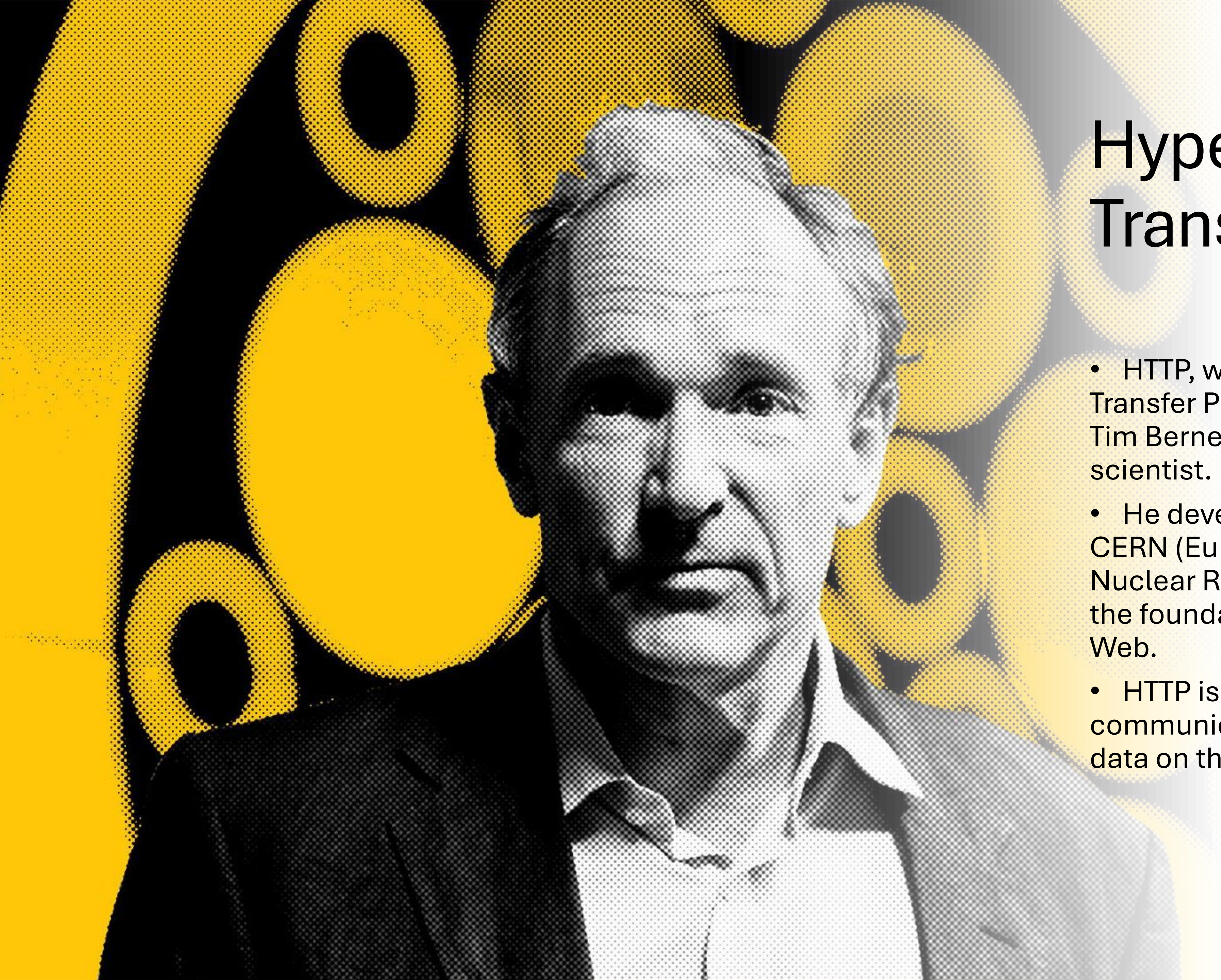
Lecture Week 03: HTTP

Dr. Hamed Hamzeh

06/02/2025

Introduction to HTTP



Hyper Text Transfer Protocol

- HTTP, which stands for Hypertext Transfer Protocol, was created by Sir Tim Berners-Lee, a British computer scientist.
- He developed HTTP while working at CERN (European Organization for Nuclear Research) in 1989, as part of the foundation for the World Wide Web.
- HTTP is the protocol that enables communication and the transfer of data on the World Wide Web.

HTTP History

1991

HTTP/0.9

- Initial version by Tim Berners-Lee.
- Simple protocol for plain text data.

1996

HTTP/1.0

- Allowed different data types.
- Enabled multiple objects over a single TCP connection.

1997–1999

HTTP/1.1

- Introduced persistent connections.
- Pipelining for concurrent requests.
- Host headers for multiple websites on one IP.
- Optimizations for caching and compression.

2015

HTTP/2

- Binary framing for concurrent streams.
- Header compression and multiplexing.
- Backward-compatible with HTTP/1.1.
- Significant performance enhancements.

2018

HTTP/3

- Uses QUIC transport protocol.
- Focus on reducing latency, improving security.
- Ongoing development for evolving internet demands.

Position of HTTP in the OSI Model

The OSI (Open Systems Interconnection) model is a conceptual framework that standardizes the functions of a communication system into seven abstraction layers.

HTTP operates at the **Application Layer (Layer 7)** of the OSI model.

Key Points:

The Application Layer deals with high-level protocols, user interfaces, and network-aware applications.

HTTP is responsible for communication between applications, specifically web browsers, and servers.

Basics of HTTP Request

Request Methods:

GET: Retrieve data from the server.

POST: Send data to the server to create a resource.

Request Headers:

Host: Specifies the domain name of the server.

