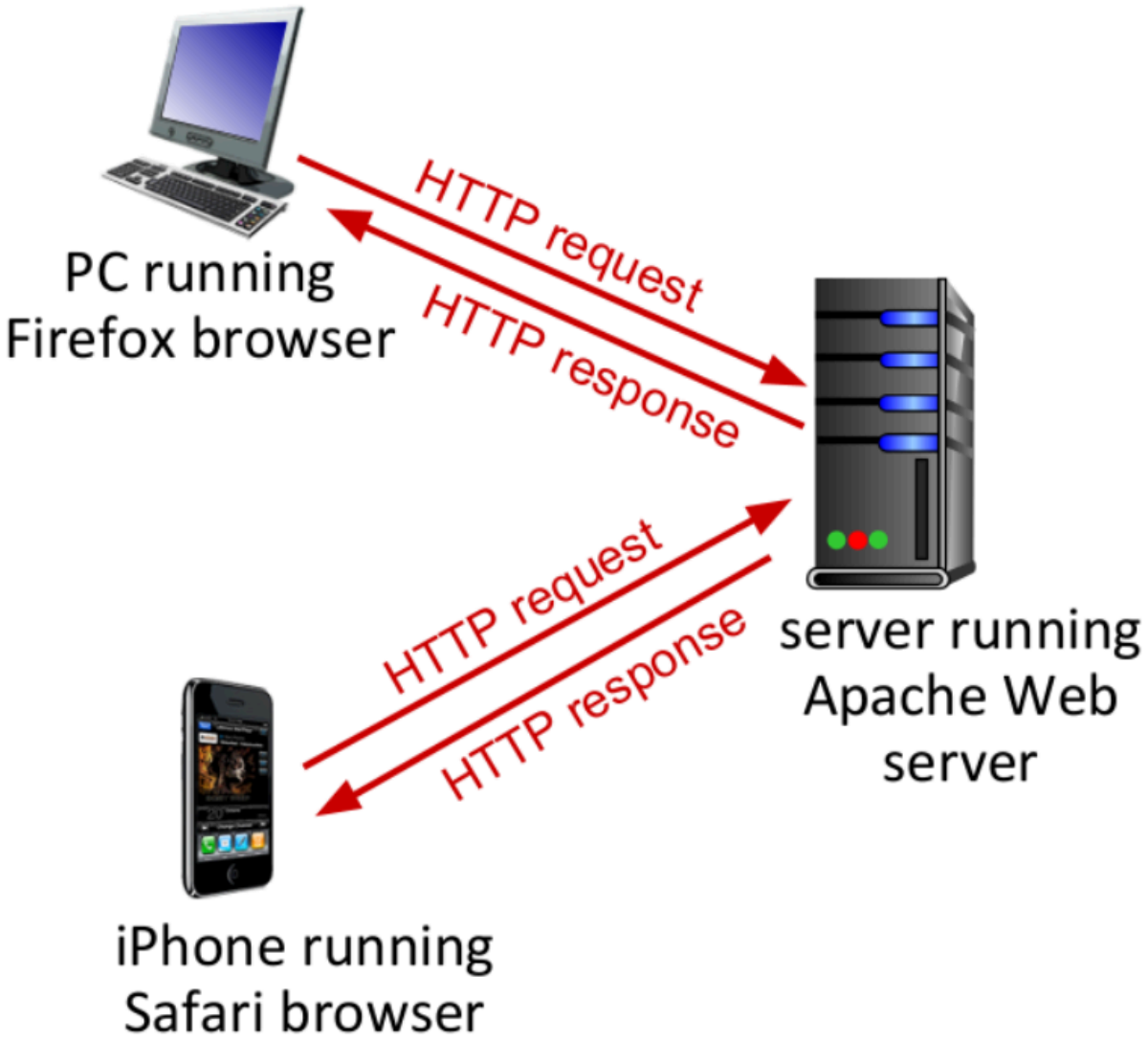
HTTP Overview

Definition

- **HTTP (Hypertext Transfer Protocol)** is the **Web's application-layer protocol**.

Client/Server Model

- **Client:**
  - Typically a **web browser**.
  - Requests web objects using **HTTP**.
  - Receives and **displays** web objects.
- **Server:**
  - **Web server** stores web objects.
  - Sends objects in **response** to client requests using **HTTP**



HTTP and TCP

How HTTP Uses TCP

1. **Client initiates TCP connection**
   - Creates a **socket** to the server on **port 80**.
2. **Server accepts TCP connection**
   - Connection established between **HTTP client** (browser) and **HTTP server**.
3. **HTTP messages exchanged**

- Application-layer messages (requests and responses) are sent over the TCP connection.

4. **TCP connection closed**
    - After the exchange, the connection is terminated.

HTTP is Stateless

- The server **does not maintain information** about past client requests.
- Maintaining **state** is complex because:
    - The server must store **past history** for each client.
    - If either the server or client crashes, their view of the state may become **inconsistent** and requires reconciliation.

HTTP Connections: Two Types

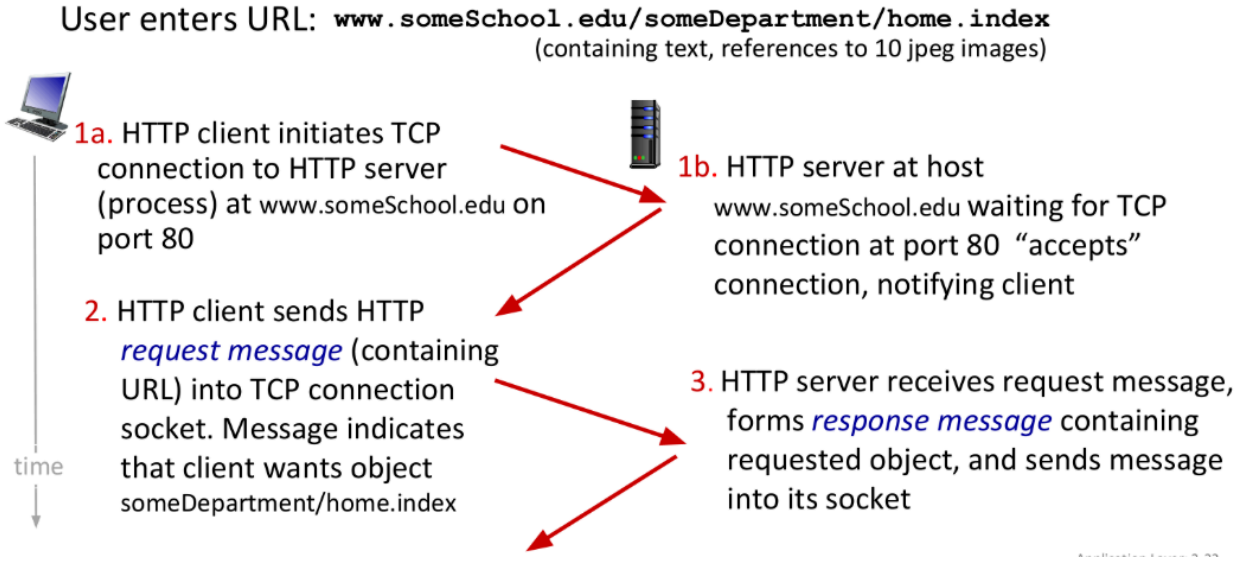
1. Non-Persistent HTTP

- **TCP connection opened** for each object.
- At most **one object sent per TCP connection**.
- **TCP connection closed** after sending the object.
- Downloading multiple objects requires **multiple TCP connections**.

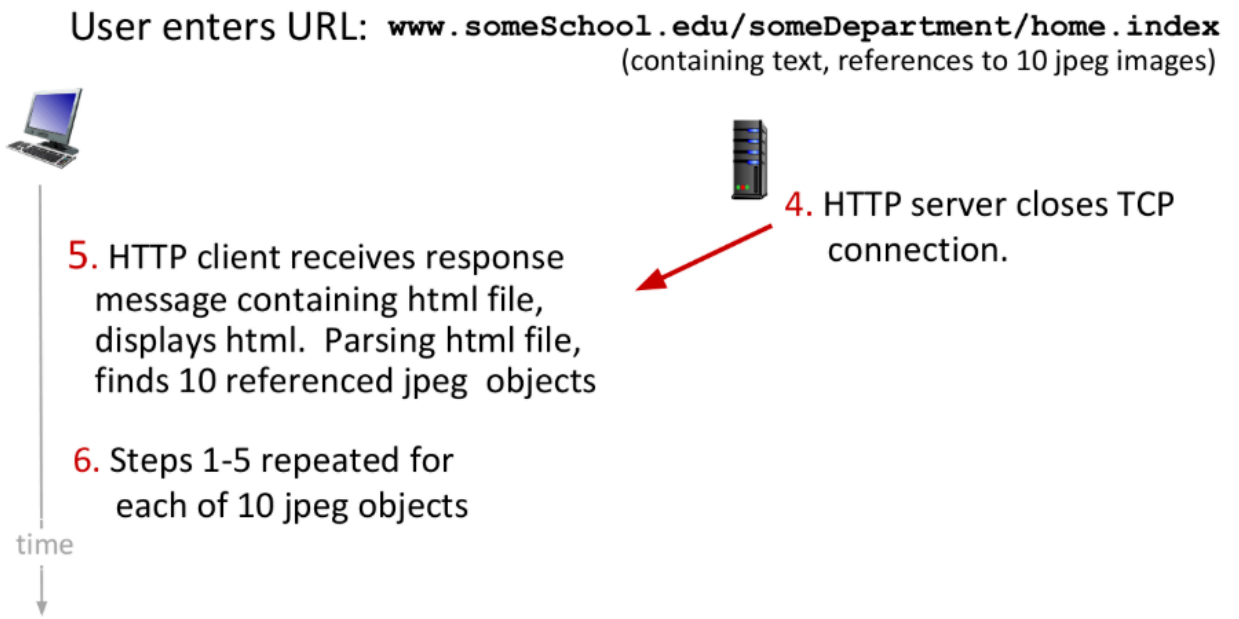
2. Persistent HTTP

- **Single TCP connection** opened to a server.
- **Multiple objects** can be sent over the **same TCP connection**.
- **TCP connection closed** only after all objects are transferred.

Non-persistent HTTP: example



Non-persistent HTTP: example (cont.)



RTT and HTTP Response Time

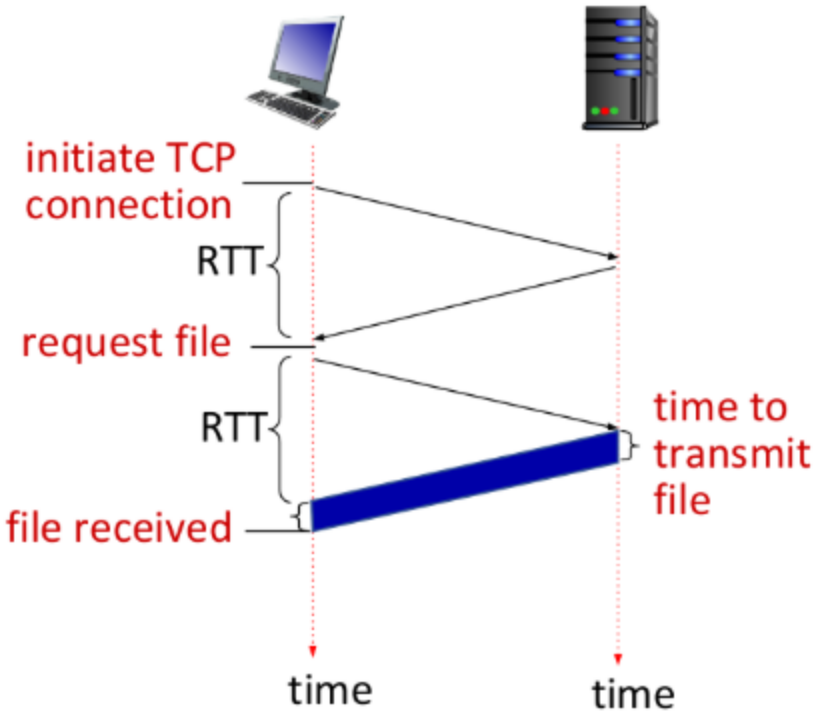
RTT (Round-Trip Time)

- **Definition:** Time for a **small packet** to travel from the **client to the server** and back.

