Sistemas Informáticos II

Tema 1: *Middleware*

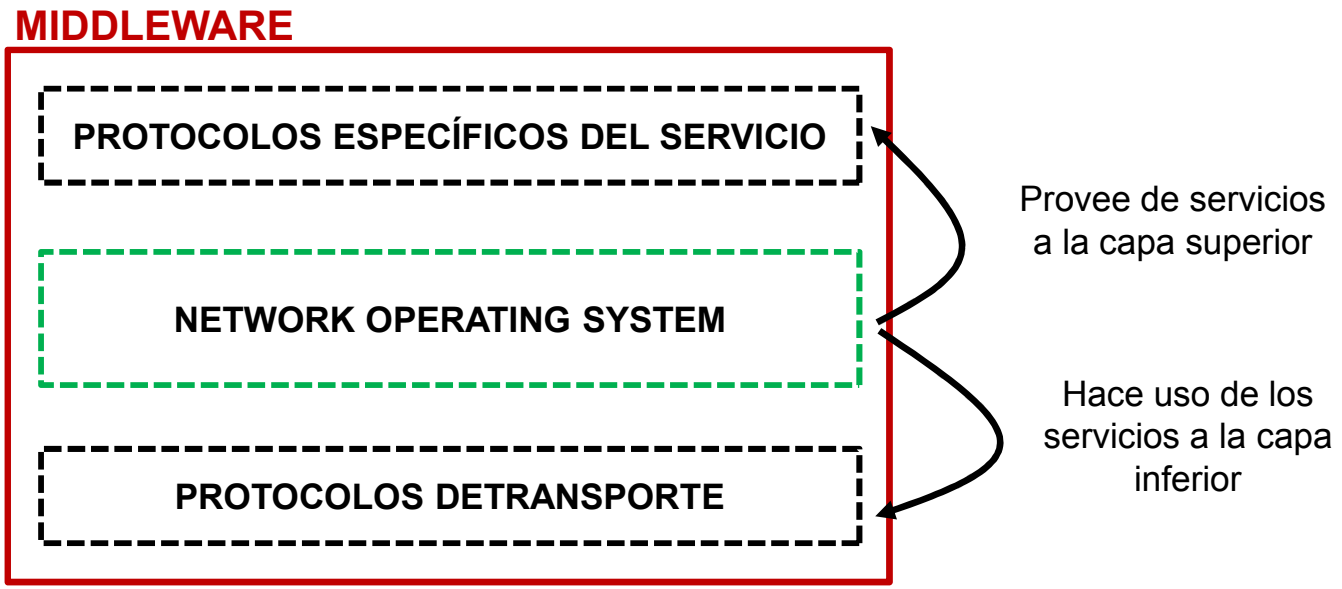
Antonio E. Martínez, Irene Rodríguez
Daniel Hernández (daniel.hernandez@uam.es)
Álvaro Ortigosa (alvaro.ortigosa@uam.es)
Manuel Sánchez-Montañés (manuel.smontanes@uam.es)

Contenido

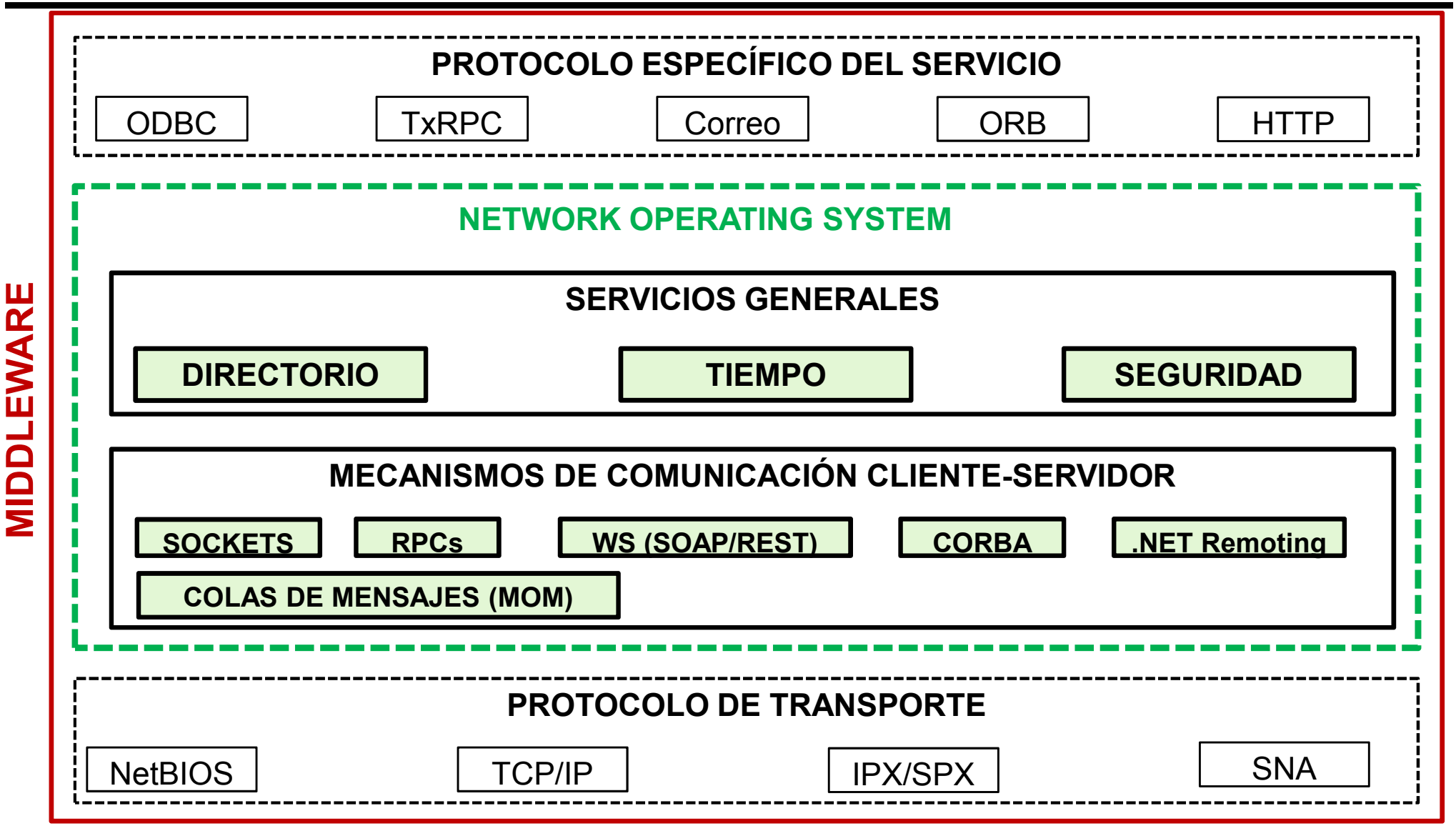
- Introducción.
- Servicios de transporte
  - APIs directas.
  - *Remote Procedure Calls*.
  - Web Services.
  - Comunicación mediante colas de mensajes.
  - Objetos distribuidos.
- Servicios de aplicación:
  - Directorio.
  - Tiempo.
  - Seguridad.
- Bibliografía especial del tema.

Introducción

- ***Middleware***: Conjunto de aplicaciones encargadas de enlazar los componentes de un sistema distribuido.
- Se puede considerar dividido en tres capas:
  - Protocolos específicos del servicio especiales para distintos tipos de sistemas Cliente / Servidor.
  - ***Network Operating System, NOS***. Este tema se centra en esta capa.
  - Protocolos de transporte comunes a otras aplicaciones.

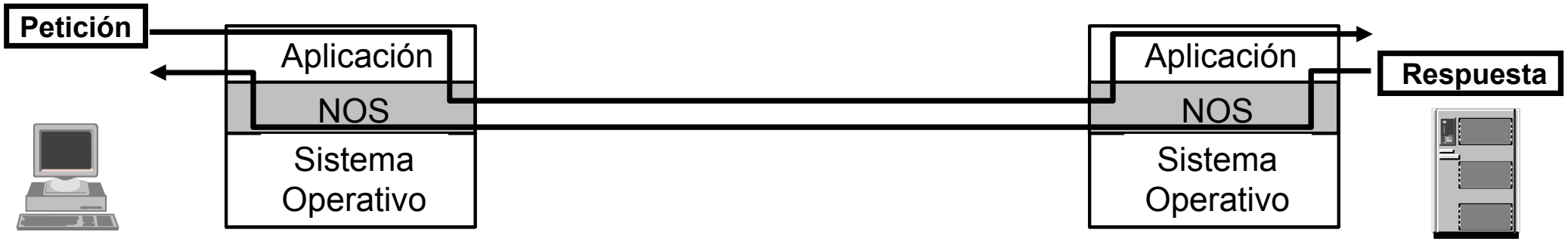


Esquema general Middleware



Network Operating System, NOS

- Encargado de proporcionar una apariencia de sistema único a un sistema distribuido.
- Extensión del Sistema Operativo:
- El cliente realiza una llamada a un servicio como si fuera local (**transparencia**).
- El NOS
  - Intercepta la llamada.
  - Redirige la llamada al servidor apropiado.
  - Devuelve contestación.



Transparencia (I)

- El NOS debe proporcionar transparencia a los procesos que lo utilizan.
- Formas de transparencia definidas por RM-ODP (*Open Distributed Processing Reference Model* - ISO 10746):
  - **De acceso**: oculta las diferencias en la representación de los datos y llamadas a procedimientos.
  - **De ubicación**: oculta dónde reside cada recurso.
  - **De movilidad**: oculta que un recurso puede moverse a otra ubicación.
  - **De reubicación**: oculta que un recurso puede moverse a otra ubicación mientras está siendo utilizado.
  - **De persistencia**: oculta la activación y desactivación de objetos desde un soporte de datos permanente.
  - **De replicación**: oculta el uso de múltiples ejemplares de cada recurso para aumentar fiabilidad y prestaciones.
  - **Frente a fallos**: oculta el fallo y recuperación de un recurso.
  - **De concurrencia o de transacción**: oculta el uso de un recurso de modo concurrente por varios procesos.

Transparencia (II)

- Otras características deseables:
  - **De prestaciones**: reconfigurar el sistema para mejorar sus prestaciones según varía la carga de trabajo.
  - **De escalado**: expansión del sistema en tamaño sin cambiar su estructura o los algoritmos de la aplicación.
  - **Espacio de nombres**: Las convenciones de los nombres de los recursos deben ser iguales, independientemente del sistema que los soporte.
  - **Conexión**: Un único usuario y contraseña para todo el sistema. Sólo se debe introducir una vez *(Single Sign On, SSO)*.
  - **Tiempo**: el mismo en todo el sistema. Los relojes de todos los elementos del sistema cliente / servidor deben estar sincronizados.
  - **Administración**: Un único sistema de gestión de todos los recursos.
  - **Protocolos**: Idéntica interfaz de programación para todos los protocolos de transporte.

Transparencia de datos

- La representación interna de datos es dependiente del ordenador, su hardware o su SO.
- Para evitar el problema del formato de representación de datos, cliente y servidor deben ponerse previamente de acuerdo en el formato en que se van a intercambiar los datos.
- Tres alternativas en el tipo de acuerdo a tomar:

1. Mecanismo de representación de los datos independiente de la plataforma.

– Antes de la transmisión, los datos se convierten a un formato genérico conocido por todos los procesos (representación externa de datos).

– ***Marshalling***: el emisor convierte los datos del formato local al formato común.

– ***Unmarshalling***: el receptor convierte el formato común al formato local de su arquitectura.

2. Para la **comunicación entre dos ordenadores con arquitectura común**, el paso anterior puede omitirse.

– Esto requiere que antes de la transmisión de los parámetros (al establecer la sesión de comunicación), los dos extremos negocien si se requiere pasar los datos a un formato genérico o no.

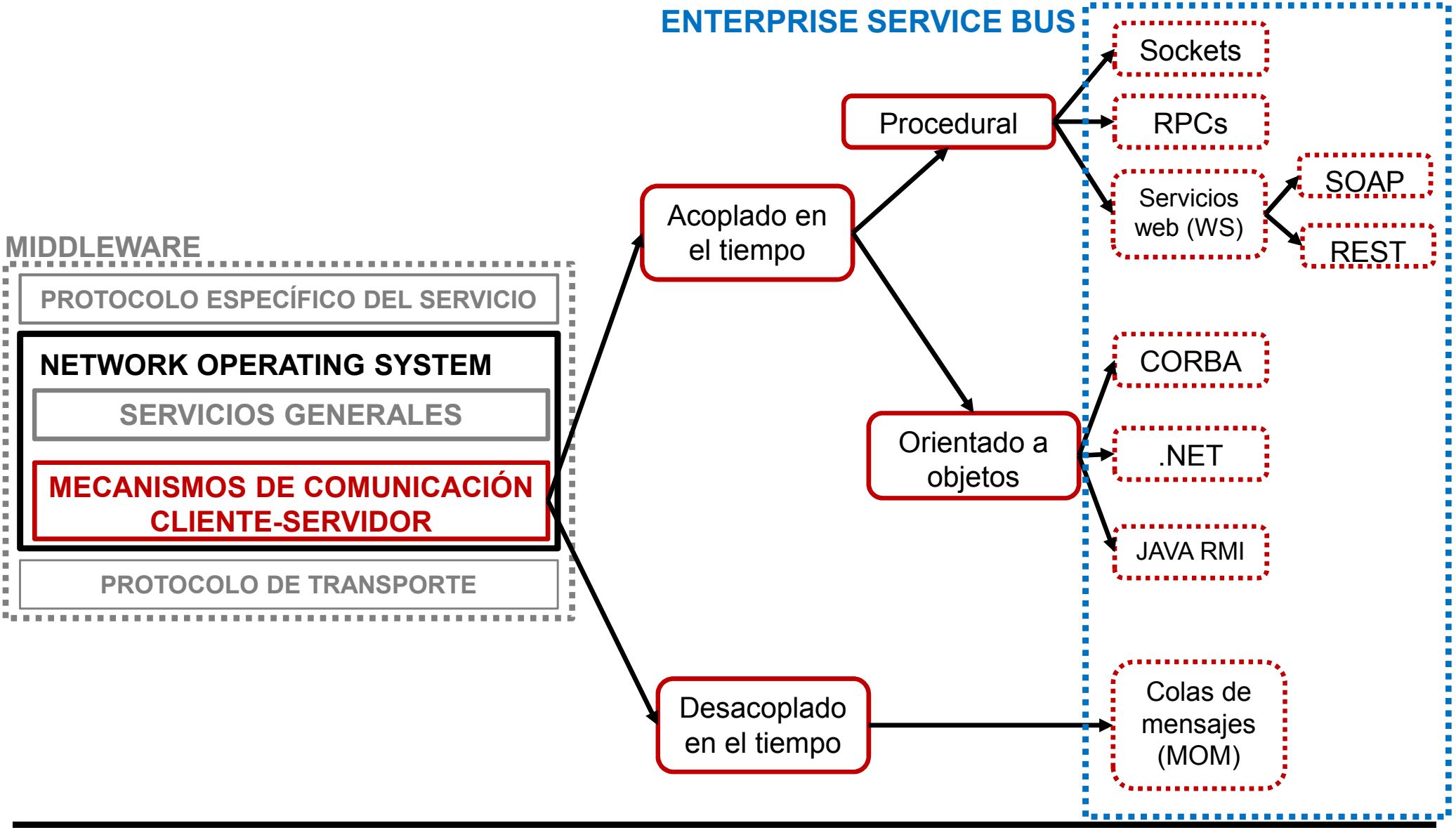
3. Transmisión de los datos en su formato nativo (el de la arquitectura del transmisor) junto con un identificador del tipo de arquitectura subyacente.

– El receptor decide si es necesario convertir los datos recibidos o no.

Transparencia de datos

- Un mecanismo de representación de los datos independiente de la plataforma (caso 1) es el que proporciona una mayor transparencia de datos.
- Distintos mecanismos empleados:
  - **Sistemas propios de aplicación o protocolo**. Ejemplos: *External Data Representation, XDR,* de SUN-RPC; *Network Data Representation (NDR)* de Apollo RPC y DCE-RPCs.
  - **Estándar *Abstract Syntax Notation 1, ASN.1*** (ITU-T X.680-ISO/IEC 8824-1:2002)
    - Orientado a optimizar la compactación de la información y su codificación y decodificación por los nodos de la red.
    - Contiene gramática de declaración de la información y reglas de codificación de datos en la red (ej: *Basic Encoding Rules,* (ITU-T X.690-ISO/IEC 8825-1:2002))
  - **Codificación XML.** Definición de la gramática y codificación en red.
    - Orientado a la interpretación humana de la información.
    - Codificación y decodificación costosa por los nodos de proceso
  - **Codificación JSON**. Formato legible y ligero para el intercambio de datos.
  - **Codificación de los caracteres.** Diversos estándares.
    - ISO-8859-1 *(Latin1)*, ISO/IEC 10646 *Universal Character Set*, Unicode *(UTF-8)*.

Mecanismos de comunicación cliente-servidor: esquema general



Mecanismos de comunicación cliente-servidor

- Modelos de interacción
  - **Síncrono**: El cliente envía una consulta y espera hasta que se le devuelven los resultados.
  - **Asíncrono**: El cliente continúa su proceso tras realizar una consulta. Los resultados se envían cuando están disponibles.
- Protocolos de intercambio:

Nombre	*Mensajes enviados por*		
	Cliente	*Servidor*	*Cliente*
R	*Petición*		
RR	*Petición*	*Respuesta*	
RRA	*Petición*	*Respuesta*	*Confirmación de la respuesta*

APIs directas

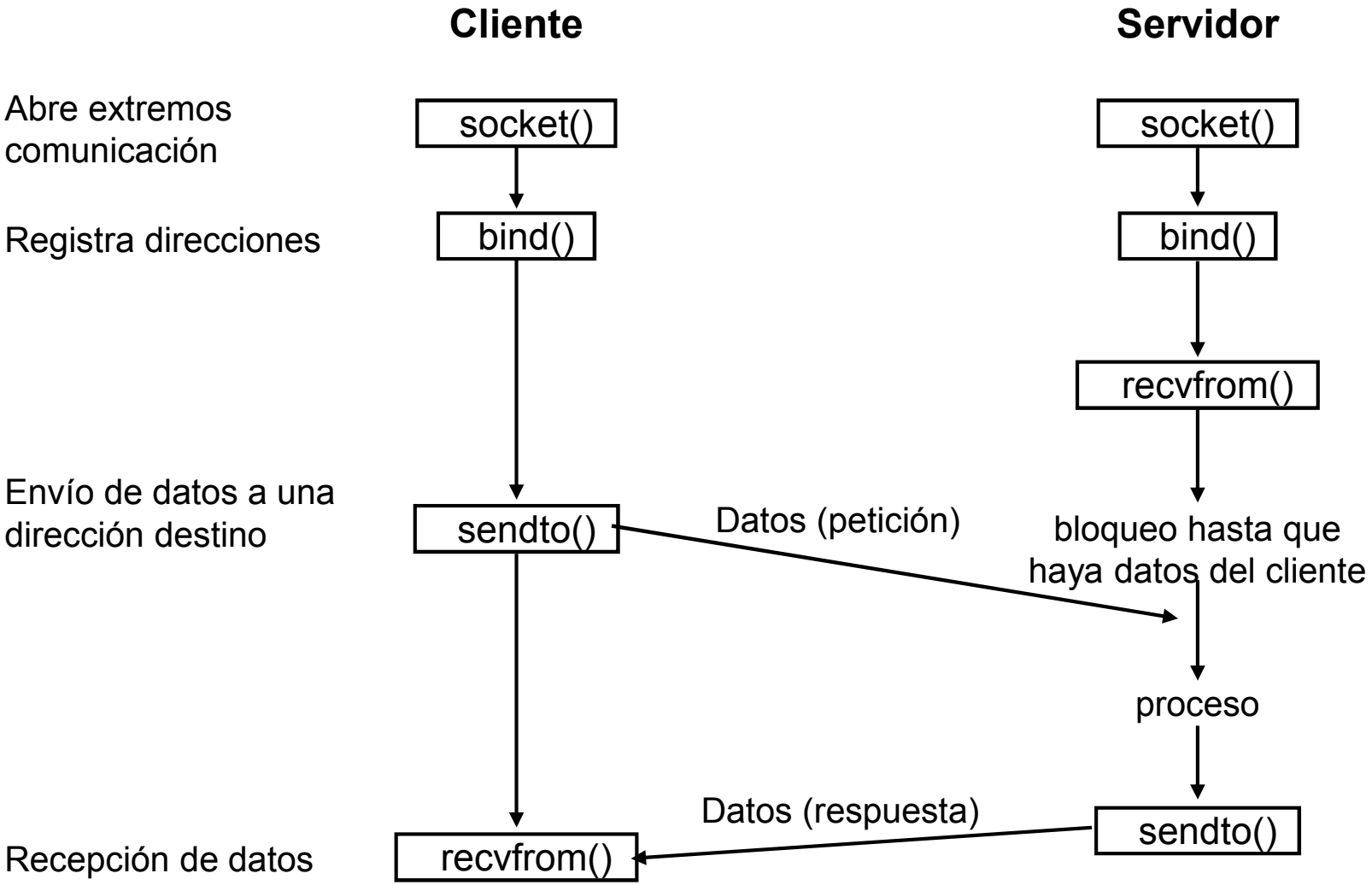
- Uso directo de una interfaz de programación para acceder a los servicios de comunicaciones.
  - Habitualmente servicios de nivel transporte o sesión.
  - Comunicaciones orientadas a conexión / datagramas.
  - Ubicación extremos no transparente para el programa ("*close to the wire*").
    - Por ejemplo, problemas de timeouts o errores de conexión deben ser resueltos por el propio programador.
  - APIs inicialmente específicos del protocolo sobre el que se utilizan.
    - Ejemplo: Sockets para protocolo TCP/IP.

