1.1 Introducción

Un **paradigma** es un patrón, ejemplo o modelo. Útil para clasificar elementos (aplicaciones) de la materia. Una aplicación puede comportarse como dos modelos diferentes.

Una **abstracción** es el proceso de encapsulación, esconder detalles. Se consigue gracias a herramientas o funcionalidades que permitan el desarrollo de software sin que el programador tenga que conocer las complejidades subyacentes.

El **middleware** es una capa de software que aumenta la abstracción, actúa como intermediario entre procesos independientes.

Lo que **distingue** a una aplicación distribuida de una convencional son las siguientes caracterísitcas: - **Comunicación entre procesos:** necesita la participación de varias entidades (*procesos*). Los procesos necesitan comunicarse entre ellos. Los **mecanismos de comunicación** consiguen esto. - **Sincronización de eventos:** el envío de mensajes debe ser sincronizado, es decir, si desde un proceso se envían datos tiene que haber otro proceso que esté esperando para recibirlos.

1.2 Paradigmas en Computación Distribuida

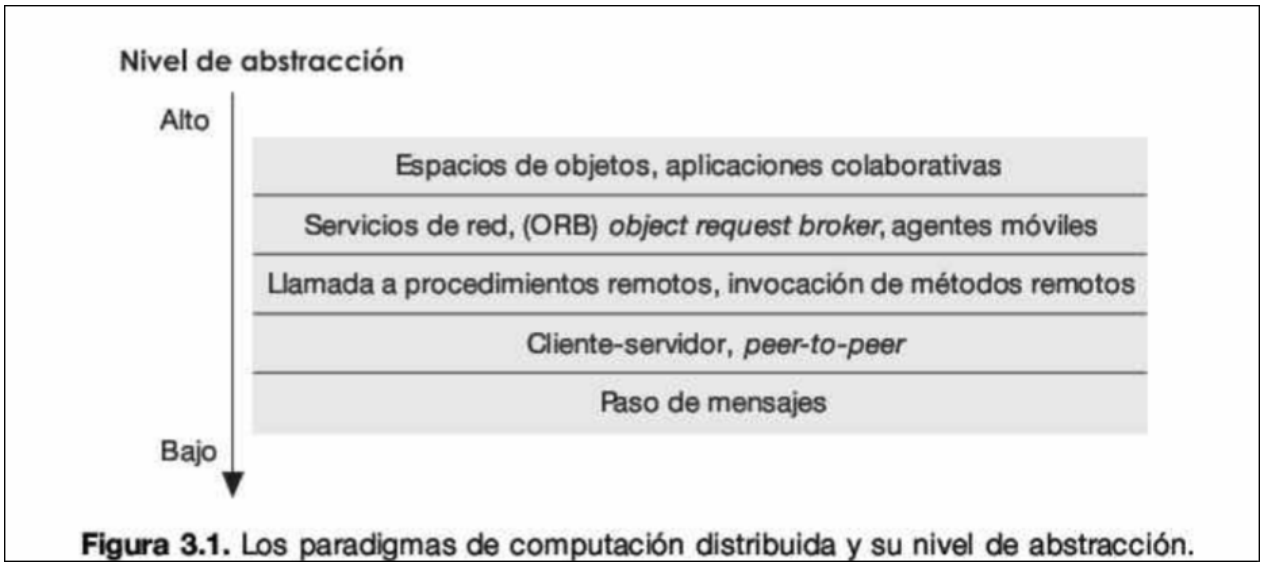


Figure 1: Pasted image 20250916092329.png

1.2.1 Paradigma del paso de mensajes

Es el más básico e importante. Un proceso envía un mensaje a otro proceso. El mensaje se entrega al receptor que lo procesa. Esta respuesta desencadena una serie de una serie de peticiones que darán lugar a más mensajes.

Las operaciones son **enviar** y **recibir**. También se necesitan **conectar** y **desconectar** para la comunicación orientada a conexión. Se realizan las operaciones como si

