Servidor de archivos: Conexión con un cliente en Java

Felipe Benítez Armando Daza Martín Lomelí David F. Monjaraz

IPN — ESCOM
Sistemas Distribuidos, equipo 6, 4CM2

19 de octubre de 2016

§ 1. Introducción

En este documento se verá una propuesta para codificar un cliente de descarga de archivos en lenguaje Java, capaz de comunicarse con un servidor de archivos codificado en C++. Asimismo, se presentarán los puntos finos en los que es preciso mostrar atención, para asegurar el envío y la recepción de datos de forma correcta.

§ 2. Desarrollo

§ 2.1. ¿Cuál es el problema?

Se tiene un servidor escrito en C++ que envía y recibe datos a través de mensajes codificados en estructuras. Los mensajes almacenan información concerniente al archivo a leer, el pedazo de archivo que se desea obtener y los bytes que se necesitan leer. El servidor responde con un mensaje análogo, conteniendo un código indicando el éxito o fracaso de la lectura, la cantidad de bytes que se leyeron realmente y los datos que realmente se leyeron.

El problema consiste en escribir un programa cliente en lenguaje Java capaz de comunicarse con el servidor. Aunque se escucha simple, el problema revela varios puntos finos, resultado de la forma de hacer las cosas en ambos lenguajes, que se necesitan estandarizar para asegurar la conexión y recibir los datos con el mínimo de errores.

§ 2.2. Los primeros pasos

Los métodos que a lo largo del curso se han programado en C++, con la finalidad de facilitar la conexión y el envío y recepción de datos mediante UDP, resultan ser similares a sus análogos en Java. Las clases `DatagramSocket` y `DatagramPacket` son equivalentes a las de C++ `SocketDatagrama` y `PaqueteDatagrama`, respectivamente. Ambas cuentan con métodos para enviar y recibir arreglos de bytes[1], e incluso Java ofrece el método `setSoTimeout()`, que permite especificar un temporizador para detener la escucha una vez transcurrido el tiempo especificado.

Por su parte, la creación del archivo a partir de los datos recibidos puede llevarse a cabo mediante la clase `RandomAccessFile`. No se recomienda utilizar la combinación `FileOutputStream` + `DataOutputStream`, debido a que ambas clases sólo son capaces de escribir *secuencialmente*. La clase `RandomAccessFile` cuenta con el método `seek()`, que permite posicionar el "cabezal de escritura" del archivo en una posición específica. Con lo anterior se reducen los errores en la creación del archivo, debidos a paquetes recibidos en desorden o duplicados.

Por todo lo anterior, se deduce que el código empleado en Java para establecer la conexión, solicitar los pedazos, recibirlos (tomando en cuenta retardos circunstanciales, como sobrecarga en el servidor o señal inalámbrica débil) y escribirlos en el archivo, debe ser equivalente al código generado en la versión de C++, presentada en una práctica anterior. Lo que falta por resolver es una cuestión de compatibilidad: *¿cómo representar estructuras de C++ con recursos disponibles en Java?*

[1] En C++ se usa el tipo de dato `char`, típicamente del mismo tamaño.

§ 2.3. El problema de las estructuras

Una estructura en C se conforma por una serie de elementos, con un tamaño específico cada uno, y almacenados de forma contigua en memoria, de forma tal que permiten definir "paquetes" de datos, facilitando su transporte y obtención entre métodos y programas. El tamaño de una estructura es como mínimo igual a la suma de los tamaños de cada miembro, y el orden especificado en su declaración determina el orden de aparición de los datos en memoria.

Se muestran de ejemplo las estructuras de la figura 1, definidas para los mensajes de ida y regreso de la práctica:

```
/* Formato del mensaje que viaja del
 ↪  cliente hacia el servidor */
struct messageCS {
  uint32_t opcode;
  uint32_t count;
  uint32_t offset;
  char name[MAX_PATH];
};
```

```
/* Formato del mensaje que viaja del
 ↪  servidor hacia el cliente */
struct messageSC {
  uint32_t count;
  uint32_t result;
  char data[BUF_SIZE];
};
```

Figura 1: Estructuras de datos de mensajes.

