Redes Informáticas II (IS18)

Práctica 3: Programación en Red. Protocolos TCP y UDP.

Comunicación entre procesos mediante Sockets

1. SOFTWARE A UTILIZAR:

1. Sistema operativo Linux.
2. Compilador gcc de lenguaje C.

2. INTRODUCCIÓN TEÓRICA

El objetivo de los protocolos del **nivel de transporte** es ofrecer a sus clientes en capas superiores un servicio de calidad, fiable y eficaz, para garantizar la comunicación con otra entidad remota de su mismo nivel con independencia del medio o medios a través de los cuales se verifique la transmisión de información. El modelo TCP/IP ofrece dos protocolos bien diferentes, TCP y UDP.

2.1 Protocolo TCP

El protocolo de control de la transmisión TCP (*Transmision Control Protocol*) ha sido diseñado para proporcionar una conexión punto a punto bidireccional fiable. Las aplicaciones envían bloques de datos a la entidad TCP local, que se encarga de descomponerlos en diferentes paquetes del tamaño apropiado que serán enviados como datagramas IP. La entidad TCP de la máquina remota recibe los diferentes paquetes, los ordena y recompone el bloque original de datos que entrega a la aplicación destino. Durante este proceso, el protocolo TCP se encarga de reenviar o reclamar los paquetes perdidos debidos a problemas en la comunicación.

La comunicación descrita entre aplicaciones se realiza gracias a los sockets, que se describirán más adelante, y que son la interfaz con el protocolo TCP. Cada socket se identifica, en el protocolo TCP/IP, mediante un número único compuesto por la dirección IP del host y un valor de 16 bits llamado puerto. Los números de puerto inferiores a 1024 son puertos bien conocidos (*well-known ports*) que se utilizan para servicios estándar. El anexo I da una lista de los más importantes.

Otro aspecto a tener en cuenta acerca del funcionamiento de los agentes TCP es la gestión temporal de los mensajes. Una vez la aplicación ha enviado una serie de datos para transferir, estos pueden ser enviados inmediatamente o almacenados en un buffer para su posterior envío. Si la aplicación desea un control más estricto sobre el tiempo de transmisión, puede utilizar los flags PUSH o URGENT. El primero causa la transmisión inmediata de los datos, mientras que el segundo fuerza al agente TCP a enviar inmediatamente todo el contenido del buffer.

Para gestionar correctamente la transmisión, las entidades TCP se comunican mediante estructuras llamadas segmentos. Cada una de ellas incluye una cabecera de 20 bytes, un campo opcional y un bloque de datos que puede estar vacío. El formato de los segmentos aparece en la figura 1. Existen dos restricciones al tamaño de los segmentos. En primer lugar, deben caber completamente en un mensaje IP, limitado a 65.535 bytes. En segundo lugar, el nivel físico de las redes suele imponer un tamaño máximo de paquete más restrictivo, como los 1500 bytes de ethernet en los que debe incluirse el paquete IP con el segmento TCP incluido.

Para entender el protocolo de comunicación, llamado de ventana deslizante, se debe saber que cada byte en una conexión TCP tiene asociado un único número de secuencia, representado mediante un entero sin signo de 32 bits. Cuando un emisor envía un segmento, activa un temporizador. Cuando el segmento llega al destino, el receptor responde con otro segmento con un número de reconocimiento igual al siguiente número de secuencia que espera recibir. Si el

temporizador llega a cero antes de recibir el reconocimiento, el segmento se vuelve a enviar. Se debe tener en cuenta que el receptor sólo reconoce los segmentos en orden, aunque los reciba desordenados.

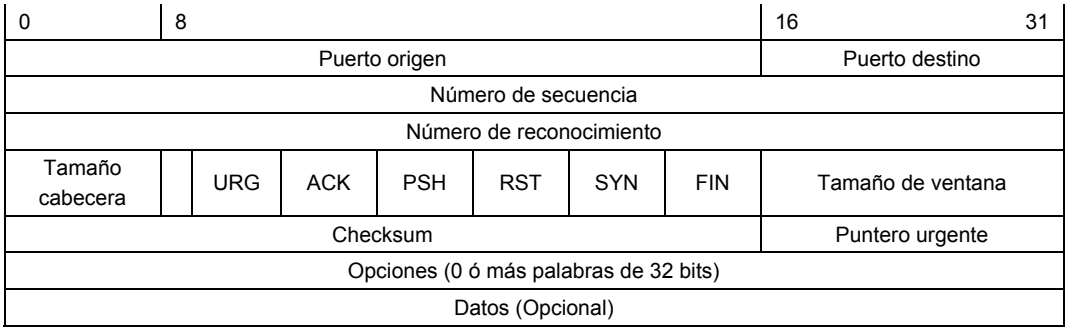


Figura 1. Formato de los segmentos TCP

Los campos de la cabecera de un segmento TCP, según aparecen en la figura 1, son los siguientes:

- **Puerto origen y destino:** identifican ambos extremos de la conexión TCP.
- **Número de secuencia y de reconocimiento:** son los valores descritos anteriormente, que se utilizan para ordenar los mensajes y verificar la transmisión correcta.
- **Tamaño de cabecera:** indica el tamaño en palabras de 32 bits de la cabecera del segmento, variable por poseer un campo de opciones de longitud variable.
- **Flag URG:** indica mesaje urgente. En este caso, el puntero urgente indica un offset dentro del número de secuencia actual donde se encuentran los datos que se deben procesar con urgencia.
- **Flag ACK:** indica mensaje de reconocimiento. En este caso el número de reconocimiento es válido.
- **Flag PSH:** datos enviados de forma inmediata mediante la opción PUSH descrita.
- **Flag RST:** usado para cancelar y restablecer la conexión en caso de problemas.
- **Flag SYN:** puesto a uno en segmentos de conexión.
- **Flag FIN:** puesto a uno para cerrar una conexión.
- **Tamaño de ventana:** indica el número de bytes que pueden ser enviados, siendo el primero de ellos el último reconocido.
- **Checksum:** se incluye por fiabilidad.
- **Opciones:** se utiliza para añadir funcionalidad al protocolo. Permite por ejemplo indicar el tamaño máximo de los paquetes que se pueden recibir.

TCP utiliza un protocolo de establecimiento de conexiones que requiere del envío de tres mensajes. La máquina que desea establecer una conexión envía un segmento dirigido a la máquina y puerto destino, con un número de secuencia X. El flag SYN estará activado y ACK desactivado. La máquina destino, en caso de aceptar la conexión, responde con otro mensaje, con SYN y ACK activados, número de secuencia Y y de reconocimiento X+1. Por último, el nodo que inicio la conexión envía un tercer mensaje con secuencia X+1 y reconocimiento Y+1. Durante este proceso se negocia también el tamaño de la ventana de datos.

2.2 Protocolo UDP

El modelo de referencia TCP/IP soporta otro protocolo en el nivel de transporte, no orientado a conexión, que se denomina protocolo de datos de usuario, UDP (*User Data Protocol*). Este protocolo permite enviar datos encapsulados con una pequeña cabecera, que aparece en la figura 2.

Este protocolo supone una cierta fiabilidad en el nivel físico, y que la comunicación va a verificarse satisfactoriamente con un solo mensaje.

0	15	16	31
Puerto origen		Puerto destino	
Longitud de datos+cabecera		Checksum	

Figura 2. Formato de la cabecera de un mensaje UDP

2.3 Comunicación mediante Sockets

El paso de mensajes es el mecanismo de comunicación entre procesos más utilizado hoy en día. Su ventaja más destacable es que puede ser empleado en cualquier tipo de arquitectura, sea monoprocesador en tiempo compartido, multiprocesador con memoria compartida, multicomputador o sistema distribuido.

El soporte lógico del sistema es el encargado de ofrecer los servicios necesarios para permitir la comunicación mediante paso de mensajes. Para ello se emplean servicios propios del núcleo del sistema operativo, funciones de librería que se enlazan en cada proceso con el código de usuario y procesos servidores especializados. Si el mecanismo de paso de mensajes está diseñado de forma adecuada, ofrece al código de usuario una única interfaz, ocultando la ubicación física del proceso con el que se establece la comunicación. De esta manera, el mismo programa servirá para comunicar dos procesos en la misma máquina o residentes en máquinas diferentes separadas por miles de kilómetros.

El proceso de comunicación mediante paso de mensajes requiere normalmente la ejecución de diferentes etapas:

- En primer lugar los procesos que quieran participar en la comunicación deben obtener un nombre o dirección, para poder ser identificados como emisores o receptores de mensajes.
- En segundo lugar, los procesos que vayan a intercambiar información deben establecer una conexión entre ellos, y deben poder deshacer dicha conexión cuando el intercambio de información haya concluido.
- Por último, se debe gestionar la emisión y recepción de información entre procesos conectados. Los servicios necesarios para cumplir con las anteriores etapas, con diversas variaciones y ampliaciones, se deben encontrar en la interfaz de paso de mensajes ofrecida por el sistema operativo.