- Muchas de las primeras aplicaciones cliente-servidor se implementaron usando protocolos de comunicación peer-to-peer de bajo nivel tales como sockets, TLI, CPIC/APPC, NetBIOS, y Named Pipes.
- Estos protocolos de bajo nivel son difíciles de implementar y mantener. En su lugar, los programadores usan RPCs, Web Services o RMI, entre otros, que proporcionan mayor nivel de abstracción.

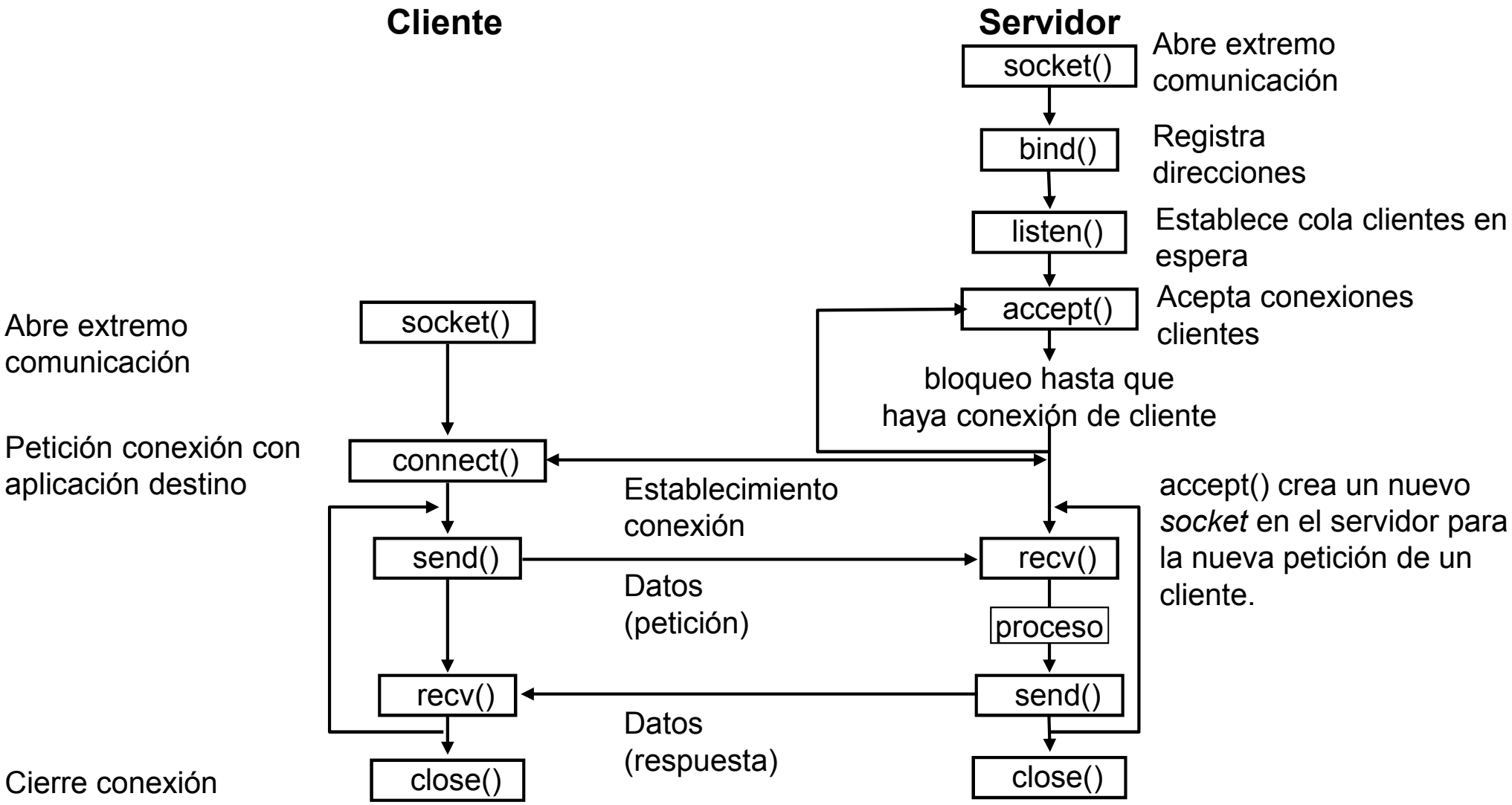
Sockets

- Incluidos como medio de programación de comunicaciones en el Unix de Berkeley (BSD) versión 4.2 en 1981.
- Adaptados a AT&T Unix como *Transport Level Interface, TLI*.
  - Unix SVR4 incorpora ambos.
- Transportados como método de programación a otras redes no IP: *NetBios*, IPX, SNA.
- Establece un camino de comunicación entre
  - Dirección IP origen, Puerto IP origen.
  - Dirección IP destino, Puerto IP destino.
  - Protocolo (TCP, UDP…).

Comunicación no orientada a conexión



Comunicación orientada a conexión

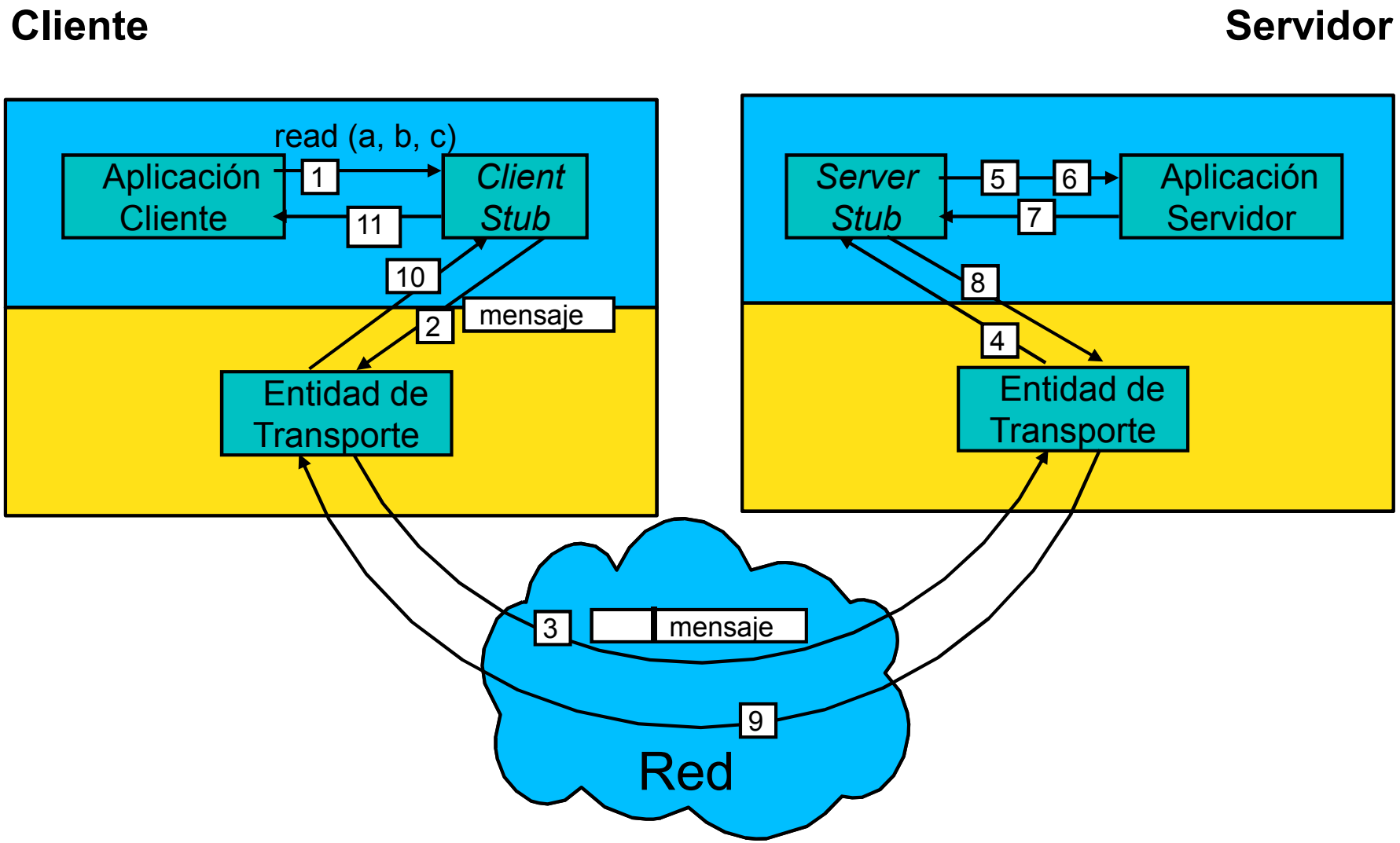


Example: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html

Remote Procedure Calls, RPC

- La ejecución del servicio en el cliente se realiza del mismo modo que una llamada a una función local:
  - El cliente llama a una función.
    - Conoce el prototipo de la función, y tiene una interfaz enlazada en su código.
  - Pasa los parámetros.
    - Conoce la estructura de parámetros y los tipos de los mismos.
  - Su proceso se detiene hasta que la ejecución de la función finaliza (funcionamiento síncrono).
  - Recibe la respuesta de la función a través de parámetros y resultado de la función.
- Múltiples tipos:
  - Sun, Apollo, XML-RPC...
  - Muchas de las interfaces de programación de los middlewares actuales están basadas en algún tipo de RPC (CORBA, RMI, Web Services...)
- Ejemplo uso RPCs: Hadoop (https://wiki.apache.org/hadoop/HadoopRpc)

Esquema del proceso de una RPC (I)



Esquema del proceso de una *RPC* (II)

1. El cliente llama a una función con una serie de parámetros *(Client Stub).* En ese momento queda bloqueado a la espera de una respuesta.
2. El *Client Stub* compone un mensaje para el servidor *(parameter marshalling*).
3. El mensaje se envía a través de un protocolo de transporte.
4. La entidad de transporte del servidor lo recibe y lo pasa al *Server Stub*.
5. El *Server Stub* recibe el mensaje y extrae de él la información que necesita *(parameter unmarshalling).*
6. El *Server Stub* pasa la petición a la rutina de servicio.

La contestación del servidor sigue un proceso análogo.

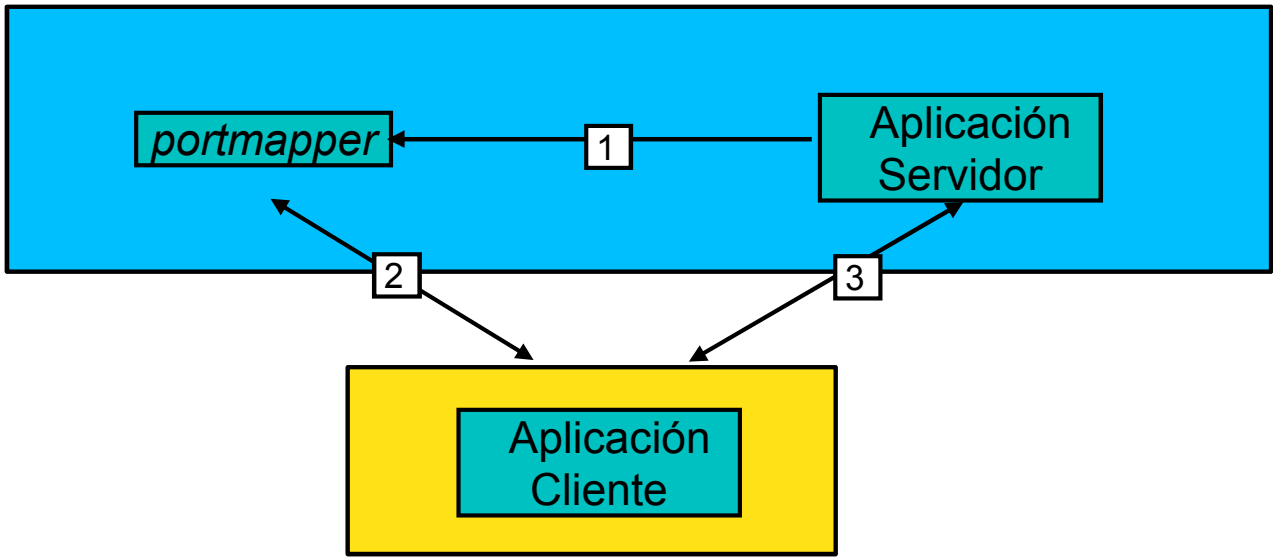
7. Respuesta del servidor
8. Generación del mensaje de respuesta *(parameter marshalling).*
9. Envío a través del protocolo de transporte.
10. Descomposición del mensaje en el *Client Stub (parameter unmarshalling).*
11. Entrega de la respuesta de la función al cliente. El cliente sale de su bloqueo y continua su trabajo.

Problemas de transparencia

- Paso de parámetros
  - No es posible pasar parámetros por referencia.
- Asociación llamada / servidor / proceso en el servidor *(Binding):*
  - Estático, dinámico, automático (servidores de nombres).
- Semántica de la llamada.
  - Tras un *Time Out* no se sabe si la llamada se ha ejecutado o no.
  - Distintos tipos de operaciones:
    - Idempotentes: Se pueden ejecutar cualquier número de veces.
    - No Idempotentes: El resultado varía con el número de veces que se ejecuten.
  - Según las operaciones puede haber distinta estrategia en las llamadas RPC:
    - Ejecución Exactamente una vez.
    - Ejecución Como máximo una vez.
    - Ejecución Al menos una vez.
- Representación de los distintos tipos de datos.
  - Necesario método de representación de datos independiente del sistema.
- Rendimiento de las llamadas.
- Seguridad.

RPC en redes IP

- Utiliza como protocolo de transporte TCP o UDP.
- Diversas versiones. La más común es la de SUN (RFC 1057).
- Problema: conocer el puerto en el que escucha el servidor.
  1. Programa servidor se registra a *Port Mapper*.
  2. Programa cliente pregunta a *Port Mapper* el número de puerto utilizado.
  3. Se establece la conexión por dicho puerto.



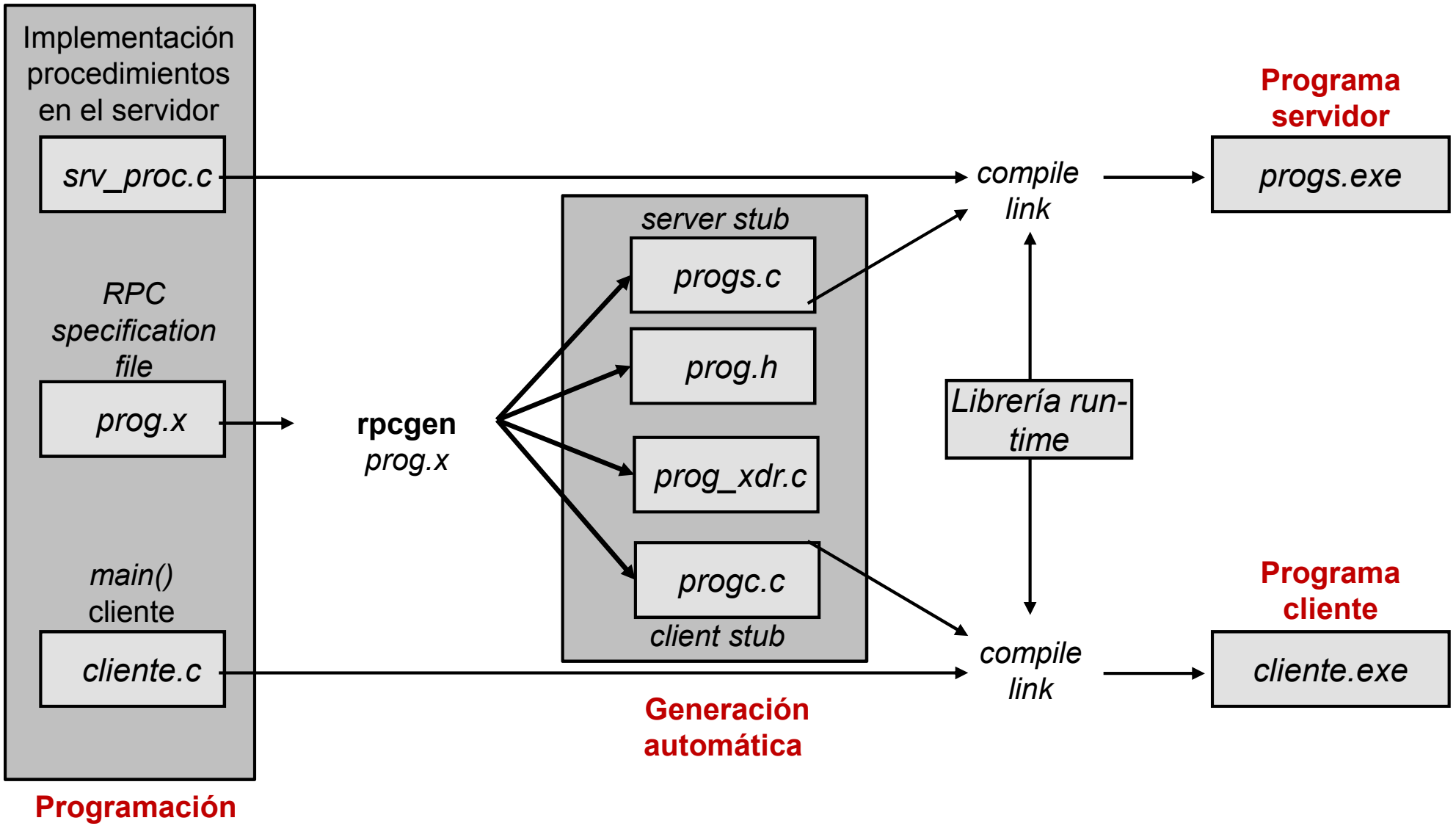
Sun RPC

- Implementado en Unix y Windows, Solaris, entre otros.
- Ahora se llama (ONC RPC) (Open Network Computing Remote Procedure Call)
- Tiene tres componentes:

1. **Lenguaje de definición de tipos de datos *(eXternal Data Representation, XDR)***:
   - Permite definir los datos a intercambiar entre cliente y servidor RPC de forma transparente al ordenador y lenguaje en que se trabaje.
   - Lenguaje de definición de datos, no de programación.
   - Definido en la RFC 1014. Incluye declaraciones y formatos de cada tipo.
   - Sintaxis similar al C.
2. **Lenguaje de especificación de RPC**.
   - Contiene declaraciones XDR y especificación de programas, versiones y procedimientos.
   - Compilador de este lenguaje, *rpcgen.*
   - Genera *client stub, server stub* y fichero de interfaz *(include),),* rutinas de marshalling de los datos *(_xdr).*
3. **Librería de implementación**.

Ejemplo en Moodle

Programación RPC



Paso de parámetros en RPCs

- **RPCs en general:**
  - Soportan múltiples parámetros.
  - Cada uno puede ser de entrada y de salida.
  - Realización *Parameter Marshalling / Demarshalling*.

- **RPCs SUN:**
  - Un único parámetro de entrada (parámetro del RPC) y un único parámetro de salida (resultado de la función).
    - Ejemplo: *string fecha (string).*
  - Los parámetros pueden ser estructuras.
  - No se cambia el valor del parámetro de entrada.
  - Proceso de *Marshalling / Demarshalling* mínimo en RPC. Responsabilidad usuario.

