Unidad 2 / Escenario 3

Lectura fundamental

Arquitecturas cliente-servidor y objetos distribuidos

Contenido

Palabras clave: arquitectura, cliente, servidor, objetos distribuidos, *request-reply*, *middleware*

Introducción

Esta Lectura fundamental está dividida en tres temas: las ventajas y desventajas de los sistemas distribuidos, las definiciones básicas de la arquitectura de sistemas distribuidos, y los fundamentos de las arquitecturas cliente-servidor.

Hoy por hoy, la computación a gran escala, como por ejemplo Internet y todos los sistemas de información, son sistemas distribuidos.

El procesamiento de datos en un sistema distribuido se reparte entre los computadores que atienden el proceso, por lo cual, dicho proceso no está confinado en una sola máquina. Es por esto y debido a la masividad de los recursos computacionales puestos en operación para el despliegue de estas aplicaciones y la generalización de los modelos de servicio y de interacción de dichos componentes, que resulta tan importante apoyarse en el concepto de arquitectura, específicamente para el diseño y estructuración de los sistemas distribuidos.

1. Definiciones de arquitectura de los sistemas distribuidos

Una arquitectura dentro del mundo de la tecnología está orientada a encontrar patrones de agrupación o configuración de componentes. Estas arquitecturas generalmente están definidas por parámetros necesarios para brindar una solución, ya sea por medio de un listado de requisitos o también, en casos más específicos, en arquitecturas de referencia (Bass L & P, 2010,paginas 39 – 51).

Figura 1. Centro de computo de altas prestaciones (Datacenter)

Fuente: ralwel

Los sistemas distribuidos están compuestos de *software* para aplicaciones distribuidas, bases de datos distribuidas y sistemas operacionales distribuidos (requeridos para las organizaciones computacionales en clúster, más comúnmente llamados "granjas de servidores – HPC (High Perfomance Computing)

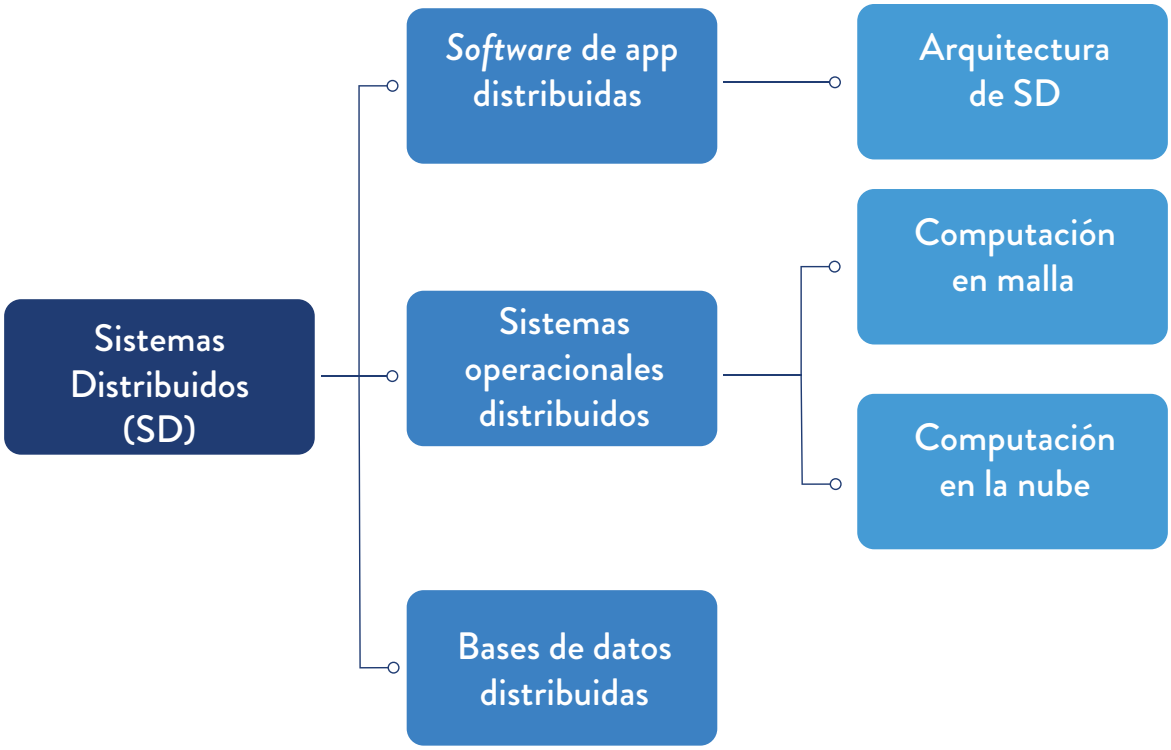

Figura 2. Elementos de los sistemas distribuidos

Fuente: elaboración propia

1.1. Arquitecturas de los sistemas distribuidos

En este Escenario se exponen dos tipos de arquitecturas: cliente/servidor y objetos distribuidos. Cada una de estas arquitecturas están orientadas a paradigmas de comunicación y agrupación; cuando se habla de comunicación y agrupación se refiere a la comunicación que existe entre los componentes o módulos de *software*, que generalmente tiene varios propósitos:

Intercambio de información de los componentes: información necesaria relacionada con los datos de la aplicación, resultados de bases de datos, llamado a métodos de los servicios remotos.

Coordinación y articulación de componentes: cada una de las arquitecturas presentadas en este Escenario tienen mecanismos de sincronización muy necesarios, debido a que, generalmente, los componentes existen en contextos de memoria diferentes y no pueden comunicarse fácilmente, como los HPC que se comunican a través la red de datos usando MPI (*Message Passing Interface*).

Protocolos de comunicaciones: los protocolos de comunicaciones de red son la base fundamental para el trabajo de diferentes computadores en red y es la base habilitadora del desarrollo de un sistema distribuido.

Depende del método de transmisión, recepción y respuesta de la información entre los componentes de *software*, que son las maneras de transmitir. La transmisión de datos debe ser controlada haciendo uso del tiempo, dado que el servidor que recibe requiere conocer el tiempo (t1) en el cual debe comenzar el proceso de recibo de datos. Así, existen dos formas de transferir datos:

a. **Transmisión sincrónica:** hace referencia al modo de transmisión en el cual, tanto emisor como receptor, intercambian mensajes intercaladamente; esto quiere decir que una vez se inicia la conexión, el emisor envía el mensaje y el receptor acusa el recibo y envía su propio mensaje. Esto sucede hasta que alguno de ellos finaliza la conversación según el funcionamiento del protocolo específico.

b. **Transmisión asincrónica:** a diferencia del anterior, en el que ambos participantes de la comunicación deben estarse respondiendo para que el protocolo pueda seguir su curso, en el caso de la comunicación asincrónica no se necesitan confirmaciones de recepción ni acuerdos en el sostenimiento de la comunicación; esto quiere decir que, en cualquier momento y sin aviso previo, cualquier participante puede abandonar la conversación.

1.2. Ventajas

Cada una de las opciones de arquitectura de sistemas distribuidos que se van a explorar comparten las siguientes características:

- **Compartir recursos.**

  La razón de ser de un sistema distribuido es compartir los recursos que atienden un servicio. Se comparten recursos de *hardware,* como los servidores, la memoria, los discos, la red, archivos, etc., los cuales se intercomunican a través de una red de computadores, ya sea una red Lan (redes locales), WAN (redes de banda ancha) o mixta.

