EUSKO JAURLARITZA
GOBIERNO VASCO

Programa Saiotek 2006

SMARTLAB

Entorno de Trabajo Inteligente Colaborativo y Programable

MoteCommunications

Tecnológico Fundación Deusto
Teknologikoa Deustu Fundazioa

RESUMEN

HISTORIAL DE CAMBIOS

Versión	Descripción	Autor	Fecha	Comentarios
V0.1	Documentación MoteCommunications	Pablo Orduña	14/01/2008	

TABLA DE CONTENIDOS

Resumen 3

Historial de cambios 4

Tabla de contenidos 5

1 Introducción 7

2 Motecommunicationsmartlab 9

2.1 Descubrimiento 9

2.2 Comunicación con el servidor 9

2.3 Comunicación con la mota base 11

2.4 Configuración del proyecto 11

2.5 Compilando el proyecto 13

2.6 Añadiendo un nuevo dispositivo 13

2.7 Pruebas 14

3 MoteCommunicationSmartLabLib 15

3.1 Introducción 15

3.2 DTOs 15

3.3 Compilando el proyecto 16

3.4 Añadiendo nuevos dispositivos al proyecto 16

4 Motecommunicationbundle 17

4.1 Introducción 17

4.2 Registros 17

4.3 Estructuras de datos 18

4.4 RMI 20

4.5 Compilando el proyecto 21

4.6 Añadiendo nuevos dispositivos al proyecto 21

4.7 Pruebas 21

© Smartlab

1 INTRODUCCIÓN

Este documento describe la adaptación del proyecto Flexeo (http://www.tecnologico.deusto.es/projects/flexeo) en el proyecto SmartLab, mediante la combinación de la plataforma empotrada Gumstix con la una red de sensores inalámbricos (WSN) basado en Mica2.

Para la realización del proyecto Flexeo, se tomaron diferentes objetos heterogéneos (una silla, una jarra de agua…), a las que se añadieron sensores para medir diferentes situaciones (la jarra de agua está vacía, está llena, hay alguien sentado, está bien apoyado…), y se conectaron estos sensores a unas motas. Para obtener esta información, hay una mota base que hace de proxy entre nosotros y las motas. Se puede enviar mensajes a las motas a través de esta mota base, y podemos configurar a qué mensajes queremos subscribirnos para que cuando llegue ese tipo de mensaje proveniente de las motas se nos notifique y procesemos el mensaje. Por tanto, para la realización del proyecto Flexeo se desarrolló toda la infraestructura hardware y el software de las motas, que ha podido ser reutilizado directamente en el proyecto SmartLab.

Además, en el proyecto Flexeo se desarrolló un software en Java que interactuaba con las motas, procesando los mensajes enviados por las motas y generando nuevos mensajes para enviar. Este software estaba desarrollado como un bundle de OSGi, y exportaba cada dispositivo (jarra de agua, silla) como servicio de OSGi, de manera que cualquier otro servicio podía acceder directamente a cualquier dispositivo y enviarle o recibir mensajes. Todo este software se desplegaba en una implementación de OSGi llamada Concierge, en un dispositivo gumstix.

El objetivo del módulo MoteCommunication es mantener la idea de que en una instancia de OSGi hay una serie de servicios que representan los diferentes dispositivos y permiten interactuar con estos. Sin embargo, dado que esta vez la instancia de OSGi no está en la misma máquina que el software que interactúa directamente con la mota base (ahora la instancia de OSGi corre en un servidor mientras que la comunicación con la mota base sigue estando delegada en el gumstix), los requisitos del módulo cambian bastante, y la reutilización de código es menos directa. Ahora tiene que haber un software en el gumstix que interactúe directamente con la mota base y reenvíe a través de la red esa información al servidor. Este software ya puede ser independiente de OSGi, y tiene que integrar un cliente del protocolo de descubrimiento de Smartlab. Además, en el lado del servidor, cada

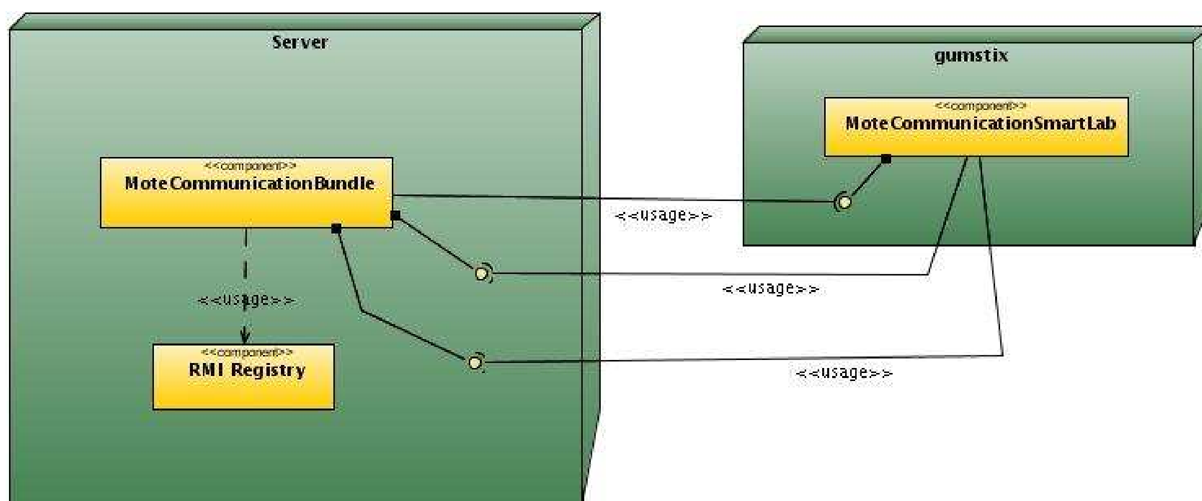
dispositivo es registrado con más interfaces que en Flexeo: el interfaz dispositivo en sí, el interfaz del servicio semántico, y el interfaz de la gestión de eventos de OSGi.

