Capítulo 4

Implementación

En este capítulo se discutirán aspectos relacionados con los problemas e inconvenientes asociados a la creación del software.

4.1. kjProtocol API

Este API se diseñó tomando en cuenta 4 clasificaciones principales para las clases, es decir, 4 paquetes:

- mx.udlap.kjProtocol.exceptions .- Es el paquete que contiene a las excepciones.
- mx.udlap.kjProtocol.kbinds .- Este paquete contiene clases relacionadas con la conexión con kannel.
- mx.udlap.kjProtocol.packets .- Este contiene una abstracción de todos los paquetes que usa kannel, y tipos de datos asociados a los mismos.
- mx.udlap.kjProtocol.tools .- Contiene herramientas relacionadas con el manejo de cadenas de bytes.

Algunas de las clases más relevantes por su dificultad o importancia son:

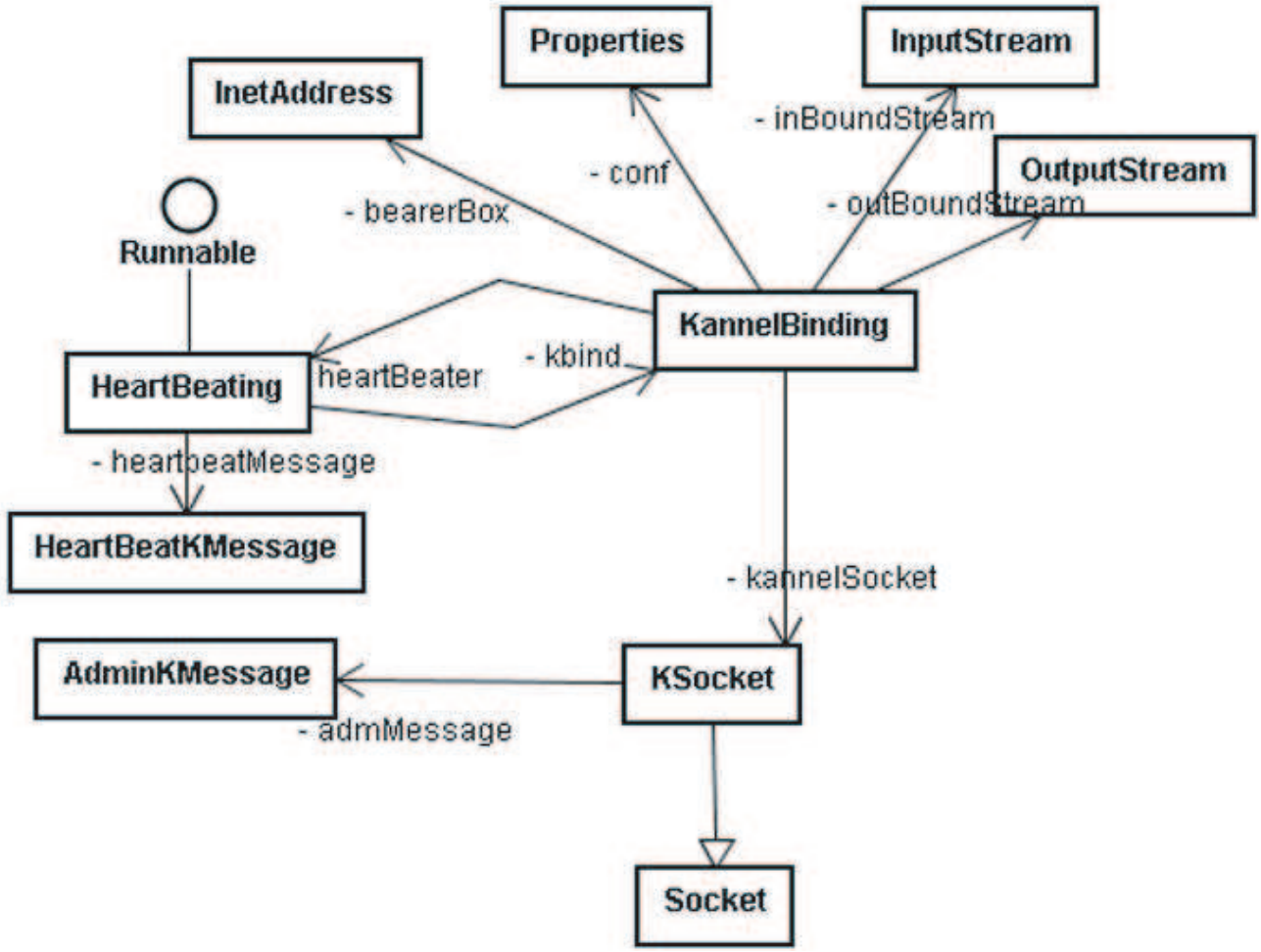

Figura 4.1: Diagrama de clases de KannelBinding.