- **Apertura o middleware.**

  Son sistemas *open source*, diseñados con protocolos estándar para que las diferentes arquitecturas de *hardware* y *software* se puedan comunicar a través de una red.

- **Concurrencia.**

  Agrupa todo aquello que opera sobre un servidor o un servicio a la vez. Por lo que en sistemas distribuidos, la concurrencia es mandataria, dado que muchos procesos necesitan operar en el mismo instante de tiempo en diferentes servidores de cómputo, usando la red y comunicándose entre sí.

- **Escalabilidad.**

  Permite que una organización computacional distribuida pueda ir agregando o quitando más componentes de acuerdo con la necesidad y la demanda de los servicios que presta el sistema.

- **Tolerancia a fallos.**

  Una de las características que hacen que los sistemas distribuidos sean exitosos es la tolerancia a fallos, ya que el servicio debe seguir prestándose sin contratiempo; eso quiere decir que el sistema debe estar en capacidad de soportar fallos en *hardware* y *software*, como ocurre, por ejemplo, cuando se daña un disco, se va la luz, se daña un equipo de aire acondicionado, se daña un *switch* o falla un programa.

1.3. Desventajas

Middleware es un *software* desarrollado en Java, que remplaza a las capas de presentación y sesión en el modelo OSI (Figura 3), para gestionar la interoperabilidad (comunicación entre máquinas y *software* de distintos tipos) en una organización computacional lo más heterogénea posible.

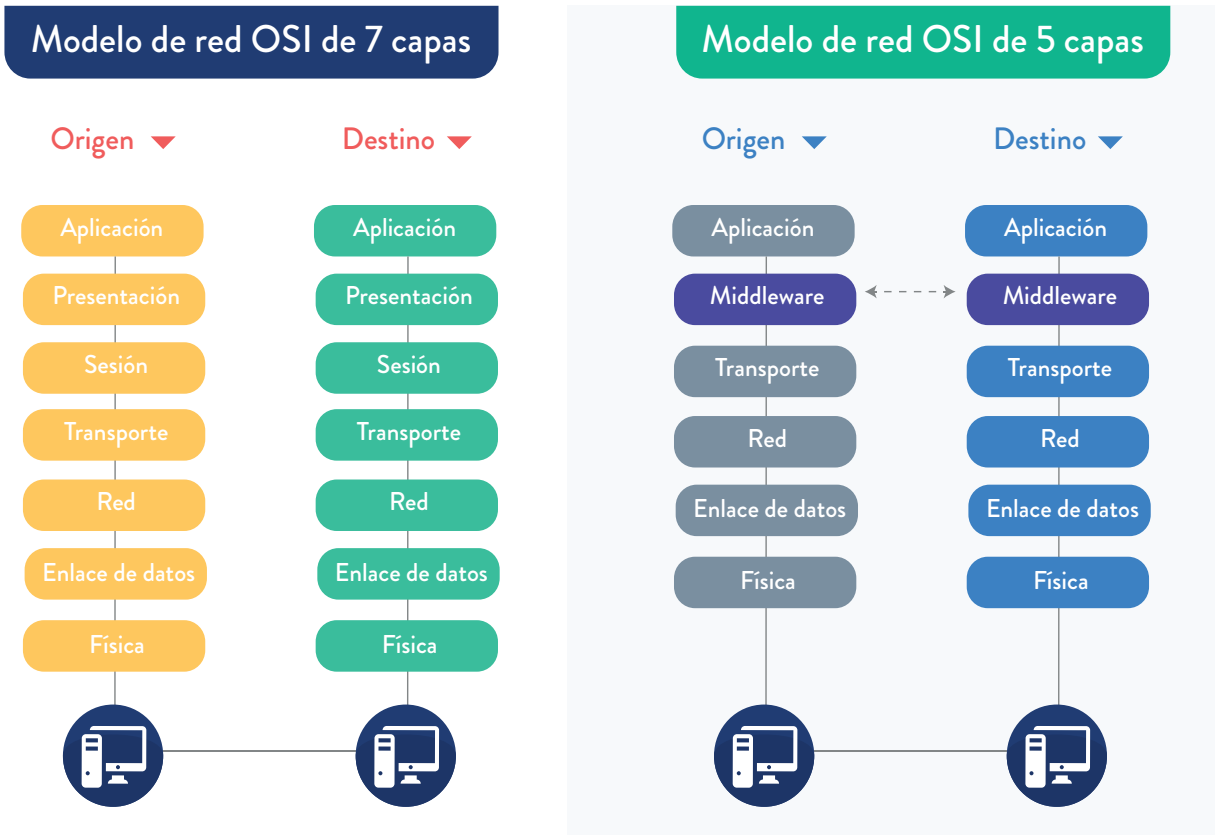

Figura 3. Modelo OSI de 7 y de 5 capas

Fuente: elaboración propia

Con el fin de evaluar una arquitectura distribuida, también es necesario contemplar las desventajas de dichas arquitecturas. Estas son algunas de las características que se pueden encontrar al momento de evaluar las desventajas de las arquitecturas distribuidas:

- **Seguridad:** como un sistema distribuido es cliente/servidor, entonces es accesado desde diferentes puertos-clientes, por lo que muchos usuarios no deseados pueden ejecutar procesos de escucha sobre la red, con el fin de extraer información. Mantener la seguridad es uno de los procesos complejos de los sistemas distribuidos.
- **Complejidad:** en un sistema distribuido, el *hardware* computacional puede ser de diferente tipo, según donde corren sistemas operacionales distintos. Al quedar eliminada la barrera de interoperabilidad entre la heterogeneidad de *hardware* y *software* gracias al *middleware*, los fallos o errores de una máquina pueden propagarse a las demás de su tipo, lo cual implica redoblar esfuerzos para mantener el sistema en funcionamiento.

- **Manejabilidad:** *middleware* es la nueva capa de comunicaciones instaurada en el modelo OSI (Organización Internacional para la Estandarización) de 7 capas, en remplazo de las capas de presentación y de sesión, para mejorar la interoperabilidad entre sistemas operacionales disímiles, *hardware* de red y de cálculo, compiladores y *software* diferente.
- **Impredecibilidad:** la predicción en un sistema distribuido es casi nula, eso los hace ser complejos, y además exige mantenerlos sincronizados todo el tiempo. Para poder acertar en algo sobre la predicción de fallos, hay que tener en cuenta aspectos tales como: carga total del sistema, carga del sistema de red y la organización del mismo sistema. Las cosas en un sistema distribuido cambian a gran velocidad, por lo que la respuesta a una petición por un fallo o de un usuario puede variar mucho.

2. Arquitectura cliente/servidor

La arquitectura cliente/servidor es una de las arquitecturas más simples al momento de crear un sistema distribuido. Esta arquitectura requiere mínimo un servidor, el cual es el que proporciona los o el servicio a los clientes, estos clientes generalmente están conectados a una red, que tiene conexión o visibilidad con los servidores para poder utilizar el servicio que estos prestan.