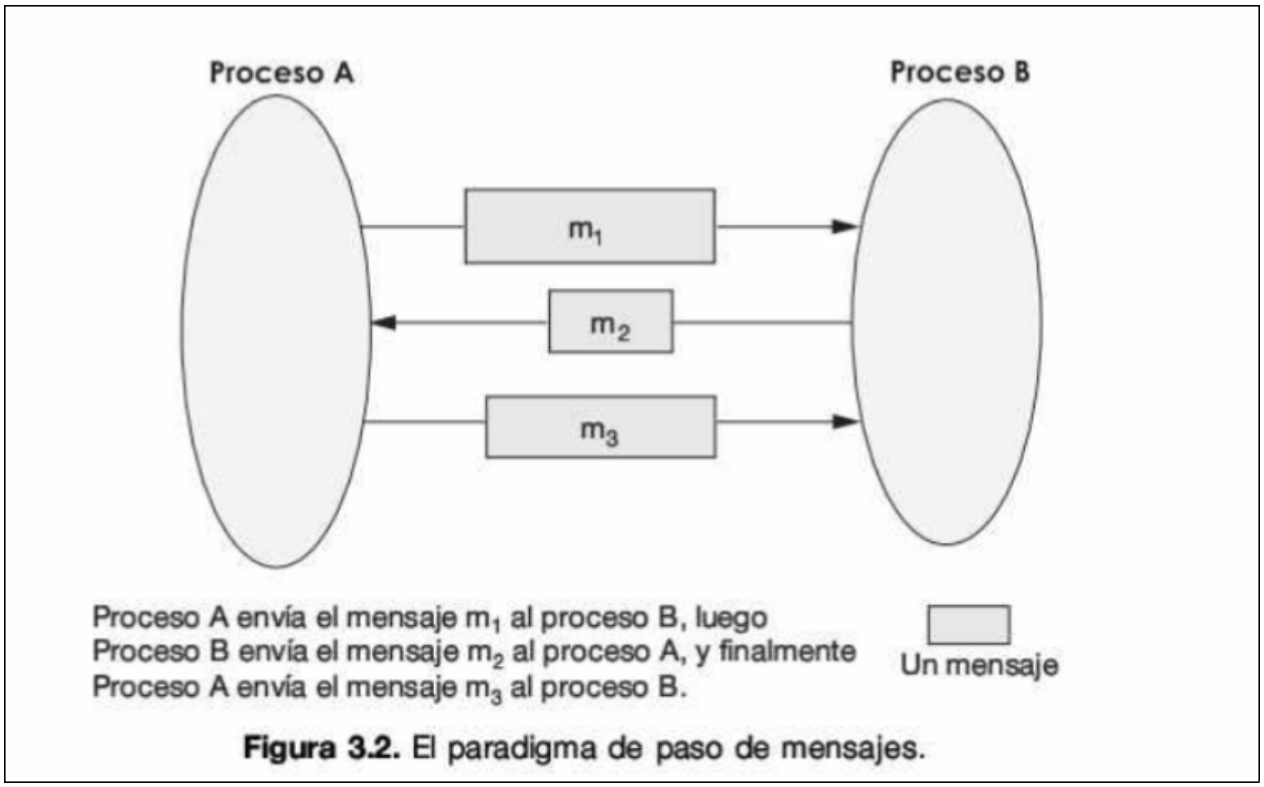


Figure 2: Pasted image 20250916092352.png

se tratara de un fichero: las operaciones sirven para encapsular el detalle de la comunicación a nivel del SO (*el programador puede hacer uso de ellas para enviar y recibir mensajes sin preocuparse por los detalles*).

La interfaz de programación de aplicaciones de sockets se basa en este paradigma. Los procesos intercambian información así: un emisor inserta un mensaje en el socket y el receptor extrae el mensaje del socket para leerlo.

> [!Recordatorio] Un **socket** es un punto de comunicación que permite el intercambio de datos entre dos programas a través de una red, como Internet o una red local. Funciona como una "puerta" que conecta aplicaciones, identificándose por una dirección IP y un número de puerto. Los sockets pueden usarse para enviar y recibir información de forma bidireccional, y son fundamentales para implementar protocolos de red como TCP y UDP, facilitando la comunicación entre clientes y servidores en diferentes dispositivos.

En este modelo **no está establecido el orden** en el envío y recepción de mensajes. Además surge el paradigma **Cliente-Servidor** para resolver que ambos procesos (emisor y receptor) estén escuchando o escribiendo a la vez.

1.2.2 Paradigma cliente-servidor

Uno de los más conocidos. Este modelo asigna **roles** diferentes a los dos procesos que colaboran. De esta manera la sincronización de eventos se simplifica. - **Servidor:**

proveedor del servicio, espera de forma pasiva la llegada de peticiones - **Cliente:** envía las peticiones al servidor y espera por las respuestas

Las operaciones serán aquellas necesarias para, en el servidor, esperar y aceptar peticiones y, en el cliente, emitir peticiones y aceptar respuestas.

Este modelo proporciona una abstracción eficiente para servicios de red por lo que se usan mucho en los servicios de Internet. *Ejemplos:* HTTP, FTP, DNS ...

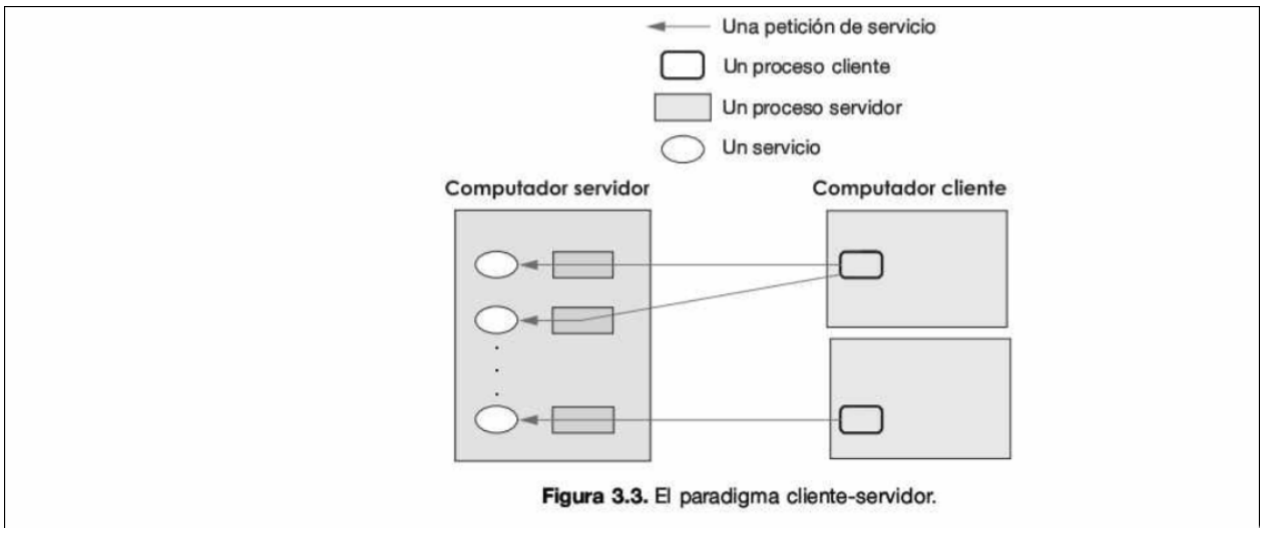


Figura 3.3. El paradigma cliente-servidor.

