Université Libre de Bruxelles
Faculté des Sciences appliquées

Helsinki University of Technology

Master's Thesis

Open Service Gateway Implementation for House Monitoring

Pierre Moermans

Supervisor at the HUT:
Professor Aarne Halme
Instructor at the HUT:
Panu Harmo, MSc
Supervisor at the ULB:
Esteban Zimanyi, PhD

Contents

Chapter 1

Introduction

1.1 Overview of the work

The work consists in the integration of different networks of sensors and actuators and the development of an architecture for a programmable logic system which uses those networks. The software developed in this work will use the OSGi (Open Service Gateway implementation) specifications.

Each network uses a different technology to access its components. This work deals with three networks. A first network called *SINE* is a prototype developed in the laboratory of applied electronics of the Helsinki University of Technology. The second one, called *X10*, uses, the X-10 technology for its devices communication. The third network is a *Linet* network, which is developed by the Linet® company.

The programmable logic system consists in a software which automates tasks, based on the values of the sensors and actuators. The system must provide a configuration mechanism to allow modifications and updates of the logic.

1.2 Integration in a larger project

The work is integrated in a larger project called *TerveTaas*, which means *Welcome/Health Again*, leaded by Panu Harmo. The project consists in building a system that can help elderly people and people with diseases to live more comfortably and more securely in their **own** house. By more comfortably, we mean that tasks which are painful or difficult to achieve, should be automated or accomplished remotely. By more securely, we mean that the house should constantly be monitored so that the house owner or other distant persons can be alerted in case of danger.
Two kinds of monitoring are used in the project:

1. **Rollo**. Rollo is a ball-robot equipped with a camera and other devices. It can roll through the house and film the environment.

2. **Sensors and Actuators**. Fixed sensors are the most common way to monitor the house. Many kinds of sensors are used: motion sensors,

temperature sensors, ... Actuators are small devices like door openers and closers, shutter controls, ... They are used to interact with the house.

This work doesn't deal with Rollo, but is highly tied up with the fixed sensors and actuators.
The usage of standard sensors to build the logic has many drawbacks:

1. **Wiring**. The house in which the TerveTaas system should be installed are not built from scratch around the system. They are generally old houses, already wired for the power-supply but with nothing planned for the sensors and actuators wiring. Installing standard sensors and actuators would require a heavy work to make holes and grooves in the walls and so on. This work is dirty and expensive.
2. **Logic**. Providing automation with standard devices is difficult since it requires even more wiring. The connections are difficult to achieve. This drawback reinforces the first one.
3. **Updatability**. A wired logic would be hard to modify or update since it requires a dirty work of rewiring.

The newer technology used by the three networks presented before overcomes those problems. The common goal of those networks is to limit the wiring, either by providing wireless communications with the network components, or by providing a network architecture that limits the *logic wiring*. The X10 and Linet network also provide their own logic system, having it in mind to provide house automation. While those network specific logic systems are suitable for "light" automation, they have a drawback for the more important automation(logic) needed by the TerveTaas project: it is not possible to make those networks communicate with each other, and therefore, a logic involving sensors and actuators from different networks is not possible either. This is where the programmable logic system developed in this work takes place. This system, using the different networks integration, will provide a logic that can use sensors and actuators from the different networks.

The safety aspect of the project involves the possibility to monitor the house from elsewhere. Several ways to achieve this are developed in the project, but one is of particular interest in the work: the monitoring through Internet. Reading the devices through the Internet requires an interface between the Internet and the different home networks. This link must act as a gateway. The project structure appears like in Figure 1.1. That shows the project as it should be in the final state. This work doesn't deal with the connection to Rollo, but it is planned to be later integrated in the architecture.

1.3 Phases of the work

As we will develop the whole software using the OSGi architecture, a first step will be to explain what this architecture consists in.
The work is split in two parts. A first part deals with the development of drivers needed to access the different networks. A second part will deal with the requirements to achieve the house-wide logic.

