



**Universidad Mariano Gálvez de Guatemala**  
**Facultad de Ingeniería en Sistemas de la Información**

**Redes de Computadoras**

**Proyecto Final**

**Integrantes del grupo**

Leslie Mariela Jiménez García                    0907-22-17016

José Carlos Cabrera Sandoval                    0907-22-9576

Jorge Antonio Hernández Nájera                0907-20-23870

## **Introducción**

En el estudio de las redes de computadoras, los modelos OSI y TCP/IP son esenciales para comprender la arquitectura de la comunicación. Sin embargo, su naturaleza teórica dificulta la visualización del flujo de datos. Por ello se desarrolló "NetSim Pro", que permite observar de forma práctica cómo se encapsula y desencapsula la información entre computadoras.

## **Objetivos**

### **Objetivos Generales:**

- Poner en práctica los conocimientos generales adquiridos durante el curso, aplicándolos en un tema específico relacionado con la materia de Redes de Computadoras II.
- Incentivar el conocimiento del flujo de los datos y la información agregada en cada fase del Modelo OSI, comprendiendo su rol como base de la transmisión de datos en una red.

### **Objetivos Específicos**

- Desarrollar una aplicación web funcional (cliente-servidor) que simule la comunicación en red entre múltiples dispositivos en tiempo real.
- Implementar un servidor backend con Node.js y Socket.IO capaz de gestionar la creación de salas privadas y la retransmisión de mensajes a destinos específicos.
- Habilitar la transferencia de tres tipos de datos distintos: **Texto, Imágenes y Video**, cumpliendo con los requisitos del proyecto.

## **Descripción del Proyecto**

NetSim Pro es una aplicación web que simula una red local (LAN) virtual. Un servidor central gestiona múltiples clientes conectados a una misma sala, permitiendo el envío de mensajes o archivos entre ellos. Los usuarios pueden observar el proceso de encapsulación y desencapsulación de los datos representando el modelo OSI.

## **Arquitectura y Tecnologías Utilizadas**

### **Backend:**

- Node.js como entorno de ejecución.
- Express.js para manejar rutas HTTP.
- Socket.IO para la comunicación en tiempo real.

### **Frontend:**

- HTML5, CSS3 y JavaScript para la interfaz.
- Manejo de archivos con FileReader, Blob y URL.createObjectURL para mostrar videos e imágenes recibidas.

## **Desafíos Técnicos y Soluciones**

Desafío 1: no sacar a mas usuario de la sala ya los sacaba

Solución estaba configurado solo para dos usuarios lo cual debimos modificar el código

Desafío 2: Reproducción de videos.

Solución: Uso de ArrayBuffer y Blob para manejo eficiente de datos binarios.

Desafío 3: Reinicio automático del servidor.

Solución: Configurar nodemon.json para ignorar la carpeta data.

## **Funcionamiento del Proyecto "NetSim Pro"**

El "NetSim Pro" opera bajo una arquitectura Cliente-Servidor, donde el server.js (basado en Node.js) actúa como el servidor central, y cualquier navegador que accede a <http://localhost:3001> se convierte en un cliente.

La comunicación se divide en dos fases:

Conexión HTTP (POST/GET): Se usa solo para la configuración inicial de la sala (crear o unirse).

Conexión WebSocket (Socket.IO): Se usa para toda la comunicación en tiempo real (unirse, notificar a otros, y transmitir los datos).

A continuación, se detalla el flujo completo, paso a paso:

### **Fase 1: Conexión y Establecimiento de la Sala**

Este proceso simula cómo dos computadoras se "descubren" en una red para poder comunicarse.

#### **Carga Inicial (Cliente A):**

Un usuario (Cliente A) abre <http://localhost:3001> en su navegador.

El servidor Express (server.js) recibe esta solicitud GET y le responde enviándole los archivos estáticos de la carpeta public/ (principalmente index.html, styles.css y script.js).

El navegador del Cliente A carga el script.js, el cual inmediatamente ejecuta `socket = io()`, estableciendo una conexión WebSocket persistente con el servidor. En la terminal del servidor, esto se ve como  Nueva conexión: [socket.id].

#### **Creación de la Sala (Cliente A):**

El Cliente A escribe su nombre (ej. "PC-Principal") y hace clic en "Crear Sala & Transmitir".

El script.js NO usa WebSockets para esto. En su lugar, realiza una solicitud HTTP POST a la ruta /create-room del servidor.

El servidor (server.js) recibe esta solicitud, genera un ID de sala aleatorio (ej. H5RX18), y almacena esta sala en un Map en su memoria.

El servidor responde al Cliente A con un JSON { success: true, roomId: "H5RX18" }.

### **Unión a la Sala (Cliente A):**

Al recibir la respuesta HTTP exitosa, el script.js del Cliente A emite un evento WebSocket al servidor: `socket.emit('join-room', { roomId: "H5RX18", computerName: "PC-Principal" })`

El servidor recibe este evento, une a este socket a la "sala" (`socket.join(roomId)`), y almacena la relación entre el `socket.id` y el nombre "PC-Principal" en el Map de conexiones.

