“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERÍA ESTADÍSTICA E INFORMÁTICA

ESCUELA PROFESIONAL DE INGENIERÍA ESTADÍSTICA E INFORMÁTICA

PROTOCOLOS DE COMUNICACIÓN Y MIDDLEWARE EN SISTEMAS DISTRIBUIDOS: UN ANÁLISIS DE TECNOLOGÍAS Y APLICACIONES CON RPC Y RMI

PRESENTADO POR:

CRISTIAN DANIEL CCOPA ACERO

CURSO:

SISTEMAS DISTRIBUIDOS / VII Semestre – A

DOCENTE:

Ing. EDSON DENIS ZANABRIA

Puno, octubre 2024

INDICE

I. RESUMEN

En esta monografía, exploramos los conceptos fundamentales detrás de los **protocolos de comunicación** y el **middleware** en sistemas distribuidos, además de cómo estos dos elementos ayudan a que las aplicaciones modernas funcionen de manera eficiente. Los **protocolos de comunicación** son como las reglas que permiten que las computadoras, dispositivos y servidores puedan "hablar" entre sí y compartir información de manera segura y confiable. Sin estas reglas, sería imposible para las diferentes partes de una red distribuida intercambiar datos y coordinarse. A lo largo de la monografía, se explican algunos de los protocolos más comunes, como **TCP/IP**, **UDP** y **HTTP**, y cómo cada uno de ellos tiene sus propias características dependiendo de la velocidad, la seguridad y la confiabilidad que se necesiten en cada aplicación.

El **middleware**, por su parte, juega un rol clave como "intermediario" entre las aplicaciones y la red que las conecta. En lugar de que cada aplicación tenga que lidiar con los detalles técnicos de la red, el middleware se encarga de gestionar esas complejidades. Gracias a esto, los desarrolladores pueden enfocarse en el diseño y funcionamiento de sus aplicaciones sin preocuparse por cómo exactamente los datos viajarán entre los diferentes servidores o dispositivos.

Para mostrar cómo todo esto se junta en la práctica, también se implementa un ejemplo de una aplicación cliente-servidor utilizando **RPC** (Remote Procedure Call) y **RMI** (Remote Method Invocation). Estas tecnologías permiten que un programa en una computadora pueda ejecutar funciones en otra computadora de manera remota, como si esas funciones estuvieran en la misma máquina. Todo esto se explica paso a paso, con ejemplos de código sencillos que ilustran cómo se implementa y cómo funciona realmente.

II. INTRODUCCIÓN

En el mundo moderno, las tecnologías que usamos a diario dependen en gran medida de los **sistemas distribuidos**. Estos sistemas permiten que varias computadoras, ubicadas en diferentes lugares, trabajen juntas como si fueran una sola. Esto es esencial para aplicaciones como sitios web, servicios de streaming, sistemas bancarios y muchas otras herramientas que utilizamos todos los días sin darnos cuenta de la complejidad que hay detrás. Pero, ¿cómo se logra que estas computadoras se comuniquen entre sí de manera eficiente? Aquí es donde entran en juego los **protocolos de comunicación** y el **middleware**.

Los **protocolos de comunicación** son las reglas que garantizan que las computadoras en una red puedan intercambiar datos de manera segura y eficiente. Piensa en ellos como el "idioma" que las máquinas utilizan para enviarse mensajes. Dependiendo de lo que necesiten hacer, pueden usar diferentes tipos de protocolos. Por ejemplo, cuando ves una película en streaming, el sistema utiliza un protocolo rápido, como **UDP**, para que la transmisión no se detenga, incluso si se pierden algunos datos. En cambio, cuando envías un correo electrónico o realizas una compra en línea, se usan protocolos más seguros como **TCP/IP** para asegurarse de que toda la información llegue correctamente.

El **middleware** es otra pieza clave en los sistemas distribuidos. Actúa como un "mediador" que facilita la comunicación entre las diferentes aplicaciones que corren en la red. Gracias al middleware, los desarrolladores no tienen que preocuparse por detalles complicados como la ubicación de las computadoras o la forma en que los datos viajan entre ellas. En lugar de eso, el middleware se encarga de gestionar esas tareas, permitiendo que las aplicaciones funcionen de manera fluida y coordinada.

Además, esta monografía se enfoca en cómo se pueden implementar aplicaciones distribuidas utilizando **RPC (Remote Procedure Call)** y **RMI (Remote Method Invocation)**. Estas tecnologías permiten que una aplicación, ubicada en un dispositivo, pueda ejecutar funciones que están en otro dispositivo remoto, como si estuvieran en el mismo lugar. Esto es extremadamente útil cuando hablamos de aplicaciones cliente-servidor, donde la lógica del negocio se encuentra en un servidor, pero el usuario accede a ella desde su computadora o teléfono.

A lo largo de esta monografía, exploraremos en detalle cómo los **protocolos de comunicación** y el **middleware** hacen posible la creación de sistemas distribuidos eficientes, y cómo **RPC** y **RMI** simplifican la implementación de aplicaciones distribuidas que pueden trabajar en diferentes máquinas sin que el usuario lo note.

III. PROTOCOLOS DE COMUNICACIÓN EN SISTEMAS DISTRIBUIDOS

Los **protocolos de comunicación** son como las reglas que permiten que diferentes dispositivos y sistemas se comuniquen entre sí en una red. Imagina que cada dispositivo, ya sea una computadora, teléfono o servidor, tiene su propio "lenguaje". Para que estos dispositivos puedan entenderse y trabajar juntos, necesitan seguir un conjunto de reglas, como cuando dos personas hablan el mismo idioma. En los sistemas distribuidos, donde varios nodos (dispositivos o computadoras) colaboran para completar una tarea, los protocolos de comunicación son esenciales para garantizar que la información llegue de un punto a otro de manera efectiva, rápida y segura.

3.1.¿Qué son los Protocolos de Comunicación?

En un **sistema distribuido**, tenemos varios dispositivos repartidos geográficamente (o incluso localmente) que necesitan comunicarse para realizar una tarea conjunta. Los **protocolos de comunicación** son esos acuerdos previos sobre cómo intercambiar datos entre los diferentes nodos. Si uno de los nodos no sigue las reglas, la información no se intercambiará correctamente, o peor, la comunicación fallará por completo.

Vamos a verlo como si fuera una llamada telefónica entre dos amigos. Para que la llamada funcione, ambos necesitan estar de acuerdo en qué idioma usar, cuándo hablar, y esperar a que el otro termine de hablar antes de responder. Si uno de ellos grita o habla al mismo tiempo, la llamada se vuelve confusa. En los sistemas distribuidos, los protocolos de comunicación aseguran que todo funcione de manera coordinada.

Los protocolos no solo se ocupan de "cómo" se comunican los dispositivos, sino también de **qué hacer cuando hay errores**, **cómo manejar los datos** y **cómo asegurarse de que toda la información llegue** de manera correcta.

3.2.¿Cómo funcionan los protocolos de comunicación?

Para entender mejor cómo funcionan los protocolos, pensemos en la transmisión de datos como si fuera el envío de una carta. Tú eres el remitente y deseas enviar un mensaje a un amigo que está lejos:

Dirección: Primero necesitas la dirección de tu amigo. En el mundo digital, esto es la **IP** del dispositivo de destino. La **IP** es como la dirección física de una casa, solo que en Internet.

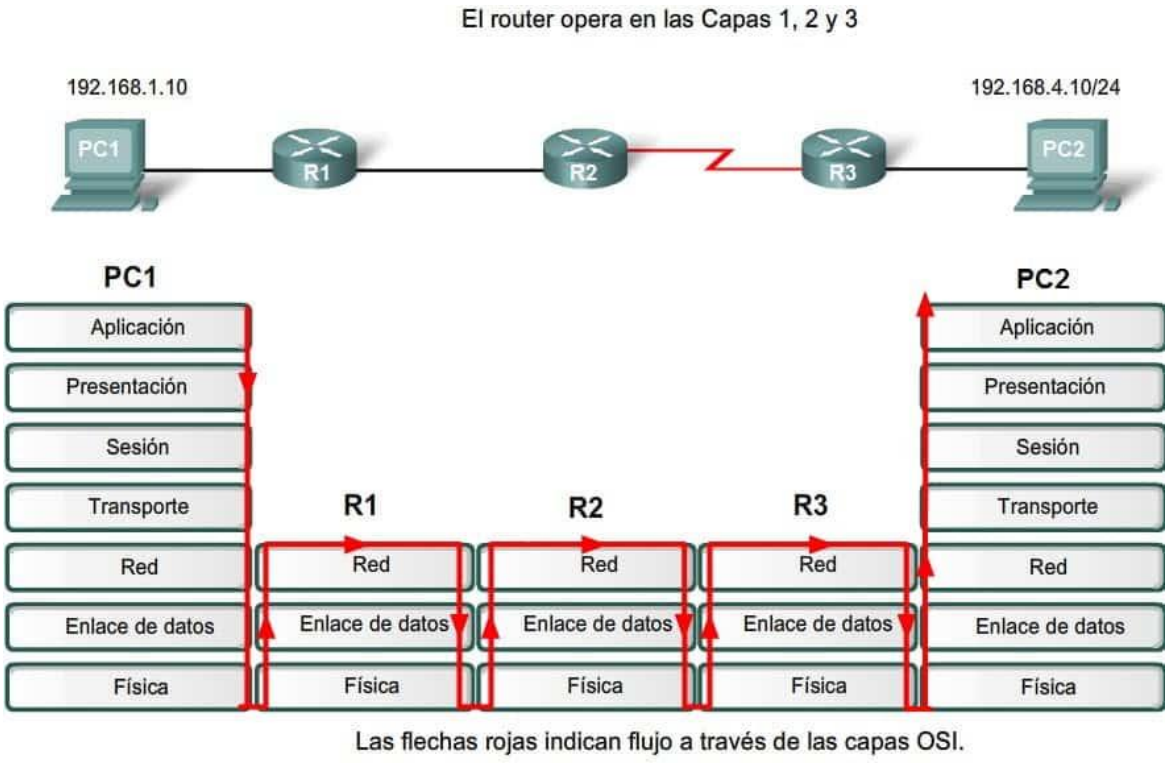
Envío del mensaje: Luego, escribes tu carta, que sería el mensaje que deseas enviar. En una red, este mensaje puede ser cualquier cosa: desde un archivo, una solicitud de información,

hasta un simple saludo. El protocolo decide **cómo dividir** este mensaje en partes más pequeñas para que sea más fácil de enviar (esto se llama "paquetes").