HTTP Response Time (Per Object)

1. **One RTT** to **initiate the TCP connection**.
2. **One RTT** for the **HTTP request** and for the **first few bytes** of the HTTP response to return.
3. **Object/File Transmission Time**
    - Time required to **transfer the full object** after the first bytes arrive.

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



Problems with Non-Persistent HTTP

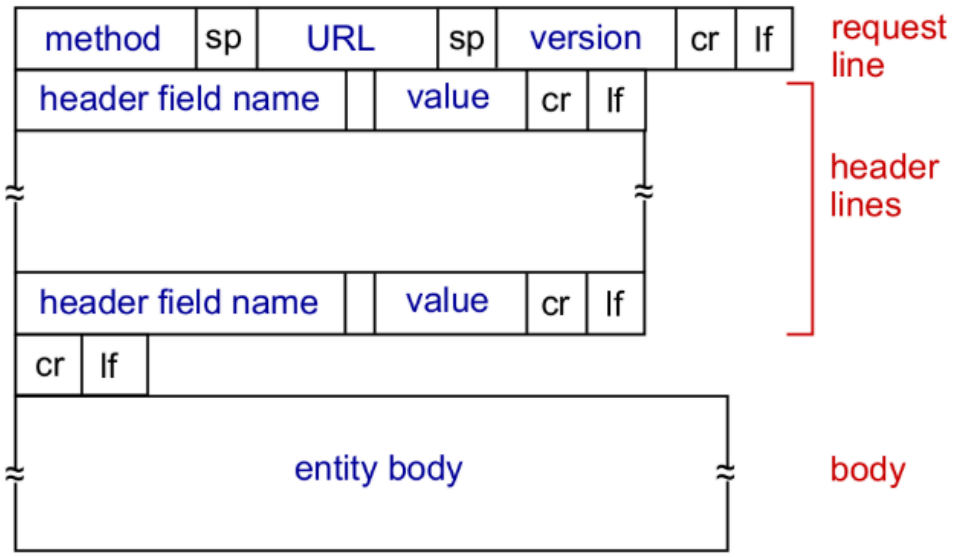
- **2 RTTs per object** required.
- **OS overhead** for opening/closing each TCP connection.

- Browsers often open **multiple parallel TCP connections** to fetch referenced objects simultaneously.

Persistent HTTP (HTTP/1.1)

- Server **leaves the TCP connection open** after sending a response.
- Subsequent **HTTP messages** between the same client and server are sent over the **existing open connection**.
- Client can **send requests immediately** as it encounters referenced objects.
- Response time can be **cut in half**, as **all referenced objects** can be fetched in as little as **one RTT**.

HTTP request message: general format



HTTP Method	Purpose / Description	Data Location	Example Use Case
POST	Send data to the server (e.g., form submission)	Data in **entity body**	Submitting a contact form or login info
HEAD	Retrieve only the headers of a resource	N/A (no body)	Checking if a file exists or metadata before download
GET	Retrieve a resource from the server	Data in **URL** (query string)	Searching: `www.somesite.com/animalsearch?monkeys&banana`
PUT	Upload or replace a resource on the server	Data in **entity body**	Updating a file at a specific URL on the server

Common HTTP Response Status Codes

Status Code	Meaning	Description
200 OK	Success	Request succeeded; requested object included in message body
301 Moved Permanently	Redirection	Requested object moved; new location specified in **Location:** header
400 Bad Request	Client Error	Server could not understand the request message
404 Not Found	Client Error	Requested document not found on the server
505 HTTP Version Not Supported	Server Error	Server does not support the HTTP version used in request

Trying Out HTTP (Client-Side)

Steps

1. **Use `netcat` to connect to a Web server**
    - Opens a **TCP connection** to port **80** (default HTTP port).
    - Example command:

```
nc gaia.cs.umass.edu 80
```

2. **Send an HTTP request manually**
   - Anything typed into `netcat` will be sent to the server.
   - Example of a simple GET request:

```
GET / HTTP/1.1
Host: gaia.cs.umass.edu
```

3. **Observe the response**
   - The **HTTP server sends back a response message**, including:
     - Status line
     - Headers
     - Data (HTML page, images, etc.)
4. **Optional**
   - Use **Wireshark** to **capture and inspect** HTTP request/response packets for deeper understanding

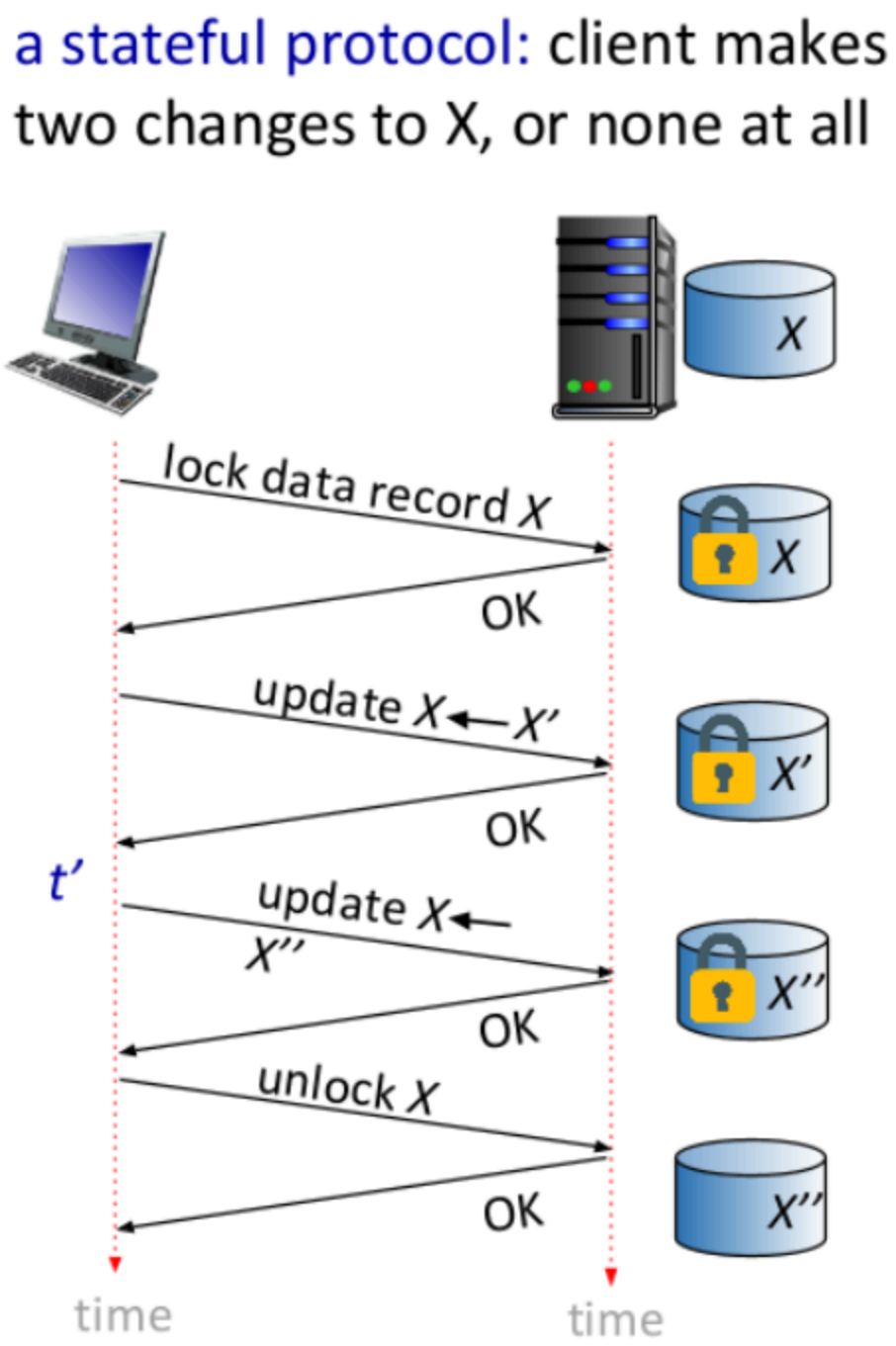
Maintaining User/Server State: Cookies

HTTP is Stateless

- A single **HTTP GET/response interaction** does **not maintain state**.
- **Multi-step exchanges** (e.g., completing a web transaction) are not tracked by HTTP itself.
- **All HTTP requests are independent** of each other.
- No need for the **client or server** to recover from a **partially-completed transaction**.

Implication

- Without extra mechanisms, the server **cannot remember previous interactions** with a client.
- **Cookies** are introduced to maintain state across multiple HTTP requests.



Maintaining User/Server State: Cookies

Purpose

- Websites and client browsers use **cookies** to maintain **state between transactions** in the otherwise stateless HTTP protocol.

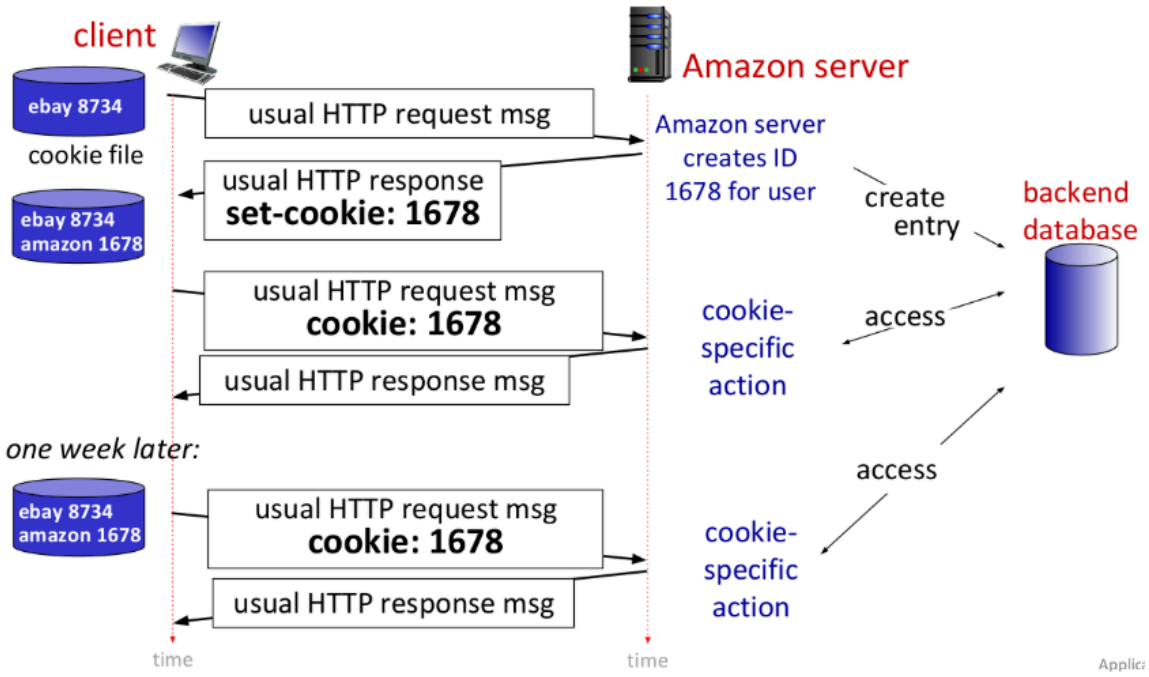
Four Key Components of Cookies

1. **Cookie header line in HTTP response**
    - Sent by the server to the client to create/store a cookie.
2. **Cookie header line in subsequent HTTP requests**
    - Sent by the client back to the server to identify itself.
3. **Cookie file on user's host**
    - Managed by the **user's browser**.
4. **Back-end database at the website**
    - Stores information associated with each **unique cookie ID**.