`mx.udlap.kjProtocol.kbinds.KannelBinding`

Mantiene una conexión con el bearerbox y permite leer y escribir paquetes completos (Figura 4.1).

El método más problemático de esta clase fue `readNext()`, ya que hasta últimas etapas de la construcción del software se descubrió que el bearerbox escribe cuatro caracteres de fin de cadena tipo C al final de cada paquete, por lo que al buffer de entrada se agregan 4 bytes con ceros. Esto generó un error `java.lang.OutOfMemoryError` y

dada la cantidad de threads usados en el programa tomó cerca de 15 horas encontrar donde estaba el error y solucionarlo. Este error se generaba por que los primeros 4 bytes del paquete se espera que formen la longitud del paquete completo:

```
        ...
// kLen sera la longitud del paquete completo.
KInteger kLen = null;

// bInt es el arreglo que alojara los
// 4 bytes que formen dicho numero.
byte[] bInt = new byte[4];

int read = 0;
        ...
// Esta linea lee bInt.length bytes en bInt
// y regresa la cantidad total de bytes leidos
read = this.inBoundStream.read(bInt);
        ...
// Se parsean los 4 bytes en una clase
// que contiene dicha representacion numerica
kLen = new KInteger(bInt);
len = kLen.getIntValue();
        ...
```

cuando se sabe la longitud del paquete completo, se reserva memoria para almacenarlo:

```
        ...
bInt = new byte[len + 4];
        ...
```

pero al haber un corrimiento de bytes en el buffer; cuando se lee la longitud del siguiente paquete, se tiene un número demasiado grande como para ser almacenado en memoria.

La solución de este problema fue muy simple, sólo hubo que remover los 4 bytes extras del buffer e ignorarlos, en 4 líneas de código la solución fue efectiva:

```
            ...
        this.inBoundStream.read();
        this.inBoundStream.read();
        this.inBoundStream.read();
        this.inBoundStream.read();
            ...
```

Despés de dicho arreglo, este módulo funcionó perfectamente.

`mx.udlap.kjProtocol.packets.BasicPacket`

Esta clase es extendida por todos los paquetes para incluir longitud y tipo.

`mx.udlap.kjProtocol.packets.BasicKannelProtocolMessage`

Esta es una interfaz que implementan todos los paquetes para poder ser usados de manera uniforme. Contiene al método `byte[] getMessage()` que devuelve un arreglo de bytes listo para escribirse a un flujo de salida conectado con kannel y al método `void setMessage(byte[] data)` encargado de crear una instancia del paquete solicitado a partir de un arreglo de bytes, éste es usado para parsear un paquete que recién ha sido recibido en el flujo de entrada.

`mx.udlap.kjProtocol.packets.*`

Todos los paquetes deben extender a la clase BasicPacket para tener los super atributos `length` y `type` que permitan identificarlos e implementar la interfaz `BasicKannelProtocolMessage` que permite crear y acceder a la representación en bytes de un paquete.

Un detalle muy importante de los paquetes es que cada miembro de un paquete es especificado por un tipo de dato (Figura 4.2, Apéndice C.1).A continuación exploraremos como es que los tipos de datos son codificados.

`mx.udlap.kjProtocol.packets.KInteger`

Este es el tipo de dato más importante ya que es usado por los otros tipos de datos: `KString`, `KTime` y `KUUID`. Como puede verse en la figura 4.2, el contenido de todos los tipos de datos es un arreglo de bytes, en el caso de los tipos numéricos (`KTime` y