Figure 3: Pasted image 20250916093514.png

Protocolo de comunicación: descripción detallada de las situaciones que se encuentran los procesos y se establece el orden de la transmisión de mensajes.

Para que el protocolos sea un estándar público debe tener un *Request for comment* (*archivo*). Este será revisado muy atentamente, si lo aceptan se le asignará un número. Cualquier protocolo de cualquier aplicación tiene su RFC asociado.

Problemas: - **Cuello de botella:** al tener un solo servidor y muchos clientes - No da soporte para que el proceso **servidor inicie la comunicación**. Estos problemas se resuelven con el paradigma peer-to-peer.

1.2.3 Paradigma peer-to-peer (P2P)

Los procesos participantes interpretan los **mismos papeles** (pueden actuar al mismo tiempo como clientes y servidores en función del rol más eficiente para la red). Los ordenadores que en cliente-servidor eran clientes y servicios (intercambio de información, almacenamiento, etc.) entre ellos de manera directa. Cualquier participante puede enviar una petición a otro y recibir una respuesta.

Para servicios centralizados es mejor el paradigma cliente-servidor. El paradigma P2P es más apropiado para **aplicaciones menos centralizadas** como transferencia de mensajes, transferencia de ficheros o videoconferencia. Un ejemplo muy conocido es **Napster**. A pesar de estos hay muchas aplicaciones que utilizan los dos modelos combinados, es mejor esta solución desde el punto de vista de la privacidad.

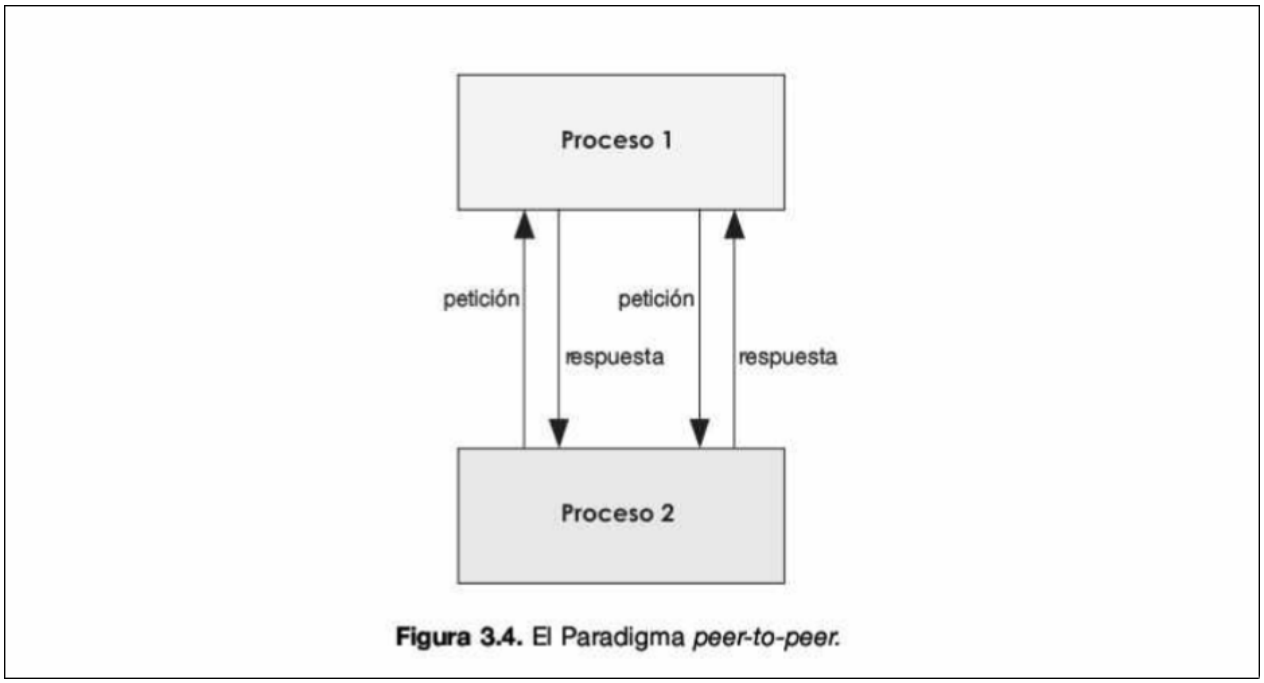


Figura 3.4. El Paradigma *peer-to-peer.*

Figure 4: Pasted image 20250916095357.png

1.2.4 Paradigma de sistema de mensajes

Este paradigma es una sofisticación de paso de mensajes. Intenta resolver el problema de la sincronización del envío de mensajes. Un servidor (sistema de mensajes) sirve de **intermediario entre procesos separados e independientes.** Este sistema tiene como funcionalidad que los procesos intercambien mensajes de forma **asíncrona** de una manera desacoplada.

El emisor deposita un mensaje en el sistema de mensajes y este lo redirige a la cola de mensajes del receptor correspondiente (cada receptor tiene asociada una cola de mensajes). Una vez enviado el mensaje el emisor queda libre para realizar cualquier otra tarea. El receptor puede consultar e cualquier momento su cola de mensajes. El problema es que la cosa es estática por lo que si el receptor se desentiende de los mensajes se colapsaría. Es responsabilidad del receptor leer los mensajes para evitar este caso.