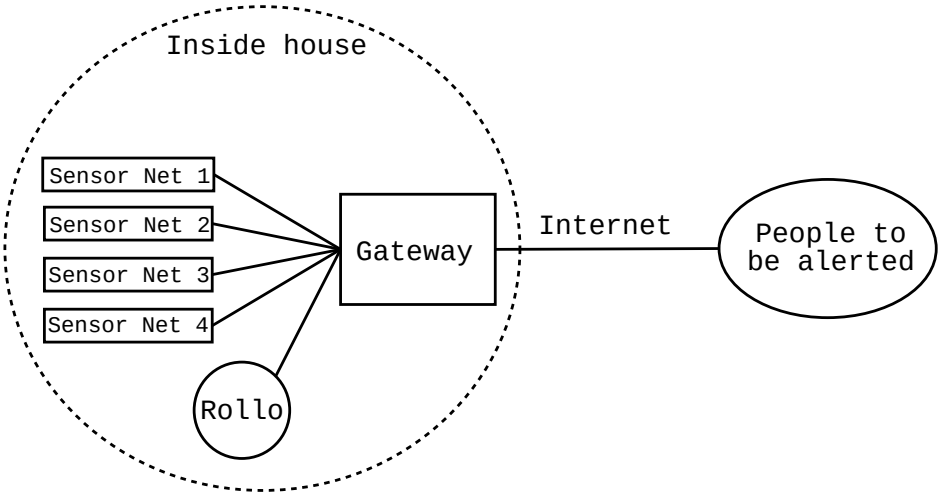

Figure 1.1: Project environment.

1.3.1 Drivers

In this part, a chapter is dedicated to each network. First, a description of the network will be provided. The driver to access it will then be explained, first on the architectural point of view, then on the implementation itself.

1.3.2 House-wide logic

This second part involves:

1. the development of a standard access to the networks.
2. the integration of the different networks in one house-wide interface.
3. the development of the programmable logic itself.

The link to the drivers will be described for the first of these steps.
The work doesn't build a complete system but establishes a complete architecture of the programmable logic system. The building blocks of the systems are explained, as well as the interfaces between those blocks, but the implementation itself is not part of this work.

Chapter 2

Open Service Gateway

The whole work is based on a new architecture called *Open Service Gateway (implementation)* (OSGi) [1]. The use of this architecture implies a new approach to programming. As this architecture is not yet widely used and popular, in this chapter, we will try to demystify the concepts around it. The programming model will be explained using a short illustrative example.

2.1 What is a service gateway?

A service gateway is a central device that interfaces the external Internet and internal devices placed in a house [2]. The internal devices can be many things, like the central heating system, the DVD recorder, the fire alarm, etc... Through the gateway, the house owner or external agents like the gas company can interact with the internal devices. The ability to communicate with internal devices from outside the house is a benefit for both the house owner and the service providers. For the consumer(the house owner), there are at least two advantages: it allows him to have the service exactly when and how he needs it; he can bypass intermediaries and therefore hope to have a cheaper service. For the service provider, accessing the internal devices allows him to provide a better focused and fitting product and therefore to improve its commercial offer. The service providers could all have their own communication devices, but this generates many complications. The service gateway offers many advantages:

- With the diversification of the internal devices, each device having its own specific connection with the outside world, the situation will rapidly become impossible to manage. The service gateway provides a unique access point and therefore overcomes this problem.
- The service gateway allows the integration of the different services. For example, the fire alarm could both call the firemen and call on the house owner's mobile.
- The service gateway allows the internal devices to communicate with each others. A good example of this is given in *'Programming Open Service Gateways with Java Embedded Server Technology'* [2]: It is

possible for the home theater system to send a signal to lower the automated window blinds when a movie starts on a Sunday afternoon.

2.2 The Open Service Gateway initiative

The OSGi consortium [1] defines itself like this: OSGi was founded in March, 1999, with the objective of providing a forum for the development of open specifications for the delivery of multiple services over wide-area networks to local networks and devices, and accelerating the demand for products and services based on those specifications worldwide through the sponsorship of market and user education programs. The OSGi specification can be found at the organization web page: `http://www.osgi.org/resources/spec_overview.asp`.
Providing a standard allows:

- **Easier communication between and with the internal devices.** Devices developed by different companies will generally use different communication technologies, both from the hardware layer point of view (Ethernet, Bluetooth, ...)[3] and from the discovery protocol (UPnP, Jini, ...), [4], [3]. A standard allows the coexistence and the integration of those different technologies.

- **Platform independence.** Different gateways use different operating systems and hardware. The same diversity is even more probable for embedded devices than for computers. It is of course impossible for the service providers to write code to support every device. Having a standard allows a service provider to *develop once and run everywhere.*

- **Vendor independence.** By defining common APIs to access services provided on the gateway, the OSGi architecture allows different service providers to communicate with different brands of devices.

