Universidad Icesi

Computación 2 — Semana 1

Redes, Sockets Java, Apache Tomcat y Servlets

Facultad de Ingeniería — Universidad Icesi

Universidad ICESI

01. Protocolos de la Capa de Transporte y Aplicación

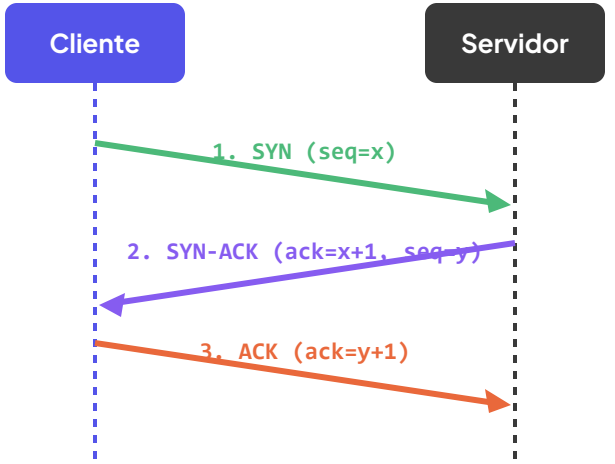
Fundamentos de TCP, UDP y el Protocolo HTTP/1.1

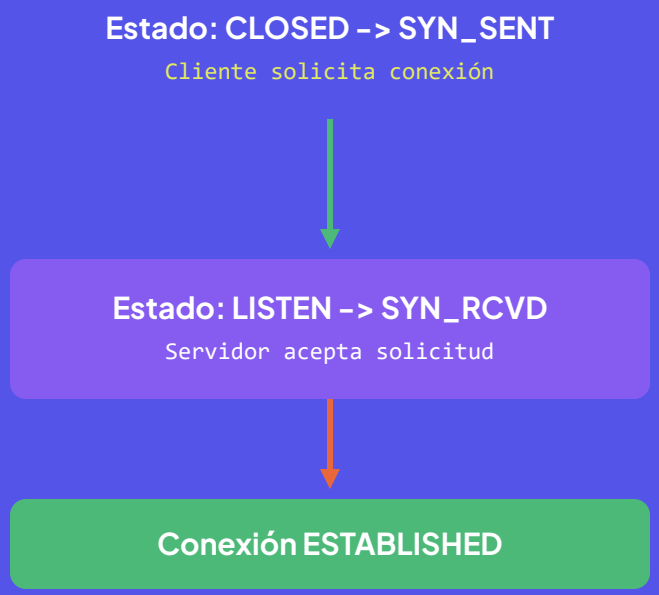
Protocolo TCP: Características Principales

Transmission Control Protocol

- **Orientado a Conexión:** Requiere negociación previa entre cliente y servidor.
- **Confiable & Ordenado:** Garantiza la entrega de cada byte en la secuencia exacta enviada.
- **Control de Flujo:** Ajusta la tasa de envío según la capacidad del receptor.

Esquema 3-Way Handshake





TCP: Establecimiento de Conexión (Handshake)

El proceso de **Handshake de 3 Vías** sincroniza los números de secuencia iniciales en ambas vías antes de transmitir datos de aplicación:

1. **Paso 1 (SYN):** El cliente envía un paquete con la bandera `SYN` y su número de secuencia inicial `ISN(c)`.
2. **Paso 2 (SYN-ACK):** El servidor responde confirmando la recepción con `ACK = ISN(c) + 1` y envía su propio `SYN` con `ISN(s)`.
3. **Paso 3 (ACK):** El cliente confirma con `ACK = ISN(s) + 1`. La socket TCP queda en estado `ESTABLISHED`.

Universidad ICESI

TCP: Transferencia de Datos y Control de Flujo

Una vez establecida la conexión, TCP garantiza la integridad del flujo mediante:

- **Sequence & Acknowledgment Numbers:** Cada byte enviado lleva un número de secuencia. El receptor confirma explícitamente los bytes recibidos.
- **Sliding Window (Ventana Deslizante):** El receptor notifica el tamaño del buffer disponible (`Window Size`) para evitar desbordamientos de memoria.
- **Retransmisión Automática:** Si un paquete no recibe un ACK antes del tiempo de expiración (RTT Timeout), TCP lo reenvía automáticamente.