A pesar de esto, diferentes componentes del software de Flexeo han podido ser adaptados (aunque hayan tenido que ser modificados):

- Módulo de configuración
- Modelo de datos de los mensajes enviados
- Gestión de mensajes de bajo nivel

A lo largo del documento, hablaremos de:

- *MoteCommunicationSmartLab*: ejecutable desarrollado en Java 1.4 que se despliega en el gumstix, interactúa con la mota base, y envía los mensajes al servidor (*MoteCommunicationBundle*).
- *MoteCommunicationBundle*: bundle de OSGi desarrollado en Java 1.6 que despliega cada dispositivo como un servicio de OSGi, un servicio semántico y un gestor de eventos.
- *MoteCommunicationSmartLabLib*: librería desarrollada en Java 1.4 con código común a *MoteCommunicationBundle* y *MoteCommunicationSmartLabLib*. Todas las clases y excepciones que se propaguen entre el cliente y el servidor estarán en este proyecto.



2 MOTECOMMUNICATIONSMARTLAB

2.1 Descubrimiento

El *MoteCommunicationSmartLab* utiliza como librería el proyecto Discovery/discoveryclient_lib, de manera que nada más comenzar el proyecto configurará dónde está el .jar, qué versión tiene, etc., y esperará a que la librería avise de que el bundle está desplegado en el servidor. Para esto, la librería buscará al servidor mediante el protocolo de descubrimiento, enviará el .jar del bundle si es necesario, se quedará esperando a que termine de lanzarse el bundle en el servidor, y una vez haya terminado, nos avisará, de manera transparente para el proyecto MoteCommunication (encapsulado en la librería discoveryclient_lib).

2.2 Comunicación con el servidor

La comunicación con el servidor se realiza mediante RMI. Utilizamos RMI para facilitar que se añadan nuevos dispositivos al proyecto, evitando que haya que programar la serialización de los objetos una a una, y pudiendo resolver la comunicación de una manera más sencilla.

Entonces, una vez el bundle está desplegado en el servidor, el proyecto se conectará a un rmiregistry que se habrá desplegado en el servidor. El *MoteCommunicationSmartLab* buscará en este registro el servicio (el nombre por defecto es SmartLabServer) que proporciona el *MoteCommunicationBundle*. Este servicio implementa la interfaz *es.deusto.tecnologico.smartlab.motes.comm.rmi.IMoteCommunicationsRegistry*, por lo que el gumstix puede añadirse a sí mismo (ya que implementa el interfaz *es.deusto.tecnologico.smartlab.motes.comm.rmi.IMoteCommunicationRMI*) en un registro que estará implementado en el bundle. A partir de ese momento, el bundle podrá enviar mensajes, añadir listeners en el gumstix (almacenados en *es.deusto.tecnologico.smartlab.motes.comm.rmi.MoteCommunicationRMI*) para ser avisado cada vez que algo cambie, obtener el último valor (o últimos valores, según desee) que tiene un sensor concreto, u obtener información del concentrador (identificador del concentrador y dispositivos que gestiona), tal y como se puede ver en el interfaz *IMoteCommunicationRMI*:

```
public interface IMoteCommunicationRMI extends Remote{
      public void sendMessage(ExteriorMessageDTO message, int to)
            throws RemoteException, MotesCommunicationException, RmiException;

      public void addAllMessageListener(IRemoteMessageListener listener)
            throws RemoteException, RmiException;

      public void removeAllMessageListener(IRemoteMessageListener listener)
            throws RemoteException, RmiException;

      public void addChangeMessageListener(IRemoteMessageListener listener)
            throws RemoteException, RmiException;

      public void removeChangeMessageListener(IRemoteMessageListener listener)
            throws RemoteException, RmiException;

      public SmartTypeBase get(DeviceID deviceID)
            throws RemoteException, DeviceMapException;

      public SmartTypeBase getLastChanged(DeviceID deviceID)
            throws RemoteException, DeviceMapException;

      public SmartTypeBase[] getHistory(DeviceID deviceID)
            throws RemoteException, DeviceMapException;

      public String getConcentratorID()
            throws RemoteException;

      public Device [] getHandledDevices()
            throws RemoteException;
}
```