2.3 Java Embedded Server technology

Sun Microsystems has developed a Service Gateway that meets the OSGi standard [5],[6],[7]. This product is called Java Embedded Server (JES). As the name implies, it is entirely developed using the Java language.
Java has been commonly accepted by developers as the language of choice to develop applications over Internet. The Java language has many advantages. The first one is the Platform independence. JES can be run on different operating systems like Solaris, Windows, Mac OS X, Linux or BSD. The second important advantage of Java is probably that is is by far less fault prone than C or C++. The development of service applications and embedded devices is therefore easier and faster.

A JES based gateway is composed of two layers. The first layer is what Sun Microsystems calls the ServiceSpace and the second is the applications of the service providers, called *services.* The JES framework contains already made services like an HTTP server, a log service and a scheduler. The JavaSpace provides several really useful facilities:

1. **On-demand Java applications.** A service running on JES can dynamically download a Java application from everywhere on the network (and Internet), install it and run it.

2. **Life-cycle management.** Services can be installed, started, stopped, customized and updated dynamically when needed.

3. **Service discovery and Dependency resolution.** Services can depend on other services. When a service is loaded, the JES finds the dependencies and looks at the available running services. If a required service is not available yet, it looks for it, and if it finds it, loads it and starts it. Some dependencies are required and some are optional. It means that, if a service is required but not found, the service that requires it will not start. On the other hand, if a service is optional, the service that asked for it, will start but with less capabilities.

More detailed information is available on the Sun web site at the address . `http://java/sun/com/software/embeddedserver/`.

2.4 Development of a service application

2.4.1 Architecture

Figure 2.1 depicts the gateway architecture. The framework is what Sun Microsys-

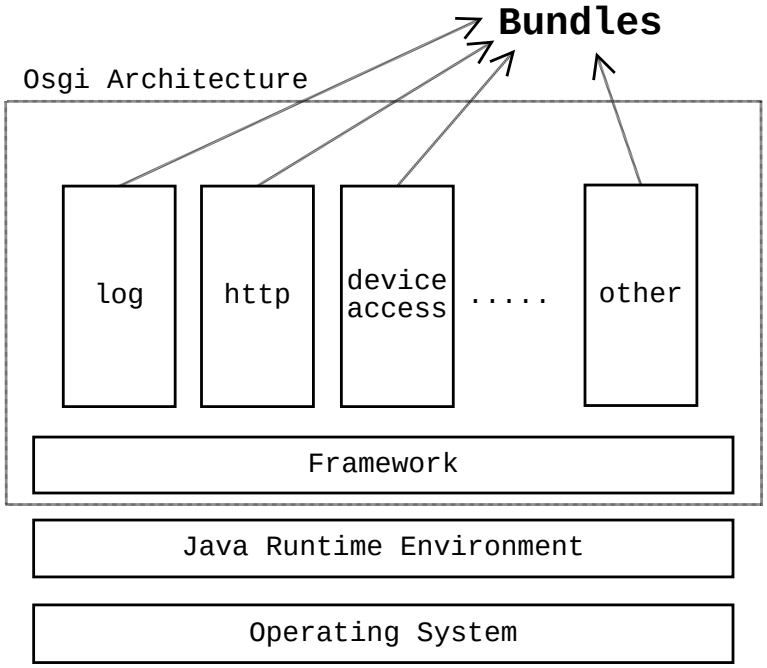

Figure 2.1: gateway architecture.

tems calls the ServiceSpace. This layer is responsible for the management (loading, starting, stopping, ...) of the service applications. The bundles are the file structures that contain the service applications. The OSGi standard defines two standard services: the HTTP service and the Log service. It also defines a structure to handle device access. The role of the service providers is to develop service applications to extend the functionalities of the service gateway.

2.4.2 Service

A service is a program developed in the Java language that does something useful. For example, a program that warns you when your house is burning, is a service. Every service code must be separated in two parts: the interface and the implementation.
The interface is a normal Java interface. It exposes the features offered by the service to the other services.
The implementation is composed of the classes (and maybe other interfaces) and is hidden from the other services. Let's take the example of a CD player service. A

user interface on this CD player is not interested in knowing how the CD is read, the only thing that matters, is that it can call readTrack(int trackNumber) on the CDPlayer class. That readTrack method must be described in the CDPlayer service interface. If we want to update the CDPlayer service, it is possible to change dynamically the implementation of the service, but not the interface. The only way to update the interface is to stop the JES and restart it but this is not what we want. This update limitation is the main reason why we should separate the service implementation from the service interface.