Example

- **Scenario:** Susan visits an e-commerce site for the first time using her laptop.
- **Steps:**
    1. Initial HTTP request arrives at the site.
    2. Site creates a **unique ID (cookie)**.
    3. An **entry is created in the backend database** for that ID.
    4. Subsequent HTTP requests from Susan include the **cookie ID**, allowing the site to **identify her** and maintain state.

Maintaining user/server state: cookies



HTTP Cookies: Comments

Uses of Cookies

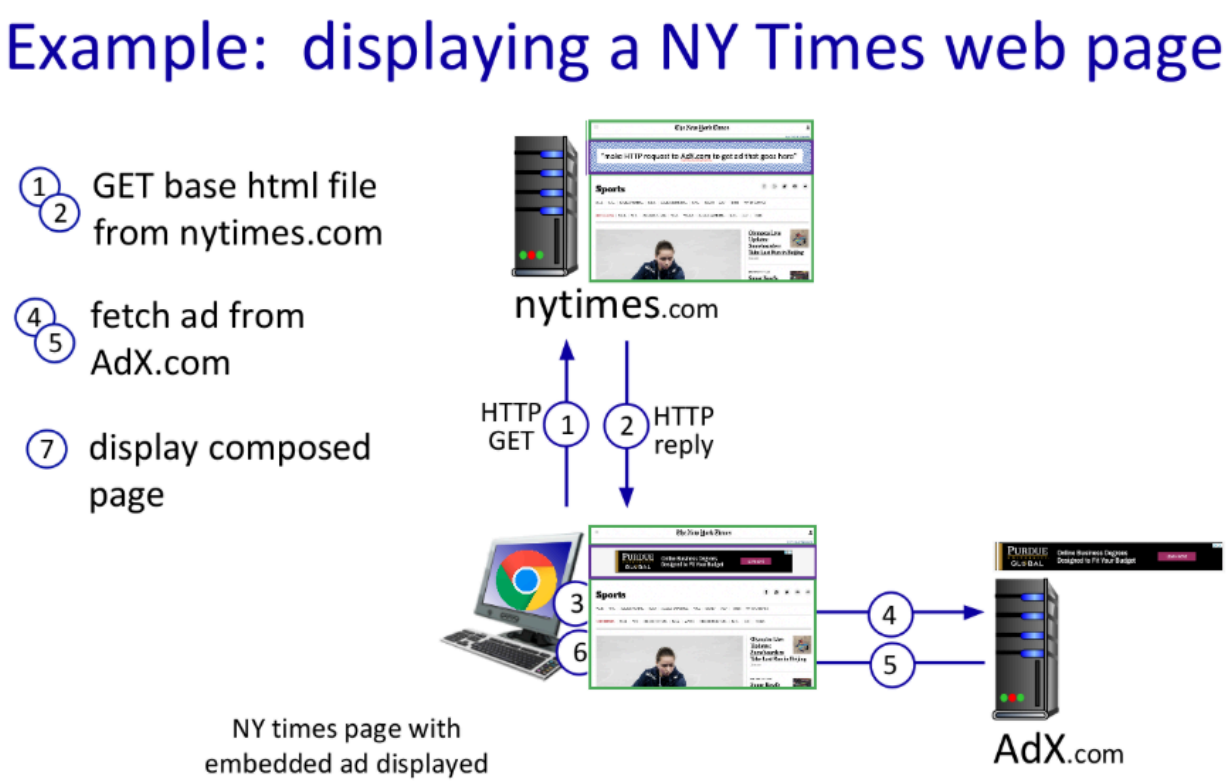
- **Authorization** → track logged-in users
- **Shopping carts** → remember items in cart across pages
- **Recommendations** → personalize content based on user behavior
- **User session state** → e.g., Web e-mail sessions

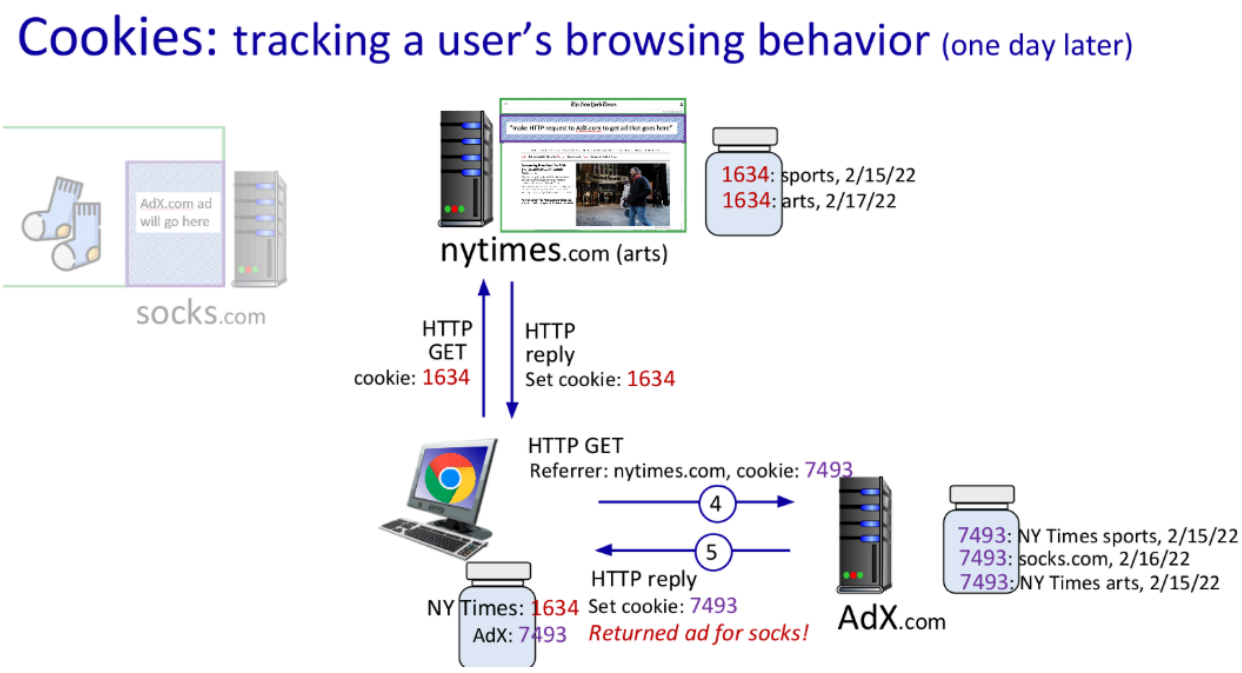
Cookies and Privacy

- Cookies allow websites to **learn a lot about you** on their site.
- **Third-party persistent cookies (tracking cookies)**:
  - Allow a **common identity (cookie value)** to be tracked across **multiple websites**.

Aside: How to Keep State?

1. **At protocol endpoints:**
    - Maintain state at the **sender/receiver** over multiple transactions.
2. **In messages:**
    - Include **cookies in HTTP messages** to carry state between client and server.





Cookies: Tracking a User's Browsing Behavior

Uses of Cookies

1. **First-party cookies**
   - Track user behavior on a **given website**.
2. **Third-party cookies**
   - Track user behavior **across multiple websites**.
   - Can operate **without the user visiting the tracker site**.
   - Tracking may be **invisible** (e.g., via invisible links rather than ads triggering GET requests).

Browser Controls

- **Firefox & Safari:** third-party tracking cookies disabled by default.
- **Chrome (2023):** third-party tracking cookies to be disabled.

GDPR (EU General Data Protection Regulation)

- Cookies can leave traces **identifying natural persons** (IP addresses, cookie IDs, etc.).

- When combined with unique identifiers and other info, **profiles of individuals** can be created.
- Users have **explicit control** over whether cookies are allowed.
- **Cookies identifying individuals** are considered **personal data** under GDPR.

> Reference: GDPR, Recital 30 (May 2018)

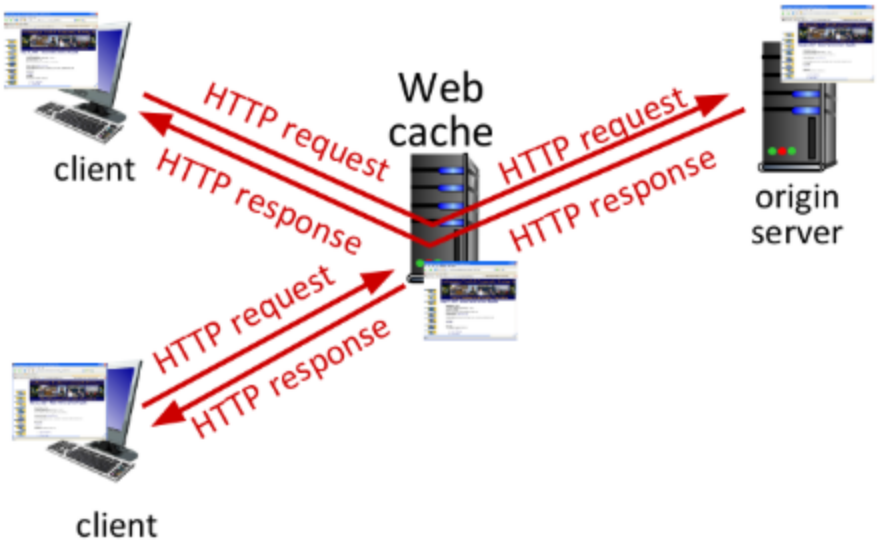
Web Caches

How It Works

1. **User configures browser** → points to a local web cache (proxy server).
2. **Browser sends all HTTP requests** to the cache.
3. Cache checks:
    - **If object is in cache** → return object directly to client.
    - **If object is not in cache** →
        - Cache requests object from **origin server**.
        - Cache stores (caches) the received object.
        - Cache returns object to client.

Goal

- Satisfy client requests **without involving the origin server**, reducing:
    - Network traffic
    - Server load
    - Response time



Web Caches (Proxy Servers)

Role of a Web Cache

- **Acts as both client and server**:
  - **Server** → for the original requesting client (serves cached objects).
  - **Client** → to the origin server (fetches new objects if not cached).

Why Web Caching?

1. **Reduce response time**
   - Cache is physically/virtually **closer to the client**.
2. **Reduce traffic**
   - Less load on an institution's **access link to the Internet**.
3. **Widespread availability**
   - The Internet is **dense with caches**, improving efficiency.
4. **Support smaller content providers**

- Even providers with limited infrastructure can deliver content effectively by relying on caching.

Cache Control

- Origin server tells cache about **caching rules** using the **HTTP response header** (e.g., `Cache-Control`, `Expires`).
- `Cache-Control: max-age=3600`

  → Object can be cached for **3600 seconds (1 hour)**.
- `Expires: Sat, 20 Sep 2025 19:00:00 GMT`

  → Explicit expiry time for cached object.
- `Last-Modified`

  → Helps caches validate if the object is still fresh.