El servidor llama a `updateRoomInfo(roomId)`, que emite un evento `room-update` a todos en la sala. Como el Cliente A es el único, él mismo recibe el evento y su interfaz cambia de "Conexión" al "Dashboard" principal.

### **Unión del Segundo Cliente (Cliente B):**

Un segundo usuario (Cliente B, en otro navegador o un celular) abre <http://localhost:3001>.

Escribe su nombre (ej. "Laptop-Secundaria") y el código de sala H5RX18, y hace clic en "Unirse a Sala".

Su script.js realiza una solicitud HTTP POST a `/join-room`. El servidor verifica que la sala H5RX18 existe. Como sí existe, responde con `{ success: true }`.

Al recibir la respuesta, el script.js del Cliente B emite el evento WebSocket `socket.emit('join-room', ...)`.

### **El servidor une al Cliente B a la sala H5RX18**

El servidor vuelve a llamar a `updateRoomInfo(roomId)`. Esta vez, emite `room-update` a todos en la sala (PC-Principal y Laptop-Secundaria).

### **El Cliente B recibe el evento y carga el "Dashboard".**

El Cliente A (que ya estaba en el Dashboard) recibe el evento y su lista de "Computadora Destino" se actualiza automáticamente para mostrar a "Laptop-Secundaria".

## **Fase 2: El Proceso de Transmisión (El Núcleo del Proyecto)**

Aquí es donde se cumple el requisito de "ejemplificar... el proceso de transmisión" y se manejan los diferentes tipos de datos. Supongamos que "PC-Principal" (Cliente A) quiere enviar un video a "Laptop-Secundaria" (Cliente B).

Preparación (Cliente A - Emisor):

El Cliente A va a la pestaña "Transmisión".

Selecciona "Laptop-Secundaria" como destino.

Selecciona "Video" como tipo de dato.

Arrastra un archivo de video (ej. mi\_video.mp4 de 10MB).

El script.js utiliza la API FileReader del navegador. De forma crucial, no usa readAsDataURL (Base64), lo cual es ineficiente y fallaría.

En su lugar, usa reader.readAsArrayBuffer(file). Esto lee el archivo en su formato binario crudo, que es mucho más eficiente y es la forma moderna de manejar archivos.

### **Encapsulación y Envío (Cliente A y Servidor):**

Cuando el FileReader termina, el ArrayBuffer (los 10MB de datos binarios) se empaqueta en un objeto transmissionData.

El Cliente A emite un único evento WebSocket al servidor: socket.emit('start-transmission', { transmissionData, targetComputer: "Laptop-Secundaria" }).

Al mismo tiempo, el Cliente A inicia su simulación visual local (simulateEncapsulation()), mostrando en su pantalla cómo los datos "pasan" por las 7 capas del modelo OSI, desde Aplicación hasta Física, añadiendo logs a su panel.

### **Enrutamiento (Servidor):**

El servidor recibe el evento start-transmission. Gracias a que configuramos maxHttpBufferSize a 50MB, el servidor acepta el paquete de 10MB sin desconectar al cliente.

El servidor mira el targetComputer: "Laptop-Secundaria". Busca en su Map de conexiones qué socket.id corresponde a "Laptop-Secundaria". Encuentra el socket.id (ej. zXyV...).

Actuando como un enrutador, el servidor reenvía el paquete completo (con el ArrayBuffer) pero solo a ese socket específico, usando un nuevo nombre de evento: io.to(targetSocketId).emit('receive-transmission', { ...datos... })

#### **Recepción y Desencapsulación (Cliente B - Receptor):**

El script.js del Cliente B, que estaba inactivo, recibe de repente el evento receive-transmission con la carga de 10MB.

Inicia su propia simulación visual (simulateDecapsulation()), mostrando cómo los datos "suben" por las capas, desde Física hasta Aplicación, llenando su panel de "Desencapsulación".

Una vez que la simulación llega a la Capa 7, llama a la función displayReceivedData().

#### **Renderizado del Video (Cliente B):**

Esta es la parte final. El Cliente B tiene ahora el ArrayBuffer del video en su memoria.

