Universidad Autónoma de Madrid
Escuela Politécnica Superior
Grado en Ingeniería Informática
Sistemas Informáticos II – Curso 2018-2019

Mecanismos de comunicación facilitados por el NOS: Sockets, RPCs, Corba y Java RMI.

Objetivo

El objetivo de este ejercicio es profundizar en algunos de los servicios de transporte utilizados por el Sistema Operativo de Red (NOS, *Network Operating System*). En concreto, la comunicación por sockets, las llamadas a procedimientos remotos (RPC, *Remote Procedure Calls*), la arquitectura Corba y la comunicación mediante objetos Java RMI (*Remote Method Invocation*).

Tareas

En equipos de 4 ó 5 personas, se pide realizar una presentación sobre el trabajo desarrollado. Dicho trabajo constará de dos partes. La primera parte es común a todos los equipos y se analizará la implementación en los cuatro mecanismos de comunicación cliente-servidor de un servicio de operaciones de números enteros. Para ello, se proporcionará el código que implementa el servicio y se deberán responder a una serie de cuestiones sobre el mismo. En la segunda parte, el profesor asignará a cada uno de los equipos un sistema de transporte (sockets, RPCs, Corba o Java RMI) y cada equipo deberá proponer un nuevo servicio para el que el sistema de transporte asignado sea especialmente idóneo. Se deberá discutir razonadamente por qué la arquitectura cliente-servidor asignada es apropiada para implementar este tipo de servicio e indicar en líneas generales los principales cambios que habría que realizar en el código proporcionado. No es necesario implementar estos cambios en el código.

Planificación:

El ejercicio implicará un total de 2 horas de clase de teoría y 2 horas de trabajo fuera de clase. Se seguirá la siguiente planificación:

- **Miércoles 13 de febrero / jueves 14 de febrero:**
  - Primera parte. Análisis del código proporcionado (1 hora, clase de teoría)

- **Semana del 13 de febrero – 20 de febrero / 14 de febrero – 21 de febrero:**

  - Primera parte. Finalización de los ejercicios correspondientes a la primera parte (30 minutos, fuera de clase).
  - Segunda parte. Diseño de una servicio apropiado para la comunicación cliente-servidor asignada al equipo (45 minutos, fuera de clase)
  - Preparación de la presentación y exposición (45 minutos, fuera de clase)
  - Entrega de las transparencias en Moodle en formato pdf.

- **Miércoles 20 de febrero / Jueves 21 de febrero:**
  - Exposición oral de 10 minutos por parte de 4 alumnos de equipos diferentes (1 hora, clase de teoría).

Evaluación:

Sólo se evaluará el contenido de las transparencias, no de las exposiciones orales.

- **Contenido:** la presentación cumple con los objetivos del trabajo, proporcionando respuestas justificadas a las preguntas (ni le faltan contenidos relevantes, ni le sobran contenidos irrelevantes).

- **Forma:** la presentación es concisa y resuelve de forma clara y sencilla los ejercicios propuestos.

La calificación de cada equipo será la calificación dada por el profesor.

Contenido de la presentación

La presentación debe contener las respuestas a las preguntas que se plantean en la primera parte de la actividad así como la propuesta de servicio adecuado al servicio de transporte asignado, justificando la elección e indicando los principales cambios en el código.

La presentación deberá constar de un **máximo de 18 transparencias** en formato PDF.

Primera parte:

Se deberá preparar una pequeña presentación que conteste a cada una de las cuestiones planteadas para cada mecanismo de comunicación en la primera parte de la actividad. Además, se deberá mostrar al final una diapositiva con la tabla comparativa entre los mecanismos de comunicación (ver más adelante dicha tabla).

Para responder a cuestiones referentes al código o la ejecución de los programas, se pueden utilizar fragmentos de código o capturas de pantalla para justificar la respuesta, siempre debidamente comentados. En la tabla comparativa, cada una de las celdas debe contener un breve comentario que resuma las características para cada mecanismo cliente-servidor y cada aspecto a analizar. Un ejemplo del tipo de respuestas a rellenar en esta tabla puede ser la diapositiva “Comparación MOM - RPC” del tema 1.

Aunque esta presentación debe cubrir todas las preguntas formuladas sobre todos los mecanismos de comunicación, deberá ser más detallada para el mecanismo de comunicación asignado por el profesor. Para ese mecanismo, además de contestar a las preguntas formuladas en la actividad, se deberá explicar en detalle el funcionamiento de los programas facilitados.