En esta práctica se va a estudiar la interfaz de paso de mensajes mediante sockets definida en la versión BSD del sistema operativo UNIX, que es idéntica a la ofrecida por Linux. Tras la descripción teórica correspondiente se realizarán prácticas de comunicación entre procesos, residentes en el mismo ordenador en primer lugar, y soportada por el modelo TCP/IP para procesos residentes en distintas máquinas.

2.3.1 Interfaz de comunicación de sockets en el BSD UNIX de Berkeley

Un *socket* (conector en algunas traducciones) es un extremo de una línea (lógica) de comunicación. Dos procesos pueden comunicarse creando un socket cada uno, y enviando mensajes entre ellos. Cada socket va asociado a un proceso y tiene una identificación (dirección) única en el sistema. Un proceso puede solicitar al sistema operativo tantos sockets como necesite. Existen diversas clases de sockets, diferenciados por el formato de la dirección y el protocolo usado para la comunicación, de manera que la tripleta *<dominio_de_dirección, tipo, protocolo>* determina cada una de estas clases.

2.3.2 Creación de sockets

Para crear un socket en un proceso se utiliza la llamada al sistema:

```
descriptor_de_socket = socket(dominio, tipo, protocolo)
```

donde `descriptor_de_socket`, `dominio`, `tipo` y `protocolo` son enteros y que devuelve un entero positivo llamado descriptor similar a los *handlers* devueltos por la llamada *open*() de

apertura de ficheros. De hecho, la fuente de números para *handlers* y descriptores de sockets es la misma, de manera que cualquier número usado identificará a un fichero abierto o a un socket, nunca a ambas cosas.

2.3.2.1 Dominios

La especificación de un dominio indica la familia de direcciones a la que el socket pertenece, que a su vez impone un formato para la dirección del mismo. Los distintos dominios se identifican mediante constantes definidas en el fichero `<sys/socket.h>`. Las más utilizadas son AF_UNIX que utiliza nombres similares a las rutas de ficheros en UNIX y AF_INET que utiliza direcciones tipo Internet.

En el primer caso, las direcciones son nombres de ficheros de hasta 108 caracteres, que incluyen la ruta desde el directorio raíz. Se recomienda utilizar /tmp o algún otro directorio temporal del disco local para situar los sockets. El dominio AF_UNIX sólo permite comunicación entre procesos de la misma máquina. Las direcciones del dominio AF_INET utilizan protocolos de TCP/IP para la comunicación y por lo tanto permiten la comunicación entre procesos residentes en máquinas distintas. Las direcciones están compuestas por dos campos: la dirección de IP, formada por una dirección de red y una dirección de máquina (de acuerdo con lo visto en la práctica anterior), y un número de puerto que distingue entre sockets de la misma máquina. Los servicios estándar utilizan siempre los mismos puertos (79 para *finger*, 513 para *rlogin*,...) de manera que los 1024 primeros puertos están siempre reservados.

Una vez se ha creado un socket, es necesario asignarle una dirección mediante la llamada al sistema:

```
estado = bind(descriptor_de_socket, direccion, longitud)
int estado, descriptor_de_socket, longitud;
struct sockaddr *direccion;
```

que devuelve -1 en caso de error y 0 en otro caso. `descriptor_de_socket` es el valor devuelto por la llamada *socket*(). Los otros dos campos tienen que ver con la dirección. Las estructuras utilizadas en los dominios descritos con anterioridad son `sockaddr_un` (definida en `<sys/un.h>`) y `sockaddr_in` (definida en `<netinet/in.h>`) respectivamente, y `longitud` es el tamaño en bytes de la estructura correspondiente.

2.3.2.2 Estilos de comunicación

El parámetro tipo indicado en la llamada *socket*() especifica el modelo abstracto de comunicación que usará el socket creado. Estos se indican también mediante constantes definidas en `<sys/socket.h>` y los más destacables son SOCK_STREAM y SOCK_DGRAM. Ambos se pueden usar en los dos dominios descritos.

El primero de ellos se utiliza para comunicaciones tipo *stream*, orientadas a conexión y fiables. El soporte ofrecido a este tipo por la capa de transporte asegura recepción única y en orden de los paquetes. El segundo ofrece una comunicación tipo *datagrama*, que no está orientada a conexión y no es fiable. Los datagramas enviados al socket por la capa de transporte pueden perderse, llegar duplicados o fuera de orden.

2.3.2.1 Protocolos

En la actualidad cada protocolo soporta únicamente su estilo de comunicación asociado. De esta manera, para especificar un protocolo basta con especificar el parámetro tipo en la llamada *socket*() y pasar el valor 0 como tercer parámetro. El tipo SOCK_STREAM obliga a utilizar el protocolo TCP, SOCK_DGRAM el protocolo UDP y SOCK_RAW el protocolo IP sin capas superiores.

2.3.3 Establecimiento de conexiones

Para utilizar sockets de tipo stream es necesario, antes de enviar o recibir datos, establecer una conexión entre los dos extremos de la comunicación. Para ello se utilizan diferentes llamadas al sistema.

Uno de los sockets, llamado *activo* porque se dispone a enviar información, establece la conexión con otro llamado *pasivo* mediante:

```
estado = connect(descriptor_de_socket, direccion, longitud)
int estado, descriptor_de_socket, longitud;
struct sockaddr *direccion;
```

que, como la llamada anterior, devuelve -1 en caso de error y 0 en otro caso. Ahora descriptor_de_socket hace referencia al socket activo (del proceso que realiza la llamada) y los campos relacionados con la dirección hacen referencia al socket remoto o destino. Es conveniente indicar que esta llamada puede ser utilizada con sockets locales tipo datagrama, en cuyo caso fijan el destino para futuros envíos de datos.

El socket pasivo debe, en primer lugar, crear una cola de conexiones mediante la llamada:

```
estado = listen(descriptor_de_socket, longitud_de_cola)
int estado, descriptor_de_socket, longitud_de_cola;
```

que crea una cola de hasta SOMAXCONN (constante definida en `<sys/socket.h>`) conexiones con el socket local descriptor_de_socket, y posteriormente aceptar una a una las conexiones mediante:

```
nuevo_socket = accept(antiguo_socket, direccion, longitud)
int nuevo_socket, antiguo_socket, longitud;
struct sockaddr *direccion;
```

que crea un nuevo socket, idéntico al original antiguo_socket con el descriptor devuelto por la llamada, y recibe la dirección del socket activo en direccion. El socket original no se ve afectado por la llamada, que bloquea al proceso en caso de no haber conexiones pendientes.

2.3.4 Transferencia de datos

Una vez establecida la conexión (para sockets tipo stream) puede dar comienzo la transmisión de datos. Las parejas de llamadas `read/write`, `recv/send`, `recvfrom/sendto`, `recvmsg/sendmsg` y `readv/writev` pueden usarse a tal efecto, dependiendo la elección de la funcionalidad que se requiera. Así recv/send se utilizan para sockets conectados según se ha visto anteriormente, recvfrom/sendto se utilizan con sockets tipo datagrama, que requieren especificar el destino y reconocer el origen en cada comunicación, etcétera.

Cada una de estas llamadas devuelve un entero indicando el número de bytes transmitidos o -1 en caso de error, teniendo en cuenta que una transmisión con éxito no siempre implica una correcta recepción del mensaje.

2.3.4.1 Sincronización

Las llamadas que realizan la transferencia de datos son por defecto bloqueantes. Cuando se quiere permitir que un proceso acepte comunicación a través de varios puertos de forma concurrente, se puede utilizar la llamada al sistema *select*(), que permite detectar si existen datos a leer o se puede escribir en un determinado socket:

```
encontrados = select(num, msc_lec, msc_esc, msc_exc, timeout)
int encontrados, num;
fd_set *msc_lec, *msc_esc, *msc_exc;
struct timeval *timeout;
```

Las máscaras de tipo `fd_set`, definido en `<sys/types.h>`, son parámetros de entrada y salida. Antes de la llamada ponen a 1 el campo correspondiente al socket o sockets a explorar; a la salida incluyen un 1 en los campos activados previamente en los que la condición buscada (lectura, escritura o excepción) se verifica. `<sys/types.h>` define además una serie de macros para manipular estas máscaras. Si n es un descriptor de socket y p una máscara, se tiene:

- `FD_SET(n, p)` activa el campo correspondiente al socket.
- `FD_CLR(n, p)` desactiva el campo correspondiente al socket.
- `FD_ISSET(n, p)` devuelve el valor del campo correspondiente al socket.
- `FD_ZERO(p)` pone a cero (desactiva) todos los campos de la máscara.

El valor entero numero indica cuántos sockets van a ser inspeccionados por la llamada. La constante `FD_SETSIZE` definida en `<sys/types.h>`, indica el número máximo de descriptores de cada `fd_set`, y es un buen valor para numero. El parámetro `timeout`, cuyo tipo está definido en `<sys/time.h>`, sirve para indicar la espera producida por la llamada. Un puntero nulo causará un bloqueo indefinido, mientras que una estructura de tiempo con valor cero permite a la llamada retornar inmediatamente. En cualquier caso, la llamada retorna cuando se ha encontrado alguna condición de entre las activas. `encontrados` indica el número de condiciones halladas.