Universidad ICESI

TCP: Cierre de Conexión (4-Way Handshake)

El cierre de conexión en TCP es dúplex e independiente para cada sentido mediante 4 pasos:

1. **1. FIN (Cliente):** El cliente solicita cerrar su canal de envío emitiendo la bandera `FIN`.
2. **2. ACK (Servidor):** El servidor confirma la recepción del `FIN`. El canal cliente $ ightarrow$ servidor queda cerrado.
3. **3. FIN (Servidor):** Cuando el servidor termina de enviar datos pendientes, emite su propio `FIN`.
4. **4. ACK (Cliente):** El cliente responde con `ACK` y la conexión se libera totalmente.

Protocolo UDP: Características y Versatilidad

User Datagram Protocol (UDP): Protocolo ligero sin conexión previa:

- **Sin Estado ni Handshake:** Envia datagramas inmediatamente sin esperar confirmaciones.
- **Baja Latencia:** Mínimo overhead de cabecera (8 bytes vs 20 bytes de TCP).
- **Casos de Uso Principales:** Streaming multimedia, juegos en línea, consultas DNS y llamadas VoIP (donde la velocidad prioriza sobre pérdidas mínimas).

Comparativa Directa: TCP vs UDP

TCP (Confiable y Ordenado)

- Conexión previa (Handshake 3 vías).
- Garantiza entrega y orden exacto.
- Mayor sobrecarga de cabecera (20B).
- Uso: Web (HTTP), SSH, DBs, Email.

UDP (Rápido sin Conexión)

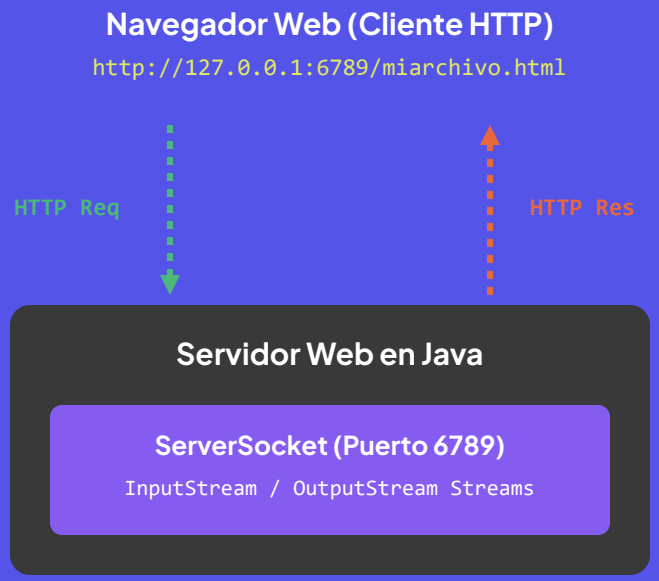
- Sin conexión ni establecimiento.
- No retransmite pérdidas ni controla flujo.
- Mínima sobrecarga (8 Bytes).
- Uso: Streaming, DNS, Videojuegos.

Universidad ICESI

El Protocolo HTTP/1.1 y sus Componentes

HTTP (Hypertext Transfer Protocol) es el protocolo de capa de aplicación usado en la Web sobre conexiones TCP (puerto 80/443):

- **Request Line:** Método (`GET` , `POST`), URI solicitada y versión de protocolo.
- **Headers:** Pares clave-valor (`Host` , `Content-Type` , `User-Agent`).
- **Separador CRLF ( ):** Línea en blanco obligatoria que indica el fin de los encabezados.
- **Body (Payload):** Datos enviados en POST/PUT o contenido retornado por el servidor.



Esquema Cliente–Servidor y Flujo HTTP

Interacción entre el Navegador Web (Cliente) y el Procesador Web mediante mensajes HTTP formateados sobre TCP:

- El navegador interpreta la URL `http://127.0.0.1:6789/index.html` y abre el socket TCP.
- Construye y envía el bloque de texto **HTTP Request** finalizado con .
- El servidor lee la petición, ubica el recurso y retorna la respuesta **HTTP Response** conteniendo las cabeceras y el archivo solicitado.

