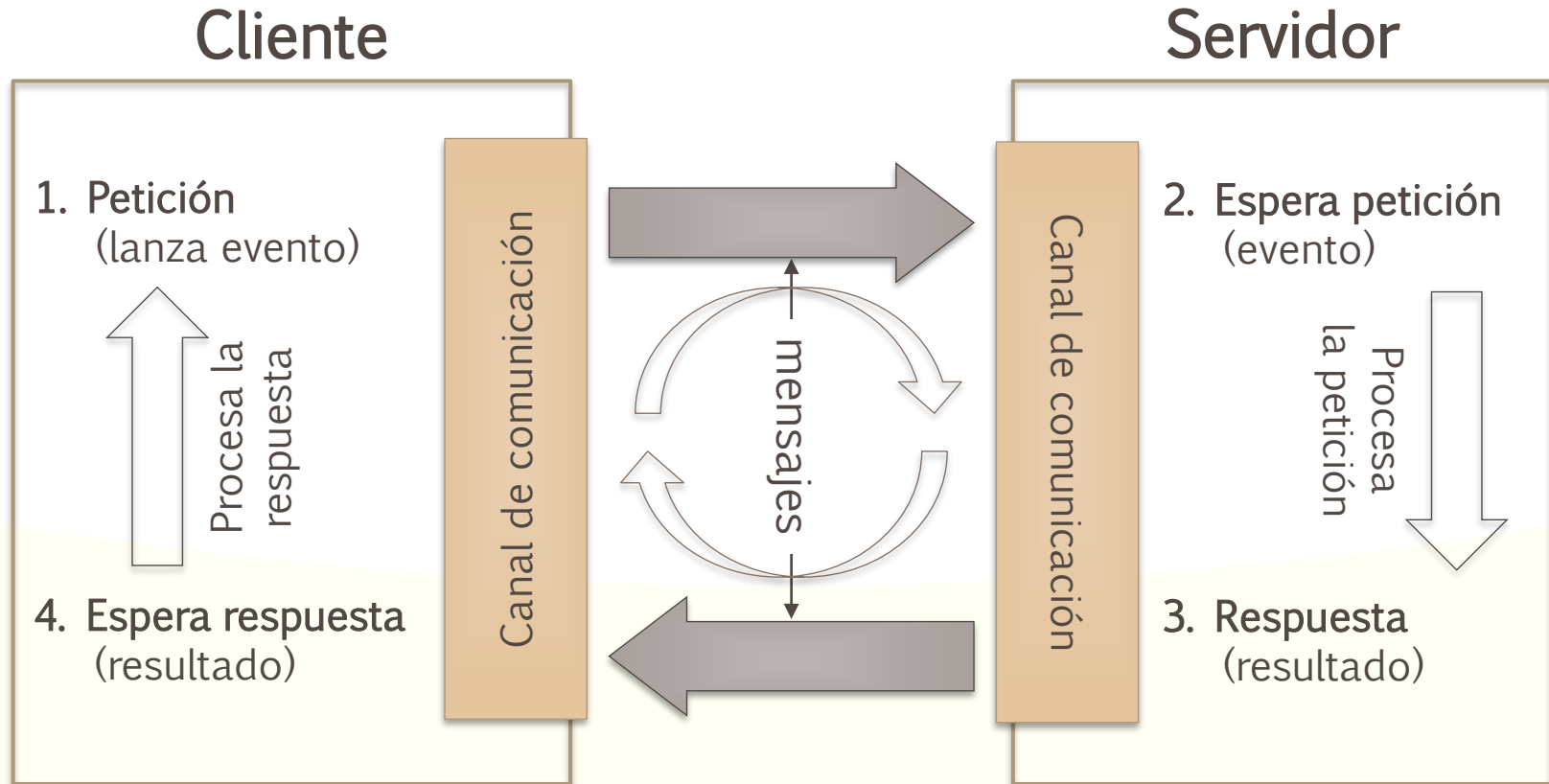


PRÁCTICAS DE POE

Primera sesión de prácticas



Aplicación cliente-servidor (1)