2.3.5 Desconexión

Para cerrar una conexión se puede utilizar la llamada *close*(), de igual manera que cuando se cierra un fichero. Si se ha utilizado el dominio AF_UNIX, será necesario también eliminar del sistema de ficheros la ruta que apunta al socket mediante la llamada:

```
estado = unlink(ruta)
int estado;
char *ruta;
```

2.3.6 Multicast.

Multicast es... una necesidad, al menos en algunas ocasiones. Si tiene información (*mucha* información habitualmente) que debe ser transmitida a varios ordenadores (pero no a *todos*) en una Internet, entonces la respuesta es Multicast. Una situación frecuente donde se utiliza es en la distribución de audio y vídeo en tiempo real a un conjunto de ordenadores que se han unido a una conferencia distribuida.

Multicast es, en gran medida, como la televisión o la radio, es decir, sólo aquellos que han sintonizado sus receptores (al seleccionar una frecuencia particular que les interesa) reciben la información. Esto es: escucha los canales que te interesan, pero no otros.

2.3.6.1 El problema con Unicast.

Unicast es cualquier cosa que no sea broadcast o multicast. Estaa definición no es muy brillante... pero cuando envías un paquete y sólo hay un emisor -tú- y un receptor (aquél al que envías el paquete), entonces estás haciendo unicast. TCP es, por propia naturaleza, orientado a unicast. UDP soporta muchos otros paradigmas, pero si estás enviando paquetes UDP y sólo se supone que hay un proceso que lo recibe, es también unicast.

Durante años las transmisiones unicast demostraron ser suficientes para Internet. La primera implementación de multicast no vio la luz hasta 1993, con la versión 4.4 de BSD. Parece que nadie lo necesitaba hasta entonces. ¿Cuáles eran los nuevos problemas que trataba de arreglar multicast?

No es necesario decir que Internet ha cambiado mucho desde aquellos años. Particularmente, el nacimiento de la World Wide Web (WWW) cambió sensiblemente la situación: la gente no quería solamente conexiones a ordenadores remotos, correo electrónico y FTP. Primero querían ver las fotos de las personas, dentro de páginas personales, pero más tarde también querían *verles* y *oírles*.

Con la tecnología actual es posible afrontar el «coste» de hacer una conexión unicast con todos aquellos que desean ver su página web. Sin embargo, si quieres enviar audio y vídeo, que necesita de un *gran* ancho de banda comparado con aplicaciones de web, tienes (*tenías*, hasta que apareció multicast) dos opciones: establecer conexiones unicast por separado con *cada uno* de los receptores, o usar broadcast. La primera solución no era factible: si hemos dicho que *cada* conexión enviando audio/vídeo consume una gran cantidad de ancho de banda, imagina tener que establecer cientos, quizás miles, de estas conexiones. Tanto el ordenador emisor como su red se colapsarían.

Broadcast parece *una* solución, pero desde luego no es *la* solución. Si deseara que todos los ordenadores en su LAN atendieran la conferencia, podrías usar broadcast. Se enviarían los paquetes una sola vez y cada ordenador lo recibiría ya que fueron enviados a la dirección de

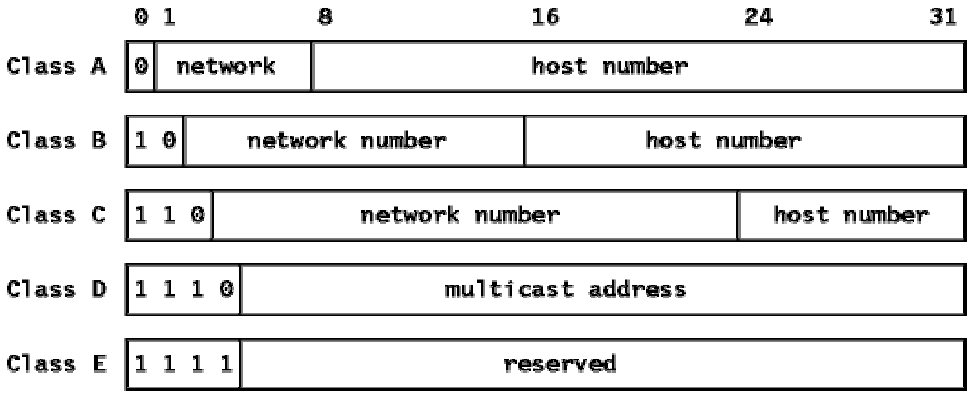
broadcast. El problema es que quizás solo *algunos* de éstos y no *todos* estén interesados en estos paquetes. Más aún: quizás algunos ordenadores están realmente interesados en su conferencia, pero están fuera de su LAN, a varios routers de distancia. Y ya se sabe que el broadcast funciona bien dentro de una LAN, pero surgen problemas cuando haces broadcast de paquetes que deben ser enviados a través de diferentes LANs.

La mejor solución parece ser aquella en la que sólo se envían paquetes a ciertas direcciones especiales (una determinada frecuencia en transmisión de radio y televisión). Así, todos los ordenadores que han decidido unirse a la conferencia conocerán la dirección de destino, de forma que los recogerán cuando pasen por la red y los enviarán a la capa IP para que sean demultiplexados. Esto es similar al broadcast en el sentido de que sólo se envía un paquete de broadcast y todos los ordenadores en la red lo reconocen y lo leen; difiere, sin embargo, en que no todos los paquetes de multicast son leídos y procesados, sino solamente los que han sido registrados en el kernel como «de interés».

Estos paquetes especiales son encaminados en el nivel del kernel como cualquier paquete porque son paquetes IP. La única diferencia puede estar en el algoritmo de encaminamiento que le dice al kernel si debe o no encaminarlos.

2.3.6.2 Direcciones Multicast.

El conjunto de las direcciones IP está divido en «clases» basado en los bits de mayor orden en una dirección IP de 32 bits.



Esto conduce a los siguientes rangos de direcciones:

- |0| Direcciones de clase A | → 0.0.0.0 - 127.255.255.255
- |1 0| Direcciones de clase B | → 128.0.0.0 - 191.255.255.255
- |1 1 0| Direcciones de clase C | → 192.0.0.0 - 223.255.255.255
- |1 1 1 0| Direc. MULTICAST | → 224.0.0.0 -239.255.255.255
- |1 1 1 1 0| Reservadas | → 240.0.0.0 - 247.255.255.255

La que nos interesa es la «Clase D». Cada datagrama de IP cuya dirección destino empieza por «1110» es un datagrama IP de Multicast.

Los 28 bits restantes identifican el «*grupo*» multicast al que se envía el datagrama. Siguiendo con la analogía anterior, debe sintonizar su radio para oír un programa que se transmite a una frecuencia determinada, del mismo modo debe «sintonizar» su kernel para que reciba paquetes destinados a un grupo de multicast específico. Cuando hace esto, se dice que el ordenador se ha *unido* a aquél grupo en el interfaz especificado. Esto se verá más adelante.

Hay algunos grupos especiales de multicast, o «grupos de multicast bien conocidos», y no debería usar ninguno de estos en una aplicación determinada dado que están destinados a un propósito en particular:

- 224.0.0.1 es el grupo de *todos los ordenadores* [*all-computers*, n. del t.]. Si hace un «ping» a ese grupo, todos los ordenadores que soporten multicast en la red deben responder, ya que todos ellos *deben* unirse a este grupo en el arranque de todos sus interfaces que soporten multicast.
- 224.0.0.2 es el grupo de *todos los encaminadores* [*all-routers*, n. del t.]. Todos los encaminadores de multicast deben unirse a este grupo en todos los interfaces de multicast.
- 224.0.0.4 es el de *todos los encaminadores DVMRP*, 224.0.0.5 el de *todos los encaminadores OSPF*, 224.0.0.13 el de *todos los encaminadores PIM* etc.

Todos estos grupos especiales de multicast son publicados cada cierto tiempo en el RFC de «Números asignados».

En cualquier caso, el conjunto de direcciones de la 224.0.0.0 a 224.0.0.255 están reservadas localmente (para tareas administrativas y de mantenimiento) y los datagramas enviados a estos nunca se envían a los encaminadores multicast. De manera similar, el conjunto 239.0.0.0 a 239.255.255.255 ha sido reservado para ámbitos administrativos [*administrative scoping*]

2.3.6.3 Nivel de cumplimiento.

Los ordenadores pueden estar en tres niveles en lo que se refiere al cumplimiento de la especificación de Multicast, de acuerdo con los requisitos que cumplen.

Se está en **Nivel 0** cuando no hay soporte para Multicast en IP. Un buen número de los ordenadores y los encaminadores de Internet están en este nivel, ya que el soporte de multicast no es obligatorio en IPv4 (sí lo es, sin embargo, en IPv6). No es necesaria mucha explicación aquí: los ordenadores en este nivel no pueden enviar ni recibir paquetes multicast. Deben ignorar los paquetes enviados por otros ordenador con capacidades de multicast.

En el **Nivel 1** hay soporte para envío pero no para recepción de datagramas IP de multicast. Nótese, por tanto, que no es necesario unirse a un grupo multicast para enviarle datagramas. Se añaden muy pocas cosas al módulo IP para convertir un ordenador de «Nivel 0» a «Nivel 1».