Seguimiento de la entrega: Imagina que en el camino hay varias oficinas de correos. El protocolo decide por dónde pasa tu carta para llegar a su destino. Cada oficina de correos (en términos de red, los "routers") ayuda a que tu mensaje llegue a su destino.

Confirmación de entrega: Finalmente, cuando tu amigo recibe la carta, puede enviar una confirmación de que la ha recibido correctamente. En términos de red, algunos protocolos, como **TCP**, aseguran que el mensaje ha llegado completo y en el orden correcto.

Este proceso puede sonar simple, pero cuando se manejan millones de mensajes a la vez (como cuando ves un video o haces una llamada de Zoom), la coordinación de los protocolos se vuelve fundamental para evitar que los mensajes se pierdan o se mezclen.



3.3.Principales Protocolos Utilizados

En los sistemas distribuidos, hay varios protocolos que se utilizan dependiendo del tipo de tarea y los requisitos de la red (como velocidad, confiabilidad y seguridad). A continuación, se describen los más comunes

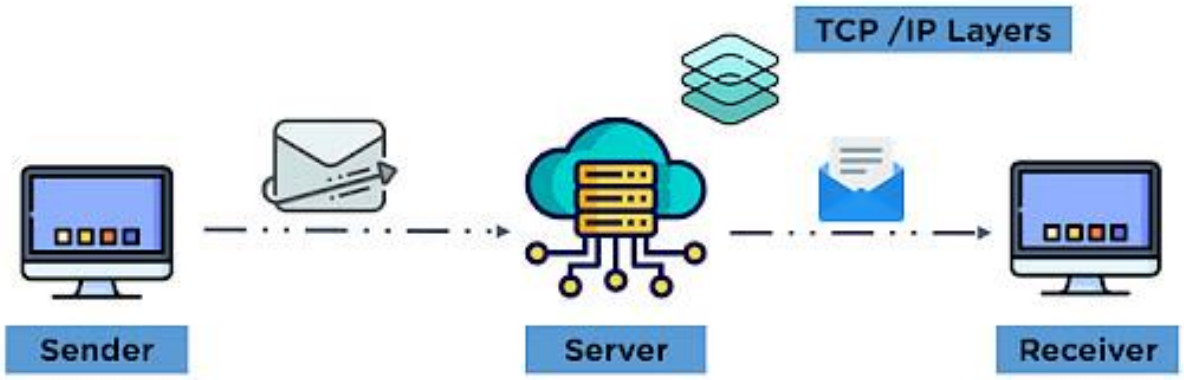
3.3.1. TCP/IP (Transmission Control Protocol / Internet Protocol)

TCP/IP es probablemente el protocolo más conocido y utilizado en los sistemas distribuidos. **TCP** es responsable de asegurarse de que los datos lleguen de manera ordenada y sin errores al destino, mientras que **IP** se encarga de "enviar" los datos a la dirección correcta.

Cómo funciona: Imagina que estás enviando un archivo grande a un amigo, pero no lo envías todo de una vez. En su lugar, lo divides en pequeños fragmentos (paquetes), los envías uno por uno, y tu amigo los recibe y los ensambla de nuevo. Si alguno de estos paquetes se pierde en el camino, **TCP** lo detecta y pide que se reenvíe. Esto asegura que el archivo completo llegue sin problemas.

Ejemplo de uso: Cuando ves un sitio web, tu computadora usa TCP para enviar solicitudes y recibir los datos completos y ordenados del servidor web. Esto es fundamental para que una página web se cargue correctamente.

Ventajas: Fiabilidad y corrección de errores, lo que asegura que la información llegue completa. **Desventajas**: Puede ser más lento que otros protocolos, ya que se enfoca en garantizar la entrega correcta de los datos.



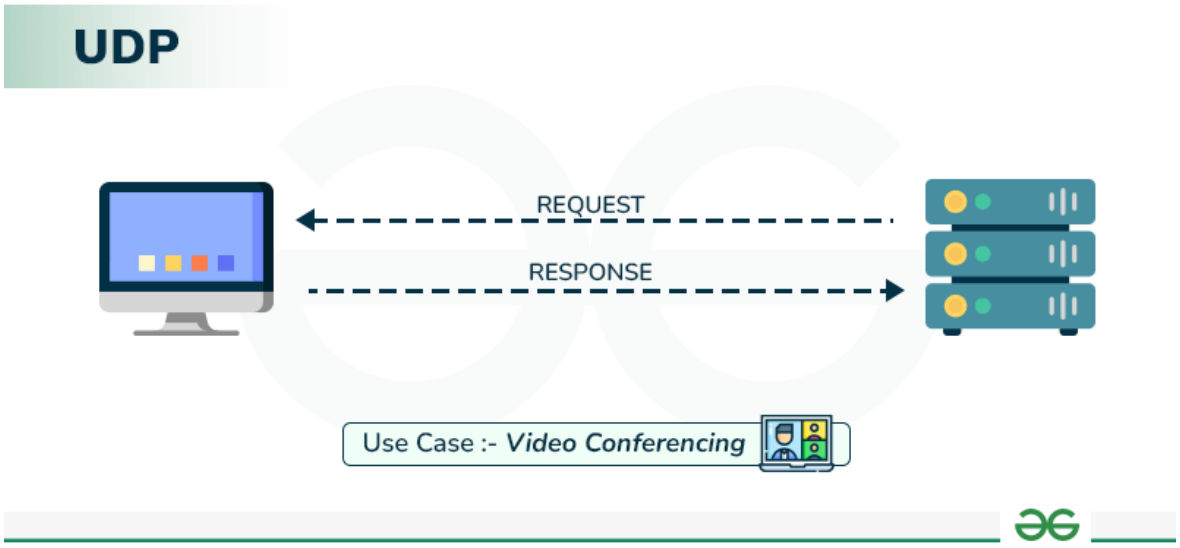
3.3.2. UDP (User Datagram Protocol)

UDP es como el hermano "rápido" de TCP. Es un protocolo más ligero y veloz que no garantiza que los datos lleguen completos o en el orden correcto. Se utiliza cuando la velocidad es más importante que la exactitud.

Cómo funciona: UDP no se preocupa tanto por verificar si cada fragmento de datos llegó correctamente o en el orden exacto. Es como enviar un mensaje rápido por mensajería instantánea: si el mensaje se pierde, sigues adelante sin detenerte.

Ejemplo de uso: Cuando haces una llamada de video en vivo (por ejemplo, en Skype o Zoom), se utiliza UDP. La prioridad es que el video y el audio lleguen lo más rápido posible, y si se pierde algún dato, el sistema no se detiene para corregirlo.

Ventajas: Es extremadamente rápido. **Desventajas**: No garantiza que los datos lleguen completos o en el orden correcto, por lo que puede haber pérdida de información.



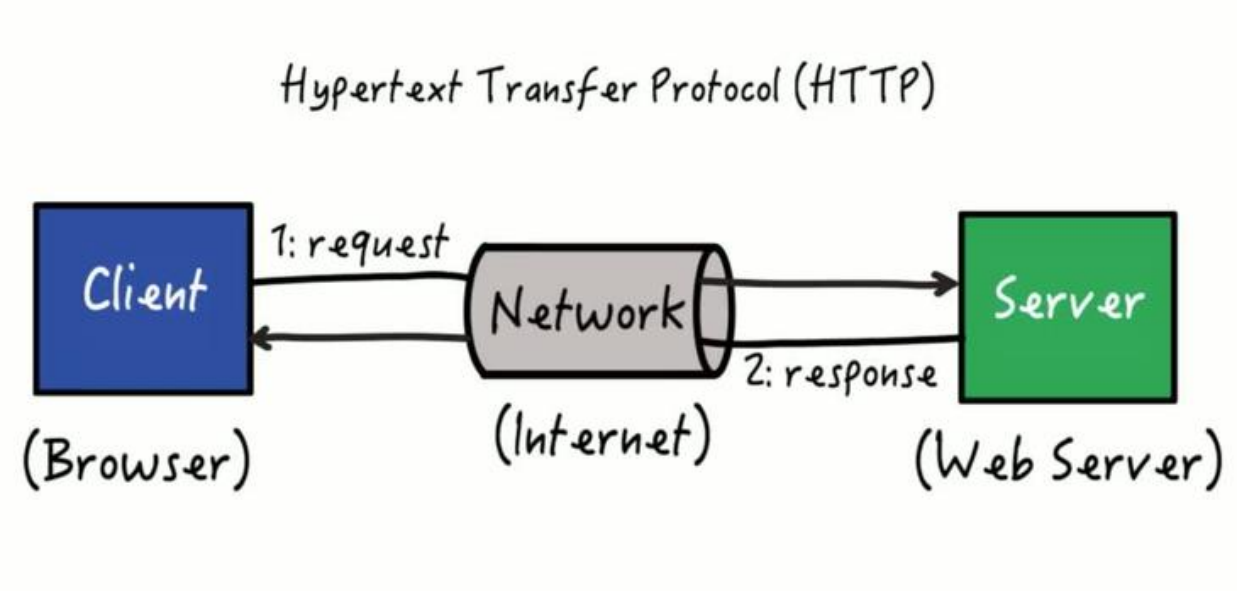
3.3.3. HTTP (Hypertext Transfer Protocol)

HTTP es el protocolo que gobierna la web. Es el que permite que navegues por Internet y accedas a páginas web. Es un protocolo que sigue el modelo cliente-servidor: tu navegador (cliente) solicita información, y el servidor web la proporciona.

Cómo funciona: Cada vez que escribes una dirección web en tu navegador (como www.google.com), se envía una solicitud **HTTP** al servidor de Google. El servidor procesa la solicitud y te envía de vuelta la página web.

Ejemplo de uso: Cada vez que visitas una página web o haces clic en un enlace, estás utilizando HTTP.