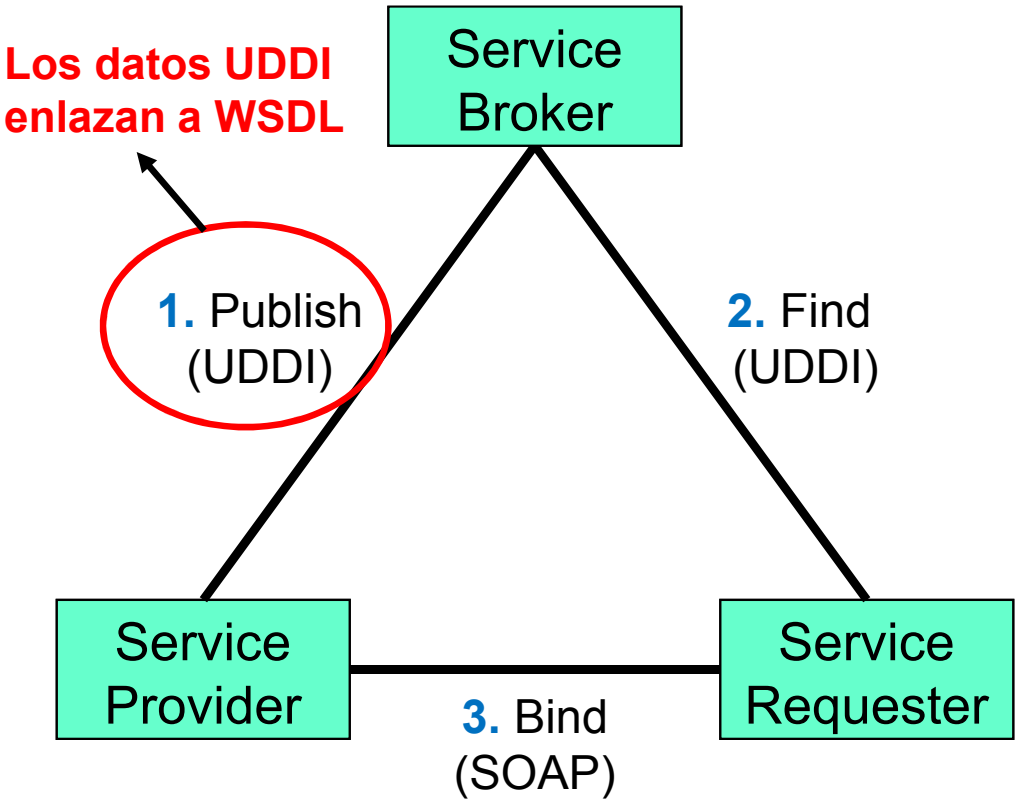
- **Apollo RPC (DEC)**
  - Base de DCE (Distributed Computing Environment) RPC.
  - Múltiples parámetros. Especificación individual de los que son Entrada y Salida.
  - Lenguaje de representación de datos: *Network Data Representation (NDR).*
  - Lenguaje representación interfaces: *Network Interface Definition Language (NIDL).*

Servicios Web *(Web Services, WS)*: WS basados en SOAP

- Concebido como un nuevo modelo de uso de la Web *(Application Centric Web)*
  - Transacciones iniciadas automáticamente por un programa, no necesariamente utilizando un navegador.
  - Se pueden describir, publicar, descubrir e invocar dinámicamente en un entorno distribuido *(just-in-time application integration)*:
    - Servidores publican las funciones que realizan.
    - Acceso universal a dichas funciones.
- *Middleware* sobre Internet:
  - Funcionalidad para publicar y descubrir servicios dinámicamente.
  - Ejecución de servicios siguiendo un modelo de RPCs.
  - Servicios multi-*tier*. Un servidor puede a su vez solicitar servicios.
- Creado por el W3 *Consortium*. Estandarizado por diversos organismos internacionales.
  - Uno de los más activos entre ellos es *OASIS: Organization for the Advancement of Structured Information Standards*. http://www.oasis-open.org
- Ejemplo: Product Advertising SOAP API de Amazon Webservices

Componentes *Web Services* SOAP

- Proporcionador de servicios *(Service Provider)*:
  - Es quien ejecuta los servicios solicitados.
  - **Publica** la disponibilidad de los servicios a través del registrador.
- Registrador de servicios *(Service Broker)*:
  - Soporte para la publicación y localización de servicios.
- Cliente *(Service Requester)*:
  - **Busca** los servicios a través del *Service Broker*.
  - **Enlaza** con los servicios del *Service Provider*.



Ejemplos detallados de ficheros WSDL, UDDI, SOAP: CERAMI, E., *Web Services*, O'Reilly, 2002. (disponlble on-line en la biblioteca)

Service Oriented Architecture Protocol, SOAP

- Protocolo **basado en XML** para el intercambio de información entre ordenadores.
- **Independiente de plataforma y lenguaje de programación.**
- Aunque se puede usar en distintos sistemas de mensajes y sobre distintos protocolos de transporte, su uso principal es transportar RPCs sobre HTTP.
- Estructura de un mensaje SOAP:
  - **Sobre** *(SOAP-ENV:Envelope)*: Contiene el resto de los elementos.
  - **Cabecera** (opcional) *(SOAP-ENV:Header)*: Información general a nivel de aplicación.
  - **Cuerpo** *(SOAP-ENV:Body)*: Contiene la petición o la respuesta.
    - Puede incluir elementos genéricos de información de errores *(SOAP-ENV:Fault)*.

Web Services Description Language, WSDL

- **Lenguaje XML** para especificar la **interfaz pública** de un servicio Web.
- Estructura de una declaración WSDL
  - ***<definitions>*:** Elemento raíz que contiene el resto. Define su nombre y los espacios de nombres que utiliza.
  - ***<types>***: Tipos de datos utilizados entre cliente y servidor. Usa W3C XML Schema (XSD) por defecto.
  - ***<message>***: Declaraciones de mensajes empleados para peticiones y respuestas y los elementos que los forman.
  - ***<portType>***: Operaciones soportadas y encadenamiento de mensajes que implica su ejecución.
  - ***<binding>***: Modo en que los mensajes se transmiten sobre un protocolo de RPC, con extensiones específicas para SOAP.
  - ***<service>***: Contiene la información de la dirección en la que se localiza el servicio.

http://uddiconsole-jbossoverlord.rhcloud.com/home.jsp

Universal Description, Discovery and Integration, UDDI

- Es el mecanismo de **registro** y **búsqueda** de servicios Web.
- Consta de **tres partes**:
  - Modelo de datos UDDI: Esquema XML que permite describir servicios. Hay cuatro tipos de información.
  - API UDDI: **RPCs SOAP** para registro y búsqueda de datos en UDDI.
  - UDDI *Cloud Services*: Conjunto de servidores sincronizados que proporcionan directorios UDDI.
- Los **datos** capturados por UDDI se dividen en **tres categorías**:
  - Páginas blancas: Información general de una compañía.
  - Páginas amarillas: Se clasifican en categorías.
  - Páginas verdes: Describen los servicios que se prestan.
- UDDI no está limitado a servicios SOAP.

Ejemplo de fichero WSDL

```
<?xml version="1.0"?>
<definitions name="HelloService"  targetNamespace="http://www.examples.com/wsdl/HelloService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap
  xmlns:tns="http://www.examples.com/wsdl/HelloService.wsdl" xmlns:xsd="http://www.w3.org/20

<message name="SayHelloRequest"><part name="firstName" type="xsd:string"/></message>
<message name="SayHelloResponse"><part name="greeting" type="xsd:string"/></message>

<portType name="Hello_PortType"><operation name="sayHello">
      <input message="tns:SayHelloRequest"/><output message="tns:SayHelloResponse"/>
</operation></portType>

<binding name="Hello_Binding" type="tns:Hello_PortType">
<soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
<operation name="sayHello"><soap:operation soapAction="sayHello"/>
        <input><soap:body encoding="literal"/></input>
       <output><soap:body encoding="literal"/></output>
</operation> </binding>

<service name="Hello_Service"><port binding="tns:Hello_Binding" name="Hello_Port">
      <soap:address location="http://www.examples.com/HelloService/" />
    </port> </service>
</definitions>
```

Espacios de nombres

Datos definidos en XSD (XMLSchema)

Definición de mensajes

Encadenamiento de los mensajes

La invocación será mediante SOAP

Dirección del servicio

Ejemplo de Mensajes SOAP

- **Petición SOAP enviada al servidor**:

```
POST /HellowService HTTP/1.1
Content-Type: text/xml; charset=utf-8
SOAPAction: “sayHello"
Content-Length: 421
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
        SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding" >
<SOAP-ENV:Body xmlns:m="http://www.examples.com/wsdl/HelloService.wsdl" >
    <m:sayHelloRequest><m:firstName>Juan Lopez Perez</m:firstName></m:sayHelloRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Campos indicados en el fichero WSDL

- **Respuesta SOAP recibida en el cliente**:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2001/12/soap-envelope"
        SOAP-ENV:encodingStyle="http://www.w3.org/2001/12/soap-encoding" >
<SOAP-ENV:Body xmlns:m="http://www.examples.com/wsdl/HelloService.wsdl" >
    <m:sayHelloResponse><m:greeting>¡Bienvenido Juan Lopez Perez!</m:greeting></m:sayHelloResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Ventajas e inconvenientes *Web Services basados en SOAP*

- **Ventajas**:
  - Estándares abiertos. Acuerdo global entre fabricantes.
  - Independientes de plataforma.
  - Gran independencia entre el cliente y el servidor.
  - Utilizan generalmente HTTP para la comunicación lo que permite pasar firewalls.
    - También cierto para *Web Services* basados en REST

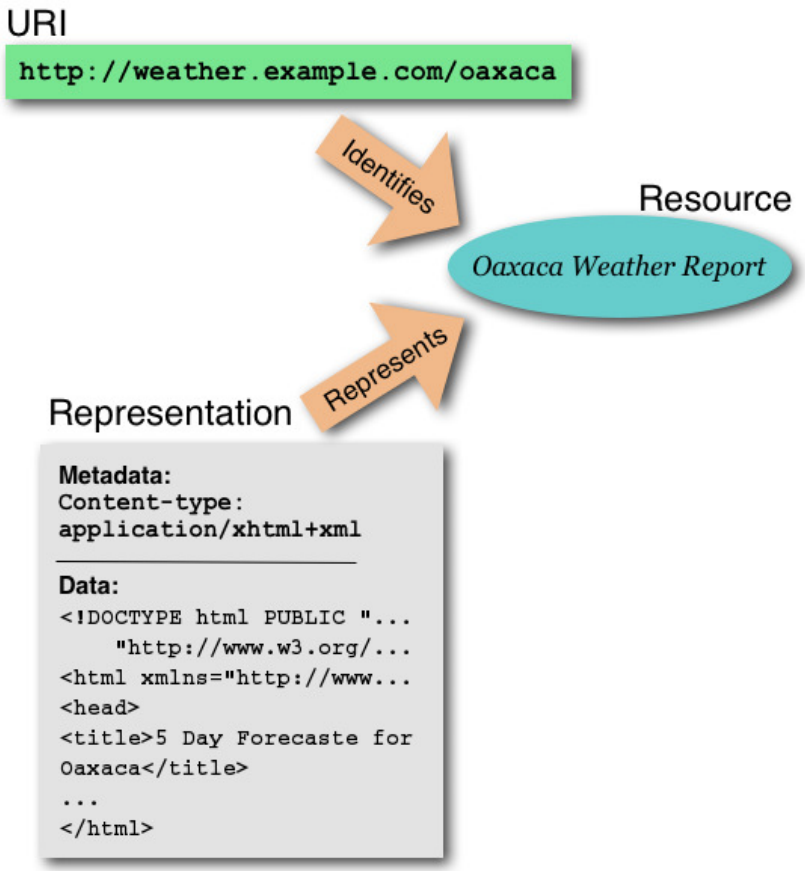
- **Inconvenientes**:
  - Sobrecarga de información en los mensajes por su formato.
  - Alto coste de procesamiento de los mensajes.
  - Seguridad.
    - También cierto para *Web Services* basados en REST

Representational State Transfer (REST)

- **Modelo de arquitectura** para sistemas hipermedia distribuidos como la World Wide Web. Definido por el W3C.
- **REST es una arquitectura, no un estándar**. Incluye el uso de estándares como HTTP, XML, MIME, etc.
- Este es el modelo de interacción habitual con la WWW.
- Los sistemas que siguen los principios REST se llaman con frecuencia ***RESTful***.
- Ejemplos:
  - Product Advertising REST API de Amazon Webservices
  - API REST de Twitter

Web Services basados en REST

- El sistema se compone de **recursos**,
  - Un recurso es cualquier elemento que debe ser accedido en el sistema distribuido.
- Los recursos se acceden a través de un identificador global (**URI**).
- Cada acceso a un recurso se contesta con una *Representación* del mismo.
  - Diferentes modos de ver el recurso, en función de la consulta o el estado.
- En la representación del recurso se pueden incluir enlaces a representaciones relacionadas o con mayor nivel de detalle.
- La recepción de la representación provoca un cambio de *Estado* en el elemento que la recibe.
- El acceso a los recursos se realiza mediante una interfaz uniforme, basada en el intercambio de mensajes HTTP, con los métodos estándar GET, PUT, DELETE, POST.



Extraído de: http://www.w3.org/TR/webarch/

Principios del diseño de Web Services con REST

- Identificar todos los recursos que se quieren exponer como servicios.
- Crear una URI para cada recurso. Al menos una.
  - Identificadas por *nombres* (parte, elemento, objeto…), no por *verbos* (get, modify…).
- Categorizar los recursos:
  - Acceso sólo de lectura. Accesibles mediante HTTP GET.
  - Acceso lectura / escritura. Accesibles mediante HTTP PUT, POST, DELETE.
- Especificar el formato de los datos intercambiados a/desde el recurso utilizando *esquemas* (normalmente en XML, JSON).
- Colocar enlaces en cada representación del recurso a niveles mayores de detalle o recursos relacionados.
- Representar los servicios mediante *Web Application Description Language*, WADL.
  - Análogo a WSDL para los Web Services tradicionales (orientados a SOAP).

Moodle: Ejercicio REST examen mayo 2014

Ejemplo: API REST de Twitter

- Documentación de la API: https://dev.twitter.com/rest/public
- Representación de datos: JSON

Moodle: Ipython Notebook

Publicar un tweet (POST)

URI

parámetros

```
# Generar JSON con los parámetros de la llamada
postData = {'status':'Mi n+1 post'}
# Codificamos los parámetros a formato URL
# (The text of your status update, typically up to 140 characters. URL encode as necessary...
urlquery = urllib.urlencode(postData)
# Generar la URL con los parametros
status_update = "https://api.twitter.com/1.1/statuses/update.json?%s" % (urlquery)
print status_update
```