Existen dos subtipos: - **Modelo punto a punto:** el sistema de mensajes redirige el mensaje desde el emisor hasta la cola del receptor. El middelware proporciona un depósito para los mensajes de manera que permite que el envío y la recepción de mensajes estén desacoplados. Comparándolo con el paso de mensajes este paradigma proporciona una abstracción para operaciones asíncronas. Para conseguir esto con el modelo básico sería necesario utilizar hilos o procesos hijo. - **Modo-delo pública/subscribe:** cada mensaje tiene asociado un evento (información que puede ser requerida o no por los otros procesos). Las aplicaciones interesadas en la información de un evento pueden subscribirse a este. Cuando ocurre el evento el proceso publica un mensaje anunciándolo y el middleware se encarga de distribuir los mensajes a todos los subscriptores. Las operaciones utilizadas son publicar, que

permite al proceso difundir a un grupo de otros procesos y suscribir que permite a un proceso escuchar esta difusión. Este modelo aporta una gran abstracción para comunicaciones multicast.

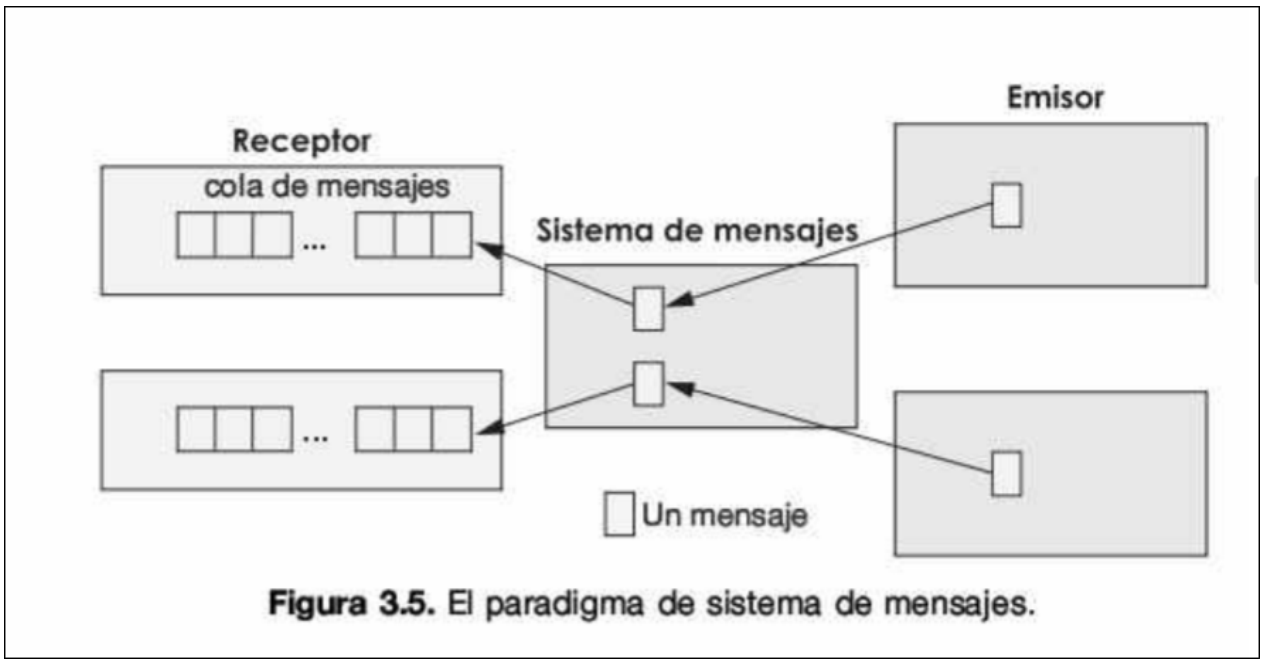


Figure 5: Pasted image 20250920212044.png

1.2.5 Paradigma de llamadas a procedimientos remotos

Según las aplicaciones crecen en complejidad se va necesitando un nivel mayor de abstracción para la programación distribuida. Sería deseable tener un paradigma que permita que el software distribuido se programase de forma similar al convencional. Esto se consigue gracias al paradigma de llamadas a procedimientos remotos que además consigue un **alto nivel de abstracción** tanto para la comunicación de procesos como para la **sincronización** de eventos.

Imaginemos 2 procesos independientes A y B. Si A decide realizar una petición a B, invoca un procedimiento de B pasando unos argumentos junto a la llamada. Una llamada a un procedimiento remoto dispara una acción predefinida en un procedimiento de B. Al finalizar dicho procedimiento el proceso B devuelve un valor a A.

Hay fundamentalmente dos APIs (intefaces, librerias, misma mierda distinto nombre) y las dos incorporan herramientas para transformar invocaciones remotas a llamadas a