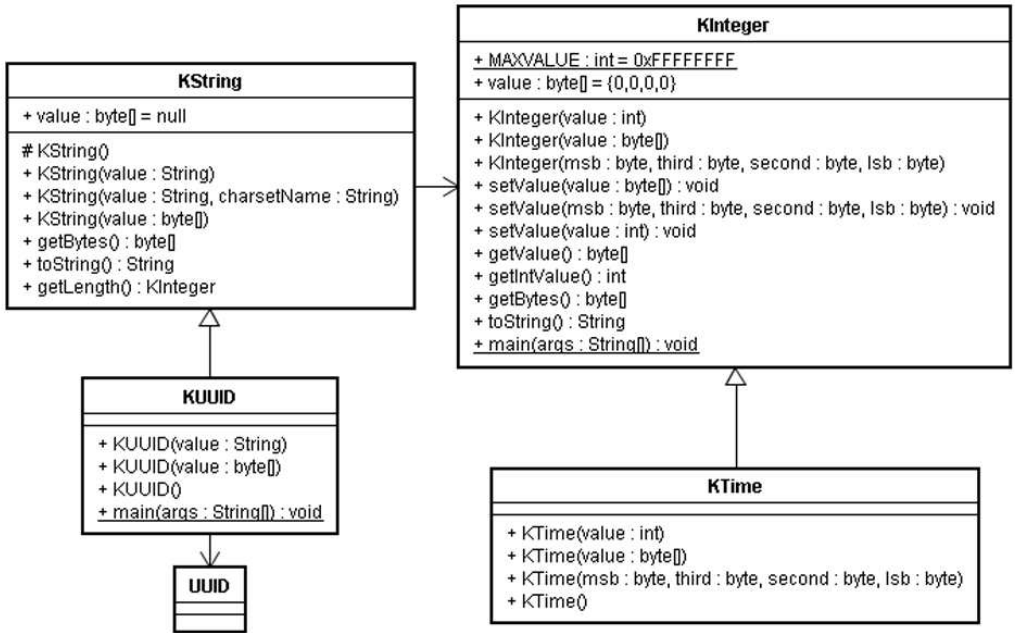

Figura 4.2: Diagrama de clases de los tipos de datos que componen los miembros de los paquetes.

`KInteger`) es sólo de 4 bytes. Dicho arreglo contiene un entero de 4 bytes construido desde un entero primitivo de Java por el método `public void setValue(int value)` usando corrimientos de bits de múltiplos de 8:

```
        ...
public void setValue(int value) {
        byte msb = 0;
        byte lsb = 0;
        byte third = 0;
        byte second = 0;

        msb = (byte) (value >> 24);
        third = (byte) ((value << 8) >> 24);
        second = (byte) ((value << 16) >> 24);
        lsb = (byte) ((value << 24) >> 24);

        setValue(msb, third, second, lsb);
}
```

```
public void setValue(byte msb, byte third, byte second,
   byte lsb) {

        this.value[0] = msb;
        this.value[1] = third;
        this.value[2] = second;
        this.value[3] = lsb;
}
        ...
```

Como podemos ver en la figura 4.2, el tipo KTime es una extensión de KInteger cuya semántica es el almacenamiento de tiempo en formato de UNIX (Número de segundos desde el EPOCH es decir 00:00 horas del 1 de enero de 1970) e incorpora un constructor vacío cuyo objetivo es inicializar KTime con el tiempo actual.
El caso de KString es un poco más elaborado. Éste está diseñado para contener una cadena de bytes de longitud variable. Para representar la longitud de dicha cadena, se usa un tipo KInteger, ya que la representación de los miembros que usan cadenas incluye 4 bytes de encabezado que indican la longitud de la cadena o -1 (0xFFFFFFFF) si la cadena es vacía.
KUUID es una extensión de KString que incluye un constructor vacío que inicializa el tipo de dato con un UUID basado en tiempo.

4.2. kjGateway

Este paquete contiene las interfaces necesarias para construir un gateway entre kannel y JMS, dentro de este paquete se incluye un ejemplo de implementación de dicho gateway y un ejemplo de implementación de un servicio (Sección 4.3).

En la figura 3.3 pudimos observar de manera general como es que se relacionan entre sí las diferentes clases de este módulo, ahora vamos a detallar las partes más compli-

cadas del mismo.