```
https://api.twitter.com/1.1/statuses/update.json?status=Mi+n+post
```

```
response,data = client.request(status_update,"POST")
print pp.pprint(response)
print pp.pprint(json.loads(data))
```

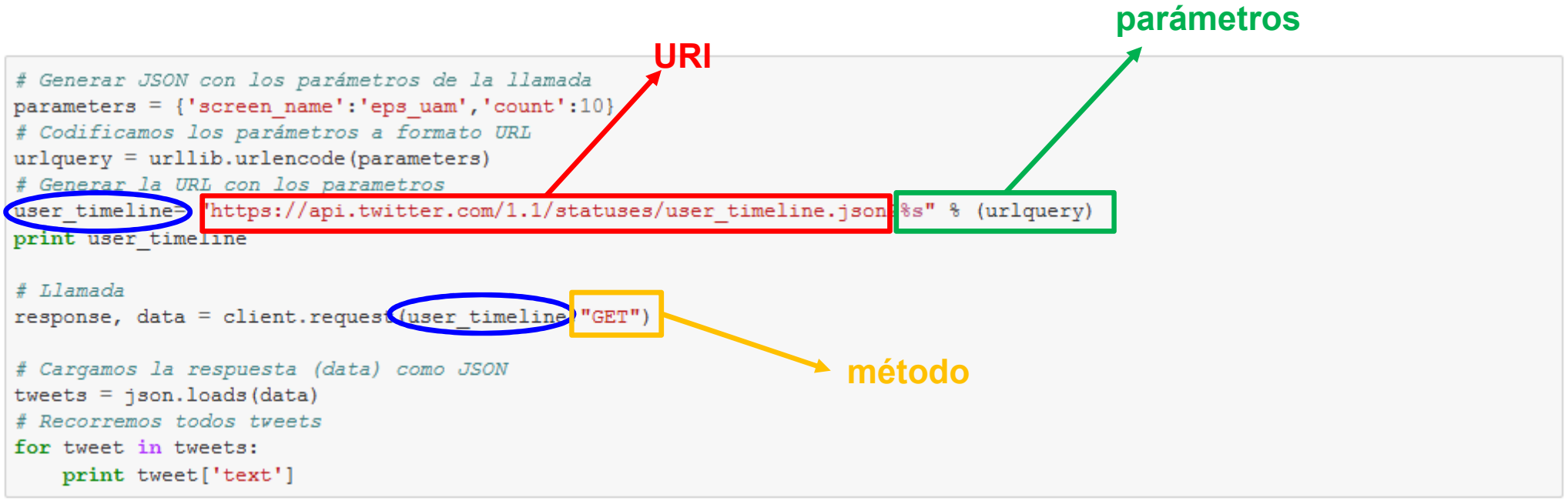
método

Ejemplo: API REST de Twitter

- Documentación de la API: https://dev.twitter.com/rest/public
- Representación de datos: JSON

Moodle: Ipython Notebook

Obtener los tweets de un usuario (GET)



Algunas diferencias WS SOAP vs WS REST

Ventajas de REST

- Ligero: no necesariamente sobre XML
  - Debe haber acuerdo/documentación en el tipo de información y cómo está se va a representar.
- Fácil de implementar: no hacen falta herramientas específicas.
- Es posible navegar de un recurso REST a muchos otros, simplemente siguiendo enlaces sin requerir el uso de registros u otra infraestructura adicional.
- Tiene problema de Seguridad (está sobre HTTP)

Ventajas de SOAP frente a REST

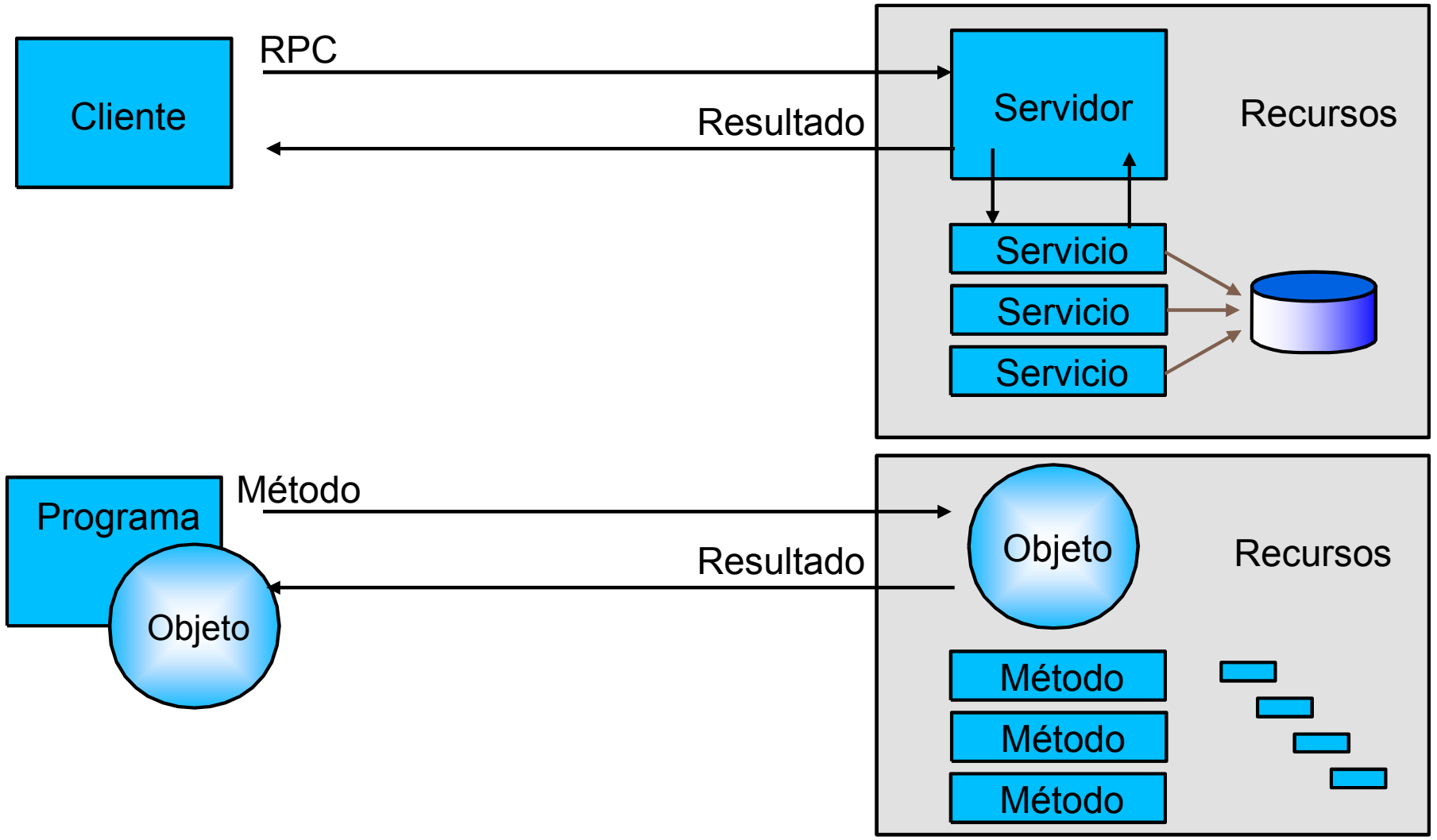
- Fácil de consumir → es un estándar.
- Rígido: tipado fuerte, sigue un contrato
- Herramientas de desarrollo → generación automática de stubs, etc

- Comparativa de implementación RPC versus REST

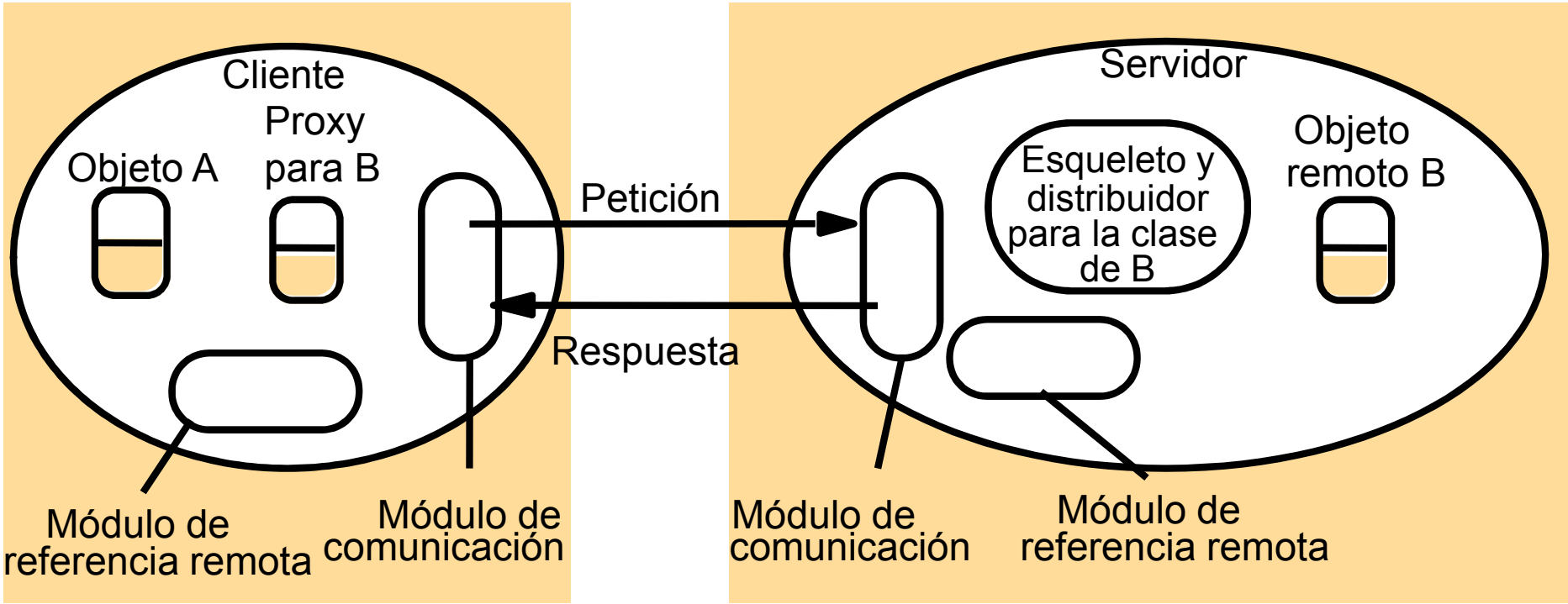
Sistemas de objetos distribuidos

- El modelo tradicional de la **programación estructurada** realiza llamadas a procedimientos para solicitar servicios de una librería o programa de aplicación.
- Los sistemas distribuidos extienden este modelo para solicitar servicios de un proceso remoto, mediante las llamadas a procedimientos remotos *(**Remote Procedure Calls, RPC**)*.
  - A través de un *middleware*.
- El modelo tradicional de **programación orientada a objetos** realiza llamadas a métodos para solicitar la ejecución de acciones por parte de un objeto.
- Del mismo modo, se puede realizar la comunicación entre objetos remotos mediante invocación de métodos remotos *(**Remote Method Invocation, RMI**).*
  - Precisa de un *middleware* específico para realizar la interconexión.

Esquema comparación C/S con POO



Implementación de RMI (I)



Implementación de RMI (II)

- **Módulo de comunicación**: Responsable de conseguir la comunicación entre los sistemas remotos y de la semántica adecuada de la llamada.
- **Módulo de referencia remota**: Responsable de traducir las referencias entre objetos locales y objetos remotos.
- **Objeto *Proxy***: Hace transparente la invocación al objeto remoto para el cliente.
  - Presenta los métodos del objeto remoto en local.
  - Realizar *parameter marshalling* y *unmarshalling*.
- **Distribuidor**: Determina el método del objeto remoto que se debe ejecutar.
- **Esqueleto**: Realiza el *parameter unmarshalling* y *marshalling* para ejecutar el método elegido y devolver los resultados.
- *Proxy*, distribuidor y esqueleto se generan de modo automático mediante un compilador de interfaces, al igual que en RPCs se generaban los *stubs* de cliente y servidor (con rpcgen en Sun RCP).

Plataformas de objetos distribuidos

- ***Object Management Architecture (OMA)***
  - Creada por el *Object Management Group*.
  - Arquitectura completa en torno a un bus de objetos, definido en la *Common Object Request Broker Architecture (CORBA).*

- ***.Net Remoting***
  - Arquitectura de objetos distribuidos de Microsoft.

- ***Java Remote Method Invocation, RMI*.**
  - Mecanismo del lenguaje Java para proporcionar soporte de objetos distribuidos.

Object Management Architecture

- Creada por el *Object Management Group*:
  - Fundado en 1989.
  - Objetivo: desarrollar, adoptar y promover estándares para el desarrollo e implantación de aplicaciones en entornos distribuidos heterogéneos.
  - El mayor consorcio de software del mundo: más de 700 miembros.
  - Establece la convergencia entre las plataformas distribuidas y Cliente / Servidor con la programación orientada a objetos.
  - http://www.omg.org.

- Establece dos modelos para el desarrollo e implementación de aplicaciones basadas en tecnología de objetos distribuidos.
  - **Modelo de objetos**: proporciona una definición abstracta de los conceptos relevantes del modelo (creación de objetos, identidad, peticiones, operaciones, tipos...) y conceptos relacionados con su implementación (métodos, entornos de ejecución, activación).
  - **Modelo de referencia**: Caracteriza las interacciones entre objetos, incluyendo componentes, interfaces y protocolos que componen la OMA.

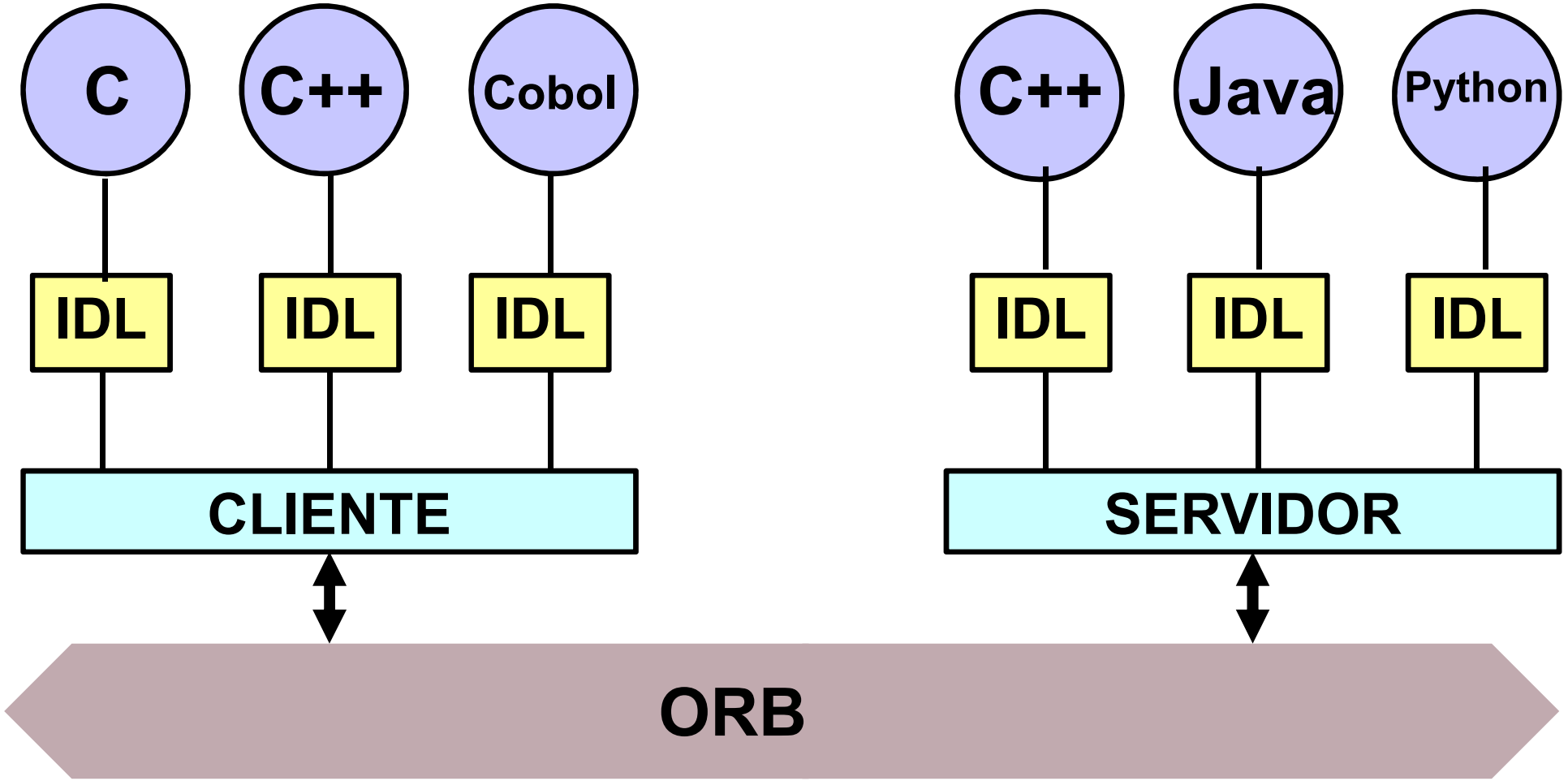
El modelo de objetos

- ¿Qué es un objeto CORBA? ¿Cómo conseguir interoperabilidad entre sistemas heterogéneos?
- Un objeto CORBA es un **objeto virtual**, es decir no tiene existencia por sí mismo si no hay detrás un objeto en un determinado lenguaje de programación que lo soporte.

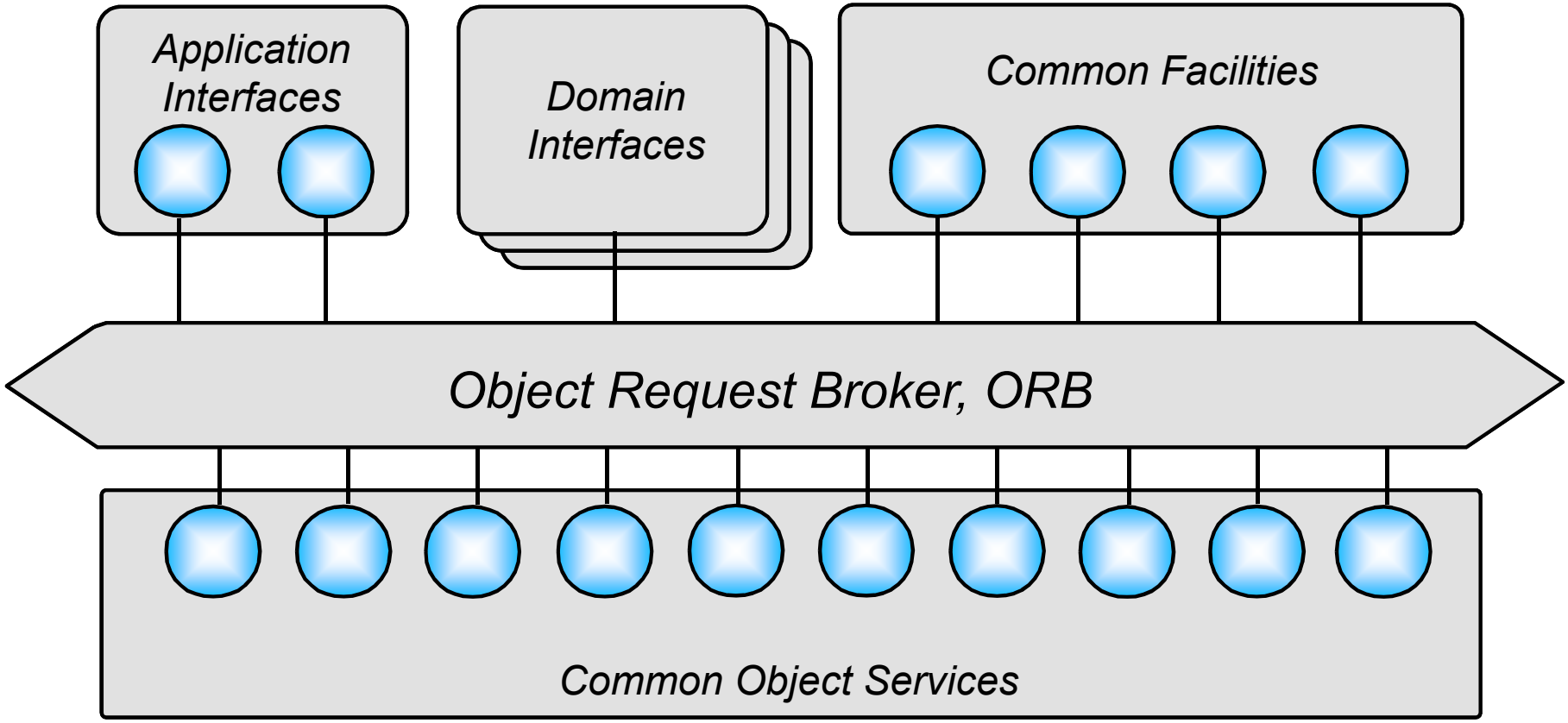
- **Objeto CORBA:**
  - Es una entidad que ofrece un servicio
  - Se define mediante el **IDL (Interface Description Language)**, pero puede estar implementado en cualquier lenguaje.
  - Se identifica de manera única mediante su referencia a objeto remoto
  - Siempre invocamos un método como `objeto.método(parámetros)` → transparencia de acceso

- La **comunicación con un objeto CORBA** puede ser:
  - Síncrona (exactamente una / como mucho una)
  - Asíncrona (como mucho una)
  - Sin respuesta o one-way
- **Cliente y servidor se relacionan siempre a través de interfaces.**

El modelo de objetos



Modelo de referencia



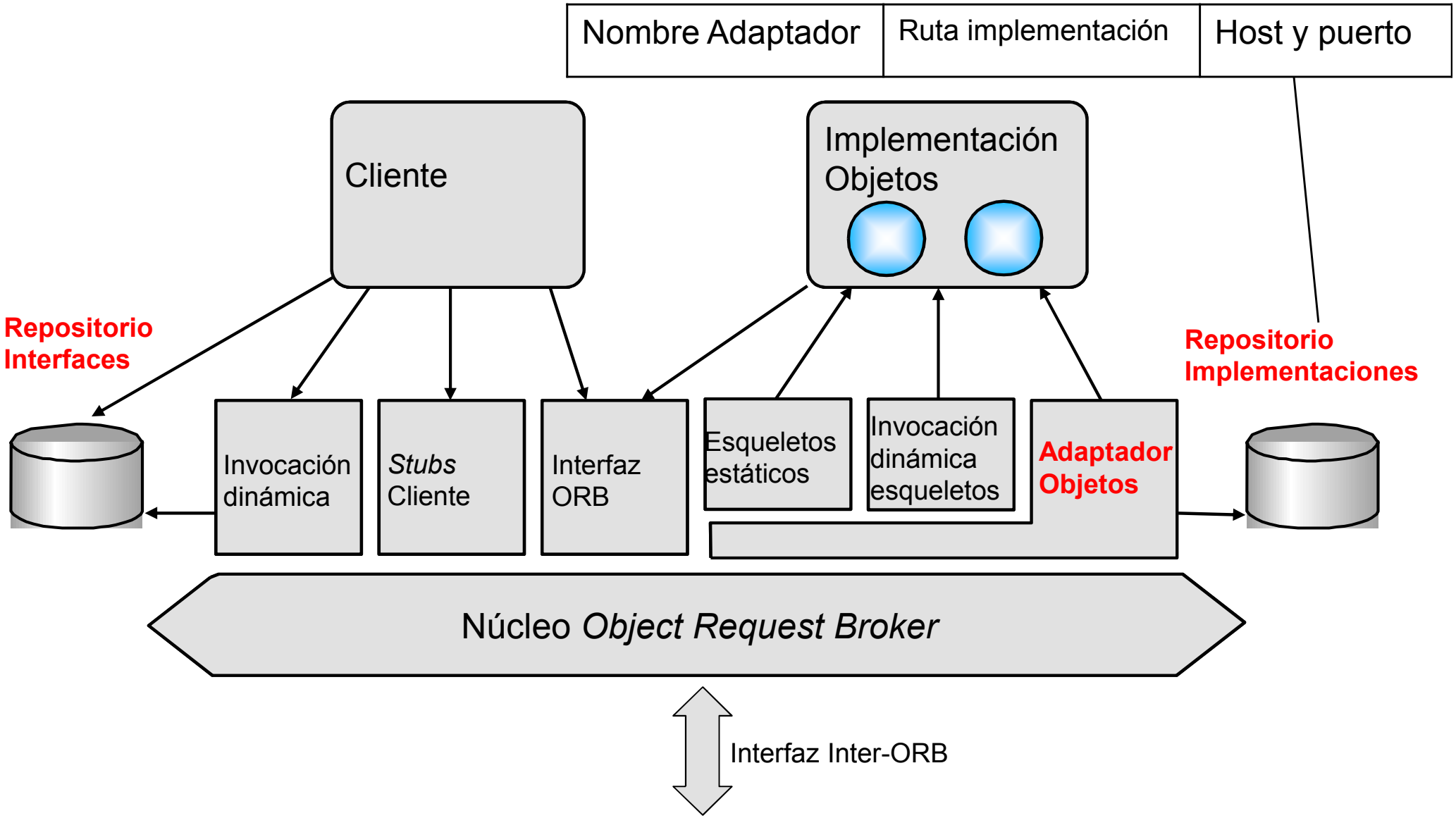
Elementos de la OMA

- ***Object Request Broker, ORB***. Bus de comunicación entre objetos.
- ***Common Object Services***: Componentes que implementan servicios al nivel del sistema. Extienden la funcionalidad del ORB.
  - Gestión del ciclo de vida, persistencia, resolución de nombres, eventos, tiempo, control de concurrencia, seguridad, colecciones…
- ***Common Facilities***: Colecciones de componentes, con funciones de tipo general, pero orientados a aplicaciones finales en vez de al sistema.
  - Interfaz de usuario, gestión de sistemas, gestión de información, gestión de tareas, servicio de correo electrónico, servicio de impresión
- ***Domain Interfaces***: Conjunto de servicios, similar a los anteriores, pero de tipo vertical: Específicos para cada área de aplicaciones.
  - Objetos comunes, comercio electrónico, telecomunicaciones, banca, salud, fabricación.
- ***Application Interfaces***: Interfaces específicas de aplicaciones concretas.

Object Request Broker, ORB

- Elemento central y principal de esta arquitectura.
- **Bus de objetos. Permite la comunicación entre ellos de forma transparente**.
- ***Middleware* avanzado**:
  - Permite llamadas estáticas y dinámicas a objetos. Incluye descubrimiento dinámico de objetos.
  - Lenguaje de descripción de interfaces independiente del lenguaje de programación (IDL).
  - Enlace directo de aplicaciones escritas en múltiples lenguajes de alto nivel (no necesariamente orientados a objetos).
  - Sistema auto-descrito (**introspección**). Genera meta-información sobre todo el sistema, consultable dinámicamente (**repositorio de interfaces**).
  - Transparencia de ubicación y de acceso. Aplicaciones funcionan igual en local que en remoto → **objeto.método(parámetros)**
  - Soporte de seguridad y autenticación de las comunicaciones.
  - Soporte de transacciones.
  - Polimorfismo en la ejecución de funciones asociadas a un mismo mensaje.
- Especificado en la *Common Object Request Broker Architecture, CORBA*.

Common Object Request Broker Architecture, CORBA



Protocolos Inter-ORB

Especifican la forma de interconectar **ORBs de distintos fabricantes** (CORBA 2.0).

Elementos definidos:

- ***General Inter-ORB Protocol (GIOP):***
  - Formatos de mensajes y representación de datos común para todos los ORBs
  - ***Common Data Representation, CDR***: estándar para la representación de los tipos de datos especificados en el IDL en mensajes GIOP.
  - ***Interoperable Object References, IOR****:* Localizador de objetos en un entorno multi-ORB.
- ***Internet Inter-ORB Protocol (IIOP):***
  - Especifica el intercambio de mensajes GIOP sobre protocolo de transporte **TCP/IP.**
  - Implementación de GIOP sobre IIOP es obligatorio en ORBs compatibles CORBA 2.0 o superior.
- ***Environment-Specific Inter-ORB Protocols (ESIOP):***
  - Interconexión de ORBs a través de *Middlewares* específicos.
  - Definido el DCE/ESIOP

Interface Definition Language, IDL

- Similar a RPC XDL y a DCE IDL.
- Definición de interfaces mediante IDL.
  - Independientes del lenguaje de programación.
- **Compilación de IDL**. Genera:
  - *Stubs* de cliente*.*
  - *Stubs* de servidor*.* CORBA los denomina esqueletos *(skeletons).*
  - Ficheros de definiciones.
- **Traductor de IDL** genera código fuente del lenguaje elegido.
  - CORBA especifica la correspondencia entre IDL y los lenguajes de programación**: *Language Mappings****.***
  - CORBA los especifica para C, C++, SmallTalk, ADA95, Cobol, Java…

Estructura IDL