Por defecto, no envía ninguna información. En el momento que el servidor registre un *es.deusto.tecnologico.smartlab.motes.comm.rmi.IRemoteMessageListener*, este recibirá todos los mensajes que la mota base recibe (si lo ha registrado mediante addAllMessageListener), o bien solamente aquellos mensajes que registran un cambio (addChangeMessageListener). Así, si no hay nadie sentado sobre la silla, la mota seguirá enviando al gumstix la información de que no hay nadie sentado sobre la silla, y el gumstix se lo comunicará a todos los listeners que se hayan registrado en “addAllMessageListener”. Sin embargo, si alguien se registra como “addChangeMessageListener”, dado que no está cambiando el estado de la silla, no recibirá ningún mensaje, y cuando alguien se siente, recibirá un único mensaje notificando el cambio. La definición de qué se considera cambio y qué no depende del dispositivo concreto, y se define en el método “isNewSample” de cada Message. Así, por lo general todos comparan simplemente los valores considera que ha habido un cambio cuando los valores son diferentes. Sin embargo, en el caso de un sensor de temperatura, se considera como cambio únicamente cuando hay una diferencia

considerable (en el caso de ContainerMessage, por defecto está configurado a 5.5 grados como ejemplo).

La jerarquía de clases que se envía al servidor está documentada en la sección *MoteCommunicationSmartLabLib*.

2.3 Comunicación con la mota base

Para interactuar con la mota base, utilizamos la librería *tinyos.jar*. Esta librería gestiona todas las operaciones de bajo nivel. Además, proporciona herramientas como mig, que dado un .h (como el que se puede encontrar en la carpeta "util"), genera una serie de ficheros .java con clases que encapsulan los mensajes. El problema es que estas clases, que heredan de *net.tinyos.message.Message*, son de bastante bajo nivel (trabaja con arrays de bytes con la posición de cada variable en el mensaje de bajo nivel) y son sobrescritas completamente cada vez que tenemos que llevar a cabo un cambio en el mensaje (cambiando el .h y necesitando relanzar el comando mig), de manera que añadir funcionalidades en estas clases podría acarrear problemas.

Para evitar tener que lidiar directamente con estos mensajes de tinyos, tenemos creada otra jerarquía de clases que heredan de *es.deusto.tecnologico.smartlab.motes.comm.messages.Message*, que encapsulan los mensajes recibidos en el formato de tinyos. De esta manera, esta nueva clase es a su vez una factoría que dado un Message de tinyos genera su equivalente en nuestra jerarquía. Así, si llega un AlarmMessageStub (cuyo código ha sido automáticamente generado por mig), nuestra clase Message devolverá una clase hija de ella misma (en este caso, un AlarmMessage), que tendrá la misma información, pero más funcionalidades. Las clases hijas tienen además la obligación de implementar operaciones que exportar la información que tienen a otros formatos (como un diccionario), o a considerar si otro mensaje es o no nuevo respecto al mensaje actual, generar DTOs (que se explican en la sección *MoteCommunicationSmartLabLib*), etc. Estas clases son por tanto dependientes de operaciones de tinyos.jar, pero precisamente encapsulan estas dependencias.

2.4 Configuración del proyecto

El proyecto *MoteCommunicationSmartLab* necesita un fichero de configuración XML que define qué dispositivos gestiona el gumstix, y establece información acerca de cada

dispositivo, como su posición espacial (x,y), su nombre único (“Silla de tal persona”), la implementación de estas clases y los interfaces que definen sus operaciones. Un ejemplo de este fichero sería el siguiente:

```
<?xml version="1.0" encoding="UTF-8" ?>

<devices>
  <device name="Mr. Smith's chair" nodeID="1"
    interface="es.deusto.tecnologico.smartlab.motes.comm.server.data.SmartChairS
    ervice"
    class="es.deusto.tecnologico.smartlab.motes.comm.server.data.impl.SmartChair
    ServiceImpl" x="1" y="5">
  <capture name="BackContact" maxcaptures="1" />
  <capture name="BaseContact" maxcaptures="1" />
    </device>
  <device name="Mr. Smith's glass" nodeID="2"
    interface="es.deusto.tecnologico.smartlab.motes.comm.server.data.SmartContai
    nerService"
    class="es.deusto.tecnologico.smartlab.motes.comm.server.data.impl.SmartCont
    ainerServiceImpl" x="10" y="50">
  <capture name="Temperature" />
    <!--
   If no "maxcaptures" is provided, the default value is 1
   -->
  <capture name="Level" maxcaptures="1" />
    </device>
  <device name="Mr. Smith's clock" nodeID="10"
    interface="es.deusto.tecnologico.smartlab.motes.comm.server.data.SmartDispla
    yService"
    class="es.deusto.tecnologico.smartlab.motes.comm.server.data.impl.SmartDispl
    ayServiceImpl" x="5" y="15">
  <capture name="panic" />
    </device>
    </devices>
```

El atributo nodeID define cuál es el identificador de la mota que gestiona el dispositivo, y los elementos “capture” que hay en su interior definen los diferentes sensores que puede tener

el dispositivo. Como vemos, cada sensor tiene opcionalmente un atributo “maxcaptures”, que especifica la longitud del historial del sensor en cuestión. Tener un historial de “5” implica que en un momento dado se puede acceder a las últimas 5 mediciones que el gumstix tiene registradas.