Universidad ICESI

02. Construcción de un Servidor Web Simple con Sockets Java

Implementación paso a paso de ServidorWebSimple.java

Implementación Java: Librerías y Estructura

Librerías Requeridas:

- `java.io.*`: Clases para manejo de flujos I/O (`BufferedReader`, `FileInputStream`, `DataOutputStream`).
- `java.net.*`: Clases de red para conexiones sockets (`ServerSocket`, `Socket`).
- `java.util.*`: Utilidades de procesamiento de texto como `StringTokenizer`.

```
// 1. Importar paquetes requeridos
import java.io.*;
import java.net.*;
import java.util.*;

// 2. Definir la clase principal
public class ServidorWebSimple {
  public static void main(String argv[]) throws Exception {
    // El código de conexión e I/O irá aquí
  }
}
```

Creación del Socket del Servidor

Socket de Escucha y Conexión:

- `ServerSocket socketdeEscucha = new ServerSocket(6789)`: Solicita al sistema operativo abrir el puerto pasivo 6789.
- `socketdeEscucha.accept()`: Suspende la ejecución del hilo hasta recibir un intento de conexión TCP entrante de un navegador.
- Retorna una instancia de `Socket` activa para comunicarse con ese cliente específico.

```
public class ServidorWebSimple {
  public static void main(String argv[]) throws Exception {

    // Crear el socket de escucha en puerto 6789
    ServerSocket socketdeEscucha =
      new ServerSocket(6789);

    // Bloquea hasta que se conecte un cliente
    Socket socketdeConexion =
      socketdeEscucha.accept();

  }
}
```

Lectura de la Solicitud HTTP del Cliente

Lectura de Streams de Entrada:

- `socketdeConexion.getInputStream()` : Obtiene el flujo de bytes crudos provenientes del socket TCP.
- `InputStreamReader` : Transforma los bytes crudos en caracteres de texto.
- `BufferedReader` : Utiliza un buffer interno en memoria para permitir la lectura eficiente línea a línea mediante `readLine()` .

```
// Obtener el flujo de entrada desde el socket de conexión
BufferedReader mensajeDesdeCliente =
  new BufferedReader(
    new InputStreamReader(socketdeConexion.getInputStream())
  );

// Leer la primera línea de la petición HTTP
// Ejemplo: "GET /miarchivo.html HTTP/1.1"
String lineaDeLaSolicitudHttp = mensajeDesdeCliente.readLine();
```

Procesamiento de la Solicitud HTTP

Parseo del Nombre del Archivo:

- `StringTokenizer`: Divide la línea de solicitud HTTP (ej. `GET /miarchivo.html HTTP/1.1`) usando espacios como delimitadores.
- Se verifica que el primer token sea el método `GET`.
- Se extrae el nombre del archivo y se remueve el carácter inicial `/` mediante `substring(1)`.

```
StringTokenizer lineaSeparada =
  new StringTokenizer(lineaDeLaSolicitudHttp);

if (lineaSeparada.nextToken().equals("GET")) {
  String nombreArchivo = lineaSeparada.nextToken();
  if (nombreArchivo.startsWith("/"))
    nombreArchivo = nombreArchivo.substring(1);

  // El código para leer y enviar el archivo irá aquí
} else {
  System.out.println("Bad Request Message");
}
```

Lectura de Archivo y Cabeceras HTTP

Construcción de la Respuesta HTTP:

- `FileInputStream`: Lee el archivo solicitado desde el disco en un arreglo `byte[]`.
- `DataOutputStream`: Escribe datos de respuesta en el socket hacia el cliente.
- Escribe la cabecera `HTTP/1.0 200 Document Follows` y detecta `Content-Type` (image/jpeg, gif).
- **Línea Vacía Obligatoria (`writeBytes(" ")`):** El protocolo HTTP exige un salto de línea en blanco para marcar el fin de los encabezados.

```
File archivo = new File(nombreArchivo);
FileInputStream archivoDeEntrada = new FileInputStream(nombreArchivo);
int cantidadDeBytes = (int) archivo.length();
byte[] archivoEnBytes = new byte[cantidadDeBytes];
archivoDeEntrada.read(archivoEnBytes);

DataOutputStream mensajeParaCliente =
  new DataOutputStream(socketdeConexion.getOutputStream());

mensajeParaCliente.writeBytes("HTTP/1.0 200 Document Follows
");
if (nombreArchivo.endsWith(".jpg"))
  mensajeParaCliente.writeBytes("Content-Type: image/jpeg
");
if (nombreArchivo.endsWith(".gif"))
  mensajeParaCliente.writeBytes("Content-Type: image/gif
");
mensajeParaCliente.writeBytes("Content-Length: " + cantidadDeBytes + "
");

// Salto de línea en blanco obligatorio antes del cuerpo
mensajeParaCliente.writeBytes("
");
```