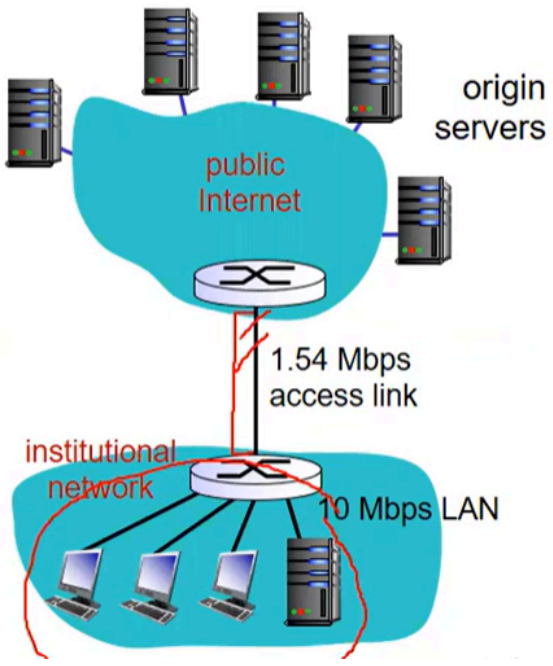
Caching example:

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



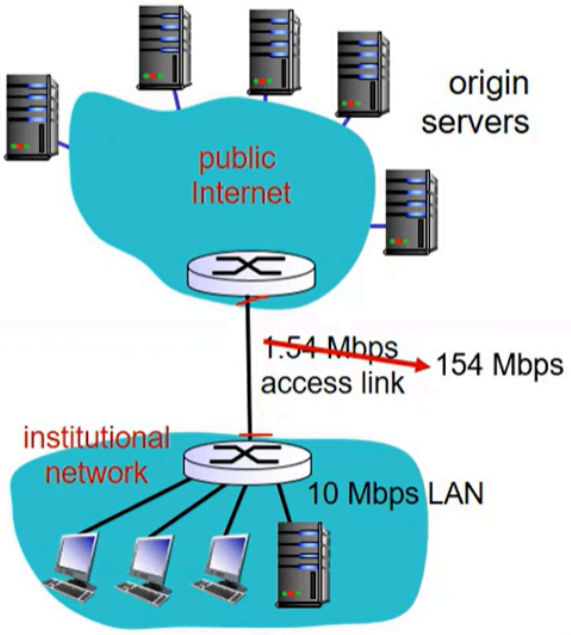
Caching example: fatter access link

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 sec + ~~minutes~~ + usecs
  msecs



Cost: increased access link speed (not cheap!)

Caching example: install local cache

assumptions:

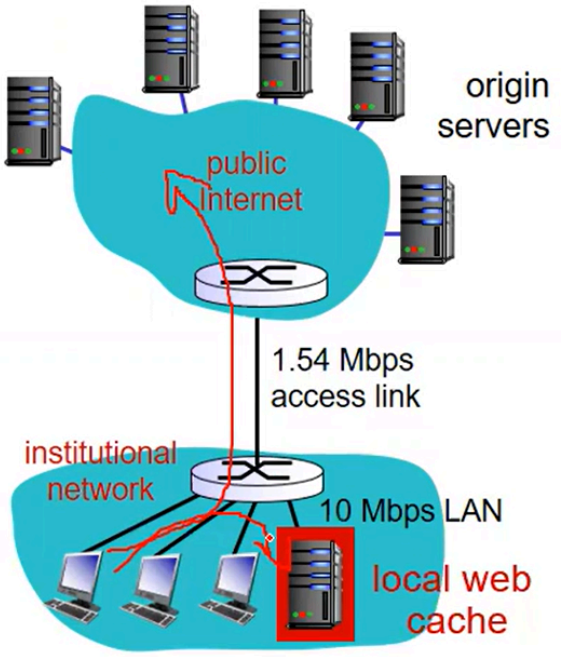
- avg object size: 100K bits
- avg request rate from browsers to origin servers: 15/sec
- avg data rate to browsers: 1.50 Mbps
- RTT from institutional router to any origin server: 2 sec
- access link rate: 1.54 Mbps

consequences:

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

How to compute link utilization, delay?

Cost: web cache (cheap!)



BROWSER CACHING: CONDITIONAL GET

🔷 Goal

- Avoid sending the full object if the browser already has the latest version in its cache.
- Saves **bandwidth**, **network resources**, and eliminates **object transmission delay**.

🔷 Mechanism: Conditional GET

1. **Client request**
   - The browser sends an HTTP request to the server.
   - It includes the timestamp of its cached copy using the header:

```
If-Modified-Since: <date>
```

- `<date>` = the last time the object was retrieved/modified.

2. **Server response (two cases)**

   Case 1: Object not modified since `<date>`

   - Server responds with:

     ```
     HTTP/1.0 304 Not Modified
     ```

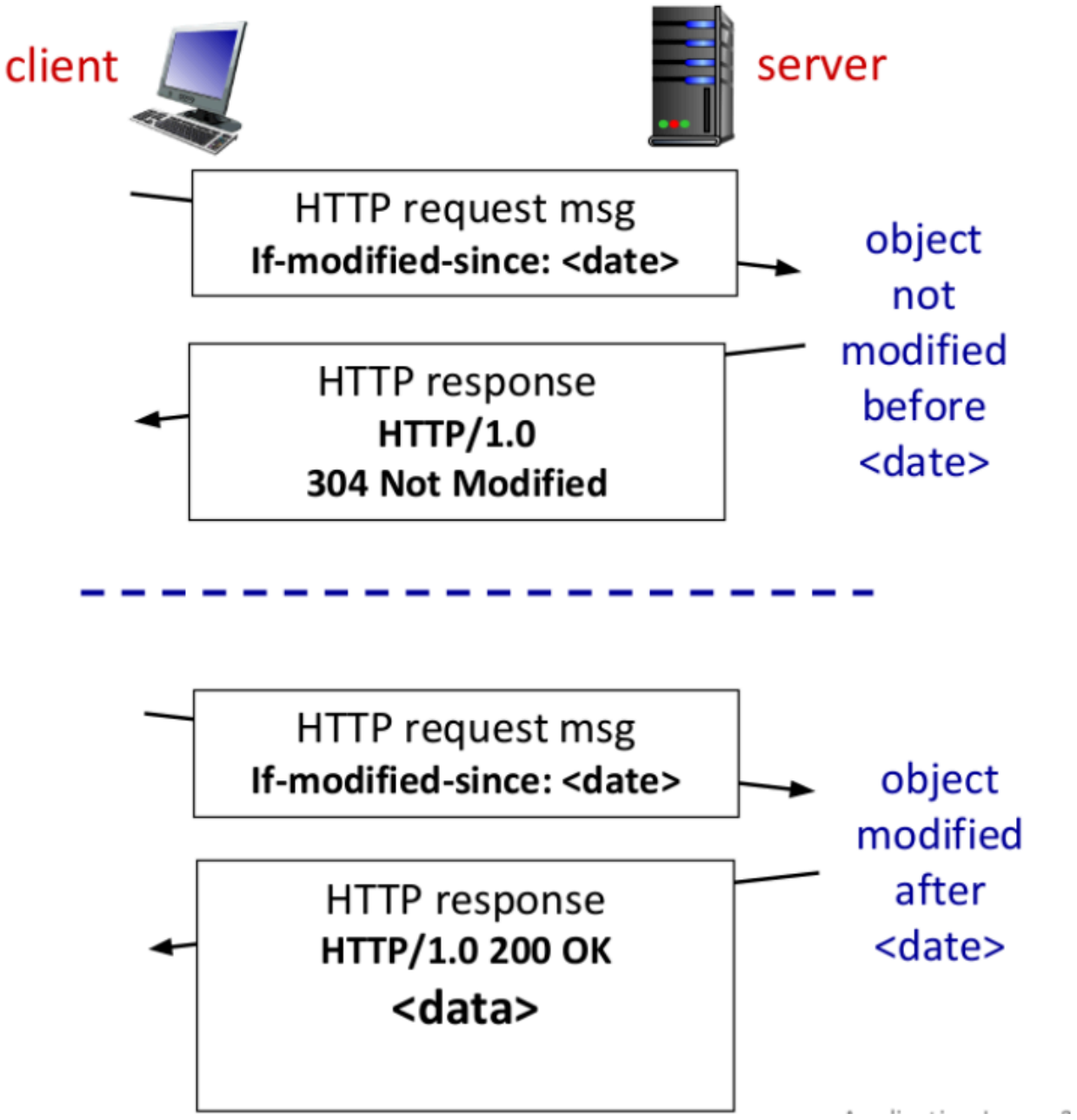
   - No object data is sent, browser continues using cached version.

   Case 2: Object modified after `<date>`

   - Server responds with:

     ```
     HTTP/1.0 200 OK
     <data>
     ```

   - The new object is sent to the browser, which updates its cache.



HTTP/1.1 and its limitations

1. **Pipelining:**
    - HTTP/1.1 allowed multiple GET requests to be sent over a single TCP connection without waiting for previous responses.
    - This is called **pipelining**.

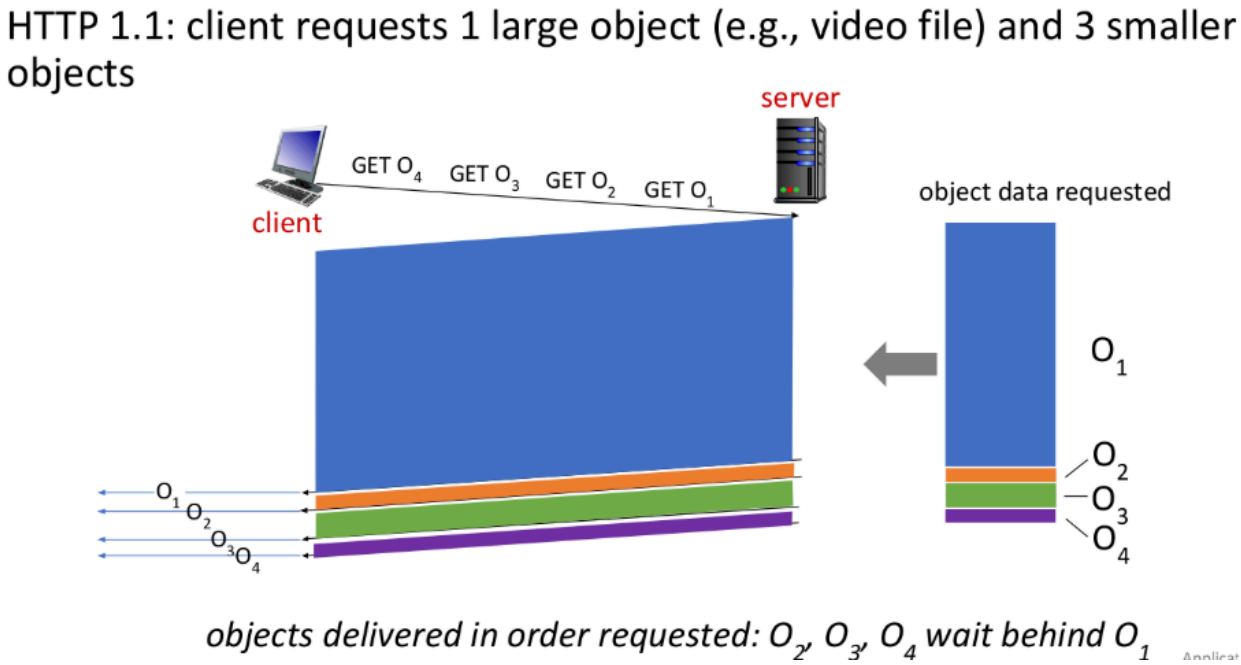
2. **In-order responses:**
    - The server must respond **in the same order** the requests were received (FCFS—First-Come, First-Served).
    - Problem: if a **large object** is first, smaller objects behind it have to **wait**.
    - This is known as **Head-of-Line (HOL) blocking**.
3. **TCP retransmission:**
    - If a TCP segment is lost, the transmission stalls until the lost segment is retransmitted.
    - HOL blocking at TCP level **amplifies delays**, especially for multiple objects.