El **Nivel 2** es el de completo soporte a multicast en IP. Los ordenadores de nivel 2 deben ser capaces de enviar y recibir tráfico multicast. Tienen que saber la forma de unirse o dejar grupos multicast y de propagar ésta información a los encaminadores multicast. Es necesario incluir, por tanto, una implementación del Protocolo de Gestión de Grupos de Internet (Internet Group Management Protocol, IGMP) en su pila TCP/IP.

2.3.6.4 Envío de Datagramas Multicast.

Por ahora debe ser evidente que el tráfico multicast es manipulado en el nivel de transporte con UDP, ya que TCP provee de conexiones punto a punto que no son útiles para tráfico multicast. (Se está llevando a cabo una fuerte investigación para definir e implementar nuevos protocolos de transporte orientados a multicast).

En principio, una aplicación sólo necesita abrir un socket UDP y poner como dirección de destino la dirección multicast de clase D donde quiere enviar sus datos. Sin embargo, hay algunas operaciones que el proceso emisor debe ser capaz de controlar.

2.3.6.4.1 TTL.

El campo TTL (Time To Live, Tiempo de vida) en la cabecera IP tiene un doble significado en multicast. Como siempre, controla el tiempo de vida de un datagrama para evitar que permanezca por siempre en la red debido a errores en el encaminamiento. Los encaminadores decrementan el TTL de cada datagrama que pasa de una red a otra y cuando este valor llega a 0 el paquete se destruye.

El TTL del multicast en IPv4 tiene también el significado de «barrera». Su uso se hace evidente con un ejemplo: supongamos que tiene una vídeo conferencia larga, que consume mucho ancho de banda, entre todos los ordenadores que pertenecen a su departamento. Así que quiere que esa gran cantidad de tráfico permanezca en su LAN y no salga fuera. Quizás su departamento es suficientemente grande para tener varias LANs. En este caso quieres que los ordenadores que

pertenecen a cada una de *sus* LANs atiendan la conferencia, pero en ningún caso quiere colapsar todo Internet con su tráfico multicast. Es necesario limitar hasta dónde se expandirá el tráfico multicast entre encaminadores. Para esto se utiliza el TTL. Los encaminadores tienen una barrera TTL asignada a cada uno de sus interfaces, de forma que sólo los datagramas con un TTL mayor que la barrera del interfaz se reenvían. Sin embargo, cuando un datagrama atraviesa un encaminador con una barrera determinada, el TTL del datagrama *no* se ve decrementado por el valor de la barrera. Sólo se hace una comparación (como antes, el TTL se decrementa en uno cada vez que un datagrama pasa por un encaminador).

Una lista de las barreras de TTL y su ámbito asociado es la siguiente:

TTL	ámbito
0	Restringido al mismo ordenador. No se enviará por ningún interfaz.
1	Restringido a la misma subred. No será reenviada por ningún encaminador.
<32	Restringido al mismo «sitio», la misma organización o departamento.
<64	Restringido a la misma región.
<128	Restringido al mismo continente.
<255	Sin ámbito restringido. Global.

Nadie sabe qué significa exactamente «sitio» o «región». Es tarea de los administradores decidir qué límite tienen estos ámbitos.

El truco-TTL no es siempre suficientemente flexible para todas las necesidades, especialmente cuando se trata con regiones solapadas o se desea establecer límites geográficos, topológicos o de ancho de banda de manera simultanea. Para resolver estos problemas, se definieron regiones de multicast de IPv4 con ámbito administrativo en 1994 (D. Meyer: *«administratively Scoped IP Multicast»*). Se definen los ámbitos basándose en direcciones de multicast en lugar de en TTLs. El rango 239.0.0.0 a 239.255.255.255 se reserva para estos ámbitos administrativos.

2.3.6.4.2 Loopback.

Cuando el ordenador emisor es de nivel 2 y también miembro del grupo al que se envían los datagramas, por defecto se envía una copia también a sí mismo, lo que se conoce con el nombre de *loopback*. Esto no significa que la tarjeta interfaz relea sus propias transmisiones, reconociéndolas como dirigidas al grupo al que pertenece y las recoja de nuevo de la red. Al contrario, es la capa IP la que, por defecto reconoce el datagrama que va a enviar y lo copia y encola en la cola de entrada de IP antes de enviarlo.

Este comportamiento es deseable en algunos casos, pero no en otros. Así que el proceso emisor puede activarlo o desactivarlo según desee.

2.3.6.4.3 Selección de interfaz.

Los ordenadores conectados a más de una red deberían ofrecer la posibilidad a las aplicaciones para que estas puedan decidir qué interfaz de red será usado para enviar las transmisiones. Si no se especifica, el kernel escogerá uno por defecto basándose en la configuración realizada por el administrador.

2.3.6.5 Recepción de datagramas Multicast.

2.3.6.5.1 Unirse a un grupo Multicast.

El broadcast es (en comparación) más sencillo de implementar que el multicast. No necesita que ningún proceso le dé ninguna regla al kernel para que éste sepa qué hacer con los paquetes de broadcast. El kernel ya sabe que hacer: leer y entregar *todos* ellos a las aplicaciones apropiadas.

Con multicast, sin embargo, es necesario avisar al kernel cuáles son los grupos multicast que nos interesan. Esto es, debemos pedir al kernel que se «una» a estos grupos multicast. Dependiendo del hardware que haya por debajo, los datagramas de multicast son filtrados por el hardware o por la capa IP (y en algunos casos por ambos). Y sólo aquellos con un grupo destino previamente registrado son aceptados.

Esencialmente, cuando nos unimos a un grupo le estamos diciendo al kernel: «Vale. Sé que, por defecto, ignoras datagramas multicast, pero recuerda que estoy interesado en *este* grupo multicast. Así que lee y entrega (a cualquier proceso interesado en estos, no sólo a mí) cualquier datagrama que veas en este interfaz de red con este grupo de multicast como su dirección destino».

Algunas consideraciones: en primer lugar, destacar que no sólo se une uno a un grupo. Se une a un grupo *en* un interfaz de red determinada. Por supuesto, es posible unirse al mismo grupo en más de una interfaz. Si no se especifica un interfaz en concreto entonces el kernel lo elegirá basándose en sus tablas de encaminamiento cuando se vayan a enviar datagramas. Es también posible que más de un proceso se una al mismo grupo multicast en la misma interfaz. Todos ellos recibirán los datagramas enviados a ese grupo vía esa interfaz.

Como se ha dicho antes, los ordenadores con capacidades multicast se unen al grupo *todos los ordenadores* en el arranque, así que «haciendo ping» a 224.0.0.1 nos devolverá todos los ordenadores de la red que tienen el multicast habilitado.

Finalmente, un proceso que desee recibir datagramas multicast tiene que pedir al kernel que se una al grupo *y* hacer un «bind» al puerto al que se dirigen los datagramas. La capa UDP utiliza tanto la dirección de destino como el puerto para demultiplexar los paquetes y decidir a qué socket/s entregarlos.

2.3.6.5.2 Abandonar un grupo multicast.

Cuando un proceso ya no está interesado en un grupo multicast, informa al kernel que *él* quiere abandonar este grupo. Es necesario entender que esto no significa que el kernel no acepte más datagramas multicast dirigidos a ese grupo multicast. Seguirá haciéndolo si hay más procesos que hicieron una petición «unirse en multicast» a ese grupo y siguen interesados. En este caso *el ordenador* recuerda los miembros del grupo, hasta que todos los procesos deciden dejarlo.

Aún más: si abandonas el grupo, pero se permanece unido al puerto donde se recibe el tráfico multicast, y hay más procesos que se unieron al grupo, seguirás recibiéndose las transmisiones multicast.

La idea es que unirse a un grupo multicast sólo dice al nivel de IP y de enlace (que en algún caso se lo dirá explícitamente al hardware) que acepten datagramas multicast para ese grupo. Quienes son realmente miembros de un grupo son los *ordenadores*, no los procesos que corren en esos *ordenadores*.

2.3.6.5.3 Proyección de direcciones IP Multicast sobre direcciones Ethernet/FDDI.

Tanto en Ethernet como en FDDI, las tramas tienen un campo de dirección destino de 48 bits. Para permitir un tipo de ARP multicast que proyecte las direcciones de multicast IP sobre las Ethernet/FDDI, el IANA reservó un conjunto de direcciones para multicast: cada trama Ethernet/FDDI con su dirección destino en el rango 01-00-5e-00-00-00 a 01-00-53-ff-ff-ff (hexadecimal) contiene datos para un grupo multicast. El prefijo 01-00-5e identifica la trama como de multicast, el siguiente bit es siempre 0, y así sólo se dejan 23 bits para la dirección multicast. Ya que los grupos de multicast IP son de 28 bits la proyección no puede ser uno a uno. Sólo los 23 bits menos significativos del grupo multicast IP se ponen en la trama. Los 5 bits más significativos son ignorados, dando lugar a 32 grupos de multicast que se proyectan a la misma dirección Ethernet/FDDI. Esto significa que el nivel de Ethernet actúa como un filtro imperfecto, y el nivel IP tendrá que decidir si aceptar o no los datagramas que el nivel de enlace le ha entregado. El nivel IP actúa como el filtro definitivo perfecto.