Envío del Contenido y Ejecución del Servidor

Envío de Bytes y Cierre de Socket:

- `mensajeParaCliente.write(...)` transmite los bytes crudos del archivo solicitado al navegador.
- `socketdeConexion.close()` cierra el socket TCP cliente.
- **Compilación y Prueba:**

```
javac ServidorWebSimple.java
java ServidorWebSimple
# Probar: http://127.0.0.1:6789/miarchivo.html
```

```
// Envío del contenido binario del archivo
mensajeParaCliente.write(archivoEnBytes, 0, cantidadDeBytes);

// Cierre del socket de conexión TCP
socketdeConexion.close();


// Demostración de prueba en terminal:
// $ javac ServidorWebSimple.java
// $ java ServidorWebSimple
// Navegador: http://127.0.0.1:6789/index.html
```

¿Por qué el Servidor Unihilo no es suficiente?

Problema de Bloqueo Secuencial:

- En este diseño simple, el hilo principal `main` procesa la petición de un cliente a la vez.
- Si la descarga de un archivo grande o una red lenta toma 5 segundos, **todos los demás clientes quedan congelados en espera** en la cola del SO.
- Una aplicación web real necesita atender decenas o cientos de peticiones simultáneas sin bloqueo mutuo.

La Solución: Servidor Web Multihilos

Paradigmas Multihilo

Para solucionar el bloqueo, se delega cada socket cliente a un hilo independiente:

- **Thread por Petición:** Crear un `new Thread(worker).start()` por cada `accept()`.
- **Pool de Hilos (ExecutorService):** Reutilizar una piscina de hilos fija para prevenir el agotamiento de memoria RAM del servidor.

```
while (true) {
  Socket clientSocket = socketdeEscucha.accept();

  // Delegación concurrente en hilo worker
  new Thread(new ManejadorCliente(clientSocket)).start();
}
```

¿Por qué un Servidor Socket Manual no escala?

COMPLEJIDAD HTTP

Parsear manualmente cookies, sesiones, query params y multipart es propenso a vulnerabilidades.

MANEJO DE HILOS

Gestionar hilos y memoria manualmente genera riesgos de Memory Leaks y saturación de CPU.

ACOPLAMIENTO

Mezclar código de sockets TCP con lógica HTML imposibilita la mantenibilidad.

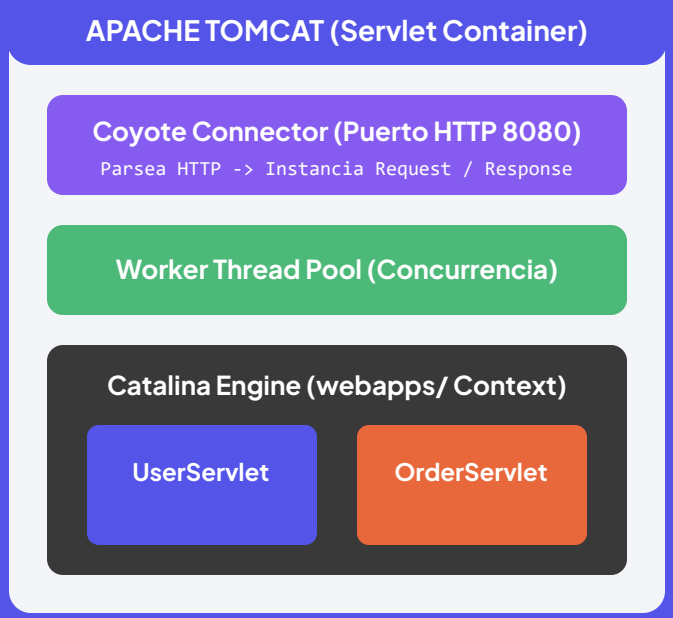
FALTA DE ESTÁNDAR

Reinventar la rueda en seguridad y enrutamiento impide la integración de frameworks.

