**ÍNDICE**

[**Antecedentes** 2](#_Toc191840935)

[Comunicación Cliente-Servidor en Redes 2](#_Toc191840936)

[**Planteamiento del problema** 2](#_Toc191840937)

[**Propuesta de solución** 3](#_Toc191840938)

[**Materiales y métodos empleados** 3](#_Toc191840939)

[**Desarrollo de la solución** 3](#_Toc191840940)

[**Resultados** 4](#_Toc191840941)

[Ejemplo de salida en la consola del Cliente 5](#_Toc191840942)

[Ejemplo de salida en la consola del Servidor 5](#_Toc191840943)

[**Conclusión** 6](#_Toc191840944)

# **Antecedentes**

La transferencia de archivos a través de redes es una funcionalidad fundamental en aplicaciones distribuidas. Existen múltiples protocolos para este propósito, como FTP y HTTP, pero en esta práctica se implementa un sistema de transferencia de archivos basado en Sockets TCP en Java.

En redes de computadoras, los Sockets permiten la comunicación entre procesos a través de una conexión establecida entre un cliente y un servidor. En este caso, el servidor maneja las solicitudes de un solo cliente a la vez, estableciendo una comunicación dedicada con cada usuario antes de cerrar la conexión y aceptar un nuevo cliente. Este enfoque facilita la comprensión del modelo de comunicación en redes, pero es una solución limitada en comparación con sistemas modernos que permiten múltiples conexiones simultáneas.

Si bien este tipo de arquitectura de un solo cliente por servidor resulta útil para aprender los fundamentos de los sockets y la transferencia de archivos, en la actualidad la mayoría de los sistemas implementan modelos multicliente y multiservidor para mejorar la eficiencia y escalabilidad. Estos modelos permiten que varios clientes interactúen con un servidor al mismo tiempo o incluso que múltiples servidores trabajen en conjunto dentro de un sistema distribuido. Sin embargo, la implementación de un modelo básico uno a uno sigue siendo una base esencial para el estudio y desarrollo de arquitecturas de red más avanzadas.

## Comunicación Cliente-Servidor en Redes

Un socket es un punto final de comunicación bidireccional entre dos dispositivos en una red. En este modelo:

* El servidor escucha en un puerto específico y espera conexiones de clientes.
* El cliente se conecta al servidor y solicita un archivo.
* El servidor verifica si el archivo existe y lo envía en paquetes de datos hasta completar la transferencia.

Para esta práctica, se utilizó TCP, un protocolo que garantiza la entrega ordenada y completa de los datos, asegurando la integridad del archivo.

# **Planteamiento del problema**

En muchas aplicaciones es necesario transferir archivos entre dispositivos en una red. Existen diversas maneras de hacerlo, pero en esta práctica se busca desarrollar una solución personalizada basada en Sockets en Java.

El problema es cómo permitir que un cliente pueda solicitar un archivo específico a un servidor y recibirlo sin pérdida de datos, asegurando la integridad de la información.

# **Propuesta de solución**

Para resolver este problema, se diseñó un sistema con los siguientes componentes:

1. Servidor de archivos: Este recibe solicitudes de un cliente y envía el archivo solicitado si está disponible.
2. Cliente: Se conecta al servidor y solicita un archivo específico.
3. Transferencia con buffers: Se usa un buffer de 4096 bytes para enviar los datos en bloques y mejorar la eficiencia de la transmisión.
4. Mensajes de confirmación: Antes de comenzar la transmisión para indicar si el archivo existe.

# **Materiales y métodos empleados**

* Lenguaje de programación: **Java**.
* Entorno de desarrollo: Visual Studio Code.
* Librerías utilizadas:
  + java.io.\* para la manipulación de archivos.
  + java.net.\* para la comunicación por sockets
* Métodos empleados:
  + Creación de sockets TCP en Java
  + Uso de DataInputStream y DataOutputStream para la comunicación
  + Manejo de archivos con FileInputStream y FileOutputStream

# **Desarrollo de la solución**

1. Primero se hablará de la configuración del servidor. Se creó un servidor en Java que inicia la escucha en el puerto 5000. Se esperan las conexiones del cliente y se recibe el nombre del archivo solicitado. Si el archivo existe, lo envía en fragmentos de 4096 bytes; de lo contrario se envía un mensaje de error al cliente.
2. Posteriormente se toca el tema de la configuración del cliente. Se creó un cliente que se conecta al servidor IP 127.0.0.1. El usuario ingresa el nombre del archivo a solicitar. Si el archivo existe, se recibe y almacena en la carpeta archivos\_cliente. Si el archivo no existe se muestra un mensaje de error.
3. Finalmente, en cuanto a las pruebas realizadas se ejecuta de la siguiente manera. Primero se ejecuta el servidor y el cliente en la misma máquina. Se solicitan los archivos

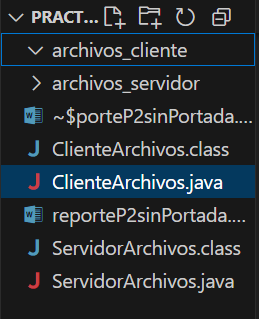
Todos los códigos utilizados en esta práctica se encuentran en el siguiente repositorio en Github.