Los detalles de Multicasting IP sobre FDDI se dan en el RFC 1390: *«Transmission of IP and ARP over FDDI Networks»*. Para más información en la proyección de direcciones multicast IP sobre Ethernet puede consultar draft-ietf-mboned-intro-multicast-03.txt: *«Introduction to IP Multicast Routing»*.

Si está interesado en Multicast IP sobre redes de área local de Token-Ring mirar el RFC 1469 para más detalles.

2.4 Información adicional

Más información sobre el uso de las llamadas descritas, la gestión de concurrencia mediante señales y threads, y los sockets en general se puede encontrar en la sección de referencias, y mediante las páginas de manual (*man*) del sistema.

3 DESARROLLO DE LA PRÁCTICA.

Se propone el desarrollo de un servicio web que juegue al juego de Batalla Naval o Guerra de Barcos.

3.1 Normas del Juego

Preparación del Tablero.

Consiste en una cuadrícula de 10x10 casillas, numeradas de 1 a 10. Sobre este tablero se disponen los barcos de modo que no estén adyacentes ni superpuestos. Es decir que alrededor de cada barco debe haber una zona de aguas prohibidas.

Los barcos pueden disponerse en horizontal y/o vertical pero no en diagonal, con lo que podemos tener unos distribuidos horizontalmente y otros verticalmente dentro del mismo tablero de juego. La cantidad y tamaño de los barcos se especifica en la siguiente tabla:

Tipo de Barco	Portaaviones	Destructor	Fragata	Submarino
Casillas	4	3	2	1
Cantidad	1 barco	2 barcos	3 barcos	4 barcos

Cualquier incumplimiento respecto a la colocación implicará perder la partida.

Tras un sorteo del turno inicial el jugador al que le ha correspondido dispara y en el caso de no hacer blanco (**agua**) el turno pasa al otro jugador. En el caso de haber acertado en el blanco (**tocado** o **hundido**) el turno no cambia de jugador sino que el que ha disparado vuelve a disparar.

Al cambiar el turno se repite la situación... Si no se hace blanco el turno cambia y si se hace se mantiene.

El juego acaba cuando uno de los jugadores hunde todos los barcos del contrincante. El primero ha ganado la partida y el segundo la ha perdido.

3.2 Objeto de la Práctica

Crear un servicio web utilizando XML-RPC que implemente la guerra de barcos. Que se trate de un servicio web implica que haya unas funciones accesibles remotamente a través del puerto 80 y con mensajes que funcionan sobre el protocolo http.

En esta práctica se utilizara el puerto 18080 en vez del 80 para ofrecer el servicio web.

Para conocer que el servicio está en marcha al iniciarse la aplicación se enviará un anuncio-presentación al grupo multicast 224.0.2.3 y puerto de destino 18080, pues no existe un directorio centralizado en el que anunciarse, donde deberían escuchar los posibles clientes del servicio como puede ser el árbitro de una liguilla de Batalla Naval.

El alumno deberá implementar la aplicación del servicio web, no es necesario crear aplicaciones cliente (véase el punto 3.5).

3.3 Funcionamiento

Tal como se ha descrito al arrancar el servicio se ha de anunciar vía multicast. Los datos a comunicar son la identificación del servicio y su dirección (IP y puerto). El mensaje estará también formado con etiquetas xml. Por ejemplo para un servicio que funcione en la ip 150.128.49.150 y el puerto 18080 el mensaje a transmitir como presentación será:

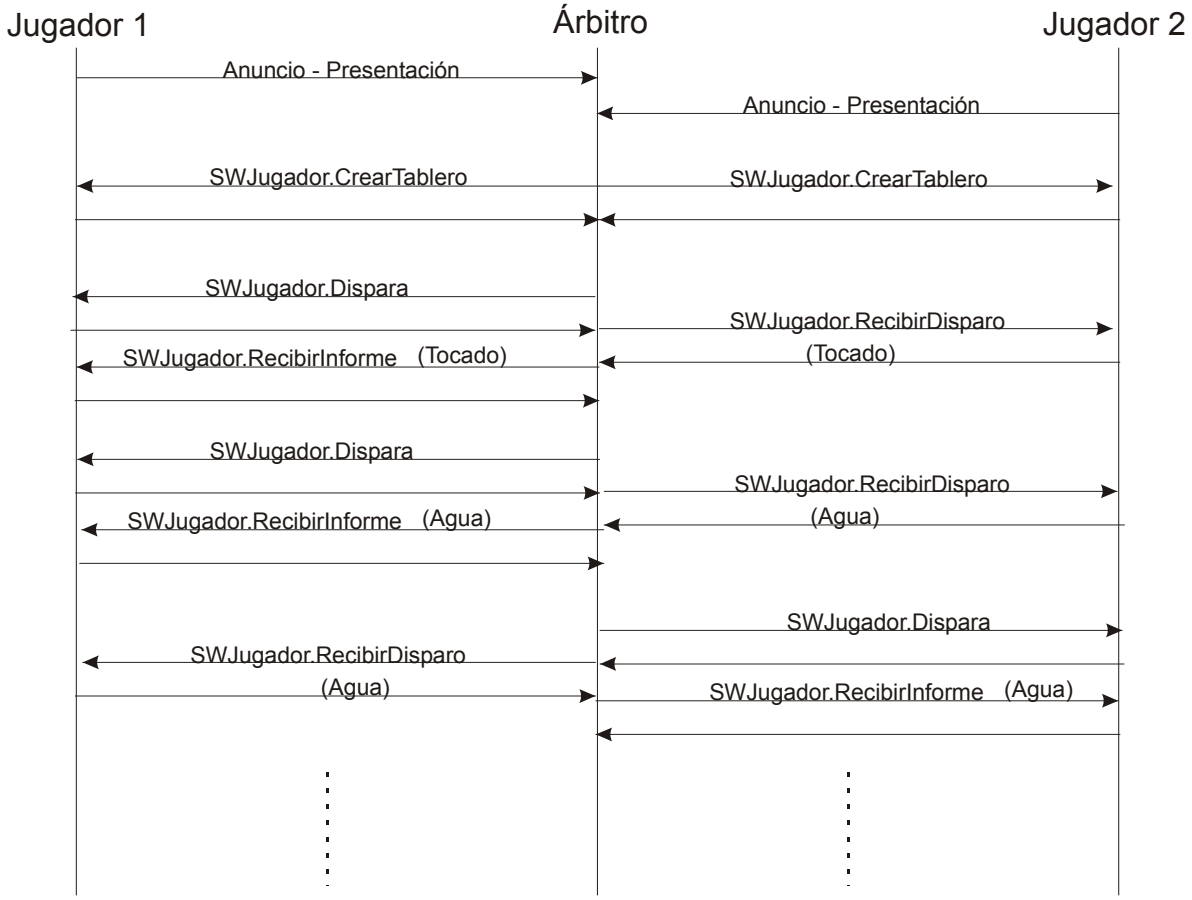
```
<?xml version="1.0"?>
<presentacionServicio>
  <params>
   <param><value><string>150.128.49.150</string></value></param>
   <param><value><i4>18080</i4></value></param>
  </params>
</presentacionServicio >
```

Las funciones accesibles remotamente toman los nombres:

- `SWJugador.CrearTablero` Crea el tablero de juego y lo comunica al árbitro. Recibe el turno inicial (o no, dependiendo del jugador)
- `SWJugador.Dispara` El servicio ofrece dos coordenadas para hacer blanco en el tablero del contrincante.
- `SWJugador.RecibirDisparo` El servicio recibe dos coordenadas y responde agua, tocado o hundido.
- `SWJugador.RecibirInforme` La respuesta del contrincante se pasa como argumento al servicio que lanzó el disparo para que pueda preparar el siguiente en consecuencia.
- `SWJugador.FinPartida` Final asíncrono de la partida. Cuando se ejecuta termina la partida independientemente de si es porque haya ganado, perdido o por alguna otra cuestión.
- `SWJugador.Status` El servicio responde dando el identificador del servicio, el estado de la partida y su identificador, estado del jugador y el del tablero.