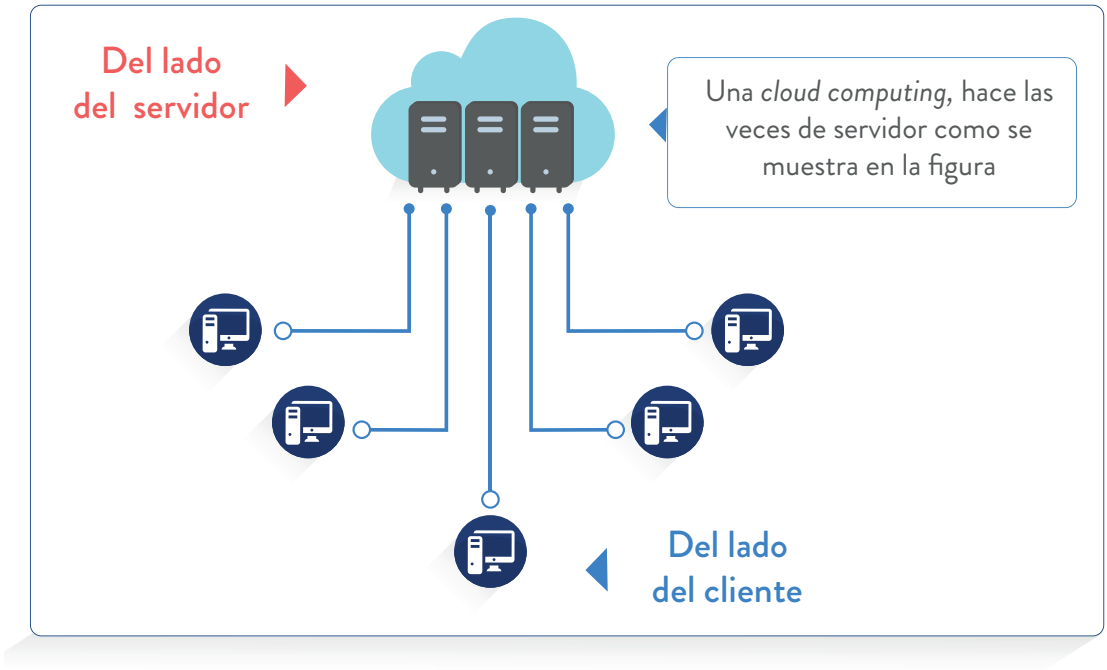

Figura 4. Arquitectura cliente/servidor

Fuente: Rajesh Rajendran Nair

El esquema de la arquitectura de este tipo ubica al servidor en la nube en la parte superior y a los clientes en el lado opuesto.

Servidor: conjunto de procesos, componentes de *software* y componentes de *hardware* que se presentan como una sola entidad y prestan un servicio.

Cliente: conjunto de módulos, librerías, programas, interfaces y dispositivos que se conectan al servicio prestado por el servidor.

Este tipo de arquitectura recalca que para cada uno de los procesos, cliente y servidor son diferentes y deben correr en instancias distintas. Sin embargo, varios procesos servidores se pueden ejecutar en el mismo equipo de cómputo, deduciendo que no en todos los casos existe una correspondencia entre un proceso y un procesador en la ejecución del sistema.

Generalmente, el diseño de sistemas del tipo cliente/servidor debe corresponder a una lógica de desarrollo, estructurada en una aplicación que contiene tres capas, teniendo en cuenta en donde se encuentran ciertas funcionalidades. Estas capas son:

- La capa de *front-end* o de usuario, relacionada con la interfaz de usuario y los componentes que permiten que el usuario realice la interacción con el sistema.
- La capa de negocio, en la cual los procesos de unión, modelamiento, interpretación y traducción de datos, y otros procesos relacionados se llevan a cabo con el propósito de aplicarlos a la gestión de esta capa.
- La capa de persistencia de datos, en donde se realizan todas las operaciones relacionadas con las consultas, inserciones y, en general, toda la interacción que se hace con las bases de datos que tiene el sistema.

Figura 5. Arquitectura a tres capas

Fuente: elaboración propia

Dentro del desarrollo de un sistema centralizado no es estrictamente obligatorio que estas capas se encuentren definidas específicamente, sin embargo, al momento de construir un sistema distribuido es necesario que se realice una clara distinción en cada una de ellas, lo anterior por la razón de que si es necesario al menos se puede intentar un tipo de escalabilidad horizontal orientada a distribución.

2.1. *Request-reply*

Una parte importante para el funcionamiento de una arquitectura es el esquema de funcionamiento. En el caso de cliente/servidor es la estructura más simple de comunicación que hay, y se conoce como *request-reply* o petición y respuesta, respectivamente.

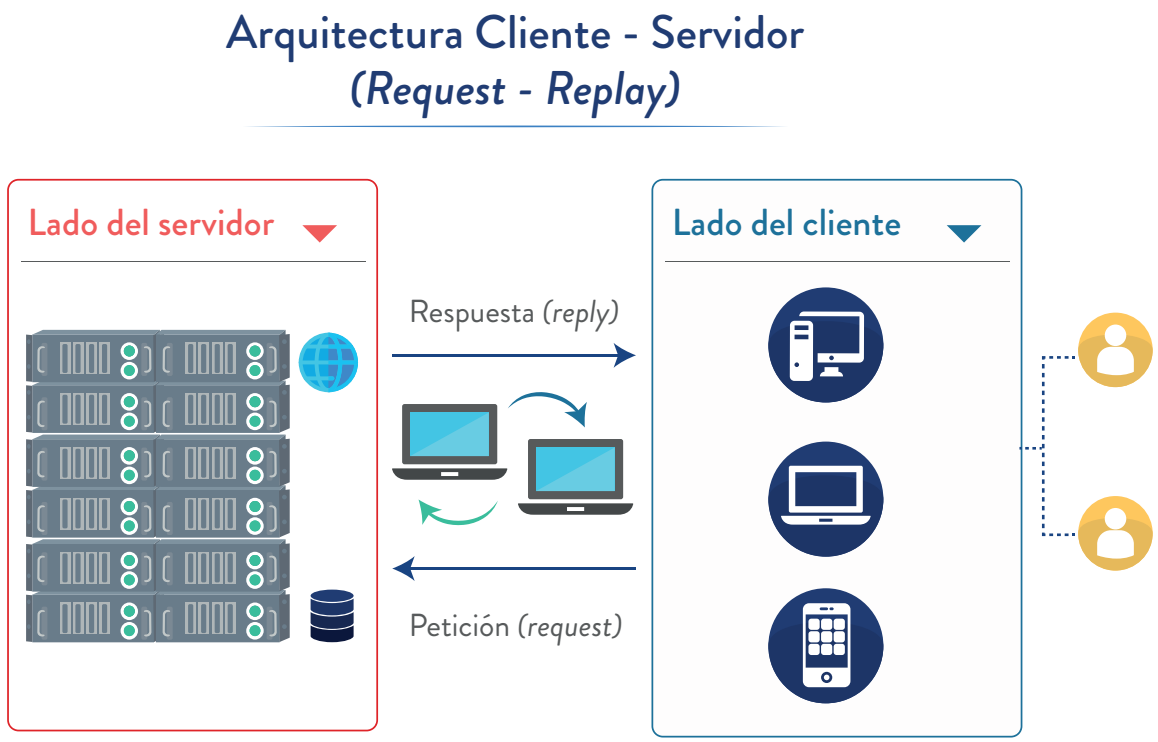

Figura 6. Arquitectura a tres capas

Fuente: elaboración propia

En el esquema de petición y respuesta (*request-reply*) se garantiza que por cada petición haya una respuesta, para que el esquema funcione correctamente.

