Documentación de Tarea 2 - Curso de interoperabilidad - Cabolabs

Datos del alumno

- **Nombre y apellido**: Pablo Muguerza Karolyi

Descripción de la solución propuesta

Lenguaje utilizado

La tarea se completó usando Groovy como lenguaje.

Clase MLLPServer

En primer lugar, se implementó la clase **`MLLPServer`**. Esta clase se encarga de crear un socket que escuchará en “0.0.0.0”, en el puerto indicado por el archivo de configuración **`config.groovy`**. El server se implementó usando como referencia:

- http://programmingitch.blogspot.com/2010/04/groovy-sockets-example.html
- La documentación de Groovy sobre ServerSocket

Se trata de una versión mejorada respecto de una clase similar que implementé para la entrega de la Tarea 1. En esta nueva implementación de **`MLLPServer`** se usa **`ServerSocket.accept()`** para esperar la conexión del cliente. Según la documentación de Groovy de la clase **`ServerSocket`**, el socket creado se pasa como parámetro al closure que se ejecuta en una nueva thread. En el código, el socket que recibe la nueva thread se llama **`clientSocket`**.
Luego se usa el método **`withStreams()`** que, (también según la documentación de Groovy), pasa los streams de output y de input a un closure. Dentro del closure, en primer lugar, se usa el método **`eachByte()`**, que permite obtener de a un byte los datos del stream de input. El stream recibido se procesa usando un buffer “ad hoc” que también fue utilizado para la tarea 1. Este buffer permite guardar de a un byte los datos de un stream, y usar el método **`pop_message()`** para obtener un mensaje en caso de que se haya recibido un mensaje completo (según los criterios del protocolo MLLP). En caso de que **`pop_message()`** devuelva un mensaje (que es un mensaje enviado por el cliente), se obtiene el id de control del mensaje y se genera un mensaje de ACK con ese id de control, para que el cliente pueda reconocer que el ACK corresponde a un determinado mensaje.

Además, se chequea que el mensaje sea de tipo **`ADT^A01`** (que es la única combinación de tipo de mensaje - evento aceptada según las pautas del ejercicio). Si el mensaje es de tipo **`ADT^A01`**, entonces en el segmento **`MSA`** del mensaje **`ACK`** se indica que se procesó correctamente el mensaje (mediante el código **`"AA"`**). En caso contrario, se devuelve un código de error (**`"AE"`**).
El server ejecuta un ciclo while(true), dentro del cual se realiza la llamada a **`ServerSocket.accept()`**, de forma que el server se siga ejecutando independientemente del hecho de que un cliente cierre la conexión (cada conexión individual establecida por un cliente se ejecuta en una thread "spawneada" por el server).

Clase MLLPClient

Esta clase permite realizar una conexión con el server MLLP (**`MLLPServer`**). El constructor recibe la IP y puerto en los que escucha el server MLLP, y además recibe el controlID del mensaje que se va a enviar y los datos del mensaje a enviar (en formato ER7). Es decir, que el cliente se inicializa para enviar un único mensaje. Una vez enviado el mensaje en formato ER7, se procesa el stream de input del socket, de a un byte por vez, usando la clase mllpbuffer. Si se recibió un mensaje, se usa una instancia de **`ACK`** (de la librería **HAPI**) para parsear el mensaje. Finalmente se chequea que el controlID recibido coincida con el del mensaje enviado.

Clase mllpbuffer

El buffer "ad-hoc" mencionado anteriormente está implementado en la clase **`mllpbuffer`** (que se encuentra en el archivo **`mllpbuffer.groovy`**). Esta clase permite agregar datos al buffer, de a un byte (que es como los recibe el server), y tratar de obtener un mensaje mediante el método **`pop_message()`**. El método **`pop_message`** devuelve **`null`** si no hay un mensaje completo; si por el contrario hay un mensaje que cumple con el protocolo MLLP (es decir, que se inicia con <SB> y finaliza con <EB><CR>), entonces el método devuelve un string correspondiente al mensaje, y elimina los bytes del buffer.

Clase mllp

Dado un String (que representa un mensaje HL7) permite encapsularlo de acuerdo al protocolo MLLP, para su posterior envío.

Clase HL7Message

Se utilizaron las clases **`HL7Message`**, **`ADTMessage`** y **`ACKMessage`** para encapsular las tareas de inicialización, parseo y encodeo de mensajes HL7 usando la librería HAPI. La clase **`HL7Message`** usa una instancia de la clase abstracta **`AbstractMessage`** de HAPI, de forma de poder representar de forma indistinta mensajes ADT y ACK. El método **`initMSH`** de HL7Message setea:

- El caracter separador
- Los caracteres de encoding
- La versión de HL7 (v2.5)
- El encoding de los mensajes (se usa ISO-8859-1 para poder enviar/recibir caracteres como “ñ” y vocales con acento). Este encoding es válido en este contexto porque se trata de una codificación de 1 byte, que no “rompe” el protocolo MLLP.
- Id y modo de procesamiento

clase ADTMessage

Se usó esta clase para encapsular las funcionalidades relacionadas con los mensajes ADT usando la librería HAPI. En resumen, las particularidades de esta clase, respecto a la clase genérica **`HL7Message`** son:

- Se setean **messageCode**, **triggerEvent** y **messageStructure** correspondientes a **`ADT^A01`**
- Se permite registrar los datos del paciente (PID segment).
- Se permite registrar los datos del segmento PV1 (patient visit 1). En los ejemplos (que se describen más adelante), los mensajes ADT se usan para representar ingresos, por lo tanto PV1 representa en todos los casos el ingreso de un paciente.