```
module <identificador>
{
    <Declaraciones de tipos>
    <Declaraciones de constantes>
    <Declaraciones de excepciones>

    interface <identificador> [:<herencia>]
    {
        <Declaraciones de tipos>
        <Declaraciones de constantes>
        <Declaraciones de excepciones>

        <Declaraciones de atributos>

        [<tipo>] <identificador> (<parámetros>)
            [raises excepción] [context]
        ...
    }
    interface <identificador> [:<herencia>]
    ...
}
```

Define contexto de nombres

Define interfaz CORBA

Define atributos

Define operación

Ejemplo IDL

```
module StockObjects {

  struct Quote {
    string symbol;
    long at_time;
    double price;
    long volume;
  };

  exception Unknown{};

  interface Stock {

    // Returns the current stock quote.
    Quote get_quote() raises(Unknown);

    // Sets the current stock quote.
    void set_quote(in Quote stock_quote);

    // Provides the stock description,
    // e.g. company name.
    readonly attribute string descript;
  };
```

```
  interface StockFactory {

    Stock create_stock(
      in string symbol,
      in string description
    );
  };
};
```

Referencia a objeto remoto

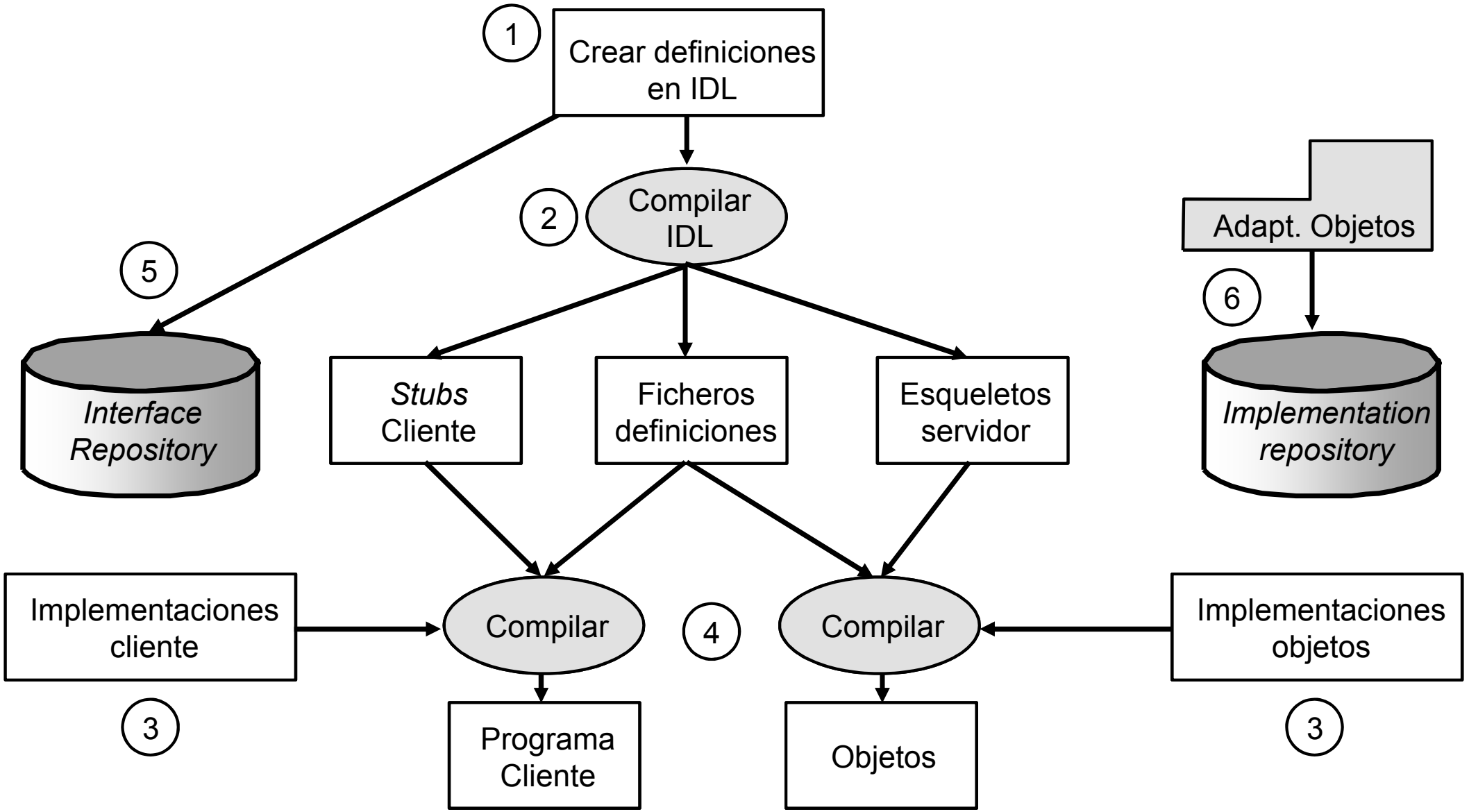
Moodle: Ejemplo de aplicación distribuida en Java mediante CORBA

Moodle: Comparativa de implementación CORBA versus RPC

Operativa CORBA

- Dos modos básicos de funcionamiento del Cliente CORBA:
- **Invocación estática:**
  - El cliente conoce la estructura del servidor, las operaciones (métodos) que realiza y sus parámetros.
  - Resolución de las llamadas en tiempo de compilación.
- **Invocación dinámica**:
  - Se descubren los objetos en tiempo de ejecución.
  - Generación dinámica de las llamadas.
  - El cliente consulta el repositorio de interfaces.
  - Ejemplo: Un “navegador” de objetos que quiera mostrar en tiempo real información sobre los objetos CORBA disponible en varios servidores de un sistema distribuido.

Ciclo de desarrollo CORBA (I)

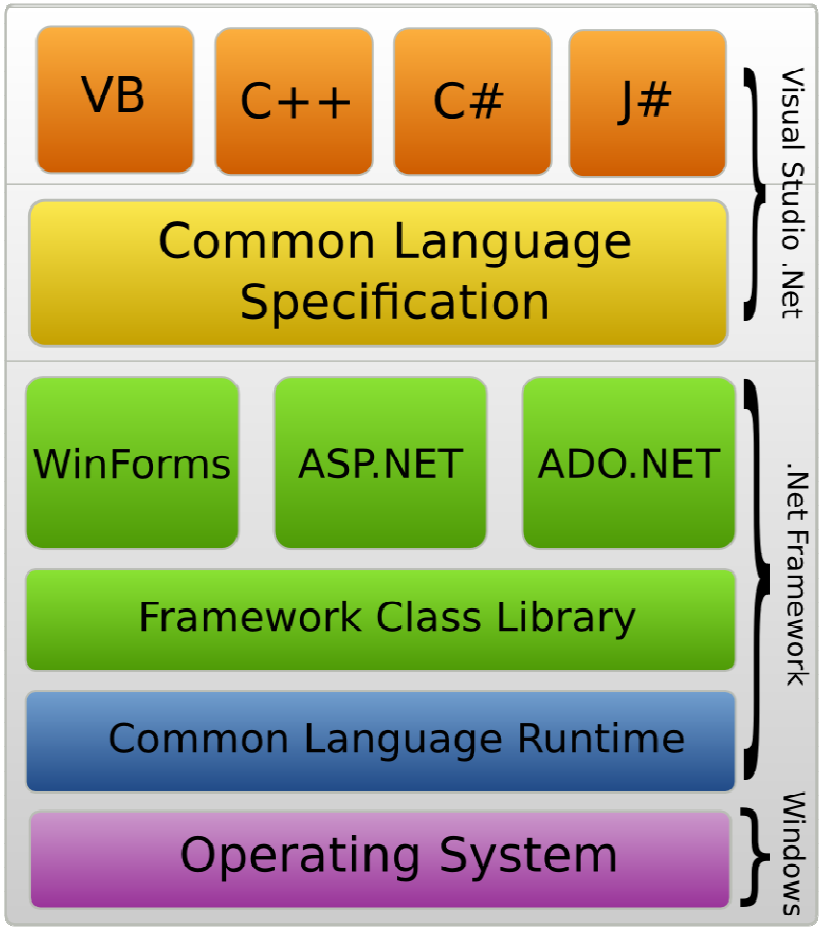


Ciclo de desarrollo CORBA (II)

1. Definir las clases de objetos utilizando el IDL.
2. **Compilar el IDL**. Genera, al menos, tres tipos de ficheros para el lenguaje de programación que se seleccione:
   - Ficheros de definiciones básicas, comunes a cliente y servidor.
   - *Stubs* de cliente para los métodos definidos en el IDL.
   - Esqueletos de servidor.
3. Desarrollar el código que implementa clientes y servidores.
4. Compilar conjuntamente el código generado por el compilador de IDL y el código desarrollado para generar los programas ejecutables.
5. Publicar las definiciones de clases realizadas en el ***Interface Repository***.
6. En el proceso de llamada a objetos definidos a través del **Adaptador de Objetos**, este registra las instancias creadas en el ***Implementation Repository***.

Arquitectura .Net

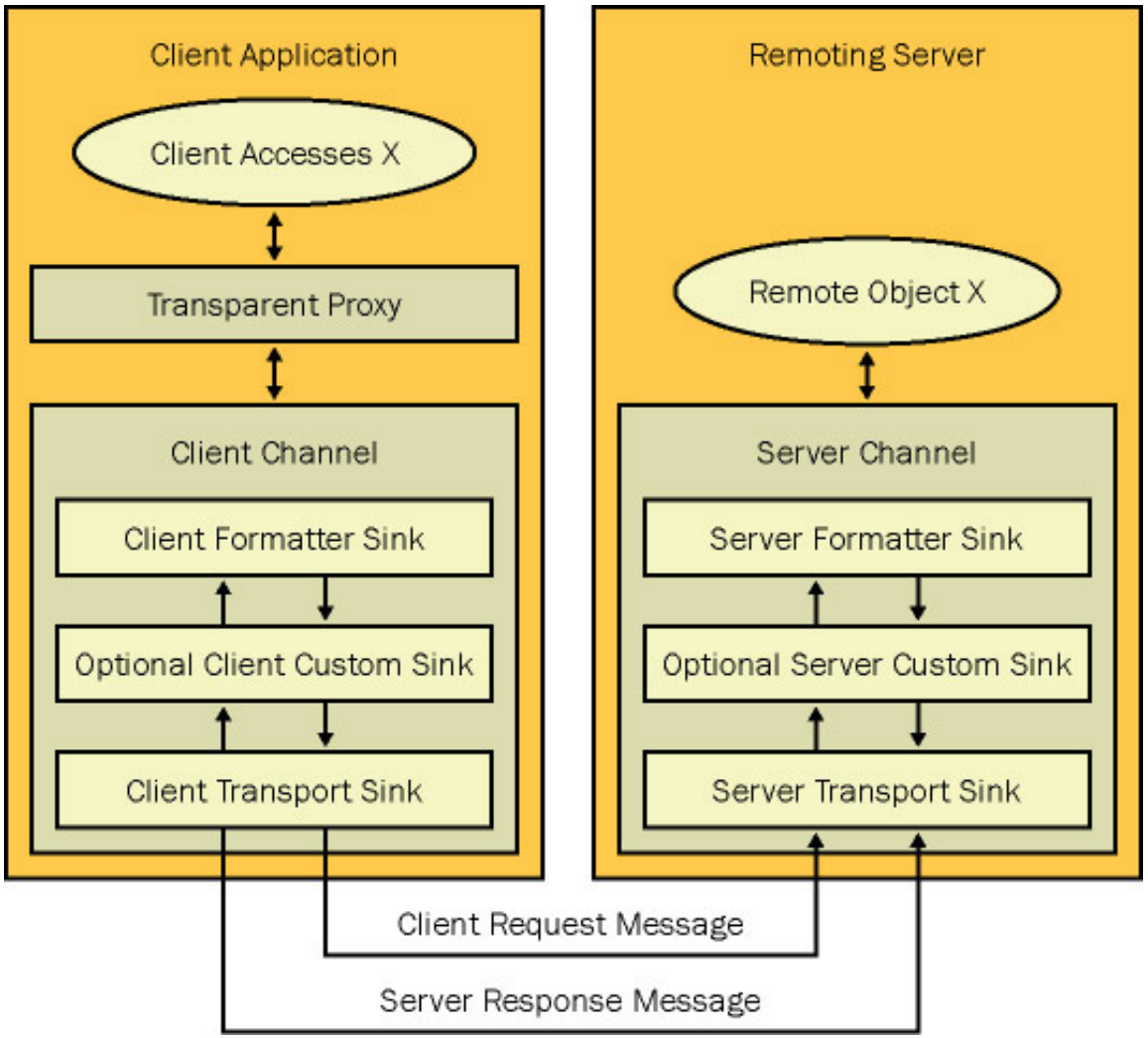
- Solución de Microsoft para el desarrollo de aplicaciones Web.
- Tres componentes principales:
  - Un entorno de ejecución de aplicaciones independiente del lenguaje: *.NET Framework*.
  - Un entorno de desarrollo de aplicaciones: *Visual Studio .NET*.
  - Un sistema operativo sobre el que ejecutarlo todo: *Windows Server*.
- Objetivo: Plataforma única, múltiples lenguajes de programación.
  - Compilados a *Microsoft Intermediate Language (MSIL)*.



.Net Remoting

- API de Microsoft para la comunicación entre objetos distribuidos.
- Evolución de varias tecnologías: OLE, COM, DCOM y COM+.
- Soporta objetos en varios lenguajes: limitada a lenguajes CLS.
- Como protocolos de comunicación soporta: SOAP y TCP/binary.
  - Fácil de adaptar a otros protocolos.
- Soporta dos tipos de activación de objetos en el servidor:
  - Activación por parte del servidor (singleton y single-call).
  - Activación por parte del cliente (método CreateInstance).
- Las interfaces se pueden describir en el mismo lenguaje CLS.
- A partir de la versión 3.0 de .NET se integra en Windows Communication Foundations (introduce un modelo de desarrollo basado en la orientación a servicios en vez de a objetos).

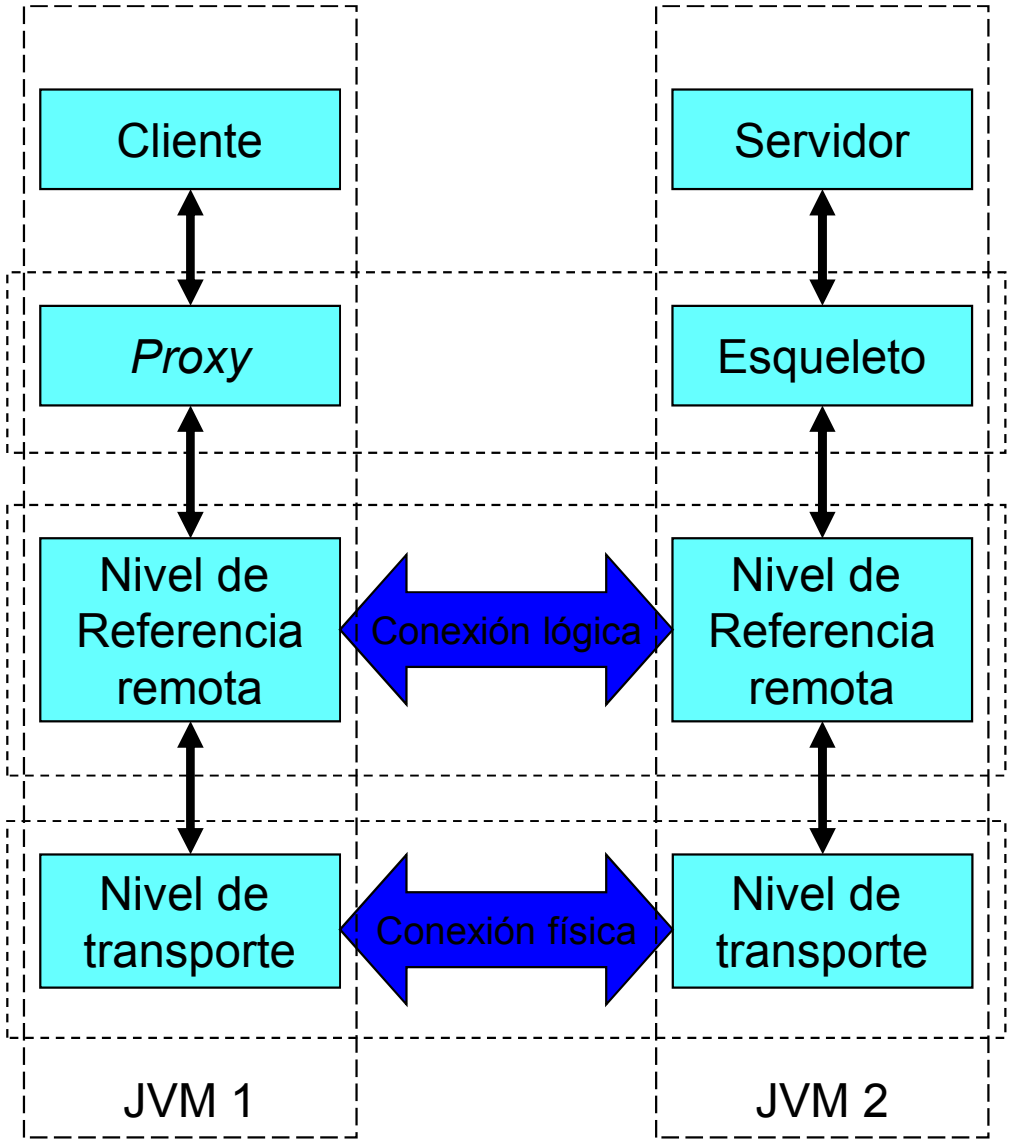
Arquitectura de .Net Remoting



- Proxy: Creado de forma automática al activar el objeto remoto. Recibe invocaciones de métodos y crea y pasa el mensaje al canal.
- Formatter Sink: serializa / deserializa el mensaje. El sistema permite añadir custom sinks personalizados.
- Transport sink: Se encarga de establecer la conexión, enviar el mensaje a través de la red, y esperar la respuesta.
- Dos canales disponibles: HttpChannel y TCPChannel. Se pueden añadir canales personalizados fácilmente.

Java Remote Method Invocation, RMI

- Sistema de objetos Java distribuidos.
  - No hay soporte para objetos programados en otro lenguaje.
- Existe desde la versión del *Java Develompent Kit (JDK) 1.1*.
- **Más sencillo que Corba.**
- Arquitectura basada en niveles
  - *Proxy* - Esqueleto.
  - Nivel referencia remota.
    - Nombres y registro objetos.
    - Gestiona referencias remotas.
    - *Remote garbage collection*.
  - Nivel transporte.
    - *Java Remote Method Protocol* sobre TCP/IP.
    - RMI sobre IIOP.



Programación con Java RMI

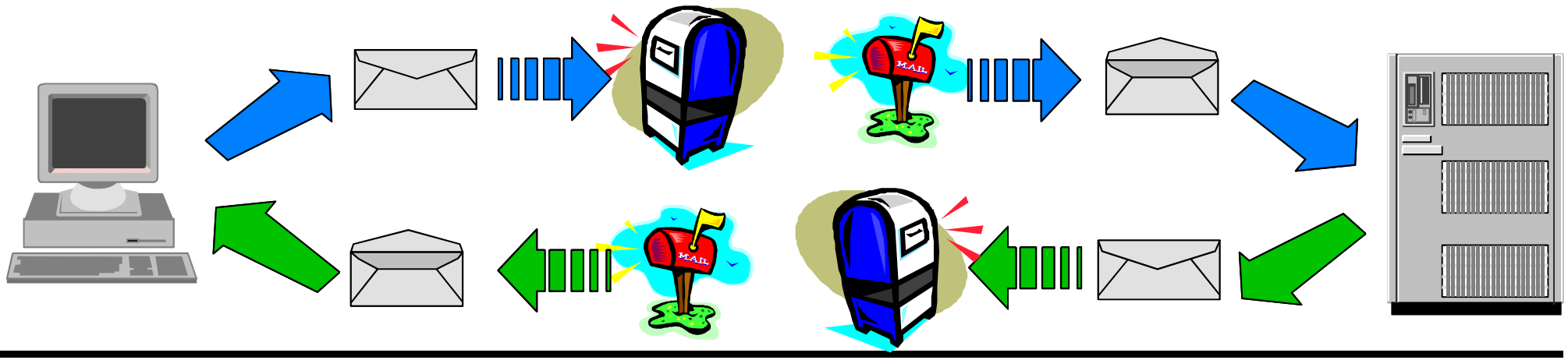
- **Definir las interfaces** remotas de las clases. Sólo incluyen los métodos.
  - Extienden la interfaz *java.rmi.Remote*.
  - Deben lanzar la excepción *java.rmi.RemoteException*.
- **Implementar las clases remotas**.
  - Implementan la interfaz remota definida previamente.
  - Extiende la clase *java.rmi.server.UnicastRemoteObject*.
  - Esta clase enlaza con el RMI e inicializa el objeto.
- Crear *proxy* y esqueleto mediante el **compilador rmic**.
  - Genera las clases *Stub* y *Skel* automáticamente a partir de la clase remota.
  - Hasta la versión 1.1 de Java había que generar tanto *Stub* como *Skel.*
  - A partir de Java 2, no hace falta generar el esqueleto. La localización del método a invocar se realiza mediante Java Reflection (invocación dinámica).
- **Creación de una aplicación como servidor** de la clase remota realizada.
  - Crea la instancia de la clase remota y la registra en el servicio de nombres.
- Arrancar ***RMIRegistry*** (servicio nombres y registro) y el servidor de la clase remota.
- Crear **programas clientes**.
  - Obtienen referencias a los objetos remotos a través del servicio de nombres.

Paso de parámetros en Java RMI

- Todos los parámetros de un método son de entrada.
- El único parámetro de salida es el resultado del método.
- Los parámetros de tipos primitivos se pasan por valor.
- Admite objetos locales como parámetros siempre que sean serializables.
  - Se pasan siempre por valor.
- Admite objetos remotos como parámetros.
  - Se pasan siempre como referencia.

Comunicación mediante colas de mensajes

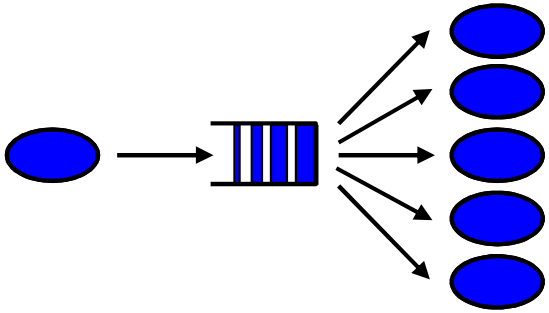
- *Message Oriented Middleware, MOM*.