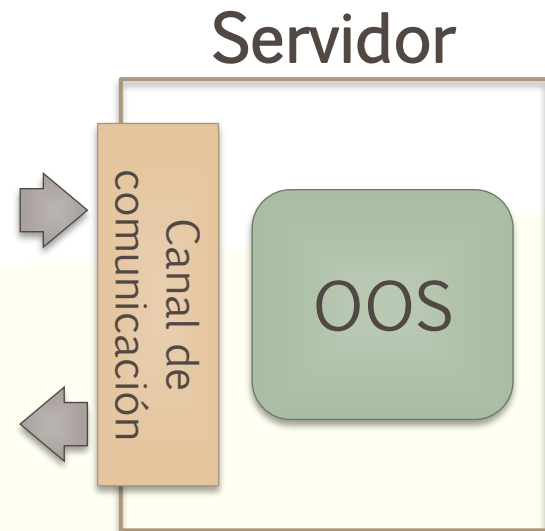


Aplicación cliente-servidor (2)

■ Servidor

- Un programa que ofrece un **servicio** a uno o más clientes.
 - El **servicio** es la *funcionalidad* ofrecida por el servidor, la cual puede estar encapsulada en un objeto. En ese caso, éste recibe el nombre de **objeto de operaciones de servicio (OOS)**.

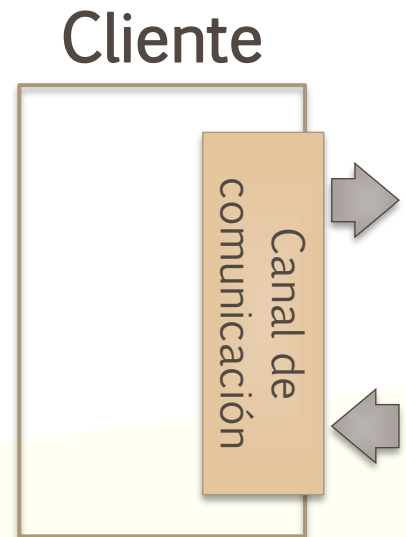
- Establece un **canal de comunicación** para recibir peticiones del **cliente** y enviar respuestas a éste
- Crea el **objeto de operaciones de servicio**
- Su ejecución es continua, estando siempre a la espera de que se conecte un **cliente**



Aplicación cliente-servidor (3)

■ Cliente

- Un programa que se conecta a un **servicio** ofrecido por un **servidor**
- Establece el **canal de comunicación** para enviar peticiones al **servidor** y recibir respuestas de éste
- Interactúa con el **OOS** a través de la red, mediante intercambio de mensajes
 - Para ello se requiere la especificación de la funcionalidad que expone el **servicio**; esto es, la **interfaz de servicio**
- A diferencia del servidor la ejecución no es continua. El cliente finaliza en algún momento, desconectándose del servicio.





Aplicación cliente-servidor (4)

- Visto como un Sistema Basado en Eventos
 - Los **clientes** invocan las operaciones del servicio (lanzan eventos y los envían al servidor)
 - El **servidor** invoca las operaciones de servicio que le llegan de los clientes (equivalente del distribuidor de eventos)
 - Los métodos del **OOS** son los manejadores de eventos
 - Biblioteca POE_library.jar
 - Para facilitar la realización de prácticas mediante este modelo, se proporciona una biblioteca de interfaces y clases de Java
-

Ejemplo (1)

- Un servicio que saluda
 - Descarga del CV la biblioteca y el zip con los archivos para la sesión:

Archivo	Descripción
comun/ISaludador.java	Interfaz del servicio (dos operaciones opcionales)
comun/AccionNoPermitida.java	Implementa la excepción AccionNoPermitida
servidor/SaludadorOOS.java	Implementa la interfaz de servicio (sólo saluda())
servidor/Servidor.java	Programa servidor

