Computación Distribuida

Traducción de las presentaciones infinitas de Presido y exámenes resueltos.

Mi recomendación personal es no estudiarse la materia; repite siempre todas las preguntas de exámenes de otros años excepto una o dos (que suelen ser cosas que se ven en prácticas)

Agradecimientos: @raquelvilas18

1. Paradigmas de la computación distribuida

- ✓ **Paradigma:** patrón, ejemplo o modelo.

Características aplicación distribuida:

1. **Comunicación entre procesos** (*participación de dos o más entidades independientes con capacidad de intercambiar datos entre ellos*)
2. **Sincronización de eventos** (*envío y recepción de datos sincronizado*)

Los paradigmas de computación distribuida son modelos que proporcionan la abstracción necesaria para aislar a los desarrolladores de los detalles de la comunicación y sincronización de procesos; permitiéndole centrarse en la propia aplicación.

Diferenciamos 8 grandes paradigmas en la computación distribuida:

1. Paradigma de envío de mensajes: paradigma más importante. El API de programación de sockets se basa en este paradigma.

- Operaciones básicas: envía y recibe
- Operaciones básicas comunicación orientada a conexión: envía, recibe, conecta, desconecta

2. Paradigma cliente-servidor: asigna roles asimétricos a los dos procesos que colaboran:

- **El servidor:** proveedor de servicio, espera pasivamente la llegada de peticiones.
- **El cliente:** envía peticiones específicas al servidor y espera su respuesta.

Los roles asimétricos hacen la sincronización de eventos sencilla (*servidor espera peticiones, cliente espera respuestas*). Este paradigma proporciona una abstracción eficiente para la provisión de servicios de red **centralizados.**

3. Paradigma de igual a igual (Peer to peer) arquitectura donde se intercambian de forma directa recursos y servicios entre ordenadores *(intercambio de información, ciclos de procesamiento, almacenamiento…).* Los ordenadores que tradicionalmente eran clientes se comunican de forma directa entre ellos jugando roles iguales, con las mismas capacidades y responsabilidades (*funcionan tanto como de servidor como de cliente*)

Este modelo es más adecuado para aplicaciones menos centralizadas (*mensajería instantánea)*

4. Paradigma del sistema de mensajes (Message-Oriented Middelware MOM)
Es una sofisticación del paradigma de paso de mensajes. En el, el servidor (**sistema de mensajes)** actúa como intermediario entre procesos independientes. Cuando un emisor quiere enviar un mensaje, lo deposita en este sistema de mensajes, que lo almacena en una cola de mensajes asociada al receptor concreto. De esta forma, los procesos intercambian mensajes de forma **asíncrona** y **desacoplada**.

Existen dos modelos para este paradigma:

- **Modelo punto a punto:** el middleware funciona como almacén de mensajes que permite desacoplamiento emisor-receptor: el sistema de mensajes reenvía el mensaje del emisor a la cola del receptor. Proporciona mayor abstracción para operaciones asíncronas, con el paradigma de envío de mensajes sería necesario el uso de hilos (*uno que reciba los mensajes y otro que los envíe para no bloquear todo el proceso esperando para recibir mensajes sin poder enviar*)
- **Modelo publica-suscribe:** en este modelo, los clientes interesados se *suscriben* a mensajes para un evento; y cada vez que este se produzca, el middleware distribuye el mensaje a los suscriptores (*publish)*

5. Remote procedure Call (RCP): proporciona abstracción para programar aplicaciones distribuidas de forma similar a las no-distribuidas. La comunicación se establece mediante llamadas a métodos que se invocan como si fueran locales aunque se vayan a ejecutar en otra máquina.

Se introduce a comienzo de los 80; existen dos APIs mayoritarias:

- *RPC de Sun Microsysrems (Open Network Computing Remore Procedure Call)*
- *DCE (Open Group DIstributed Compting Enviroment)*

Ambas proporcionan el método **rpcgen**: herramienta que transforma invocaciones remotas a llamadas a procedimientos locales del stub.

6. Paradigma de objetos Distribuidos: aplicar orientación a objetos en las aplicaciones distribuidas: las aplicaciones acceden a través de una red a objetos distribuidos (*que ofrecen métodos para el acceso a servicios*).

1. ***Remote Method Invocation (RMI)*** equivalente al RPC orientado a objetos. Un proceso invoca los métodos de un objeto que puede residir en otra máquina.

2. ***Network Services (servicios de red)***: extensión del RMI con un servicio de directorios global para la localización y acceso a los objetos (*los proveedores de servicios se registran en el servidor de directorio. Cuando un proceso desea un servicio, consulta en el servidor de directorio, y si el servicio está disponible le devuelve una referencia a dicho objeto para que pueda invocar sus métodos*)

3. ***Object Request Broker ORB (agente de solicitud de objectos)*** las aplicaciones envían peticiones a los Object Request Broker, que redirigen la petición al objeto adecuado. A diferencia del RMI, el ORB funciona como middleware que permite acceder a múltiples objetos que pueden ser heterogéneos (*diferente API o plataforma*)

4. ***Espacio de objetos:*** asume la existencia de entidades lógicas (espacios de objetos). Los participantes convergen en un espacio de objetos común; el suministrador coloca objetos como entidades dentro de este espacio de forma que los solicitantes pueden acceder a ellas. Se ocultan detalles de búsqueda de recursos u objetos necesarios en RMI, ORB y Network Services y asegura exclusión mutua porque cada objeto solo puede ser usado por un participante a la vez.

7. Paradigma de Agentes Móviles: se lanza un agente (programa transportable) desde un ordenador, este agente viaja de máquina en máquina siguiendo un itinerario; realizando las tareas y accediendo a recursos/servicios necesarios para cumplir su misión. En lugar de intercambiar mensajes, los datos se transportan dentro del objeto-agente que viaja entre las máquinas.

8. Paradigma de aplicaciones colaborativas (Groupware): un conjunto de procesos participan en una sesión colaborativa. Cada proceso puede hacer contribuciones enviando datos a todos/parte del grupo (**multicasting)** o puede leer y escribir datos en una pantalla compartida (**pizarras)**

Anotaciones:

Las **Tecnologías basadas en componentes** *(COM, DCOM, Java Bean, Enterprise Java Bean)* se basan en paradigmas de computación distribuida. Los componentes son paquetes de objetos especializados para interactuar entre ellos a través de interfaces estandarizadas.

Los **servidores de aplicaciones** (Java EE) son middleware para proporcionar acceso a objetos y componentes.

- ✓ **Middleware** software que actúa como intermediario entre procesos independientes.

2. El API de los Sockets

Api de los Sockets: interfaz de programación para la comunicación entre procesos (IPC: *Inter Process comunications*) originaria de UNIX que se ha extendido a todos los sistemas operativos modernos. Es el estándar de *facto* para la programación IPC y base de interfaces IPC más importantes (RPC, RMI)

Protocolo TCP/IP: protocolo para comunicaciones en redes introducido a principios de los 80; es el protocolo más extendido en Internet. Sigue un modelo en capas:

- ✓ ***Capa de red:*** proporciona mecanismo no fiable de comunicación entre sistemas. Emplea la dirección IP (dirección única de 32 bits). Existen 4 tipos de direcciones IP (A: comienza por 0; B: comienza por 10; C: comienza por 110; D: comienza por 1110)

**Los datos pueden almacenarse en memoria de dos maneras (Big Endian o Little Endian) pero a través de la red siempre se transmiten en big endian.*

- ✓ ***Capa de transporte (UDP y TCP):*** se encarga de la transferencia de datos libre de errores.

  Existen 2 alternativas:

  1. **UDP:** protocolo no orientado a conexión; no garantiza recepción correcta ni en orden de los datos. Se caracteriza por ser simple y rápido; solo añade el número de puerto y la verificación de contenido.
     Se emplea en aplicaciones que requieren alta tasa de transferencia y toleran pérdidas (*videoconferencias*). No existe QoS (*Quality of Service*) es decir no garantiza ancho de banda, pudiendo suponer retardos.

  2. **TCP:** protocolo orientado a conexión (*antes de enviar mensajes tiene que establecerse una conexión*). Garantiza una recepción correcta y en orden de los paquetes.

En la transferencia de datos intervienen una serie de factores:

- ✓ **Puerto**: entero de 16 bits que se emplea para identificar sin ambigüedad los procesos que intervienen en un dialogo.
- ✓ **Asociación**: datos de identificación del dialogo: protocolo (UDP/TCP) + la IP y puerto del emisor) + la IP y puerto del receptor.
- ✓ **Fragmentación**: las capas de red limitan el tamaño máximo del paquete (**MTU**) obligando a fragmentar la información. Esta fragmentación hace posible que paquetes de un mismo mensaje se reciban fuera de orden. Al enviar los paquetes se establece un número de orden del paquete que permita reconstruir el mensaje (**ensamblado**) y detectar fallos en la transmisión.

La transmisión y recepción de mensajes se hace a través de una memoria intermedia, que puede suponer retaros en el envío. Como alternativa existe el mecanismo **Out of Band (OOB),** que envía los paquetes por un canal independiente al principal; enviándose antes que el resto de los datos.

Funcionamiento IPC

(a partir de UNIX BSD 4.2 se implementa por llamadas al sistema)

Socket: mecanismo de comunicación que permite el envío y recepción de mensajes; que se ponen en la cola del socket transmisor hasta que el protocolo de red los transmita e introduzca en la cola del socket receptor.

Por ello, antes de comunicarse es necesario la creación del socket; mediante la llamada al sistema ***socket*** donde es necesario especificar protocolo (TCP/UDP) y dominio (UNIX si es local o Internet). Esta llamada devuelve **descriptor del socket**. A este socket se le da un identificador para que otros procesos lo utilicen como dirección de destino (mediante la llamada al sistema **bind)**

Comunicación por datagramas (UDP): Emplea sockets no orientados a conexión: múltiples procesos pueden enviar datagramas simultáneamente al receptor siendo el orden de recepción impredecible.

La comunicación se produce cuando se empareja un *sendto* y un *recvfrom*:

- ✓ **Sendto** (primitiva de envío de mensaje): necesario especificar mensaje, descriptor del socket propio e identificador del socket destino.
- ✓ **Recvfrom** (primitiva recepción) necesario especificar descriptor socket propio y buffer en el que se almacenará mensaje recibido.

En Java

Clases principales para la comunicación por datagramas: **DatagramSocket** (*representa el socket de conexión UDP) y* **DatagramPacket** (*representa un datagrama enviado/recibido por el socket*).

Los principales **métodos** en Java para la comunicación UDP son:

- ✓ **DatagramPacket(byte[] buf, int length):** *construcción del datagrama para recepción de mensajes.*
- ✓ **DatagramPacket(byte[] buf, int length, InetAdress adress, int port):** *construcción datagrama para envío de mensajes.*
- ✓ **DatagramSocket***(): instancia socket y lo asigna a cualquier puerto disponible (útil para enviar datos que no esperan respuesta)*
- ✓ **DatagramSocket(int port):** *instancia socket y lo asigna al puerto dado.*
- ✓ **DatagramSocket.close():** *cierra la conexión*
- ✓ **DatagramSocket.receive(DatagramPacket p):** *recibe en este socket y almacena mensaje en el datagrama p.*
- ✓ **DatagramSocket.send(DatagramPacket p):** *envía por este socket el datagrama p.*
- ✓ **DatagramSocket.setSoTimeOut(int timeout):** *establece un tiempo máximo de espera para recibir mensajes (limita el tiempo de bloqueo)*

Comunicación por datagramas (UDP) en Java

ENVIAR UDP

```
InetAdress receiver = InetAdress.getByName(ip);
DatagramSocket socket = new DatagramSocket();
String msg = "holiii";
Byte[] data = msg.getBytes();
DatagramPacket paquete = new datagramPacket(data, data.length, receiver, puerto);
Socket.send(paquete)
```

RECIBIR UDP

```
DatagramSocket ds = new DatagramSocket(puerto);
DatagramPacket dp = new DatagramPacket(buffer, MAXlenn)
Ds.receive(dp);
String s = new String(dp.getData(), 0, len);
System.out.println( dp.getAdress() +  s);
```

Comunicación encauzada/stream (TCP): comunicaciones orientadas a conexión. El socket en modo stream se crea para el intercambio de datos entre dos procesos específicos. No puede utilizarse para comunicarnos con más de un proceso.

1. Antes de poder enviar mensajes hay que establecer la conexión (*emparejar un **connect** y un **accept**)*
2. Una vez establecida la conexión la comunicación se hace por las llamadas : ***read-write*** y ***send-recv***
3. La conexión concluye al cerrar el socket: ***close***

En Java

Clases principales para la comunicación por datagramas: **ServerSocket** (*representa la base del socket para establecer la conexión sobre el)* **Socket** (*representa el socket con la conexión establecida*).