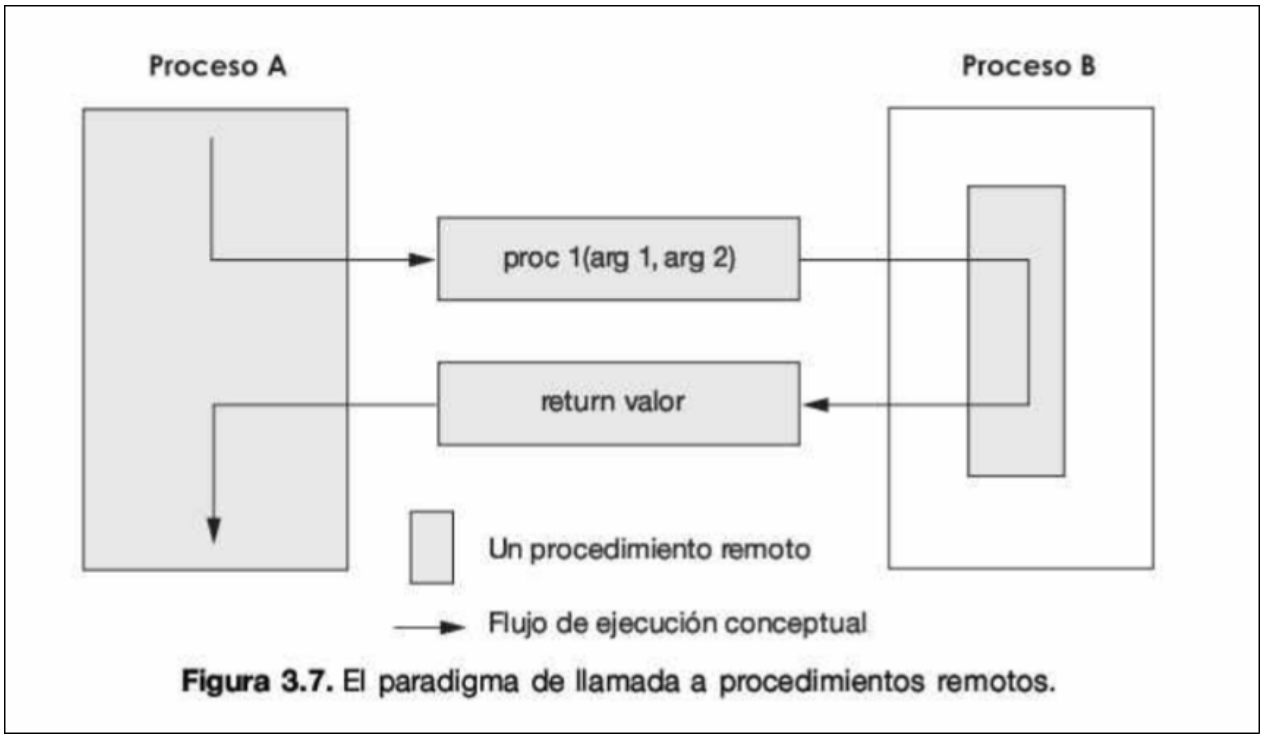
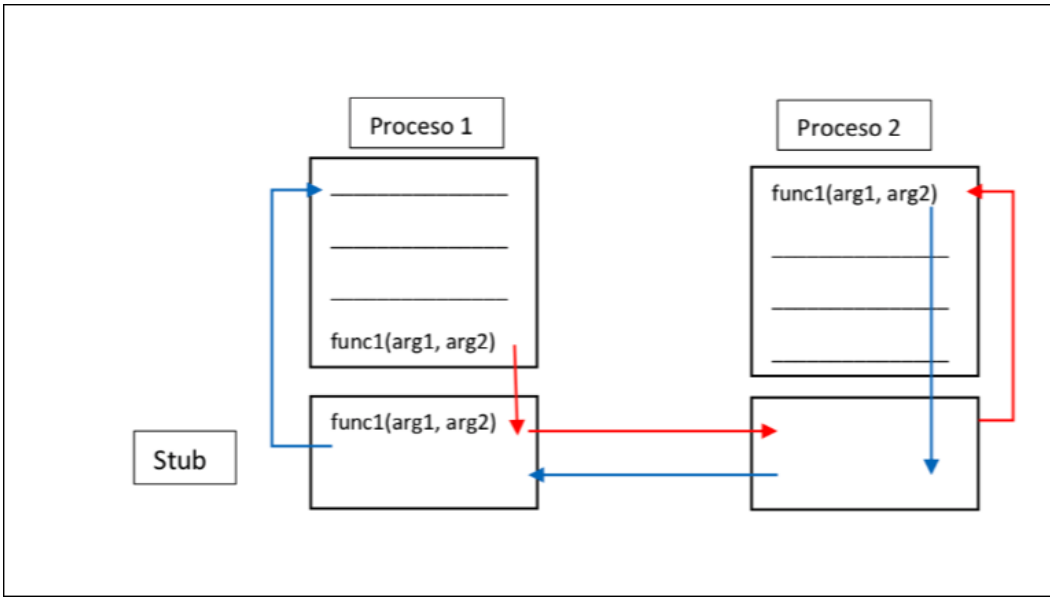


Figura 3.7. El paradigma de llamada a procedimientos remotos.

Figure 6: Pasted image 20250920212328.png



procedimientos locales:

Los módulos se generan automáticamente. Se busca que el programador se abstraiga de los módulos y que para él sea como una llamada local.

1.2.6 Paradigma de objetos distribuidos

La idea es aplicar una orientación a objetos a las aplicaciones distribuidas de manera que estas **acceden a objetos distribuidos sobre una red**. Los objetos propor-

cionarán métodos e invocándolos desde una aplicación se tendrá acceso a los servicios.

Invocación de métodos remotos

Este paradigma es el equivalente orientado a objetos de llamada a procedimientos remotos. Se basa es invocar a un método de un objeto que reside en otro ordenador. Igual que RPC se pueden pasar los argumentos y se puede devolver un valor.

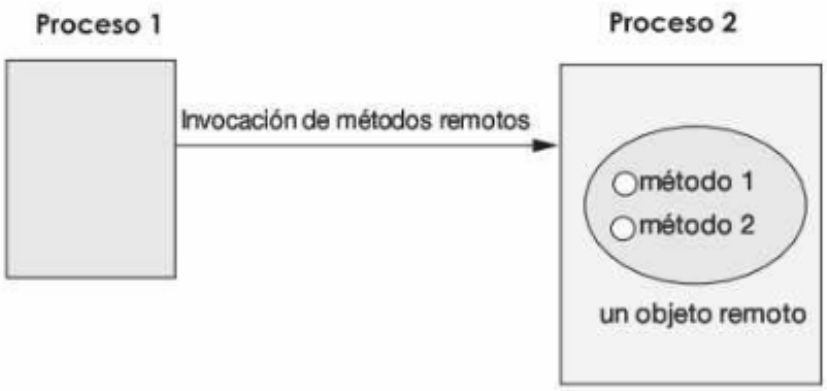


Figura 3.8. El paradigma de invocación de métodos remotos.