A pesar de que las estructuras aglutinan miembros juntos en memoria, no se puede afirmar siempre que todos los miembros están "pegados", uno después del otro. Raymond [1] menciona que existe un detalle denominado *alineación de datos*, que exige a las estructuras tamaños múltiplos de una unidad fundamental (como el tamaño de una variable numérica entera). Para la figura 1, el orden en los tipos de datos hace que se cumpla la alineación para múltiplos de 4 bytes (el tamaño de un entero). Además, los arreglos de tipos de datos primitivos no contienen relleno después de los mismos, por lo que se puede asegurar que después de la cadena no hay nada. Lo anterior hace que sea innecesario prestar atención a la alineación de datos, pero de todos modos, es importante conocer que existe para el desarrollo de futuros programas.

Lo anterior permite deducir una propiedad muy importante de las estructuras: *sus miembros se colocan en orden, uno después del otro, en la memoria.* Como ejemplo de lo anterior, y utilizando el código propuesto por Foo [2], se presentan los arreglos de bytes de las estructuras de la figura 1, cada una llenada con datos arbitrarios:

```
Estructura messageCS: { 2 , 0 , 3000 , "psetup.xpm" }

02000000 00000000 b80b0000 70736574 75702e78 706d0000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
/* 6 líneas idénticas a la anterior */
00000000 00000000 00000000 2f

Estructura messageSC: { 12 , 0 , "Hello World!" }

0c000000 00000000 48656c6c 6f20576f 726c6421 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
/* 29 líneas idénticas a la anterior */
00000000 00000000 00000000 00000000
```

Figura 2: Representación en bytes de ejemplos de estructuras.

Por lo tanto, la solución en Java deberá *agrupar los datos* de la misma forma en la que la de C++ lo hace, con el fin de establecer comunicación de forma exitosa con el servidor.

§ 2.4. ¡A trabajar!

Se emplearán *clases* como lo más parecido a las estructuras de C++, pues una clase aglutina elementos de diferente tipo. Se aprovechará el hecho de que una clase puede contener métodos propios, con el afán de proveerle lo necesario para convertirse en un arreglo de bytes que siga la convención vista arriba, así como el camino inverso—dado un arreglo de bytes, reconstruir la clase correspondiente.

Lo más importante de la clase se mostrará explícitamente: el código de conversión entre la clase y el arreglo de bytes. El código específico de la clase `MessageCS` para llevar a cabo la conversión de un arreglo de bytes a una clase se muestra en el listado 1, mientras que el proceso inverso se encuentra en el listado 2. Para la clase `MessageSC` el proceso completo es similar.

```
public static MessageCS getClassFromBytes(byte[] buf) {
    ByteBuffer bb = ByteBuffer.wrap(buf);
    bb.order(ByteOrder.LITTLE_ENDIAN);

    int opcode = bb.getInt();
    int count = bb.getInt();
    int offset = bb.getInt();

    StringBuilder sb = new StringBuilder();

    byte nameByte;
    while ( (nameByte = bb.get()) != '\0' )
        sb.append((char) nameByte);

    return new MessageCS(opcode, count, offset, sb.toString());;
}
```

Listado 1: Conversión `byte[]` → `MessageCS`.

```
public byte[] getByteRepr() {
    ByteBuffer bb = ByteBuffer.allocate(3 * Integer.BYTES + MAX_PATH);
    bb.order(ByteOrder.LITTLE_ENDIAN);

    bb.putInt(this.opcode);
    bb.putInt(this.count);
    bb.putInt(this.offset);

    for (int i = 0; i < this.name.length(); i++)
        bb.put((byte) this.name.charAt(i));

    return bb.array();
}
```

Listado 2: Conversión `MessageCS` → `byte[]`.

En ambos casos, se hacen uso de *búferes* para almacenar los datos en bruto, ya sea los que se reciben o los que se están convirtiendo. La clase `ByteBuffer` se utiliza como búfer, puesto que está diseñada para insertar y obtener tipos primitivos de datos directamente en él (véanse los métodos `getInt()` y `putInt()` como ejemplo).