KannelJMSGateway

Esta clase es la que crea los objetos y threads que de hecho hacen todo el trabajo. Su ejecución es lineal y es la encargada de preparar las variables globales, intercambiar instancias entre objetos y cargar las propiedades del sistema desde un archivo, el cual contiene la configuración del sistema en modo de propiedades (Apéndice B). A continuación se muestra un listado de tareas que realiza esta clase para inicializar el sistema:

1. Leer propiedades de un archivo.
2. Cargar implementación de la interfaz `JMSTransport`
3. Inicializar clase `JMSTransport`.
4. Inicializar clase `KannelBinding`
5. Inicializar clase `KjReadingThread` y ejecutar.
6. Inicializar clase `KjWritingThread` y ejecutar.
7. Inicializar clase `AckCycleThread` y ejecutar.

Después de estos pasos, la ejecución de esta clase termina y pasa el control a los threads inicializados.

kjReadingThread

En el capítulo 3 hemos descrito de manera general el uso de esta clase. Ahora vamos a mostrar más de cerca su comportamiento. Al iniciar esta clase con el método `run` se bloquea en un ciclo a la espera de la lectura de un paquete:

```
...
public void run() {
        ...
 while(!halted){
   ...
   // Este llamado se bloquea hasta leer
   // un paquete de kannel
   recPacket = kbind.readNext();
    ...

    if(type == BasicPacket.ACK_PKT){ // Si es un acknowledge

     if(this.ackAdminThread != null){
      this.ackAdminThread.confirmAck((AckMessage)recPacket);
     }

    }else if(type == BasicPacket.SMS_PKT) {
     // Si es un paquete tipo SMS
     // Le indicamos a nuestra instancia de
     // la interface JMSTransport que un mensaje
     // SMS ha sido recibido.
     this.jmsTransport.gotMOMessage((SMSPacketMessage)recPacket);
    }
    ...
 }
 ...
}
...
```

Como puede verse en la figura 4.3, esta pieza de código pasa el mensaje a distintos objetos para su procesamiento. El flujo pasa a `JMSTransport` (Ver ejemplo de implementación de esta interface en la sección 4.3) en el caso de mensajes SMS y a `AckCycleThread` en el caso de los acknowledges, mediante llamados a los métodos `gotMOMessage` y `configmAck` respectivamente.

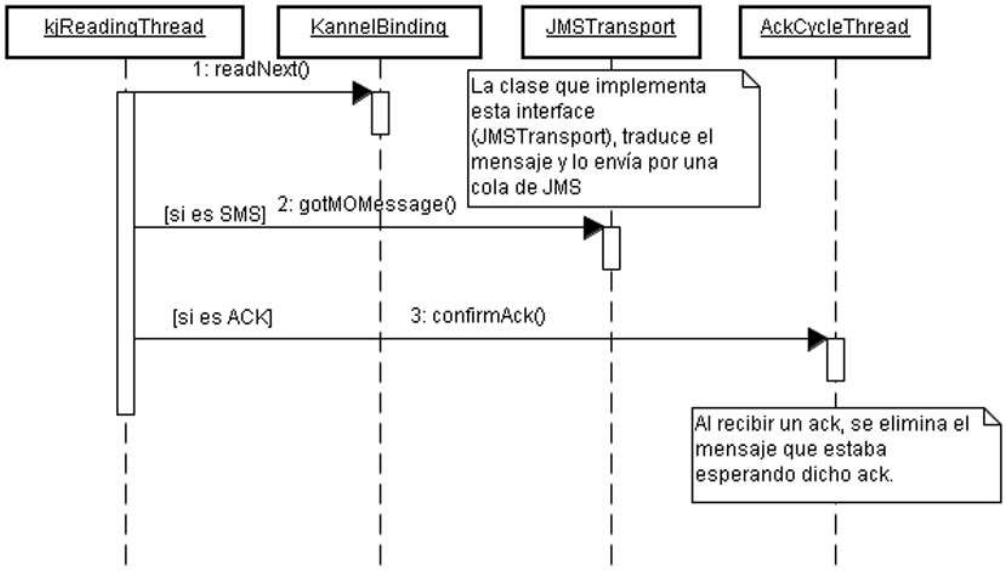

Figura 4.3: Diagrama de secuencias de la clase KjReadingThread

4.3. Ejemplo simple

Las clases de ejemplo tienen como prefijo la palabra `Simple`:

`mx.udlap.kjGateway.SimpleJMSTransport` implementa `JMSTransport` y `javax.jms.MessageLi`
y es cargada de manera dinámica por el gateway, es decir, que después de crear una clase que implemente dichas interfaces, la clase `KannelJMSGateway` puede usarla simplemente incluyéndola en el classpath y añadiéndola a las propiedades bajo la clave: `kjGateway.JMSTransportClass`