Un problema grave es la localización de recursos. Es necesario un mecanismo de localización de recursos dentro de la máquina: **servicio de nombres** (proceso activo todo el tiempo en la máquina del servidor. El cliente puede recurrir a él para conectarse a un recurso).

El diálogo entre el cliente y el servidor de nombres es una aplicación cliente–servidor. El servidor de nombres es obligatorio que exista en cada máquina si hay objetos o recursos remotos.

Paradigma de los servicios de red

Existen proveedores de servicios que se registran en servidores de directorio de una red. Un proceso que desea un servicio contacta con el servidor de directorio/nombres (en tiempo de ejecución) y si está disponible se le dará una referencia al servicio deseado.

El proveedor de servicios debe registrarse (primero de todo) en el servicio de nombres. Este registro caduca así que es necesario renovar la entrada. De esta forma el cliente acude al servidor de directorio y hace uso del servicio.

Este paradigma es una extensión del RPI. La diferencia es que los objetos de servicio se registran en un directorio permitiéndoles ser localizados y accedidos por solicitantes de servicios.

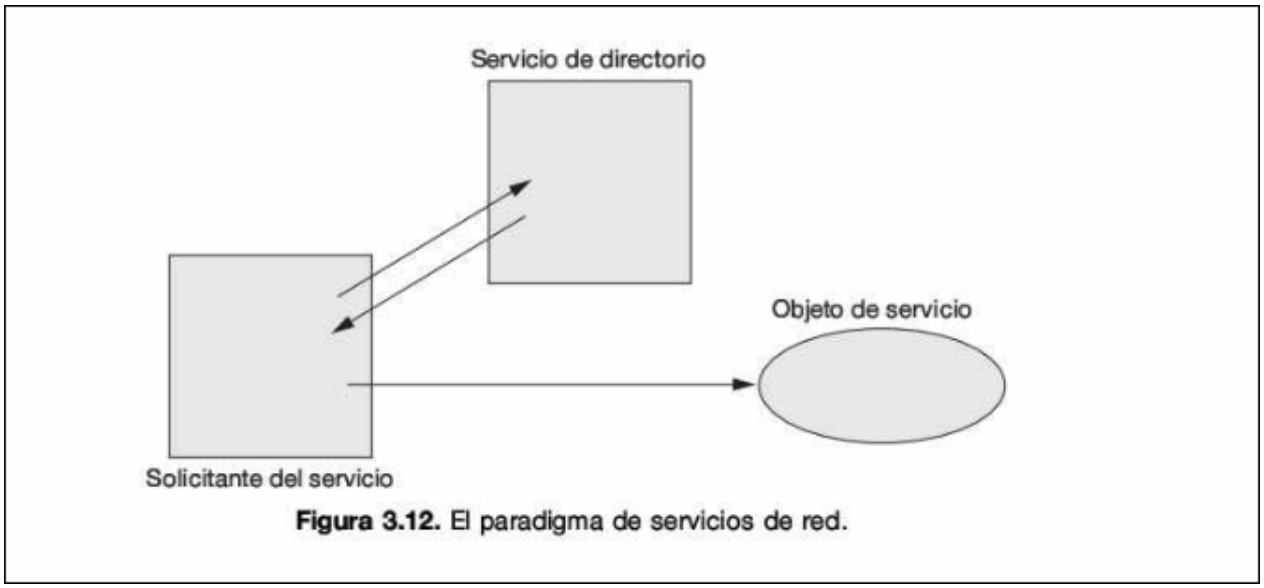


Figura 3.12. El paradigma de servicios de red.

Figure 7: Pasted image 20250921121700.png

Paradigma del Object Request Broker

Un proceso solicita una petición a un **ORB** (Object Request Broker) y este la **redirige al objeto apropiado**. Este paradigma se parece al anterior (RPI) en que los dos proporcionan un soporte para acceder remotamente a objetos. Sin embargo, la diferencia es que este paradigma funciona como middleware (permite a la aplicación acceder a varios objetos remotos o locales). Además este modelo funciona también como mediador para objetos heterogéneos (permite la interacción entre objetos implementados de forma diferentes o ejecutados en diferentes plataformas).

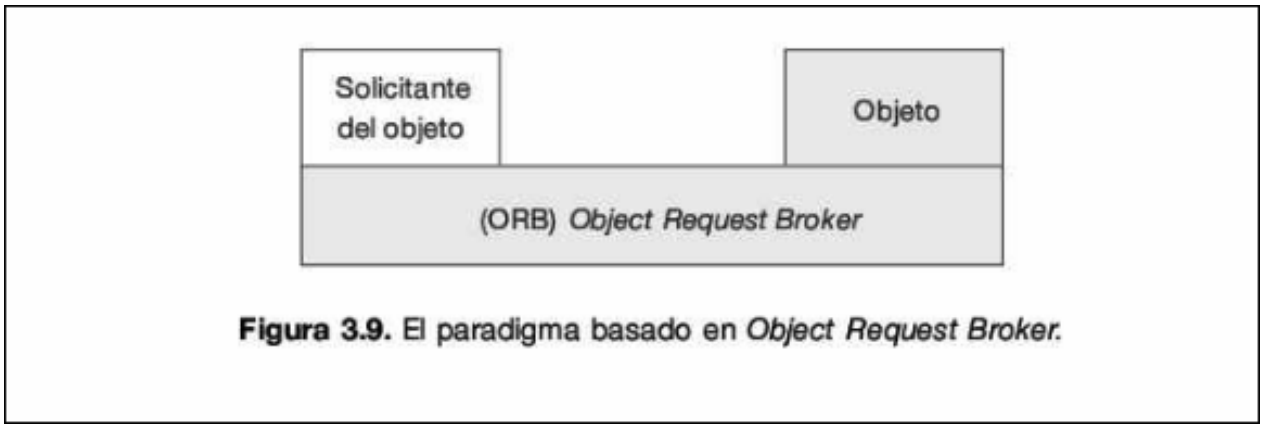


Figura 3.9. El paradigma basado en *Object Request Broker.*

Figure 8: Pasted image 20250921121817.png

Paradigma del Espacio de Objetos

Quizás el más abstracto de los paradigmas orientados a objetos (todos los participantes van a converger en un espacio común abstracto). En este modelo existen **entidades**

lógicas llamadas espacios de objetos.