User-Agent: Identifies the user agent (e.g., browser) making the request.

```
GET /index.html HTTP/1.1 Host: www.example.com User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
```

HTTP request message

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

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

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
```

Overview of HTTP Methods

HTTP Methods:

- HTTP defines several methods that indicate the desired action to be performed on a resource.
- Commonly used methods include GET, POST, PUT, DELETE, and more.

Purpose:

- Each method serves a specific purpose in interacting with resources on the server.

HTTP GET Method

GET Method:

Used to request data from a specified resource.

Requests should only retrieve data and should not have any other effect on the server.

Example:

A browser requesting a webpage.

HTTP Request:

```
GET /index.html HTTP/1.1
```

HTTP POST Method

POST Method:

Used to submit data to be processed to a specified resource.

Often used when uploading a file or submitting a form.

Example:

Submitting a form with user data.

HTTP Request:

```
POST /submit-form HTTP/1.1 Content-Type:
application/x-www-form-urlencoded
username=johndoe&password=secretpassword
```

HTTP PUT Method

PUT Method:

Used to update a resource or create a new resource if it doesn't exist.

The request typically contains the full representation of the resource.

Example:

Updating the content of an existing document.

HTTP Request:

```
PUT /update-document HTTP/1.1
Content-Type: text/plain This
is the updated content.
```

HTTP DELETE Method

DELETE Method:

Used to request the removal of a resource from the server.

The server decides whether to delete the resource or not.

Example:

Deleting a user account.

HTTP Request:

```
DELETE /user/johndoe HTTP/1.1
```

Additional HTTP Methods

Other Methods:	HTTP defines additional methods like HEAD, OPTIONS, PATCH, and more. HEAD: Similar to GET but retrieves headers only. OPTIONS: Describes the communication options for the target resource. PATCH: Applies partial modifications to a resource.
Usage:	These methods provide additional functionality and flexibility in various scenarios.

Basics of HTTP Response

Status Codes:

- 200 OK: Successful request.
- 404 Not Found: Requested resource not found.
- Response Headers:

Content-Type: Specifies the type of data in the response.

Server: Identifies the server software.

