Formatos de mensajes de texto

Redes de Comunicaciones - Curso 2024/25

Introducción

Durante esta sesión vamos a analizar distintas formas posibles para especificar un protocolo de comunicación. No hay una forma óptima de llevar a cabo la especificación de un protocolo puesto que esta dependerá del propósito del mismo, de los dispositivos involucrados, de las propiedades de la conexión de red, etc. Para un mismo protocolo puede haber distintas formas correctas de llevar a cabo la especificación.

El diseño del protocolo puede dividirse en dos partes bien diferenciadas, que no necesariamente deben realizarse por este orden:

- Definir el formato de los mensajes intercambiados entre las entidades participantes.
- Definir la temporización o secuencia correcta de intercambio de dichos mensajes, que normalmente suele especificarse mediante modelos computacionales más complejos como son, por ejemplo, los autómatas de estados finitos.

En este boletín nos centraremos en el formato de los mensajes, concretamente en el uso de mensajes basados en texto. Dicho formato textual será el que los alumnos deben utilizar a la hora de definir los mensajes de su protocolo de comunicación con el directorio del sistema *nanoFiles*.

Los *peers* se comunican con el servidor de directorio para diferentes fines, tales como consultar la lista de ficheros compartidos por otros *peers*, publicar los ficheros que desean compartir, obtener las direcciones de los servidores que sirven un determinado fichero, etc. Por tanto, el objetivo de esta sesión es ir definiendo los mensajes que involucran al directorio.

Hemos de tener en cuenta consideraciones que afectarán a la definición y al formato de los mensajes:

- En los mensajes **no se especifican las direcciones IP** de quienes actúan como cliente y directorio, porque están implícitas en la comunicación de los niveles inferiores.
- El directorio estará a la escucha en una **IP y puerto conocido a priori** por los clientes.
- El protocolo de comunicación con el directorio usará un **canal UDP no confiable**, por lo que necesitará confirmación de mensajes, y usará un flujo de datos basado en **parada y espera**. Aunque el canal sea no confiable, sí que asumiremos que ofrece ciertas garantías, como que no entrega paquetes duplicados o reordenados.

Mensajes en formato *field:value*

Aunque existen varias posibilidades a la hora de especificar el formato de los mensajes de texto, en este boletín nos centraremos en el **formato campo:valor** (*field:value*). El formato *field:value* utiliza delimitadores basados en texto y facilita la legibilidad de los mensajes, es decir, son fáciles de interpretar a simple vista. Este formato es usado en protocolos tan importantes como SMTP. Otros protocolos como HTTP también usan el formato de texto mediante una representación basada en lenguaje de marcas (en lugar de *field:value*), mientras que FTP puede usar tanto el formato binario como el de texto ASCII dependiendo del

fichero a intercambiar. Un ejemplo de uso de *field:value* es el siguiente mensaje, que se podría usar para comprobar si el directorio utiliza un protocolo compatible:

```
operation: ping\n
protocol: 123456789A\n
\n
```

- Cada línea es un campo del mensaje, con un nombre determinado (generalmente representativo de la información que representa) y su valor asociado.
- El primer campo suele indicar el significado del mensaje. Así, en el ejemplo anterior, el significado del campo "operation" indica que se trata de un mensaje de *ping*, como el que se podría enviar al directorio para comprobar que es compatible con nuestro protocolo.
- Al final de cada línea debe enviarse un retorno de carro ('\n') para marcar el final de la línea, mientras que una línea vacía (con sólo un retorno de carro) marca el final del mensaje.
- Puesto que existe una forma explícita de indicar el fin del mensaje, el número de campos (líneas) puede ser variable, por lo que un determinado campo puede aparecer varias veces, o no hacerlo en absoluto.
- En este tipo de mensajes debe evitarse el uso de los delimitadores como parte del valor del propio texto del campo (en nuestro ejemplo, el retorno de carro '\n'). Existen varias técnicas para resolver este problema, por ejemplo, excluir el retorno de carro de los caracteres posibles de uso dentro del texto o codificar el retorno de carro por otro carácter no usado en el texto de los mensajes (por ejemplo, &, # ó %).

Un inconveniente de este tipo de formato es que no permite anidar elementos, si bien existen formas alternativas de representar datos compuestos. Por ejemplo, el siguiente mensaje podría usarlo el directorio para enviar a un cliente de NanoFiles la lista de ficheros que otros *peers* han publicado en el directorio:

```
operation: availableFiles\n
files: document.pdf,letter.odt,image.jpg\n
\n
```

En caso de usar un carácter como delimitador de elementos que conforman el valor de un campo compuesto, se ha asegurar que dicho delimitador no aparece como valor de ningún elemento. En el ejemplo anterior en el que el valor del campo es una lista de nombres de fichero separados por comas, los nombres de dichos ficheros no podrán incluir comas y, en caso de que sea necesario, dicho carácter ',' se habrá de codificar de forma especial para no interpretarlo como un delimitador cuando no corresponda.