- Intercomunicación mediante colas de mensajes. Proceso **asíncrono ("desacoplado en el tiempo")**:
  - Aplicación genera un mensaje.
  - Lo pone en una cola, y sigue procesando.
  - La aplicación destino lo extrae de la cola cuando pueda procesarlo.
  - La respuesta sigue el mismo procedimiento.



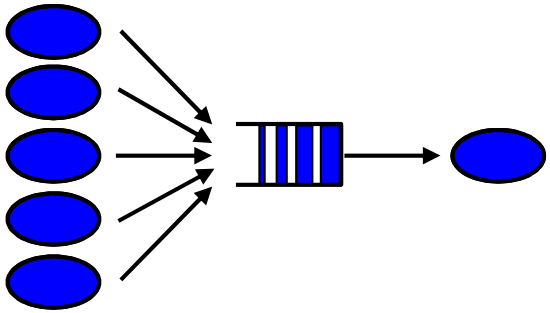
Características

- Sistema no orientado a conexión desde el punto de vista cliente-servidor (los mensajes pueden no llegar en orden)
- Permite conexiones:
  - Uno a uno.

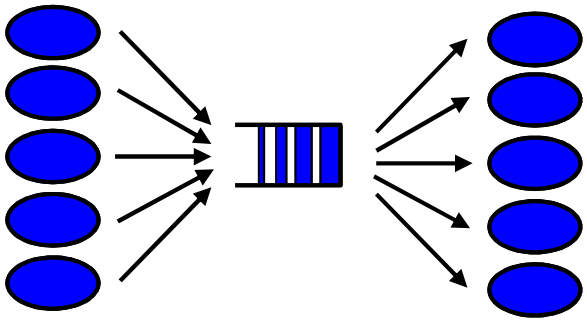
  - Uno a muchos.



- Muchos a uno



- Muchos a muchos



Situaciones ideales para MOM

- Conectividad Cliente / Servidor no permanente y costosa.
- Paso de mensajes temporizado (copias bases de datos, entrada de datos masiva, generación de informes).
- Múltiples servidores procesan mensajes de clientes.
  - Permite realizar **balanceo automático de carga**.
- Llegada de mensajes impredecible o en ráfagas.
- El cliente no puede esperar la respuesta para continuar proceso.
- Proceso de mensajes con prioridades (no *FIFO*).
- Sistemas tipo publicación / suscripción.
- *Workflow*: envío de mensajes en cadena de aplicaciones.

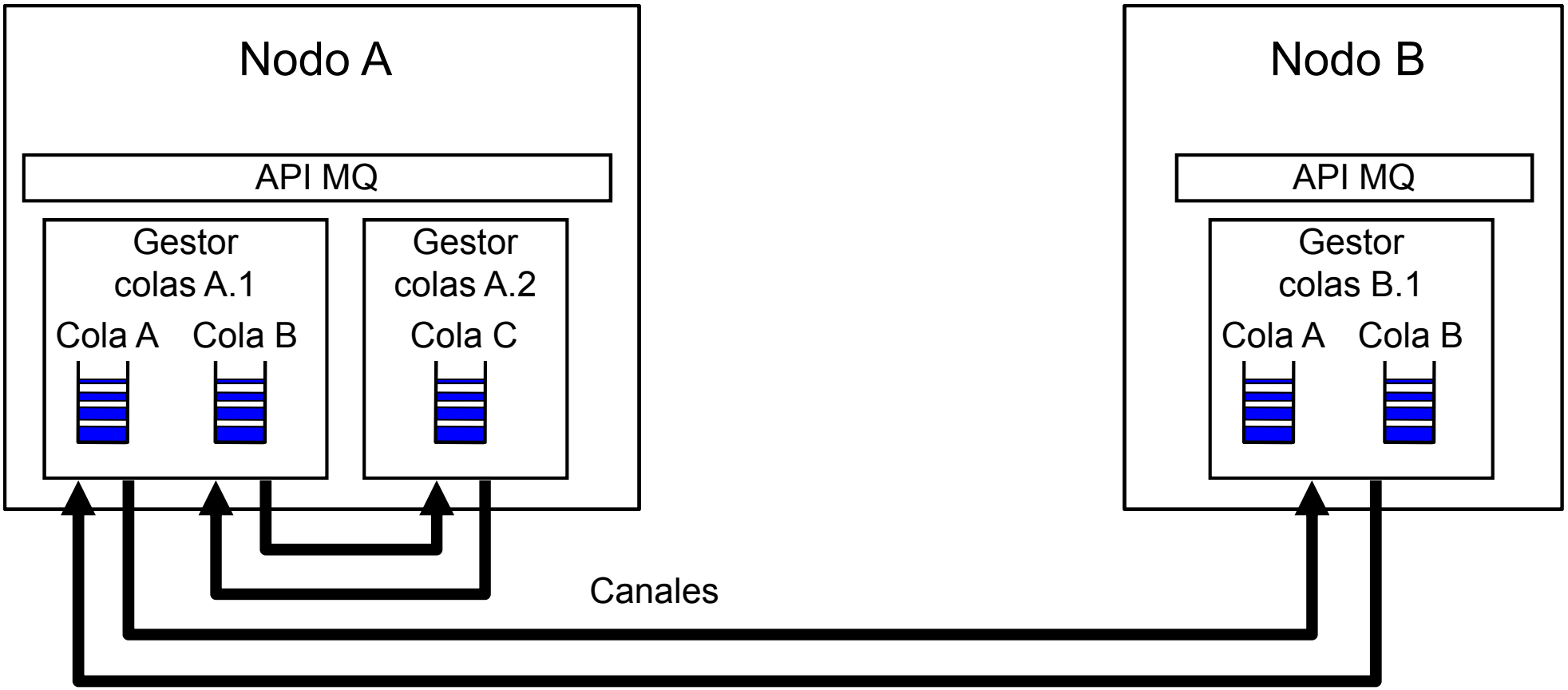
Ejemplos de gestores de colas

- *IBM WebSphere MQ*
  - Producto de IBM. MOM más extendido (Antes *MQSeries*).
  - **Multiplataforma**:
    - Hardware: IBM (zSeries, iSeries, xSeries, pSeries) y no IBM (Sun, HP, Tandem, Vax, PCs, etc).
    - Software: zOS, Unix (AIX, Solaris, HP-UX, Linux…), Windows, etc.
  - **Multiprotocolo**: TCP/IP, NetBios, DecNET, IPX/SPX, APPC...
- Productos similares de otros fabricantes:
  - Microsoft MSMQ (Integrado en Windows NT-2000).
  - BEA Systems *MessageQ*.
  - Oracle *Advanced Queueing*.
  - *xmlBlaster (open source).*

Elementos básicos *WebSphere MQ*

- **Gestores de colas *(Queue Managers, QM)*:**
  - Crea y gestiona el resto de los elementos.
  - Maneja el envío y recepción de mensajes en las colas.
  - Comunica con los programas de aplicación a través de una API:
    - *Message Queuing Interface, MQI.*
    - *Application Messaging Interface, AMI.*
    - ***Java Messaging Services, JMS.***
  - Puede haber más de uno en cada nodo de la red. Definido uno por defecto.
- **Colas de mensajes:**
  - Almacenes de mensajes de comunicación entre aplicaciones.
  - Múltiples tipos.
- **Canales de comunicación:**
  - Conexiones unidireccionales entre **gestores de colas**.
  - Necesario definirlos en los dos gestores de colas, origen y destino.
  - Compartición conexiones de transporte (*sockets*, sesiones...) por muchas aplicaciones.
  - Envío de mensajes planificado *(scheduled)*.
- **Canales cliente:** utilizados para acceder de forma remota al gestor usando RPCs.

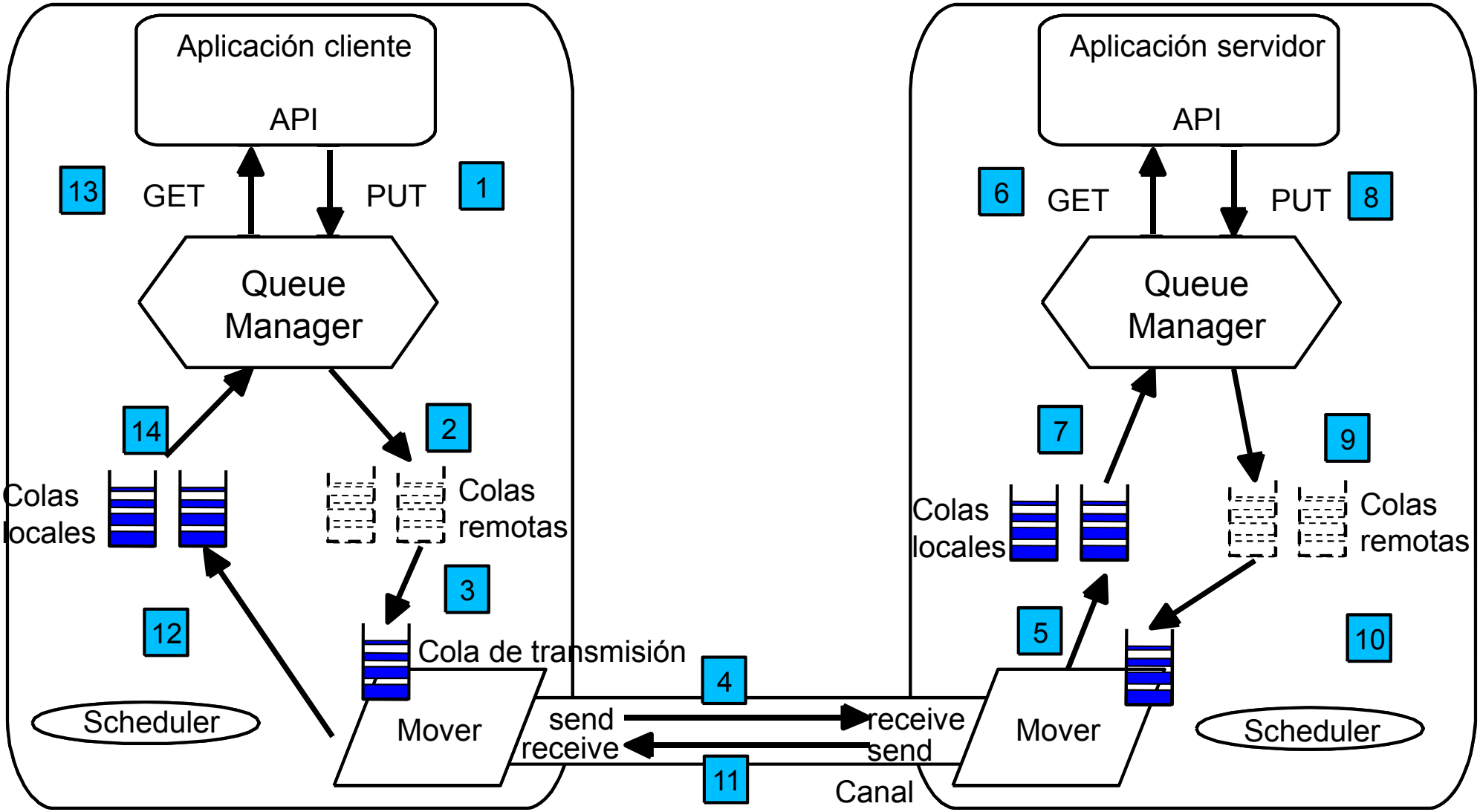
Estructura básica *WebSphere MQ*



Principales tipos de colas

- **Locales**: Pertenecen al gestor al que se conecta el programa.
  - **Remotas**: Pertenecen a un gestor distinto.
- **Persistentes**: Los mensajes que contienen se mantienen aunque se pare el gestor de colas.
  - **Temporales**: Los mensajes desaparecen al cerrar la cola.
- **Estáticas**: Definidas permanentemente en el sistema.
  - **Dinámicas**: Colas locales creadas por una aplicación. Ambas pueden ser persistentes o temporales.
- **Modelos**: Colas empleadas para definir una cola dinámica, heredando los atributos de ésta.
- **De transmisión**: Empleadas para enviar mensajes entre gestores de colas.
- **De activación *(initiation queues)***: Reciben mensajes que producen activación de procesos cuando ocurren determinados eventos *(triggers)*.
- **De cartas muertas *(dead-letter queues)*:** El gestor coloca en ellas los mensajes que no ha podido entregar.

Dentro de un sistema de colas de mensajes



Dentro de un sistema de colas de mensajes (II)

1. La aplicación cliente usa la API MQ PUT para enviar un mensaje a su gestor de colas local.
2. El gestor de colas reconoce la cola destino como una cola remota.
3. De la definición de la cola remota se conoce el gestor de colas destino. Se envía el mensaje a la cola de transmisión que enlaza con su sistema.
4. El proceso de comunicaciones *(mover)* extrae mensajes de su cola de transmisión y los envía al sistema destino a través de un canal de comunicación.
5. El proceso de comunicaciones *(mover)* del sistema destino recibe el mensaje y lo deja en la cola local correspondiente.
6. La aplicación servidora utiliza la API MQ GET para recibir un mensaje de una cola local.
7. El gestor de colas local extrae un mensaje de la cola solicitada y lo entrega al servidor.
8. (y ss.) La aplicación servidora realiza un proceso similar para enviar el mensaje de contestación.

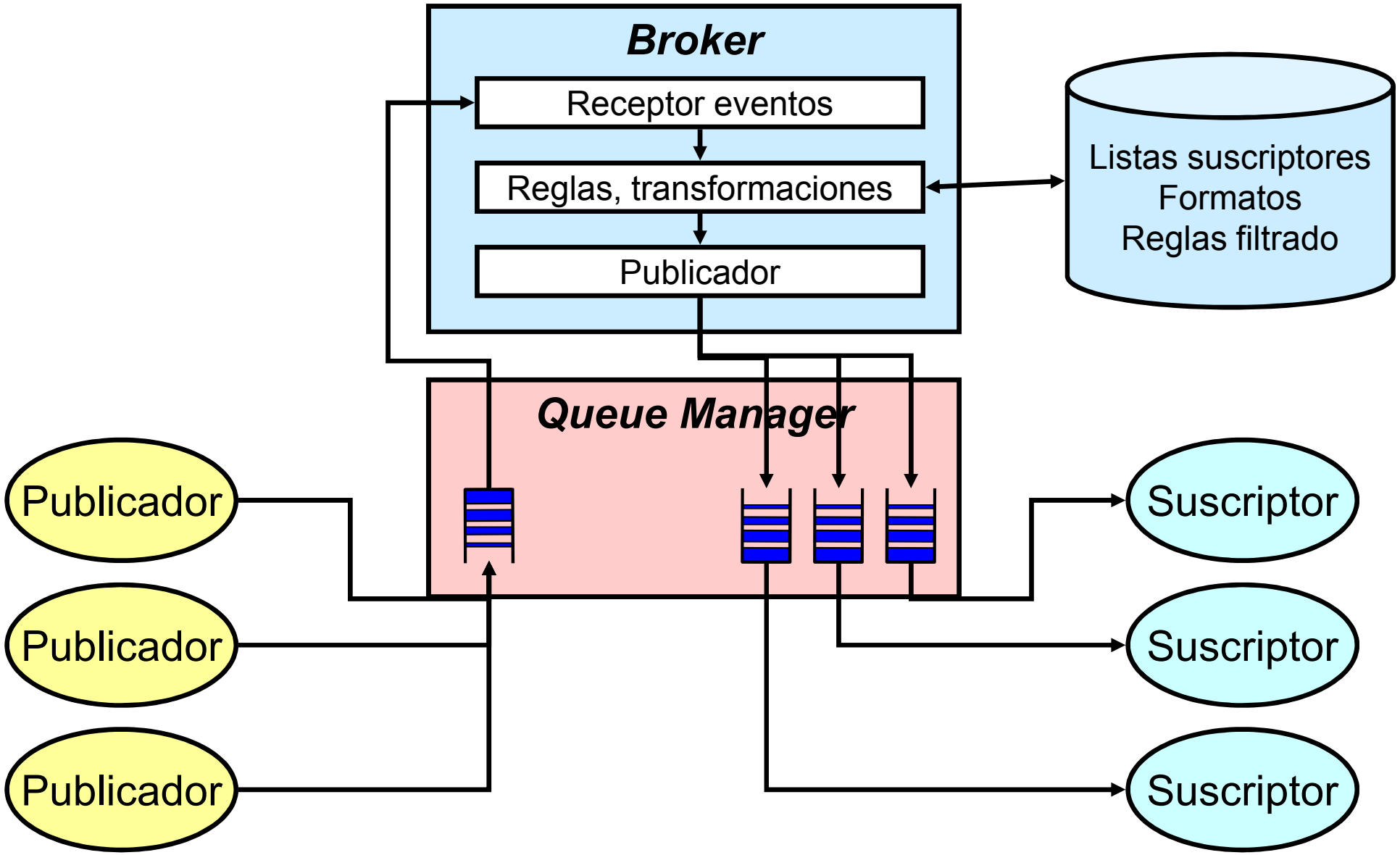
Comparación MOM - RPC

Características	***Message Oriented Middleware***	***Remote Procedure Calls, RPC***
Modelo	Correo	Llamada telefónica
Temporización	Asíncrona. Clientes y servidores pueden trabajar a distintas velocidades.	Síncrona. Trabajo concurrente.
Secuencia de arranque sistema	No fija	Los servidores deben estar disponibles antes que comiencen los clientes
Estilo	Mensajes encolados.	Llamada / Respuesta.
Extremos listos simultáneamente	No	Sí
Filtrado de mensajes	Posible	No posible
Rendimiento	Bajo. Requiere almacenamiento intermedio	Alto. Comunicación directa sin intermediarios.
Proceso asíncrono	Sí	Limitado. Requiere creación de tareas y sincronización entre procesos.

Modelo Publicador / Suscriptor con MOM

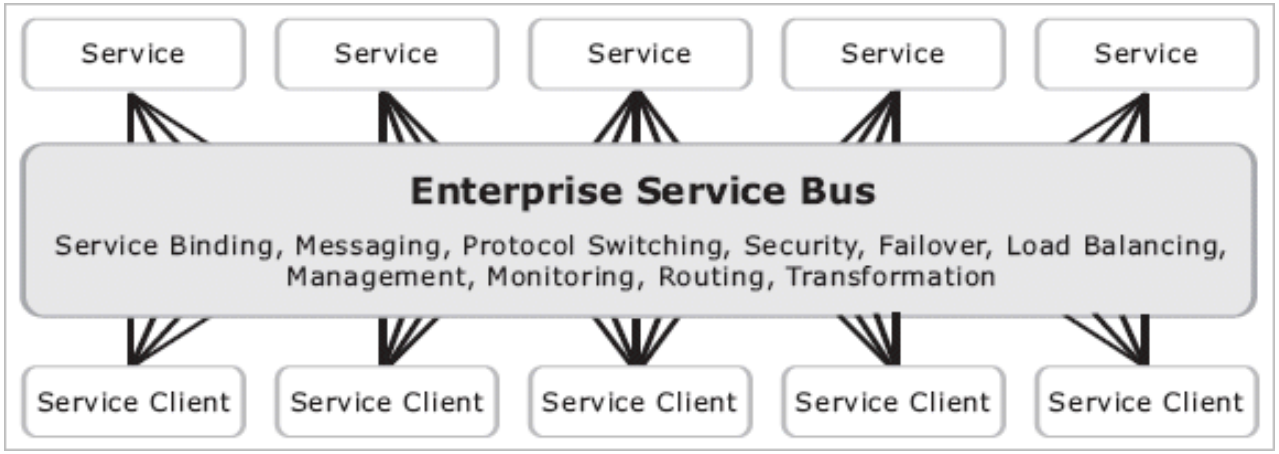
- **Modelo de diseño de aplicaciones** aplicable a sistemas distribuidos.
- Existe un **proceso intermediario *(broker)*** que gestiona los intercambios de información.
- Los procesos publican información al *broker*.
- Los procesos envían al *broker* solicitudes de suscripción a información concreta.
  - Pueden realizarlo a través de condiciones de filtrado.
- Cuando el *broker* recibe información de un publicador:
  - Busca la lista de suscriptores de dicha información.
  - Para cada suscriptor:
    - Evalúa las reglas de filtrado. Descarta información no deseada.
    - Adapta la información al formato en el que la espera recibir el suscriptor.
    - La envía al suscriptor.
- Adecuado a la comunicación mediante colas de mensajes asíncronas.
- Muchos sistemas de mensajes aceptan modelo publicador/suscriptor.

Esquema de funcionamiento



Enterprise Services Bus, ESB

- **Arquitectura software. Extensión del concepto de Broker de mensajes a modelos sin relación Publicador-Suscriptor**. Generalmente implementado mediante colas de mensajes.
- **Objetivo**: integrar servicios y clientes en sistemas heterogéneos, reduciendo acoplamiento.
- Su finalidad también es eliminar los intercambios uno a uno entre componentes de un sistema distribuido.
- El ESB actúa como proceso centralizador de solicitudes de los clientes (normalmente mensajes o solicitudes de ejecución de acciones tipo RPC – Web Services) para su distribución a los servidores.