- ✓ **ServerSocket(int port) :** *crea el socket asociado al puerto*
- ✓ **Socket ServerSocket.accept():** *espera por una conexión y la acepta. Es una llamada bloqueante. Devuelve el socket correspondiente a la conexión establecida.*
- ✓ **ServerSocket.close()** *cierra el socket de conexión.*
- ✓ **ServerSocket.setSoTimeoit(int timeout)** *fija el tiempo máximo de espera para el método accept()*
- ✓ **Socket(Inet adress, int port)** *crea un socket stream asociado a la dirección y puerto dado.*
- ✓ **Socket.close()** *cierra el socket de dialogo concreto.*
- ✓ **Socket.getInputStream()** *: devuelve el objeto a través del que se reciben datos por el socket.*
- ✓ **Socket.getOutputStream():***devuelve el objeto a través del que se envían datos por el socket.*
- ✓ **Socket.setSoTimeout(int timeout) :** *tiempo máximo de espera del método read (del InputStream)*

SERVIDOR TCP

```
ServerSocket s = new ServerSocket(puerto)
Socket socket = s.accept()
DataInputStream input = new DataInputStream(socket.getInputStream());
DataOutputStream output = new DataOutputStream(socket.getOutputStream());
System.out.println(input.readByte());
Out.writeByte(5);
Socket.close();
s.close()
```

CLIENTE TCP

```
Socket socket = new Socket("nombre", puerto);
DataInputStream input = new DataInputStream(socket.getInputStream());
DataOutputStream output = new DataOutputStream(socket.getOutputStream());
Out.writeByte(5);
System.out.println(input.readByte());
Socket.close();
```

El puerto siempre se fija en la parte del servidor (el del cliente se genera automáticamente)

Tipos de sockets:

1. **Sockets modo Stream:** no orientados a conexión
2. **Sockets orientados a conexión**
3. **Sockets Seguros:** realizan cifrado de los datos transmitidos. En Java el paquete JSSE permite estas comunicaciones seguras (*cifrado de datos, autenticación del servidor, integridad mensajes..)*

Multidifusión

La multidifusión IP (*construida sobre el protocolo IP*) permite que un emisor transmita un único paquete IP a un conjunto de receptores (*grupo de multidifusión*) simultáneamente. Estos grupos se especifican con IPs de clase D (*empiezan por 1110)*

La pertenencia a estos grupos es dinámica; pertenecer a uno permite recibir los paquetes enviados al grupo; pero también se pueden enviar mensajes a un grupo sin pertenecer a él.

Puede emplearse en local o en Internet; en el caso de internet se emplea la multidifusión de los routers y puede limitarse fijando un **TTL** (número de routers máximo por el que puede pasar el paquete).

Es necesario disponer de una **dirección multidifusión libre para evitar conflictos**: en Internet las direcciones 224.0.0.1-224.0.0.255 están reservadas de forma permanente para multidifusión.

En otro caso para una comunicación local basta con limitar el TTL para evitar estos conflictos; pero a nivel de internet (*si no se usan las direcciones reservadas de forma permanente*) es necesario reservarla temporalmente: usando el **programa de directorio de sesiones (sd)** que permite detectar y unirse a sesiones multidifusión existentes o crear una, especificando su duración.

Hilos

Hilo: cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de manera independiente. Su estructura se almacena en el espacio de usuario y los cambios de contexto son menos costosos.

Dos maneras de creación de hilos:

- Implementar Runnable
- Implementar Thread

Java permite un método cerrojo para el control de secciones críticas: synchronised().

Métodos básicos

- ✓ Start() →prepara el hilo para ejecución en la JVM
- ✓ Run() →pasa a ejecución
- ✓ Yield() →cede la CPU y pasa a bloqueado
- ✓ Join() →espera a que un hilo termina; lanza un InterruptedException en el caso de qu ese haya interrumpido el hilo por el que se espera interrupted()
- ✓ Notify() →activa un hilo cualquiera de los que están bloqueados en la lista de espera
- ✓ NotifyAll()→ despierta a todos los procesos de la lista de espera
- ✓ Wait() → se llama dentro de una sección synchronised, espera el tiempo dado y abre el cerrojo mientras

3. Objetos distribuidos

El tradicional **paradigma de paso de mensajes** es un modelo natural que simula la comunicación humana; apropiado para servicios de red donde los procesos interactúan entre ellos a través del intercambio de mensajes. Esta abstracción no es escalable a aplicaciones de red complejas por:

1. **Participantes fuertemente acoplados** (*los procesos se comunican directamente entre ellos y, si la comunicación se pierde toda la colaboración falla*)
2. **Orientado a datos:** los mensajes contienen un formato mutuamente acordado, y cada uno desencadena una acción específica del proceso receptor; este sistema no es escalable a aplicaciones complejas con un gran número de peticiones y respuestas.

Paradigma de objetos distribuidos: proporciona una abstracción mayor que el paso de mensajes. Los recursos de la red se presentan como objetos distribuidos y, para solicitar un servicio, un proceso invoca los métodos del objeto distribuido necesario, que se ejecuta en la máquina remota y envía la salida del método como la respuesta al proceso solicitante.

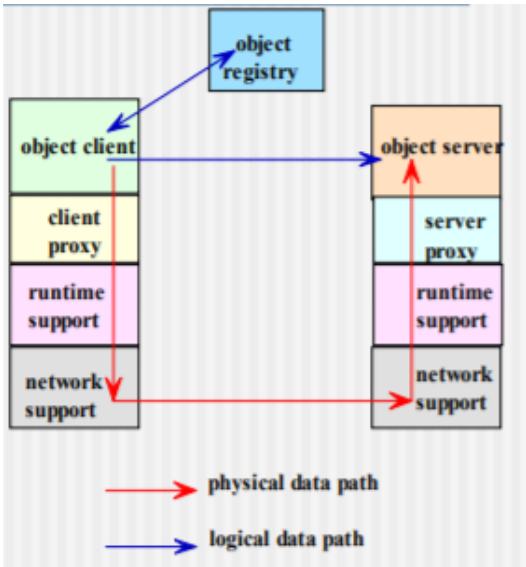
Objeto distribuido: objeto cuyos métodos pueden invocarse por un proceso remoto (*es decir, que un proceso que se está ejecutando en un equipo remoto pueda invocar los métodos de este objeto conectándose a través de la red)*

Es un paradigma orientado a acciones: hace hincapié en la invocación de operaciones, dejando los datos en un puesto secundario. Es un paradigma menos intuitivo para humanos pero más natural para desarrollo de software orientado a objetos.

Arquitectura de objetos distribuidos

Elementos de la arquitectura:

1. **Objeto servidor:** objeto distribuido que proporciona los métodos.
2. **Registro de objetos:** almacena las referencias a los objetos distribuidos (*información que permite localizar al objeto remoto*)
3. **Proxy cliente:** interactúa con el software encargándose del envío de los datos.
4. **Soporte en tiempo de ejecución *(runtime support)*:** responsable de la comunicación entre procesos. En el envío de datos (llamada al método / envío salida del método) se encarga encapsular los argumentos para que el **soporte de red *(network suport)*** los envíe a la maquina remota. En la recepción de datos se encarga de desempaquetar el mensaje y redirigir la petición al proxy del servidor.
5. **Proxy servidor:** invoca la llamada al método local pasándole los argumentos desempaquetados.
6. **La salida de la ejecución del método** es empaquetado y enviado por el proxy servidor al proxy cliente a través del soporte en tiempo de ejecución y soporte de red.



La abstracción propia del paradigma hace pensar que el cliente realiza directamente la llamada al método del objeto remoto. (logical data path).

Mientras que la información de la que se abstrae es como el proxy toma la petición, la encapsula en un mensaje y la envía por la conexión establecida con el servidor del objeto remoto; que tiene que desempaquetarlo y redirigirlo al objeto y viceversa con la salida del método.

Herramientas más conocidas: basadas en objetos distribuidos

- ✓ *Java Remote Method Invocation* (**RMI**)
- ✓ *Common Object Request Broker Architecture* (**CORBA**)
- ✓ *Distributed Component Object Model* (**DCOM**)
- ✓ *Simple Object Acces Protocol* (**SOAP**)

Remote Method Invocation (RMI) implementación orientada a objetos del modelo RPC, exclusiva para Java. Elementos principales:

- ✓ Objeto Remoto se declara para el cliente como una **Interfaz remota** (interfaz que hereda de Java **Remote** y permite ser implementada usando sintaxis RMI. Todos sus métodos deben especificar excepción **RemoteException** que trata todos los errores de comunicación: *fallo acceso, de conexión, no encontrar el objeto, el stub, el skeleton…)*

Objeto servidor implementa dicha interfaz. (*debe implementar todos los métodos declarados en la interfaz remota, normalmente en una clase a parte*) y registra en el servicio de directorio el objeto con la implementación.

Normalmente se divide en dos clases; donde el **servidor de objetos** es el que instancia y registra al objeto que implementa la interfaz. El **servidor de objetos es concurrente:** *cada solicitud de un objeto se procesa a través de un hilo independiente del servidor. Importante una implementación del objeto remoto thread-safe*

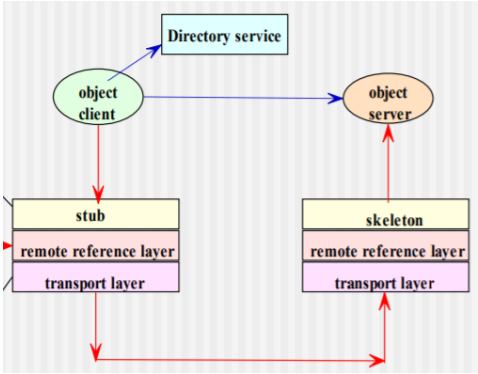
- ✓ Objeto cliente **accede al objeto invocando los métodos de la *interfaz remota*** que se implementan en el objeto servidor.
- ✓ **Stub:** (proxy del cliente) y **Skeleton (** proxy del servidor). Se crean con el siguiente comando:
  - **Rmic ImplementacionMetodos** (genera ImplementacionMetodos_skel.class y ImplementacionMetodos_stub.class)

- ✓ **Servicio de directorios:** en RMI existen diferentes servicios de directorios: entre ellos **rmiregistry** es un servicio propio de RMI cuyo servidor se ejecuta en la máquina del servidor de objetos.

JNDI es una alternativa como servicio de directorios más general que rmiregistry (puede ser usado por aplicaciones que no usan RMI)

Java RMI se compone de 3 capas de abstracción:

1. **Proxys**: stub y skeleton (comunicación entre aplicaciones y métodos remotos)
2. **Capa de referencia remota**: realiza protocolos de referencia remota, mapeando los métodos del proxy a la capa de transporte y viceversa.
3. **Capa de transporte:** gestión de conexión y envío de los datos



Aplicación RMI

Colocación de ficheros:

Archivos en el cliente

- Cliente.class
- InterfazMetodos.class
- ImplementacionMetodos_stub.class

Archivos en el servidor

- Servidor.class
- InterfazMetodos.class
- ImplementacionMetodos.class
- ImplementacionMetodos_skel.class

Pasos servidor:

1. Activar servidor de registro (RMIRegistry):
   a. de forma dinámica: LocateRegistry.createRegistry(puerto) puerto 1099 por defecto.
   b. Manualmente: comando rmiregistry <numPuerto>
2. Crear objeto que implementa los métodos.
   a. InterfazMetodos = (InterfazMetodos) new ImplementacionMetodos();
3. Registrar el objeto distribuido en el RMIRegistry:
   Naming: proporciona métodos para almacenar y obtener referencias del registro. En concreto para registrar el objeto ofrece 2 métodos:
   a. **Rebind**: *almacenar en el registro una referencia al objeto distribuido. sobrescribe cualquier referencia registrada con el mismo nombre.*
   b. **bind** *solo escribe el registro si no existe ninguna referencia registrada con el nombre dado.*

```
registryURL = "rmi://localhost:"+portNum+"/nombre";
Naming.rebind(registryURL, exportedObj)
```

Pasos cliente:

1. Localizar registro RMI en el nodo servidor

2. Buscar referencia remota del objeto y castearla a la clase Interfaz.
    a. Naming.**lookup** devuelve la referencia del objeto si existe en el registro.

```
registryURL = “rmi://localhost:”+portNum+”/nombre”;
InterfazMetodos h = (InterfazMetodos)Naming.lookup(registryURL)
```

RMI vs API de sockets:

La API RMI herramienta eficiente para construir aplicaciones de red, y se puede usar substituyendo el API de sockets pero:

- API de sockets es más cercano al sistema operativo (*menos sobrecarga de ejecución*) por lo que es mejor opción para aplicaciones que requieren alto rendimiento.
- API RMI mayor abstracción que facilita desarrollo software, por lo que se generan programas más comprensibles y fáciles de depurar.