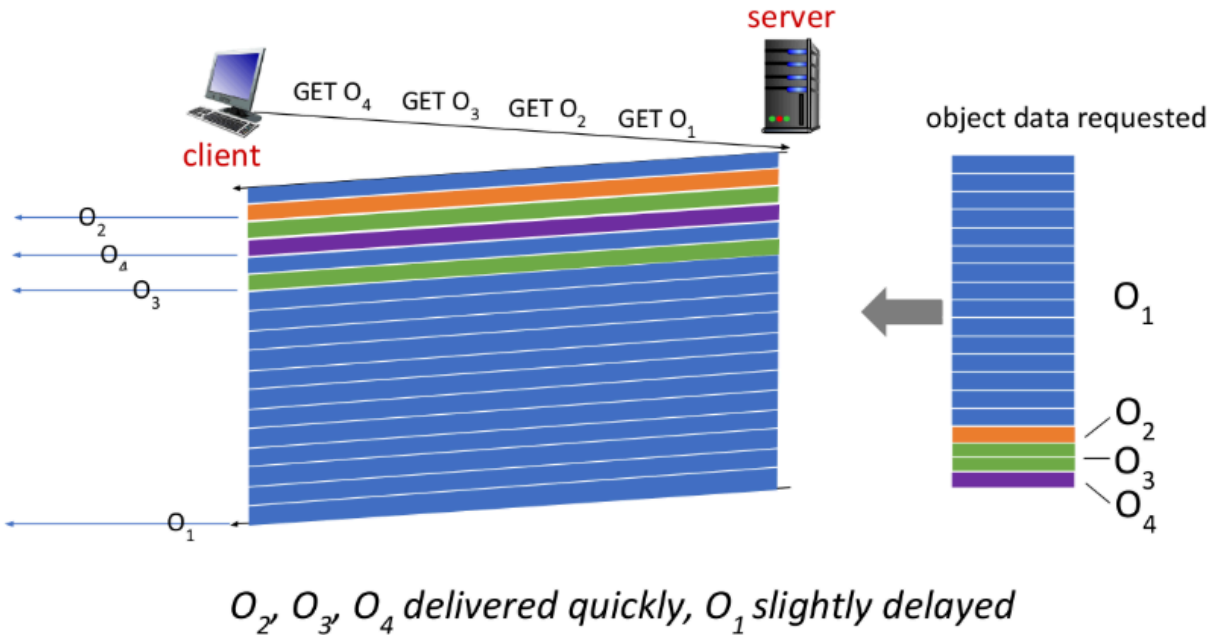
HTTP/2 improvements

1. **Multiplexing:**
    - Multiple HTTP requests and responses can be sent **concurrently** over a single TCP connection.
    - Responses do **not need to be in order**, avoiding HOL blocking at the application level.
2. **Header compression:**
    - HTTP/2 compresses headers to reduce overhead in requests/responses.
3. **Stream prioritization:**
    - Clients can assign **priority** to streams (e.g., small objects can be sent before large ones).
4. **Reduced latency:**
    - By multiplexing streams and allowing out-of-order responses, HTTP/2 **decreases delay** in multi-object requests significantly.

Summary:
HTTP/2 tackles both **HOL blocking at the application layer** and improves

efficiency for multi-object web pages, whereas HTTP/1.1 suffers from delays due to sequential responses and TCP retransmission stalling.





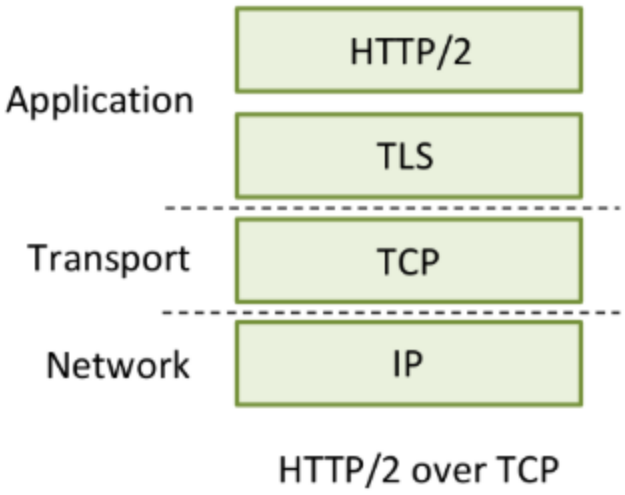
HTTP/2 over TCP

1. **Single TCP connection:**
    - All streams (multiple requests/responses) share the same TCP connection.
    - **Problem:** TCP-level packet loss stalls **all streams** because TCP ensures in-order delivery.
2. **Parallel connections workaround:**
    - Browsers open **multiple TCP connections** to reduce stalling and improve throughput.
    - This is inefficient and adds overhead (more connections to manage).
3. **Security:**
    - HTTP/2 itself doesn't mandate security; it runs over **vanilla TCP** unless combined with TLS (HTTPS).

HTTP/3 over UDP

1. **Protocol base:**
    - Uses **QUIC**, which runs over UDP instead of TCP.
2. **Per-stream reliability:**
    - Each object (stream) has **independent error and congestion control**.
    - A lost packet in one stream **does not stall others**.
3. **Improved pipelining:**
    - Multiplexing is enhanced because streams are truly independent at the transport layer.
4. **Built-in security:**
    - QUIC integrates **TLS 1.3**, so HTTP/3 is secure by default.

QUIC Overview



HTTP/2 over TCP

- **QUIC** = Quick UDP Internet Connections
- Built on **UDP** with TCP-like reliability and congestion control.
- Integrates **security (TLS 1.3)** directly into the transport layer.
- Reduces connection latency and avoids TCP head-of-line blocking.

Connection Establishment: TCP + TLS vs QUIC

Feature	TCP + TLS	QUIC
Transport	TCP handshake	QUIC handshake over UDP
Security	TLS handshake	Integrated TLS 1.3
Number of handshakes	2 (serial)	1 (1-RTT)
Application data	Sent after 2 RTTs	Sent after 1 RTT
Reliability & congestion	TCP	QUIC implements both
Encryption & authentication	TLS layer	Built-in QUIC layer

Flow Comparison:

TCP + TLS:

1. TCP handshake → establish reliability & congestion control.
2. TLS handshake → establish authentication & crypto state.

3. Send application data.

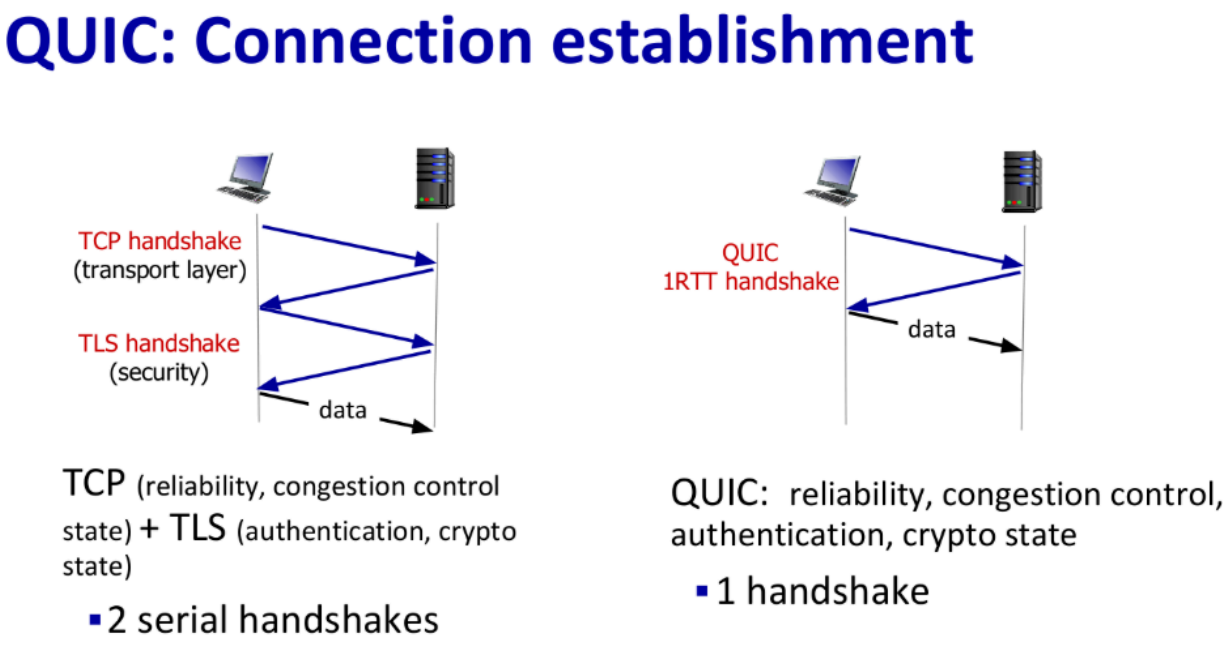
QUIC (1-RTT):

1. Single handshake → reliability, congestion control, authentication, crypto state all established.
2. Send application data immediately.

QUIC 0-RTT (Session Resumption)

- **Purpose:** Reduce handshake latency for repeat connections.
- **Mechanism:**
    1. Server issues **TLS session ticket** during first connection.
    2. Client caches the ticket.
    3. For a subsequent connection, the client sends ticket to server.
    4. Authentication & encryption established **without waiting** (0 RTT).
    5. Application data can be sent immediately.

Benefit: Zero handshake delay for repeated connections, improving performance for frequent clients.



TCP VS UDP

Feature	TCP	UDP
Connection	Oriented	Connectionless
Reliability	Reliable	Unreliable
Order	In-order delivery	No order guarantee
Flow/Congestion Control	Yes	No
Overhead	High	Low
Speed	Slower	Faster
Use Cases	Web, Email, FTP	Streaming, Gaming, DNS

SMTP & Mail Servers – Short Notes

1. Mail Servers

- **Mailbox:** stores incoming messages for users.
- **Message queue:** stores outgoing messages waiting to be sent.
- **SMTP:** protocol used between mail servers to send emails.
- **Roles:**
  - **Client:** sending mail server
  - **Server:** receiving mail server

2. SMTP Overview

- Defined in **RFC 5321**.
- Uses **TCP** (port 25) to reliably transfer email.
- Direct transfer: sending server → receiving server.
- Operates in **three phases**:
  1. **SMTP handshaking** (greeting)

2. **Message transfer**
3. **Connection closure**

- **Command/response interaction:**
  - Commands: ASCII text
  - Response: status code + phrase

---

3. SMTP Interaction Example

Scenario: Alice sends email to Bob

Steps:

1. Alice composes email using her **User Agent (UA)**.
2. UA sends message to Alice's **mail server** via SMTP → placed in message queue.
3. SMTP client at Alice's server opens TCP connection with Bob's mail server.
4. SMTP client sends message over TCP.
5. Bob's server stores message in **Bob's mailbox**.
6. Bob reads message via his **User Agent**.