Ventajas: Fácil de implementar y ampliamente utilizado en la web. **Desventajas**: No fue diseñado para transmitir información en tiempo real, como video o audio en directo.



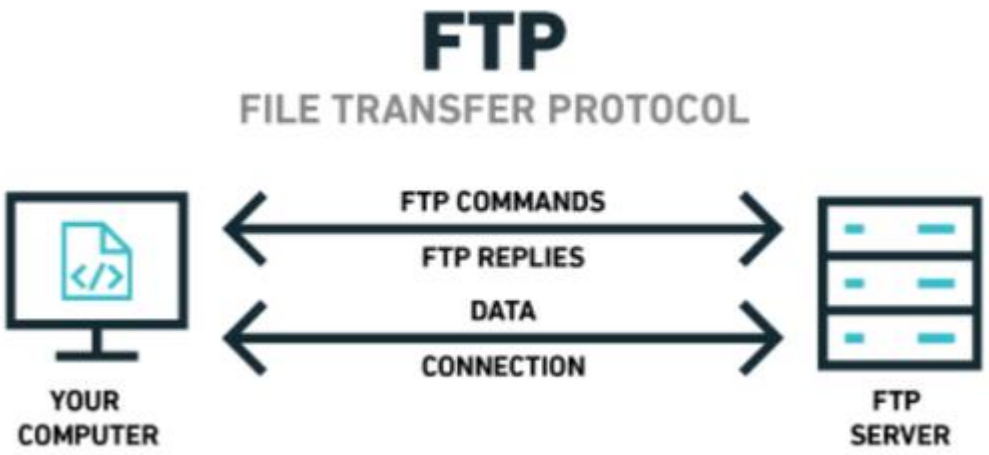
3.3.4. FTP (File Transfer Protocol)

FTP es un protocolo diseñado específicamente para transferir archivos entre un cliente y un servidor. Se utiliza cuando necesitas mover archivos grandes o muchos archivos de un lugar a otro.

Cómo funciona: FTP permite que un cliente acceda a un servidor y transfiera archivos hacia y desde él. Es como un servicio de mensajería para archivos.

Ejemplo de uso: Cuando descargas un software grande o actualizaciones, es probable que se estén transfiriendo archivos a través de FTP.

Ventajas: Es ideal para transferir grandes volúmenes de datos. **Desventajas**: Es relativamente antiguo y puede ser menos seguro que otros métodos más modernos de transferencia de archivos.



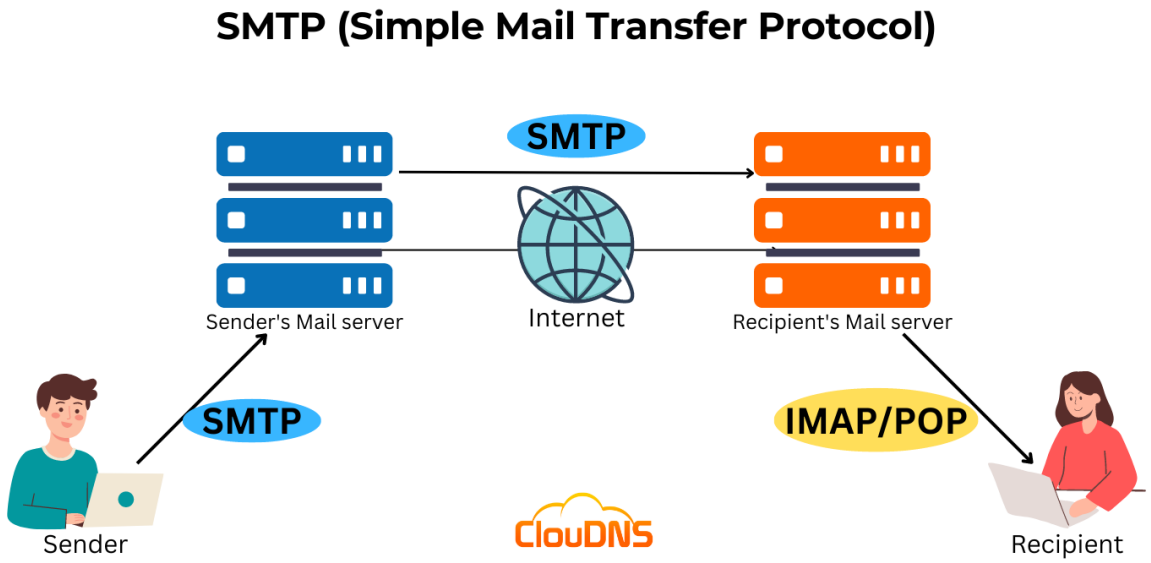
3.3.5. SMTP (Simple Mail Transfer Protocol)

SMTP es el protocolo estándar para el envío de correos electrónicos a través de Internet. Es utilizado por los servidores de correo para enviar mensajes de un lugar a otro.

Cómo funciona: Cuando envías un correo electrónico, tu cliente de correo utiliza **SMTP** para transferir el mensaje al servidor de correo del destinatario. Luego, el servidor del destinatario utiliza otros protocolos (como **POP** o **IMAP**) para hacer llegar el mensaje a la bandeja de entrada del destinatario.

Ejemplo de uso: Cada vez que envías un correo electrónico desde Gmail, Outlook o cualquier otro servicio de correo, estás utilizando **SMTP**.

Ventajas: Es eficiente para transferir mensajes. **Desventajas**: Originalmente no tenía medidas de seguridad integradas, lo que hizo que los correos fueran vulnerables a ataques (hoy en día se usan capas de seguridad adicionales como **SSL/TLS**).



3.4.Comparación entre los Protocolos

En esta sección, se pueden hacer algunas comparaciones entre los diferentes protocolos mencionados para ayudar a entender cuándo utilizar uno sobre el otro. Aquí algunos puntos clave para discutir:

Fiabilidad vs. Velocidad: **TCP** es más confiable que **UDP**, pero UDP es más rápido. Si necesitas asegurar que los datos lleguen completos y correctos, como en una transferencia de archivos, usa TCP. Si necesitas rapidez y puedes tolerar la pérdida de algunos datos, como en una videollamada, usa UDP.

Uso en tiempo real: Para aplicaciones que requieren comunicación en tiempo real, como videoconferencias o juegos en línea, **UDP** es preferido por su velocidad. En cambio, **HTTP** es más adecuado para transferir datos que no dependen de la velocidad inmediata, como páginas web.

Transferencias de archivos grandes: **FTP** es el protocolo estándar para transferencias de archivos, especialmente cuando se trata de grandes volúmenes de datos o cuando es necesario realizar una transferencia entre sistemas diferentes.

IV. EL ROL DEL MIDDLEWARE EN SISTEMAS DISTRIBUIDOS

Cuando hablamos de **sistemas distribuidos**, una de las grandes preguntas es: ¿cómo logramos que diferentes aplicaciones, que pueden estar corriendo en distintas máquinas en diferentes partes del mundo, funcionen de manera coordinada y efectiva? Aquí es donde el **middleware** entra en juego. Pero, ¿qué es realmente el middleware? Imagina que estás organizando una fiesta con amigos que no se conocen entre sí y no hablan el mismo idioma. Necesitarías un traductor que se asegure de que todos se entiendan y de que la fiesta siga funcionando sin problemas. El **middleware** juega un rol similar en los sistemas distribuidos: es un "intermediario" que ayuda a las aplicaciones distribuidas a comunicarse, coordinarse y trabajar juntas, aunque estén en sistemas o entornos completamente diferentes.

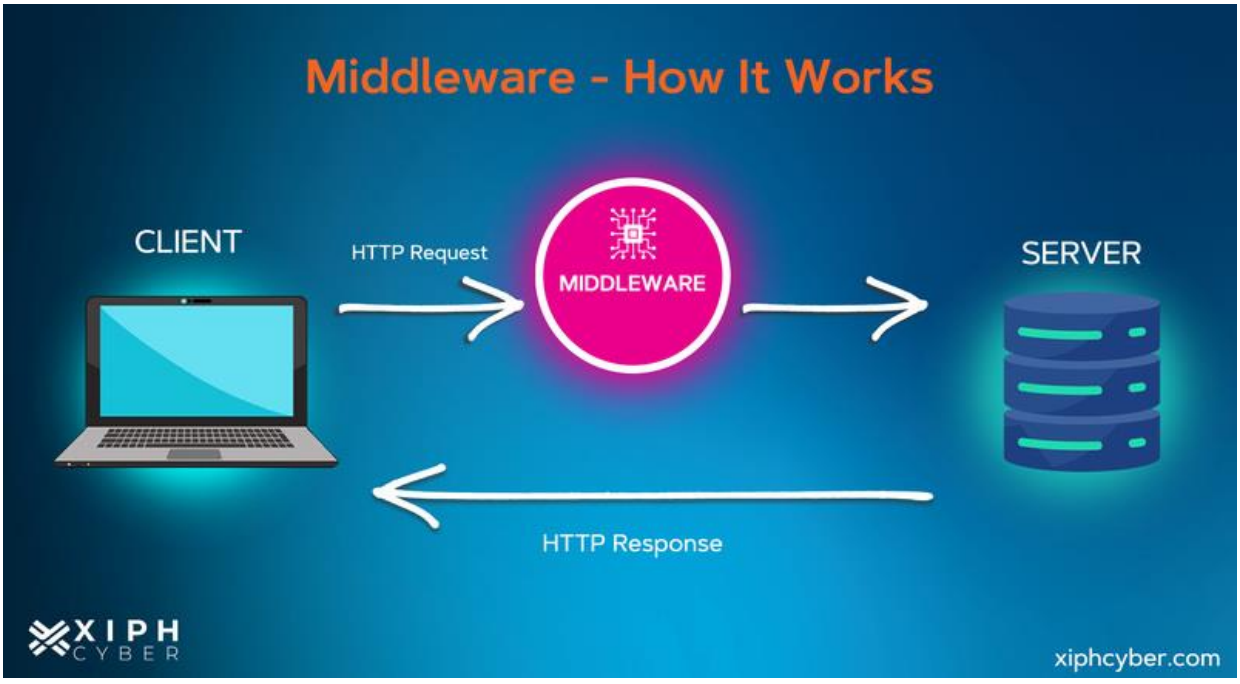
4.1.¿Qué es el Middleware?