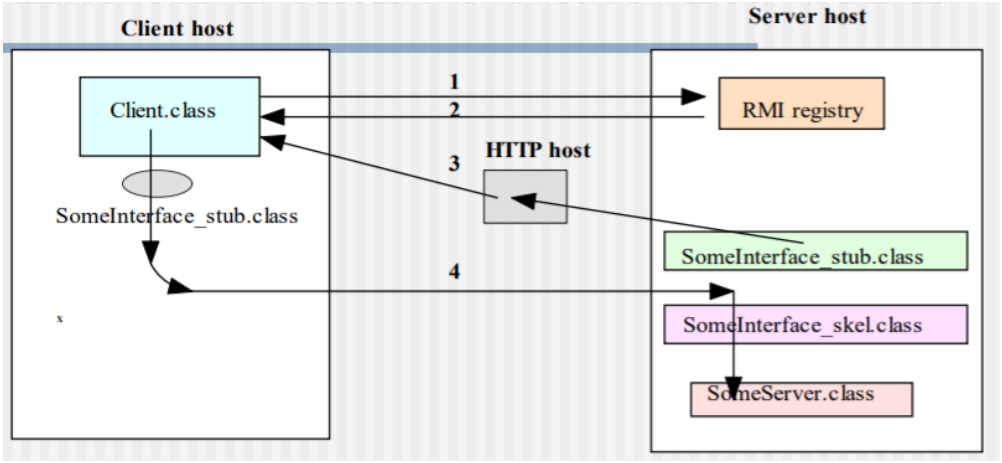
4. RMI Avanzado

El API de Java RMI ofrece múltiples funcionalidades; entre ellas destacan:

1. *Stub downloading*

En RMI básico, cada vez que cambiabas la implementación de los métodos del objeto distribuido, actualizar al cliente (*pasándole el stub actualizado*).

Para que los cambios realizados no afecten al cliente, interesa que estos puedan obtener dinámicamente el stub. Para ello, se coloca el stub en un servidor web y se descarga mediante el protocolo HTTP.



La descarga de archivos de objetos ajenos al cliente hace necesarias **ciertas medidas de seguridad**: debe realizarse una instancia de **Java Security Manager** *(objeto que define las políticas de seguridad para Java***)** tanto en el cliente como en el servidor.

El gestor de seguridad de RMI no permite el acceso a la red: es necesario modificar el archivo **java.policy** para dar permisos de acceso, y especificarlo al activar el cliente:

```
java –Djava.security.policy = java.policy Cliente
```

2. *RMI Security Manager*

RMI involucra acceso y posiblemente la descarga de objetos desde/a una maquina remota, por ello es importante protegerse de posibles accesos inadecuados. RMI Security Manager es una clase Java que puede ser instanciada (*tanto en el cliente como en el servidor*) para limitar los privilegios de acceso.

Es posible crear nuestro propio gestor de seguridad:

```
System.setSecurityManager(new RMISercurityManager());
```

3. *RMI Callbacks*

Modelo cliente servidor → servidor pasivo (*nunca inicia él la comunicación*). Algunas aplicaciones necesitan que sea el servidor el que inicie la comunicación; por ejemplo si están esperando a que

una condición se cumpla y solo el cliente puede iniciar la comunicación; este tendría que estar llamando todo el rato a un método del servidor para ver si se cumple o no dicha condición (**polling o sondeo**).

Con callbacks, el cliente indicaría el evento al que está esperando y, cuando ocurra es el servidor el que se lo comunica al cliente *(sin necesidad de comprobaciones infinitas por parte del cliente).*

Si el servidor también puede iniciar comunicación (*comunicación bidireccional*) implica que el cliente debe ofrecer una interfaz de métodos remotos que el servidor pueda invocar.

Ficheros de una aplicación Callback

Archivos en el cliente

- Cliente.class
- InterfazServidor.class
- ImplementacionServidor_stub.class
- InterfazCliente.class
- ImplementacionCliente.class
- ImplementacionCliente_skel.class

Archivos en el servidor

- Servidor.class
- InterfazServidor.class
- ImplementacionServidor.class
- ImplementacionServidor_skel.class
- InterfazCliente.class
- ImplementacionCliente_stub.class

En esta arquitectura, además de los ficheros del RMI básico se incluye como objeto distribuido el cliente también, con sus correspondientes clases: **interfaz remota** (*InterfazCliente.class*), **implementación** de la interfaz (*ImplementaciónCliente.class*) y **proxys** correspondientes (*ImplementaciónCliente_skel.class* e *ImplementaciónCliente_stub.class*)

4. Serialización y envío de objetos

En RMI básico solo es posible el envío de datos primitivos (*Strings, int, float*..) pero puede ser necesario pasar objetos complejos. Para ello es necesario "**serializarlos**", es decir, encapsular su contenido *(código+datos)* en una cadena de caracteres (*carácter = byte*) susceptible de ser enviada a través de la red.

Para que un objeto sea serializable, tiene que implementar la clase serializable de Java, y todos sus atributos tienen que ser o bien primitivos o también serializables. La máquina virtual de Java nos garantiza la reconstrucción correcta del objeto.

5. The Common Object Broker Architecture (CORBA)

Corba: arquitectura estándar para sistemas de objetos distribuidos; diseñada para permitir que objetos distribuidos interoperen sobre **entornos heterogéneos** (*los objetos pueden estar implementados en diferentes lenguajes o desplegados sobre diferentes plataformas*)

- RMI middleware propietario desarrollado por Sun MicroSystems que solo soporta objetos escritos en Java.
- CORBA es una arquitectura desarrollada por el consorcio industrial Object Management Group (OMG)

CORBA no es una herramienta, es un conjunto de protocolos. Una herramienta es **CORBA compliant** (*compatible con CORBA*) si da soporte a dichos protocolos: los objetos que se desarrollen sobre ella podrán interoperar con otros objetos CORBA-compatibles.

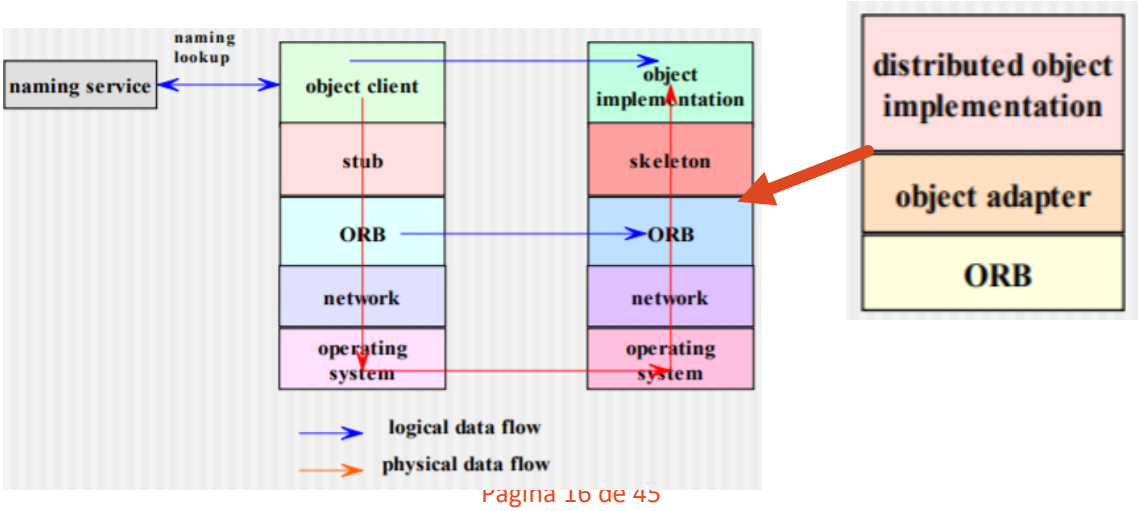
Arquitectura básica

Al poder comunicarse objetos heterogéneos; las interfaces no se pueden escribir en lenguajes concretos**. CORBA IDL** (*CORBA Interface Definition Language*) es un lenguaje universal con sintaxis específica que actúa como la interfaz remota RMI pero de forma independiente al lenguaje. Para cada lenguaje CORBA-compatible existe una traducción o mapeo de los datos IDL.

ORB: middleware que permite a los objetos realizar llamadas a métodos situados en máquinas remotas a través de la red. Su misión es identificar el objeto al que tiene que invocar, traducir la referencia al lenguaje concreto e invocar el método. El ORB es el punto de encuentro entre IDL y el lenguaje específico del objeto.

Adaptador de objetos: Con el avance de CORBA se añade este componente al skeleton: el **Adaptador de objetos** (Object Adapter) simplifica responsabilidades del ORB, de forma que el ORB únicamente tiene que encontrar el Object adapter correspondiente al objeto de la petición; y este **object adapter** se encarga de el mapeo de la referencia a la implementación concreta y de la invocación del método dado.

Es decir, ahora el ORB se limita a identificar el objeto concreto al que dirigir la petición. La traducción e invocación de métodos se delega en el Object Adapter.



Portable Object Adapter: es un tipo de adaptador de objetos definido por CORBA. Permite que una implementación de objeto funcione en varios ORB, haciendo la implementación portable a través de varias plataformas. (*Algo así como que no haya una única instancia de la implementación para la gestión de las peticiones; sino que pueda haber varias instancias simultáneas funcionando en diferentes ORB que hacen peticiones al mismo objeto)*

Protocolos comunicación

GIOP (*General Inter-ORB Protocol*) protocolo que proporciona un marco general para que se construyan protocolos interoperables por encima de un nivel de transporte específico. Protocolo abstracto por el cual dos ORBs se comunican. *Digamos que es como la Interfaz del protocolo; que debe ser implementada por un protocolo concreto sobre una capa de transporte concreta*

IOP (*Internet Inter-ORB Protocol*) GIOP aplicado sobre el nivel de transporte TCP/IP. Es el protocolo por el cual dos ORBS se comunican a través de internet. Elementos:

1. **Requisitos de gestión de transporte** (especifican qué se necesita para conectarse y desconectarse, y papel del cliente y del servidor en establecer y liberar conexiones)
2. **Definición de la representación común de datos:** definir esquema de codificación para empaquetar/desempaquetar los datos para cada tipo de dato IDL.
3. **Formato de los mensajes:** definir los diferentes formatos de os distintos tipos de mensajes.

***I**nternet se ve como un **bus de objetos** CORBA interconectados.*

- ✓ **Referencia a objeto CORBA:** entidad abstracta traducida a una referencia de un objeto específico de cada lenguaje por medio del ORB.

Interoperable Object Reference (IOR): protocolo para referencias abstractas de objetos. Un ORB compatible con IOR permite obtener y registrar referencias a objetos desde un servicio de directorio compatible con IOR.

Las referencias a objetos CORBA en este protocolo se denominan también **IOR** (*Interoperable Object References*). Esta referencia es una cadena de caracteres que incluye información de el tipo de objeto, IP y puerto del servidor del objeto y una clave de objetos (identificador del objeto, empleado por el servicio de directorio para localizar el objeto internamente)

Servicios CORBA

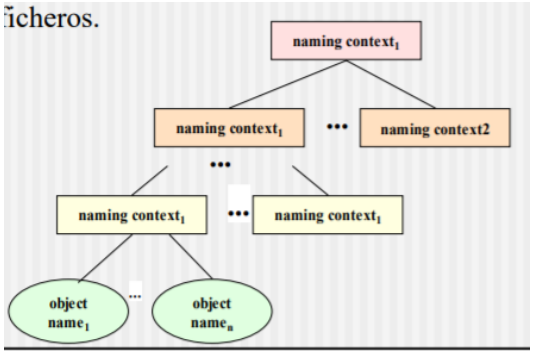
CORBA especifica una serie de servicios para construir aplicaciones, cada servicio se defina por medio de un IDL estándar (este puede ser implementado por los desarrolladores) y sus métodos se pueden invocar desde cualquier cliente CORBA. Entre estos servicios están Naming Service, Concurrency Service (*control concurrencia*), Event Service (*sincronización eventos*), Security Service (*gestión de seguridad..).*

1. Naming Service (servicio de nombres de CORBA) análogo al registro RMI pero de manera independiente de la plataforma y lenguaje de programación. Su API está especificada en una interfaz IDL. Ofrece 2 funciones básicas para los clientes basados en ORB:

- **Bind** asocia/registra un objeto con un nombre simbólico.

- **Resolve**: obtiene/busca una referencia a un objeto.