  - Más fácil de monitorizar fallos y rendimiento y facilita el reemplazo de componentes.
- Núcleo base para la ***Services Oriented Architecture, SOA***.



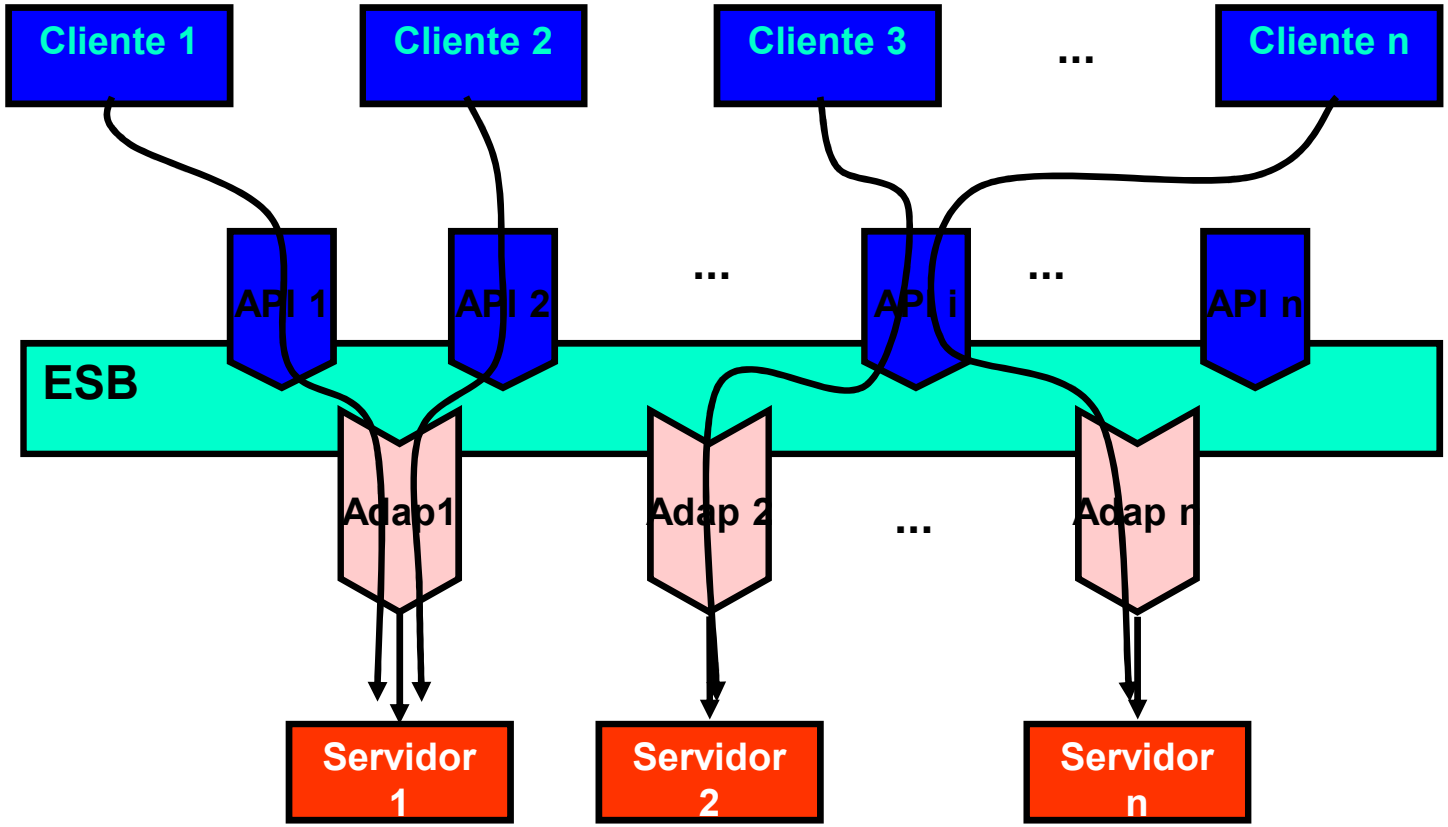
http://docs.oracle.com/cd/E21764_01/doc.1111/e15020/introduction.htm#OSBCA107

Funciones del *ESB*

Se organizan en **tres niveles**:

- **Conexión**: esta capa nos aporta “conectividad universal”
  - APIs clientes estándar: MQ API, JMS, .NET,
  - Protocolos estándar: SOAP/HTTP...
  - Adaptadores para enlazar con sistemas o productos existentes: SAP, CICS...
- **Comunicaciones**:
  - Interconexión entre todos los servicios conectados a él.
  - Proporciona alto rendimiento, escalabilidad, alta disponibilidad, seguridad y fiabilidad.
  - Modelos punto a punto, petición-respuesta y publicación-suscripción.
  - Diversas calidades de servicio según sea necesario (entrega inmediata, garantizada, con integridad transaccional, etc.).
- **Mediación**:
  - Transformación semántica de los mensajes conforme sea necesario entre extremos de la comunicación.
  - Es posible implementar flujos de procesos en el propio ESB que se arranquen tras la recepción de un mensaje.

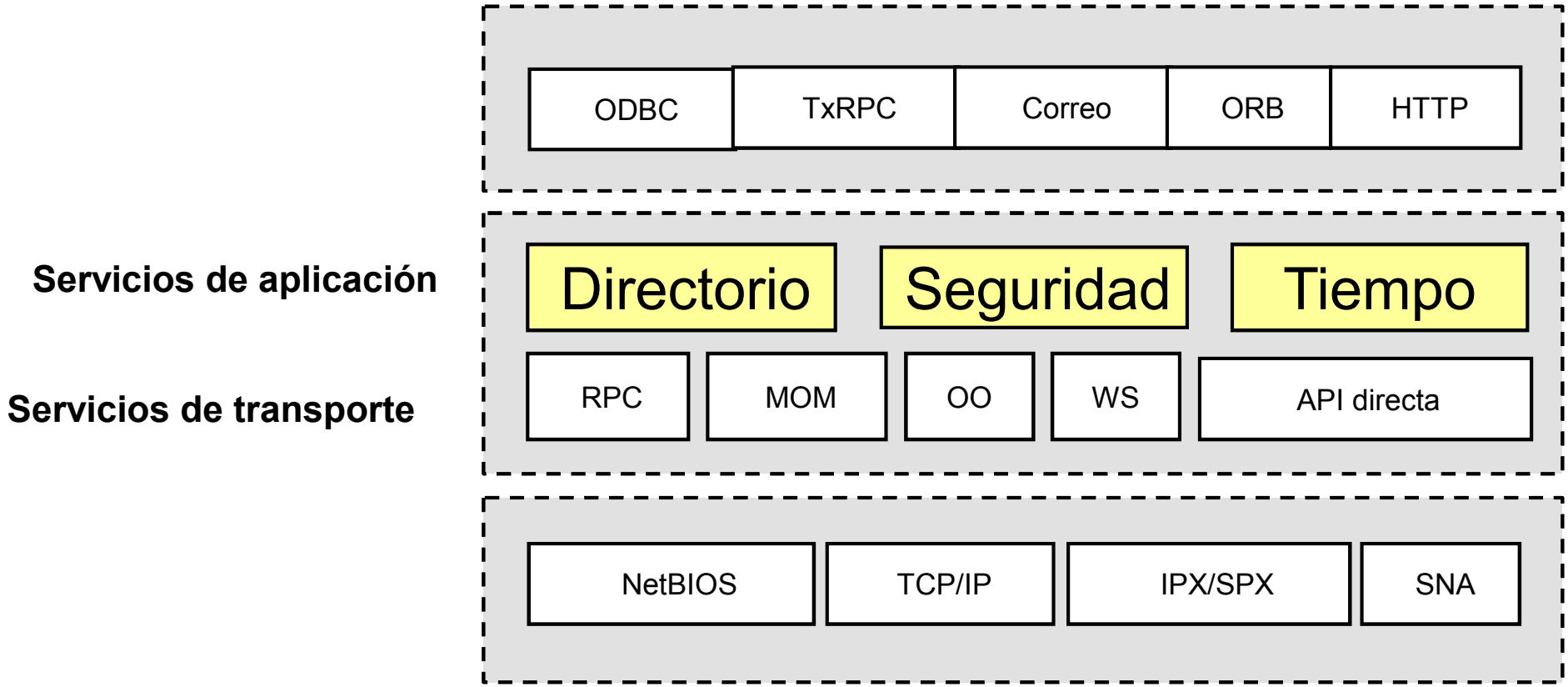
Enterprise Services Bus, ESB



Ventajas e inconvenientes *ESB*

- **Ventajas**:
  - Adaptación rápida en entornos existentes.
  - Flexibilidad. Fácil de adaptar a nuevos requerimientos.
  - Basado en estándares (aunque no hay un estándar sobre conceptos e implementación del ESB).
  - Existencia de tipos de servicios y APIs predefinidas y listas para su uso.
  - Convierte tareas de programación en configuración (manipulación de datos, por ejemplo).
  - Facilita la gestionabilidad del sistema, al proporcionar un punto único de control para todos los intercambios.
- **Inconvenientes**:
  - Posible punto único de fallo.
  - Fácil saturación del ESB a cargas altas de comunicación.
  - Sin una planificación correcta de APIs y conectores no evita la conexión lógica punto a punto entre clientes y servidores, sólo la física.
  - Requiere más sistemas en ejecución, para soportar el propio ESB.
  - Introducción de un elemento adicional en la cadena de procesamiento, con lo cual el rendimiento se puede ver afectado.
  - Escasas ventajas para entornos sencillos. Las ventajas se ven más en situaciones complejas, con muchos tipos de clientes y servidores.

Servicios de nombres y directorio



Servicios de nombres y directorio

- Los nombres se usan para referirse a una amplia variedad de recursos tales como ordenadores, servicios, objetos remotos, ficheros e incluso usuarios.
- Encargados de resolver la **transparencia con la ubicación**.

- Los **nombres** facilitan la comunicación y la compartición de recursos.
  - **Servicios de nombres ("páginas blancas")**: proporcionan a los clientes datos sobre objetos en el sistema distribuido a partir del nombre del objeto. Ejemplo: **Domain Name Service (DNS),** Servicio de nombres de CORBA**.**

- Además de los nombres, también se pueden utilizar **atributos** descriptivos (asociados a los nombres) para identificar los recursos del sistema distribuido.
  - Los clientes pueden no saber el nombre del recurso, pero conocen alguna información que lo describe. Incluso pueden no conocer las características del servicio pero no qué entidad lo implementa.
  - En un sistema distribuido, un atributo fundamental es la **dirección** del recurso.
  - **Servicios de directorio ("páginas amarillas")**: proporcionan a los clientes información sobre objetos en el sistema distribuido que cumplen cierta descripción. Ejemplo: Microsoft's Active Directory Services, **X.500** y **LDAP**.

Servicios de nombres

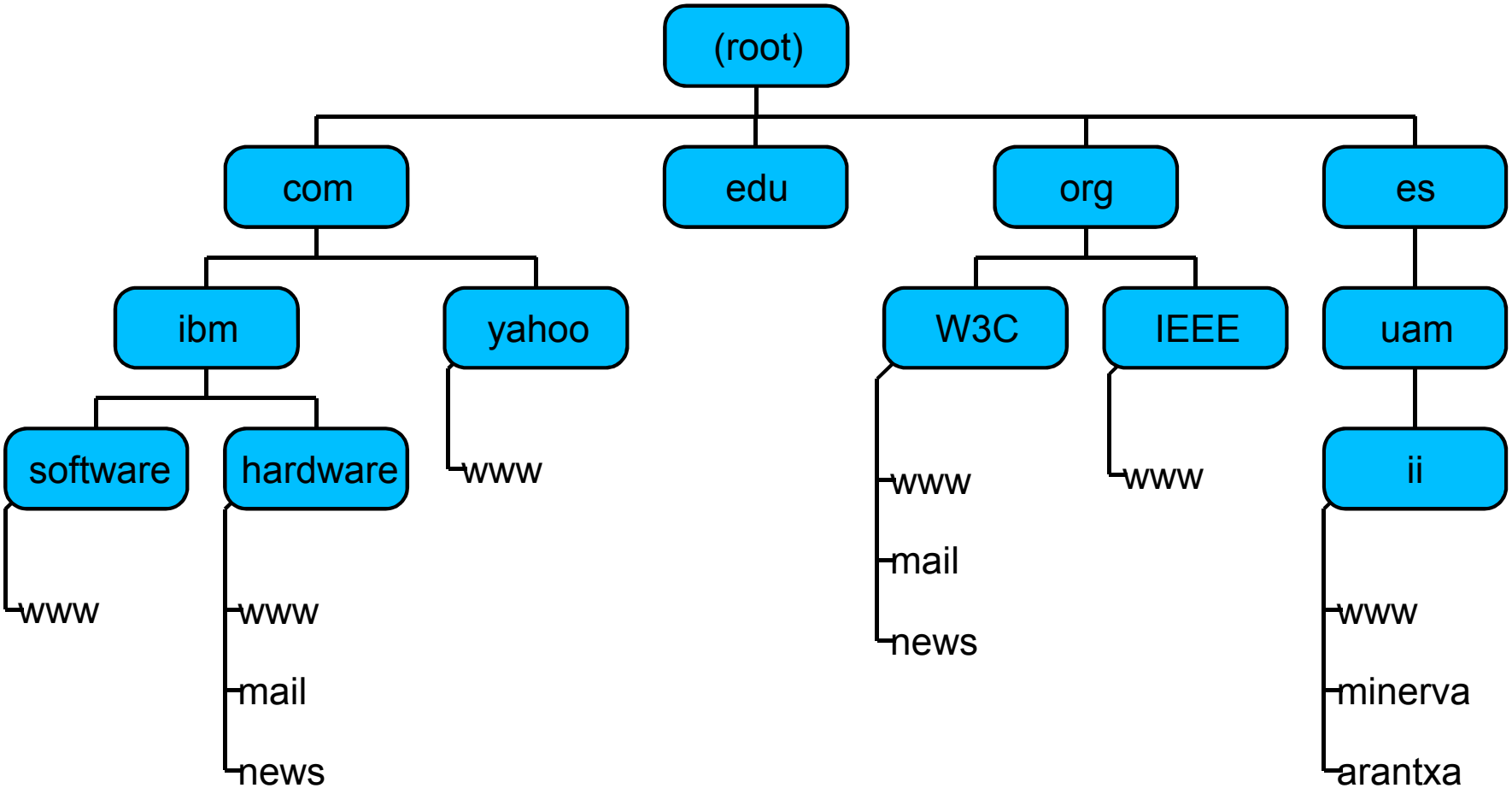
- Un **servicio de nombres** guarda información sobre una colección de nombres en forma de vínculos (*bindings*) entre nombres y atributos de las entidades del sistema distribuido.
- La principal tarea de un servicio de nombres es la **resolución de nombres**.
- Otras operaciones tales como crear nuevos *bindings*, borrar *bindings*, listar los nombres existentes, añadir y borrar contextos también son responsabilidad del servicio de nombres.
- **Espacio de nombres (*namespace*)**: Colección de todos los nombres válidos reconocidos por un servicio particular.
- **Dominio de nombres**: espacio de nombres para el que existe una única autoridad administrativa para asignar nombres.

Espacios de nombres *(namespaces)*

- Cada elemento del sistema se reconoce por un nombre.
  - Es el modo de localizarlo en el directorio.
- El nombre no debe contener información de localización física.
  - Garantizar **transparencia**. Localización se resuelve en el directorio.
- Los nombres deben ser únicos.
  - Asignación por una autoridad única dentro de su dominio.
- Estructura del espacio de nombres:
  - **Asignación de nombres planos** $\Rightarrow$ conflictos en dominios grandes.
  - **Asignación de nombres jerárquica**.
    - Descomposición del dominio en subdominios.
    - Existencia de una autoridad que asigna nombres por subdominio.
    - Establecimiento de una jerarquía de subdominios. Cada servidor de subdominio sólo conoce su directorio local, sus servidores descendientes y su predecesor en la jerarquía (o los nodos raíz).
    - Un nombre en la red global debe incluir todos los nombres de subdominios de la jerarquía a la que pertenece hasta la raíz.

Ejemplo espacio de nombres jerárquico: Internet

Sistema de nombres de dominio de Internet (DNS): base de datos distribuida y jerárquica (conjunto jerárquico de servidores DNS)

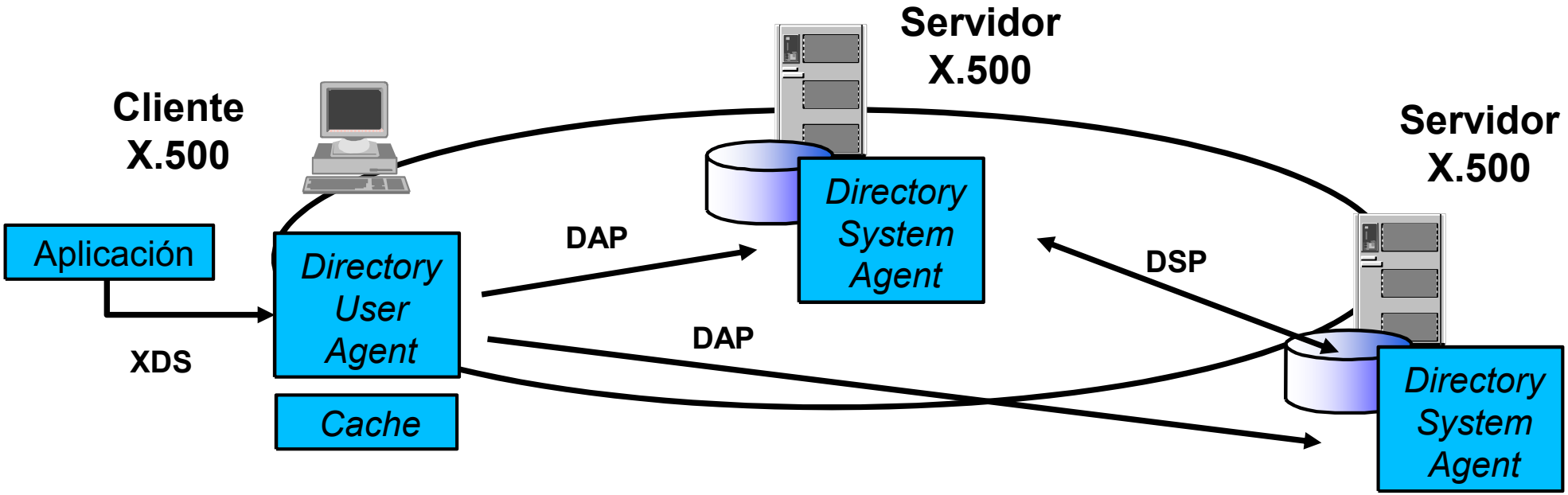


Servicios de directorio global

- Refleja la composición del sistema Cliente / Servidor en todo momento:
  - Sistemas que pertenecen a él (pertenencia dinámica).
  - Estado de los sistemas.
  - Aplicaciones Cliente y Servidor que contiene cada uno.
- Distintas implementaciones posibles:
  - Estructura plana.
  - Estructura jerárquica.
- Cada entrada contiene todos los datos asociados al elemento:
  - Dirección(es), ubicación física.
  - Estado.
  - Otras variables: uso, estadísticas...

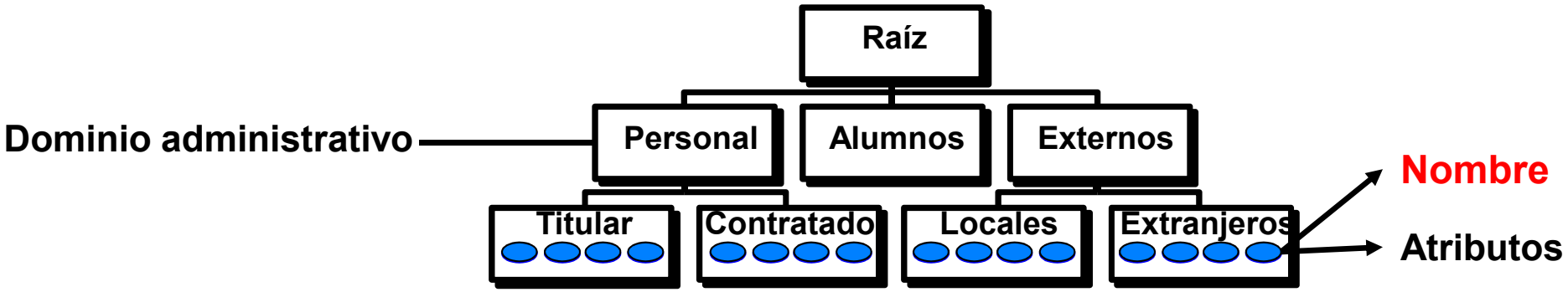
Estándar de directorios X.500

- Estándar ITU-T (antes CCITT). Servicio de propósito general de acceso a información de entidades (servicios, dispositivos hardware y software).
- Acceso al directorio mediante APIs ***X/Open Directory Service* (XDS)**
  - Añadir, eliminar, actualizar, leer, comparar, listar, búsqueda con atributos.
- Clientes contienen el ***Directory User Agent, DUA***. Comunica con servidores por *Directory Access Protocol*, DAP.
- Servidores contienen el ***Directory System Agent, DSA***. Comunicación entre DSAs mediante *Directory System Protocol*, DSP.



Directorio jerárquico

- **Ejemplo:** Directory Information Tree (DIT) de X.500
- **Distribuido**: Organizado en dominios administrados separadamente (por ejemplo, DSA)
  - Acceso dentro de un dominio sólo requiere nombre local.