Como referencia, esta parte debería ocupar unas 15 transparencias.

Segunda parte:

Se debe proponer y describir un servicio adecuado al mecanismo de comunicación asignado, justificando la elección, e indicando por qué es adecuado utilizar ese mecanismo y no otro. También se deberá informar de los principales cambios que habría que llevar a cabo en el código facilitado para implementar ese servicio.

Como referencia, esta sección debería ocupar 3 transparencias.

Parte I: Análisis de un servicio de operaciones con números enteros implementado mediante sockets, RPCs, Corba y Java RMI.

La mayoría de cuestiones de esta parte de la actividad no requieren la ejecución del código proporcionado, si no que se pide un análisis del mismo.

Para la ejecución del código se utilizará la misma máquina virtual a utilizar en las prácticas de la asignatura, asignando las IPs **10.X.Y.Z** de cada máquina virtual como sigue:

> **X=**Grupo de teoría. Para ello deberá escogerse en el asistente el grupo de prácticas 3001 en el caso del turno de mañana y 3002 en el caso del turno de tarde.
> **Y**=Número de grupo para la actividad 1. A asignar por el profesor.
> **Z**=1 (máquina virtual para el servidor) / 2 (máquina virtual para el cliente) / 3 (máquina virtual para el ORB de Corba)

Recordar que es necesario ejecutar el script `virtualip.sh` en el PC host para poder comunicarse con las máquinas virtuales.

En cada máquina virtual, instalar los siguientes paquetes:

```
sudo apt-get update
sudo apt-get install gcc
sudo apt-get install portmap
```

Nota:

La ubicación del repositorio de la distribución de linux de la máquina virtual ha cambiado. Para la correcta instalación de los paquetes es necesario ejecutar los comandos:

cat /etc/apt/sources.list | sed s/us.archive/old-releases/g > /tmp/hola

sudo mv /tmp/hola /etc/apt/sources.list

Tras este cambio la instalación de los paquetes gcc y portmap debería ejecutarse de forma correcta sin dar error.

El servicio de números enteros implementado puede realizar cuatro operaciones (suma, resta, multiplicación y división) que involucran dos operandos enteros. Cada una de estas operaciones se representa con los números 0, 1, 2 y 3. Por tanto, las cuatro implementaciones del servicio (sockets, RPCs, Corba, Java RMI) necesitan como parámetros tres enteros (operador y dos operandos). Dependiendo de la implementación, otros parámetros también pueden ser necesarios.

SOCKETS

Como primera aproximación, implementaremos el servicio de operaciones con números enteros utilizando la librería de sockets en C. En este caso, los ficheros a editar son un fichero con el código fuente del servidor (**`server.c`**) y un fichero con el código fuente del cliente (**`client.c`**). Estos ficheros se encuentran en el subdirectorio **`sockets`** del código fuente adjunto a la actividad.

Los pasos necesarios para la ejecución del servicio basado en sockets son:

Paso 0: implementar **`server.c`** y **`client.c`**

Servidor:

1. En la máquina virtual del servidor (10.X.Y.1), generar el subdirectorio “sockets”.
2. Copiar los ficheros necesarios en la máquina virtual del servidor:
   ```
   >>scp <ruta codigo fuente>/sockets/server.c si2@10.X.Y.1:/home/si2/sockets
   ```

3. Editar el fichero server.c indicando a la función inet_addr la dirección IP de la máquina virtual del servidor (10.X.Y.1).
4. Compilar el código fuente del servidor.
   ```
   >> gcc server.c -o server
   ```

5. Ejecutar el servidor
   ```
   >> ./server
   ```

Cliente:

1. En la máquina virtual del cliente (10.X.Y.2) generar el subdirectorio “sockets”.
2. Copiar los ficheros necesarios en la máquina virtual del cliente:
   ```
   >> scp <ruta codigo fuente>/sockets/client.c
   si2@10.X.Y.2://home/si2/sockets
   ```

3. Compilar el código fuente del cliente.
   ```
   >> gcc client.c -o client
   ```

6. Ejecutar el cliente
   ```
   >> ./client
   ```

   Ejemplo:

   ```
   si2@si2srv02:~/sockets$ ./client
   Usage: ./client <direccion IP servidor>
   si2@si2srv02:~/sockets$ ./client 10.3.99.1
   String received from the server:
   Bienvenido a la libreria de numeros enteros entre 0 y 9: <type>(0: suma / 1:resta / 2: multiplicacion / 3: division)<operador1><operador2>
   **************

   023
   String received from the server:
   5
   **************
   ```

Ejercicios:

1. Código fuente del servidor (server.c). Identificar la secuencia de funciones de sockets utilizadas en la implementación del servidor. ¿Qué información debe proporcionarse en cada uno de los pasos?
2. Código cliente (client.c). ¿Cómo se conecta al servidor? ¿Qué información debe conocerse? Identificar la secuencia de funciones de sockets utilizadas en la implementación del cliente.
3. **Opcional**. Ejecutar el comando netstat -a antes y después de ejecutar el servidor. ¿Cómo se detecta que el servidor está escuchando? ¿Qué información especificada en el fichero fuente se observa?
4. Discutir características de sockets en términos de transparencia de ubicación,

transparencia de acceso, facilidad de implementación para servidor y cliente y describir servicios en los que el empleo de sockets sería adecuado y servicios en los que no sería la opción más apropiada. Completar la tabla.

SUN RPCs

Vamos a implementar el servicio de operaciones con números enteros utilizando SUN RPCs. En este caso, los ficheros a editar son un fichero de interfaz (**`int_library.x`**), un fichero que implemente la funcionalidad del servidor (**`server.c`**) y un fichero que implemente la funcionalidad del cliente (**`client.c`**). Como se verá más adelante, todos los demás ficheros que se utilizan para la correcta generación de cliente y servidor son generados automáticamente.

Los pasos necesarios para la ejecución del servicio basado en RPCs son:

Paso 0: Editar el fichero de interfaz **`int_library.x`** e implementar **`server.c`** y **`client.c`**

Servidor:

1. En la máquina virtual del servidor (10.X.Y.1), generar el subdirectorio “rpcs”.
2. Copiar el código necesario en la VM con el servidor: fichero de especificación de la interfaz (`int_library.x`) y fichero con la implementación del servidor (`server.c`)
   ```
   >> scp <ruta codigo fuente>/rpcs/server.c si2@10.X.Y.1:/home/si2/rpcs
   >> scp <ruta codigo fuente>/rpcs/int_library.x si2@10.X.Y.1:/home/si2/rpcs
   ```
3. Desde la máquina virtual del servidor, generar de manera automática (`rpcgen`) el server stub, ficheros de cabeceras y declaraciones XDR.
   >> rpcgen int_library.x
4. Desde la máquina virtual del servidor, generar el ejecutable del servidor.
   ```
   >> gcc server.c int_library_svc.c int_library_xdr.c -o server
   ```
5. Ejecutar el servidor
   ```
   >> ./server
   ```

Cliente:

1. En la máquina virtual del cliente (10.X.Y.2), generar el subdirectorio “rpcs”.
2. Copiar el código necesario en la máquina virtual con el cliente: fichero de especificación de la interfaz (`int_library.x`) y fichero con la implementación del cliente (`client.c`)
   ```
   >> scp <ruta codigo fuente>/rpcs/client.c si2@10.X.Y.2:/home/si2/rpcs
   >> scp <ruta codigo fuente>/rpcs/int_library.x si2@10.X.Y.2:/home/si2/rpcs
   ```
3. Desde la máquina virtual del cliente, generar de manera automática (`rpcgen`) el client stub, ficheros de cabeceras y declaraciones XDR
   ```
   >> rpcgen int_library.x
   ```
4. Desde la máquina virtual del cliente, generar el ejecutable del cliente.
   ```
   >> gcc client.c int_library_clnt.c int_library_xdr.c -o client
   ```
5. Ejecutar el cliente.
   ```
   >> ./client
   ```

Ejemplo:

```
si2@si2srv02:~/rpcs$ ./client
usage:./client <hostname> <tipo de operacion (0:suma / 1: resta /
2: multiplicacion / 3:division)> <operador 1> <operador 2>
si2@si2srv02:~/rpcs$ ./client 10.3.99.1 0 2 3
Result:5
```

Ejercicios:

1. Observar los ficheros generados por rpcgen y comentar su funcionalidad en base a lo visto en las clases de teoría.
2. Definición de la interfaz (fichero int_library.x). ¿Por qué es necesario agrupar todos los parámetros de entrada en una estructura? Observar que es necesario indicar el número de programa, versión y procedimiento.
3. Ejecutar el comando rpcinfo -p antes y después de ejecutar el servidor. ¿Cómo se detecta que el servidor está escuchando? ¿Qué datos especificados en el fichero .x se reflejan?
4. Código fuente del servidor (server.c). Observar que el registro del servicio en el portmapper es transparente para el programador, que solo debe preocuparse de implementar la funcionalidad del servicio. El código encargado de realizar el registro con el portmapper se encuentra en la función main del fichero int_library_svc.c. Identificar las partes de este código responsables de:
   - Crear sockets UDP y TCP.
   - Dar de alta el servicio en el portmapper. ¿Qué información debe proporcionarse?
   - Llamada a la función dispatcher encargada de procesar las llamadas del cliente.

   En la función *dispatcher* identificar para qué sirve el número de procedimiento declarado en el fichero .x. Localizar las partes de código encargadas del marshalling/unmarshalling de los argumentos y el envío de respuesta al cliente.

5. Código cliente (client.c). ¿Cómo se conecta al servidor?¿Qué información debe conocer el cliente?
6. Discutir características de RPCs en términos de transparencia de ubicación, transparencia de acceso, facilidad de implementación para servidor y cliente y describir servicios en los que el empleo de RPCs sería adecuado y servicios en los que no sería la opción más apropiada. Completar la tabla.
7. **Opcional**. Modificar el prototipo de la función `PROCESAPETICION` para que en lugar de recibir una estructura reciba tres enteros indicando el tipo de operación y los dos operadores. Compilar el código. ¿Qué sucede? ¿Por qué?

CORBA

La librería de números enteros se implementará utilizando CORBA. El servidor se implementará en Java y el cliente en Python. En este caso, los ficheros a editar son similares a los ficheros a editar en RPCs. Un fichero de interfaz (**`intLibrary.idl`**), un fichero con el código fuente que implementa la funcionalidad del servidor (**`server.java`**) y un fichero con el código fuente que implementa la funcionalidad del cliente (**`client.py`**). Como se verá más adelante, todos los demás ficheros que se utilizan para la correcta generación de cliente y servidor son generados automáticamente.

Los pasos necesarios para la ejecución del servicio basado en Corba son:

Paso 0: Editar el fichero de interfaz **`intLibrary.idl`** e implementar **`server.java`** y **`client.py`**

Bus de comunicación entre objetos (*Object Request Broker*, ORB):

1. Arrancar ORB en una tercera máquina virtual (10.X.Y.3). Para más información, es útil consultar man (`man orbd`).
   `>> orbd -ORBInitialPort 1050 -ORBInitialHost localhost &`
   `>> netstat -l -t` → observar que el orbd está escuchando por el puerto 1050.

Servidor:

1. En la máquina virtual del servidor (10.X.Y.1), generar el subdirectorio “corba”.
2. Copiar los ficheros necesarios a la máquina virtual del servidor.

```
>> scp <ruta codigo
fuente>/Corba/IntLibrary.idlsi2@10.X.Y.1:/home/si2/corba
>> scp ./corba/server.java si2@10.X.Y.1:/home/si2/corba
```

3. En la máquina virtual del servidor, compilar la interfaz IDL utilizando el compilador de interfaces para Java (*language mappings*), puesto que el servidor está implementado en este lenguaje.
   `>> idlj -fall IntLibrary.idl`

4. En la máquina virtual del servidor, compilar el servidor a partir de los ficheros generados automáticamente por el compilador de interfaces y el código fuente del servidor.
   `>> javac *.java ./IntLibraryApp/*.java`

5. Arrancar el servicio indicando la IP donde se encuentra el ORB.
   `>> java server -ORBInitialPort 1050 -ORBInitialHost 10.X.Y.3`

   Ejemplo:

```
si2@si2srv01:~/corba$ java server -ORBInitialPort 1050 -
ORBInitialHost 10.3.99.3
IntLibrary Server ready and waiting ...
IOR NameServer:
IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e6
16d696e67436f6e746578743a312e300000000001000000000000000a80001020000
00000a31302e332e39392e330004190000003cafabcb0000000022000003e800000
0010000000000000002000000008526f6f74504f41000000000c4e616d6553657276
6963650000000034e433014000000030000000100000020000000000010001000
0000205010001000100200001010900000001000101000000002600000002000200
000000003000000140000000000000000a31302e332e39392e3300041a
```