Un proveedor de servicios coloca los objetos en un espacio de objetos y el proceso quiere un proceso se suscribe al espacio y accede al objeto.

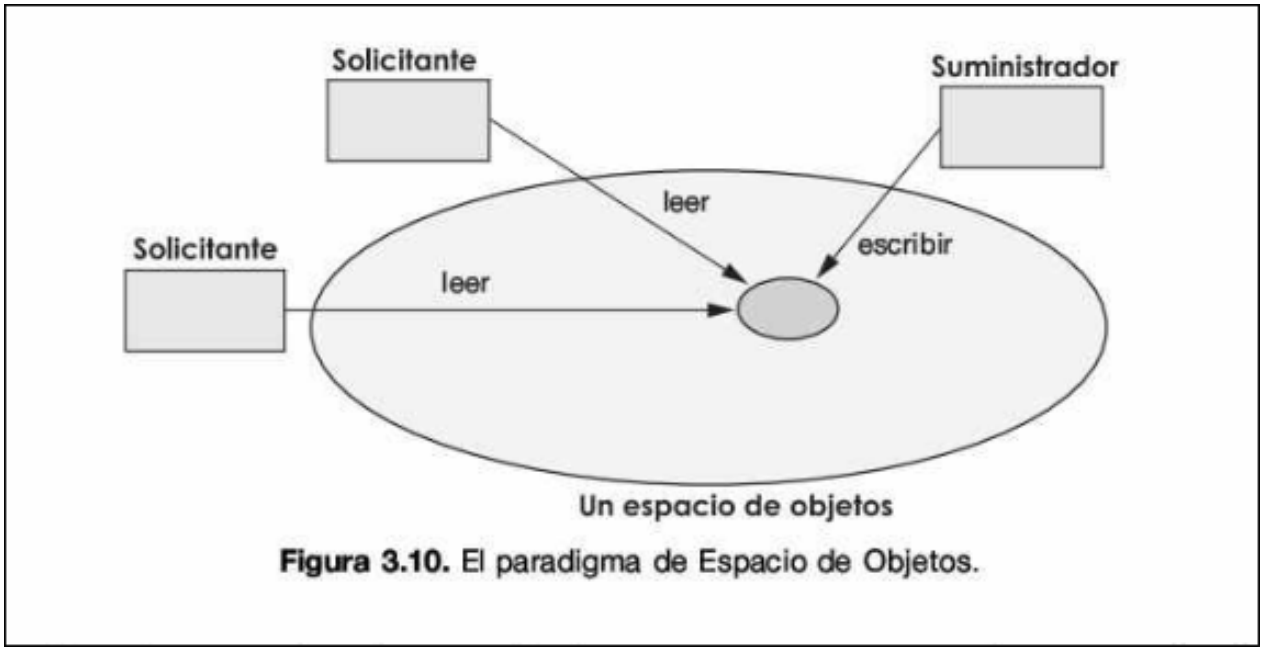


Figura 3.10. El paradigma de Espacio de Objetos.

Figure 9: Pasted image 20250921122117.png

El tener el espacio de objetos cumple que la abstracción adquirida oculta los detalles implicados en la búsqueda de recursos u objetos que son necesarios en otros paradigmas como RPI u ORB. Una herramienta basada en este paradigma es JavaSpaces

1.2.7 Paradigma de agentes móviles

Un **agente** es un programa u objeto transportable.

En este modelo, el agente se lanza desde un ordenador viajando de forma autómata a otro de acuerdo con el itinerario que posea. En cada parada accederá a los recursos y servicios necesarios para realizar su tarea. Es necesario proporcionarle unos argumentos y él resolverá el problema.

Un agente ha de ser móvil, autónomo e inteligente. Además tiene un lenguaje de programa muy complejo. El nivel de abstracción es muy alto ya que en lugar de intercambio de mensajes los **datos son transportados**.

1.2.8 Paradigma de aplicaciones colaborativas

Los procesos participan en una sesión colaborativa como grupo. Cada participante puede hacer contribuciones a todos o parte del grupo. Esto se logra utilizando multidifusión (multicasting) para enviar los datos o usando pizarras virtuales (permiten a cada participante leer y escribir datos sobre una visualización compartida).