2.4.3 Bundle

A bundle can be seen from two points of view. It is both a packaging for services and a functional unit with its own life cycle and class loading capability.

Bundle as a packaging

A bundle is a JAR file that contains class files and resources like images or XML documents. Packaging both the class files and the resources together makes the deployment a lot easier. The jar file must contain a manifest file that describes the bundle. The content of the manifest will be explained later.

Bundle as a functional unit

A bundle has also a life-cycle. It can be loaded(installed), started, stopped and unloaded(uninstalled).
Figure 2.2 depicts the life-cycle.

- **install**. The framework retrieves the files of the bundle and installs it on the gateway. The newly installed bundle is assigned a unique id. The bundle goes in the *resolved* state when all the packages it imports have been found and all the packages it exports have been so.
- **start**. The framework calls the *start* method of the bundle activator. The bundle is in the *Starting* state until the *start* method returns.
- **stop**. The framework calls the *stop* method of the bundle activator. The bundle is in the *Stopping* state until the *stop* method returns.
- **uninstall**. The framework removes the bundle from the gateway.
- **update**. The framework stops the bundle, reloads the bundle files and restarts it.

After each step, the framework broadcasts an event to specify the new bundle state. A bundle is a self contained package, a class that belongs to one bundle cannot be imported by a class from another package. Of course, the classes in CLASSPATH are always accessible to every bundle. We will see later that a bundle can get a reference to another bundle specific class through the usage of services.

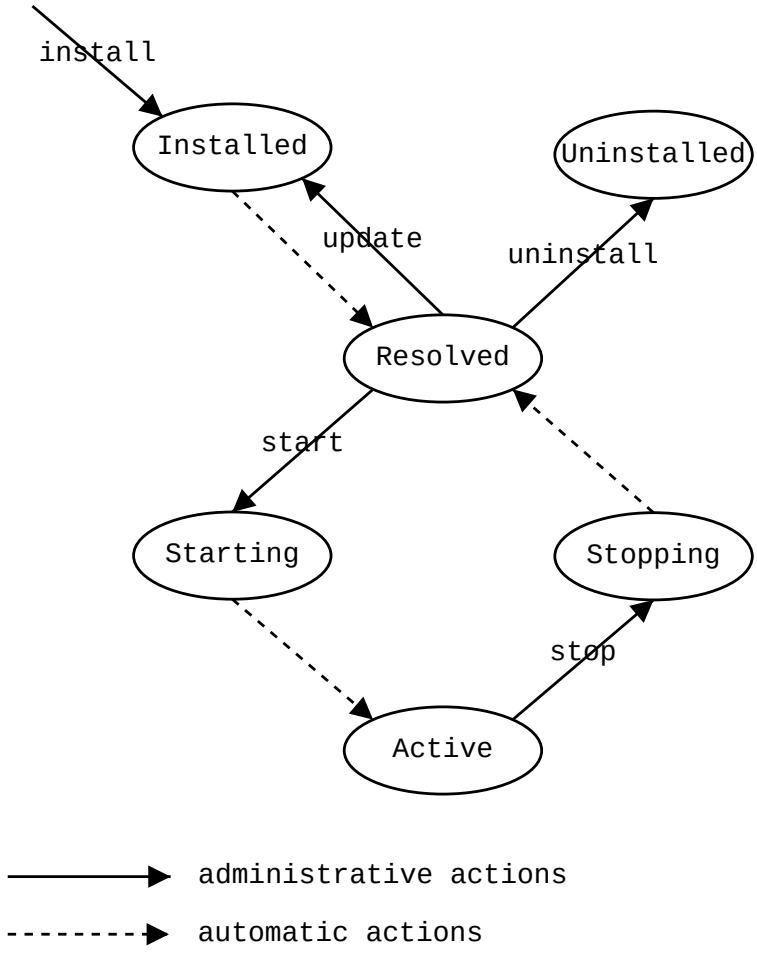

Figure 2.2: bundle lifecycle.