2.5 Compilando el proyecto

Para compilar el proyecto, se ha definido un fichero de ant (llamado “build.xml”), que lleva a cabo todos los pasos necesarios para construir el .jar del proyecto. Se han definido tareas que crean los .jar necesarios, crean los stubs de RMI, despliegan el software en el gumstix a través de ssh, y que ejecutan el proyecto en el gumstix también mediante ssh. Hay un problema con esta tarea, ya que al lanzar el proyecto luego no se muere bien el programa, por lo que la próxima vez hay que primeramente matar el proyecto del todo.

Es importante tener en cuenta que en el gumstix, la máquina virtual de java (jamvm) no es capaz de leer los .class si estos están en .jars dentro de otros .jar. Es por ello que en la tarea “deploy” no sólo envía el .jar completo, sino que envía los .jar internos fuera del .jar para que la máquina virtual pueda ejecutarlos.

2.6 Añadiendo un nuevo dispositivo

Para añadir un nuevo dispositivo (pongamos como ejemplo un timbre), los pasos son los siguientes:

1. Añadir al fichero util/comm_definition.h la definición a bajo nivel del mensaje que se recibirá del dispositivo.

2. Generar el stub correspondiente. Para ello, añadir el nombre del mensaje al script “util/comm_stubs_generator.sh”. El stub generado debería estar en el paquete “*es.deusto.tecnologico.smartlab.motes.comm.stubs*”.

3. Crear una clase (por ejemplo, TimbreMessage), que herede de “*es.deusto.tecnologico.smartlab.motes.comm.messages.Message*” y encapsule los argumentos del stub generado.

4. En la clase Message:

a. Añadir el stub al método getTemplates

b. Permitir que se instancia TimbreMessage desde el método getMessage(net.tinyos.message.Message).

Además, será necesario generar un DTO correspondiente (lo cual se trata en la sección de la documentación de *MoteCommunicationSmartLabLib*), y una vez generado, implementar el método “createDTO()” de la clase “TimbreMessage” generada, así como añadir el DTO al método fromDTO(MessageDTO dto) de la clase “*es.deusto.tecnologico.smartlab.motes.comm.messages.Message*”.

2.7 Pruebas

El proyecto cuenta con un directorio “test”, que cuenta con un pequeño número de tests desarrollados utilizando JUnit 3.8 (dada la dependencia de Java 1.4 del proyecto). Desafortunadamente estas pruebas tienen una cobertura bastante pequeña del proyecto.

Además, entre los parámetros que el programa recibe como argumentos, está el modo en el que se desea ejecutar. Estos dos modos son:

1. Normal: En este modo, los paquetes se reciben de la mota base directamente.

2. Testing: En este modo, se crean una serie de paquetes de bajo nivel y se introducen en el receptor de información como si viniese de la mota. Estos paquetes pasan a través de todas las capas de la librería de tinyos.jar, y de esta manera se notifica el *MoteCommunicationSmartLab* exactamente como se llama cuando se usa hardware real.