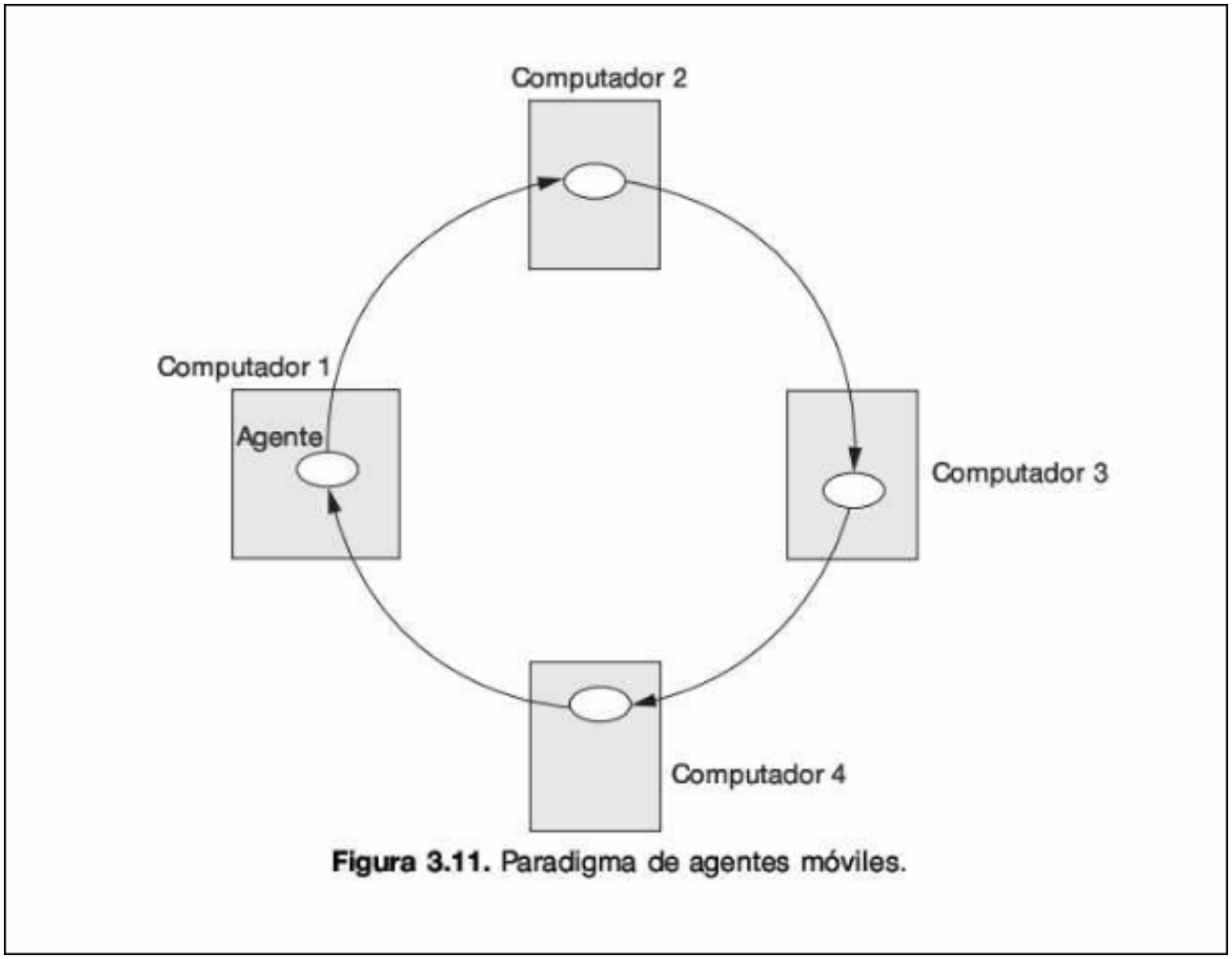


Figura 3.11. Paradigma de agentes móviles.

Figure 10: Pasted image 20250921122236.png

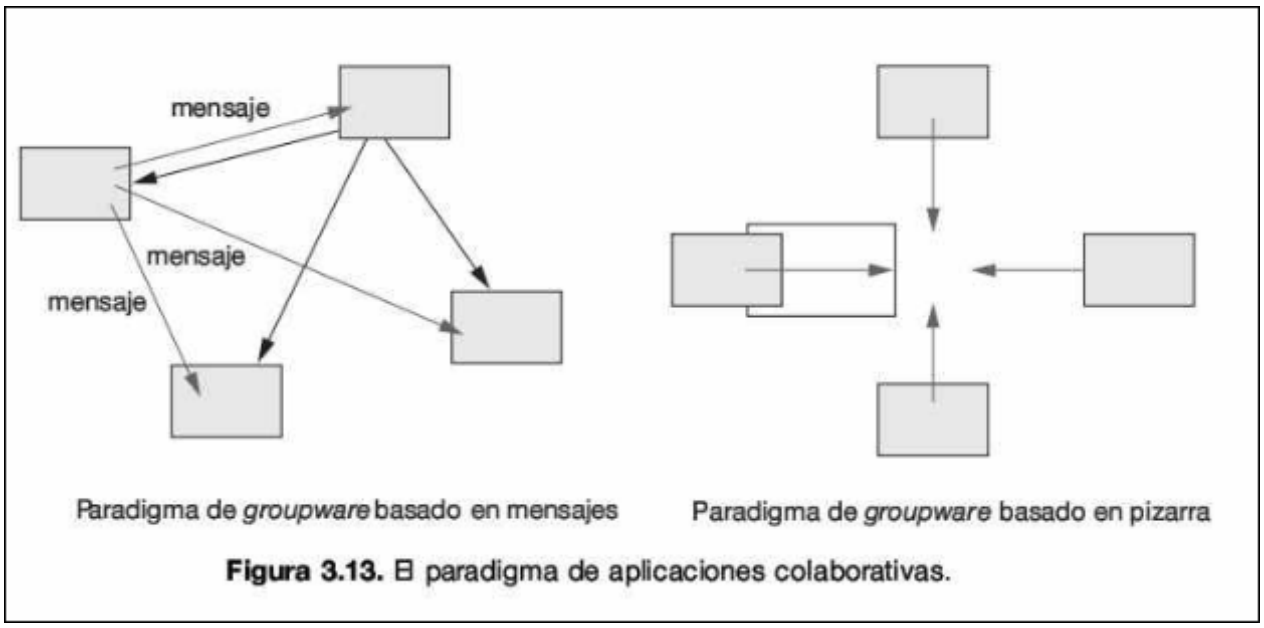


Figura 3.13. El paradigma de aplicaciones colaborativas.

Figure 11: Pasted image 20250921122510.png