   El programa servidor muestra por pantalla el IOR del servidor de nombres del ORB y también lo guarda en un fichero IOR_ns.txt. Recordar que el IOR (Interoperable Object Reference) es la referencia que identifica de manera única a un objeto remoto en Corba. El objeto que traduce nombres a referencias a objetos remotos en CORBA es también un objeto remoto. Se utilizarán los servicios de dicho objeto para obtener una referencia al objeto remoto que implementa el servicio.

Cliente:

1. En la máquina virtual del cliente (10.X.Y.2), generar el subdirectorio “corba”.
2. Copiar los ficheros necesarios a la máquina virtual del servidor.

```
>> scp ./corba/IntLibrary.idl si2@10.X.Y.2:/home/si2/corba
>> scp ./corba/client.py si2@10.X.Y.2:/home/si2/corba
```

3. En VM cliente, instalar los paquetes python-omniorb-omg y omniidl4-python. OmniORB implementa un ORB que cumple las especificaciones de CORBA e incluye *bindings* para Python.
```
>> sudo apt-get install python-omniorb-omg omniidl4-python
```

4. Compilar la interfaz IDL utilizando el compilador de interfaces para python, puesto que el cliente está implementado en este lenguaje.

```
>> omniidl -bpython IntLibrary.idl
```

5. Ejecutar el cliente. Para ello es necesario conocer el IOR del servidor de nombres donde se ha registrado el servidor.
```
>> python client.py -ORBInitRef NameService=`cat IOR_ns.txt`
```

Ejemplo:

```
Uso: python client.py -ORBInitRef NameService=`cat IOR_ns.txt`
<tipo de operacion (0:suma / 1: resta / 2: multiplicacion /
3:division)> <operador 1> <operador 2>

si2@si2srv02:~/corba$ python client.py -ORBInitRef NameService=`cat
IOR_ns.txt` 2 5 7
El resultado de la operacion es: 35
```

El anterior comando pasa al ORB información sobre la referencia al objeto "NameService" que se utilizará para resolver nombres a referencias a objetos remotos. (Para no pegar todo el IOR como un argumento usamos `cat IOR_ns.txt`. Es necesario copiar previamente el fichero IOR_ns.txt generado por el servidor (10.X.Y.1) en el directorio "corba" del a máquina virtual del cliente (10.X.Y.2). La copia se puede llevar a cabo utilizando el comando scp (man scp para más información).

Ejercicios:

1. Abrir el fichero IntLibrary.idl que contiene la declaración de la interfaz y observar que se declaran tanto el módulo como la interfaz. ¿Qué representaría el objeto remoto Corba y cuáles serían los métodos de este objeto?
2. **Opcional.** Observar los ficheros generados por los compiladores de interfaces idlj y omniidl. Comentar su funcionalidad en base a lo visto en las clases de teoría. Las siguientes referencias también pueden ser de utilidad:
    - Implementación CORBA en Java: http://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html
    - Implementación CORBA en Python: http://omniorb.sourceforge.net/omnipy3/omniORBpy/omniORBpy002.html

3. Fichero fuente servidor (server.java). Identifica y analiza los fragmentos de código responsables de:
- Inicialización del ORB
- Obtención del indentificador POA (Portable Object Adapter) raíz
- Activación del POA manager
- Creación de una implementación del servicio (servant)
- Creación del objeto Corba y su asociación al servant
- Localización del servicio de nombres

- Registro de la librería de operaciones con números enteros en el servicio de nombres.
- Espera de peticiones

4. Fichero fuente cliente (client.py). Identifica y analiza los fragmentos de código responsables de:
   - Inicializar el ORB
   - Obtener una referencia al servidor de nombres.
   - Obtener una referencia al objeto Corba remoto a partir del nombre.
   - Llamada a la función principal de la librería.

5. Discutir características de Corba en términos de transparencia de ubicación, transparencia de acceso, facilidad de implementación para servidor y cliente y describir servicios en los que el empleo de Corba sería adecuado y servicios en los que no sería la opción más apropiada. Completar la tabla comparativa.