Links between bundle and framework

As a bundle is not started in its own JVM (Java Virtual Machine) but instead is started by the framework, a specific class that contains a *start* and a *stop* method must exist in the bundle. This class is called the *bundle activator*. The name of the bundle activator class is one of the information found in the manifest. The bundle activator implements an interface *BundleActivator* that looks as follows:

```
1      public interface BundleActivator
2      {
3          public void start(BundleContext);
4          public void stop(BundleContext);
5      }
```

Manifest

In each bundle, there must be a manifest file to describe it. This manifest file is read by the framework when a bundle is installed. The manifest contains a list of key-value pairs. Three kinds of information are found in the manifest file:

- **Bundle Activator.** The bundle activator gives the name of the class that implements the BundleActivator interface. This information is required; without it, the framework will not install the bundle.
- **Exported Package.** It describes the package(s) exported by the bundle. The exported package contains the interface the service offers. This feature will become clear while looking at the example below. The *Export-Package* key-value pair is only needed if the bundle exports a service (it is not always the case).
- **Imported Package.** It describes the package(s) that the bundle needs to import in order to start. Of course, the *Import-Package* key-value pair is only needed if the bundle must import a package exported by another bundle.

2.5 Example

We give in this section an example illustrating the previous concepts.

Let us consider a cook's house. In that house, there is a kitchen that only contains a microwave. Given the weight of the meat, the microwave can tell the cook how much time is needed to cook the meat. To represent this with an service gateway, we can consider two bundles: *kitchen* and *cook*. The microwave can be considered as a service exported (provided) by the *kitchen* bundle and imported (needed) by the *cook* bundle.

2.5.1 *kitchen* bundle

The kitchen bundle is composed of the four following files

- **Kitchen.java** is the bundle activator. It is responsible to "say to the framework" that it exports the microwave service. It will also instantiate a Microwave object.
- **MicrowaveServ.java** contains the interface of the microwave service.
- **MicrowaveImpl.java** contains the implementation of the microwave service.
- **manifest** describes the bundle.

The code of *MicrowaveServ.java*

```
1  /**
2   * MicrowaveServ.java
3   *
4   * @author Pierre Moermans
5   */
6
7  package kitchen.service;
8
9  public interface MicrowaveServ
10 {
11         public void setWatt(int watt);
12
13         public int getWatt();
14
15         public float getCookingTime(int weight);
16 }
17
```

The code of *MicrowaveImpl.java*

```
1  /**
2   * MicrowaveImpl.java
3   *
4   * @author Pierre Moermans
5   */
6
7  package kitchen.implementation;
8
9  import kitchen.service.MicrowaveServ;
10
11 public class MicrowaveImpl implements MicrowaveServ
12 {
13         private int watt;
14
15         public void setWatt(int watt)
16         {
17                 this.watt = watt;
18         }
19
20         public int getWatt()
21         {
22                 return watt;
23         }
24
25         public MicrowaveImpl(int watt) {
```

```
26                      this.watt = watt;
27                      System.out.println("microwave: 1kg --> 30 min");
28              }
29
30              public float getCookingTime(int weight) {
31                      return ((float)weight/watt)*30;
32              }
33      }
34
```

The *MicrowaveServ.java* and *MicrowaveImple.java* have nothing different from a "standard" code.

The code of the bundle activator *Kitchen.java* follows.

```
1
2  /**
3   * Kitchen.java
4   *
5   * @author Pierre Moermans
6   */
7
8  package kitchen;
9
10 import org.osgi.framework.BundleActivator;
11 import org.osgi.framework.BundleContext;
12
13
14 import kitchen.service.*;
15 import kitchen.implementation.*;
16
17 import java.util.Properties;
18
19 public class Kitchen implements BundleActivator
20 {
21         public void start(BundleContext ctxt) {
22                 Properties props = new Properties();
23                 props.put("watt", new Integer(1000));
24                 MicrowaveServ micro =
25                         new MicrowaveImpl(1000);
26                 ctxt.registerService(
27                         "kitchen.service.MicrowaveServ",
28                         micro,props);
29                 System.out.println(
30                         "Microwave service registered.");
31         }
32
33         public void stop(BundleContext ctxt) {
34                 System.out.println("bye.");
35         }
36 }
37
```

The class consists of two methods. The *start* method registers a service *kitchen.service.MicrowaveServ* with an instance of it and a property. The variable *micro* is initialized with the interface *MicrowaveServ* type. The properties describing the service can be used by other bundles to find the right service. In this case, if two microwave services are provided with different power (watt), a bundle could request one of at least 1000 watt. The *stop* method is called by the framework to stop the bundle.