Al ser un espacio de nombrado global, la organización de archivos en el servicio de nombres CORBA es más complejo: este sigue una estructura jerárquica.



Naming context (*contexto de nombrado*) cada uno de los nodos intermedios (directorios) en la jerarquía. Los nodos finales (archivos) se corresponden con el nombre de los objetos registrados.

Nombre compuesto: nombre completo de un objeto que incluye todos los contextos de nombrado asociados. Similar al path de un fichero que indica todos los directorios padre

Servicio de Nombres Interoperable (INS) sistema de nombrado para el servicio de Nombres de CORBA que está basado en el formato URL.

Nombre compuesto en INS:
corbaname::<hostname>:<puerto>#context1/context2/context3/objectName1

Java IDL

Además de varias herramientas compatibles con CORBA (como RMI sobre IIOP) ofrece un conjunto de herramientas para desarrollar aplicaciones CORBA:

- ✓ **Idlj** : compiladr IDL-a-Java
- ✓ **Orbd:** proceso servidor que da soporte al Servicio de Nombrado y otros servicios CORBA
- ✓ **Servertool:** interfaz en línea de comandos para registrar/desregistrar objetos y arrancar/parar servidores.
- ✓ **Tnameserv** : versión desfasada del Servicio de nombrado

Compilar el fichero IDL:

Idlj –fall Hello.idl genera los ficheros:

- **HelloOperations.java** → interfaz de operaciones Java *(traducción a interfaz Java del IDL CORBA)*
- **Hello.java** → fichero de firma de interfaz, extiende la interfaz HelloOperations.java y las clases estándar de CORBA (Interfaz que agrupa métodos CORBA con los métodos específicos del objeto)
- **HelloHelper.java** → funcionalidades auxiliares necesarias para dar soporte a los objetos CORBA dentro de Java. En concreto, el método **narrow** transforma una referencia a objeto CORBA en su correspondiente objeto Java.
- **HelloHolder.java** →contiene la referencia al objeto que implementa la interfaz Hello.java, proyecta los parámetros del tipo out (salida) e inout (entrada).
- **_HelloStub.java** → proxy del cliente, extiende la clase org.omg.CORBA.portable.ObjectImpl e implementa la clase Hello.java.

- **HelloPOA.java** → *skeleton + adaptador de objetos POA*, extiende la clase org.omg.PortableServer.Servant e implementa las interfaces HelloOperations e InvokeHandler. Es la base para la implementación del **Servant**.

Servidor

Se compone de dos 2 clases

- **Servant (Objeto servidor)** : *(HelloImpl)*, implementación de métodos de la interfaz IDL Hello.
- **Servidor de objetos:** programa principal que se encarga de:
  1. inicializar el ORB
     ```
     ORB orb = ORB.init(args, null)
     ```
  2. Buscar e iniciar el POA
     ```
     POA rootpoa = (POA) orb.resolve_initial_references("rootPOA");
     rootpoa.the_POAManager().activate();
     ```
  3. Crear objetos servant
     ```
     helloImpl = new HelloImpl();
     helloImpl.setORB(orb);
     ```
  4. Obtener referencia del objeto servant
     ```
     org.omg.CORBA ref = rootpoa.sevant_to_reference(helloImpl);
     Hello refObj = HelloHelper.narrow(objPoa); //traduce a obj java
     ```
  5. Obtener referencia al servicio de nombres (con INS es necesario Naming Context)
     ```
     Org.og.CORBA.Object objRef = orb.resolve_iniciatl_references("NameService");
     NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
     ```
  6. Registrar objeto en el servicio de nombres
     ```
     NameComponent path[] = ncRef.to_name("nombreServicio");
     ncRef.rebind(path, refObj)
     ```
  7. Pasarle el control al ORB
     ```
     orb.run()
     ```

Cliente

Puede ser una alicación Java, un applet o un servlet que se encarga de:

1. inicializar el ORB
   ```
   ORB orb = ORB.init(args, null)
   ```
2. obtener referencia al Servicio de Nombres
   ```
   Org.og.CORBA.Object objRef = orb.resolve_iniciatl_references("NameService");
   NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
   ```
3. obtener referencia a la implementación del objeto y ejecutar métodos
   ```
   HelloImpl = HelloHelper.narrow(ncRef.resolve_str("nombreServicio");
   helloImpl.sayHello();
   ```
4. desactivar el servicio
   ```
   HelloImpl.shutdown();
   ```

NOTA: CORBA avanzado no entra en el examen

7. Redes P2P

Cliente-Servidor vs PeerToPeer

La tradicional arquitectura Cliente-Servidor es el paradigma posiblemente más conocido, de confianza y de gran potencia donde los clientes solicitan datos a un servidor. Es un modelo que cuenta con gran éxito (Http, FTp, servicios Web…) pero cuenta con ciertas limitaciones:

- Escalabilidad compleja
- Presenta un punto único de fallo (si se cae el servidor, cae todo el servicio)
- Requiere administración

La arquitectura Peer to peer trata estas limitaciones. La computación P2P se basa en la **compartición de recursos y servicio**s (información, ciclos de procesamiento, almacenamiento de disco..) mediante el intercambio directo entre sistemas (*que tradicionalmente eran clientes*)

Todos los nodos son a la vez clientes y servidores (proveen servicios e invocan servicios de otros). Y los datos no están centralizados.

Características

1. Cada nodo aporta contenido, almacenamiento, memoria y capacidad de procesamiento (CPU).
2. Los nodos son autónomos
3. La red es dinámica: los nodos entran y salen de forma frecuente.
4. Los nodos colaboran directamente entre ellos (sin necesidad de intermediarios)
5. Los nodos presentan capacidades diversas

Ventajas

- ✓ **Uso de recursos eficiente** (emplea todo el ancho de banda, almacenamiento y potencia de la red*. Ya que ni toda la información se dirige/procesa al servidor; ni las comunicaciones siempre van hacia él; sino que emplean los diferentes canales de la red al ser una comunicación directa entre nodos*)
- ✓ **Escalabilidad:** los recursos y la red crecen de forma natural con su utilización (*cada nuevo miembro supone también más recursos para la arquitectura*)
- ✓ **Mayor robustez**: los datos no están centralizados, sino que existen diversas réplicas de datos en computadores diversos (*se elimina el punto único de fallo*)
- ✓ **Administración sencilla:** los nodos se auto organizan; sin necesidad de desplegar servidores ni administración centralizada de las peticiones.

Aplicaciones P2P

1. Compartición de archivos: compartición de archivos a gran escala; cada nodo puede a la vez ofrecer sus archivos y descargar archivos que otros nodos ofrezcan de forma directa del nodo remoto. (*Puede conllevar problemas de copyright*)

- *Napster*: compartición de música. Cada nodo sube su lista de música y consultar al servidor central por una canción; este le responde con la ip de los usuarios que tenga dicho archivo para que se conecte y descargue el archivo. El servidor central supone un cuello de botella para la escalabilidad, y un punto único de fallo.
  Se caracteriza por **Búsqueda centralizada** y descarga **P2P**.

- *Gnutella*: compartición de cualquier archivo. Se caracteriza porque **la búsqueda es descentralizada**: preguntas a tus contactos por los archivos en los que estás interesado; estos a su vez preguntan a sus contactos. Elimina el punto único de fallo pero no asegura resultados correctos en las búsquedas. Presenta también el problema de **inundación de la red**; por la propagación de busquedas que emplea; si se producen muchas búsquedas simultáneamente se saturará la red.

- *Kazaa*: hibrido entre Napster y Gnutella. Se escoge entre los usuarios de forma automática los **super-peers** (*basándose en su disponibilidad, almacenamiento, ancho de banda…*). Cada super-peer funciona como un servidor Napster para una pequeña porción de la red (*almacena archivos disponibles y las IPs asociadas a os mismos*). Periódicamente todos los super-peers intercambian su información para tener datos de toda la red.

- *Freenet*: ninguna de las alternativas anteriores proporcionaba anonimato. Freenet proporciona anonimato mediante el siguiente protocolo: *cuando un nodo busca un archivo, le hace la petición a un único nodo; este, si tiene el archivo responde, sino, escoge otro nodo de la red y le hace la misma petición y así simultáneamente.* Ningún nodo sabe si la petición que le llega es del que hizo la búsqueda o simplemente una redirección.

*Todas ellas dependen de que los usuarios compartan datos: se conoce como **problema del polizón** o **consumidor parásito (free riding)** los usuarios que consumen información pero no proveen ninguna útil. ´*

Tipos de redes P2P

Redes P2P no estructuradas: enlaces arbitrarios, si un usuario desea encontrar información específica la petición tiene que recorrer toda la red.

Redes P2P estructuradas (segunda generación): los pares se conectan entre si mediante algún diseño que simplifique las búsquedas. Mantienen una **tabla de hash distribuida (DHT)** de forma que si un usuario busca ciertos datos, utilizará el protocolo global para determinar los usuarios que tienen la información.

La administración de la tabla hash es una responsabilidad distribuida entre varios nodos (*por ejemplo si fuesen 5 nodos con una tabla de 100 entradas, cada uno almacenaría 20 entradas*). Estas redes ofrecen:

- Autoorganización
- Carga equilibrada
- Tolerancia a fallos

Las tablas Hash distribuidas almacenan datos del tipo clave-valor donde la clave es el nombre del archivo y el valor el contenido del archivo. Permiten encaminar eficientemente mensajes al dueño de una clave determinada (*es decir, obtener el archivo de un nodo específico de forma directa, sin pasar por redirecciones infinitas*)

API Chord: algoritmo para la implementación de tablas hash distribuidas.

- Cada nodo se identifica con una secuencia de n bits.
- Cada nodo puede tener múltiples claves; estas claves también se identifican con una secuencia de n bits.

La distribución de las claves entre los nodos sigue la norma de **que el código de la clave nunca puede ser mayor al del nodo**. Por ejemplo el nodo 34 no puede tener la clave identificada con un 35.

El protocolo Chord se encarga de asignar dinámicamente claves a los nodos activos *(se obtiene calculando el hash de la IP)* **y a las claves activas (***hash de su contenido*). Una clave de id 5 se asignará al nodo 5 si está disponible; sino al nodo con el id automáticamente superior a 5. De esta forma, si conocemos los nodos 20 y 34, sabemos que la clave con id 15 iría asociado al nodo 20 (es más cercano a 15) pero la clave con id 21 iría asociado el 34.

Cada nodo almacena información de **O(log(TotalNodos))** nodos. *Al realizar una búsqueda de la clave k, el nodo examina su tabla de rutas, si no está, pregunta al nodo con id más cercana menor a k, que devolverá la información del nodo más cercano menor a k de su tabla.*

Desventaja: nodos próximos en el anillo pueden estar físicamente muy lejos (suponiendo un retardo de comunicación mucho mayor)

Pastry: similar al algoritmo Chord pero considerando la localización de los nodos para evitar los saltos de mensajes entre nodos muy separados geográficamente. En lugar de organizar el espacio

en círculo como Chord, el routing se basa en cercanía numérica de los IDs. (mayor coincidencia de prefijo)

CAN: la topología de red es un espacio cartesiano multidimensional; cada nodo es responsable de una zona del espacio; y cada clave es el hash de un punto del espacio.

8. Agentes

El campo de computación por sistemas multiagente nace de la demanda de sistemas de comunicación que puedan actuar de manera independiente y defendiendo nuestros intereses, mediante la cooperación (o competición) y acuerdos con otros sistemas con los mismos intereses.

- ✓ **Agente**: sistema de computación capaz de realizar una acción de forma independiente y a favor de su dueño.
- ✓ **Sistema Multiagente:** sistema que agrupa un número de agentes que interactúan entre sí. Para una interacción apropiada es necesaria la habilidad de cooperar, coordinarse y negociar entre ellos.

Este paradigma supone dos grandes problemas:

1. **Diseño de agentes:** como construir agentes capaces de tomar decisiones de manera independiente y autónoma para llevar a cabo las tareas que delegamos en ellos.
2. **Diseño de sociedades:** como construir agentes que sean capaz de cooperar.

Sistemas Multiagente (SMA)

Los agentes son entidades con 5 características identificativas:

1. **Entidades autónomas:**
   - Autonomía: pueden trabajar sin la intervención directa del usuario, tienen cierto control sobre sus acciones y estado interno.
   - Reactividad: pueden percibir su entorno y responder oportunamente a cambios que se produzcan en el mismo.
   - Iniciativa: su comportamiento está determinado por los objetivos que persiguen (pueden iniciar ellos las acciones)
   - Influencia del entorno**:** lo más común es que el agente solo tenga un control parcial del entorno; de forma que una misma acción puede tener resultados muy diversos y el agente debe estar preparado para fallar o incertidumbre de no saber si ha tenido éxito o no.
2. **Inteligencia**
   - Razonamiento**:** puede decidir qué meta perseguir, cómo actuar para conseguirlo y si suspender el objetivo para dedicarse a otro.
   - Aprendizaje**:** puede adaptarse progresivamente a cambios en entornos dinámicos.
3. **No actúan solos (multiagente)**
   - Reparto de responsabilidades
   - Heterogeneidad
   - Concurrencia y distribución: distribución del conocimiento y flexibilidad, escalabilidad, tolerancia a fallos y gestión de recursos.