Dentro de la estructura de comunicación de cliente/servidor hay dos tipos de mensajes; la petición, la cual es un mensaje que inicia en el cliente cuando consulta información o servicios al servidor (el cliente crea la conexión hacia el servidor y espera hasta recibir una respuesta), y la respuesta a la petición.

No puede haber respuestas arbitrarias de un servidor sin que exista previamente una petición por parte del cliente. En el caso en que el servidor no pueda responder por efectos de fallo o demora, existe la posibilidad de implementación de un *timeout* con el fin de romper una espera infinita y que el cliente pueda volver a hacer la petición al servidor.

La arquitectura cliente/servidor se puede separar en dos subcategorías, en las que cada una se diferencia particularmente por la capacidad del cliente y las responsabilidades que este maneja, por ejemplo:

- Cliente pesado/gordo/rico
- Cliente ligero/delgado/pobre

2.2. Cliente gordo

La arquitectura de cliente pesado se caracteriza, especialmente, porque los servicios del modelo de datos están exclusivamente en el servidor, esto quiere decir que el cliente tiene la responsabilidad de realizar tareas en la capa de negocio y procesamiento. Este esquema es muy utilizado en aplicaciones que necesitan realizar tareas en el cliente, y sobre todo en las que este cliente tiene un acceso de red con más prestaciones. Estas aplicaciones se pueden encontrar generalmente en:

- Aplicaciones móviles
- Cajeros automáticos
- Usadas dentro de Redes de Área Local (LAN)
- Sistemas accesados por una VPN
- Sistemas Legacy

2.2.1. Ventajas

Una de las ventajas más significativas del modelo de cliente gordo o pesado es que utiliza el poder de cómputo disponible del cliente en tareas que son sencillas y relegan la parte crítica de la funcionalidad hacia el servidor. Esto quiere decir que para poder quitarle carga de procesamiento al servidor, parte de esa carga no crítica de negocio se pasa al cliente, y así se puede mantener el servidor libre para que pueda aceptar más clientes.

De acuerdo con lo anterior, se puede definir que el servidor se vuelve más crítico para la operación, ya que este solo estaría encargado del manejo y procesamiento de las transacciones importantes. En otras palabras, el servidor se encargaría de todas las operaciones relacionadas con la sincronización de datos por parte de múltiples clientes y, por supuesto, de la capa de gestión frente al motor de base de datos utilizado en el sistema.

Es importante aclarar que para que esto sea posible el motor de base de datos que esté utilizándose en este tipo de arquitectura requiere soporte para transacciones y control de concurrencia.

Modelo cliente/servidor
(Modelo de cliente Pesado/Gordo/Rico)

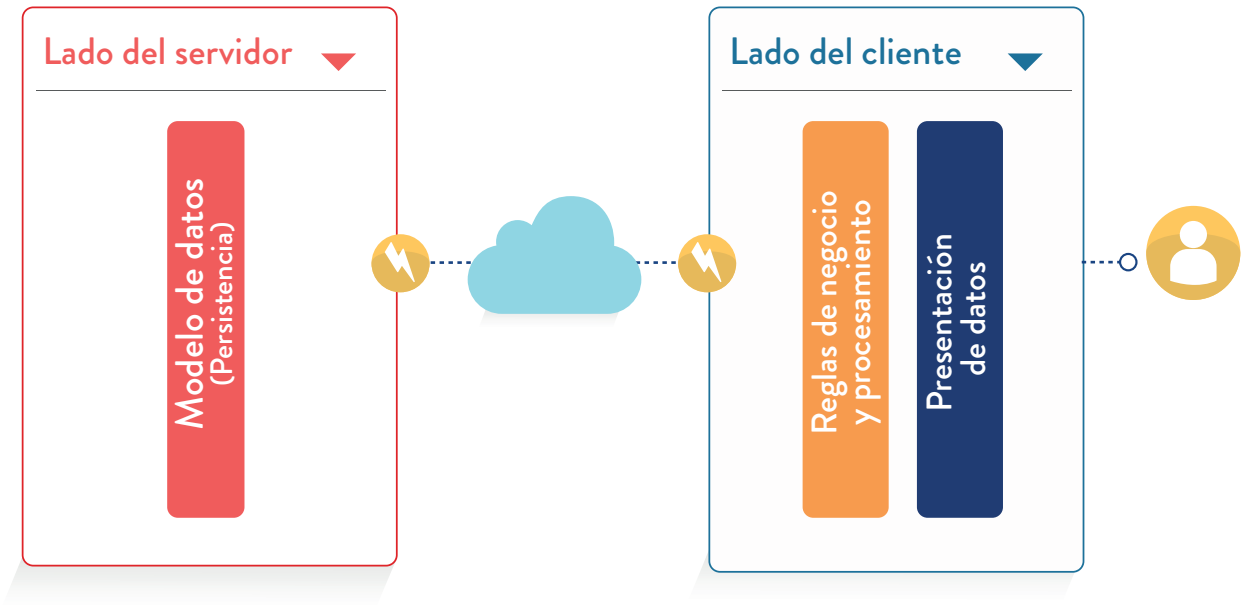

Figura 7. Arquitectura cliente/servidor – Cliente pesado

Fuente: elaboración propia

Un ejemplo clásico de uso de un sistema de cliente gordo es el que está distribuido a través de las redes bancarias del país con los cajeros automáticos, en donde el cajero es un computador con un *software* cargado que actúa como cliente, el cual mantiene conexión casi que permanente con el servidor, y tiene el motor de base de datos de los individuos que pueden usar esos sistemas bancarios.

Es importante entender que el computador que se encuentra en el cajero automático no se conecta de forma directa a la base de datos del banco, para esto hay un proceso intermedio conocido como "monitor de teleproceso" que es el que se encarga de hacer el puente con el servidor que tiene la base de datos.

Por otro lado, antes de comunicar con la base de datos del banco, el cajero automático realiza una serie de validaciones y procesamientos internos con el fin de no recargar el servidor con operaciones específicas del cliente que está utilizando los servicios.

El gestor transaccional o monitor de teleproceso es un *middleware* que se encarga de organizar el proceso de comunicaciones entre el servidor central y los clientes remotos, mediante la serialización de las transacciones de los clientes, para ser procesados en el servidor central, el cual a través del programa servidor o demonio de transacciones, hace el acceso a la base de datos.

La serialización de transacciones quiere decir que en un sistema transaccional espejado (redundante) con un motor de base de datos que tiene activos los *archive* logs permite que el sistema se recupere en la eventualidad de que ocurran errores, sin que se corrompan los datos del sistema.

2.2.2. Desventajas

Así como existen ventajas en la implementación de un cliente pesado, también existen serias desventajas que el arquitecto de la solución debe tener muy presentes. Una de estas desventajas es que al ser un modelo de cliente/servidor, existe la posibilidad de que haya muchos clientes y en la medida que estos tengan más responsabilidad, cada uno se vuelve más complejo. Por tal razón, la posibilidad de un fallo y su respectivo soporte se vuelve más tediosa e incrementa el nivel de carga en la operación del sistema como un todo.