El **middleware** es una capa de software que se encuentra entre las aplicaciones y los servicios subyacentes (como el sistema operativo, la red o el hardware) en un sistema distribuido. Su principal función es **facilitar la comunicación** y **gestionar la complejidad** de interactuar con otros componentes distribuidos. Sin el middleware, los desarrolladores tendrían que preocuparse por demasiados detalles técnicos de bajo nivel, como la configuración de redes, la seguridad y la transmisión de datos. En lugar de eso, el middleware se encarga de esas complejidades, proporcionando una **abstracción** que simplifica el desarrollo y la gestión de aplicaciones distribuidas.

Piénsalo como un intermediario eficiente. Si cada aplicación en una red distribuida tuviera que hablar directamente con todas las demás, el proceso sería lento, caótico y propenso a errores. El middleware se asegura de que las aplicaciones puedan comunicarse de manera clara, sin importar en qué tipo de sistema operativo o hardware estén corriendo.

Ejemplo de la vida real:

Imagina que estás usando una aplicación de banca en línea en tu teléfono. El sistema bancario tiene que interactuar con varios servicios distribuidos: la base de datos para consultar tu saldo, un servidor de autenticación para verificar tu identidad, y una aplicación de seguridad para asegurarse de que la transacción sea segura. El middleware hace que todas estas comunicaciones entre diferentes servicios y dispositivos ocurran de manera fluida, sin que tú como usuario tengas que preocuparte por cómo todo eso funciona "detrás de escena".



4.2.Funciones Principales del Middleware

El middleware no es solo una herramienta para facilitar la comunicación; también desempeña una serie de funciones críticas en los sistemas distribuidos. Veamos las más importantes:

4.2.1. Comunicación Transparente

La **comunicación transparente** es una de las características más importantes del middleware. ¿Qué significa? Básicamente, el middleware permite que las aplicaciones se comuniquen entre sí sin que los desarrolladores o usuarios tengan que preocuparse por cómo están conectados los diferentes nodos o máquinas. Las aplicaciones no necesitan saber si el nodo con el que están comunicándose está en la misma red, en otro país, o incluso en un entorno completamente diferente.

Por ejemplo, si estás usando una aplicación en tu computadora que necesita acceder a datos almacenados en un servidor remoto, el middleware se encarga de que esa solicitud llegue de manera correcta, sin que tengas que preocuparte por cómo se transfieren los datos a través de Internet.

Ejemplo práctico:

Imagina un juego multijugador en línea. Cuando juegas, no piensas en cómo tu computadora envía y recibe información de los servidores de juego y de otros jugadores que están en diferentes partes del mundo. El middleware facilita esta comunicación, asegurándose de que todas las acciones del juego se sincronicen sin que los jugadores experimenten retrasos visibles.

4.2.2. Coordinación de Procesos

En un sistema distribuido, varias aplicaciones o procesos pueden estar ejecutándose simultáneamente, lo que significa que hay que coordinar cómo y cuándo acceden a los recursos compartidos. El middleware se encarga de **sincronizar estos procesos**, evitando conflictos y garantizando que todas las tareas se realicen de manera ordenada.

Por ejemplo, si dos aplicaciones están tratando de acceder a la misma base de datos al mismo tiempo, el middleware gestiona ese acceso para evitar que las dos modifiquen los datos de manera conflictiva. Sin este control, podrían ocurrir problemas de integridad de los datos, como una aplicación sobreescribiendo cambios hechos por otra.

Ejemplo práctico:

Imagina un sistema de compras en línea, donde varios usuarios están realizando pedidos simultáneamente. El middleware garantiza que cada transacción se procese correctamente, incluso si varias personas intentan comprar el mismo producto a la vez, evitando errores como el cobro doble o la sobreventa de productos.

4.2.3. Acceso a Datos Distribuidos

Los sistemas distribuidos a menudo tienen **bases de datos distribuidas**, es decir, bases de datos que están repartidas en varios servidores o ubicaciones. El middleware permite que las aplicaciones accedan a estos datos como si estuvieran en un solo lugar, ocultando la complejidad de la distribución de datos y asegurando que todas las aplicaciones obtengan la información correcta de manera eficiente.

Por ejemplo, cuando consultas una base de datos en un sistema distribuido, el middleware se encarga de buscar los datos en los diferentes servidores, combinarlos y presentarlos de manera unificada a la aplicación.

Ejemplo práctico:

Un sistema de reservación de vuelos puede tener servidores en diferentes regiones del mundo. Si un usuario busca vuelos desde América del Norte a Europa, el middleware consulta las bases de datos en varios servidores (cada uno quizás en una región distinta) y devuelve una lista de vuelos como si todos los datos provinieran de un solo lugar.

4.2.4. Gestión de Transacciones y Fallos

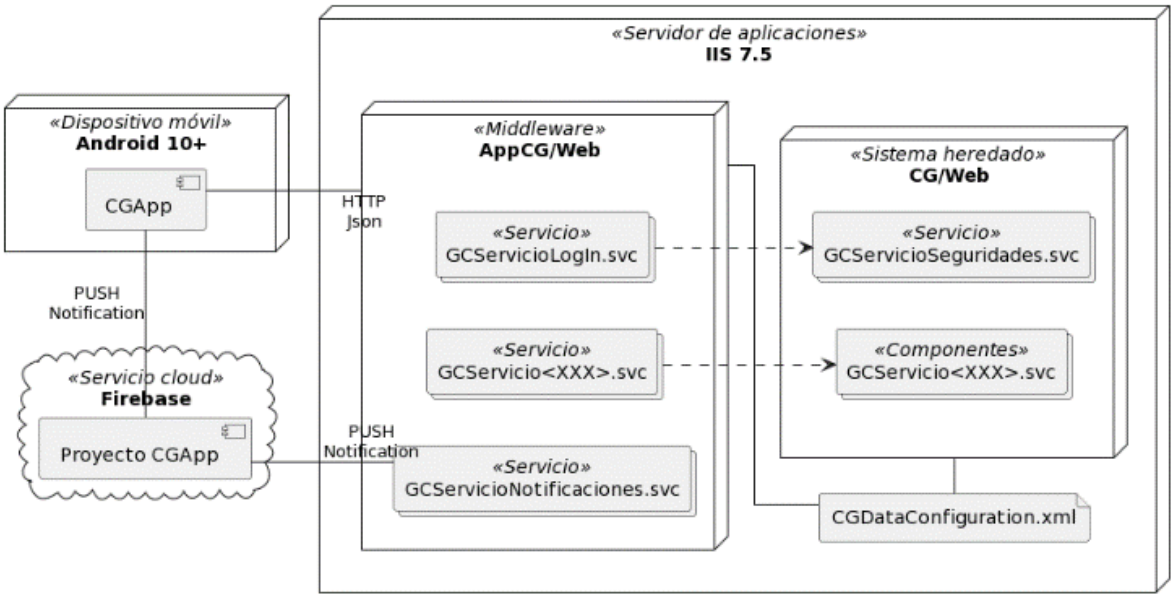
Otra función crítica del middleware es manejar las **transacciones distribuidas**, que son operaciones que involucran múltiples nodos y recursos. Las transacciones deben ser atómicas, lo que significa que deben completarse completamente o no completarse en absoluto. El middleware garantiza que si ocurre un fallo en medio de una transacción, se reviertan todos los cambios para mantener la coherencia del sistema.

Por ejemplo, si un sistema bancario distribuye una transacción entre varios servidores y algo falla, el middleware garantiza que la transacción no sea registrada de manera parcial. De esta manera, no hay riesgo de que el sistema se quede con un saldo incorrecto.

Ejemplo práctico:

Imagina que estás transfiriendo dinero entre cuentas en diferentes bancos que están ubicados en distintas regiones. Si la transferencia falla en algún punto intermedio, el middleware

se asegura de que todo se revierta para evitar problemas como que el dinero salga de una cuenta pero no llegue a la otra.



4.3.Ejemplos de Middleware

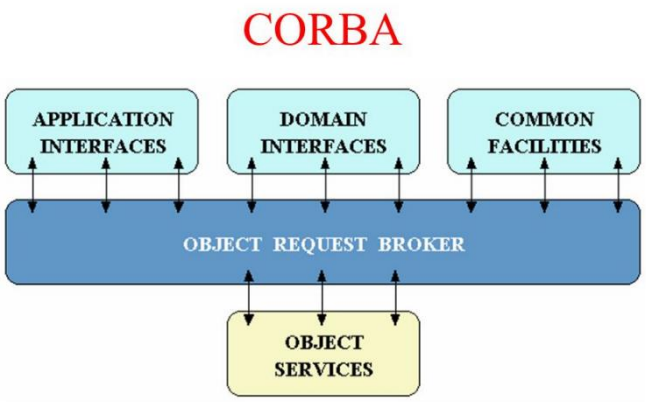
Existen diferentes tipos de middleware, diseñados para diferentes tipos de sistemas distribuidos. Aquí se presentan algunos de los más comunes y sus aplicaciones:

6.3.1. CORBA (Common Object Request Broker Architecture)

CORBA es un middleware que permite que aplicaciones escritas en diferentes lenguajes de programación y que se ejecutan en diferentes plataformas puedan interactuar entre sí. CORBA define cómo los objetos pueden solicitar y recibir servicios de otros objetos de manera transparente, sin importar en qué lenguaje estén programados o dónde estén ubicados.

Ejemplo:

Imagina que tienes una aplicación en **C++** corriendo en un servidor y necesitas comunicarte con otra aplicación escrita en **Java** en otro servidor. **CORBA** permite que estos dos programas intercambien datos y trabajen juntos, sin importar que sean completamente diferentes en términos de lenguaje y plataforma.