3.3.1 Diagrama de Interacción

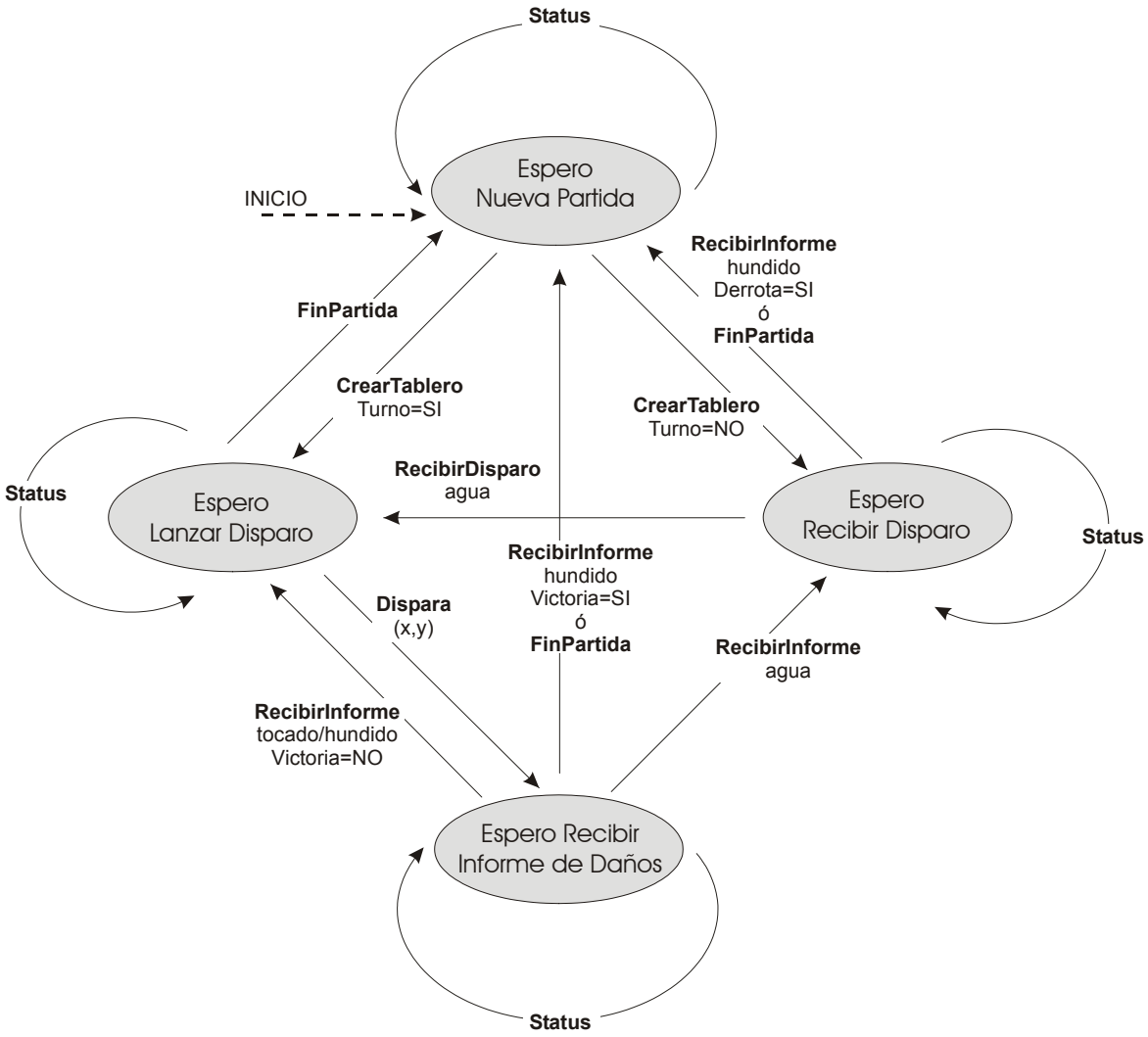
A continuación se presenta un posible diagrama de interacción



Las llamada a `SWJugador.Status` puede ocurrir en cualquier momento y no supone un cambio de estado del servicio web. Por otro lado `SWJugador.FinPartida` puede ejecutarse en cualquier momento pero si que supone un cambio de estado (el servicio pasará a esperar un nuevo inicio de partida) excepto en el caso de que ya se hubiese llegado al fin de partida previamente.

3.3.2 Máquina de Estados

Al iniciarse la aplicación notificará su disposición a aceptar partidas y entrará directamente al estado de “Espero nueva partida”. Los cambios de estado se producen con las llamadas a las funciones ofrecidas por el servicio.



3.4 Mensajes

Como todo servicio **web** que se precie los mensajes específicos de llamada a las funciones deben estar encapsulados en un mensaje http.

3.4.1 Servidor HTTP

Dado que los mensajes del servicio van como cuerpo dentro de mensajes http la aplicación deberá comportarse como un pequeño servidor web. Entre las referencias está la especificación del protocolo.

Cuando se pida la dirección base (mensaje de tipo GET sobre la dirección /) se ofrecerá una página web con el listado de las funciones, sus parámetros de entreda y de salida (ej http://localhost:18080/). El servicio web atenderá las peticiones que lleguen a / como dirección de destino (ej http://localhost:18080/) y de tipo POST. Para cualquier otra petición a cualquier otra dirección relativa dirigida al mismo puerto deberá responderse con un mensaje de error 404 (Not found) y en el cuerpo del mensaje una página web descriptiva (simple, pero que se vea que ha

habido este error si usamos un navegador, por ejemplo <html><head><tittle> Error 404. Not Found </tittle></head><body> Error 404. Recurso no encontrado <body></html>)

Requerimientos de las cabeceras de los mensajes HTTP para una petición de servicio o su respuesta están especificados en el anexo II.

Según la especificación del protocolo HTTP la línea de comando (GET, POST) y respuesta (200 OK) siempre son las primeras y a continuación van los campos con el siguiente formato "parámetro: valor\n" sin un orden especificado (los descritos y otros como fecha,…). Los que no interesen para el servicio web deberán ser descartados.

3.4.2 Mensajes XML-RPC

Como puede verse en el anexo II las peticiones al servicio web se realizan a través de mensajes http (de tipo POST para la petición y del tipo estándar de respuesta para la contestación) en los que el cuerpo está la petición descrita utilizando XML .

A continuación se verá la estructura y ejemplo de cada una de las posibles peticiones de servicio y respuestas

SWJugador.CrearTablero y su respuesta.

El parámetro enviado es el que indica si el turno inicial es de este jugador (1) o no (0). Los parámetros devueltos es el struct de estado. Este struct incluye información de identificación (jugador y partida), del estado y el tablero como una cadena de texto.

La identidad del jugador es un string prefijado. El estado de la partida es un boolean (1 indica en juego, 0 indica esperando el inicio). El identificador de partida es un string (que puede representar un número o no) que cambie con cada partida que se inicie. Si la partida no se ha iniciado el identificador estará vacío. En cuanto al turno este será 1 (boolean) si el jugador tiene el turno y un 0 si no lo tiene o no hay partida en marcha.

La cadena de texto que describe el tablero tiene un carácter por casilla, siendo 0 agua, 1 casilla con un submarino, 2 casilla parte de una fragata, 3 casilla parte de un destructor y 4 casilla parte del portaaviones. En las casillas que ya se haya disparado será X para blanco agua, T para blanco tocado (si no hay números es que el barco está hundido).

Por ejemplo para el tablero:

000000X000
033300T200
0000000000
0301022010
0300000000
…

tendríamos una variable string:
000000X000033300T2000000000000030102201003000000000…

De este modo esta respuesta será la misma que se obtendrá para la petición de estado.

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>**SWJugador.CrearTablero**</methodName> <params> <param> <value><boolean>**1**</boolean></value> </param> </params> </methodCall>	<?xml version="1.0"?> <methodResponse> <params> <param> <struct> <member> <name>IdJugador</name> <value><string>Loquillo</string></value> </member> <member> <name>EstadoPartida</name> <value><boolean>1</boolean></value> </member> <member> <name>IdPartida</name>

	<value><string>00A3F</string></value> </member> <member> <name>TurnoJugador</name> <value><boolean>1</boolean></value> </member> </struct> </param> <param> <value> <string>000000X00003330….</string> </value> </param> </params> </methodResponse>

SWJugador.Dispara y su respuesta

Para el disparo no se requieren parámetros si bien podría considerarse uno, que sea el identificador de partida para evitar que distintos clientes ataquen la misma partida.

La respuesta serán las coordenadas del disparo. El primer parámetro será la fila y el segundo la columna (recuerdesé que los valores han de ir de 1 a 10). En el mensaje del ejemplo (fila=4, columna=2)

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>SWJugador.Dispara</methodName> </methodCall> (Si se incluye el id de partida añadir …) <params> <param> <value><string>00A3F </string></value> </param> </params>	<?xml version="1.0"?> <methodResponse> <params> <param> <value><i4>4</i4></value> </param> <param> <value><i4>2</i4></value> </param> </params> </methodResponse>

SWJugador.RecibirDisparo y su respuesta

En este caso los parámetros de entrada son los mismos que en el anterior los de salida, es decir fila y columna como enteros. En este caso también podría ponerse el identificador de partida como en el caso anterior. Claro que en cada servicio web el identificador sería uno y debería ser el árbitro el que en cada caso fijara el adecuado. Como en el caso anterior, no vamos a implementarlo.

La respuesta es el tipo de blanco como entero (0=agua, 1=tocado, 2=hundido, 5 = error, coordenadas fuera del margen permitido, que es de 1 a 10 tanto en filas como columnas). También responderá con un booleano que indique si ha perdido la partida. Solo puede aparecer como cierto si la respuesta del blanco es 2 (hundido) y era el último barco. En ese caso 1 = partida perdida.