Adicionalmente, al momento de existir una actualización de la aplicación, ya sea por motivos de adición de nuevas funcionalidades o temas relacionados con la seguridad, este esquema se convierte en un problema crítico para la organización debido a la pérdida por la demora en la actualización de los clientes remotos, sobre todo si hay cientos o miles de clientes asociados.

2.3. Cliente delgado

El modelo de cliente ligero tiene como propósito todo lo contrario al cliente gordo, que es relegar toda la mayor capacidad de procesamiento al servidor con el fin de dejar al cliente con la menor cantidad de tareas posibles. Por eso también se conoce como esquema de cliente bruto.

El cliente solo está encargado de la capa de presentación de datos del *software*, esto quiere decir que lo único que tiene son las ventanas o la posibilidad de desplegar una interfaz, pero toda la carga se encuentra en el servidor.

2.3.1. Ventajas

Una arquitectura de dos capas con clientes livianos es la más sencilla que puede utilizarse cuando los sistemas heredados centralizados migran a una arquitectura cliente/servidor.

Uno de los casos de éxito más visible de esta arquitectura es el mismo protocolo WWW, el cual ayudó a que Internet se popularizara. Este funciona de manera que todos los datos y capacidad importante de control o lógica de negocio permanecen en el servidor y el usuario solo necesita de un navegador web para acceder a múltiples aplicaciones sin necesitar nada más para su consumo.

Esto trae consigo una ventaja significativa y quizá la más importante: la capacidad de hacer cambios en millones de clientes de forma rápida y efectiva ya que, al encontrarse la lógica de negocio en el servidor, los cambios se tienen que realizar en pocos lugares o a veces en uno solo.

Este tipo de arquitectura es una de las más simples de implementar, gracias a que casi toda la gestión se encuentra centralizada.

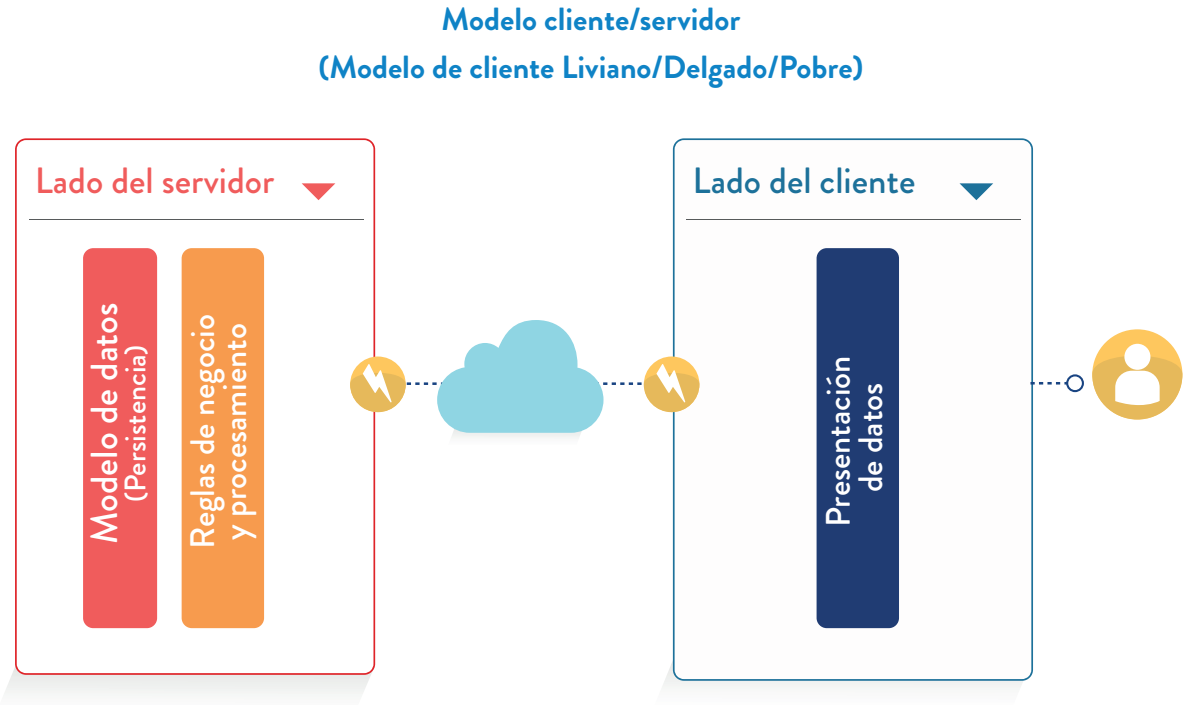

Figura 8. Arquitectura cliente/servidor – Cliente ligero

Fuente: elaboración propia

Es de notar que el cliente ligero es el más utilizado a gran escala en la actualidad, ya que no se requiere que haga grandes cálculos.

2.3.2. Desventajas

Una desventaja importante en la arquitectura cliente/servidor y el cliente ligero es que la ubicación de los componentes de negocio y de procesamiento de datos pone una carga significativa en el servidor. Esto significa que la arquitectura que presta servicios complementarios al servidor debe ser lo suficiente robusta como para soportar fallas súbitas del sistema.

Por otro lado, al ser un sistema distribuido, existe una dependencia muy grande a la red de computadores, y más si debe existir una comunicación constante entre cada una de las capas exploradas previamente, por lo que la carga en el tráfico de la red se convierte en un tema muy crítico al diseñar y operar un sistema distribuido de estas características.

3. Arquitectura de objetos distribuidos

Los objetos distribuidos modifican el paradigma de la definición de responsabilidades y permite que tanto clientes como servidores intercambien servicios, esto quiere decir que un cliente puede consumir servicios de un servidor y ese mismo servidor puede volverse cliente para poder consumir servicios del cliente, y este, a su vez, tiene la posibilidad de prestar servicios.

Es una arquitectura útil en grandes sistemas, aunque una idea de Windows 10 es convertir a los computadores de las personas en servidores de sus actualizaciones, para que no todos busquen las actualizaciones en los servidores de Microsoft, sino que se puedan encontrar en el computador del vecino que las haya bajado antes. Esto tiene serias implicaciones como que el computador y la red del vecino se degraden.

Esto permite que exista una flexibilidad, sobre todo en el tema de roles y permite construir servicios mucho más escalables de forma más natural. Sin embargo, de la misma manera que la arquitectura cliente/servidor, los consumidores deben tener conocimiento previo de donde está publicado el servicio y cómo usar cada uno de estos servicios remotos.

En este caso, y debido a que no hay una distinción entre el concepto de cliente y de servidor, las entidades que existen en esta arquitectura se conocen como **Objetos**. Los objetos realizan llamadas a los servicios sin tener una frontera entre lo que es un cliente y un servidor, que en esta arquitectura se conocen como:

- Receptor de un servicio
- Proveedor de un servicio

Estos objetos se distribuyen en diferentes contextos de memoria, las cuales pueden distribuirse a través de diferentes computadores en una red y conectarse por intermedio de una capa de *software* transversal en todas las máquinas, esta capa intermedia es la que se encarga de interceder, coordinar y redireccionar las peticiones de los receptores a los proveedores de servicio. Esta composición de *software* es una línea adicional del *middleware*.