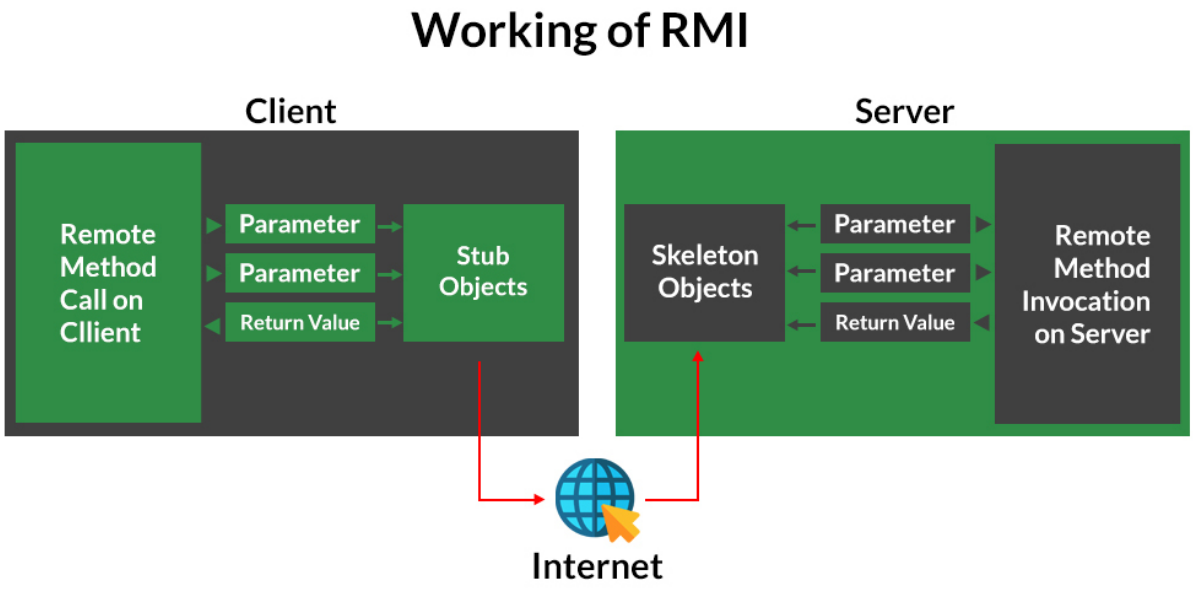
OMG Reference Model architecture

6.3.2. Java RMI (Remote Method Invocation)

RMI es una tecnología específica de Java que permite que los objetos Java en diferentes máquinas puedan invocar métodos entre sí de manera remota, como si estuvieran en la misma máquina. RMI es muy útil cuando se está trabajando exclusivamente dentro del entorno Java, ya que facilita la creación de aplicaciones distribuidas sin preocuparse por la comunicación subyacente.

Ejemplo:

Imagina una aplicación de control de inventarios donde los objetos distribuidos en servidores remotos necesitan actualizar los registros de inventario. **RMI** permite que un cliente Java invoque un método en un objeto remoto para modificar el inventario sin necesidad de crear manualmente la lógica de comunicación.



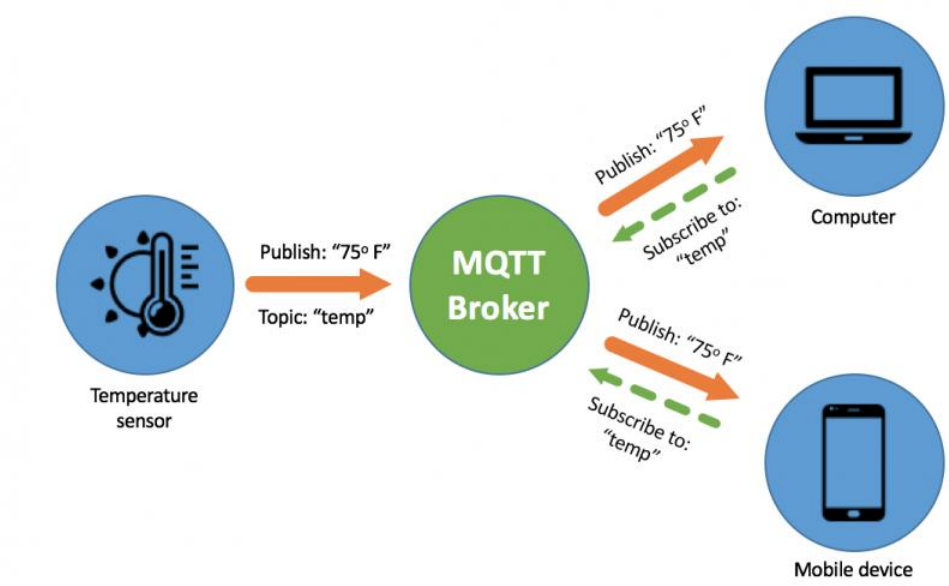
6.3.3. MQTT (Message Queuing Telemetry Transport)

MQTT es un middleware muy popular en sistemas de **Internet de las Cosas (IoT)**. Es ligero y eficiente, diseñado para que los dispositivos pequeños y de baja potencia puedan comunicarse en una red distribuida. Se basa en un modelo de **publicación/suscripción**, donde los dispositivos publican información y otros dispositivos la reciben si están suscritos a esos datos.

Ejemplo:

Imagina una red de sensores de temperatura distribuidos en una ciudad para medir la calidad del aire. Cada sensor envía sus datos de temperatura a través de MQTT, y cualquier

dispositivo que esté suscrito a esos datos (como una aplicación de monitoreo) los recibe en tiempo real.



4.4.Ventajas y Desventajas del Middleware

Como toda tecnología, el middleware tiene **ventajas** que lo hacen invaluable en muchos sistemas distribuidos, pero también presenta algunas **desventajas**. Aquí una visión general de ambas:

Ventajas:

Simplificación del desarrollo: Los desarrolladores no tienen que preocuparse por los detalles técnicos de la comunicación entre los sistemas, lo que les permite enfocarse en la lógica de la aplicación.

Portabilidad: El middleware facilita que las aplicaciones distribuidas funcionen en diferentes plataformas y sistemas operativos sin modificaciones importantes.

Manejo de complejidades: Se encarga de problemas complicados como la sincronización de datos, la gestión de fallos y la seguridad, reduciendo la carga de los desarrolladores.

Desventajas:

Sobrecarga de rendimiento: Dado que el middleware introduce una capa adicional de procesamiento, puede aumentar la latencia y reducir el rendimiento en redes muy grandes.

Complejidad adicional: A veces, el middleware puede ser tan complejo que requiere un conocimiento profundo para configurarlo y mantenerlo adecuadamente.

Dependencia: Las aplicaciones distribuidas que dependen fuertemente de middleware específico pueden volverse difíciles de modificar o escalar sin cambiar el middleware subyacente.

V. IMPLEMENTACIÓN DE UNA APLICACIÓN CLIENTE-SERVIDOR CON RPC O RMI

Cuando hablamos de **sistemas distribuidos**, una de las principales necesidades es que las aplicaciones puedan interactuar entre sí, aunque estén ejecutándose en diferentes máquinas, servidores o incluso en diferentes ubicaciones geográficas. Para facilitar esta interacción entre aplicaciones que pueden estar en máquinas separadas, se utilizan tecnologías como **RPC (Remote Procedure Call)** y **RMI (Remote Method Invocation)**, que hacen que la comunicación remota parezca local. Esto permite a los desarrolladores crear aplicaciones distribuidas sin tener que preocuparse por los detalles técnicos de la red. A continuación, profundizaremos en estos conceptos y veremos cómo funcionan en la práctica, con ejemplos que ilustran cómo se implementan.

5.1. ¿Qué es RPC (Remote Procedure Call)?

RPC es una tecnología que permite que un programa, que está corriendo en una máquina, ejecute procedimientos o funciones en otro servidor remoto de una manera que parece local. Básicamente, con **RPC**, el programador escribe el código como si estuviera invocando una función local, pero en realidad, la función está siendo ejecutada en un servidor remoto. Esto es extremadamente útil en aplicaciones distribuidas donde diferentes partes del sistema pueden estar distribuidas en varias máquinas.

Imagina que tienes una aplicación de compras en línea. El cliente está en tu computadora, pero la base de datos y la lógica de los pedidos están en un servidor remoto. En lugar de que el cliente realice todas las operaciones localmente y luego intente coordinarse con el servidor, puedes

usar **RPC** para que el cliente llame directamente a los procedimientos remotos (como “agregar al carrito” o “procesar pago”) en el servidor.

Cómo funciona RPC:

El cliente (es decir, tu programa en tu computadora) envía una solicitud al servidor pidiéndole que ejecute una función específica.

El servidor recibe esa solicitud, ejecuta la función en su máquina y luego envía los resultados de vuelta al cliente.

Para el cliente, parece que la función se ejecutó localmente, pero en realidad ocurrió en un servidor remoto.

El verdadero valor de **RPC** es que **esconde la complejidad de la red**. El cliente no tiene que preocuparse por cómo enviar los datos, cómo recuperarlos o cómo gestionar las conexiones de red. **RPC** se encarga de todo eso.

Ejemplo práctico con RPC:

Veamos un ejemplo sencillo en **C**. Supongamos que queremos hacer que un cliente pregunte al servidor por el precio de un producto en una tienda.