```
HTTP/1.1 200 OK Content-Type: text/html
Server: Apache/2.4.41 (Unix)
```

Common HTTP Response Codes

HTTP 200 OK – Success

Explanation:

- The server successfully processed the request, and the response contains the requested information.
- This is the standard response for successful HTTP requests.

HTTP response message

status line (protocol status code status phrase)

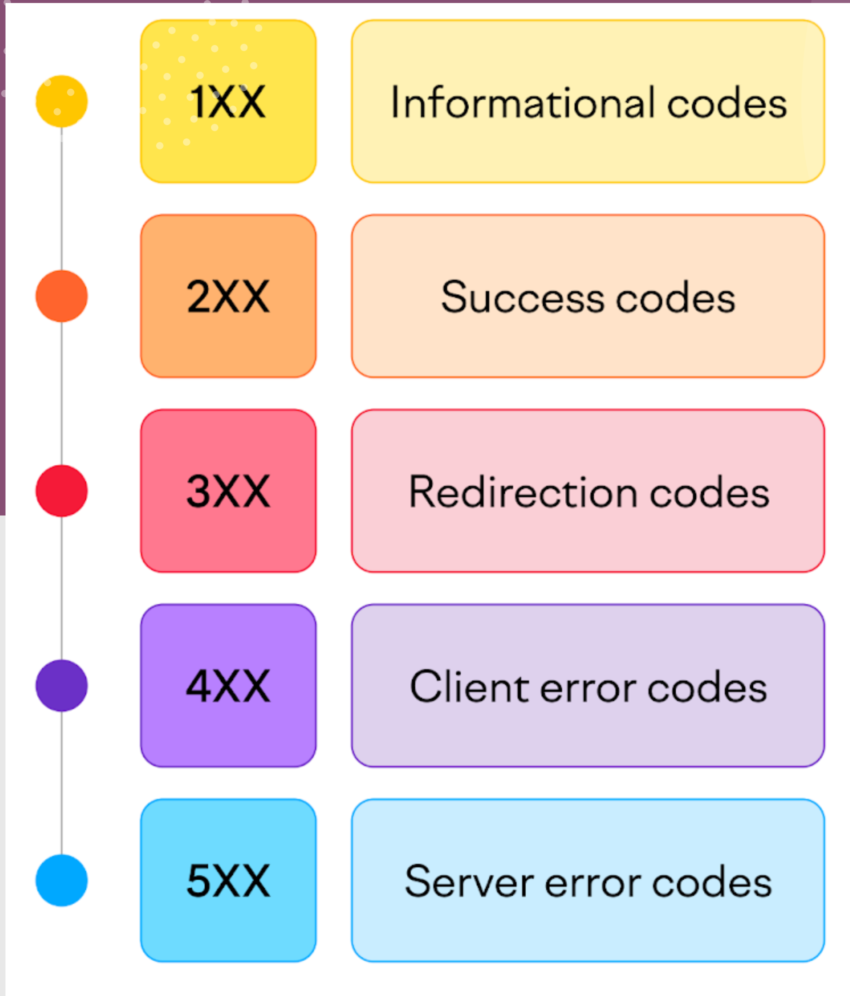
header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

HTTP response codes

- Three-digit numbers that are returned by a server in response to a client's request.
- These codes provide information about the status of the request and help in troubleshooting and understanding the outcome of the request.



1xx Informational Codes

100 Continue:

- The server has received the initial part of the request and is ready to proceed.

101 Switching Protocols:

- The server is switching protocols as requested by the client.

2xx Success Codes

200 OK:	• The request was successful.
201 Created:	• The request was successful, and a new resource was created.
204 No Content:	• The request was successful, but there is no content to send back.

3xx Redirection Codes

301 Moved Permanently:

- The requested resource has been permanently moved to a new location.
- Example: URL structure change.

302 Found (or 303 See Other):

- Indicates that the requested resource is temporarily located at another URI.
- Example: Temporary redirection.

307 Temporary Redirect:

- Similar to 302 but explicitly indicates that the request method should not change.

Client Error Codes

400 Bad Request: The server cannot understand the request due to a client error.

401 Unauthorized: The request requires user authentication.

404 Not Found: The requested resource could not be found on the server.

5xx Server Error Codes

500 Internal Server Error:

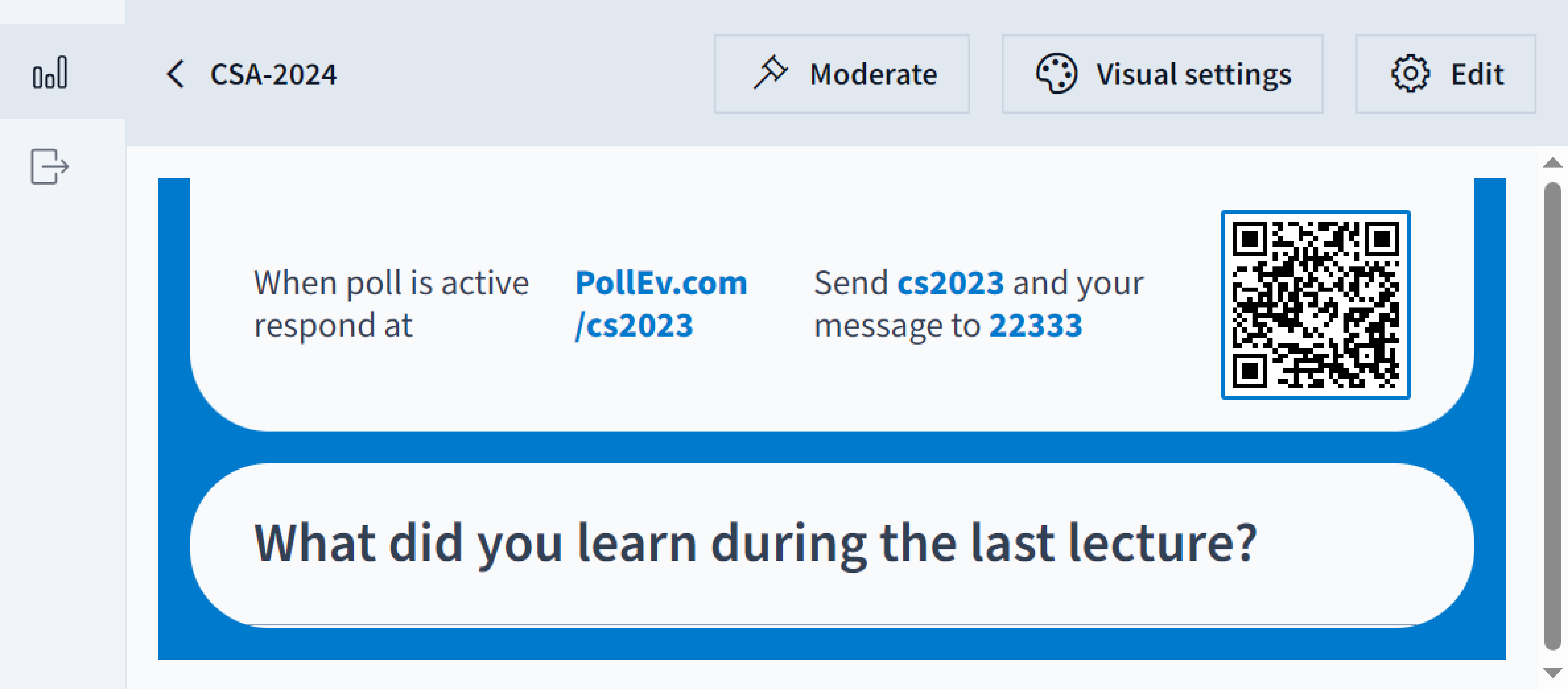
- The server encountered an unexpected condition that prevented it from fulfilling the request.

503 Service Unavailable:

- The server is currently unavailable to handle the request due to maintenance or overload.

Question

- Imagine a scenario where a user submits a form on a website to perform an action, such as submitting a comment or making a purchase. After the action is completed, the server wants to redirect the user back to a specific page, typically the page they were on before submitting the form. Which http response code you may get from the server?



Statelessness of HTTP

HTTP is a stateless protocol, meaning each request is independent and has no knowledge of previous requests.

Cookies and sessions are used to maintain state between requests.