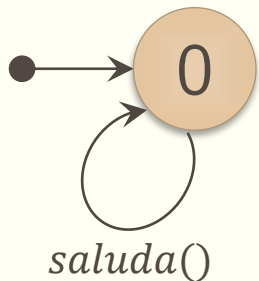
- Crea en el Eclipse un proyecto Java de nombre *poe-01* y copia las carpetas *comun* y *servidor* al *src* del proyecto.
 - *comun* y *servidor* pasarán a ser *packages*
-

Ejemplo (2)

- Configuración del proyecto *poe-01*
 - Hay que indicar que el proyecto utiliza la biblioteca *POE_library.jar*:
 - Pon el ratón sobre el nombre del proyecto y abre el menú contextual (botón derecho del ratón) para seleccionar Build Path y luego, en el submenú, Configure Build Path...
 - Finalmente, en el cuadro de diálogo selecciona la pestaña Libraries para añadir la biblioteca con el botón Add External JARs... y, posteriormente, pulsa el botón Apply and Close.
 - Ahora ya no debería haber errores en el proyecto.
-

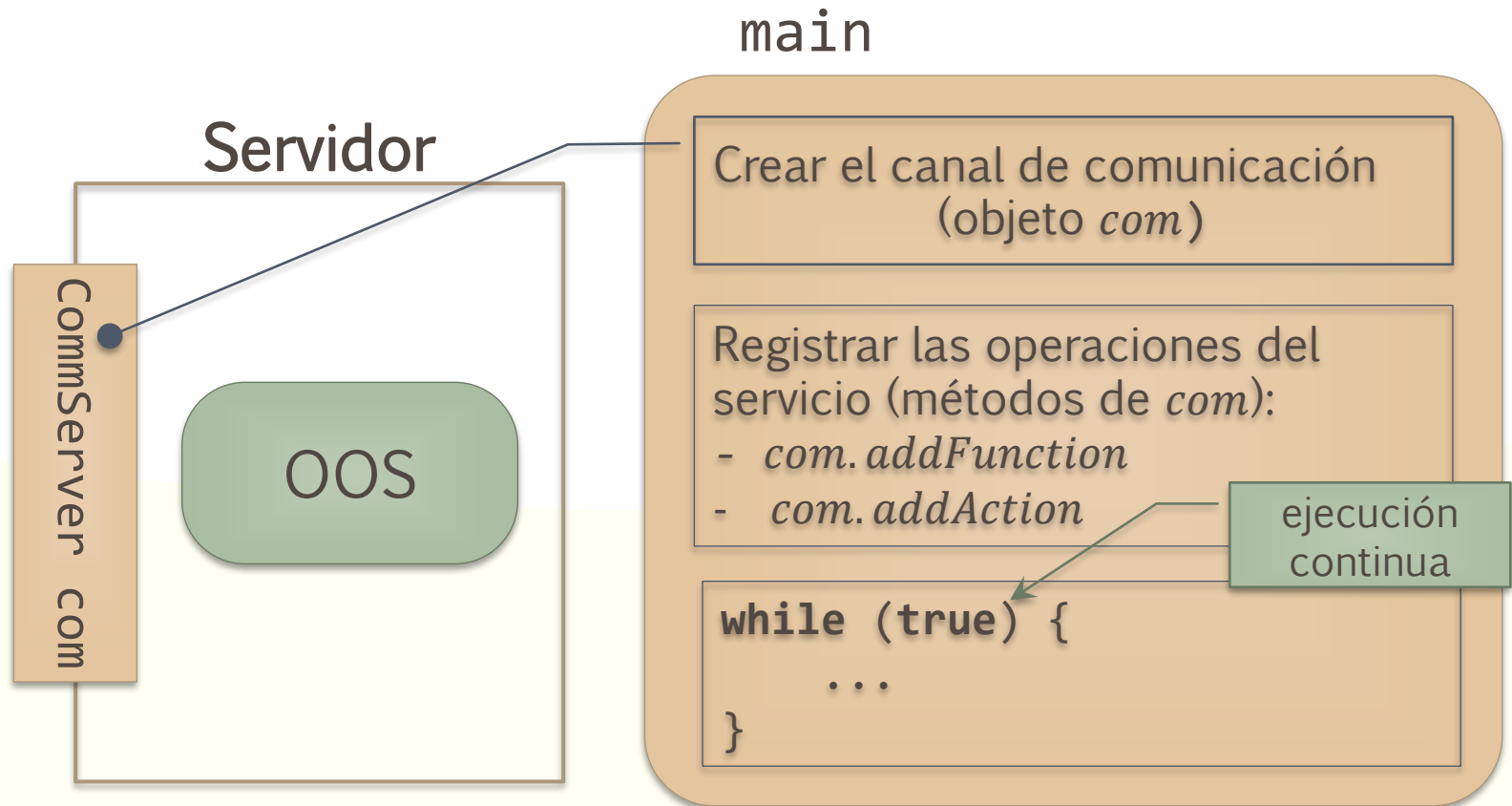
Ejemplo (3)