A la hora de representar datos compuestos, también puede resultar útil que un mismo campo aparezca en repetidas ocasiones en el mismo mensaje. En este caso, el orden de los campos puede usarse para establecer la correspondencia entre los diferentes valores que conforman un mismo dato compuesto:

```
operation: availableFiles\n
filename: document.pdf\n
```

```
size: 142544\n
filename: letter.odt\n
size: 41524\n
filename: image.jpg\n
size: 2104214\n
\n
```

Implementación de los mensajes en el proyecto NanoFiles

En este apartado se proporcionan indicaciones generales acerca de cómo crear y procesar los mensajes transmitidos entre el cliente y el servidor de directorio a través de datagramas.

En el marco del proyecto de NanoFiles, el paquete `nanoFiles.udp.message` contiene la clase `DirMessage.java`, que esboza las líneas generales acerca de cómo se podrían implementar mensajes textuales definidos en el protocolo de comunicación con el directorio.

Creación de mensajes `DirMessage` para su posterior envío

La clase `DirMessage` contendrá como atributos los valores de los campos de los mensajes del protocolo, tales como el tipo de operación, metadatos de ficheros (nombre, tamaño, hash), direcciones de socket de un servidor, etc. Por tanto, será necesario añadir un atributo a la clase por cada uno de los campos de los diferentes tipos de mensajes definidos para este protocolo. En el código de partida ya existe, entre otros, un campo `operation` que se debe usar para indicar en el constructor el tipo de mensaje a crear. Tras crear un mensaje de un tipo de operación dado, el código "cliente" de la clase `DirMessage` (es decir, las clases `NFConnector` y `NFDirectoryServer`) se encargará de establecer mediante métodos *setter* los atributos necesarios con la información que se desea enviar en el mensaje.

NOTA: Es recomendable que los métodos *setter* para establecer los atributos del objeto `DirMessage` comprueben que dicho atributo es relevante para el tipo de mensaje en cuestión, con el fin de detectar cuándo se está tratando de establecer un atributo que no corresponde a ningún campo definido para el tipo de mensaje.

Conversión de `DirMessage` a `String` para envío de mensajes textuales

Los mensajes de este protocolo siempre van a ser transmitidos como cadenas de caracteres. Para ello tendremos que implementar en la clase `DirMessage` el método `toString()` que se encargue de generar realmente el mensaje en formato `String` a partir de los valores de los atributos del objeto de la clase `DirMessage`. La forma de hacerlo dependerá del formato de mensaje que hayamos elegido, tal como se ha visto en la sección anterior.

Los protocolos basados en *field:value* se caracterizan porque sus mensajes están compuestos por líneas en las que aparece el identificador del campo y el valor del mismo, separados por el carácter dos puntos. Para construir una cadena con el mensaje en este formato tendremos que ir generando, línea por línea, los distintos campos que queremos transmitir con sus correspondienes valores. Para la construcción línea por línea podemos hacer uso de un `StringBuffer`. Supongamos que queremos enviar un mensaje que tiene dos campos. El código de abajo nos da un ejemplo de cómo hacerlo:

```
final char DELIMITER = ':';     //Define el delimitador
final char END_LINE = '\n';     //Define el carácter de fin de línea
StringBuffer sb = new StringBuffer();
String field1 = "operation";     //Nombre del campo
String value1 = "ping";  //Valor del campo
String field2 = "protocol";     //Nombre del campo
String value2 = "123456789A";  //Valor del campo

sb.append(field1+DELIMITER+value1+END_LINE); //Construimos el campo
sb.append(field2+DELIMITER+value2+END_LINE); //Construimos el campo
sb.append(END_LINE);  //Marcamos el final del mensaje
String message = sb.toString(); //Se obtiene el mensaje
```

Creación de objetos `DirMessage` a partir de un `String` recibido

La clase `DirMessage` cuenta con un método de clase llamado `fromString(String str)` que será utilizado por el extremo receptor del mensaje. Este método se deberá encargar, partiendo de una cadena de caracteres recibida en un datagrama, de interpretar su contenido y construir un objeto de la clase `DirMessage` cuyos atributos contendrán los valores de los campos extraídos del mensaje. La cadena habrá sido generada mediante el método `toString` antes de ser enviada en el datagrama por el emisor. Así, el método `fromString` nos permite crear un objeto de dicha clase `DirMessage`, en el cual los valores de los distintos campos del mensaje contenido en el `String` hayan sido guardados en los correspondientes atributos del objeto.