Example: A user logging into a website might receive a session cookie to maintain their authenticated state.

HTTPS and Security

HTTPS (Hypertext Transfer Protocol Secure) is the secure version of HTTP.

It encrypts data using SSL/TLS to ensure confidentiality and integrity.

Security Measures:

SSL/TLS encryption.

Secure Sockets Layer (SSL) and its successor Transport Layer Security (TLS) protocols.

Example:

https://www.example.com indicates a secure connection.

Maintaining user/server state: cookies

Web sites and client browser use cookies to maintain some state between transactions

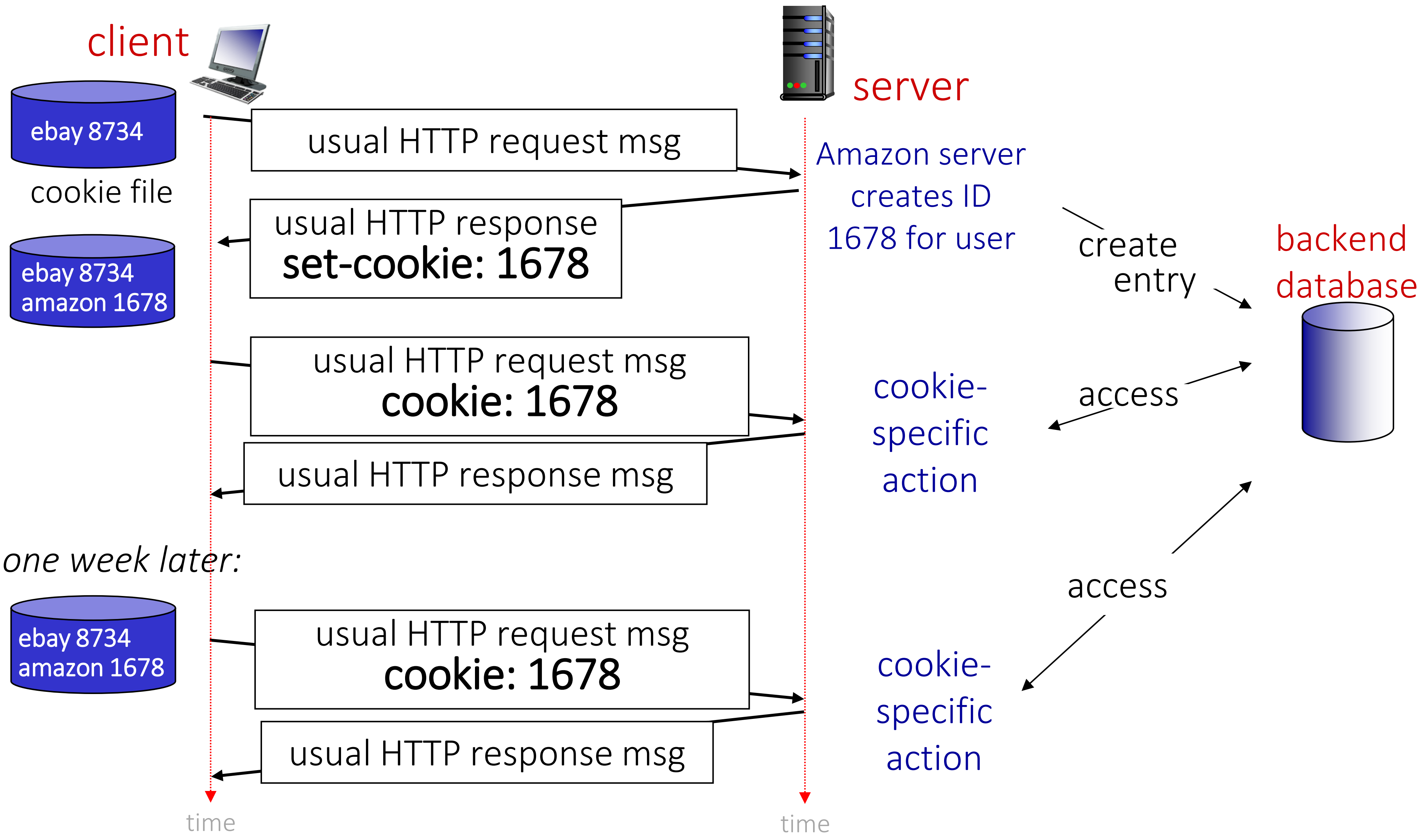
four components:

1) cookie header line of HTTP response message
2) cookie header line in next HTTP request message
3) cookie file kept on user's host, managed by user's browser
4) back-end database at Web site

Example:

- Susan uses browser on laptop, visits specific e-commerce site for first time
- when initial HTTP requests arrives at site, site creates:
  - unique ID (aka "cookie")
  - entry in backend database for ID
- subsequent HTTP requests from Susan to this site will contain cookie ID value, allowing site to "identify" Susan

Maintaining user/server state: cookies



HTTP cookies: comments

What cookies can be used for:

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

Challenge: How to keep state:

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

aside

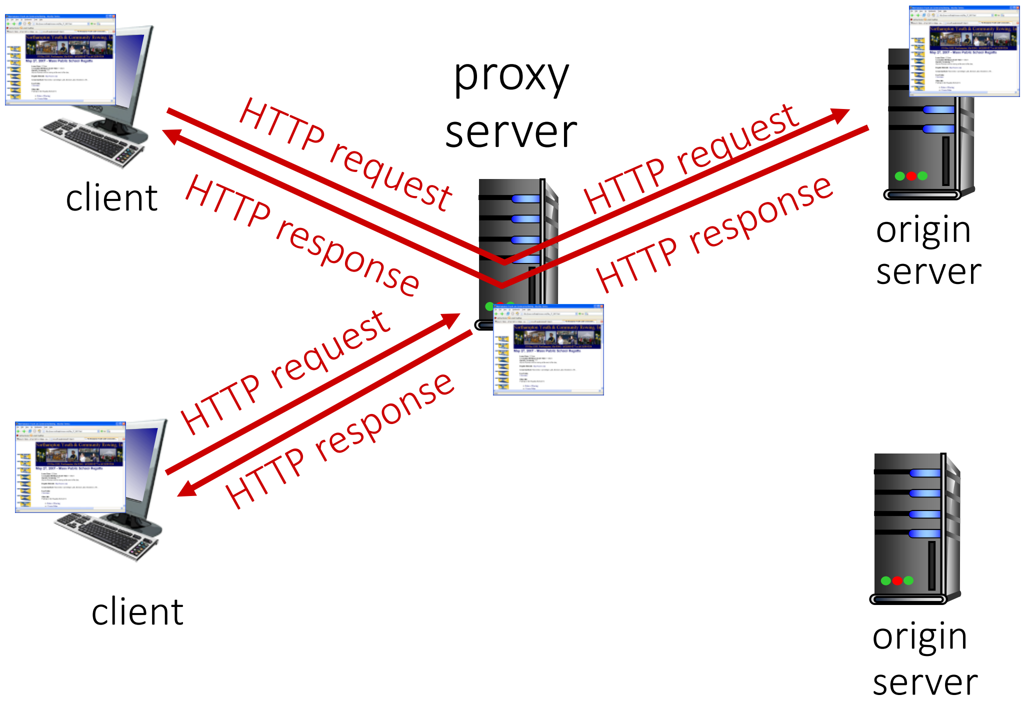
cookies and privacy:

- cookies permit sites to learn a lot about you on their site.
- third party persistent cookies (tracking cookies) allow common identity (cookie value) to be tracked across multiple web sites

Web caches (proxy servers)

Goal: satisfy client request without involving origin server

- user configures browser to point to a Web cache
- browser sends all HTTP requests to cache
  - if object in cache: cache returns object to client
  - else cache requests object from origin server, caches received object, then returns object to client



Caching example

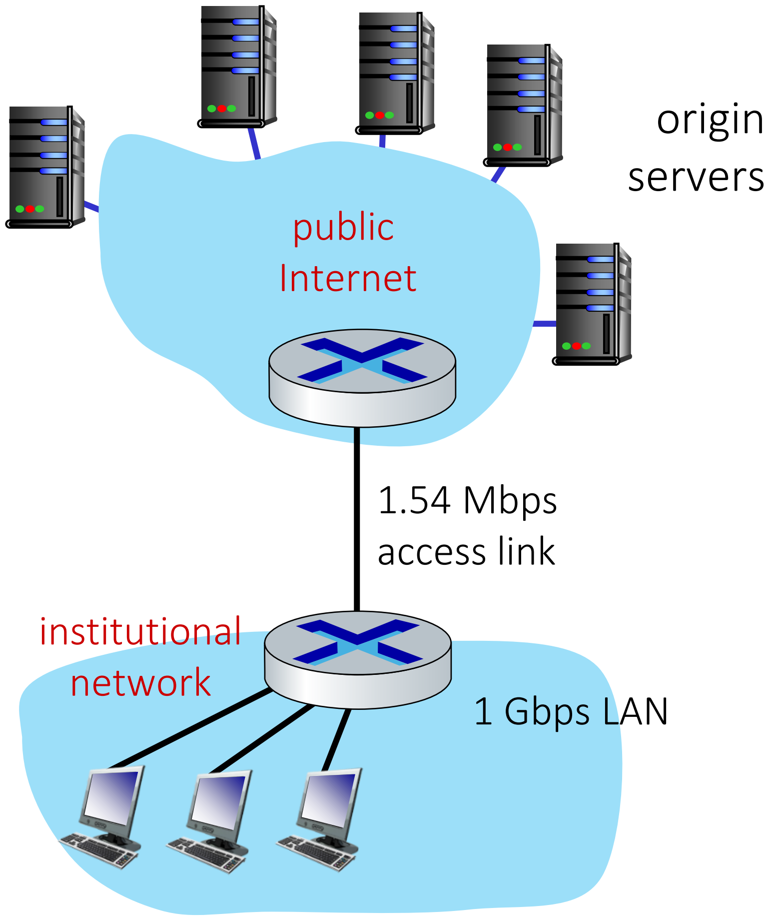
Scenario:

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

Performance:

- LAN utilization: .0015
- access link utilization = .97
- end-end delay = Internet delay + access link delay + LAN delay = 2 sec + minutes + usecs

problem: large delays at high utilization!



Caching example: buy a faster access link

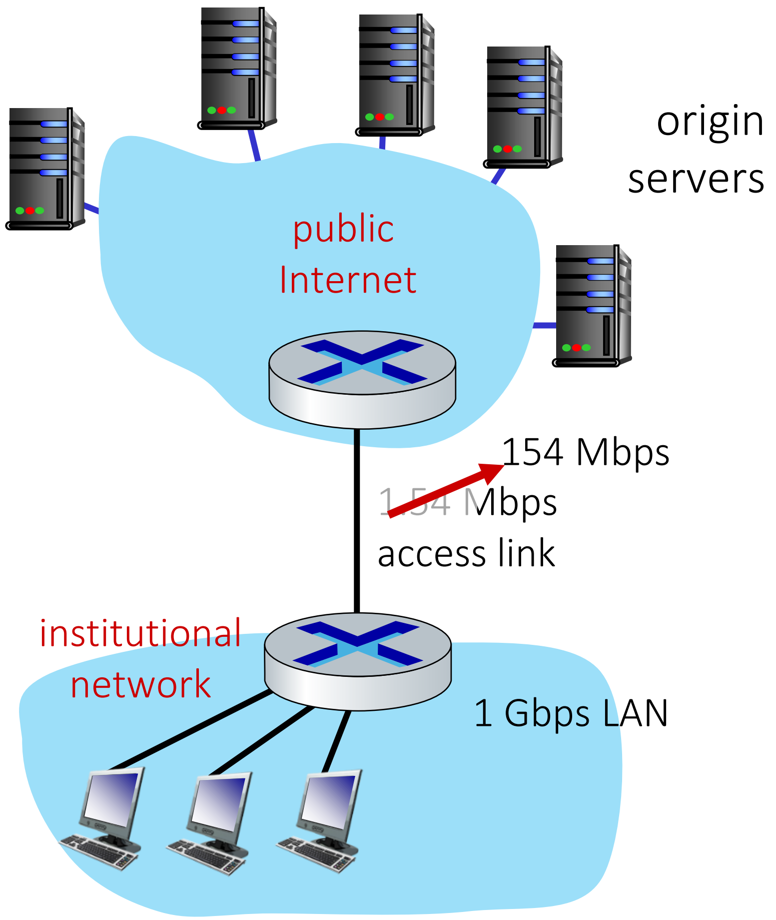
Scenario:

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

Performance:

- LAN utilization: .0015
- access link utilization = ~~.97~~ → .0097
- end-end delay = Internet delay + access link delay + LAN delay
  = 2 sec + ~~minutes~~ msecs + usecs

Cost: faster access link (expensive!)



Caching example: install a web cache