Código del servidor:

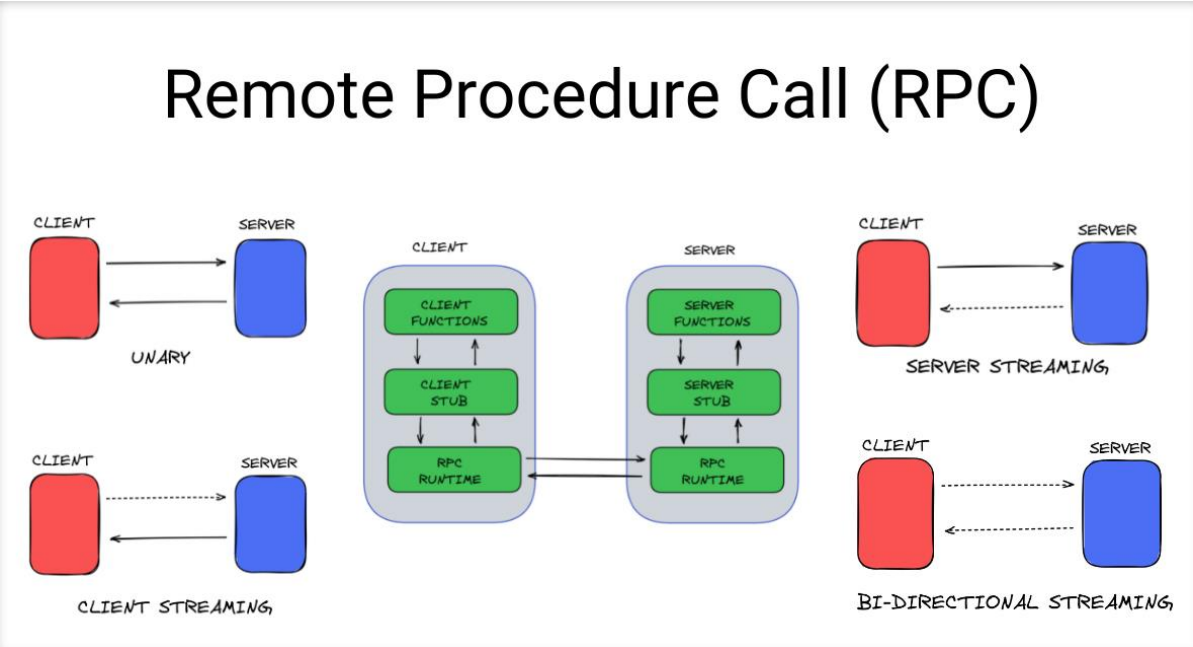
```
#include <stdio.h>
#include <rpc/rpc.h>

// Definir la función remota
int *obtener_precio_1_svc(void *argp, struct svc_req *rqstp) {
    static int precio = 150;
```

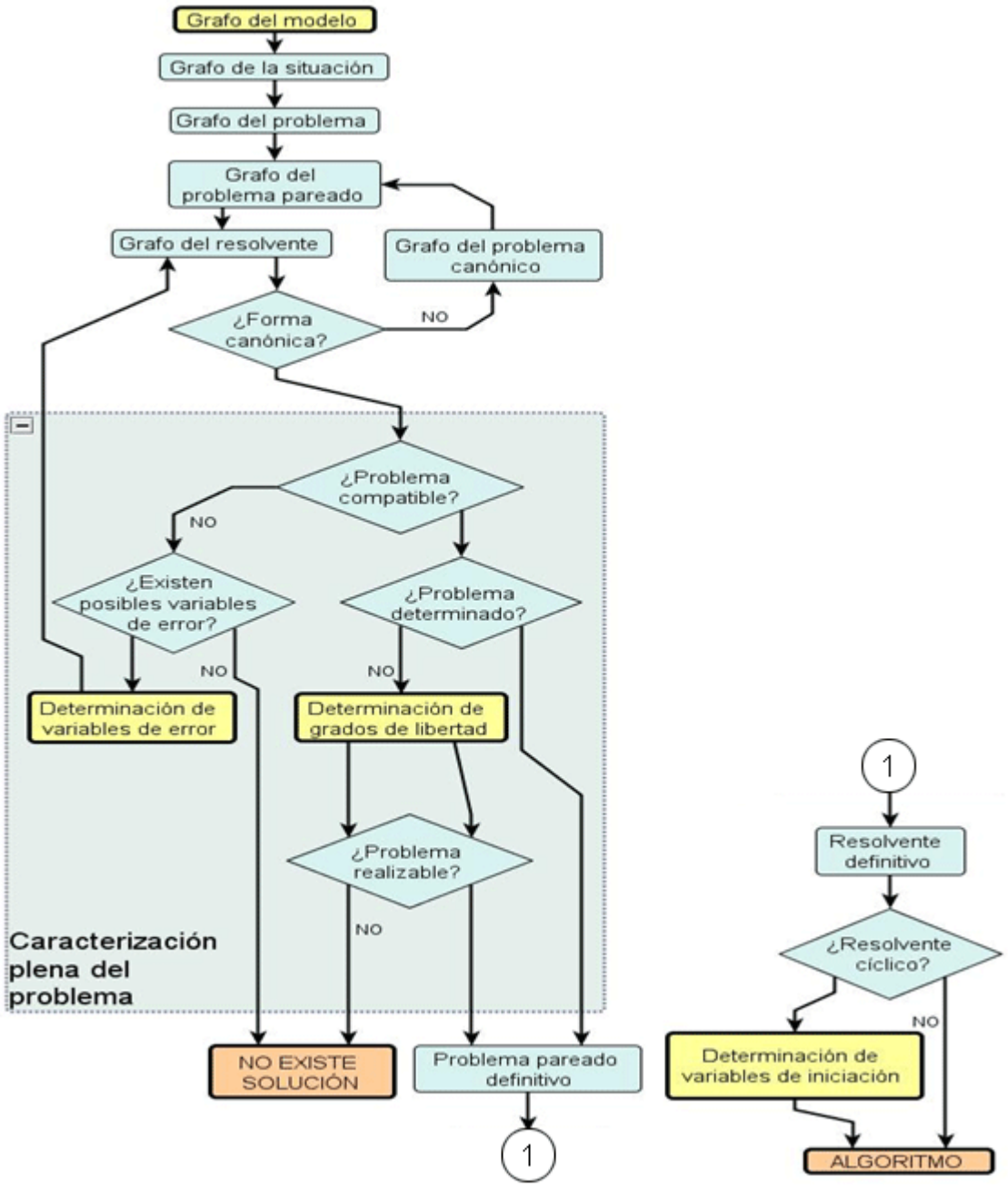
```
    return &precio;
}
```

Código del cliente:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "tienda.h"

int main() {
    CLIENT *cl;
    int *resultado;

    // Crear cliente RPC
    cl = clnt_create("localhost", TIENDA_PROG, TIENDA_VERS, "udp");
    if (cl == NULL) {
        clnt_pcreateerror("Error al crear cliente");
        return 1;
    }

    // Llamada RPC
    resultado = obtener_precio_1(NULL, cl);
    if (resultado == NULL) {
        clnt_perror(cl, "Error en la llamada RPC");
        return 1;
    }

    printf("El precio del producto es: %d\n", *resultado);
    return 0;
}
```

En este ejemplo, el cliente invoca la función obtener_precio_1 en el servidor, que le devuelve el precio del producto. Desde la perspectiva del cliente, parece que está llamando a una función local, pero en realidad, la función se ejecuta en el servidor remoto.





5.2.¿Qué es RMI (Remote Method Invocation)?

RMI (Remote Method Invocation) es una tecnología específica del lenguaje **Java** que permite a los objetos distribuidos en diferentes máquinas llamarse entre sí como si estuvieran en la misma máquina. Aunque **RMI** es similar a **RPC** en su propósito (permitir la ejecución remota

de procedimientos), está más orientada a la **programación orientada a objetos** y se integra directamente con Java, lo que facilita la interacción entre objetos remotos.

En **RMI**, un objeto en un cliente puede invocar métodos en un objeto que reside en un servidor remoto. Los métodos invocados pueden devolver objetos completos, no solo datos simples como en **RPC**, lo que hace que **RMI** sea una herramienta poderosa para desarrollar aplicaciones distribuidas en Java.

Cómo funciona RMI:

El cliente tiene una referencia a un objeto remoto (servidor) y llama a uno de sus métodos, como si el objeto estuviera en la misma máquina.

RMI serializa (convierte a bytes) los datos y los envía al servidor, donde el objeto remoto ejecuta el método.

El servidor luego devuelve los resultados al cliente, también de forma transparente.

Todo el proceso está oculto para el desarrollador, quien escribe el código como si los objetos remotos estuvieran en el mismo entorno.

Ejemplo práctico con RMI:

Aquí tienes un ejemplo de cómo implementar una simple aplicación cliente-servidor en Java usando **RMI**. Imagina que tenemos un servidor que almacena información sobre productos y el cliente quiere consultar el precio de uno de ellos.

Interfaz remota (definida en el servidor) Java:

```
import java.rmi.Remote;
```

```
import java.rmi.RemoteException;

public interface Tienda extends Remote {
    public int obtenerPrecio() throws RemoteException;
}
```

Implementación del servidor:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class TiendaImpl extends UnicastRemoteObject implements Tienda {
    private int precio = 100;

    public TiendaImpl() throws RemoteException {
        super();
    }

    public int obtenerPrecio() throws RemoteException {
        return precio;
    }
}
```

Código del servidor para publicar el objeto remoto:

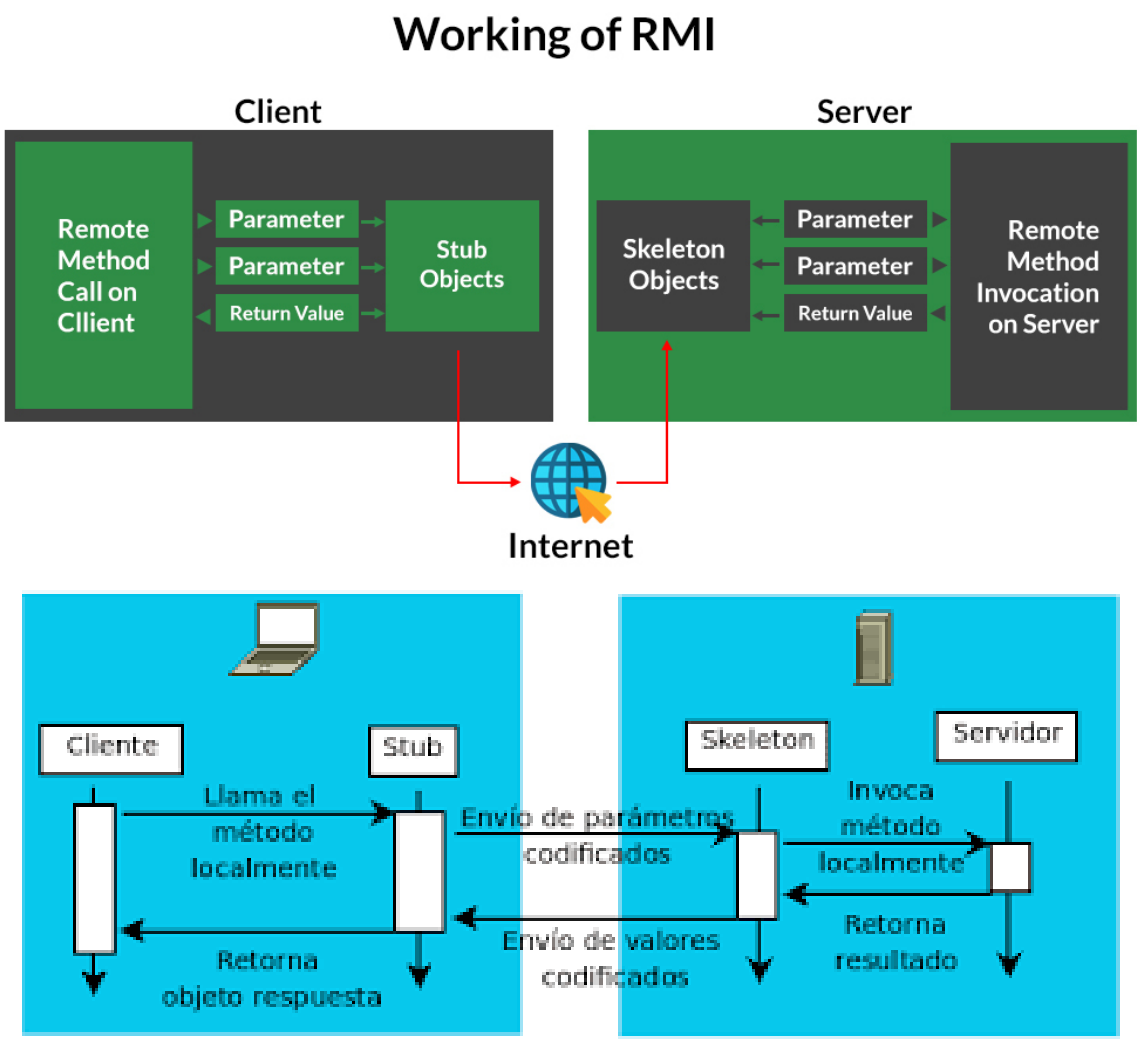
```
import java.rmi.Naming;

public class Servidor {
    public static void main(String[] args) {
        try {
            Tienda tienda = new TiendaImpl();
            Naming.rebind("rmi://localhost/TiendaService", tienda);
            System.out.println("Servidor listo");
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Código del cliente:

```
import java.rmi.Naming;

public class Cliente {
    public static void main(String[] args) {
        try {
            Tienda tienda = (Tienda) Naming.lookup("rmi://localhost/TiendaService");
            int precio = tienda.obtenerPrecio();
            System.out.println("El precio del producto es: " + precio);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

En este ejemplo, el cliente invoca el método obtenerPrecio de un objeto remoto **Tienda** que está almacenado en el servidor. El cliente no necesita saber nada sobre cómo el servidor gestiona la comunicación; **RMI** se encarga de todo.



5.3.Comparación entre RPC y RMI

Ahora que hemos visto cómo funcionan **RPC** y **RMI**, es importante entender las **similitudes** y **diferencias** entre ambas tecnologías, y cuándo utilizar una sobre la otra.

Similitudes:

Ambas permiten la invocación de funciones o métodos en un servidor remoto como si fueran locales.

Ocultan la complejidad de la comunicación en red.

Son ideales para aplicaciones cliente-servidor distribuidas.

Diferencias:

RPC es un concepto más general y se puede implementar en varios lenguajes de programación (C, Python, etc.), mientras que **RMI** está diseñado específicamente para el lenguaje **Java**.

RMI trabaja con objetos y métodos, permitiendo enviar y recibir objetos completos, mientras que **RPC** se enfoca en la invocación de funciones con tipos de datos primitivos o estructuras simples.

RPC es más adecuado para sistemas que no dependen de la orientación a objetos, mientras que **RMI** es ideal para sistemas completamente orientados a objetos.

Ventajas de RPC:

Es independiente del lenguaje de programación y se puede usar en diferentes plataformas.

Más sencillo de implementar en aplicaciones que solo necesitan funciones simples y no dependen de la orientación a objetos.

Ventajas de RMI:

Permite aprovechar todo el poder de la programación orientada a objetos, permitiendo el paso de objetos complejos entre cliente y servidor.

Totalmente integrado con el entorno **Java**, lo que simplifica la creación de aplicaciones distribuidas dentro de este ecosistema.

	RPC	RMI	Web Services
Birth	1976 - 1981	~ 1990	~ 2000
Platform	Library and OS-dependant	Java	Independent ☺
Transport	OS-Dependent	HTTP or IIOP	HTTP(s)
Dev Cost	Huge	Reasonable	Low
Security	None	Client-level	Transport Level
Overhead	None	OOP + HTTP	XML + HTTP
Dynamic invocation	None	Yes, using RDMI	Natural
Versioning	Huge problem	Possible using RDMI	Natural
Service lookup	Impossible	Java Naming and Directory	UDDI

VI. COMPARACIÓN ENTRE PROTOCOLOS DE COMUNICACIÓN Y MIDDLEWARE

En los sistemas distribuidos, tanto los **protocolos de comunicación** como el **middleware** juegan un papel fundamental en la forma en que los diferentes nodos (dispositivos, computadoras o servidores) se comunican, coordinan y colaboran. Aunque ambos se encargan de la **comunicación** y **sincronización** entre las aplicaciones distribuidas, tienen roles y enfoques diferentes. En esta sección, realizaremos una comparación estructurada que clarifica cómo cada uno de estos componentes contribuye a la arquitectura general de los sistemas distribuidos.

6.1. Diferencias Conceptuales

Protocolos de Comunicación y **Middleware** son conceptos que, aunque relacionados, tienen enfoques distintos:

Protocolos de Comunicación: Se enfocan en las **reglas y normas** que permiten que dos o más sistemas se conecten entre sí y transfieran datos. Los protocolos de comunicación se encargan de detalles como el formato de los mensajes, la dirección de los paquetes de datos, y la corrección de errores. Ejemplos incluyen **TCP/IP**, **UDP**, y **HTTP**.

Middleware: Es una capa de software que actúa como **intermediaria** entre las aplicaciones y los servicios de comunicación subyacentes. El middleware es responsable de abstraer la complejidad de la comunicación y ofrecer una interfaz más simple a los desarrolladores de aplicaciones. También gestiona tareas como la sincronización, la distribución de datos y la coordinación de procesos. Ejemplos de middleware incluyen **CORBA**, **Java RMI**, y **MQTT**.

Ejemplo:

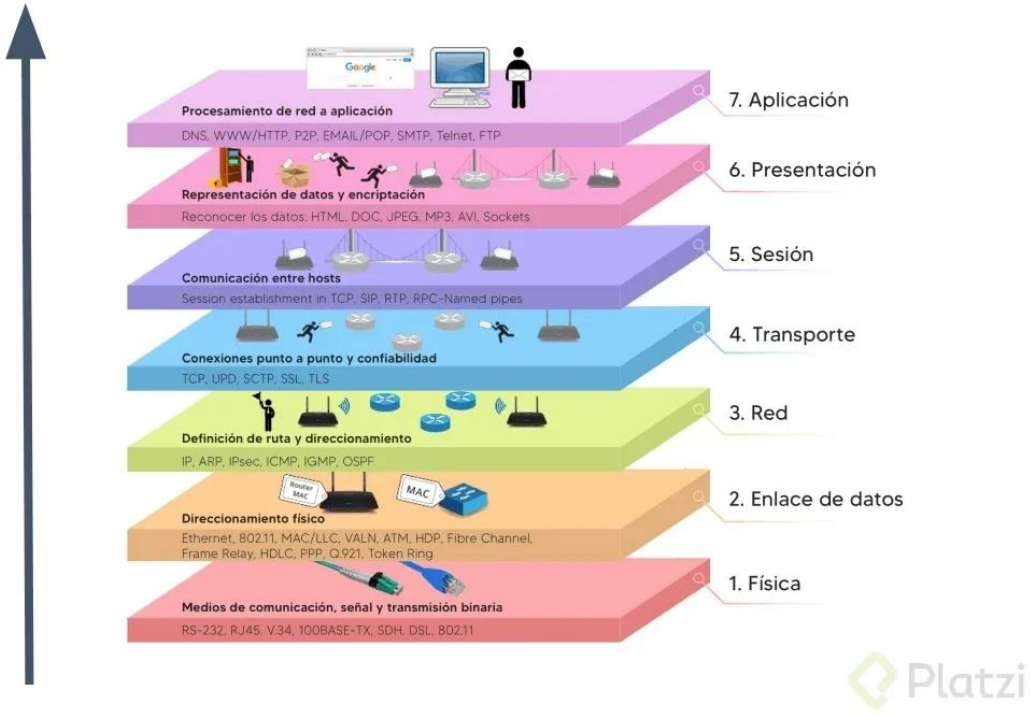
Si pensamos en una aplicación bancaria que se ejecuta en múltiples servidores, los **protocolos de comunicación** serían como el idioma en el que los servidores se hablan, mientras que el **middleware** sería como un traductor que se encarga de que todas las partes se entiendan, gestionando el flujo de información, el acceso a datos y asegurándose de que los mensajes lleguen a su destino sin errores.