***Test de Huhns-Singh:** *Un sistema que contiene uno o más agentes debe cambiar substancialmente si otro de los agentes es añadido al sistema.* Mide el grado en que un agente interactúa con otros o cambia su comportamiento en la presencia de otros agentes.

4. **Habilidad Social:**
    - Interacción
    - Delegación (asignar la realización de tareas)
    - Cooperación (trabajo en común para lograr un fin)
    - Negociación (formular un acuerdo)
5. **Movilidad:**
    - Agentes móviles: migrar de un nodo a otro en una red preservando su estado en los saltos entre nodos.
    - Multi-acceso y multi-modal.

Ventajas agentes:

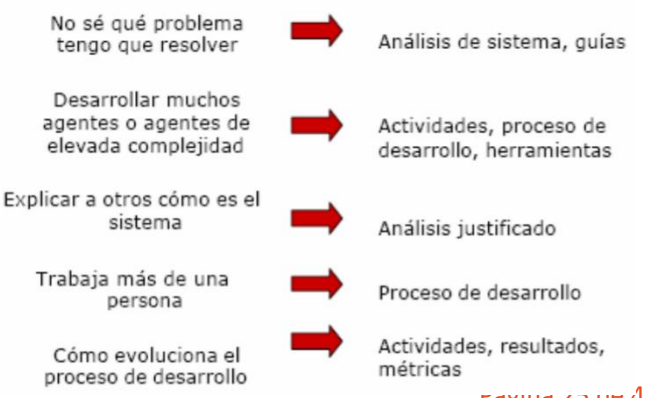
1. Para sistemas distribuidos proporcionan: aspectos sociales, lenguajes y protocolos de comunicación, distribución de datos, control, conocimiento y recursos.
2. Los agentes tienen un grado de abstracción mayor que los objetos, debido a:
    - Mayor autonomía y capacidad de decisión
    - Varios componentes heterogéneos que mantienen relaciones
    - Modelado de sistemas naturales y sociales
3. Facilitan la evolución (adaptación al entorno, escalabilidad añadiendo más agentes, permite añadir y quitar funcionalidad en tiempo de ejecución, un desarrollo incremental y son sistemas abiertos, es decir, aceptan nuevos elementos)

Desventajas: ausencia de control y de visión global del sistema.

Aplicaciones basadas en multiagentes:

- ✓ Personalización de servicios:
    - Servicios de información en Internet
    - Comercio electrónico
- ✓ Flexibilidad de la distribución:
    - Equipos móviles y PCs en el hogar
    - Redes públicas de telecomunicaciones.
- ✓ Delegación de tareas:
    - Gestión de procesos
    - Juegos
    - Robótica

Metodología: necesario porque el desarrollo de SMA es complejo



Los entornos en los que se ejecutan los agentes se pueden caracterizar por:

- **Accesibilidad**: si el agente puede obtener información completa y exacta del estado de su entorno. La mayoría de entornos son inaccesibles; pero cuanto más accesible sea, más fácil es crear agentes que operen sobre él.
- **Determinístico**: cuando cualquier acción tiene garantizado un único efecto (n*o hay incertidumbre sobre el estado que resultará de la acción realizada*). EL mundo real es no determinístico, sobre los entornos no determinísticos es mucho más complejo desarrollar agentes.
- **Episódico**: prestaciones de un agente dependen de un número discreto de episodios, independientemente del escenario concreto.
- **Estático**: entorno que permanece inalterado salvo por las acciones propias que tome el agente. En un entorno dinámico existen otros procesos operando y modificando este entorno; interfiriendo en acciones que tome el agente.
- **Discreto**: si existe un número fijo y finito de acciones que puede percibir (ajedrez) vs continuo (conducción de un taxi).

***Intención**: tipo de actitudes empleadas para describir acciones (*Ana cogió su paraguas porque **creía** que iba a llover*)

***Sistema de intenciones:** describe entidades cuyo comportamiento puede ser predicho mediante la atribución de creencias y deseos.

Arquitecturas de agentes: metodologías particulares para construir agentes. Se pueden clasificar según el tipo de procesamiento:

1. Arquitectura basada en la lógica: representación del estado interno según un conjunto de sentencias lógicas; empleando **reglas de deducción lógica para tomar decisiones (la decisión sobre qué acción ejecutan se basa solo en el estado interior del agente)**

Representación clara, pero con una complejidad temporal elevada. Además, resulta difícil encontrar representación simbólica para entidades reales.

2. Arquitecturas deliberativas: arquitecturas que utiliza modelos de representación simbólica del conocimiento. L**as decisiones se toman por deliberación interviniendo las creencias y deseos** del agente.

- **Agentes audaces**: no se paran para reconsiderar sus intenciones (coste temporal y computacional bajo) aptos en entornos que no cambian rápidamente.
- **Agentes cautos:** constantemente se detienen a reconsiderar las intenciones; explotando nuevas posibilidades (aptos para entornos que cambien rápidamente)

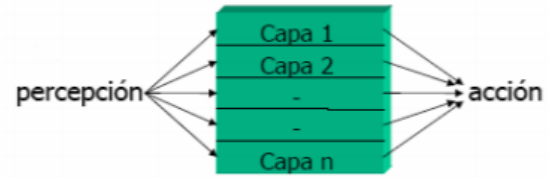
3. Arquitecturas reactivas: se caracterizan por no tener como elemento central de razonamiento un modelo simbólico, **simplemente reaccionan a eventos** . Ejemplo típico: arquitectura de subsunción: la toma de decisión está realizada en base a un conjunto de tareas; pudiendo dispararse muchas conductas simultáneamente.

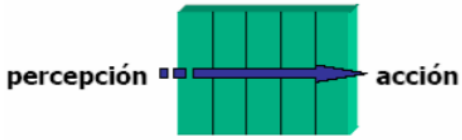
Respuesta inmediata del agente y elimina problema de la representación simbólica, pero es difícil diseñar agentes puramente reactivos; y generan unas interacciones difíciles de entender con muchas conductas.

4. Arquitecturas híbridas: no resulta acertado emplear arquitectura puramente deliberativa ni totalmente reactiva; por ello los sistemas híbridos combinan ambas. Agente compuesto por dos subsistemas (uno deliberativo que genere planes de acción, y otro reactivo centrado en reaccionar a eventos) El sistema típico se estructura en 2 capas: capa para la conducta reactiva y capa para la iniciativa.

Arquitectura en capas:

- **Capas horizontales:** cada capa esta conectada de forma directa a sensores y actuadores. (permite mayor paralelismo pero requiere un alto conocimiento de control para coordinarlos)
- **Capas verticales:** sensores conectados a una única capa, encargada de transmitir los estímulos recibidos a las demás capas hasta la capa final, que desencadena la acción (solo esta última capa está conectada con los actuadores). *Requiere menor conocimiento de control, pero supone mayor complejidad en la capa que interactúa con los sensores).*





Comunicación entre agentes.

Sistema multi agente: sistema formado por un conjunto de componentes (semi) autónomos caracterizados por:

1. Cada agente no tiene información completa ni capacidad para resolver el problema por si solo.
2. Tienen puntos de vista limitados
3. No existe sistema de control global.
4. Los datos están descentralizados.
5. Computación asíncrona.

Teoría de los actos del habla ("speech acts") La comunicación entre agentes se basa en esta teoría: *Quien habla no solo declara sentencias verdaderas y falsas, sino que realiza **actos de habla**: peticiones, sugerencias, promesas, amenazas…*

Tipos de acto de habla

1. **Actos asertivos** (dan información general)
2. **Actos directivos** (solicitan algo)
3. **Acto de promesa:** comprometen a realizar acciones en el futuro
4. **Actos expresivos:** indicaciones del estado mental del locutor
5. **Actos declarativos** (declaran algo)

Componentes en un acto de habla ("cierra la puerta")

1. **Locución**: quien habla, quien escucha, qué puerta..
2. **ILocución**: acto del locutor sobre el destinatario (quiere que el receptor cierre la puerta)
3. **Perlocución**: efectos que puede tener en el estado del destinatario (receptor cierra la puerta)

Servicio de transporte: se encarga del envío y codificación de mensajes para su transmisión como una secuencia de bytes; el agente que lo usa puede decidir si emplea procesamiento síncrono o asíncrono.

Se emplea los actos de habla para conseguir **completitud** (*cubrir diversas situaciones de comunicación*), **simplicidad** y **concisión** (*minimizar redundancia y ambigüedad*). Requerimientos:

- Un agente cuando reciben un mensaje que no reconoce debe enviar un mensaje **not-undesrtood**; y deben estar preparados para recibir y manejar estos mensajes.
- Agente ACL (*Agent Communication Languaje*) puede escoger implementar cualquier subconjunto de tipos de mensajes y protocolos predefinidos. Así como implementar actos comunicativos no estándar siempre que se encargue de que el receptor los entienda.

Protocolos FIPA ACL:

1. **FIPA-query**: se emplea para solicitar a un agente que realice una acción de tipo inform (query-if y query-ref)
2. **FIPA-request:** solicita a otro agente que realice una acción; este debe realizarla o responder que no puede.
3. **FIPA-request-when**: análogo a request pero debe esperar a que se cumpla una condición para responder.
4. **FIPA-contract-net**: agente desea que se realice una acción para la que hay varios candidatos.
5. **FIPA-iterated-contract-net**: cntract-net con varias rondas (se inicia con una cfp, cada participante emite su oferta, y el iniciador puede aceptar una, rechazar todas o emitir nuevo cfp)
6. **FIPA-english-auction** (subasta al alza) el iniciador emite cfp con el precio en cada ronda.
7. **FIPA-dutch-auction** (subasta a la baja)
8. **FIPA-brokering** (intermediación entre agentes), bróker envía petición a conjunto de agentes y proporciona la respuesta.
9. **FIPA-recruiting**: análogo a brokering pero son los agentes los que envían la respuesta.
10. **FIPA-suscribe**: iniciador solicita ser avisado cada vez que se cumpla la condición dada.
11. **FIPA-propose**: emisor propone a los participantes realización de una acción.

Lenguaje KQML: lenguaje para comunicar actitudes sobre la información, indiferente al formato de la información.

Define la comunicación a cuatro niveles:

1. **Transporte** (como se envían y reciben mensajes)
2. **Lenguaje**: qué significa cada mensaje
3. **Política**: como se estructuran las conversaciones
4. **Arquitectura**: cómo conectar los sistemas

JADE (Java Agent Development Framework) middleware que sirve para desarrollar aplicaciones distribuidas multiagente, totalmente compatible con Java y cumple estándar FIPA.

Arquitectura JADE:

- ✓ Compuesta de una colección de componentes activos (agentes)
- ✓ Cada agente tiene un nombre único, forma parte de una aplicación P2P puesto que puede comunicarse de forma bidireccional con los demás agentes.
- ✓ Cada agente vive en un contenedor que proporciona su entorno de ejecución.

Ejecución: java jade.Boot -gui nombre1:paquete.agente1;nombre2:paquete.agente2

Herramientas gráficas de Jade:

- ✓ **RMA (Remote Management Agent):** monitoriza y controla la plataforma y todos sus contenedores remotos. Permite la gestión remota del ciclo de vida de agentes (*creación, suspensión, eliminación…)*, permite el envío de mensajes a los agentes y el lanzamiento de las otras herramientas gráficas.
- ✓ **Dummy Agent**: compone y envía mensaje, almacena la cola de mensajes en un archivo.
- ✓ **Sniffer Agent:** muestra flujo de interacciones entre los agentes seleccionados; carga/almacena el flujo desde un archivo.
- ✓ **Introspector Agent:** monitoriza el estado interno del agente y permite depuración en ejecución.
- ✓ **Log manager Agent:** interfaz para modificar en tiempo de ejecución el almacenamiento de la plataforma.
- ✓ **DFGUI:** interfaz para interacción con el servicio de páginas amarillas

Programación en Jade

Crear Agente: extendiendo la clase **jade.core.Agent** y sobrescribiendo el método setup().

Cada agente se identifica por un Agent ID (AID). El nombre de un agente debe ser único: <nombreLocal>@<nombrePlataforma> donde el nombre por defecto de la plataforma es <host>:<port>/JADE.