- Abre la interfaz de servicio y su implementación
 - Todas las interfaces de servicio tienen que extender la interfaz de la biblioteca *lib.DefaultService*.
 - Esta interfaz proporciona una implementación por defecto del método *close()* para cerrar de forma ordenada el OOS.
 - Más adelante se verá en que casos es necesario refinar la implementación por defecto.
 - Como se puede ver el servicio únicamente implementa la operación obligatoria *saluda()*.
 - En estas condiciones el servicio dispone de un único estado, el estado inicial, donde se puede lanzar varias veces el evento *saluda()*
 - Estado de datos: el *String str*
 - En este ejemplo tan simple, el estado de control es innecesario, pero también se ha incluido: *estado* (por eso hay un aviso)



Práctica. Programa Servidor (1)

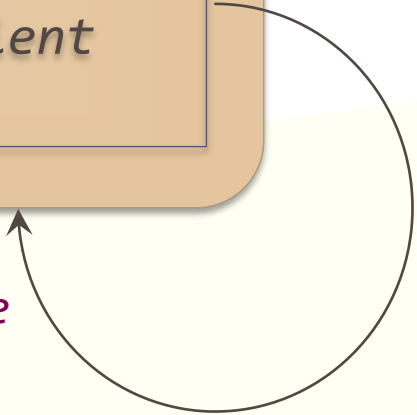
- Ahora abre el programa **servidor**



Servidor

1. Esperar por un cliente e identificar éste (*idClient*). Operación bloqueante
2. Crear el objeto de servicio para el cliente *idClient*. Un OOS por cliente
3. Intercambiar mensajes con el cliente *idClient*. Ciclo petición-respuesta
4. Cerrar el OOS del cliente *idClient*

indefinidamente
while (true)



Biblioteca POE_library.jar (1)

- Intercambio de mensajes (cliente \Leftrightarrow servidor)
 - Tanto los *mensajes* como los *canales de comunicación* son objetos.
 - Los *mensajes* son instancias de la clase ProtocolMessages
 - El *canal de comunicación* del *cliente* es una instancia de la clase CommClient
 - El *canal de comunicación* del *servidor* es una instancia de la clase CommServer, una instancia por servicio.
 - Habitualmente un servidor ofrecerá un servicio, pero no está restringido a ello

Biblioteca POE_library.jar (2)

- Clase ProtocolMessages

- Mensajes hacia el servidor (*petición*) constan de:
 - Un **identificador**: como por ejemplo, la cadena de caracteres (String) del nombre de la operación de servicio que solicita un cliente, o un indicativo de desconexión de éste.
 - Un **array de Object**: con los argumentos requeridos (datos) para ejecutar la operación de servicio.
- Mensajes hacia un cliente (*respuesta*), constan de:
 - Un **identificador**: una cadena de caracteres para indicar que un evento se ha ejecutado correctamente (OK), o bien, que se ha producido una excepción.
 - Un **array de Object**: si el evento produce un resultado (o una excepción), éste se guarda en la primera componente.