Universidad ICESI

03. Apache Tomcat y el Modelo de Java Servlets

Abstracción de red, Contenedores Web y Java Servlets



Apache Tomcat: El Contenedor de Servlets

Apache Tomcat es el Web Container / Servlet Engine open-source por excelencia en el ecosistema Java:

- **Coyote Connector:** Se encarga de abrir los sockets TCP, escuchar en los puertos e interpretar las cabeceras HTTP.
- **Catalina Engine:** Administra el contexto de las aplicaciones web y la piscina de hilos.
- **Abstracción:** Transforma peticiones TCP en objetos Java `HttpServletRequest` y `HttpServletResponse`.

¿Qué es un Java Servlet?

Definición y Rol Principal

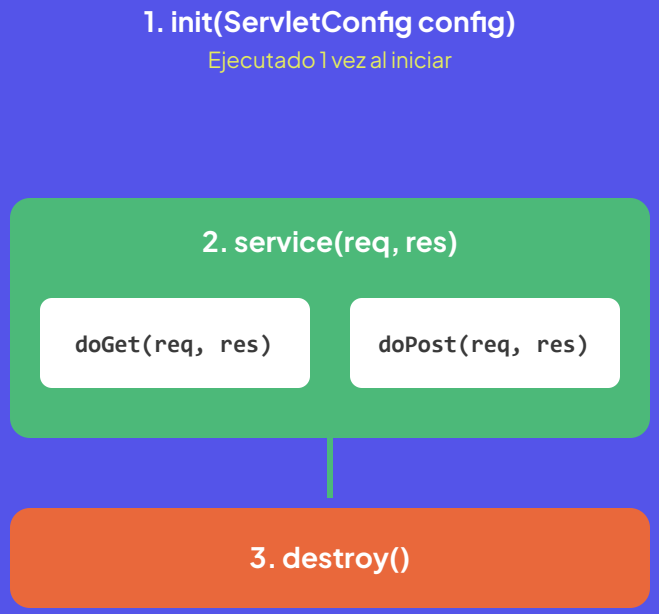
Un **Servlet** es una clase Java que extiende `HttpServlet` y se ejecuta dentro de Tomcat para responder a peticiones dinámicas.

- Recibe la petición procesada por el contenedor.
- Ejecuta la lógica de negocio o BD.
- Retorna una respuesta formateada.

Ventajas de la Abstracción

- **Desacoplamiento:** Olvídate de manejar sockets o lectura de bytes.
- **Eficiencia:** Una única instancia en memoria atiende múltiples hilos concurrentes.
- **Estándar:** Base de Spring MVC y Jakarta EE.

Ciclo de Vida de un Servlet (Lifecycle)



Tomcat administra estrictamente las 3 etapas del ciclo de vida del Servlet:

1. **1. init(ServletConfig):** Se ejecuta **UNA SOLA VEZ** al cargar la clase en memoria.
2. **2. service():** Se ejecuta **REPETIDAMENTE** en cada petición del usuario (dirige a `doGet` / `doPost`).
3. **3. destroy():** Se ejecuta **UNA SOLA VEZ** al apagar Tomcat.

Mapeo de Servlets: @WebServlet vs web.xml

Anotación Moderna: @WebServlet

Servlet 3.0+ permite declarar la ruta directamente sobre la clase:

```
@WebServlet(
  name = "MiServlet",
  urlPatterns = {"/saludo"}
)
public class HolaServlet
    extends HttpServlet { ... }
```

Descriptor Clásico: web.xml

Configuración en archivo `WEB-INF/web.xml`:

```
<servlet>
 <servlet-name>HolaServlet</servlet-name>
 <servlet-class>com.icesi.HolaServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>HolaServlet</servlet-name>
 <url-pattern>/saludo</url-pattern>
</servlet-mapping>
```

Implementación de un HttpServlet

Estructura del Código:

- `HttpServletRequest req`: Acceso a parámetros (`req.getParameter()`) y headers.
- `HttpServletResponse res`: Configuración del tipo de respuesta (`res.setContentType()`).
- `PrintWriter out = res.getWriter()`: Stream de emisión de HTML.