JAVA RMI

Se implementará la librería de operaciones con números enteros utilizando Java RMI como mecanismo de comunicación cliente-servidor. En este caso, se necesitará implementar la interfaz remota (**IntLibrary.java**), la funcionalidad del servidor (**server.java**) y la funcionalidad del cliente (**client.java**). Como se verá más adelante, el resto de ficheros necesarios para la generación de los programas servidor y cliente se generarán automáticamente.

Paso 0. Definir la interfaz remota (IntLibrary.java), implementar el servidor (server.java) y el cliente (client.java)

Servidor

1. En la máquina virtual del servidor (10.X.Y.1), generar el subdirectorio “javarmi”.
2. Copiar los ficheros necesarios a la máquina virtual del servidor.
   ```
   >> scp ./javarmi/intLibrary.java si2@10.X.Y.1:/home/si2/javarmi
   >> scp ./javarmi/server.java si2@10.X.Y.1:/home/si2/javarmi
   ```

3. Desde la máquina virtual del servidor, compilar el servidor.
   ```
   >> javac -d . intLibrary.java server.java
   ```

4. Desde la máquina virtual del servidor, arrancar el Java RMI Registry y comprobar que está a la escucha.

   ```
   >> rmiregistry &
   >> netstat -l -t
   ```

5. Arrancar el servicio.
   ```
   >> java -classpath . intLibrary.server
   ```

Cliente:

1. En la máquina virtual del cliente (10.X.Y.2), generar el subdirectorio “javarmi”.
2. Copiar los ficheros necesarios a la máquina virtual del cliente.
   ```
   >> scp ./javarmi/intLibrary.java si2@10.X.Y.2:/home/si2/javarmi
   >> scp ./javarmi/client.java si2@10.X.Y.2:/home/si2/javarmi
   ```

3. Desde la máquina virtual del cliente, compilar el cliente.
   ```
   >> javac -d . intLibrary.java client.java
   ```

4. Lanzar petición al servidor desde el cliente:
   ```
   >> java  -classpath . intLibrary.client 10.X.Y.1 0 1 2
   ```

   Ejemplo:

   ```
   si2@si2srv02:~/javarmi$ java  -classpath . intLibrary.client
   Usage: <naming service host> <tipo de operacion (0:suma / 1:
   resta / 2: multiplicacion / 3:division)> <operador 1> <operador 2>

   si2@si2srv02:~/javarmi$ java  -classpath . intLibrary.client
   10.3.99.1 0 1 2
   Respuesta del servidor: 3
   ```

Ejercicios:

1. Abrir el fichero intLibrary.java que contiene la declaración de la interfaz y observar que la interfaz extiende la interfaz java.rmi.Remote y el método remoto lanza excepciones del tipo java.rmi.RemoteException.
2. Fichero fuente servidor (server.java). Observar que la clase remota se define como una implementación de la interfaz intLibrary definida en intLibrary.java. Comenta brevemente la funcionalidad de cada una de las instrucciones de la función main.
3. Fichero fuente cliente (client.java). Comenta brevemente la funcionalidad de cada una de las instrucciones de la función main.
4. Discutir características de Java RMI en términos de transparencia de ubicación, transparencia de acceso, facilidad de implementación para servidor y cliente y describir servicios en los que el empleo de Java RMI sería adecuado y servicios en los que no sería la opción más apropiada. Completar la tabla.

TABLA COMPARATIVA

Completar la siguiente tabla de acuerdo al análisis de las diferentes arquitecturas (sockets, RPCs, CORBA y Java RMI) llevado a cabo en los apartados anteriores.

	Sockets	**RPCs**	**CORBA**	**Java RMI**
Transparencia de ubicación				
Transparencia de acceso				
Facilidad implementar servidor				
Facilidad implementar cliente				
Interoperabilidad				
Servicios adecuados				
Servicios menos apropiados				

Parte II: Propuesta de un servicio implementado mediante sockets, RPCs, Corba o Java RMI.

Proponer un nuevo servicio para el que el sistema de transporte asignado por el profesor sea especialmente idóneo.

- Discutir razonadamente por qué la arquitectura cliente-servidor asignada es apropiada para implementar este tipo de servicio.
- Indicar en líneas generales los principales cambios que habría que realizar en el código proporcionado. No es necesario implementar estos cambios en el código.