6.2. Comparación Técnica

Aquí analizaremos algunos aspectos clave de los **protocolos de comunicación** y el **middleware**, y cómo se diferencian en términos de funcionalidad, uso y desempeño:

Aspecto	Protocolos de Comunicación	Middleware

Propósito	Define cómo se transfieren los datos entre dispositivos	Facilita la interacción entre aplicaciones distribuidas
Nivel de Abstracción	Bajo: Los desarrolladores deben manejar detalles de red	Alto: Oculta la complejidad de la comunicación entre nodos
Responsabilidades	Envío y recepción de datos, control de errores, direccionamiento	Coordinación de procesos, sincronización, manejo de transacciones
Ejemplos	TCP/IP, HTTP, UDP, SMTP	CORBA, RMI, MQTT, Web Services
Uso Típico	Transferencia de datos en bruto entre sistemas	Ejecución distribuida de aplicaciones y servicios
Independencia de la Plataforma	Limitado por la infraestructura de red	Total, ya que se abstrae de la red subyacente
Control de la Red	Requiere que los desarrolladores gestionen los detalles de red	Se encarga de todos los detalles técnicos de la red



6.3. Ejemplo Práctico de Uso Combinado

Imaginemos una **aplicación de videoconferencia** como Zoom o Microsoft Teams. Este tipo de aplicación distribuye datos en tiempo real, como video, audio y texto, entre varios participantes ubicados en diferentes partes del mundo. Aquí podemos ver cómo trabajan juntos los **protocolos de comunicación** y el **middleware**:

Protocolos de Comunicación: Envían los datos de video y audio entre los servidores y los dispositivos de los usuarios utilizando **UDP** para garantizar una transmisión rápida, aunque algunos paquetes de datos se puedan perder sin afectar gravemente la experiencia del usuario.

Middleware: Gestiona la sincronización de los participantes, controla el acceso a los recursos y asegura que, si un participante pierde temporalmente la conexión, pueda reconectarse

sin problemas. Además, el middleware puede encargarse de tareas como la autenticación de los usuarios y el manejo de las transacciones financieras si el servicio es de pago.

En este caso, los **protocolos de comunicación** garantizan la velocidad y la transmisión de datos, mientras que el **middleware** asegura que la aplicación funcione de manera coherente y sin fallos, a pesar de que los usuarios estén distribuidos en distintas redes.

8.4. Ventajas y Desventajas

Protocolos de Comunicación:

Ventajas:

Control total sobre la red y la transmisión de datos.

Fiabilidad y velocidad ajustables según el protocolo utilizado (por ejemplo, TCP para fiabilidad, UDP para velocidad).

Desventajas:

Complejidad para los desarrolladores, ya que deben manejar directamente los detalles técnicos de la red.

Limitaciones en cuanto a la independencia de la plataforma.

Middleware:

Ventajas:

Simplifica el desarrollo de aplicaciones distribuidas al abstraer la complejidad de la red.

Proporciona una interfaz uniforme para aplicaciones distribuidas, independientemente de la infraestructura de red.

Desventajas:

Puede introducir una sobrecarga de rendimiento, ya que agrega una capa adicional de procesamiento.

Dependencia en soluciones específicas de middleware, lo que puede limitar la flexibilidad.

8.5. Cuándo Utilizar Protocolos de Comunicación vs Middleware

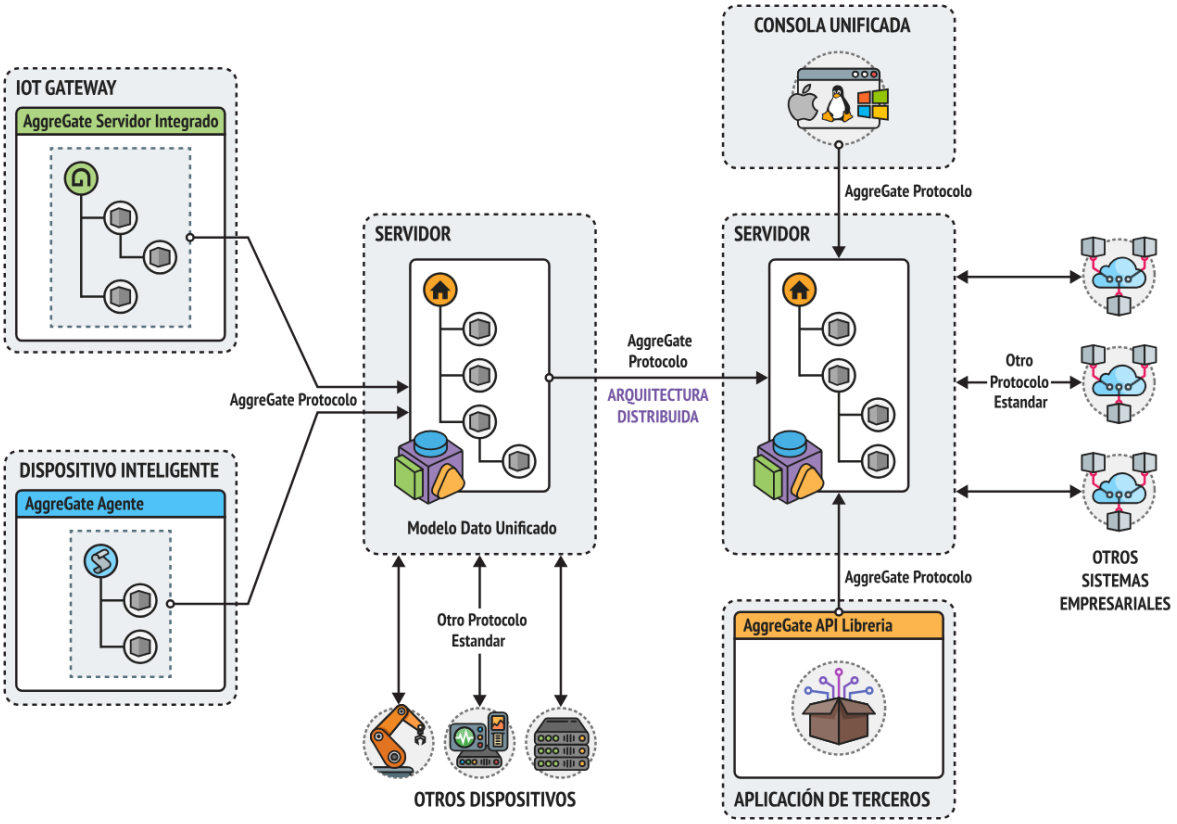
Protocolos de Comunicación son ideales cuando se requiere un control directo y detallado sobre cómo se transfieren los datos, o cuando se necesitan optimizaciones específicas en la red, como en aplicaciones que requieren un alto rendimiento de red, como el streaming de video o los videojuegos en línea.

Middleware es la mejor opción cuando el enfoque está en la simplificación del desarrollo de aplicaciones distribuidas y cuando es necesario manejar aspectos más allá de la simple transmisión de datos, como la coordinación de múltiples procesos, la gestión de transacciones y la sincronización de bases de datos distribuidas.

Ejemplo:

En un sistema de banca en línea, es más práctico usar middleware para gestionar las transacciones financieras distribuidas, ya que debe garantizarse que cada operación se complete de manera atómica y segura. En cambio, en una aplicación de transmisión en vivo, como una

retransmisión deportiva, los protocolos de comunicación como UDP son más útiles, ya que priorizan la velocidad y toleran pequeñas pérdidas de datos.



VII. CONCLUSIONES

A lo largo de esta monografía, hemos explorado los conceptos fundamentales que permiten el funcionamiento de los **sistemas distribuidos**, desde los **protocolos de comunicación** que permiten la transmisión de datos entre nodos, hasta el **middleware**, que simplifica la creación y gestión de aplicaciones distribuidas. Cada uno de estos componentes juega un papel crucial para que las aplicaciones modernas puedan operar de manera eficiente, escalable y confiable, especialmente en un entorno donde la comunicación entre dispositivos ubicados en distintas geografías es esencial.

Los **protocolos de comunicación**, como **TCP/IP**, **UDP** o **HTTP**, proporcionan las reglas básicas que garantizan que los datos se transfieran de manera correcta y segura a través de redes complejas. Son los pilares sobre los que se construye cualquier red distribuida, ya que permiten que los sistemas distribuidos puedan compartir información sin importar las diferencias en hardware o ubicación geográfica. Sin estos protocolos, sería imposible establecer conexiones confiables o asegurar la transmisión adecuada de los datos.

Por otro lado, el **middleware** añade una capa de abstracción que facilita la interacción entre los nodos distribuidos. A través de ejemplos como **CORBA**, **RMI** y **MQTT**, hemos visto cómo el middleware oculta la complejidad de la red y permite que los desarrolladores se centren en la lógica de negocio de sus aplicaciones, en lugar de preocuparse por los detalles técnicos de la comunicación. El middleware también proporciona características avanzadas como la **gestión de transacciones**, la **sincronización de procesos** y el **acceso distribuido a los datos**, lo que lo convierte en un componente esencial para aplicaciones más complejas.

Además, se ha discutido la implementación de sistemas cliente-servidor usando **RPC** y **RMI**, lo que nos ha permitido ver cómo estas tecnologías simplifican la interacción remota entre aplicaciones. RPC y RMI son ejemplos claros de cómo las aplicaciones pueden llamar a procedimientos o métodos en sistemas remotos como si estuvieran en la misma máquina, ocultando la complejidad de la red subyacente y proporcionando una interfaz intuitiva para los desarrolladores.

En resumen, tanto los protocolos de comunicación como el middleware son fundamentales para el diseño y la implementación de **sistemas distribuidos eficientes y escalables**. Cada uno tiene su propio rol, y ambos se complementan para garantizar que las aplicaciones distribuidas

puedan operar de manera fluida, sin importar la ubicación o el número de nodos involucrados. A medida que la tecnología avanza y la necesidad de sistemas distribuidos continúa creciendo, el entendimiento y la correcta aplicación de estos conceptos será clave para desarrollar soluciones robustas y sostenibles en el futuro.

VIII. BIBLIOGRAFÍA

Coulouris, G., Dollimore, J., & Kindberg, T. (2005). *Distributed Systems: Concepts and Design*. Addison-Wesley.

Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Prentice Hall.

Birrell, A. D., & Nelson, B. J. (1984). *Implementing Remote Procedure Calls*. ACM Transactions on Computer Systems (TOCS), 2(1), 39-59.

Sun Microsystems. (1999). *Java Remote Method Invocation Specification*. Oracle.

Foster, I., & Kesselman, C. (2004). *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann.

Amazon Web Services (AWS). (2021). *What is Cloud Computing?*. Recuperado de https://aws.amazon.com/what-is-cloud-computing/

Huang, R. et al. (2010). *MQTT-S—A Publish/Subscribe Protocol for Wireless Sensor Networks*. In 3rd International Conference on Consumer Electronics.

BitTorrent Inc. (2020). *How BitTorrent Works*. Recuperado de https://www.bittorrent.com/guide.