3 MOTECOMMUNICATIONSMARTLABLIB

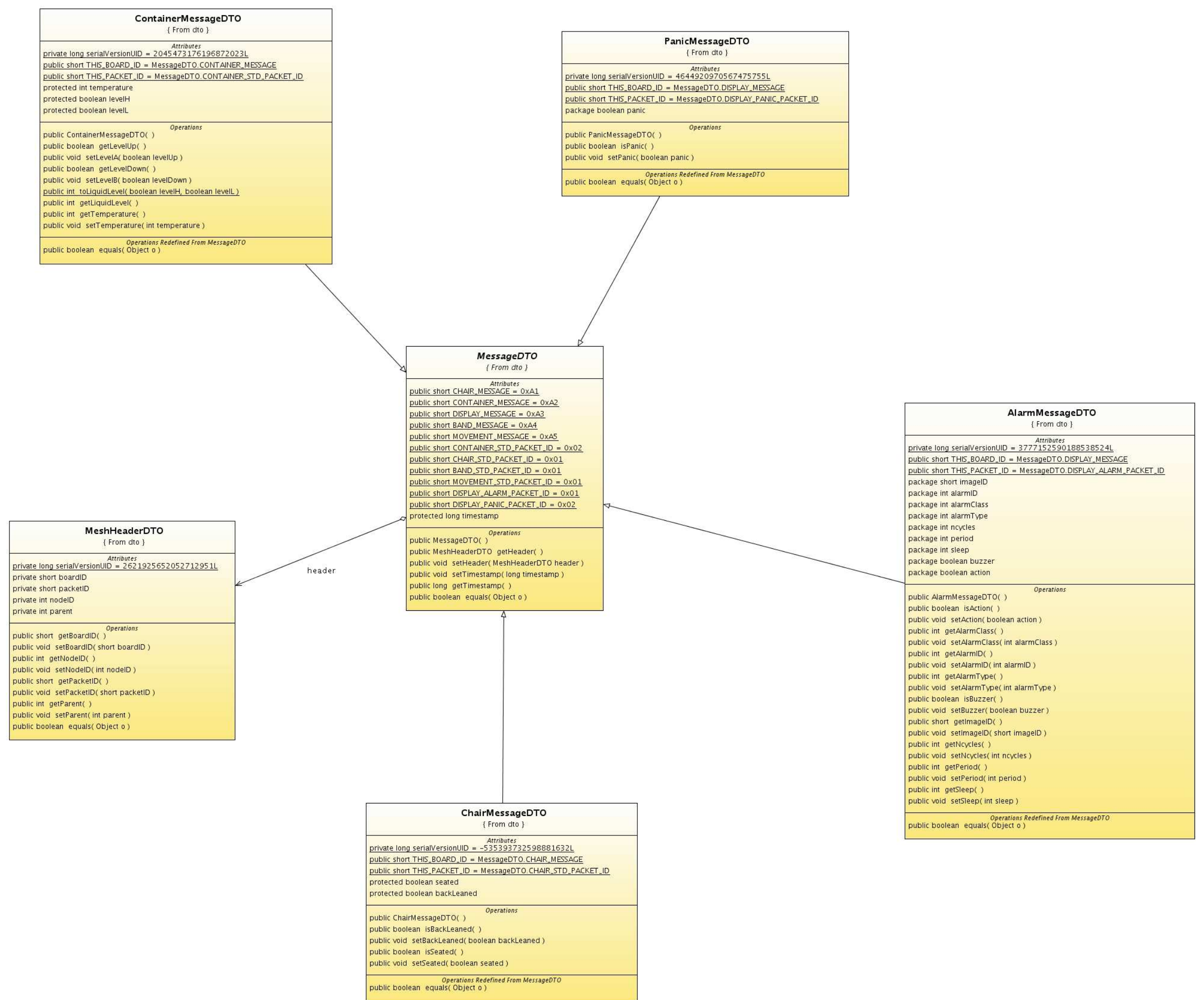
3.1 Introducción

El proyecto *MoteCommunicationSmartLabLib* recoge todas aquellas clases que van a ser utilizadas tanto desde el cliente (*MoteCommunicationSmartLab*) como desde el servidor (*MoteCommunicationBundle*). Dado que el cliente debe correr en gumstix, todas estas clases están implementadas utilizando Java 1.4.

Como ya hemos explicado en la sección *MoteCommunicationSmartLab*, la comunicación entre cliente y servidor se lleva a cabo a través de RMI. En este proyecto se encontrarán todas las clases que se comunican, así como todas las excepciones que se envían y los interfaces comunes que un lado u otro tendrán que implementar.

3.2 DTOs

La información que se envían son DTOs (Data Transfer Object) que representan sus objetos correspondientes en el lado del cliente. El problema de estas clases es que tienen funcionalidades que no tienen sentido en el servidor, así como dependencias de tinyos que no se deben pasar al servidor. De esta manera, los DTOs únicamente contienen los datos y las relaciones de los mensajes que se envían y reciben. La jerarquía de clases de los DTOs es la siguiente:



3.3 Compilando el proyecto

El proyecto cuenta con un fichero de Ant llamado build.xml que compila el proyecto y crea un .jar llamado MoteCommunicationSmartLabLib.jar.

3.4 Añadiendo nuevos dispositivos al proyecto

Para añadir nuevos dispositivos, en este proyecto hay que añadir el mensaje que estos dispositivos envían como DTO, como en el caso del ejemplo sería la clase *es.deusto.tecnologico.smartlab.motes.comm.dto.TimbreDTO*, que debería heredar de MessageDTO. Cuando estos nuevos mensajes se implementasen, se debería revisar la sección de añadir nuevos dispositivos del proyecto *MoteCommunicationSmartLab*, ya que las clases que se crean en este proyecto dependerán de los DTO.

4 MOTECOMMUNICATIONBUNDLE

4.1 Introducción

El *MoteCommunicationBundle* es el bundle que se encarga de gestionar la comunicación con el gumstix y la exposición de la información recibida para el resto de la instancia de OSGi.

En principio, el *MoteCommunicationBundle* no está en el servidor, ya que es el servicio de descubrimiento el que descarga el bundle del gumstix para posteriormente instalarlo. El Bundle lanza un rmiregistry y se queda esperando a que el gumstix contacte con él para posteriormente interrogarle acerca de qué dispositivos gestiona el gumstix.

Con cada uno de estos dispositivos, el bundle registra como tres servicios cada dispositivo. En el caso de una silla, el bundle registrará el interfaz que la silla en sí defina (que está configurado en la configuración del proyecto *MoteCommunicationSmartLab*), y posteriormente registrará un ISmartLabService de manera que el resto de la instancia de OSGi puede acceder a la información semántica del dispositivo. Además, el bundle registrará un EventHandler de OSGi que filtrará los mensajes dirigidos al dispositivo. Así pues, la silla realmente no obtendrá nunca ningún mensaje, ya que no hay ningún escenario que defina una acción que tiene que llevar a cabo la silla, pero por ejemplo el Display (una pantalla integrada en un reloj de pulsera) sí está definido que muestre un mensaje por pantalla cuando haya una inundación. De esta manera, el EventHandler del Display recibirá un evento que defina que hay una inundación, y el Bundle decidirá qué mensaje debe enviar a la mota que controla el Display.