Después de instanciar esta clase, lo primero que hay que hacer antes de poder ponerla en acción, es iniciarla. La inicialización consiste en preparar el ambiente para el funcionamiento con JMS, para ello existe el método `start`, que en esta implementación, realiza los siguientes pasos (Código fuente C.2):

1. Cargar propiedades.

2. Cargar traductor de objetos de kjProtocol a JMS ObjectMessage.

3. Inicializar contexto para búsquedas en JNDI.

4. Obtener acceso a JMS mediante la obtención de los objetos `QueueSender` y `QueueReceiver`.

5. Iniciar thread para leer en la cola de entrada de JMS.

Después de dicha inicialización, el Objeto queda a la espera de mensajes entrantes, usando los métodos: `gotMOMessage` y `onMessage`, se notifica la llegada de mensajes procedentes de Kannel y de JMS respectivamente.

`gotMOMessage`

Este método es activado cuando un mensaje de entrada es recibido por parte de kannel, esto quiere decir, que el mensaje ha sido originado en un dispositivo móvil (Mobile Originated). Esta implementación del método realiza las siguientes operaciones:

1. Traducir el paquete `SMSPacketMessage` a un objeto tipo `SimpleMessage`.

2. Crear un `ObjectMessage` que es el tipo de objeto que puede transportar objetos entre colas de JMS.

3. Asignar el objeto tipo `SimpleMessage` as `ObjectMessage` para que éste pueda ser enviado.

4. Enviar el mensaje a la cola.

5. Crear y enviar un mensaje de acknowledge para kannel.

`onMessage`

Este método es activado cada vez que un mensjae procedente de la cola JMS de recepción es recibido, es decir, mensajes originados por una aplicación de servicio, con destino en un dispositivo móvil. Esta implementación realiza las siguientes operaciones:

1. Traducir el objeto mensaje a un paquete objeto tipo `SMSPacketMessage`.
2. Enviar el paquete a kannel.

Como pudimos observar, esta clase es donde se lleva a cabo el intercambio de mensajes entre los dos sistemas. Es importante recalcar el hecho de que esta clase está implementando dos interfaces, y que puede ser cargada dinámicamente por la clase `KannelJMSGateway`. Éstas son características que hacen factible el uso de este sistema para funcionar con conexiones reales y con clientes reales, ya que su diseño puede ser ajustado para modelar las necesidades de un proveedor de contenido SMS en el mercado mundial.

`mx.udlap.kjGateway.SimpleMessage` es una abstracción de ejemplo para un mensaje que transmitirá el gateway al servicio(Código fuente C.3).

`mx.udlap.kjGateway.SimpleJMSTranslator` es una clase que implementa la interfaz `mx.udlap.kjGateway.JMSTranslator` encargada de convertir los mensajes tipo SMS al tipo de objeto definido para abstraerlos y viceversa.

`mx.udlap.kjGateway.SimpleQueueReceiverThread` esta clase es un thread que se bloquea a la espera de un mensaje en la cola de entrada del gateway y llama al método onMessage del message listener (`mx.udlap.kjGateway.SimpleJMSTransport`).

`mx.udlap.kjGateway.SimpleService` es un ejemplo mínimo de una aplicación de servicio. Todo lo que hace, es usar el texto recibido como llave en un objeto properties y enviar de regreso el valor de dicha propiedad. Las propiedades que tienen este contenido están en un archivo que se ve de la siguiente manera:

```
PRUEBA = Prueba exitosa !
HOLA = Hola mundo !
ADIOS = Good bye !
```

Lo que significa que si un dispositivo envía un mensaje de texto que diga: hola, este va a ser convertido a mayúsculas, usado como llave para buscar dentro del objeto de propiedades donde encontrará el texto: Hola mundo ! y enviará dicho mensaje al dispositivo que originó el primer mensaje.
Esta clase implementa la interfaz `MessageListener`(Código fuente C.4) para recibir los mensajes mediante un llamado a `onMessage`.

Conclusión

En este capítulo hemos repasado los detalles de implementación del proyecto. Finalmente, podemos decir que el diseño nos ha permitido usar las clases creadas como un conjunto de herramientas para construir diferentes gateways entre Kannel y otras tecnologías como JMS, con mucha flexibilidad.