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>SWJugador.RecibirDisparo</methodName> <params> <param> <value><i4>4</i4></value> </param>	<?xml version="1.0"?> <methodResponse> <params> <param> <value><i4>0</i4></value> </param> <param>

<param> <value><i4>2</i4></value> </param> </params> </methodCall>	<value><boolean>0</boolean></value> </param> </params> </methodResponse>

SWJugador.RecibirInforme y su respuesta

La petición es como la respuesta anterior solo que en este caso el booleano significa partida ganada. En el caso de un blanco 5(error: fuera del rango) el jugador habrá perdido la partida pero esto lo indicará el árbitro con una petición siguiente de fin de partida.

En este caso el booleano significa partida ganada (1 = partida ganada)

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>SWJugador.RecibirInforme</methodName> <params> <param> <value><i4>0</i4></value> </param> <param> <value><boolean>0</boolean></value> </param> </params> </methodCall>	<?xml version="1.0"?> <methodResponse> <params> </params> </methodResponse>

SWJugador.FinPartida y su respuesta

Al servicio web no se le ofrece la razón del final asíncrono de la partida, solo si ha ganado o perdido (a través del booleano). Las posibles causas son que haya habido alguna irregularidad en el discurrir de la partida (fuera de rango en los disparos, cambio del tablero a mitad de partida)

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>SWJugador.FinPartida</methodName> <params> <param> <value><boolean>0</boolean></value> </param> </params> </methodCall>	<?xml version="1.0"?> <methodResponse> <params> </params> </methodResponse>

Tal como se indica en el anexo II no puede eliminarse la etiqueta <params> de la respuesta pues esto indica que ha funcionado correctamente (comparar con el formato de respuesta tras una excepción) . Lo que hacemos es no enviar ningún parámetro.

SWJugador.Status y su respuesta

La petición al servicio web no llevará parámetros. La respuesta tiene el mismo formato que en el caso de creación del tablero.

Petición	Respuesta
<?xml version="1.0"?> <methodCall> <methodName>**SWJugador.Status**</methodName>	<?xml version="1.0"?> <methodResponse> <params>

```
</methodCall>
```

```
    <param>
      <struct>
        <member>
          <name>IdJugador</name>
          <value><string>Loquillo</string></value>
        </member>
        <member>
          <name>EstadoPartida</name>
          <value><boolean>1</boolean></value>
        </member>
        <member>
          <name>IdPartida</name>
          <value><string>00A3F</string></value>
        </member>
        <member>
          <name>TurnoJugador</name>
          <value><boolean>1</boolean></value>
        </member>
      </struct>
    </param>
    <param>
      <value>
       <string>000000X00003330….</string>
      </value>
    </param>
  </params>
</methodResponse>
```

Formato de la respuesta si hay error

El formato de la respuesta si aparece un error en el servicio es el genérico...

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
          </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
         </member>
      </struct>
    </value>
  </fault>
</methodResponse>
```

Errores posibles

Las posibles excepciones que se han de controlar son las siguientes:

faultCode	faultString	Descripción
1	Partida YA Iniciada	Se ha llamado CrearPartida cuando ya estaba creada y sin llamar a FinPartida previamente.
2	Petición Incorrecta en el estado actual	Petición de lanzar disparo o de recibir informe mientras se está esperando una petición de recibir disparo, o petición de recibir disparo o de recibir informe mientras se está

faultCode	faultString	Descripción
		esperando una petición de lanzar disparo, o petición de lanzar o de recibir disparo mientras se está esperando una petición de recibir informe.
11	Fuera de Rango	Coordenadas del disparo fuera del rango permitido (1-10)
12	Disparo Repetido	Se ha repetido un disparo volviendo a enviar las mismas coordenadas

Las excepciones 2, 11 y 12 debería ser interpretada por el árbitro como mal funcionamiento del servicio-jugador que la origina y por tanto darle la partida por perdida.

3.5 Pruebas de Funcionamiento

Se proporcionará al alumno un cliente simple para que pueda realizar los tests que crea necesarios.

3.6 Evaluación

Con la fecha que se indique como límite el alumno entregará un informe en el que se contengan dos partes. Por un lado el esquema funcional de la aplicación y por otro el código completo.

Tanto el trabajo como el informe podrán realizarse en grupos de 2 si bien las entrevistas se realizarán individualmente.

Posteriormente deberá acudir a una entrevista personal en la que se le plantearán cuestiones relacionadas con la implementación y con los sockets.

Del informe y la entrevista personal, así como de los posibles informes de otras prácticas se extraerá la nota de prácticas para la asignatura.

ANEXO I. ALGUNOS PUERTOS BIEN CONOCIDOS

Keyword	Decimal	Description	References
Echo	7/tcp	Echo	
Daytime	13/tcp	Daytime	
Msp	18/tcp	Message Send Protocol	
ftp-data	20/tcp	File Transfer	[Default Data]
ftp	21/tcp	File Transfer	[Control]
telnet	23/tcp	Telnet	
smtp	25/tcp	Simple Mail Transfer	
time	37/tcp	Time	
nameserver	42/tcp	Host Name Server	
nicname	43/tcp	Who Is	
login	49/tcp	Login Host Protocol	
re-mail-ck	50/tcp	Remote Mail Checking Protocol	
domain	53/tcp	Domain Name Server	
xns-auth	56/tcp	XNS Authentication	
xns-mail	58/tcp	XNS Mail	
sql*net	66/tcp	Oracle SQL*NET	
bootps	67/tcp	Bootstrap Protocol Server	
bootpc	68/tcp	Bootstrap Protocol Client	
tftp	69/tcp	Trivial File Transfer	
gopher	70/tcp	Gopher	
netrjs-1	71/tcp	Remote Job Service	
deos	76/tcp	Distributed External Object Store	
finger	79/tcp	Finger	
www-http	80/tcp	World Wide Web HTTP	
xfer	82/tcp	XFER Utility	
kerberos	88/tcp	Kerberos	
npp	92/tcp	Network Printing Protocol	
rtelnet	107/tcp	Remote Telnet Service	
pop2	109/tcp	Post Office Protocol - Version 2	
pop3	110/tcp	Post Office Protocol - Version 3	
sunrpc	111/tcp	SUN Remote Procedure Call	
auth	113/tcp	Authentication Service	
uucp-path	117/tcp	UUCP Path Service	
sqlserv	118/tcp	SQL Services	
nntp	119/tcp	Network News Transfer Protocol	
dls	197/tcp	Directory Location Service	
dls-mon	198/tcp	Directory Location Service Monitor	

ANEXO II. Especificación del XML-RPC

Esta especificación documenta el protocolo XML-RPC usado en UserLand Frontier 5.1.

Para una explicación no técnica, véase “XML-RPC for Newbies”.

Esta página provee de toda la información que el implementador necesita.

Resumen

El XML-RPC es un protocolo de llamadas a procedimientos remotos que se trabaja a través Internet.

Un mensaje XML-RPC es una petición HTTP-POST. El cuerpo de esta petición es un XML. Un procedimiento se ejecuta en el servidor y el valor devuelto también está formateado en XML.

Los parámetros del procedimiento pueden ser escalares, números, cadenas de caracteres, datos, etc.; y también pueden ser complejas estructuras de registros y listas.

Ejemplo de petición

Éste es un ejemplo de petición XML-RPC:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181

<?xml version="1.0"?>
<methodCall>
  <methodName>examples.getStateName</methodName>
  <params>
    <param>
      <value><i4>41</i4></value>
    </param>
  </params>
</methodCall>
```

Requerimientos de la cabecera

El formato del URI en la primera linea de la cabecera no está especificado. Por ejemplo, ésta podría estar vacía, tener una simple barra, si el servidor sólo maneja llamadas XML-RPC. Sin embargo, si el servidor trabaja con una mezcla de peticiones HTTP, se permite el URI para ayudar a enrutar la petición al código que manipula las peticiones XML-RPC. (En el ejemplo, el URI es /RPC2, indica al servidor que entrute la petición al encargado de reponder el “RPC2”.)

Un User-Agent (Agente-de-Usuario) y un Host deben ser especificados.

El Content-Type (tipo de contenido) es text/xml.

La Content-Length (longitud del contenido) debe ser especificada y debe ser correcta.

Formato del cuerpo

El formato del cuerpo esta en XML, en una simple estructura <methodCall>.

El <methodCall> debe contener una subetiqueta <methodName>, que contiene un string con el nombre del método que será llamado. El string debe sólo contener los caracteres del identificador, mayúsculas y minúsculas, los caracteres numéricos, 0-9, subrayado, punto, coma y barra. Queda completamente a cargo del servidor el cómo interpretar los caracteres contenidos en el methodName.

Por ejemplo, el methodName podría ser el nombre de un fichero que contenga un script que se ejecuta en cuando le llega la petición. Podría ser el nombre de un campo en la tabla de la base de datos. O podría ser la dirección de un fichero contenido en una jerarquía de carpetas y ficheros.

Si la llamada al procedimento contiene parámetros, el <methodCall> debe contener una subetiqueta <params>. Esta subetiqueta <params> puede, a su vez, contener cualquier número de subetiquetas <param>, cada una de las cuales tiene una subetiqueta <value>.