- ✓ **getAID()** →devuelve el AID del agente
- ✓ **New AID(localname, AID.ISLOCALNAME)** →crea el aid a partir del nombre local
- ✓ **New AID(name, AID.ISGUID**)→ crea el aid a partir del nombre global.
- ✓ **getArguments()→** recupera los argumentos pasados al agente
- ✓ **doDelete()→** concluye la ejecución del agente → ejecuta automáticamente el **takeDown()**

Comportamientos: el trabajo de un agente se realiza normalmente dentro de *behaviours.* Se pueden crear comportamientos extendiendo la clase jade.core.behaviours.Behaivour. Para que un agente ejecute un comportamiento basta con invocar el método addBehaviour() de la clase Agent.

Cada Behaviour puede implementar:

- ✓ **Public void action()** → lo que ejecuta el comportamiento
- ✓ **Public boolean done()** → devuelve true cuando concluye el comportamiento.

El **scheduling de comportamientos es cooperativo** y todo se ejecuta en un único thread Java. Se conmutan comportamientos cuando concluye el método action() del comportamiento activo (o bien este se bloquee)

Tipos de comportamientos:

- ✓ **One Shot** (extiende OneShotBehaviour): concluye de forma inmediata (done siempre es true), su action se ejecuta una única vez.
- ✓ **Cíclico** (extiende CyclicBehaviour): nunca terminan (done siempre falso).
- ✓ **Complejos** (Complex): mantienen un estado interno; el método action ejecuta operación diferente dependiendo de este estado. Termina cuando verifica una condición especificada cuando se implementa.

JADE proporciona dos comportamientos a mayores (que ya tienen implementados sus métodos action y done)

- ✓ **WakerBehaviour**: la subclase debe implementar el método **onWake()** que es ejecutado después de un cierto tiempo. Este comportamiento concluye al terminar la ejecución de este método.
- ✓ **TickerBehaviour**: la subclase debe implementar el método **onTick()** que es ejecutado peri´dicamente. Nunca termina a menos que se ejecute el método **stop().**

Funciones de los comportamientos

- ✓ **onStart()** → se ejecuta una única vez antes del método action().
- ✓ **onEnd()** → se ejecuta una única vez después de que done() devuelva true.
- ✓ **myAgent** → es una variable protegida desde la que el comportamiento puede referenciar el agente que lo está ejecutando.
- ✓ **Agent.removeBehaviour()** → suprime un comportamiento sin que se invoque su método onEnd().

**Cuando la lista de comportamientos de un agente está vacía, el agente entra en estado IDLE (su thread se duerme)*

Comunicación: basado en el paso de mensajes que sigue el formato **ACL (FIPA).** Los mensajes que intercambian son de la clase de Jade **ACLMessage** *(get/setPerformative(), get/setSender()…)*

Envio de mensaje ACL

```
ACLMessage msg = new ACLMessage(ACLMessage.Performativa)
Msg.addReceiver(new AID("Peter", AID:ISLOCALNAME));
Msg.setLanguaje("English");
Msg.setOntology("ontologia");
Msg.setContent("Contenido del mensaje jeje");
Send(msg)
```

Recepción de mensaje ACL

```
ACLMessage msg = receive()
```

Si la recepción de mensaje se hace desde un comportamiento; hay que tener en cuenta que los comportamientos son colaborativos (funcionan en un único thread) e interesa ceder la CPU mientras se espera por la recepción de un mensaje.

Receive() devuelve null cuando la cola de mensajes está vacía; para evitar estar comprobando de manera continuada ocupando la CPU, se emplea **block()** que elimina el comportamiento de la cola de comportamientos y lo coloca como bloqueado. Cuando un agente recibe un mensaje, todos sus comportamientos vuelven a la cola de comportamientos y tienen la oportunidad de leer el mensaje.

```
Public void action(){
  ACLMessage msg=myAgent.receive();
    If(msg ¡= null){
          ….
     }else{
          block();
      }
`
```

Como alternativa a block() existe **blockingReceive();** que solo concluye cuando hay un mensaje en la cola. Solo se recomienda su uso dentro de setup() y takeDown().

Robo de mensajes: El método receive() devuelve el primer mensaje de la cola de mensajes eliminándolo de esta. Si dos comportamientos están leyendo mensajes es posible que se "roben" los mensajes; por ello se definen unas plantillas/filtros para determinar qué mensajes se deben sacar de la cola.

> **MessageTemplate tpl = MessageTemplate.MatchOntology("holi");**
> **ACLMessage msg = myAgent.receive(tlp)** → solo recibe los mensajes que cumplen la plantilla (*en este caso que su ontología sea "holi")*

Servicio de Páginas Amarillas (DF-Directory Facilitator): permite buscar/registrar agentes en el servicio de páginas amarillas para poder ser localizado por otros agentes. DF es un agente, por lo tanto se comunica por ACL.

- Registrar Agente(unregister): necesario especificar una descripción del agente (**DFAgentDescription**) que contenga el AID del agente y una **ServiceDescription** (*colección de descripciones de servicios, tipo de servicio, nombre…)*
- Buscar Agente (search): emplea un **DFAgentDescription** con las características del agente/s que está buscando

Otras características de JADE

1.Agent Management System (ASM) representa la autoridad en la plataforma JADE; todas las operaciones de gestión están bajo su control.

getAMD() → devuelve el AID del AMS.

2.Movilidad JADE soporta **movilidad fuerte**, es decir:

- **movilidad de estado** (*un agente en ejecución puede moverse a otro contenedor y continuar la ejecución en el punto en que estaba*)
- **movilidad de código (***si el código del agente no está disponible en el nuevo contenedor puede obtenerse bajo demanda).*

Solo si el agente es **Serializable**; esta movilidad puede ser auto iniciada (doMove()) o forzada por el AMS. Además puede ser clonado (doClone()).

3.Seguridad: Jade previene amenazas (*posibles agentes maliciosos y sniffer de mensajes*) mediante Autenticación y autorización e Integridad y confidencialidad punto a punto. Todos los aspectos de seguridad se incluyen en el **SecurityHelper** (*se puede obtener con* getHelper()

4.Características avanzadas:

- **Interfaz con procesos** (permite usar JADE desde un programa externo de Java)
- **Comportamientos en thread:** ejecutar comportamientos en JAVA, con un thread dedicado *(es decir, sin ser dentro de un agente JADE*)
- **Persistencia**: guardar y recuperar el estado del agente en una BD relacional

9. Ontologías

Formas de enviar datos dentro de un mensaje ACL:

1. **Formato String.** Difícil leer para una máquina, requieren uso de Split()
2. **Objetos Serializables de Java.** Requiere que todos los agentes estén en el mismo lenguaje de programación.
3. **Ontologías** Definen clases que representan los contenidos a transmitir, se traducen directamente al contenido de un mensaje ACL..

Ontología: nomenclatura y definición formal de los tipos y propiedades de las entidades que forman un dominio y las relaciones entre ellos.

- ✓ Elimina la ambigüedad: es imposible formar mal un empleando una ontología. Múltiples entidades tendrán un modelo común del mismo dominio, por lo que se entenderán entre si.
- ✓ Es independiente de la implementación: dos agentes con distintos objetos que representen la misma ontología se entenderán.
- ❖ La elaboración de ontologías es compleja, en particular para dominios inestables.

En Jade: las ontologías se componen de dos partes:

- Clase derivada de Ontology (define el esquema: elementos del dominio y sus relaciones)
  - contiene Schemas
  - no varía durante la ejecución del programa
- Clases que implementan los elementos del esquema (se implementan como *beans*)
  - **Concept**: concepto básico de la ontología; los demás elementos se construyen en base a Concepts y, un Concept puede contener otros Concept. (*en una puja de libros, un concept sería Libro)*
  - **Predicate**: afirman algo sobre el estado del mundo real; se puede decir que son falsos o verdaderos (*un predicado sería Juan vende LosJuegosdelHambre, que puede ser verdadero o falso*)
  - **AgentAction**: acción que puede llevar a cabo un agente (pujar por un libro). No se envían directamente AgentAction en los mensajes, sino un new Action(AIS, AgentAction)

Slot: *atributo de un Concept, Predicate o AgentAction, ya que funcionan como huecos genéricos que pueden ser rellenados por los agentes.*

Lenguaje de contenido (codec) proporcionan una manera estándar de codificar los beans de una ontología en un formato transmitible dentro de un mensaje ACL. En JADE hay dos:

- **SL** *(uso mayoritario)*: elementos codificados en base a cadenas de caracteres, siendo legibles para los humanos.
- **LEAP**: elementos codificados como secuencia de bytes (no legible para humanos). Solo es compatible con JADE, se emplea en contextos de alto rendimiento porque es más ligero.

SL

```
((action
    (agent-identifier
      :name a@192.168.1.35:1099/JADE
      :addresses (sequence
http://main:7778/acc))
    (Pedir
      :vinilo
        (Vinilo
          :titulo Ontologia
          :artista ComDis))))
```

LEAP

```
00000011 00000011 01100001 01100011
01110100 01101001 01101111 01101110
00100000 01100001 01100011 01110100
01101111 01110010 00100000 01100001
01100111 01100101 01101110 01110100
00101101 01101001 01100100 01100101
01101110 01110100 01101001 01100110
01101001 01100101 01110010 00100000
01101110 01100001 01101101 01100101
00100000 01110001 01000000 00110001
00111001 00110010 00101110 00110001
00110110 00111000 00101110 00110001
00101110 00110011 00110101 00111010
00110001 00110000 00111001 00111001
00101111 01001010 01000001 01000100
01000101 00100000 01100001 01100100
01100100 01110010 01100101 01110011
01110011 01100101 01110011 00100000
01110011 01100101 01110001 01110101
01100101 01101110 01100011 01100101 …
```

ContentManager: clase de servicio que se encarga de la administración de ontologías y lenguajes de contenido. Se obtiene mediante Agent.getContentManager().

- registerOntology() → registrar instancia de la Ontología a usar
- registerLanguage()→ registrar instancia del codec a usar
- fillContent() → codifica e inyecta el Bean en el contenido del mensaje
- extractContent() → decodifica el contenido de un mensaje ACL.

10. Jess y RETE

Jess (Java Expert System Shell) es un lenguaje que proporciona una programación basada en reglas para la automatización de un sistema experto. Se emplea en el desarrollo de sistemas de agentes inteligentes para una toma de decisión basada en reglas.

Cada agente decide que acciones realizar en base a un sistema de reglas, que permiten percibir su entorno y seleccionar comportamientos dependiendo de su estado actual.Las reglas de Jess son del tipo if-then. Comprueban una condición del entorno y, si se verifica, ejecutan la consecuente acción.

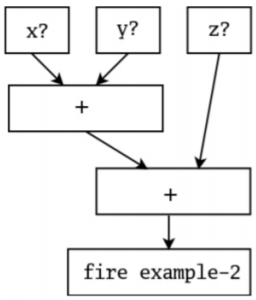
Algoritmo RETE: la idea de que, en todo momento se tengan que revisar secuencialmente TODA la lista de normas que determinen el comportamiento del Agente es muy ineficiente. El **algoritmo RETE** supone una solución más eficiente para esto, de forma que solo es necesario actualizar aquellos elementos del entorno que se han actualizado:

Se implementa mediante la construcción de una red de nodos. Cada nodo representa un patrón (condicional) de una regla. Los nodos raíz (finales) representan la acción final.

Por ejemplo; la norma : ***If (x and y and z) then fireexample-2***

El algoritmo consiste en que cada vez que se produce un cambio en un elemento concreto del entorno; active/desactive el nodo asociado a dicho elemento. Cuando todos los nodos hasta un nodo hoja están activos; debe ejecutarse la acción correspondiente al nodo.

Consigue un tiempo de ejecución muy alto sacrificando memoria.



Integración en Jade: Jess incluye una clase especial llamada **Rete**, que implementa el motor de inferencia basado en reglas (*implementa el algoritmo de Rete*). Para integrar Jess en Jade basta con crear el objeto Jess.Rete y manipularlo apropiadamente.