Scenario:

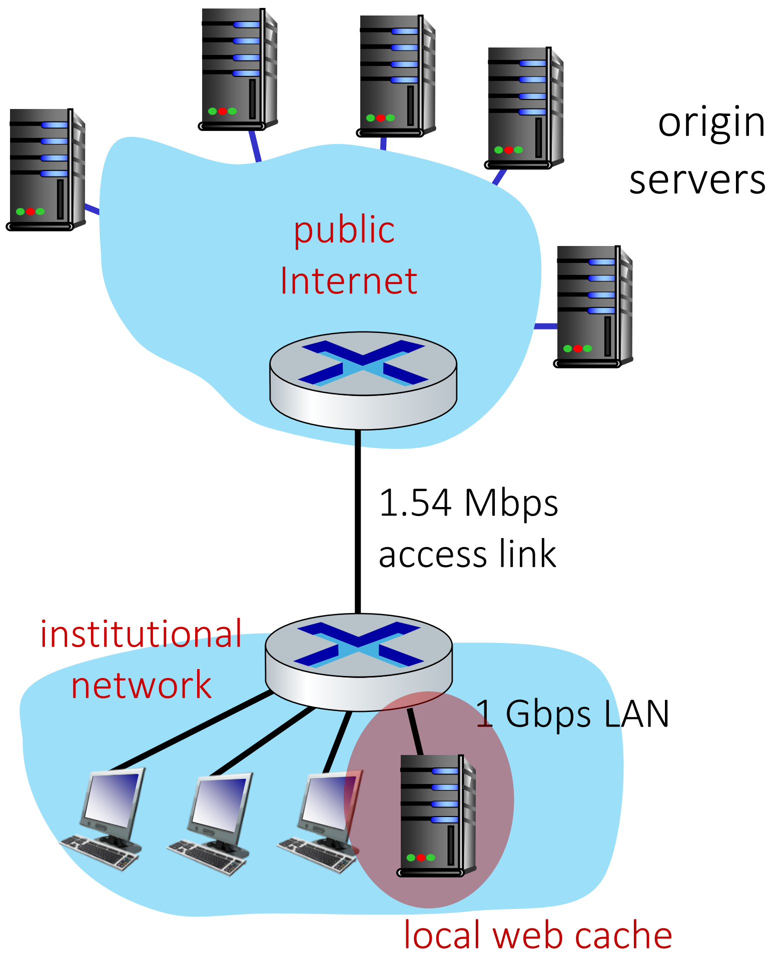
- access link rate: 1.54 Mbps
- RTT from institutional router to server: 2 sec
- Web object size: 100K bits
- Avg request rate from browsers to origin servers: 15/sec
  - avg data rate to browsers: 1.50 Mbps

Performance:

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

How to compute link utilization, delay?

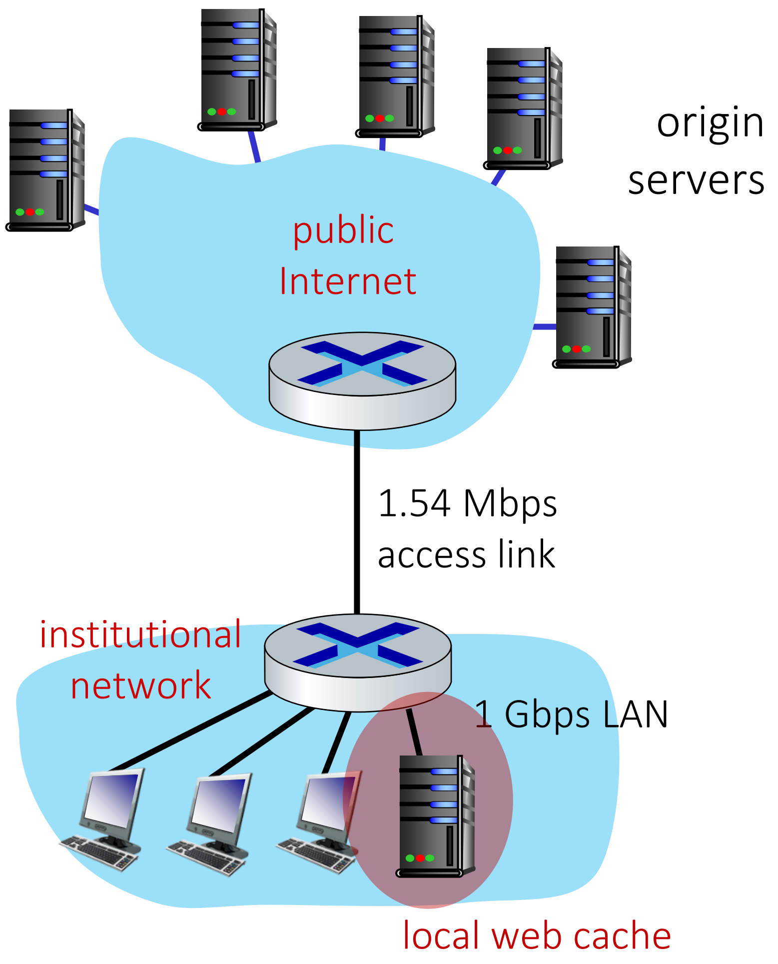
Cost: web cache (cheap!)



Caching example: install a web cache

Calculating access link utilization, end-end delay with cache:

- suppose cache hit rate is 0.4: 40% requests satisfied at cache, 60% requests satisfied at origin
- access link: 60% of requests use access link
- data rate to browsers over access link
  = 0.6 * 1.50 Mbps = .9 Mbps
- utilization = 0.9/1.54 = .58
- average end-end delay
  = 0.6 * (delay from origin servers)
  + 0.4 * (delay when satisfied at cache)
  = 0.6 (2.01) + 0.4 (~msecs) = ~ 1.2 secs



lower average end-end delay than with 154 Mbps link (and cheaper too!)

Domain Name System(DNS)

It is a decentralized naming system that translates domain names (e.g., example.com) into IP addresses (e.g., 192.0.2.1) . There are different record types:

A (Address) Record: This record type maps a domain name to an IPv4 address. It provides the IP address associated with a hostname.

AAAA (IPv6 Address) Record: This record type maps a domain name to an IPv6 address. It provides the IPv6 address associated with a hostname.

CNAME (Canonical Name) Record: This record type provides an alias or canonical name for a domain. It allows a domain name to be associated with another domain name.

MX (Mail Exchanger) Record: This record type specifies the mail servers responsible for accepting incoming emails for a domain. It provides information about the email servers that should be used to send emails to a particular domain

DNS (Name Server) Record: This record type specifies the authoritative name servers for a domain. It provides information about the servers responsible for handling DNS queries for a particular domain.

HTTP/2

Content:HTTP/2 Overview:

Developed to overcome limitations of HTTP/1.1.

A binary protocol that brings performance improvements.

Multiplexing, header compression, and prioritization are key features.

Advantages:

Faster loading times, reduced latency, and improved efficiency.

Key Features of HTTP/2

Multiplexing: Multiple requests and responses can be sent concurrently over a single connection.

Header Compression: Headers are compressed to reduce overhead, improving efficiency.

Prioritization: Requests can be assigned priority levels to optimize resource loading.

Example: Loading multiple resources in parallel without waiting for one to complete before starting another.