Existen diversas alternativas para procesar los mensajes de un protocolo basado en *field:value*. Una de las más elementales es usar el método `split()` de la clase `String` con el delimitador adecuado ('\n') para obtener un array con todas la líneas, sobre el que ir iterando posteriormente.

A continuación dejamos un extracto de código que nos permitiría procesar

```
        String[] lines = message.split(END_LINE + "");
        for (String line : lines) {
            int idx = line.indexOf(DELIMITER); // Posición del delimitador
            String fieldName = line.substring(0, idx).toLowerCase(); //
minúsculas
            String value = line.substring(idx + 1).trim();
            // DO SOMETHING WITH fieldName AND value
            ...
        }
```

Ejercicios propuestos

Para avanzar en el desarrollo de la práctica, has de tener en cuenta la funcionalidad prevista para el directorio, según los requisitos de la práctica. Para más detalles, consulta el documento de enunciado de *NanoFiles* con la especificación, disponible en el Aula Virtual.

A continuación se proponen una serie de ejercicios para avanzar en el diseño e implementación de la práctica.

1. Haciendo uso de un documento de texto plano, **diseña** el formato del mensaje necesario para comprobar que el directorio está a la escucha y es compatible con nuestro protocolo.

    1. Debes definir los campos (líneas en formato *field:value*) que contiene cada mensaje, empezando por la operación que lleva a cabo. Indica entre otros aspectos quién envía y quién recibe dicho mensaje, su función, en qué situaciones se envía, y qué acciones se llevan a cabo cuando se recibe. Pon ejemplos de cada mensaje, usando valores ficticios para ello como se ha visto anteriormente en el boletín.
    2. Define el formato del mensaje de respuesta que contendría la confirmación que el directorio está a la escucha y es compatible. Contempla la posibilidad de que el *ping* pueda fallar, por ejemplo, por tratarse de un *protocolId* que corresponda a otro grupo de prácticas.

2. Observa el código de partida para modelar los mensajes en el proyecto *NanoFiles*.

    1. Observa que `DirMessage` ya tiene un atributo `operation` de tipo String, que representa el primer campo del mensaje, con el tipo de operación. Su valor por defecto es `DirMessageOps.OPERATION_INVALID` (operación no válida), el cual se usa para indicar que un objeto (mensaje) está sin inicializar.
    2. En la clase `DirMessageOps` verás que también hay una constante de clase String llamada `OPERATION_PING` con el valor "ping"; puedes utilizarla como el valor que debe tomar el atributo `operation` al crear el tipo de mensaje que debe ser enviado al directorio para realizar un *ping*.
    3. En `DirMessage` también hay otro atributo `protocolId`, que podría usarse para albergar el identificador de protocolo en un mensaje cuya operación sea *ping*.

> **NOTA: Recuerda desactivar `NanoFiles.testModeUDP` a partir de este momento**

1. Implementa la funcionalidad necesaria en los métodos `toString` y `fromString` de la clase `DirMessage`, de forma que ambos métodos sean capaces de manejar el tipo de mensaje que el cliente *NanoFiles* enviará al directorio al hacer un *ping*.

2. Tras completar el boletín de prácticas anterior, tu código de `pingDirectoryRaw` debería enviar la cadena de caracteres "ping&protocolId" y comprobar que se recibe "welcome" como respuesta si el protocolo que usan ambos programas, `NanoFiles` y `Directory`, es compatible.

    1. Basándote en el código de `pingDirectoryRaw`, implementa el método `pingDirectory` de la clase `DirectoryConnector` para reemplazar el envío de la cadena de caracteres "ping&protocolId" por un mensaje de *login* bien formado, que siga el formato *field:value*, utilizando el código desarrollado en la clase `DirMessage`.

2. A la hora de enviar un mensaje, en lugar de crear directamente un String con los valores, debes crear un objeto `DirMessage` del tipo (operación) adecuado, establecer los campos adicionales necesarios (p.ej. establecer el valor del atributo *protocolId* usando el método setter* adecuado), y finalmente convertir el objeto `DirMessage` a String, cuyos bytes serán enviados en un datagrama.

3. Tras completar el boletín de prácticas anterior, tu código del servidor (`sendResponseTestMode`) actualmente debería comprobar que se ha recibido la cadena "ping&protocolId", en cuyo caso envía la cadena de caracteres "welcome".