La llamada al método **Rete.run()** hará que se apliquen todas las comprobaciones y reglas lógicas hasta que no queden más normas que ejecutar, bloqueando el hilo completo del agente. Como alternativa existe otro método que permite limitar el número de ciclos que debe ejecutarse **Jess.run(MAX_JESS_PASSES);**

La idea es que Jess reciba estímulos del entorno (mensajes de otros agentes) y reaccione según las reglas con alguna acción (envío de mensajes). Para el envío de mensajes, puede implementarse desde Jess o bien implementar la clase **Jess.Userfuction.**

Exámenes Resueltos

Cuando un objeto servidor se registra en el registro de RMI, debe construir una dirección URL que será un argumento de la función rebind del objeto Naming. Describe brevemente el formato de dicha URL, indicando los valores posibles de cada uno de sus componentes. ¿En qué se diferencia (o se puede diferenciar) de la URL utilizada por el cliente y que se le suministra al método lookup también del objeto Naming?
Formato URL: rmi://<hostname>:<puerto>/<nombre>

- **Hostname**: ip del servidor en que reside el objeto distribuido. Cuando se hace la llamada a bind o rebind siempre toma el valor "localhost" puesto que al registrarlo desde el servidor la dirección del objeto distribuido necesariamente coincide con la dirección local. Puede variar cuando en vez de llamar al método bind, se llama al método lookup (buscar la referencia a un objeto remoto) puesto que la llamada se puede estar haciendo desde un equipo diferente al equipo en que reside el objeto, en cuyo caso hay que especificar la ip del equipo en que resida.
- **Puerto**: puerto para acceder al objeto distribuido; debe ser igual en ambas llamadas.
- **Nombre**: nombre simbólico que se le da al objeto; debe ser único en el registro.

Describe brevemente la lista de archivos necesarios, tanto en la parte del servidor como en el cliente para una aplicación Java RMI con callback de cliente.
Una aplicación con callback implica la comunicación bidireccional (el servidor también puede invocar métodos ofrecidos por el cliente) por lo que se incorporan archivos para que el cliente exporte también sus métodos:

Objetos cliente:
- Cliente.class : clase cliente que registra su objeto e invoca métodos del servidor.
- InterfazCliente.class : interfaz remota de métodos ofrecidos por el cliente
- ImplCliente.class : implementación de la InterfazCliente
- ImplCliente_skel: proxy para las peticiones que se reciban a ImplCliente
- InterfazServidor.class: interfaz de métodos remotos exportados por el servidor.
- ImplServidor_stub.class: proxy para las llamadas a métodos del Servidor.

Analogamente objetos en el servidor:
Objetos servidor:
- Servidor.class : clase cliente que registra su objeto e invoca métodos del cliente.
- InterfazServidor.class : interfaz remota de métodos ofrecidos por el servidor
- ImplServidor.class : implementación de la InterfazCliente
- ImplServidor_skel: proxy para las peticiones que se reciban a ImplCliente
- InterfazCliente.class: interfaz de métodos remotos exportados por el servidor.
- ImplCliente_stub.class: proxy para las llamadas a métodos del Servidor.

En el contexto de Java RMI, ¿cuál es el propósito de la serialización de objetos?
El objetivo es el poder enviar objetos complejos (por ejemplo objetos creados por nosotros) como argumento o valor de retorno en llamadas a métodos remotos. Para ello el objeto debe implementar la clase Serializable y a suvez, todos sus atributos deben ser o bien primitivos o serializables.

Comparada con Java RMI, ¿cuáles son los principales puntos fuertes de una herramienta CORBA, si hay alguno?. ¿Cuáles son los puntos débiles, si hay alguno?.

La principal ventaja de CORBA frente Java RMI es que es independiente del lenguaje; Java RMI es una herramienta exclusiva de Java, CORBA permite la comunicación entre objetos heterogéneos (diferentes lenguajes o plataformas) mediante un lenguaje universal, el CORBA IDL. Además el uso de IDL fuerza la separación de la interfaz y la implementación; permitiendo crear diferentes implementaciones para una misma interfaz.

No obstante, la implementación de aplicaciones con CORBA es más compleja, en especial para desarrolladores Java, requiere el aprendizaje de IDL y supone limitaciones como el mapeo de datos no exacto entre IDL y Java y la necesidad de clases auxiliares como los Holders. También limita más los datos que permite pasar en los métodos remotos

Ilustra, por medio de un diagrama de bloques, la arquitectura CORBA. Compárala con la arquitectura de Java RMI, incluyendo componentes equivalentes. Basándose en sus diagramas, escribe un párrafo que describa las diferencias principales entre las dos arquitecturas. Trata de explicar dichas diferencias.

La principal diferencia es en CORBA, se incluye el ORB como middleware para el mapeo de peticiones en CORBA IDL al lenguaje concreto y viceversa.

En la arquitectura Java RMI, se diferencian tres niveles de abstracción: al invocar un método remoto, el **proxy** se encarga de gestionar esta invocación remota, la redirige a la capa inferior , que encapsula sus argumentos y la transmite a la capa más baja que se asegura de ocupa de las conexiones y el envío del mensaje a la máquina remota, donde se recorre el camino inverso.
En CORBA las capas inferiores son equivalentes, que hacen llegar las peticiones (en CORBA IDL) al ORB, este identifica el object adapter al que corresponde la petición y se la envía, para que la traduzca al lenguaje concreto (narrowing) e invoque la llamada al método.

¿Qué problemas crees que tienen los protocolos Napster y Gnutella?
¿Cuándo utilizarías cada uno de ellos?

Ambos protocolos son mecanismos de compartición de archivos basados en una arquitectura P2P. La principal diferencia es que Napster emplea un mecanismo de búsquedas centralizado, donde todas las búsquedas de archivos se vuelcan sobre un servidor central, que responde (si existe) con la IP de aquellos clientes que dispongan del archivo buscado. Este servidor central supone un punto único de fallo para todo el sistema, un cuello de botella para el rendimiento y el punto crítico para que este modelo no sea escalable. Por otro lado, Gnutella elimina este servidor central, las búsquedas se realizan “preguntándole” a los contactos de la red peer por el archivo deseado, que a su vez preguntan a sus contactos y así secuencialmente (se limita la propagación mediante el TTL). Con esto, se eliminan los problemas de Napster pero por lo general supone que las búsquedas son más lentas, no asegura resultados correctos de las búsquedas (puede existir el archivo en la red pero no haberle llegado la petición al peer adecuado) y, el tipo de propagación de las peticiones supone que la red se sature cuando se hacen muchas peticiones simultaneas (inundamiento de red).

Por lo tanto, optaría por Napster si busco un archivo de música, quiero hacer búsquedas rápidas y puedo tolerar caídas del sistema. Por otro lado, si el archivo que busco NO es de música o no me puedo permitir caídas del sistema, optaría por Gnutella.

En Jade los comportamientos no son apropiativos sino colaborativos.
Explica qué significa eso.
Se refiere a que todos los comportamientos comparten un único thread de ejecución; por lo que solo uno puede estar ejecutándose simultáneamente. Al programar los comportamientos se debe tener esto en cuenta y no ocupar la CPU de forma abusiva, cediendo su uso en casos como la espera por la recepción de un mensaje para que los demás comportamientos puedan ejecutarse y así lograr la meta común en la que "colaboran".

¿Para qué sirve el método block() de la clase Behaviour en Jade?. ¿Sepuede usar indistintamente los métodos block() y blockingReceive() dentro de un comportamiento?. Explicar por qué.
Block() bloquea al comportamiento que lo ejecuta, eliminándolo de la cola de hilos activos y cede el uso de la CPU para que los demás comportamientos se puedan ejecutar. El comportamiento dejará de estar bloqueado cuando se reciba un nuevo mensaje, que se devolverá a la cola de comportamientos activos. Se usa normalmente para recibir mensajes, para que cuando se esté esperando a la llegada de un mensaje se ceda el uso de la CPU.
blockinReceive() es una llamada bloqueante, no finaliza hasta que se reciba un mensaje, ni cede la CPU mientras tanto, es decir, bloquea el hilo entero. Su uso no es apropiado por lo general dentro de un comportamiento excepto de manera justificada o bien en los métodos setUp() o takeDown() porque va contra la naturaleza de una aplicación colaborativa.

Indica cuáles son las principales características que hacen que un programa se convierta en un agente. ¿Qué diferencia a un agente de un cliente peer to peer?

1. **Entidad autónoma:** capacidad de ejecutarse sin la intervención directa del usuario, de reaccionar a estímulos del entorno, de iniciar acciones para lograr sus metas y estar preparado para que sus acciones puedan fallar o la incertidumbre de no saber si han fallado o no por el poco control sobre el entorno.
2. **Inteligencia:** capacidad de razonar y aprender progresivamente adaptando su comportamiento para lograr sus objetivos.

3. **No están solos (multiagente)** capacidad de cooperación, delegación y reparto de responsabilidades (divide y vencerás)
4. **Habilidad social:** capacidad de cooperar (o competir), coordinarse, dialogar y negociar con otras entidades para lograr pactos y alcanzar sus metas.
5. **Movilidad:** capacidad de migrar de nodo en nodo.

Un agente se caracteriza por estos cinco rasgos. Por otro lado, un cliente peer to peer se caracteriza por participar en una red P2P donde tiene las mismas responsabilidades y capacidades que los demás usuarios; todos actúan a la vez como cliente y como servidor.
No obstante, no son conceptos excluyentes, un agente puede ser cliente en una red P2P. De hecho, ambos paradigmas aportan numerosas ventajas: el servicio de páginas amarillas, la movilidad y la alta modularidad de los agentes; así como la conectividad y balanceo de cargas de una red P2P que puede ser muy beneficioso combinar.

Define brevemente el concepto de middleware.
Software que funciona como intermediario entre dos procesos independientes. Suele funcionar como pieza clave en la abstracción; por ejemplo en el paradigma del sistema de mensajes, el sistema de mensajes funciona como middleware entre procesos para un intercambio desacoplado de mensajes.

¿Cuáles son las principales diferencias entre el paradigma del paso de mensajes y el paradigma del sistema de mensajes?.
El paso de mensajes se basa en la comunicación directa entre procesos; suponiendo una fuerte acoplación de los procesos. Por el otro lado, el paradisma de sistema de mensajes incorpora el servidor (sistema de mensajes) como middleware entre los procesos que se quieren comunicar, los procesos depositan en el sistema de mensajes los mensajes que desean enviar y este se ocupa de conmutarlos. Ofrece una comunicación desacoplada y asíncrona entre procesos. Para lograr esta comunicación asíncrona con el paradigma de paso de mensajes sería necesario el uso de hilos.

¿Qué es una asociación en Internet?.
Es una relación que incluye información del protocolo (TCP/IP) el puerto y hostname local y el puerto y hostname remotos.

La clase DatagramPacket tiene dos constructores distintos ¿Para qué sirve cada uno de ellos?
DatagramPacket(byte[] data, int lenght, InnetAdress receiver, int port)
Se emplea para el envío de mensajes, puesto que encapsula el contenido del mensaje (data) para enviárselo al receptor dado (por la ip y puerto)
DatagramPacket(byte[] data, int lenght)
Se usa para el recibo de mensajes, el contenido del mensaje recibido se almacena en el parámetro data y su tamaño no debe superar el máximo fijado.

Explica la diferencia entre broadcast y multicast.
Ambos son mecanismos para el envío de un paquete a múltiples receptores; no obstante con broadcast se envía a todos los miembros de una red el mensaje; mientras que con multicast se envía solo a los miembros de un grupo multicast (subred)

El stub y el skeleton son, respectivamente, los proxies de cliente y servidor en la arquitectura RMI. Explica cómo se generan dichos proxies.
Se crean invocando al compilador rmic; existen varias opciones válidas:

- *Rmic -vcompat NombreImplementacion*
- *Rmic -v1.1 NombreImplementacion*

¿Cuál es la razón de que todos los métodos en una interfaz remota lancen la excepción RemoteException?.
Que la invocación de métodos remotos puede generar errores de conexión con la máquina remota o errores propios de la invocación de métodos remotos, como no encontrar el skeleton, el objeto remoto.. Todos estos errores lanzan esta RemoteException y deben ser tratados desde el programa que invoca los métodos.

La clase Naming tiene, entre otros, dos métodos denominados bind y rebind que sirven para que el servidor de objetos publique en el registro RMI un objeto remoto. ¿En que se diferencian a nivel de comportamiento?.

Bind() solo escribe en el registro si no existe ningún otro objeto ya registrado con el nombre pasado por parámetro.

Rebind() siempre escribe en el registro; si ya existía algún objeto con ese nombre lo sobreescribe.

Explica bajo qué circunstancias puede ser necesario el uso de un SecurityManager en java RMI.

El SecurityManager es una clase Java que limita el acceso del software. Es importante usarlo con aplicaciones como las que usan Java RMI porque permiten el acceso desde una máquina remota para así evitar accesos inadecuados.

¿Qué es un IOR?

IOR (Interoperable Object References) protocolo de referencias abstractas a objetos. Una aplicación compatible con IOR puede registrar/obtener referencias de objetos de un registro compatible con IOR.