<https://github.com/EmilianoLedesma16/PracticaDos.git>

# **Resultados**

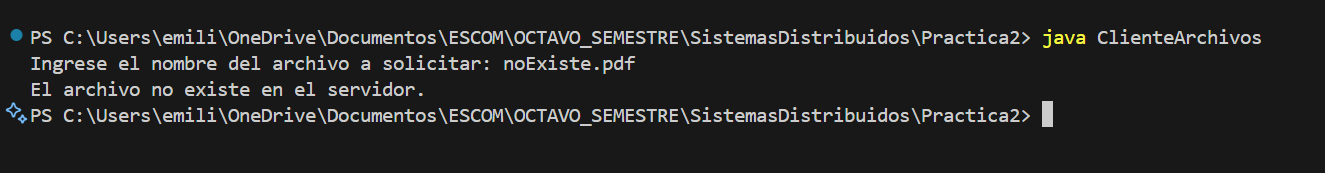
Se hicieron dos pruebas para comprobar el correcto funcionamiento del sistema. Los archivos por transferirse fueron prueba.pdf y texto.txt para verificar la comunicación con diferentes archivos.

* La transferencia de archivos fue exitosa sin pérdida de datos.
* El servidor pudo manejar múltiples solicitudes secuenciales, aunque solo acepta un cliente a la vez.
* El cliente recibió correctamente los archivos y los guardó en la carpeta archivos\_cliente.

Texto

El contenido generado por IA puede ser incorrecto.

* El sistema detectó correctamente archivos inexistentes.

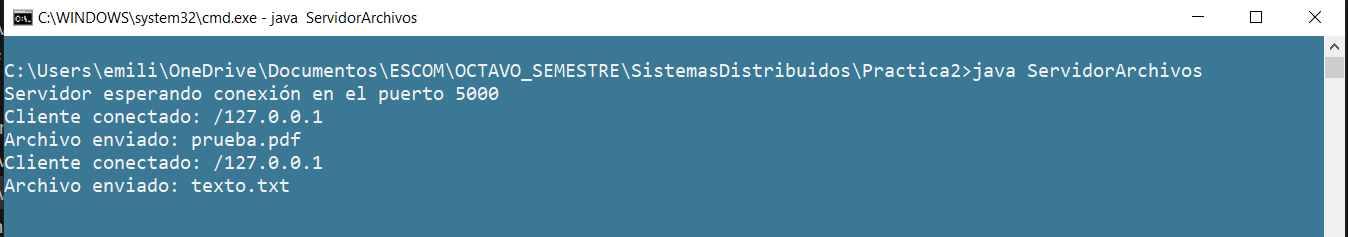


## Ejemplo de salida en la consola del Cliente

Texto

El contenido generado por IA puede ser incorrecto.

## Ejemplo de salida en la consola del Servidor



# **Conclusión**

En esta práctica se desarrolló una aplicación cliente-servidor basada en sockets TCP, en la cual la comunicación se establece en un esquema uno a uno, donde solo un cliente puede conectarse al servidor en cada instancia. Si bien este enfoque resulta adecuado para comprender los fundamentos de la comunicación en redes y la transferencia de archivos, en la actualidad los sistemas de comunicación han evolucionado hacia arquitecturas más avanzadas.

Actualmente, la mayoría de las aplicaciones utilizan modelos multicliente y multiservidor, en los cuales varios clientes pueden interactuar con un mismo servidor de manera simultánea o, incluso, acceder a múltiples servidores en un entorno distribuido.

Este tipo de arquitectura permite mejorar el rendimiento, la escalabilidad y la disponibilidad del sistema, adaptándose mejor a las necesidades de aplicaciones modernas como almacenamiento en la nube, servicios de transmisión en tiempo real y plataformas de mensajería.

En este sentido, aunque la solución desarrollada en esta práctica cumple su objetivo dentro de un esquema básico, representa una base para la construcción de sistemas más complejos. Implementaciones futuras podrían incluir la gestión de múltiples conexiones simultáneas, el uso de hilos para permitir concurrencia y la optimización del manejo de archivos en servidores con mayor capacidad de respuesta.

Finalmente se menciona que la transición hacia sistemas distribuidos permitiría una mejor tolerancia a fallos y una distribución eficiente de los recursos, aspectos clave en el desarrollo de software actual.