<value>s (valores escalares)

Los valores <value> pueden ser escalares, el tipo se indica junto al valor dentro de uno de los tipos listados en esta tabla:

Etiqueta	Tipo	Ejemplo
<i4> o <int>	Entero de cuatro bytes con signo	-12
<boolean>	0 (falso) ó 1 (verdadero)	1
<string>	Cadena de caracteres ASCII	Hola mundo
<double>	Número de coma flotante con doble precisión	-12.214
<dateTime.iso8601>	fecha/hora	19980717T14:08:55
<base64>	Binario codificado en base64	eW91IGNhbid0IHJlYWQgdGhpcyE=

Si no se indica, el tipo es una cadena de caracteres.

<struct>s

Un valor puede ser también del tipo <struct>.

Un <struct> contiene <member>s y cada <member> contiene un <name> y un <value>.

Éste es un ejemplo de un <struct> de dos elementos:

```
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
    </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>
```

Los <struct>s pueden ser recursivos, cada <value> puede contener otro <struct> o cualquier otro tipo, incluido un <array>, descrito abajo.

<array>s

Un valor puede también ser del tipo <array>.

Un <array> contiene un único elemento <data>, el cual puede contener cualquier número de <value>s.

Éste es un ejemplo de un array de cuatro elementos:

```
<array>
  <data>
    <value><i4>12</i4></value>
    <value><string>Egypt</string></value>
    <value><boolean>0</boolean></value>
    <value><i4>-31</i4></value>
  </data>
</array>
```

Los elementos del <array> no tienen nombres.

Se pueden mezclar los tipos como el anterior ejemplo ilustra.

Los <arrays>s pueden ser recursivos, cualquier valor puede contener un <array> o cualquier otro tipo, incluido el <struct>, descrito arriba.

Ejemplo de respuesta

Éste es un ejemplo de respuesta a una petición XML-RPC:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>South Dakota</string></value>
    </param>
  </params>
</methodResponse>
```

Formato de la respuesta

A menos que hubiese un error de bajo nivel, siempre devuelve un 200 OK.

El Content-Type (tipo de contenido) es text/xml. Content-Length (longitud del contenido) debe estar presente y ser correcto.

El cuerpo de la respuesta es una simple estructura XML, una <methodResponse> (respuesta del proceso remoto), que puede contener un solo <params> el cual contiene un solo <param> el cual contiene un <value>.

El <methodResponse> también puede contener un <fault> (fallo) el cual contiene un <value> que es un <struct> que contiene dos elementos, uno llamado <faultCode>(código de fallo), un <int> (entero) y otro llamado <faultString>(explicación en caracteres del fallo), del tipo <string>.

Un <methodResponse> no puede contener a la vez un <fault> y un <params>.

Ejemplo de fallo

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT

<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>faultCode</name>
          <value><int>4</int></value>
          </member>
        <member>
          <name>faultString</name>
          <value><string>Too many parameters.</string></value>
         </member>
      </struct>
```

```
    </value>
  </fault>
</methodResponse>
```

Estategias/Objetivos

Firewalls. El objetivo de este protocolo es extender una comunicación compatible sobre diferentes entornos, no hay un poder nuevo por debajo(oculto) de las capacidades de los CGI. Los programas firewall pueden vigilar dentro de los POSTs cuyo contenido es texto/xml.

Discoverability.(rápido de entender) Nosotros queríamos un formato limpio, extensible y muy simple. Debía ser posible para un codificador de HTML ser capaz de mirar en un fichero que contuviese una llamada a un procedimiento XML-RPC, entender qué está haciendo, ser capaz de modificarlo y tenerlo funcionando en el primer o el segundo intento.

Fácil de implementar. También queríamos un protocolo sencillo de implementar que pudiese ser rápidamente adaptado para ejecutarse en otros entornos u otros sistemas operativos.

Actualizado 21/1/99

Las siguientes preguntas se hicieron en el UserLand discussion group por ser el XML-RPC implementado en Python.

- La Sección del Formato de Respuesta dice “El cuerpo de la respuesta es una única estructura XML, un <methodResponse>, que puede contener un único <params>..." Esto es confuso. ¿Podemos quitar el <params>?

  No, no se puede quitar si el procedimiento se ejecutó con éxito. Sólo hay dos opciones, o la respuesta contiene una estructura <params> o contiene una estructura <fault> . Por eso hemos usado la palabra “puede” en la frase.

- ¿Es el "booleano" un tipo de dato distinto, o pueden los valores booleanos ser intercambiado por enteros (p. ej.: cero=falso, distinto-de-cero=verdadero)?

  Si, los booleanos forman un tipo distinto de datos. Algunos lenguajes/entornos lo permiten para una fácil conversión desde el cero al falso y uno al verdadero, pero si se desea poner verdadero, se debe enviar un tipo booleano con el valor true(verdadero), por lo que tú intento no tiene posibilidad de ser malentendido.

- ¿Cuál es la sintaxis legal (y el rango) de los enteros? ¿Cómo se manejan los ceros delanteros (0000 por 0)? ¿Son permitidos los ceros delanteros con signo? ¿Cómo se manejan los espacios en blanco?

  Un entero es un número de 32-bits con signo. Se puede incluir un más o un menos al principio de la cadena de caracteres numéricos. Los ceros delanteros desaparecen. No se permiten los espacios en blanco. Sólo los caracteres numéricos precedidos de un más o un menos.

- ¿Cuál es la sintaxis legal (y el rango) de los números con coma flotante (dobles)? ¿Cómo se representa el exponente? ¿Cómo se maneja el espacio en blanco? ¿Pueden el infinito o un “no número” ser representados?

  No hay representación para el infinito positivo o negativo ni para el “no número”. Esta vez, sólo se permite notación decimal, un más o un menos, seguidos por un número de caracteres numéricos, seguidos por un punto y cualquier número de caracteres numéricos. El espacio en blanco no está permitido. El rango de valores permitidos es dependiente de la implementación, no está especificado.

- ¿Qué caracteres se permiten en las cadenas de caracteres? ¿Caracteres no imprimibles? ¿Caracteres nulos? ¿Puede un “string” ser usado para una cadena arbitraria de datos binarios?

Cualquier carácter es permitido en un string excepto < y &, que se codifican como < y &. Un string puede ser usado para codificar datos binarios.

- ¿Proteje el elemento "struct" el orden de las claves? O, en otras palabras, ¿es o no "foo=1, bar=2" equivalente a "bar=2, foo=1"?

  El elemento struct no preserva el orden de las claves. Los dos structs son equivalentes.

- ¿Puede el struct <fault> contener otros elementos que el <faultCode> y el <faultString>? ¿Existe una lista global de códigos de fallo (Tal que puedan ser mapeadas a distintas excepciones para lenguajes como Python y Java)?

  Un struct <fault> no debe contener otros miembros que los especificados. Eso es cierto para todas las otras estructuras. Nosotros creemos que la especificación es suficientemente flexible para que todas las transferencias de datos razonables puedan ser acomodadas en las estructuras especificadas. Sí realmente crees que no es cierto, por favor envía un mensaje al grupo de discusión.

  No hay una lista global de los códigos de fallo. Está en manos del implementador del servidor, o de los estándares de más alto nivel especificar los códigos de fallo.

- ¿Qué zona horaria debe ser asumida por el tipo dateTime.iso8601? ¿UTC? ¿hora local?

  No asumas una zona horaria. Debe ser especificada por el servidor en su documentación indicando qué suposiciones hace sobre la zona horaria.

Añadidos

- Tipo <base64> 21/1/99 DW.

Derechos de autor y restricciones

ANEXO III. REFERENCIAS.

- http://ditec.um.es/laso/docs/tut-tcpip/
  Tutorial y descripción técnica de TCP/IP
- http://ditec.um.es/laso/docs/
  Otros documentos de IPC (interprocess communication)
- http://www.arrakis.es/~dmrq/beej/home.htm
  Guía Beej de Sockets en español.
  (y la original en: http://www.ecst.csuchico.edu/~beej/guide/net/)
- http://www.developerweb.net/forum/viewforum.php?f=53
  UNIX Sockets FAQs
- http://www.scit.wlv.ac.uk/~jphb/comms/sockets.html
  Sockets Programming. Resumen de las funciones.
- http://lowtek.com/sockets/
  Spencer's Socket Site: Network Programming with Sockets. Reunión de direcciones útiles
- http://jungla.dit.upm.es/~jmseyas/linux/mcast.como/Multicast-Como.html
  Multicast. Las secciones anteriores sobre multicast están extraidas de aquí.
- http://jungla.dit.upm.es/~jmseyas/linux/mcast.lj/mcast-lj.html
  Multicast: From Theory to Practice. Incluye un ejemplo de multicast.
- http://www.linuxfocus.org/Castellano/January2001/article144.shtml
  Multicast. Incluye ejemplo de multicast tanto 'emisor' como 'receptor'.
- http://www3.uji.es/~ochera/curso_2003_2004/e52/
  Material variado, con ejemplos de comunicaciones unicast.
- http://www.w3.org/Protocols/rfc2616/rfc2616.html
  Protocolo HTTP (ingles)
- http://cdec.unican.es/libro/HTTP.htm
  Protocolo HTTP (español)
- http://bulma.net/body.phtml?nIdNoticia=1682
  Especificación XML-RPC (español)
- http://www.xmlrpc.com
  Especificación XML-RPC y mucha más documentación (inglés)