4.2 Registros

Una vez desplegado, el gumstix contactará con el bundle para añadirse al registro, tal y como se ha explicado previamente en la sección “Comunicación con el servidor” de *MoteCommunicationSmartLab*. El *MoteCommunicationBundle* tendrá que registrar en diferentes partes del proyecto tanto las ServiceReferences de OSGi de todos los servicios registrados, como todos los dispositivos a los que se tiene acceso, como todas las referencias de todos los concentradores que tenemos (gumstix) y los propios listeners remotos que el proyecto *MoteCommunicationBundle* instancia y pasa como referencias

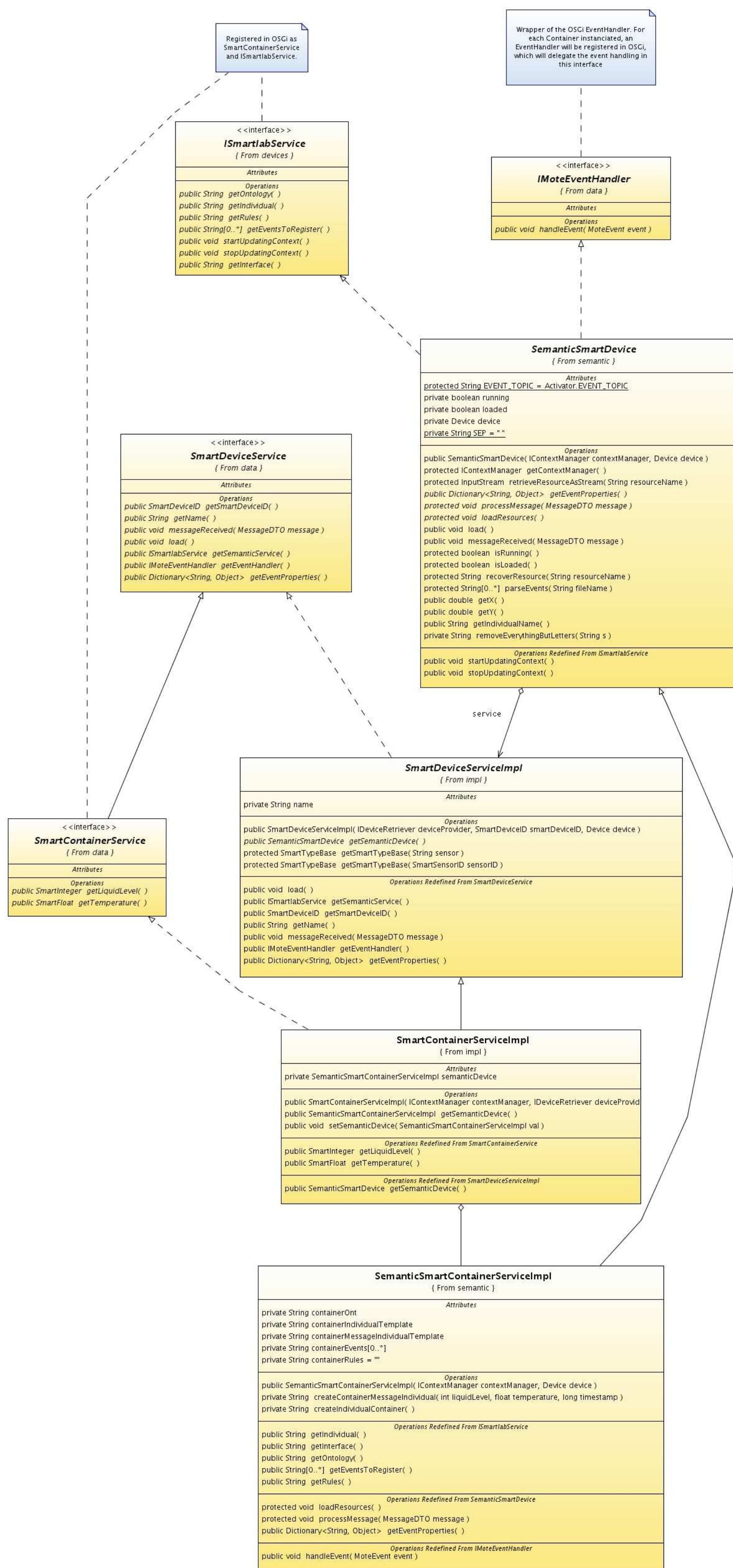
remotas a los diferentes gumstix. De esta manera, el proyecto tiene diferentes registros dependiendo del ámbito de lo registrado. Así pues, los concentradores y los listeners remotos se registrarán en un registro dentro del paquete rmi (MoteCommunicationsRegistry), mientras que el registro de todas las ServiceReferences de OSGi se registrarán en el OSGiDeviceRegisterer del paquete osgi.