Biblioteca POE_library.jar (3)

- La clase debe ser `Serializable`
 - El intercambio de mensajes se realiza por la red Internet, necesariamente, como una secuencia de bytes
 - Métodos:
 - `ProtocolMessages(String id, Object... args)`
 - Crea un mensaje como se indicó en la transparencia previa
 - `String getID()`
 - Retorna el identificador de este mensaje
 - `Object[] getArgs()`
 - Retorna los argumentos de este mensaje
 - `String toString()`
 - Retorna la cadena: `ID(arg0, arg1,...)`, siendo ID el valor retornado por `getID()` y `arg0, arg1,...` los valores de las componentes de `getArgs()`
-

Servicio

- Interfaz de operaciones del servicio
 - La interfaz que especifica las operaciones del servicio
 - *Isaludador*
 - Implementación de la interfaz
 - Clase *SaludadorOOS*
 - Área de datos
 - Estado de control (un entero que, para este ejemplo, puede ser 0 o 1).
 - Estado de datos (información del sistema)
 - Un *String*, el saludo
 - El OOS es una instancia de esta clase
-

Biblioteca POE_library.jar (4)

■ Clase CommServer

- Una instancia de esta clase establece el canal de comunicación del servidor y ofrece un **servicio** a los **clientes** (mediante el OOS)

■ Métodos

- `CommServer()`: crea el canal de comunicación en el servicio por defecto
 - `processEvent(id, obj, msg)`: el servidor procesa el mensaje `msg` recibido del cliente `id` para el objeto de operaciones de servicio `obj`
 - `addFunction(idOp, f)` y `addAction(idOp, f)`: registran una operación del servicio (función o acción, respectivamente), asociando su identificador (su nombre) con su función/acción anónima. Tal y como se verá a continuación.
-

Biblioteca POE_library.jar (5)

- Invocación automática de las operaciones del OOS
 - Es necesario registrar cada operación indicando como ha de evaluarse ésta mediante una función/acción anónima que se asocia al **identificador** del mensaje (por ejemplo, el mismo nombre de la operación).
 - Así, la operación de servicio:
 - *TipoResultado op(T0 arg0, T1 arg1, ...)*
 - Se podría registrar como:
addFunction("op", (o, x) -> o.op(x[0], x[1], ...))
 - Pero en la función anónima se requieren conversiones de tipo explícitas (*castings*) porque en la función que procesa la invocación todo está declarado como de tipo Object (para que se pueda utilizar con cualquier tipo de parámetro)

Biblioteca POE_library.jar (6)

- Otras operaciones de CommServer
 - CommServer(idService): crea un canal de comunicación en el servicio especificado
 - waitForClient(): **espera** la conexión de un cliente, retornando el identificador de éste (operación bloqueante)
 - closed(id): determina si el cliente **id** se ha desconectado
 - waitEvent(id): el servidor **espera** por un mensaje del cliente **id** (un evento). Retorna el mensaje recibido.
 - sendReply(id, msg): envía el mensaje **msg** de respuesta al cliente **id**
 - activateMessageLog(): activa el registro de mensajes del servicio que por defecto está desactivado (*operación opcional*)
-