Sample SMTP Commands:

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr
C: MAIL FROM: <alice@crepes.fr>
S: 250 Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 Recipient ok
C: DATA
S: 354 Enter mail, end with "."
C: Message body
C: .
S: 250 Message accepted
```

```
C: QUIT
S: 221 Closing connection
```

4. Observations

- SMTP uses **persistent connections**.
- Messages must be in **7-bit ASCII**.
- **End of message:** CRLF.CRLF
- Comparison with HTTP:
  - HTTP = **client pull**, SMTP = **client push**
  - Both: ASCII command/response, status codes
  - HTTP: each object in separate response
  - SMTP: multiple objects in multipart message

5. Mail Message Format

- Defined in **RFC 2822**.
- **Structure:**

```
Header
Blank line
Body
```

- **Header lines:** To, From, Subject (different from SMTP MAIL FROM/RCPT TO commands)
- **Body:** message content in ASCII

6. Retrieving Email

- **SMTP:** for delivery/storage to receiver's server.
- **Mail Access Protocols:** for retrieving from server, e.g., IMAP, POP3, HTTP.

- **IMAP (RFC 3501):**
  - Messages remain on server
  - Provides retrieval, deletion, folder management
- **Webmail:** Gmail, Yahoo, Hotmail use:
  - SMTP → sending
  - IMAP/POP → retrieving

PAST PAPERS

Q1. What are the advantages and disadvantages of Persistent HTTP?

Advantages:

1. Multiple objects can be fetched within a single connection, reducing overhead of repeated connection establishments.
2. Minimizes TCP handshakes and reduces network congestion, improving overall network efficiency.

Disadvantages:

1. If many objects (e.g., images) are fetched in one connection, it may cause **resource blocking** (subsequent requests wait until the connection is free).
2. Connections may remain open longer, consuming more server resources and increasing server load, especially with many concurrent users.

Q2. What are the advantages and disadvantages of Non-Persistent HTTP?

Advantages:

1. Each resource uses a separate connection, avoiding resource blocking.
2. Conserves server resources by closing connections after each request, preventing idle open connections.

Disadvantages:

1. Higher overhead due to repeated connection establishments.
2. Each resource requires multiple RTTs (connection setup + data transfer), leading to higher latency and slower performance.

Q3. Compare the number of RTTs required in Persistent vs Non-Persistent HTTP for 20 objects.

- **Non-Persistent HTTP**: $20 \times 2 = 40$ RTTs

  Each object takes **2 RTTs** (1 for connection setup, 1 for data transfer).

- **Persistent HTTP**: $1 + 20 = 21$ RTTs

  1 RTT for connection setup + 1 RTT for each of the 20 objects.

  $1 + 20 = 21$ RTTs

✅ **Persistent HTTP is much faster (21 RTTs vs 40 RTTs).**

Q4. How does Peer-to-Peer (P2P) improve scalability?

- In P2P, each user acts as both **client and server**, sharing content.
- This reduces server load and increases scalability.
- Downloads are faster since multiple peers can serve the same object simultaneously.

Q5. How does HTTP/2 improve over HTTP/1.1 regarding Head-of-Line (HOL) blocking?

- **HTTP/1.1**:
  - Uses **FCFS transmission**.
  - Small objects get delayed behind large ones.
  - Packet loss stalls transmission due to TCP retransmissions.
- **HTTP/2**:
  - Enables **multiplexing**: objects are split into frames and sent in parallel.
  - Allows **prioritization** of important/smaller objects.
  - Minimizes HOL blocking and reduces impact of TCP loss recovery.

Q6. Explain how data is encapsulated through different layers.

- **Application Layer**: Adds application-specific headers to the message.
- **Transport Layer**: Creates a **segment** by adding headers like source/destination port numbers, sequence number, checksum, etc.
- **Network Layer**: Creates an **IP datagram** by adding source and destination IP addresses along with routing information.
- **Link Layer**: Encapsulates each IP packet into a **frame** with headers such as source and destination **MAC addresses**.
- **Physical Layer**: Converts the frame into a stream of **bits** suitable for transmission over the physical medium.

Q7. Explain conditional GET in HTTP.

1. Client sends a **GET request** with a conditional header (e.g., `If-Modified-Since`).
2. If the resource is **unchanged**, the server responds with:
    - **304 Not Modified** (no resource body sent).
    - Reduces unnecessary data transfer and network traffic.
3. If the resource **has changed**, the server responds with:
    - **200 OK** and the updated resource.

Question 1: Consider the scenario shown below, with four different servers connected to four different clients over four three-hop paths. The four pairs share a common middle hop with a transmission capacity of R = 100 Mbps. The four links from the servers to the shared link have
a transmission capacity of RS = 20 Mbps. Each of the four links from the shared middle link to a client has a transmission capacity of RC = 90 Mbps per second. You might want to review Figure 1.20 in the text before answering the following questions:
a) What is the maximum achievable end-end throughput (in Mbps) for each of four client-to-server pairs, assuming that the middle link is fair-shared (i.e., divides its transmission rate
equally among the four pairs)?
b) Which link is the bottleneck link for each session?
c) Assuming that the senders are sending at the maximum rate possible, what are the link utilizations for the sender links (RS), client links (RC), and the middle link (R)?

(a) Maximum achievable throughput per pair

- **Server links (RS)**: Each server can send at **20 Mbps** max.
- **Shared middle link (R = 100 Mbps)**:
    - Shared fairly among **4 pairs**.
    - Each pair gets 100/4=25Mbps.
- **Client links (RC)**: Each client can receive up to **90 Mbps**, which is not limiting here.

So, **throughput per pair = min(RS, R/4, RC)**

=min(20,25,90)=20Mbps

Each client–server pair gets **20 Mbps**.

(b) Bottleneck link for each session

- Server link RS=20 < middle link share (25 Mbps) < client link (90 Mbps).

  The **bottleneck is the server link (RS)** for all 4 sessions.

(c) Link utilizations

Let's calculate utilization = (actual throughput) ÷ (capacity).

- **Sender links (RS = 20 Mbps)**:
  - Each carries **20 Mbps**, capacity = 20 Mbps.
  - Utilization = 20/20=100%.
- **Middle link (R = 100 Mbps)**:
  - Total traffic = 4×20=80 Mbps.
  - Utilization = 80/100=80%.
- **Client links (RC = 90 Mbps)**:
  - Each carries **20 Mbps**, capacity = 90 Mbps.
  - Utilization = 20/90≈22.2%.

Packet Switching vs Circuit Switching

1. Circuit Switching

- A **dedicated communication path** (circuit) is established between sender and receiver before data transfer begins.
- Resources (bandwidth, buffers, etc.) are **reserved** for the entire duration of the session.
- Example: Traditional telephone networks.

Advantages:

- Fixed, predictable performance (guaranteed bandwidth, delay).
- No congestion once circuit is established.
- Simple data transfer (no need for headers per packet).

Disadvantages:

- Resource wastage if the link is idle (reserved but unused).
- Call setup time (latency before communication begins).
- Poor scalability with many users.

2. Packet Switching

- Data is split into **packets** (with headers), sent individually through the network.
- Each packet may take a **different path** depending on availability and routing.
- Resources are **shared dynamically** among users.
- Example: The Internet.

Advantages:

- Efficient resource utilization (links shared among multiple users).
- High scalability.
- Robustness (packets can reroute if a link fails).

Disadvantages:

- Packets may face delay, jitter, or loss due to congestion.
- No guaranteed bandwidth or fixed delay.
- Requires more complex protocols (e.g., TCP/IP).

Protocols & Concepts

1. TCP (Transmission Control Protocol)

- **Layer:** Transport Layer (OSI Layer 4).

- Provides **reliable, connection-oriented** communication with acknowledgments, sequencing, and error recovery.
- Ensures data arrives **in order, without loss or duplication**.
- Used in applications like **web (HTTP/HTTPS), email, FTP**.

2. UDP (User Datagram Protocol)

- **Layer:** Transport Layer.
- Provides **connectionless, best-effort** delivery without acknowledgments or ordering.
- Much **faster** than TCP but less reliable.
- Commonly used in **DNS, VoIP, video streaming, gaming**.

3. Cookies

- **Layer:** Application Layer (works with HTTP/HTTPS).
- Small pieces of data stored in the browser by servers.
- Help maintain **state** (e.g., user logged in, cart items in shopping).
- Used for **sessions, personalization, and tracking** across sites.

4. HTTP (HyperText Transfer Protocol)

- **Layer:** Application Layer.
- Protocol for **fetching web pages, images, and files** from web servers to browsers.
- Uses **TCP port 80** (unsecured) or **443 with TLS (HTTPS)** for secure communication.
- **Stateless**, so relies on cookies/sessions for continuity.

5. IMAP (Internet Message Access Protocol)

- **Layer:** Application Layer.
- Used by email clients to **read, organize, and manage emails** directly from the server.

- Emails remain on the server (supports access from multiple devices).
- Works on **TCP port 143 (or 993 with SSL/TLS)**.

6. SMTP (Simple Mail Transfer Protocol)

- **Layer:** Application Layer.
- Standard protocol for **sending emails** between clients and servers or between mail servers.
- Relies on TCP for reliable delivery.
- Runs on **TCP port 25 (server-to-server), 587 (submission)**.

7. Routing Protocols

- **Layer:** Network Layer (Layer 3).
- Enable routers to **exchange routing information** and determine the best path for packets.
- Two types: **Interior Gateway Protocols (RIP, OSPF, EIGRP)** and **Exterior Gateway Protocols (BGP)**.
- Ensure data takes the **fastest, most efficient route** across networks.

Quick memory tip:

- **TCP/UDP → Transport** (reliable vs fast).
- **HTTP, IMAP, SMTP, Cookies → Application** (web + email).
- **Routing → Network Layer** (finds best path).

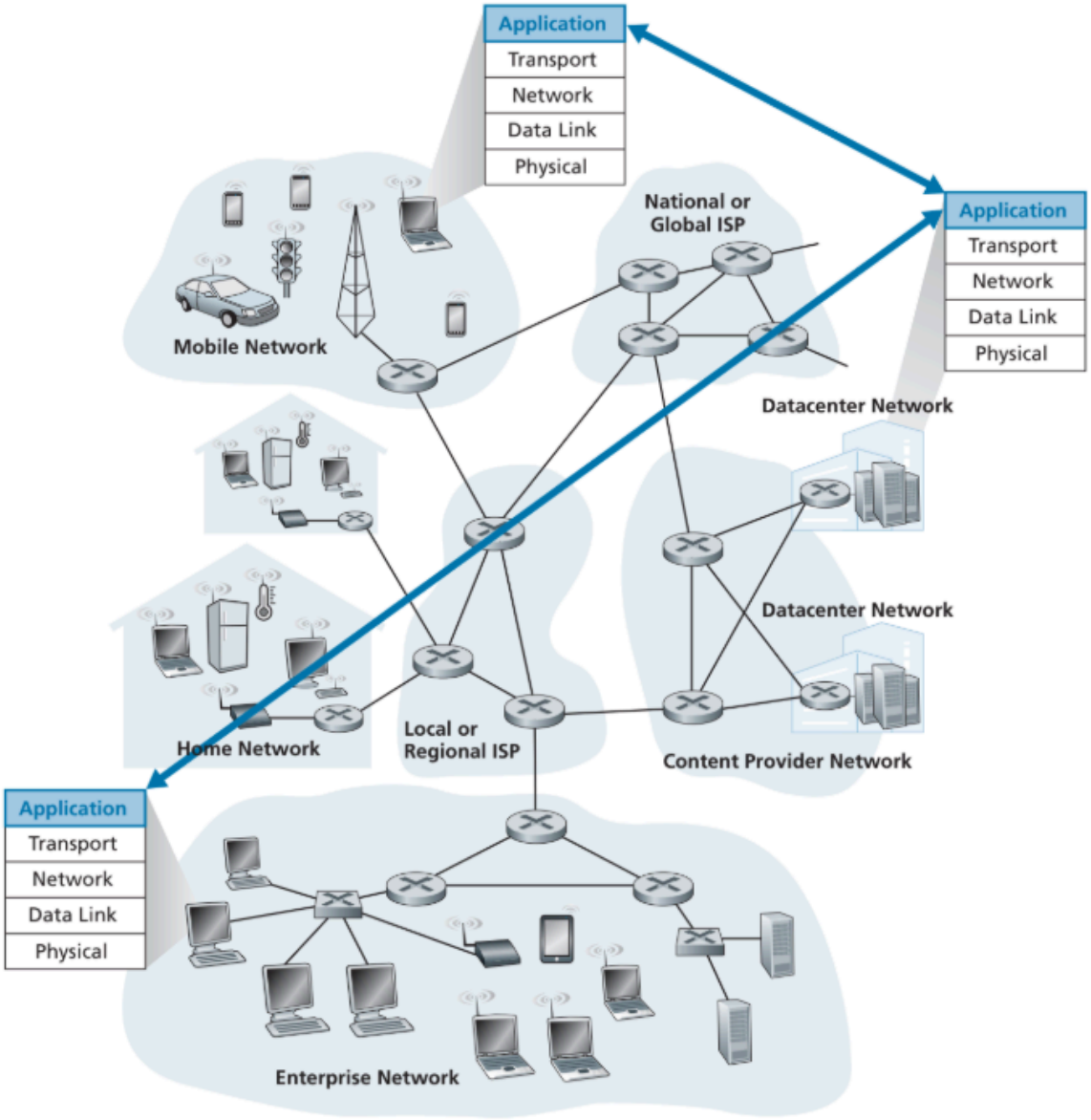


Figure 2.1 ♦ Communication for a network application takes place between end systems at the application layer

QS

1. Packet Switching Delay Components

In packet switching (used by the Internet), a packet goes through several types of delay:

1. **Processing Delay (dproc)**
   - Time for a router to process the packet header (error checking, forwarding decision).
2. **Queuing Delay (dqueue)**
   - Time spent waiting in the router's queue before transmission.
   - **Variable**, depends on congestion.
3. **Transmission Delay (dtrans)**
   - Time to push all bits of the packet onto the link.

   dtrans = L/R

   where L = packet length (bits), R = link bandwidth (bps).
4. **Propagation Delay (dprop)**
   - Time for the bits to travel through the physical medium.
   - dprop = d/s

✅ **Total End-to-End Delay (Packet Switching):**

dtotal=dproc+dqueue+dtrans+dprop

---

2. Circuit Switching Delay Components

In circuit switching (used in traditional telephone networks), a dedicated path is set up **before communication starts**.

1. **Circuit Setup Delay (dsetup)**
   - Time to establish a dedicated circuit from source to destination before data can flow.
2. **Propagation Delay (dprop)**
   - Time for bits to travel along the circuit's links.
3. **Transmission Delay (dtrans)**

- Time to push data bits onto the circuit (like packet switching).
- Often less important if bandwidth is guaranteed.

✅ **Total End-to-End Delay (Circuit Switching):**

$d_{total} = d_{setup} + d_{prop} + d_{trans}$

Key Differences

Aspect	Packet Switching	Circuit Switching
Path setup	None (send immediately)	Required before communication (setup delay)
Main delays	Processing, Queuing, Transmission, Propagation	Setup, Transmission, Propagation
Queuing delay	Yes (variable, depends on congestion)	No (resources are reserved)
Efficiency	High utilization but possible congestion	Guaranteed bandwidth but less efficient
Example	Internet data (IP)	Traditional telephone network

In short:

- **Packet switching:** Delay comes from **processing, queuing, transmission, propagation**.
- **Circuit switching:** Delay comes from **setup, transmission, propagation** (no queuing).

R1. What is the difference between a host and an end system? List several different types of end systems. Is a Web server an end system?

- **Host vs. End System**:

  In Internet terminology, *host* and *end system* mean the same thing.

  Both refer to any device connected to the Internet that can send or receive data and is either the **source** or **destination** of communication.

- **Types of End Systems**:

  End systems include a wide variety of devices that can run applications:

  - **Personal devices**: Desktop PCs, laptops, smartphones, tablets
  - **IoT devices**: Smart TVs, home assistants (like Alexa), smart appliances, security cameras
  - **Servers**: Web servers, email servers, game servers, cloud servers
  - **Other**: Workstations, thin clients, etc.

- **Is a Web Server an End System?**

  ✔️ Yes. A **Web server** is a classic example of an end system because it receives HTTP requests from clients (e.g., browsers) and sends back HTTP responses (web pages, data).

👉 **In short:**

- **Host = End System**
- Examples: PCs, smartphones, IoT, servers
- **Yes, a Web server is an end system.**

R2. The word protocol is often used to describe diplomatic relations. How does Wikipedia describe diplomatic protocol?

- **Diplomatic Protocol (from Wikipedia):**

  Diplomatic protocol is *a set of rules, procedures, and conventions that govern the official interactions between representatives of different states*.

  It covers aspects such as ceremonies, forms of address, and the order of precedence.

- **Connection to Networking:**

  Just like diplomats need shared rules to communicate smoothly, computers also need **protocols**.

  A **network protocol** (e.g., HTTP, TCP, DNS) defines:

- **Syntax** → how messages are structured (format).
- **Semantics** → what the messages mean.
- **Timing** → when and in what order the messages should be sent.

R3. Why are standards important for protocols?

Standards are **essential** in networking because they ensure that devices and software built by different vendors can work together. Without standards, the Internet would not function as a single global system.

Reasons why standards are important:

1. **Interoperability**
   - Devices and applications from different manufacturers can communicate seamlessly.
   - Example: An Apple laptop can connect to a Wi-Fi router from TP-Link because both follow IEEE 802.11 Wi-Fi standards.
2. **Innovation and Growth**
   - Standards create a common platform.
   - Developers don't need to reinvent the wheel for every new product—they can build on top of existing protocols (like TCP/IP, HTTP).
   - This accelerates new applications (e.g., streaming, social media).
3. **Market Expansion and Competition**
   - Standards prevent vendor lock-in.
   - New companies can enter the market if their devices follow agreed protocols.
   - Example: Any phone manufacturer can join the Internet ecosystem as long as it supports TCP/IP.
4. **Global Communication**
   - Standards act as a *common language* for the Internet.

- Just like English can serve as a lingua franca among people, TCP/IP, DNS, and HTTP serve as universal languages for computers.

R4. List four access technologies. Classify each one as home access, enterprise access, or wide-area wireless access.

1. **DSL (Digital Subscriber Line)**
   - Uses existing telephone lines.
   - **Classification:** Home access.
2. **Ethernet (wired LAN)**
   - Uses twisted-pair copper cables or fiber within local area networks.
   - **Classification:** Enterprise access (common in offices, campuses).
3. **Cellular Networks (3G, 4G, 5G)**
   - Uses radio spectrum to provide Internet access over a wide geographic area.
   - **Classification:** Wide-area wireless access.
4. **Satellite Internet**
   - Uses communication satellites to reach remote or rural areas.
   - **Classification:** Home access (but can also support mobile users, e.g., ships, airplanes).

R5. Is HFC transmission rate dedicated or shared among users? Are collisions possible in a downstream HFC channel? Why or why not?

1. **Transmission Rate (Shared vs. Dedicated):**
   - In **Hybrid Fiber Coax (HFC)**, the available bandwidth is **shared among all users** in the same neighborhood.

- This means if many people are streaming or downloading at the same time, each user's effective rate can decrease.
- Unlike DSL, which provides a more **dedicated link** per subscriber, HFC is a **shared medium**.

2. **Collisions in the Downstream Channel:**
   - The **downstream channel** (from the ISP's CMTS → to homes) is a **broadcast channel**.
   - Only the **CMTS transmits**, and all homes only receive (listen).
   - Since there is only **one sender**, **collisions are not possible** in the downstream direction.
   - Collisions occur only when **multiple devices try to transmit at the same time on the same channel**, which does not happen in downstream HFC.

R9. HFC, DSL, and FTTH for Residential Access

Technology	Typical Downstream Rate	Typical Upstream Rate	Shared or Dedicated?
HFC (Hybrid Fiber-Coaxial, Cable Internet)	10 Mbps – 1 Gbps (sometimes higher in DOCSIS 3.1/4.0)	5 – 50 Mbps (up to 100+ Mbps with newer standards)	**Shared** among users in a neighborhood (bandwidth can fluctuate at peak times).
DSL (Digital Subscriber Line, e.g., ADSL/VDSL)	1 – 100 Mbps (distance dependent; closer to central office = higher rate)	128 Kbps – 30 Mbps (often lower than downstream)	**Dedicated** point-to-point copper line from each home to the telco central office (not shared with neighbors).
FTTH (Fiber-to-the-Home)	100 Mbps – 1 Gbps (can go to 2–10 Gbps with modern deployments)	Often **symmetric** (same as downstream, e.g., 100 Mbps up / 100 Mbps down)	Often **effectively dedicated** (or shared among very few users in passive optical networks), usually provides very consistent performance.

Store and forward packet switching:

1. **Transmission delay for one link**

   dtrans=L/R

2. In **store-and-forward switching**, the packet must be **completely received at each intermediate router** before being forwarded to the next link.
3. So, for N links, the packet is transmitted **N times** (once per link).
4. Therefore, the **total end-to-end delay** (ignoring propagation, queuing, processing) is:dend−to−end=N·RL

- L: Packet length (bits)
- R: Transmission rate of each link (bps)
- N: Number of links

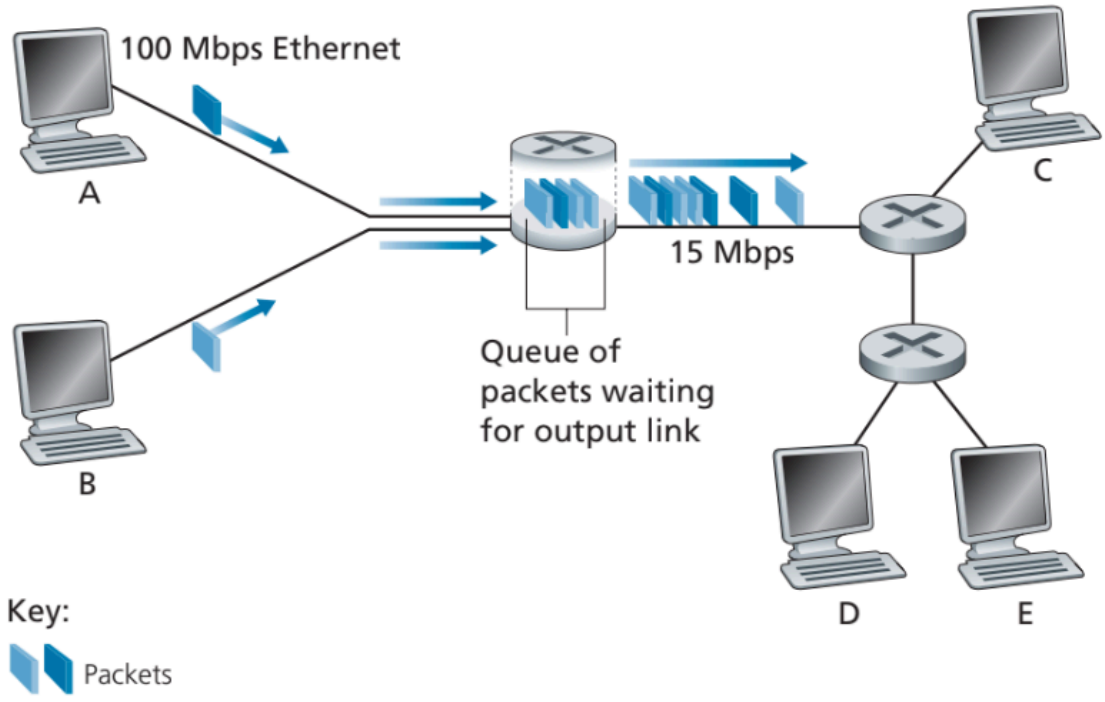


Figure 1.12 ♦ Packet switching

R22. List five tasks that a layer can perform

Five common tasks that can be performed by different layers in a network protocol stack are:

1. **Error Control** – Detecting and/or correcting errors in transmitted data (e.g., checksums in transport and link layers).
2. **Flow Control** – Preventing a fast sender from overwhelming a slow receiver (e.g., TCP's sliding window).
3. **Segmentation and Reassembly** – Breaking large messages into smaller units for transmission and reassembling them at the destination (done at the transport layer).
4. **Multiplexing/Demultiplexing** – Allowing multiple applications or connections to share the same communication channel (e.g., port numbers in TCP/UDP).
5. **Connection Setup and Teardown** – Establishing, maintaining, and closing a logical connection between two end systems (e.g., TCP three-way handshake).

P3. Consider an application that transmits data at a steady rate (for example, the sender generates an N-bit unit of data every k time units, where k is small and fixed). Also, when such an application starts, it will continue running for a relatively long period of time. Answer the following questions, briefly justifying your answer:
a. Would a packet-switched network or a circuit-switched network be more appropriate for this application? Why?
b. Suppose that a packet-switched network is used and the only traffic in this network comes from such applications as described above. Furthermore, assume that the sum of the application data rates is less than the capacities of each and every link. Is some form of congestion control needed? Why?

(a) circuit-switched.

Why:

- Circuit switching **reserves capacity** along the entire path for the duration of the connection. That reservation exactly matches a steady, long-lived, constant-rate flow and therefore:

    - Gives **guaranteed throughput** (the reserved bandwidth =r= r=r per call).
    - Gives **low and predictable delay and jitter** (no queueing variability caused by multiplexing other bursty flows).
- Packet switching can still carry steady flows, but it relies on statistical multiplexing with other flows. For truly constant, long-lived flows where predictable performance is required (e.g., real-time streams with strict bounds), circuit switching is the cleaner fit.

(b)

- Let the set of flows crossing a link have rates r1,r2,.... If for every link the capacity C satisfies ∑iri<C, then the link can forward all arriving bits without building an unbounded queue. Hence **no persistent congestion** occurs and no congestion-control mechanism is required to avoid collapse. Packets won't get queued indefinitely due to overload.

Caveats (why you might still want some control in practice):

- **Transient synchronization or small timing misalignments** can cause momentary bursts even if average load < capacity, producing short queues and jitter.
- **New flows arriving** or flows that are not perfectly steady could push instantaneous load above capacity. Congestion control provides robustness to such changes.
- **Fairness, loss recovery, and flow management:** Even without congestion, transport-layer mechanisms (e.g., TCP's reliability, retransmission, and flow-control) are useful for correctness and fairness if some flows change rate.
- **Failures or reroutes** could temporarily reduce available capacity on some links, creating congestion unless rate adaptation exists.

Q: Why connectionless packet switching (used in IP) was preferred over circuit switching:

1. **Resource utilization efficiency**

- **Circuit switching** (like in telephone networks) requires reserving a dedicated path (with bandwidth) for the entire session, even during silent/idle times.
- **Packet switching** allows multiple users to share the same links dynamically. When one user is not sending, others can use the bandwidth → much higher efficiency.

2. **Robustness & fault tolerance**
    - In circuit switching, if any link or switch on the dedicated path fails, the whole connection is broken.
    - In packet switching, each packet can take a different path. If one path/link fails, packets are re-routed → more resilient to failures (important for military/academic use in early Internet design).
3. **Scalability and flexibility**
    - Setting up circuits for millions of flows would require huge signaling overhead.
    - Packet switching is *connectionless* at the IP layer: no setup is needed, packets are simply forwarded hop by hop, which makes the system simpler and more scalable.
4. **Support for bursty computer traffic**
    - Computer data traffic is naturally bursty (big file transfers followed by idle periods).
    - With packet switching, bandwidth is used only when data is actually being transmitted, unlike circuit switching where resources are locked for the whole session.

Q: How Traceroute Works

Traceroute is a diagnostic tool that uses **TTL behavior** to discover the path packets take to reach a destination.

Step-by-Step Working:

1. **Send packets with increasing TTL values**
    - First packet: TTL = 1 → reaches the first router.
        - Router decrements TTL → 0, drops it → sends back **ICMP Time Exceeded**.
        - Source learns the IP of the **first hop**.
    - Second packet: TTL = 2 → reaches the second router, expires there → ICMP reply.
        - Source learns the IP of the **second hop**.
    - And so on... until it reaches the destination.
2. **At the destination:**
    - When TTL is finally large enough to reach the target host, instead of "Time Exceeded," the host responds with an **ICMP Echo Reply (ping)** or **UDP/TCP Port Unreachable** message (depending on traceroute implementation).
    - This tells the source it has reached the destination.
3. **Traceroute output:**
    - For each TTL step, traceroute shows the IP (and often hostname) of the router that replied, plus the round-trip times (RTT) for multiple probes.

- **TTL** = a countdown value decremented by routers to avoid infinite loops.
- **Traceroute** = sends packets with increasing TTL to force routers to reveal themselves one by one, mapping the full path to the destination.

🌐 Example Network