El *middleware* se convierte entonces, para la arquitectura de objetos distribuidos, en una interfaz, la cual le garantiza transparencia de localización y de comunicación a los objetos que componen los servicios o funcionalidades del sistema (Coulson & Baichoo, s.f.). Sin embargo, esta no es la única tarea de un *middleware*. Entre las características que tiene, también controla memoria y las instancias de estos objetos en las respectivas máquinas del sistema.

Debido a lo anterior, la arquitectura de objetos distribuidos puede verse con un modelo más abstracto, el cual permite estructurar un sistema complejo de una forma más simple, teniendo en cuenta la proporción de funcionalidades por medio de la combinación de servicios más pequeños, con el fin de que al final de un proceso se pueda prestar una funcionalidad más grande y compleja al usuario, para al final distribuir los objetos que componen el servicio en máquinas diferentes con el fin de lograr escalabilidad en el mismo.

Así como la arquitectura cliente/servidor tiene ventajas y desventajas, la arquitectura de objetos distribuidos también. A continuación se presentan las ventajas y desventajas del esquema de objetos distribuidos.

3.1. Ventajas

Esta arquitectura permite al diseñador tomarse un poco más de tiempo en la fase de diseño debido a la flexibilidad de este, ya que la ubicación de estos servicios en el componente de infraestructura podría ser de vital importancia para la integridad de la aplicación. Adicionalmente, los objetos distribuidos permiten que al momento del despliegue o en caso de una falla el administrador de este servicio pueda desplegar cualquier objeto en cualquier máquina del sistema.

La diferencia entre los modelos del cliente flaco y el cliente gordo es notable dado que se debe decidir en donde dejar el objeto de reglas del negocio, si del lado del cliente o del lado del servidor. Todo depende de las características, lo pesadas que sean las aplicaciones y la cantidad de datos que tengan que transmitir.

Un servicio de cajeros automáticos no puede pertenecer al modelo gordo porque la base de datos está en la máquina central y se tornaría muy pesado hacer cálculos en el cajero para mandar cada cosa por la red “n” veces, congestionaría la red. No sucede lo mismo con el modelo de objetos distribuidos porque el paradigma es diferente, dado que la escalabilidad de los objetos es horizontal, pero depende del tipo de aplicaciones.

En la Figura 9 se puede observar cómo el servidor de objetos 01 puede ofrecer el servicio S(01), que puede ser invocado por los demás servidores que quieran consumir ese servicio.

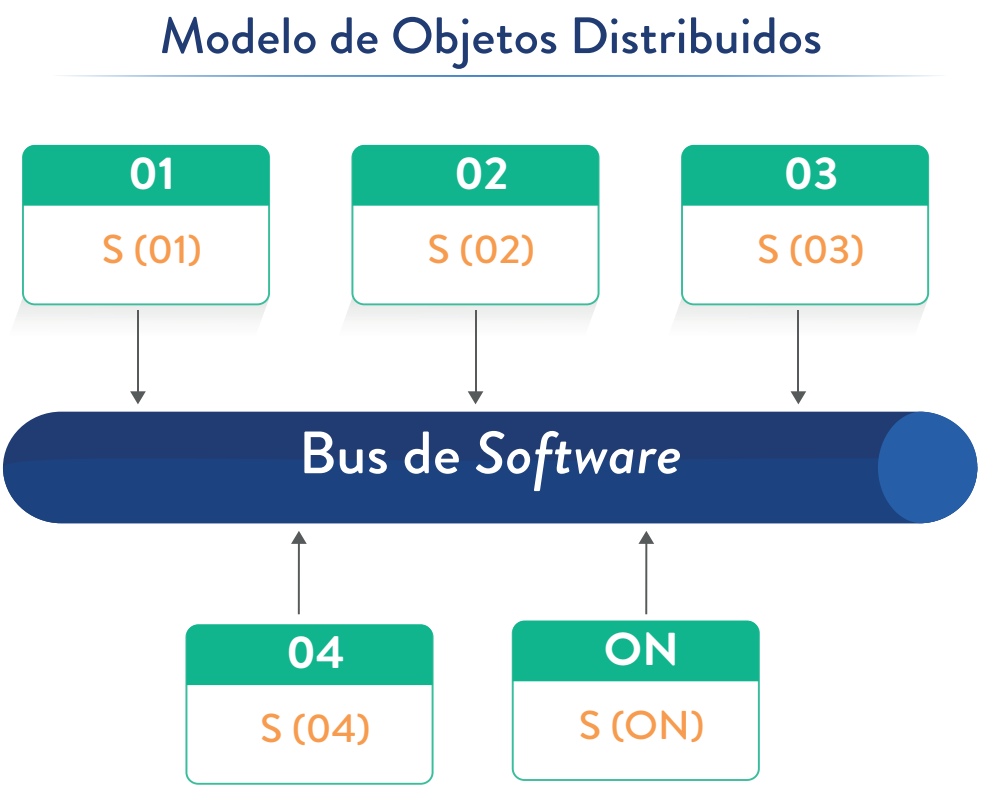

Figura 9. Arquitectura de objetos distribuidos

Fuente: elaboración propia

El anterior es un esquema de arquitectura básica de objetos distribuidos, en el cual cada máquina contiene una instancia de un objeto diferente y su respectiva publicación del objeto. Estos objetos están conectados por medio de un bus de *software* el cual se encarga de hacer el puente entre los objetos y permite su respectivo uso.

En la actualidad, en la industria del *software* se han desarrollado objetos en lenguajes de programación distintos, de tal manera que puedan comunicarse y proporcionar entre ellos los servicios que ofrecen uno y que el otro necesita. Tienen la ventaja de que pueden ir añadiendo a través de la red, nuevos objetos si el rendimiento del sistema no se afecta.

3.2. Desventajas

La desventaja fundamental del modelo de objetos distribuidos es que el diseño, la sincronización y el desarrollo son altamente complejos y no es fácil hacerle mantenimiento a esta clase de *software*.

4. Fundamentos de *sockets*

¿Qué es un *socket*?

Para poder comprender la funcionalidad de una transacción, primero se debe entender el concepto de *socket* y su funcionamiento.

Uno de los grandes aportes de la Universidad de Berkeley a las operaciones computacionales en línea fue la definición del *socket*. Para entender este concepto, de una manera sencilla, se debe hacer un paralelo con una conversación por teléfono o una conversación usando un chat, en donde cada uno de los participantes de dicho evento está en lugares diferentes.

En general, un *socket* es un programa que se desarrolla para servir, es decir, para que haga un trabajo desde el lado del computador que opera como servidor, a este programa se le llama *Socket Server* y debe ser capaz de resolver las peticiones de otro programa instalado en otro u otros computadores remotos que se llaman *Socket Cliente*, y es el que hace peticiones al servidor.

Se denomina dominio de un *socket* al conjunto de *sockets* con los cuales se establece comunicación. Los protocolos de comunicación usados en los *sockets* son: TCP, UDP.