Por defecto, `ByteBuffer` organiza los datos compuestos por varios bytes según el esquema *Big endian*: *en orden decreciente* de significación, siendo el primero que se lee el más significativo. Sin embargo, y como probablemente se habrá podido ver en el programa de C++, no se utilizan las funciones de conversión `htons()` o `ntohs()` (que incidentalmente "normalizan" los datos a *Big endian*), consiguiéndose de todas formas la conexión. La figura 2 en la página 2 esclarece el misterio: en la mayoría de las computadoras actuales, se utiliza el esquema *Little endian*, que *invierte el orden* con respecto a *Big endian*: el primer bit ahora es el menos significativo. Por lo tanto, para conservar esta organización *de facto* en la comunicación, es preciso invocar al método `order()`, que recibe como parámetro el orden deseado, y que para los propósitos del programa será `ByteOrder.LITTLE_ENDIAN`.

Como en C++ se emplea el tipo de dato `uint32_t`, que garantiza un número sin signo con tamaño exacto de 32 bits (o 4 bytes), en Java es necesario buscar otro equivalente. El más cercano es `int`, que de acuerdo con [3], en Java 8 puede emplearse para representar un número *sin signo*.

El problema se presenta cuando se intentan hacer compatibles los tipos de dato `char` de C++ y su homólogo en Java. Mientras que en C++ el tamaño no se definió, según [4], hasta el estándar C++14—lo suficientemente grande como para almacenar caracteres UTF-8, que son 256 posibles combinaciones o 1 byte de longitud—, en Java el tamaño se establece en *2 bytes* para almacenar caracteres Unicode de 16 bits. Un método para convertir datos entre ambos sistemas de representación debe emplearse para evitar malas interpretaciones de los datos.

Una de las posibles soluciones a este problema consiste en tomar en cuenta *una parte de la codificación*. En específico, Oracle Corp. [5] indica que los códigos `00-7F` se reservaron en ambos sistemas para representar *caracteres ASCII*, utilizados mayoritariamente en aplicaciones dirigidas a audiencias que utilicen el alfabeto inglés. Tomando en cuenta este detalle, el problema de representación se puede reducir a una conversión de tipos `char` ↔ `byte`, ya que Java, al forzar una conversión a un tipo con menos bits, hace que se conserven *los menos significativos*. Para otras audiencias, no obstante, una conversión formal es necesaria (véase [6] para una solución en Java 7 o superior).

Como último detalle, es de mención que el valor representado por `Integer.BYTES` fue añadido en Java 1.8, por lo que se recomienda tomar el valor directamente de [3] para versiones anteriores (el cual es 4).

§ 3. Conclusiones

La discusión anterior muestra lo importante que es conocer las peculiaridades de un lenguaje en particular, en especial cuando se torna necesario comunicar aplicaciones programadas en diferentes lenguajes y no se tiene acceso a una de ellas en código (pero sí en especificaciones). Esto sucede a menudo con programas antiguos que por cualquier razón deben continuar ofreciendo sus servicios.

Una práctica de este tipo acrecienta el aprendizaje e incentiva a la investigación por cuenta propia, al presentar un problema difícil de resolver, pero lo suficientemente interesante como para ser tomada en cuenta.

§ 4. Referencias

[1] E. S. Raymond. (19 de dic. de 2015). The Lost Art of C Structure Packing, dirección: http://www.catb.org/esr/structure-packing/#_padding (visitado 18-10-2016).

[2] F. Foo. (18 de mayo de 2011). Print a struct in C, dirección: http://stackoverflow.com/a/5349944 (visitado 18-10-2016).

[3] Oracle Corp. (2016). Primitive Data Types, dirección: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (visitado 18-10-2016).

[4] cppreference.com. (4 de jul. de 2016). Fundamental types, dirección: http://en.cppreference.com/w/cpp/language/types#Character_types (visitado 18-10-2016).

[5] Oracle Corp. (2016). Unicode Character Code Assignments, dirección: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html (visitado 18-10-2016).

[6] rzymek. (27 de nov. de 2013). Encode String to UTF-8, dirección: http://stackoverflow.com/a/20243062 (visitado 18-10-2016).