```
Host (Source) → R1 → R2 → R3 → Destination (Server)
```

- 3 routers between source and destination.
- We'll show how TTL works when traceroute runs.

Step-by-Step with TTL

🔷 **First probe (TTL = 1)**

- Packet leaves source with TTL = 1.
- R1 decrements TTL → 0 → packet **dropped**.
- R1 sends back **ICMP Time Exceeded**.

  👉 Traceroute learns: **Hop 1 = R1 (IP)**.

🔷 **Second probe (TTL = 2)**

- Packet leaves with TTL = 2.
- R1 decrements → 1, forwards to R2.
- R2 decrements → 0 → packet **dropped**.
- R2 sends back ICMP reply.

  👉 Traceroute learns: **Hop 2 = R2 (IP)**.

🔷 **Third probe (TTL = 3)**

- Packet leaves with TTL = 3.
- R1 decrements → 2, forwards to R2.
- R2 decrements → 1, forwards to R3.
- R3 decrements → 0 → packet dropped.
- R3 sends back ICMP reply.

  👉 Traceroute learns: **Hop 3 = R3 (IP)**.

🔷 **Fourth probe (TTL = 4)**

- Packet leaves with TTL = 4.
- R1 → TTL = 3 → forward.
- R2 → TTL = 2 → forward.
- R3 → TTL = 1 → forward to Destination.
- Destination decrements → TTL = 0, BUT since it's the **final host**, it replies with **ICMP Echo Reply** (Windows) or **Port Unreachable** (Linux traceroute).

  👉 Traceroute learns: **Hop 4 = Destination**.

2.1. Suppose Host A wants to send a large file to Host B. The path from Host A to Host B has three links with transmission rates *R*1 = 250 kbps, *R*2 = 1 Mbps, and *R*3 = 500 Kbps.

a) **Assuming no other traffic in the network, what is the throughput for the file transfer? (3 points)**
b) **Suppose the size of the file is 8 million bytes. How long will it take to transfer the file from Host A to Host B? (3 points)**
c) **Repeat (a) and (b), when the transmission rate of *R*2 is reduced to 200 kbps. (4 points)**

2.2. Consider a packet of length 1500 bytes that needs to be transmitted over a link of 3100 km with a transmission rate 3 Mbps and a propagation speed 2.5 x 10^8 m/s.

a) **How long does a packet take to propagate over a link ? (4 points)**
b) **More generally, how long does it take for a packet of length *L* to propagate over a link of distance *d*, propagation speed *s*, and transmission rate *R* bps? (4 points)**
c) **Does this delay depends on the packet length? Does this delay depends on transmission rate? (2 points)**

Problems

2.1. A ——+——+——→ B

a). $R_1 = 250$ Kbps, $R_2 = 1$ Mbps, $R_3 = 500$ Kbps

$R_1 = 250{,}000$ bps, $R_2 = 1{,}000{,}000$ bps, $R_3 = 500{,}000$ bps

Throughput $= \min(R_1, R_2, R_3) = 250{,}000$ bps

b). File size = 8 Million bytes $= 8{,}000{,}000$ bytes $\times 8$

$= 64{,}000{,}000$ bits

250,000 —— 1s.

64,000,000 —— $\frac{1}{256}$ s $= 4$ min 16 s

c) $R_2 = 200\,000$ bps

Throughput $= \min(R_1, R_2, R_3) = 200\,000$ bps

$t = \frac{64{,}000{,}000}{200{,}000} = 320$ s $= 5$ min 20 s

Assumption :- ignored d_{queue}, d_{proc} & d_{prop}.

2.2. 1500 bytes, $d = 3100$ km, $R = 3$ Mbps, $s = 2.5 \times 10^8$ m/s.

a) $d_{prop} = \frac{3100 \text{ km}}{2.5 \times 10^8 \text{ m/s}} = \frac{3100 \times 10^3 \text{ m}}{2.5 \times 10^8 \text{ m/s}} = 0.0124$ s $= 12.4$ ms

b) $d_{trans} = \frac{1500 \times 8}{3 \times 10^6} = \frac{L}{R} = \frac{1}{250}$ s $= 4$ ms $= 4 \times 10^{-3}$ s

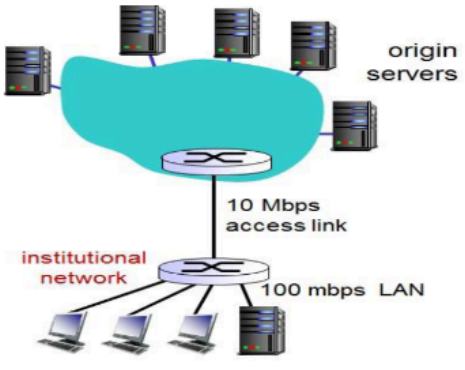
$t = d_{prop} + d_{trans} = 12.4$ ms $+ 4$ ms $= \boxed{16.4 \text{ ms}}$

Dated:

$d_{prop} \rightarrow \frac{d}{s}$ doesn't depend on packet (length (L)) & transmission rate (R).

$d_{trans} \rightarrow L/R$ depends on (L) packet length & (R) transmission rate.

2.3. Consider the following network topology where an institutional network of 100 Mbps capacity is connected to the Internet with an Access Link of 10 Mbps. Lets assume the browsers in the institutional network request objects of an average size of 600 Kbits at a rate of 20 requests/second. Find the:

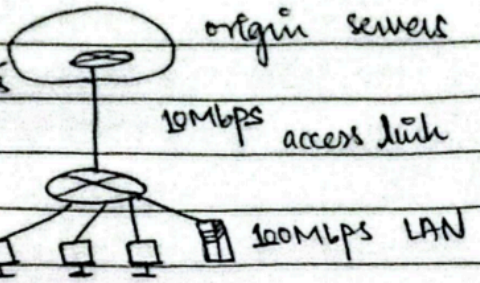
4. 

a. **Average data rate to browsers over the access link. (1 points)**
b. **LAN utilization. (2 points)**
c. **Access link utilization. (2 points)**

Now suppose that you have installed a web cache in the institutional network to reduce the load on the access network. With the addition of web cache 33% of the requests are satisfied locally and 67% of the requests are forwarded to the origin servers. Find the:

d. **Average data rate to browsers over access link. (1 points)**
e. **LAN utilization. (2 points)**
f. **Access link utilization. (2 points)**

2.3



(a). Average data rate to browsers over the access link.

Object size = 600 Kbits = 600,000 bits.

Request rate = 20 req/s.

Rate = 600,000 x 20 = 12,000,000 bits/s = **12Mbps.**

(b). LAN utilization = $\frac{12}{100}$ = 0.12 = **12%**

(c). Access link utilization = $\frac{12}{10}$ ← Average data rate to browsers over the access link.

→ R of access link

= **120%** PROBLEM!!! CONGESTION OCCURS!

Web cache → 33% requests are satisfied locally, 67% requests are forwarded to origin servers.

Now only 67% of request go to origin server through the access link.

0.67 x 12 Mbps = 8.04 Mbps

Access link utilization = $\frac{8.04}{10}$ = **80.4%**

LAN utilization = $\frac{12}{\frac{12}{100} \times 100}$ %, LAN utilization remain unchanged.

- Caching reduces external (access link) load & relieves the internet link, but the LAN still carries the full delivered contents to browser (so LAN utilization stay the same).

DIAGRAMS