A su vez, las referencias a objetos CORBA también se denominan IOR, y encapsulan datos del tipo de objeto, puerto y hostname de la máquina en que reside y la clave de objeto en una cadena de caracteres del tipo: IOR:01010101000101…

En la sintaxis de IDL que significa que un método sea de tipo oneway?. ¿Qué repercusiones al respecto de los tipos de entrada o los parámetros de salida tiene que un método sea oneway?.

Quiere decir que es un método no bloqueante, el proceso que lo invoca puede continuar con su ejecución sin esperar a que el método invocado haya finalizado.

No debe tener valor de retorno ni argumentos del tipo out o inout; puesto que ni se va a esperar a que devuelva el valor, ni se debe permitir que modifique valores pasados como parámetros de forma paralela a la ejecución del programa que lo invocó.

Define lo que se entiende por widening y narrowing?

Widening (wide=ampliar): proceso para traducir un objeto Java a una referencia de objeto CORBA

Narrowing (narrow=estrechar): proceso de traducción de una referencia de objeto CORBA a un objeto Java.

¿Cuáles son las diferencias fundamentales entre un sistema P2P y un sistema multiagente?.

Un sistema multiagente se caracteriza por ser una agrupación de agentes (caracterizados por ser entidades autónomas, con inteligencia, que no están solas, movilidad y habilidades sociales) que tienen una visión limitada del problema, no tienen capacidad individual para la resolución del problema, los recursos están descrentralizados, no hay control central…

Mientras que un sistema P2P se caracteriza porque todos los nodos tienen las mismas capacidades y responsabilidades, actuando a la vez como clientes y como servidores (consumen servicios y ofrecen servicios).

No obstante, no son modelos exluyentes, se puede constituir un sistema multiagente que siga una arquitectura P2P, donde cada agente se pueda comunicar de forma

bidireccional con los demás agentes y compartir de forma directa sus recursos; combinando las ventajas que ofrecen ambos paradigmas.

Qué es un super-peer en Kazaa y qué criterios se utilizan para hacer que un nodo P2P se transforme en super-peer?.

Es un peer que funciona como servidor para una subred de la red P2P global; administrando las búsquedas de los nodos de su red. Se escogen de entre los peers disponibles en base a características como la disponibilidad, conectividad, capacidad de procesamiento, almacenamiento… Para proporcionar el mejor servicio a la subred.

Define los conceptos de protocolo y conversación en FIPA-ACL.

Protocolo: patrón de intercambio de mensajes agentes.

Conversación: instancia concreta de un protocolo.

¿Cuál es la función del agente DF en JADE?.

DF (Directory Facilitor) es el servicio de páginas amarillas. Ofrece métodos que permiten la interacción con estas páginas amarillas, como el registro, modificación, búsqueda y desregistro de agentes.

Además mediante el DFGUI permite la federación con otros FD y las búsquedas federadas.

Cuando deseamos modificar el contenido de una variable desde un entorno gráfico en JADE, debemos crear un comportamiento de tipo OneShotBehaviour que realice dicha acción. Indica por qué.

Porque es un comportamiento que se ejecuta una única vez y finaliza. Es decir, actualizaría el contenido de la variable y finalizaría automáticamente, eliminándose de la cola de comportamientos sin afectar a los demás comportamientos.

El método run de la clase jess.Rete tiene un argumento numérico. ¿Qué significa dicho argumento y por qué debe utilizarse dentro de un comportamiento en JADE en lugar del correspondiente método sin argumentos?.

El método run ejecuta las comprobaciones de reglas necesarios del algoritmo RETE; por defecto (sin argumentos) se comprobarían TODAS las condiciones necesarias hasta que no quedasen más, sin ceder la CPU mientras tanto, y sin saber a priori el tiempo real que puede suponer. Sería un comportamiento muy ineficiente dentro de una aplicación colaborativa; por ello se emplea el método con argumentos; donde limitas estas operaciones para asegurar no abusar del uso de la CPU.

¿Cuál es la función de la clase ContentManager en JADE?.

Maneja todo lo relacionado con las ontologías en Jade y con los lenguajes.Permite definir el lenguaje y ontología de un agente así como el encapsular o extraer el contenido de los mensajes que emplean ontologías.

En qué paradigma de computación distribuida se asignan deliberadamente roles asimétricos a los participantes en la comunicación? ¿Cuál es el objetivo que se persigue con ese tipo de asignación de roles?

En el paradigma cliente servidor, es un paradigma eficiente para aplicaciones centralizadas donde un servidor pasivo espera por peticiones de clientes. Busca facilitar la sincronización de eventos, el servidor espera por peticiones y el cliente envía peticiones y espera por las respuestas.

¿Si tuvieses que implementar una solución de computación distribuida basada en llamadas a procedimientos remotos y tuvieses que elegir entre Java RMI y CORBA, que criterio utilizarías para decidirte por una u otra tecnología?

Si no hay exigencias sobre los lenguajes o simplemente la elección de lenguaje de las aplicaciones es abierta la mejor solución es Java RMI, ya que su implementación es más sencilla y más natural para los desarrolladores Java; además es más flexible en el tipo de argumentos a pasar.

Si se requiriese el uso de un lenguaje que no sea Java o la comunicación de lenguajes heterogéneos, la única solución sería CORBA, ya que ofrece interoperabilidad entre lenguajes/plataformas.

En una aplicación cliente/servidor explica cómo se utilizarían las clases ServerSocket y Socket de API de Sockets de Java.

La clase ServerSocket sirve como base para crear la conexión. La clase Socket en cambio se emplea para el envío y recepción de mensajes en sí, una vez se haya establecido la conexión.

En el siguiente código, Hilo es una clase que hereda propiedades de Thread. El código pretende crear dos hilos que ejecutan el mismo código.

```
public class pruebaHilo {
        public static void main(String[] args){
                Thread hilos = new Hilo();
                hilos.start();
                hilos.start();
        }
}
```

Indica los posibles inconvenientes de usar esta forma de crear los hilos y, en el caso de ser necesario, propón las posibles modificaciones al código arriba presentado que solucionen dichos inconvenientes.

Es una invocación incorrecta, si se desea ejecutar dos hilos deben crearse dos identificadores direrentes. La ejecución de start dos veces generaría una excepción, por llamar a start sobre un hilo que ya está en ejecución.

Dado el siguiente código correspondiente a un servidor de objetos en Java RMI, donde SomImpl es una clase que implementa una interfaz remota

```
Import java.rmi.*
public class SomeServer{
        public static void main(String[]args){
        try{
                SomeImpl exportedObj = new SomeImpl();
                registryURL = “rmi://localhost:1099/some”;
                Naming.bind(registryURL, exportedObj);
                System.out.println(“Some Server ready”);
        }catch(Exception e){
        }
}
```

Explica bajo qué circunstancias podría no ejecutarse correctamente
Hay dos fuentes de error:

- No se activa el registro de objetos: puede funcionar correctamente si ya se había activado desde fuera del programa por línea de comandos (rmiregistry puerto); de lo contrario debería incluirse el código: LocateRegistry.createRegistry(puerto)

- Método bind: el método bind solo escribirá en el registro si no existe otro objeto con el nombre dado. Puede estar registrado un objeto obsoleto o totalmente diferente al que intentamos registrar.

Teniendo en cuenta que el ejemplo del ejercicio anterior es un servidor de objetos en Java RMI, define los conceptos de "servidor de objetos" y "objeto servidor".
El servidor de objetos es el encargado de instanciar el objeto a exportar y registrarlo en el rmi registry. Por otro lado, el objeto servidor es la implementación de los métodos de la interfaz remota; métodos que ofrece el objeto.

Supongamos que ejecutamos desde un objeto cliente un método remoto en un objeto servidor Java RMI y que antes de que concluya su ejecución, otro objeto cliente desea ejecutar dicho método. ¿Cómo resuelve java RMI dicho problema (si es que lo resuelve)?
El servidor de objetos es un servidor concurrente, es decir, genera diferentes threads para la gestión de llamadas simultáneas, por lo que no supondría ningún problema. Esto supone que la implementación del objeto remoto debe ser thread-safe.

IIOP e IOR son protocolos usados por CORBA. Explica para que sirve cada uno de ellos.
IIOP: GIOP (General Interoperable ORB Protocol) sobre IOP, es el protocolo por el que dos ORBS se comunican a través de internet.
IOR (Interprocess Object References) protocolo de referencias abstractas a objetos. Un servidor compatible con IOR puede obtener y registrar referencias a objetos en un directorio compatible con IOR. A su vez, las referencias a objetos CORBA también se denominan IOR, y encapsulan datos del tipo de objeto, hostname y puerto del equipo en que reside y clave del objeto en un formato del tipo IOR:01010111101

¿En un archivo de especificación IDL, que significa que el argumento del atributo de una operación sea de tipo in, out o inout? ¿Y qué significa que una operación sea de tipo oneway?
In: atributo de entrada, solo lectura no puede ser modificado
Out: atributo de salida, debe escribirse en el el valor para que el emisor lo obtenga
Inout: atributo ya inicializado con un valor, que se puede leer y modificar
Oneway es un método no bloqueante

Que es una Distributed Hash Tabla (DHT) y en qué contexto se utilizaría?
Las tablas hash distribuídas se emplean para las redes P2P no estructuradas como mecanismo para una redirección más eficiente (por ejemplo para búsqueda de archivos). Estas tablas contienen elementos del tipo clave y el valor que se distribuye de forma más o menos equitativa entre los nodos de la red. La idea es que a partir de la clave puedas dirigirte directamente al nodo que contiene el valor. Existen diferentes modos de implementarlas, por ejemplo el API Chord, Pastry…

Supongamos que estamos desarrollando una aplicación cliente/servidor capaz de proporcionar la fecha y hora actual al cliente mediante el API de RMI.
La fecha y hora actual que debe proporcionar el servidor, puede ser generada usando la clase Date
Completar la definición de la interfaz remota:

```
Import java.rmi.*;

Public Interface DayTimeINterface extends Remote{
        Public String getFecha() throws RemoteException;
}
```

Completar el fichero con la implementación:

```
Public class DayTimeImpl implements DayTimeInterface{
        Public String getFecha() throws RemoteException{
                Date hoy = new date();
                String fecha = hoy.toString();
                Return fecha;
        }
}
```

Supongamos que el siguiente fragmento de código aparece en el servidor para exportar la implementación del objeto: proporcionar la sentencia que falta para declarar la referencia a exportedObj y asignarle un valor.

```
startRegistry(RMIPortNum);
registryURL = "rmi://rmi.usc.es:12345/daytime";
DayTimeImpl exportedObj = new DayTimeImpl();
Naming.rebind(registryURL, exportedObj);
```

Enumera los ficheros que deben de estar presentes en la parte del cliente en tiempo de ejecución.
Cliente.class, DayTimeInterface.class, DayTimeImpl_stub.class

Completar el código para el programa cliente. El programa puede asumir que el registro RMI se ejecuta en rmi.usc.es en el puerto 12345.

```
public static void main(String args[ ]) {
        try {
                String URLRegistro = “rmi://rmi.usc.es:12345/daytime”;
                DayTimeInterface= (DayTimeInterface ) Naming.lookup(URLRegistro);
        } // end try
        catch (Exception e) {
        System.out.println("Exception in DaytimeClient: " + e);
}
```

Supongamos que deseamos que los clientes sean capaces de registrarse para callback, de forma que cada vez que un cliente contacta con el servidor de tiempo, todos los clientes registrados reciben el tiempo enviado a ese cliente. Escribe la interfaz remota del cliente que permitiría dicho callback.

```
Public interface interfazCliente extends Remote{
        Public void notificar(String fecha);
}
```

Reescribe la interfaz remota del servidor que permitiría dicho callback – no es necesario contemplar la eliminación de un registro de callback

```
Public Interface DayTimeINterface extends Remote{
        Public String getFecha(InterfazCliente cliente) throws RemoteException;
}
```

Describe con palabras los cambios que serían necesarios en la implementación del servidor para acomodar las callbacks.
El servidor ahora debe mantener un vector/array/lista con las referencias de los clientes registrados para callback. Cada vez que uno llame al método, si no esta en la lista lo añade, y recorre esta lista invocando el método dado

En el primer apartado del ejercicio anterior, describe como sería el archivo idl si pretendiésemos realizar la aplicación utilizando CORBA.

```
Module DayTimeApp{
        Interface ClientInterface{
                String notificar();
        }
        Interface DayTimeInterface {
                Void getFecha(ClientInterface client);
        }
}
```

***En 2019 primera convocatoria Repitió todas las preguntas menos esta:**
Para que se utiliza Protege? Que plugin se suele incluir en Protege para desarrollar aplicaciones en Jade?
Se emplea para la creación de Ontologías de manera sencilla.
Se incluye el plugin BeanGenerator que crea las clases Java necesarias para incluir la ontología en una aplicación java.