4.3 Estructuras de datos

Como se ha explicado anteriormente, cada dispositivo registra un interfaz propio del dispositivo, así como un ISmartLabService y un EventHandler de OSGi. Para llevar esto a cabo, cada dispositivo tendrá su interfaz propio (siguiendo con el ejemplo del timbre, SmartTimbreService) que heredará de *es.deusto.tecnologico.smartlab.motes.comm.server.data.SmartDeviceService*. Además, tendrá la implementación de este interfaz, por ejemplo bajo el nombre de SmartTimbreServiceImpl. Esta implementación heredará además de una clase abstracta, SmartDeviceServiceImpl, que implementa algunos de los métodos definidos en el interfaz SmartDeviceService, y provee de algunos métodos que ayudan a la implementación de las clases que heredan de él (como es el caso de SmartTimbreServiceImpl).

Además, para proveer los servicios semánticos del dispositivo, se ha de crear una clase SmartSemanticTimbreServiceImpl, que implementará el interfaz ISmartLabService (que define métodos para acceder a la información semántica del dispositivo, como obtener la ontología, un individuo, etc. en OWL) y el interfaz IMoteEventHandler (que encapsula al interfaz EventHandler de OSGi, de manera que todo lo que está fuera del paquete “osgi” es prácticamente independiente de OSGi). Sin embargo, esta clase (SmartSemanticTimbreServiceImpl) debería heredar de la clase abstracta SmartSemanticDeviceServiceImpl, que ya implementa ambos interfaces y provee de ciertas funciones de ayuda para las clases que heredan de ella.

A la hora de crear estos objetos, basta con instanciar el SmartTimbreServiceImpl, ya que este debería instanciar al SmartSemanticServiceImpl, y de hecho mantendrá una referencia a esta instancia en todo momento.



Registered in OSGi as SmartContainerService and ISmartlabService.
Wrapper of the OSGi EventHandler. For each Container instanciated, an EventHandler will be registered in OSGi, which will delegate the event handling in this interface
<<interface>>
ISmartlabService
{ From devices }
Attributes
Operations
public String getOntology( )
public String getIndividual( )
public String getRules( )
public String[0..*] getEventsToRegister( )
public void startUpdatingContext( )
public void stopUpdatingContext( )
public String getInterface( )
<<interface>>
IMoteEventHandler
{ From data }
Attributes
Operations
public void handleEvent(MoteEvent event)
SemanticSmartDevice
{ From semantic }
Attributes
protected String EVENT_TOPIC = Activator.EVENT_TOPIC
private boolean running
private boolean loaded
private Device device
private String SEP = " "
Operations
public SemanticSmartDevice(IContextManager contextManager, Device device)
protected IContextManager getContextManager( )
protected InputStream retrieveResourceAsStream(String resourceName)
public Dictionary<String, Object> getEventProperties( )
protected void processMessage(MessageDTO message)
protected void loadResources( )
public void load( )
public void messageReceived(MessageDTO message)
protected boolean isRunning( )
protected boolean isLoaded( )
protected String recoverResource(String resourceName)
protected String[0..*] parseEvents(String fileName)
public double getX( )
public double getY( )
public String getIndividualName( )
private String removeEverythingButLetters(String s)
Operations Redefined From ISmartlabService
public void startUpdatingContext( )
public void stopUpdatingContext( )
<<interface>>
SmartDeviceService
{ From data }
Attributes
Operations
public SmartDeviceID getSmartDeviceID( )
public String getName( )
public void messageReceived(MessageDTO message)
public void load( )
public ISmartlabService getSemanticService( )
public IMoteEventHandler getEventHandler( )
public Dictionary<String, Object> getEventProperties( )
service
SmartDeviceServiceImpl
{ From impl }
Attributes
private String name
Operations
public SmartDeviceServiceImpl(IDeviceRetriever deviceProvider, SmartDeviceID smartDeviceID, Device device)
public SemanticSmartDevice getSemanticDevice( )
protected SmartTypeBase getSmartTypeBase(String sensor)
protected SmartTypeBase getSmartTypeBase(SmartSensorID sensorID)
Operations Redefined From SmartDeviceService
public void load( )
public ISmartlabService getSemanticService( )
public SmartDeviceID getSmartDeviceID( )
public String getName( )
public void messageReceived(MessageDTO message)
public IMoteEventHandler getEventHandler( )
public Dictionary<String, Object> getEventProperties( )
<<interface>>
SmartContainerService
{ From data }
Attributes
Operations
public SmartInteger getLiquidLevel( )
public SmartFloat getTemperature( )
SmartContainerServiceImpl
{ From impl }
Attributes
private SemanticSmartContainerServiceImpl semanticDevice
Operations
public SmartContainerServiceImpl(IContextManager contextManager, IDeviceRetriever deviceProvid
public SemanticSmartContainerServiceImpl getSemanticDevice( )
public void setSemanticDevice(SemanticSmartContainerServiceImpl val)
Operations Redefined From SmartContainerService
public SmartInteger getLiquidLevel( )
public SmartFloat getTemperature( )
Operations Redefined From SmartDeviceServiceImpl
public SemanticSmartDevice getSemanticDevice( )
SemanticSmartContainerServiceImpl
{ From semantic }
Attributes
private String containerOnt
private String containerIndividualTemplate
private String containerMessageIndividualTemplate
private String containerEvents[0..*]
private String containerRules = ""
Operations
public SemanticSmartContainerServiceImpl(IContextManager contextManager, Device device)
private String createContainerMessageIndividual(int liquidLevel, float temperature, long timestamp)
private String createIndividualContainer( )
Operations Redefined From ISmartlabService
public String getIndividual( )
public String getInterface( )
public String getOntology( )
public String[0..*] getEventsToRegister( )
public String getRules( )
Operations Redefined From SemanticSmartDevice
protected void loadResources( )
protected void processMessage(MessageDTO message)
public Dictionary<String, Object> getEventProperties( )
Operations Redefined From IMoteEventHandler
public void handleEvent(MoteEvent event)