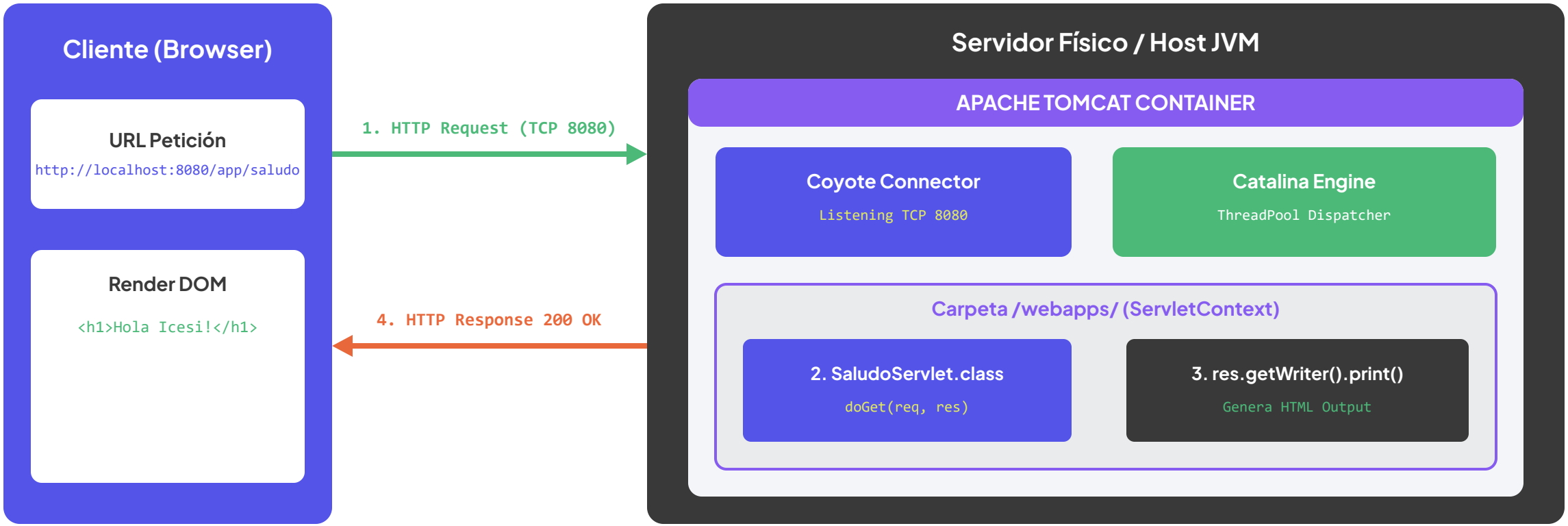
```
import java.io.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;
import jakarta.servlet.annotation.*;

@WebServlet("/saludo")
public class SaludoServlet extends HttpServlet {
  @Override
  protected void doGet(HttpServletRequest req,
            HttpServletResponse res)
            throws ServletException, IOException {

    res.setContentType("text/html;charset=UTF-8");
    PrintWriter out = res.getWriter();
    String usuario = req.getParameter("nombre");

    out.println("<h1>Hola " +
      (usuario != null ? usuario : "Invitado") + "!</h1>");
  }
}
```

Arquitectura de Extremo a Extremo: Cliente a Tomcat



Resumen y Conclusiones de la Semana 1

PROTOCOLOS WEB

TCP garantiza transporte confiable; HTTP estructura la semántica cliente-servidor.

SOCKETS A BAJO NIVEL

`ServerSocket` demuestra el funcionamiento interno de I/O y puertos TCP.

APACHE TOMCAT

Resuelve la concurrencia masiva y el parseo HTTP liberando al desarrollador.

JAVA SERVLETS

Estándar robusto para lógica dinámicas y cimiento de frameworks modernos.

¿Preguntas o Dudas sobre la Semana 1?

Computación en Internet 2 — Universidad Icesi