HTTP/2 in Comparison to HTTP/1.1

Performance Improvements:

- HTTP/2 performs better than HTTP/1.1 due to its advanced features.

Multiplexing Comparison:

- HTTP/1.1 relies on multiple connections for parallelism, while HTTP/2 uses a single connection.

Example:

- Faster page load times and improved user experience.

Considerations for Migrating to HTTP/2

Server and Browser Support:

- Ensure that both servers and clients support HTTP/2.

Configuration:

- Update server configurations to enable HTTP/2.

Testing:

- Thoroughly test the website/application for compatibility.

Example:

- Verifying that both the server and the client support HTTP/2.

Future Trends - HTTP/3

Introduction to HTTP/3:

- Ongoing development to further enhance web communication.
- Based on the QUIC protocol, aiming to improve performance and security.

Advancements:

- Reduced latency, improved security, and better handling of packet loss.

Example:

- Potential for even faster and more secure communication.

Sockets and HTTP Interaction

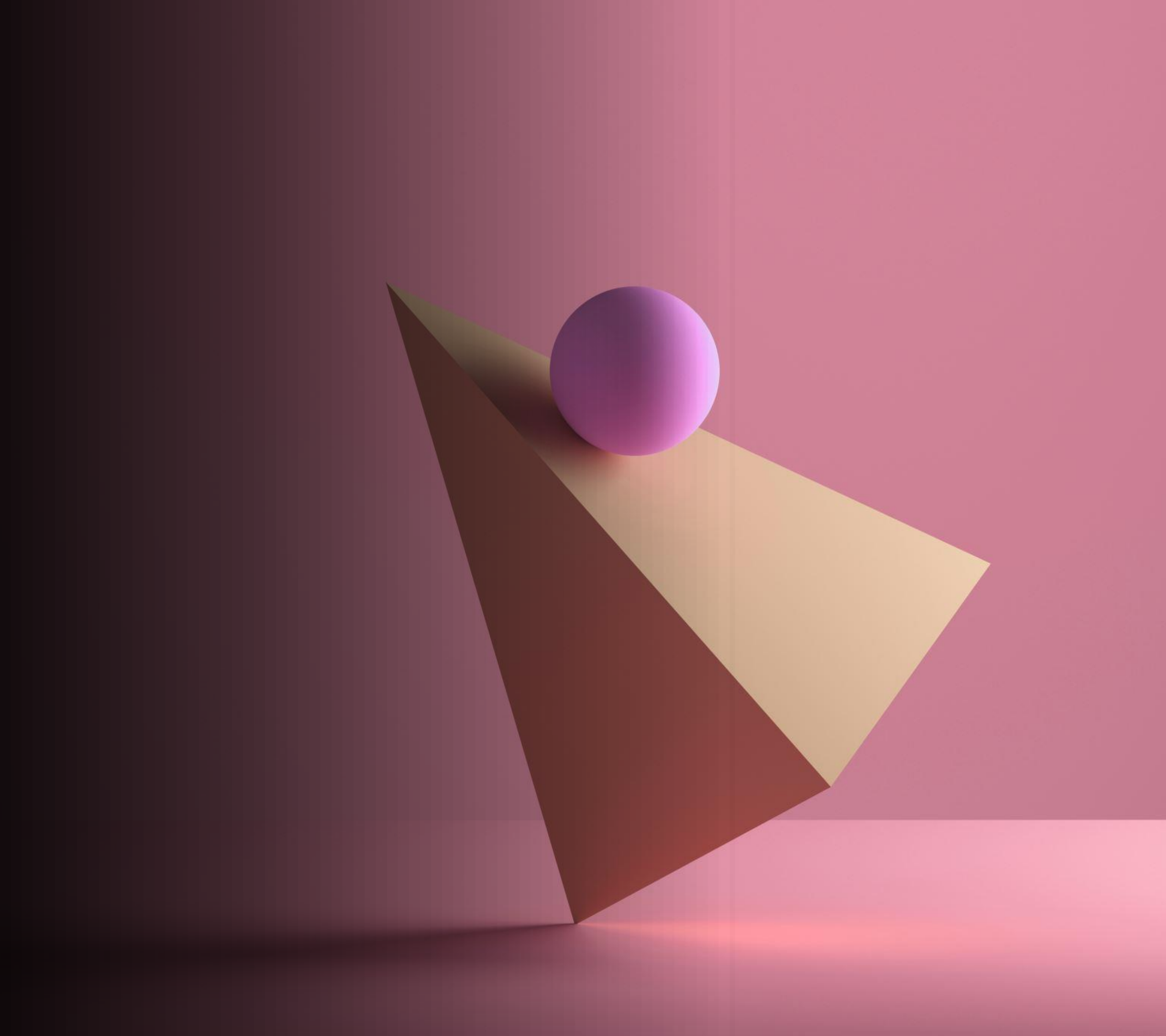
Understanding Sockets:

- Sockets are communication endpoints that allow data transfer between a client and a server.
- HTTP, being a protocol, utilizes sockets for communication between clients and servers.

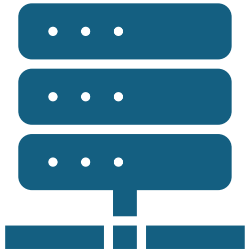
Connection Establishment:

- Sockets are used to establish connections between clients and servers, facilitating HTTP communication.

Java and HTTP



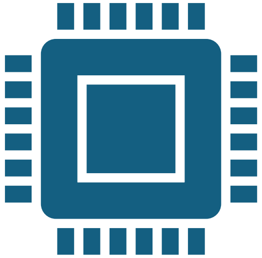
Socket Interaction in HTTP - Overview



Client-Side Interaction:

The client creates a socket and initiates a connection to the server.

The client sends an HTTP request using the socket.



Server-Side Interaction:

The server accepts the connection, creates a new socket for communication, and processes the HTTP request.

The server sends an HTTP response back to the client using the socket.

HTTP in Java

The com.sun.net.httpserver package includes classes to create an HTTP server and handle HTTP requests.

The main class in this package is HttpServer, which is responsible for creating and managing the server.

It allows you to create contexts for different URIs and associate handlers with these contexts to handle incoming HTTP requests.

Overview of Java's java.net Package

The java.net package in Java is a core package that provides fundamental networking functionality.

It includes classes and interfaces for working with URLs, sockets, and network connections.

java.net is crucial for implementing HTTP communication in Java applications.

URL Class

- Represents a Uniform Resource Locator.
- Used for creating, parsing, and manipulating URLs.
- Example:

```
URL url = new URL("https://example.com");
```

URLConnection Class

- Abstract class that represents a communication link between an application and a URL.
- Serves as the superclass for HttpURLConnection.
- Example:

```
URLConnection connection = url.openConnection();
```

HttpURLConnection Class

- Extends URLConnection and provides specific support for HTTP.
- Allows setting HTTP request methods, headers, and handling responses.
- Example:

```
HttpURLConnection httpURLConnection = (HttpURLConnection) url.openConnection();
```

HTTPServer in Java

HTTPServer is a class in the com.sun.net.httpserver package in Java.

It provides a simple HTTP server framework for creating and handling HTTP requests.

```
import com.sun.net.httpserver.HttpServer;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpExchange;
```

```
HttpServer server = HttpServer.create(new InetSocketAddress(port), 0);
```

HTTPServer Class

static HttpServer	create() creates a HttpServer instance which is initially not bound to any local address/port.
static HttpServer	create(InetSocketAddress addr, int backlog) Create a HttpServer instance which will bind to the specified InetSocketAddress (IP address and port number) A maximum backlog can also be specified.
abstract HttpContext	createContext(String path) Creates a HttpContext without initially specifying a handler.
abstract HttpContext	createContext(String path, HttpHandler handler) Creates a HttpContext.
abstract InetSocketAddress	getAddress() returns the address this server is listening on
abstract Executor	getExecutor() returns this server's Executor object if one was specified with setExecutor(Executor), or null if none was specified.

HTTP Handlers

- HTTP handlers handle incoming requests and generate responses.
- Implement the HttpHandler interface:

```
class MyHandler implements HttpHandler {
    public void handle(HttpExchange exchange) throws IOException {
        // Handle the incoming request and generate a response
    }
}
```

HTTP Exchange Methods

abstract void	close() Ends this exchange by doing the following in sequence:
abstract Object	getAttribute(String name) Filter modules may store arbitrary objects with HttpExchange instances as an out-of-band communication mechanism.
abstract HttpContext	getHttpContext() Get the HttpContext for this exchange
abstract InetSocketAddress	getLocalAddress() Returns the local address on which the request was received
abstract HttpPrincipal	getPrincipal() If an authenticator is set on the HttpContext that owns this exchange, then this method will return the HttpPrincipal that represents the authenticated user for this HttpExchange.
abstract String	getProtocol() Returns the protocol string from the request in the form *protocol/majorVersion.minorVersion*.
abstract InetSocketAddress	getRemoteAddress() Returns the address of the remote entity invoking this request
abstract InputStream	getRequestBody() returns a stream from which the request body can be read.
abstract Headers	getRequestHeaders() Returns an immutable Map containing the HTTP headers that were included with this request.

Example

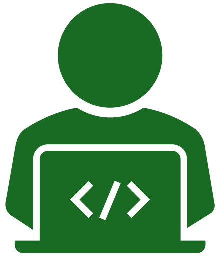
```java
public class MyHttpHandler implements HttpHandler {

    @Override
    public void handle(HttpExchange exchange) throws IOException {
        // Retrieve request information
        String requestMethod = exchange.getRequestMethod();
        String requestURI = exchange.getRequestURI().toString();

        // Set response headers
        exchange.getResponseHeaders().set("Content-Type", "text/plain");

        // Generate response message
        String response;
        if (requestMethod.equals("GET")) {
            response = "Hello from GET request to " + requestURI;
        } else {
            response = "Unsupported request method: " + requestMethod;
        }

        // Set response status code and send response
        exchange.sendResponseHeaders(200, response.length());
        OutputStream outputStream = exchange.getResponseBody();
        outputStream.write(response.getBytes());
        outputStream.close();
    }
```

HTTP Status Codes in Java

HTTP Status Codes:

HTTP responses include status codes that indicate the result of the request.

Java provides classes like HttpURLConnection to handle HTTP responses.



Handling Responses:

Demonstrates how to use HttpURLConnection to check and handle different HTTP status codes.

Handling HTTP Status Codes - Example 1

- Checking Status Code:
  - Demonstrates checking the HTTP status code using HttpURLConnection.
  - Example shows handling a successful response (status code 200).

```java
int statusCode = connection.getResponseCode();
if (statusCode == HttpURLConnection.HTTP_OK) {
 // Process the successful response
} else {
// Handle other cases, e.g., errors or redirects
}
```

Handling HTTP Status Codes - Example 3

```
int statusCode = connection.getResponseCode();

if (statusCode == HttpURLConnection.HTTP_NOT_FOUND) {
    // Handle the 404 Not Found error
} else {
    // Handle other client errors or server errors
}
```

Multi-Threading for Sockets

Single-Threaded Socket Programming

Traditional (Single-threaded) Socket Communication:

- In single-threaded socket programming, each client request is processed sequentially.
- The server handles one client at a time, moving on to the next once the current task is completed.
- Simple and straightforward approach but has limitations.

Limitations and Challenges:

- **Limited Scalability:**
  - Handling multiple clients concurrently becomes challenging.
  - Performance may degrade as the number of clients increases.
- **Blocking Nature:**
  - The server blocks while waiting for a client request to be processed.
  - If one client takes a long time, other clients have to wait.

```java
public class SingleThreadedServer {

    public static void main(String[] args) {
        final int port = 8080;

        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Server is listening on port " + port);

            while (true) {
                Socket clientSocket = serverSocket.accept(); // Blocking call, waits for a client to
                    connect
                handleClient(clientSocket);
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket clientSocket) throws IOException {
        try (
            Scanner in = new Scanner(clientSocket.getInputStream());
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)
        ) {
            // Read data from the client
            String clientMessage = in.nextLine();
            System.out.println("Received from client: " + clientMessage);

            // Process data (in this case, just echoing back to the client)
            out.println("Server: " + clientMessage);
        }
    }
}
```

Multithreading in Sockets

- **Handling Multiple Clients Simultaneously:**
  - In traditional single-threaded socket programming, handling one client at a time may lead to scalability issues.
  - Multithreading enables concurrent processing of multiple client requests.
  - Each client connection is assigned its own thread, allowing parallel execution.
- **Improving Performance and Responsiveness:**
  - **Parallel Execution:**
    - Multithreading allows the server to process multiple client requests concurrently, significantly improving throughput.
    - Tasks that would otherwise block the server can be parallelized.
  - **Reduced Latency:**
    - With multithreading, the server can respond to clients more quickly, reducing the overall latency of the system.
    - Clients experience improved responsiveness and faster service.

```java
    public static void main(String[] args) {
        final int port = 8080;

        try (ServerSocket serverSocket = new ServerSocket(port)) {
            System.out.println("Multithreaded Server is listening on port " + port);

            while (true) {
                Socket clientSocket = serverSocket.accept(); // Blocking call, waits for a client to connect
                System.out.println("New client connected: " + clientSocket);

                // Delegate the client handling to a new thread
                Thread clientThread = new Thread(() -> handleClient(clientSocket));
                clientThread.start();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private static void handleClient(Socket clientSocket) {
        try (
            Scanner in = new Scanner(clientSocket.getInputStream());
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true)
        ) {
            // Read data from the client
            String clientMessage = in.nextLine();
            System.out.println("Received from client " + clientSocket + ": " + clientMessage);

            // Process data (in this case, just echoing back to the client)
            out.println("Server: " + clientMessage);
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                clientSocket.close();
                System.out.println("Closed connection for client: " + clientSocket);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```