4.1. Tipos de *sockets*

4.1.1. Sockets UDP

Se les llama *sockets UDP* o Protocolo de Datagramas de Usuario (*User Datagram Protocol*). Características:

a. Es un protocolo no orientado a conexión por lo tanto es asincrónico, un ejemplo es el correo electrónico. El emisor envía un mensaje, pero el receptor puede que no lo lea al instante, lo lee en otro momento.

b. Envía paquetes de datos conocidos como datagramas y no ofrece garantía de que lleguen al destino.

c. Protocolos a nivel de aplicación que son el TFTP, DNS, etc.

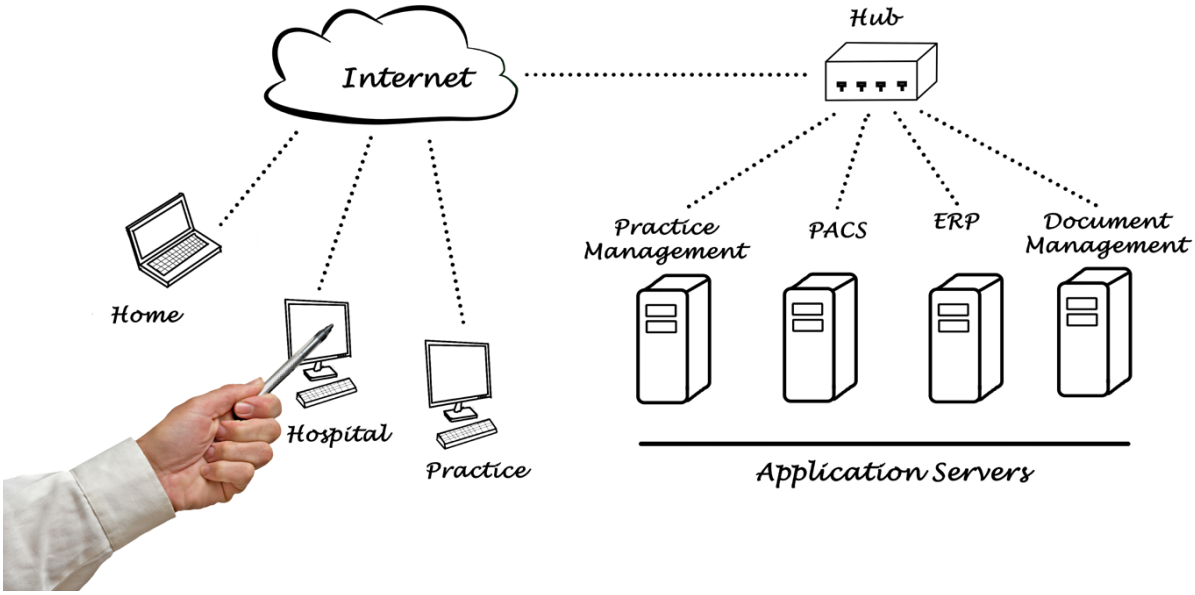

Figura 10. Diagrama de un *socket* cliente/servidor

Fuente: Vaeenma

En la figura 10 se puede observar que del lado derecho están las computadoras servidoras las cuales tienen las aplicaciones *server*, estas son usadas por los computadores clientes que tienen los programas clientes y que están en sitios remotos. Por ejemplo, el servidor de manejo de documentos puede estar en Moscú y el cliente de prácticas está en el Campus del Poli. Los dos programas deben conversar usando las redes.

4.1.2. *Sockets* TCP. Protocolo de control de transmisiones.

Características:

- Es orientado a conexión.
- Envía conjuntos de *bytes* en bloques ordenados, desde el origen al destino y garantiza que lleguen todos.
- El nivel de aplicación OSI que los usa son: https, telnet, http, ftp, etc.

Algunos tipos son:

a. SOCK_DGRAM/*: para mensajes en modo asincrónico, el cual envía datagramas de tamaño fijo y limitado, usa el protocolo UDP */.

b. SOCK_STREAM/*: se usa para comunicaciones bidireccionales, en modo conectado, es confiable. Usa el protocolo TCP. La transmisión de datos se hace de manera ordenada */.

c. SOCK_RAW/*: se usa con protocolos de red más bajos como IP. Es un tipo de uso reservado para súper usuarios (root) */.

d. SOCK_SEQPACKET/*: se usan en comunicaciones bidireccionales, altamente confiable, pada datagramas de longitud fija, la transmisión de datos se hace de manera ordenada */.

Los tipos más usados, son los tipos “a“ y “b”.

***Sockets* TCP en Java**

Arquitectura requerida para programar una ampliación con sockets.

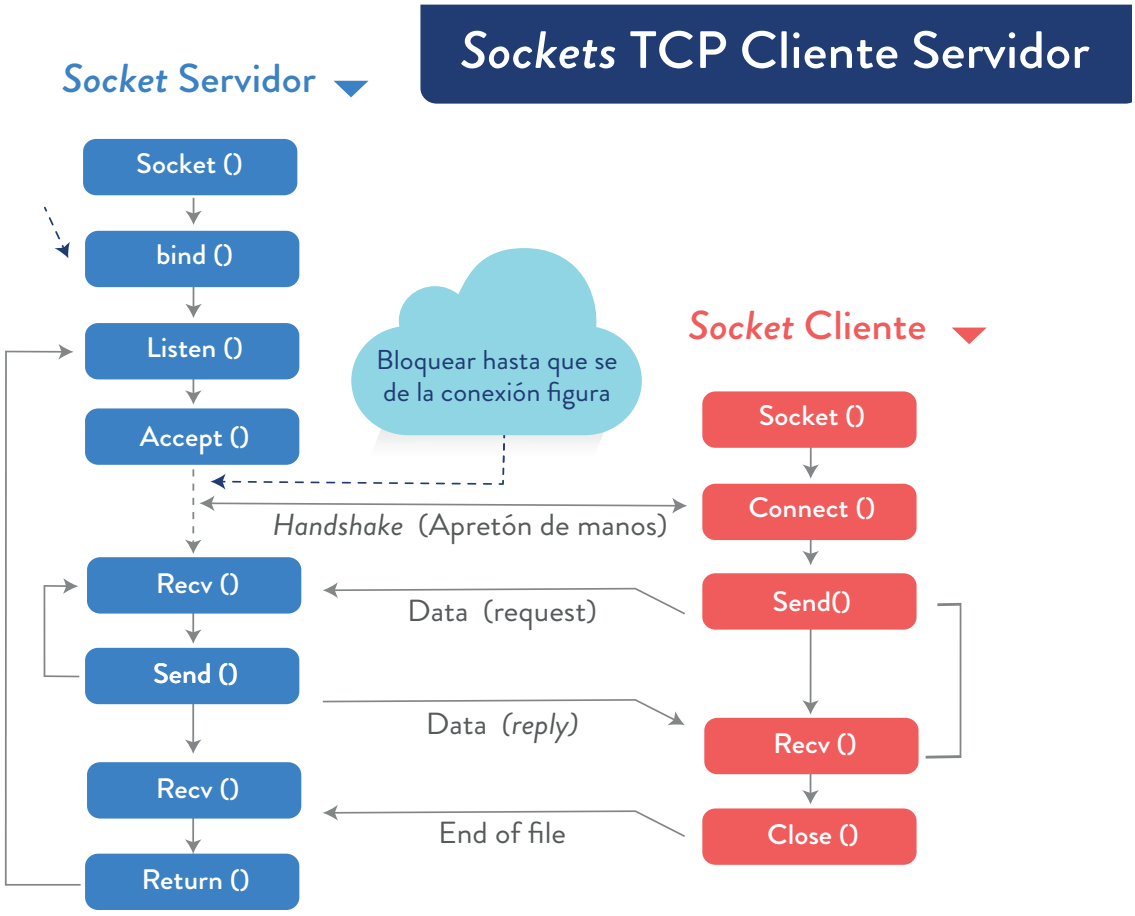

Figura 11. Diagrama de comunicación entre *sockets*

Fuente: elaboración propia

4.1.3. *Socket* cliente

Por definición, los *sockets* TCP son orientados a conexión, es decir, son sincrónicos, como las llamadas telefónicas. Como son sincrónicos implica que, para enviar y recibir datos, tanto el cliente como el servidor deben comunicarse y sincronizarse entre sí. Una vez establecida la conexión, los datos son envidos, recibidos y ordenados en el destino en el orden en que estaban en el origen, pero esto es garantizado por el protocolo TCP.