    1. Basándote en el código del método `sendResponseTestMode` de la clase `NFDirectoryServer` que has programado en el boletín anterior, implementa el método `sendResponse` .
    2. En lugar de acceder a los datos de un mensaje *en crudo*, a la hora de procesar los datos del mensaje debes crear un objeto `DirMessage` a partir de la cadena de caracteres recibida en el datagrama (usando el método de clase `DirMessage.fromString()`). Una vez creado dicho objeto, podrás acceder mediante métodos *getter* a los valores de los campos del mensaje para realizar el procesamiento: obtener el tipo de operación, o el valor del atributo *protocolId* (por ejemplo, para comprobar si en un mensaje de *ping*, el protocolo que usa el cliente es compatible con el del directorio).

4. Establece puntos de ruptura en los métodos `fromString` y `toString` que acabas de programar, y depura tu código ejecutando paso a paso tanto servidor (*Directory*) como cliente (*NanoFiles*), hasta comprobar que el servidor, a partir de los datos recibidos en el datagrama, construye un objeto `DirMessage` con valores idénticos al que creó el cliente antes de enviarlo como cadena de caracteres.

5. Sigue los `TODO` del método `sendResponse` de la clase `NFDirectoryServer` para terminar de programar la lógica asociada al procesamiento de un *ping* en el directorio.

    1. Debe actuar de una u otra forma en función del tipo y datos del mensaje recibido (sacando dichos datos de un objeto `DirMessage` construido previamente con `fromString`). En general, la lógica de `sendResponse` deberá consultar en caso necesario las estructuras de datos del servidor para devolver una respuesta, y actualizarlas con los datos recibidos cuando corresponda. En el caso del *ping*, no es necesario consultar ni actualizar ninguna estructura de datos en el directorio.
    2. Debe construir un mensaje `DirMessage` , con el tipo de operación y valores adecuados en función del resultado de la solicitud (éxito/fracaso). En el caso del mensaje de respuesta a un *ping* compatible, el directorio deberá enviar un mensaje de bienvenida que sirva de confirmación acerca de la compatibilidad, invitando al cliente a solicitar otros servicios de este directorio.
    3. El objeto `DirMessage` de respuesta construido deberá ser convertido a String y enviado en un datagrama.

6. Termina de implementar el método `pingDirectory` de la clase `DirectoryConnector` para procesar la respuesta a la solicitud de inicio de sesión mediante un bien formado, que siga el formato *field:value*, implementado en la clase `DirMessage` .

    1. Tras completar el boletín de prácticas anterior, tu código de `pingDirectoryRaw` comprueba que se ha recibido la cadena "welcome".
    2. Basándote en lo anterior, en `pingDirectory` deberás crear un objeto `DirMessage` mediante `DirMessage.fromString()`, usarlo para comprobar si el ping tuvo éxito para imprimir por pantalla el resultado de la operación.

7. Una vez que hayas realizado los pasos anteriores, habrás completado exitosamente tu primera *conversación* con el directorio, con lo que ya deberías estar suficientemente familiarizado con la estructura básica del proyecto y los métodos involucrados en la comunicación UDP entre cliente y servidor. Así pues, **continúa diseñando e implementando el resto de mensajes** necesarios en tu protocolo, para ir dotando de funcionalidad al programa.

    1. Para incorporar la funcionalidad mínima exigida al programa *NanoFiles*, diseña los mensajes para obtener la lista de ficheros publicados al directorio por otros *peers*, enviar al directorio la lista de ficheros que se desea compartir, y obtener del directorio la lista de servidores que comparten ficheros cuyo nombre contiene una determinada subcadena. Después, implementa la lógica requerida por los comandos `filelist`, `serve` y `download` que consistirá en el envío y recepción de tales mensajes (en el caso de servir y descargar ficheros, preocúpate sólo de la comunicación necesaria con el directorio).
        - En el caso de `filelist`, necesitas implementar el método `getFileList` de `DirectoryConnector`, enviando el tipo de mensaje adecuado que hayas diseñado para tu protocolo, y recibiendo su correspondiente respuesta.
        - En el caso de `serve`, necesitas implementar el método `registerFileServer` de `DirectoryConnector`, enviando el tipo de mensaje adecuado que hayas diseñado para tu protocolo, y recibiendo su correspondiente respuesta.
        - En el caso de `download`, necesitas implementar el método `getServersSharingThisFile` de `DirectoryConnector`, enviando el tipo de mensaje adecuado que hayas diseñado para tu protocolo, y recibiendo su correspondiente respuesta.