4.4 RMI

Como se ha comentado previamente, el proyecto utiliza RMI para la comunicación con el gumstix, y el rmiregistry es lanzado por el bundle cuando este es activado. Para ello, crea una carpeta temporal en la que pone los archivos que rmiregistry necesita (los .jar). El código que gestiona el rmiregistry se encuentra en el paquete *es.deusto.tecnologico.smartlab.motes.comm.server.rmi.deployment*.

4.5 Despliegue en OSGi

Las opciones de arranque de OSGi han de ser, al menos, las siguientes:

```
-Dorg.knopflerfish.verbosity=0
-Dorg.knopflerfish.gosg.jars=file:jars/
-Dorg.knopflerfish.framework.debug.packages=false
-Dorg.knopflerfish.framework.debug.errors=true
-Dorg.knopflerfish.framework.debug.classloader=false
-Dorg.knopflerfish.framework.system.export.all_13=false

#Anyadidos
-Dorg.osgi.framework.system.packages=sun.reflect
-Dorg.osgi.framework.system.packages.file=packages1.5.txt
-Dorg.osgi.framework.bootdelegation=*
-Djava.security.policy=java.policy


-Dorg.knopflerfish.startlevel.use=true

-init

-install log/log_api-2.0.0.jar
-install console/console_api-2.0.0.jar
-istart  cm/cm_api-2.0.0.jar
-istart  cm/cm-2.0.0.jar
-istart  log/log-2.0.0.jar
-istart  console/console-2.0.0.jar
-istart  consoletty/consoletty-2.0.0.jar
-istart  frameworkcommands/frameworkcommands-2.0.0.jar
-istart  logcommands/logcommands-2.0.0.jar
-istart  useradmin/useradmin_api-2.0.0.jar
-istart  component/component_all-2.0.0.jar
-istart  event/event_all-2.0.0.jar


-istart bundlerepository_smartlab-1.0.0.jar
-istart localrepository-1.0.0.jar
-istart installer-1.0.0.jar
-istart discovery_service-1.0.0.jar

-launch
```

4.6 Compilando el proyecto

El proyecto cuenta con un fichero Ant llamado build.xml encargado de compilar el proyecto. La tarea dist genera el .jar del bundle, que debería ser suficiente para desplegar el proyecto, ya que es el gumstix el que, a través del protocolo de descubrimiento, enviará ese .jar, y será el servicio de descubrimiento el que instale y active al bundle. Sin embargo, para poder realizar pruebas sin este protocolo, se han definido más tareas en el build.xml, como la tarea "deploy", que copia el bundle en OSGI_HOME.

4.7 Añadiendo nuevos dispositivos al proyecto

Para añadir nuevos dispositivos al proyecto, es necesario:

1. Haber creado previamente los DTO en el proyecto *MoteCommunicationSmartLabLib*, (por ejemplo *TimbreMessageDTO*).

2. En el paquete *es.deusto.tecnologico.smartlab.motes.comm.server.data*, definir un interfaz que herede de SmartDeviceService y que añada métodos para preguntar al dispositivo por valores de diferentes sensores (por ejemplo, SmartTimbreService).

3. Crear una clase (por ejemplo, *SmartSemanticTimbreServiceImpl*) en el paquete *es.deusto.tecnologico.smartlab.motes.comm.server.data.impl.semantic*, crear una clase que herede de *SmartSemanticDevice*, y que tenga la información semántica del dispositivo. Esta clase dependerá de TimbreMessageDTO.

4. En el paquete *es.deusto.tecnologico.smartlab.motes.comm.server.data.impl*, crear una clase que herede de *SmartDeviceServiceImpl* e implemente el interfaz creado en el paso 2 (*SmartTimbreService*). Esta clase además instanciará la clase *SmartSemanticTimbreServiceImpl*.

4.8 Pruebas

Al igual que el proyecto *MoteCommunicationSmartLab*, el proyecto *MoteCommunicationBundle* cuenta con un pequeño número de pruebas unitarias en la carpeta "test", desarrolladas utilizando JUnit 4. Estas pruebas cubren desafortunadamente una pequeña parte del proyecto.