Como ejemplo se implementa un cliente ECO en java, donde el protocolo de conexión tiene la siguiente estructura:

Class ClienteTCP (computador servidor) (puerto del servidor) (mensaje)

Con la siguiente descripción:

- **Computador servidor:** es el nombre del dominio o dirección IP del computador servidor donde se está ejecutado el programa *Socket Server* ECO.
- **Puerto del servidor:** es el puerto por el que escucha el *Socket Server* ECO.
- **Mensaje:** es el mensaje que se quiere enviar al *Socket Server*. Recuerde que estamos armando el programa *Socket* Cliente.

4.1.4. *Socket Server*

```
import java.net.*;
import java.io.*;

/** Ejemplo que implementa un cliente de eco usando TCP. */

public class ClienteTCP {

  public static void main(String argv[]) {
    if (argv.length != 3) {
      System.err.println("Formato: ClienteTCP <maquina> <puerto> <mensaje>");
      System.exit(-1);
    }

    Socket socket = null;
    try {

      // Obtenemos la dirección IP del servidor
      InetAddress dirServidor = InetAddress.getByName(argv[0]);
      // Obtenemos el puerto del servidor
      int puertoServidor = Integer.parseInt(argv[1]);
      // Obtenemos el mensaje
      String mensaje = argv[2];

      // Creamos el socket y establecemos la conexión con el servidor
      socket = new Socket(dirServidor, puertoServidor);

      // Establecemos un timeout de 30 segs
      socket.setSoTimeout(30000);

      System.out.println("CLIENTE: Conexion establecida con "
              + dirServidor.toString() + " al puerto " + puertoServidor);
      // Establecemos el canal de entrada
      BufferedReader sEntrada = new BufferedReader(new InputStreamReader(
              socket.getInputStream()));
      // Establecemos el canal de salida
      PrintWriter sSalida = new PrintWriter(socket.getOutputStream(), true);

       System.out.println("CLIENTE: Enviando " + mensaje);
       // Enviamos el mensaje al servidor
       sSalida.println(mensaje);

       // Recibimos la respuesta del servidor
       String recibido = sEntrada.readLine();

       System.out.println("CLIENTE: Recibido " + recibido);
       // Cerramos los flujos y el socket para liberar la conexión
       sSalida.close();
       sEntrada.close();

     } catch (SocketTimeoutException e) {
       System.err.println("30 segs sin recibir nada");
     } catch (Exception e) {
       System.err.println("Error: " + e.getMessage());
     } finally {
       try {
         socket.close();
       } catch (IOException e) {
         e.printStackTrace();
       }
     }

  }
}
```

Figura 12. Programa cliente Java para hacer un ECO

Fuente: implementación del servidor

El programa *socket* servidor debe atender múltiples peticiones, lo cual se muestra en el seudo código de la Figura 12, requerido para elaborar un *socket server*.

Para el ejemplo se establecen los pasos que deben desarrollarse dentro del *socket* servidor:

a. Crear un *socket server* asociado a un número de puerto específico.

b. Crear un lazo infinito.

c. Invoca el método *accept* para escuchar las peticiones del cliente. Una vez que recibe una petición, crea una sesión del *socket* (un hilo) y genera un número que le devuelve al cliente para indicarle que le ha escuchado y que está atento a recibir los mensajes que el cliente le escriba. Recibe, analiza el mensaje, elabora el trabajo (en este caso es un eco) y lo devuelve al cliente.

d. El cliente se cierra, pero el *socket* servidor queda abierto escuchando nuevas peticiones.

```
import java.net.*;
import java.io.*;

/** Ejemplo que implementa un servidor de eco usando TCP. */

public class ServidorTCP {

  public static void main(String argv[]) {
    if (argv.length != 1) {
      System.err.println("Formato: ServidorTCP <puerto>");
      System.exit(-1);
    }

    try {
      // Creamos el socket del servidor
      // Establecemos un timeout de 30 segs
      while (true) {
        // Esperamos posibles conexiones
        // Establecemos el canal de entrada
        // Establecemos el canal de salida
        // Recibimos el mensaje del cliente
        // Enviamos el eco al cliente
        // Cerramos los flujos
      }
    } catch (SocketTimeoutException e) {
      System.err.println("30 segs sin recibir nada");
    } catch (Exception e) {
      System.err.println("Error: " + e.getMessage());
      e.printStackTrace();
   } finally {
       //Cerramos el socket
    }
  }
}
```

Figura 13. *Socket* servidor ECO

Fuente: elaboración propia

Referencias

Bass L, C. y P, K. R. (2010). *Software Architecture in Practice* (2da edición).

Cobb, J. A. y Gouda, M. G. (1997). *The request reply family of group routing protocols. IEEE Transactions on Computers*, 46(6), 659–672. Recuperado de: https://doi.org/10.1109/12.600824

Coulson, G. y Baichoo, S. (s.f.). A distributed object platform for multimedia applications. *Proceedings IEEE International Conference on Multimedia Computing and Systems 2, 122–126.* Recuperado de https://doi.org/10.1109/MMCS.1999.778193

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. McGraw-Hill.

Tanenbaum, A. S. y Steen, M. (s.f.). *Distributed systems: principles and paradigms.*

Referencias de imágenes

Vaeenma (s.f.). *Centro de computo de altas prestaciones* (Datacenter). [Fotografía]. Recuperado de https://www.gettyimages.es/detail/foto/medical-network-diagram-imagen-libre-de-derechos/517851799

Bsdrouin (2017). *Diagrama de un socket cliente/servidor*. [Fotografía]. Recuperado de https://pixabay.com/es/red-servidor-sistema-2402637/

INFORMACIÓN TÉCNICA

FACULTAD DE
INGENIERÍA, DISEÑO
E INNOVACIÓN

Módulo: Sistema Distribuidos

Unidad 2: Arquitecturas cliente-servidor y objetos distribuidos y arquitecturas SOA, punto a punto y Multiprocesador

Escenario 3: Arquitecturas cliente-servidor y objetos distribuidos

Autor: Alexis Rojas

Asesor Pedagógico: Jeimy Lorena Romero Perilla

Diseñador Gráfico: Katherinne Pineda Rodriguez

Asistente: Maria Elizabeth Avilán Forero