- **Replicado**: Distintas copias para cada dominio.
  - Replicación inmediata o diferida.
- **Orientado a objetos**: Cada entrada se trata como una instancia de una clase de objeto.
- Puede permitir **consultas complejas** sobre los objetos.



Lightweigth Directory Access Protocol, LDAP

- Alternativa de acceso a directorio X.500 **más simple que DAP.**
  - No requiere stack de comunicaciones OSI. Usa TCP/IP.
  - Reemplaza la codificación ASN.1 de la información por otra textual.
  - No requiere un directorio X.500. Puede implementarse sobre otros modelos de directorio (Ejp: *Microsoft Active Directory*).
- Es un **protocolo de comunicaciones**
  - No una especificación de directorio.
    - En la actualidad la mayoría de los servidores LDAP implementan uno propio.
  - No es un API de programación.
    - Existe API de uso opcional para C.
    - El *Java Naming and Directory Interface, JNDI*, admite implementación sobre LDAP.
- **Ampliamente adoptado** por la industria (p. ej en servicios de directorio de intranet). Acceso seguro a datos.
- Versión actual: 3. Muchos productos implementan aún la 2.

Servicios de tiempo (fecha y hora)

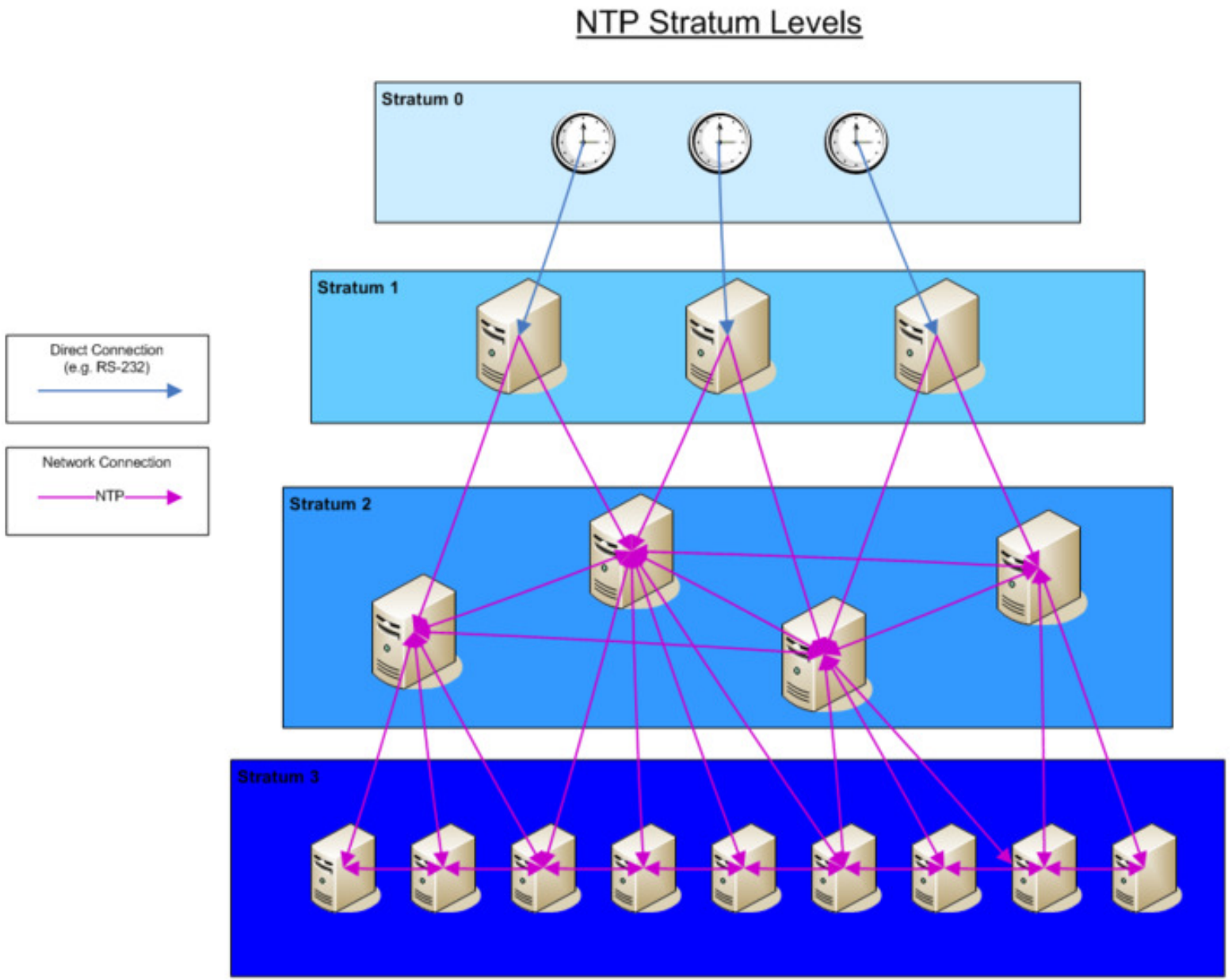
- Mantienen consistente la información de tiempo *(timestamp)* de todos los componentes de un sistema Cliente / Servidor.
- Sincronizan periódicamente cada ordenador de la red.
  - Servidores de tiempo en algunos ordenadores.
  - Agente en cada ordenador le realiza peticiones periódicas.
- Mantienen un componente de inexactitud en cada ordenador.
  - Cada agente conoce las desviaciones esperadas del sistema en que residen (**velocidad de deriva**)
  - Resincronizan cuando la desviación prevista sea mayor que un umbral determinado.
- Imprescindibles para mantener consistencia en la información de tiempo en todos los ordenadores de un sistema distribuido
  - Muy importante para seguridad.

Servidores de tiempo

- Uno o más servidores (de tiempo) deben estar conectados a un proveedor externo de información de tiempo *(timer ticks)*
  - Poseen dispositivo *Hardware* que se sincroniza por radio.
  - Actualizan reloj del administrador del sistema.
- Resto servidores realizan consultas para sincronizar sus relojes.
- Formato utilizado:
  - **Estándar UTC** (Tiempo Universal Coordinado), cuenta desde el principio del calendario Gregoriano.
    - Disponible a través de emisoras de radio que emiten señales horarias mediante satélites geoestacionarios y GPS.
  - Establecer la diferencia de zonas horarias: (Ejp.: Variable TZ)

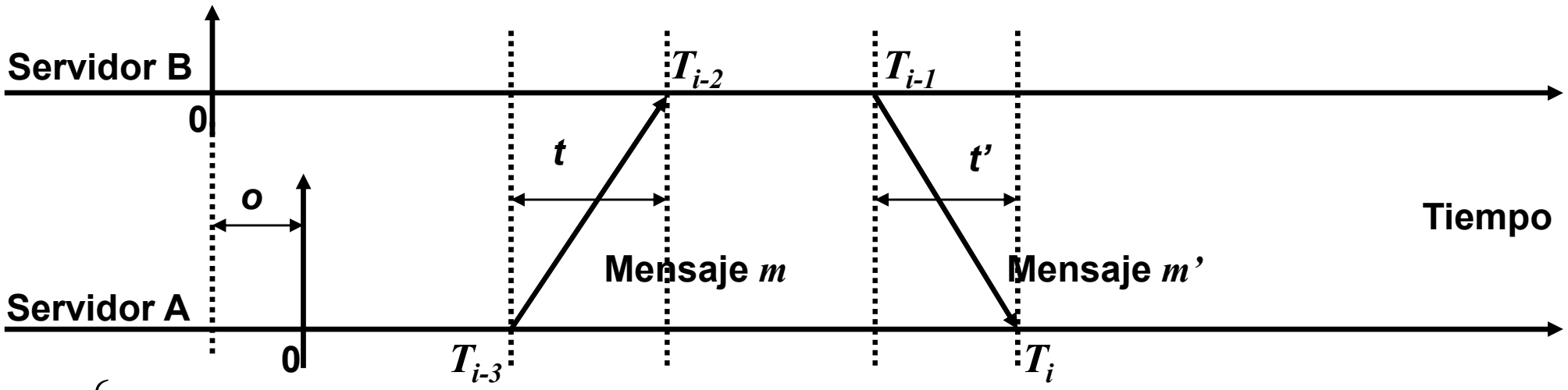
Network Time Protocol, NTP

- Define una arquitectura para un servicio de tiempo y un protocolo para distribuir la información del tiempo sobre Internet (UDP, puerto 123).
- **Objetivos**:
  - Sincronización a UTC de forma **precisa**, pese a retardos de comunicación por la red. Estadísticas para filtrado y calidad de los datos.
  - **Fiabilidad**: redundancia para evitar pérdidas largas de conexión. Reconfiguración de servidores.
  - **Resistente** a alto nivel de carga para permitir actualizaciones muy frecuentes.
  - **Protegido** contra interferencias externas. Autenticación de los orígenes de la información.
- Red de servidores con **estructura jerárquica** (subred de sincronización) cuyos niveles se denominan **estratos**.
  - Servidores primarios (estrato 1) conectados a fuente de tiempo recibiendo UTC.
  - Servidores secundarios (estrato 2) sincronizan con los primarios.
  - Servidores estratos inferiores reciben consultas de los clientes. Sincronizan con servidores estrato 2 y superior.



Fuente: http://en.wikipedia.org/wiki/Image:Ntp.png

Estimación del tiempo real utilizando NTP



$$\begin{cases} T_{i-2} = T_{i-3} + t + o \\ T_i = T_{i-1} + t' - o \end{cases}$$

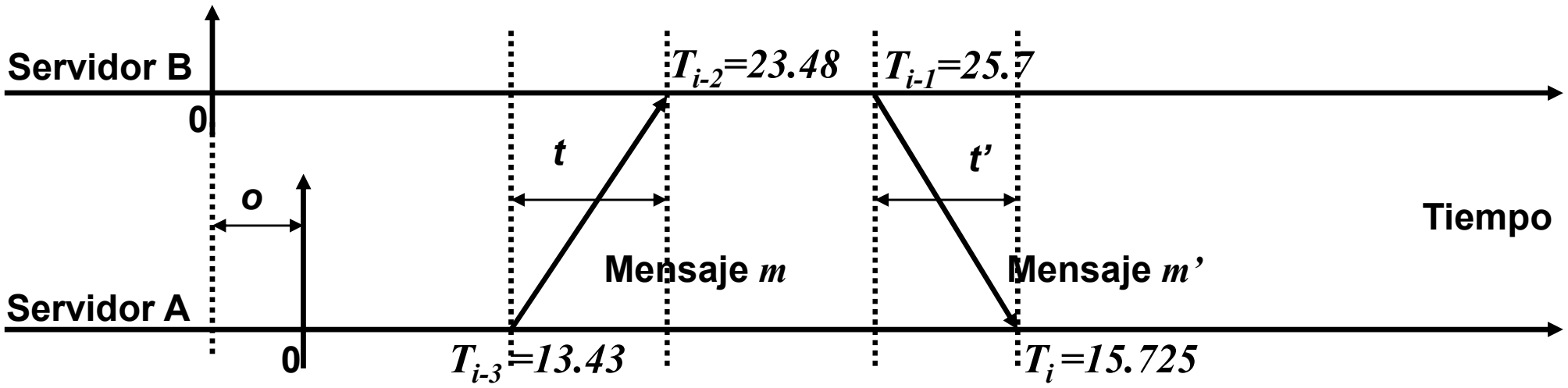
Restando : $o = \frac{(T_{i-2} - T_{i-3}) - (T_i - T_{i-1})}{2} + \frac{t' - t}{2} = o_i + \frac{t' - t}{2}$

Sumando : $t' + t = (T_{i-2} - T_{i-3}) + (T_i - T_{i-1}) = d_i$

Como $t, t' \geq 0 \Rightarrow t' + t \geq t' - t \Rightarrow o_i - d_i/2 \leq o \leq o_i + d_i/2$

o_i es una estimación de o, y d_i una medida de su precisión.

Ejercicio 16



$$o_i = \frac{\left(T_{i-2} - T_{i-3}\right) - \left(T_i - T_{i-1}\right)}{2} = \frac{\left(23.48 - 13.43\right) - \left(15.725 - 25.7\right)}{2} = 10.013s$$

$$\frac{d_i}{2} = \frac{\left(T_{i-2} - T_{i-3}\right) + \left(T_i - T_{i-1}\right)}{2} = \frac{\left(23.48 - 13.43\right) + \left(15.725 - 25.7\right)}{2} = 0.038s$$

Servicios de seguridad

- Los sistemas distribuidos plantean nuevos problemas de seguridad sobre los sistemas centrales:
  - **Confidencialidad** en la transmisión de datos en la red.
  - **Integridad** de la información transmitida.
  - **Autenticación** de los ordenadores y usuarios de la red.
    - Permitir acceso a los usuarios adecuados.
    - Garantizar que el ordenador es quien dice que es.
- El punto clave para el acceso a los servicios de seguridad es el Middleware, pero su implementación afecta a todos los niveles de un sistema distribuido.
  - Por su importancia y extensión se estudiarán en un capítulo independiente.

Bibliografía especial del tema

- COULOURIS, G., DOLLIMORE, J. y KINDBERG, T., *Sistemas distribuidos. Conceptos y diseño*, Addison-Wesley, 2001. 3ª ed.
- ORFALI, R., HARKEY, D. y EDWARDS, J., *The Essential Client/Server Survival Guide*, Willey, 1999. 3ª ed.
- RENAUD, P., *Introduction to Client / Server Systems: A Practical Guide for Systems Professionals,* John Wiley, 1996. 2ª ed.
- TANENBAUM, A., *Distributed Systems*, Prentice Hall, 2002.

Bibliografía especial del tema

- BLAKELEY, B., *Message & Queuing Using the MQI: Concepts, Analysis, Design & Implementation,* McGraw-Hill, 1995.
- **CERAMI, E., *Web Services*, O'Reilly, 2002. (disponlble on-line en la biblioteca)**
- Microsoft, *Component Object Model*, COM, http://www.microsoft.com/com/
- *Middleware Resource Center*: http://www.middleware.org/.
- OMG, *A Discussion on the Object Management Architecture,* OMG, 1997. (http://www.omg.org/library/oma/oma-all.pdf).
- OMG, *The Common Object Request Broker Architecture and Specification,* Object Management Group, 1997. Rev. 2.1. (ftp://ftp.omg.org/pub/docs/formal/97-09-01.pdf.gz).
- ORFALI, R., HARKEY, D. y EDWARDS, J., *Instant Corba*, Wiley, 1997.
- POPE, A.L., *The CORBA Reference Guide,* Addison-Wesley, 1998.
- **RICHARDSON, L. y RUBY, S., *RESTful Web Services*, O'Reilly, 2007.**
- SIEGEL, J., *CORBA: Fundamentals and Programming,* Wiley, 1996.
- **STEVENS, R., *Unix Network Programming*, Prentice-Hall, 1999. 2ª ed.**
- Sun Microsystems, *Java Remote Method Invocation*, http://java.sun.com/products/jdk/rmi/.
- TANENBAUM, A., *Computer Networks*, Prentice-Hall, 1996. 3ª ed.
- TANENBAUM, A., *Computer Networks*, Prentice-Hall, 2002. 4ª ed.
- An introduction to .Net remoting, (https://msdn.microsoft.com/en-us/library/ms973864.aspx)