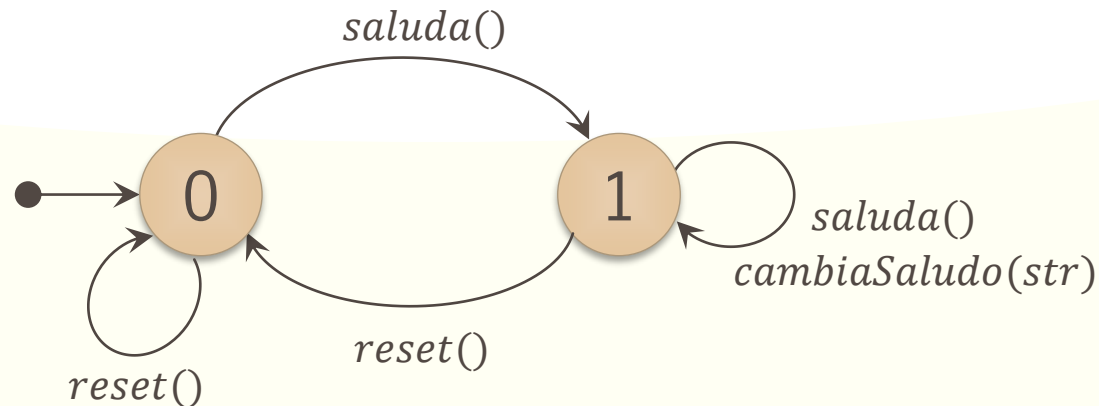
Biblioteca POE_library.jar (7)

- Clase Trace (*package* Optional)
 - Consta de funciones de clase (*static*) que facilitan el seguimiento de las distintas acciones que realiza el servidor.
 - La traza se muestra directamente en la salida estándar
 - Por defecto, el seguimiento está desactivado
 - Principal método de clase
 - `void activateTrace(CommServer com):` activa el seguimiento en el servidor.

Se recomienda que siempre se active la traza del servidor (después de instanciar su canal de comunicación) porque internamente la biblioteca incluye algunos seguimientos que os informarán en consola de algunas de las acciones que realiza el servidor cuando se está ejecutando

Ejemplo (4)

- Saludador más funcional
 - Realiza los cambios que sean necesarios para que el servicio ofrecido por el servidor también disponga de las operaciones opcionales: *cambioSaludo(str)* y *reset()*.
 - El evento *cambioSaludo(str)*, sólo se podrá lanzar si previamente se ha lanzado el evento *saluda()*, lo que da lugar al siguiente grafo de estados:



Biblioteca POE_library.jar (8)

■ Clase CommClient

- Una instancia de esta clase establece la comunicación del cliente con un servicio del servidor

■ Métodos

- `CommClient()`: crea una conexión con la propia máquina y servicio por defecto
 - `CommClient(server, idService)`: crea una conexión con el servidor y servicio indicados
 - `sendEvent(msg)`: envía el mensaje especificado al servidor
 - `waitReply()`: espera del servidor un mensaje de respuesta a una petición previa (operación bloqueante)
 - `processReply(msg)`: el cliente procesa un mensaje de respuesta del servidor (que puede ser una excepción)
-



Biblioteca POE_library.jar (9)

- `disconnect()`: el cliente se desconecta del servidor
- `activateMessageLog()`: activa el registro de mensajes del cliente.

Ejemplo (5)

- Programa Cliente1
 - Lanza 3 veces el evento *saluda()* y muestra en consola el saludo del servicio.
 - Activa la traza del servidor y ejecuta éste
 - En otra consola del Eclipse ejecuta el cliente
 - Añade al cliente dos funciones para lanzar los eventos *cambiaSaludo(str)* y *reset()* y procesar las respuestas, si las hubiera.
 - Desde el main, lanza los siguientes eventos en el orden indicado y muestra en consola las posibles respuestas:
 - *cambiaSaludo("Cambiando el saludo")* , *saluda()* , *saluda()* , *cambiaSaludo("Otro cambio de saludo")* , *saluda()* , *reset()* , *saluda()*
 - Antes de ejecutar, ¿cuál debería ser el saludo del servicio para el último evento?
-