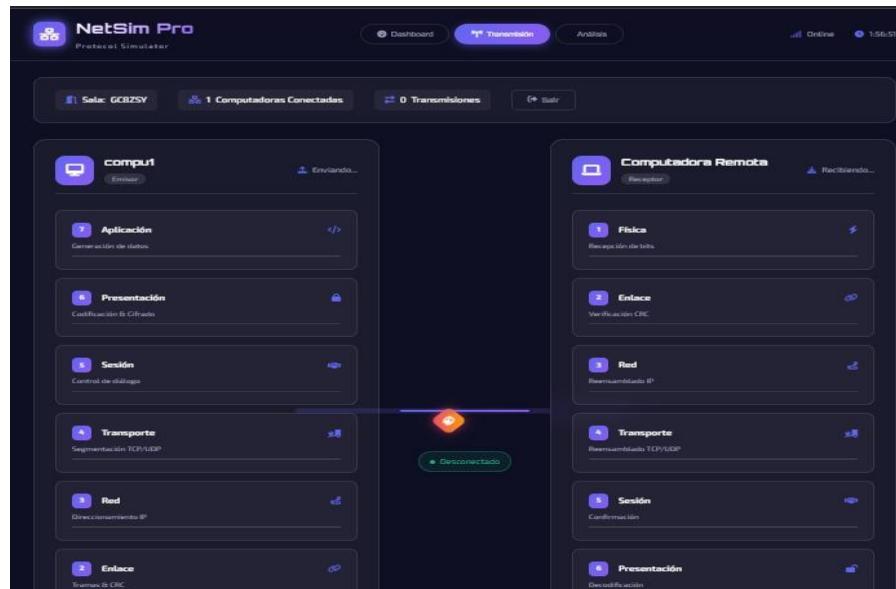
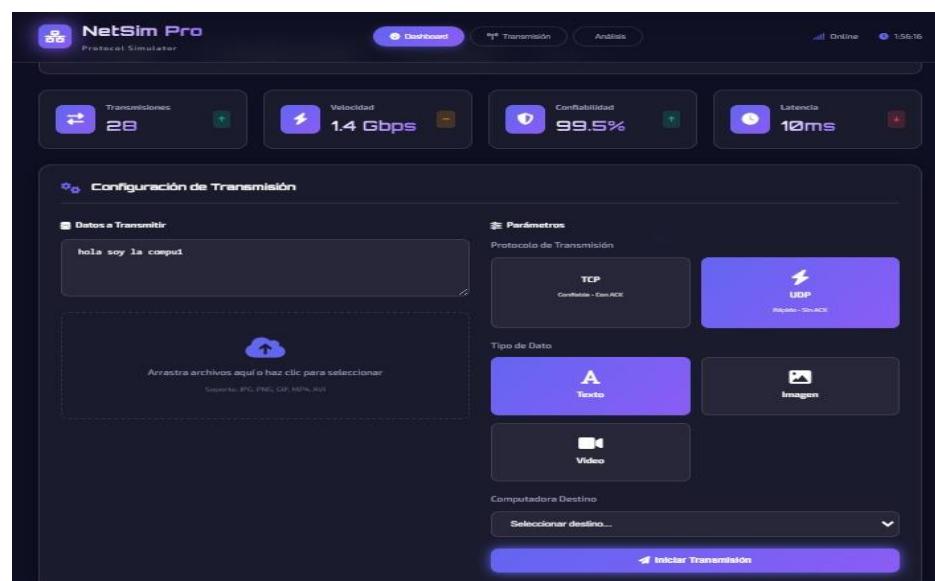
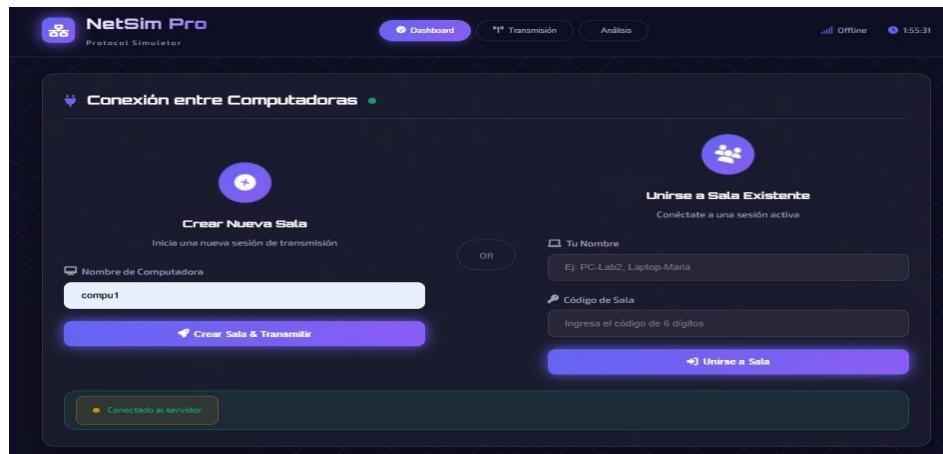
Para que el navegador pueda reproducirlo, la función displayReceivedData realiza dos pasos críticos:

Crea un Blob (Binary Large Object) a partir del ArrayBuffer y el tipo de archivo (ej. video/mp4) que también venía en los metadatos.

Crea un Object URL (ej. blob:http://localhost:3001/abc-123...) usando URL.createObjectURL(blob). Esta es una URL local temporal que apunta directamente al dato en la memoria del navegador.

Finalmente, el script crea un elemento <video>, establece su src a esta Object URL, le añade controles (controls = true) y lo inserta en la pestaña "Análisis".

El Cliente B ahora ve un reproductor de video y puede darle play al mi\_video.mp4 que el Cliente A le envió, completando la transmisión de extremo a extremo.



## **Conclusión**

El proyecto "NetSim Pro" cumplió con los objetivos planteados al simular de forma funcional el proceso de transmisión de datos mediante las capas del modelo OSI. La experiencia permitió comprender los aspectos técnicos y prácticos del flujo de datos, el manejo de paquetes y la comunicación en tiempo real entre dispositivos.