ACKMessage

Igual que en el caso de HL7Message y ADTMessage, se encapsula el manejo de HAPI para los mensajes ACK. Esta clase permite:

- Setear el controlID (de forma de poder establecer correspondencia entre el mensaje enviado por el cliente y la respuesta)
- Inicializar el segmento **`MSA`**, para indicar si se pudo procesar correctamente el mensaje (código “AA”) o si hubo un error (código “AE”).

config.groovy

Archivo de configuración, que indica host y puerto a utilizar en run_server.groovy y example_messages.groovy (que a su vez utilizan, MLLPServer y MLLPClient, respectivamente)

run_server.groovy

Este script se usa para crear una instancia de MLLPServer que quedará escuchando en el puerto indicado por el archivo de configuración (config.groovy)

example_messages.groovy

En este script se generan tres mensajes de tipo ADT, usando la clase ADTMessage mencionada más arriba (que encapsula el uso de la clase ADT_A01 de HAPI).
Los tres mensajes representan ingresos de tres pacientes diferentes, en diferentes horarios.

- En el segmento **`PID`** de los tres mensajes se usan dos identificadores, para representar dos formas de identificar a un paciente; uno correspondiente al DNI argentino (autoridad emisora = **RENAPER**), y otro un número de HCE única para el sistema público de la ciudad de buenos aires (autoridad emisora = **HCE_GCBA**).
- En el segmento PV1 se indica la ubicación del paciente, usando Facilty, Building, Floor y Bed. Lo que se quiso representar es una ubicación de la forma más genérica posible, de manera que puedan interoperar subsistemas independientes (en el ejemplo, el subsistema público de GCBA y el subsistema privado argentino).
- Para cada uno de estos mensajes se crea una instancia de **`MLLPClient`**, que se usa para enviar el mensaje ADT al server y esperar el ACK correspondiente.

Problemas encontrados

- No encontré la forma correcta de indicar el timezone en los mensajes (tanto para el timestamp del MSH como para la fecha/hora en el segmento PV1 de los mensajes ADT)
- No encontré la forma de generar automáticamente el timestamp de los mensajes
- No tengo claro cómo mostrar por consola los caracteres especiales del encoding ISO-8859-1. Aunque entiendo que los caracteres se reciben correctamente, no sé cuál sería la forma en Java/groovy de imprimir con println los caracteres como “ñ” o “á”
- No sé cuál es la forma correcta de castear entre el formato “YYYYMMDDHHMMSS” usado en los mensajes, y un formato de fecha/hora de java/groovy para mostrar de forma más prolija los datos por pantalla.
- No me queda claro, respecto al campo Assigned Patient Location, cuál es el uso que se le daría a “point of care” (teniendo en cuenta el criterio de que “*This data type contains several location identifiers that should be thought of in the following order from the most general to the most specific: facility, building, floor, point of care, room, bed*”).
- No sé si la forma correcta de mantener corriendo el server y el cliente luego de que un cliente cierre la conexión es la que implementé (que consiste, tanto en el cliente como en el server, en usar un bloque try… catch en el que se procesa la excepción **`SocketException`**). Hasta donde pude probar esto funciona, es decir, el cliente cierra la conexión y el server sigue ejecutándose, pero quizás hay una forma más prolija de cumplir con lo que pide el ejercicio.
- En el instructivo para instalar HAPI (de las guías prácticas) creo que faltaban incluir un par de archivos necesarios para que HAPI funcionara, esto lo pude solucionar rápidamente de todas formas. En resumen, en el documento se mencionaba que es necesario copiar las siguientes librerías de hapi (en mi caso, descargué hapi-dist-2.2-all)

- **hapi-base-2.2.jar**
- **hapi-structures-v25-2.2.jar**
- **slf4j-api-1.6.6.jar**

Pero además de estas librerías, era necesario también copiar:

- **Log4j-1.2.17.jar**
- **Slf4j-log4j12-1.6.6.jar**

Aclaro también que no llegué a incluir ejecutables Java, sino sólo los fuentes. Aunque a través del foro recibí un instructivo, llegué un poco justo con el tiempo, espero poder usar el instructivo para la próxima entrega.

Instrucciones para compilar / ejecutar

Nota: Las instrucciones corresponden a un entorno linux, aplicar las variantes correspondientes en windows:

1. Instalar Groovy 2.x

2. Descomprimir el archivo con la tarea

```
unzip pablo_muguerza_karolyi_tarea_2.zip
```

3. Agregar las librerías necesarias para que se pueda usar HAPI

NOTA 1: (las instrucciones valen para Linux, no tengo el detalle de cuál es el directorio en Windows pero el criterio sería el mismo). Esto implica copiar las librerías al directorio donde están las librerías de Groovy
Nota 2: Los archivos .jar que se incluyen como librerías se obtuvieron de **hapi-dist-2.2-all**

```
cd tarea_2/lib
cp * /usr/share/groovy2/lib
```

4. Setear el path de groovy para que se encuentren correctamente las librerías

NOTA 3: (igual que lo que se mencionó en el punto 3, las instrucciones valen para Linux)

```
export GROOVY_HOME=/usr/share/groovy2
```

5. Ir al directorio donde se encuentran los fuentes .groovy

```
cd ..
cd ..
cd tarea_2/src
```

6. En una terminal, iniciar el server

```
groovy run_server.groovy
```

7. En otra terminal, ejecutar el script que envia los mensajes ADT

```
groovy example_messages.groovy
```