The manifest is as follows:

```
1  Export-Package: kitchen.service
2  Bundle-Activator: kitchen.Kitchen
```

2.5.2 *cook* bundle

This bundle is composed of two classes:

- **Cook.java**, the bundle activator, and
- **CookImpl.java**, an implementation of a cook.
- **manifest** described the bundle.

The code of the *CookImpl* class shows the "standard" usage of a class (as an interface) that has been imported by the bundle, in this case *MicrowaveServ*.

```
1  /**
2   * CookImpl.java
3   *
4   * @author Pierre Moermans
5   */
6
7  package cook.implementation;
8
9  import kitchen.service.*;
10
11 public class CookImpl
12 {
13         private MicrowaveServ micro;
14
15         public boolean proceed = true;
16
17         public CookImpl(MicrowaveServ micro) {
18                 this.micro = micro;
19         }
20
21         public void cook() {
22                 System.out.println(
23                         "To cook 1.5kg I need " +
24                         micro.getCookingTime(1500) +
25                                 " minutes.");
26         }
```

```
27  }
28
```

The bundle activator does two things in its start method.

- Getting an instance of a microwave service.
- Creating an instance of CookImpl.

```
1   /**
2    * Cook.java
3    *
4    * @author Pierre Moermans
5    */
6
7   package cook;
8
9   import org.osgi.framework.BundleActivator;
10  import org.osgi.framework.BundleContext;
11  import org.osgi.framework.ServiceReference;
12
13  import kitchen.service.MicrowaveServ;
14  import cook.implementation.CookImpl;
15
16  public class Cook implements BundleActivator
17  {
18          private CookImpl cook;
19
20          public void start(BundleContext ctxt) {
21                  System.out.println(
22                          "starting cook bundle");
23                  ServiceReference ref =
24                          ctxt.getServiceReference(
25                          "kitchen.service.MicrowaveServ");
26                  MicrowaveServ micro =
27                          (MicrowaveServ)ctxt.getService(ref);
28                  cook = new CookImpl(micro);
29                  cook.cook();
30          }
31
32          public void stop(BundleContext ctxt) {
33                  System.out.println(
34                          "cook bundle stopped");
35                  cook.proceed = false;
36          }
```

```
37    }
38
```

In this short example, we want any microwave service but we could be more selective by using an extended

getServiceReferences(String service, String searchString)

method. The *searchString* string follows the LDAP search construction.

The following listing is the manifest:

```
1   Bundle-Activator: cook.Cook
2   Import-Package: kitchen.service
```

The developer should be aware of one pitfall: the *CLASSPATH*. When the cook bundle is compiled, the *CLASSPATH* must include the *kitchen* package but at run-time, the *CLASSPATH* CANNOT contain the *kitchen* the package. This is due to the order in which the JES resolves the dependencies. It first looks in the bundle, then in the *CLASSPATH* and finally in the list of packages exported by other bundles. If, at run-time, the *CLASSPATH* contains the *kitchen* package, the dependency of JES is bypassed. To look further in OSGi and the JES framework, you can refer to the *Programming Open Service Gateways with Java Embedded Server* [2].

Part I

Drivers

Chapter 3

SINE Network

3.1 Network description

The SINE network we use is developed by the laboratory of applied electronics. It is still in a development phase and must still be improved. The SINE network is composed of several sensors and a controller. Whenever it gets data from one of the sensors, the controller forwards the information to the computer through a serial connection. The sensors use radio communication to send data to the controller. A sensor sends two different kinds of information. Every fixed time (about 20 seconds), the sensor sends a heartbeat (a message to say *I'm still there*). The first kind of message is thus a "presence" message. The other kind of information is a "state changed" message. When a sensor status changes, it notifies the controller by sending a message containing its new state. The sensor actually have only two states: alarm-on and alarm-off. The big advantage of the SINE sensors is that they can be directly connected to the standard 220V power supply. In this way, we can avoid extra wiring. The sensors are composed of two components: the SINE connection and a "standard" sensor acting as a switch. To the author's opinion, the SINE network suffers from a defect: the "state-changed" message is not sent directly after the change. Under this restriction, it is currently not possible to use the sine network for any kind of instantaneous alarm.
The only thing we are actually concerned about is the communication with the SINE controller through the serial port.

3.2 Protocol

A short protocol draft has been provided by the laboratory of electronics. Every time the controller receives information, it sends a 6 bytes packet to the serial port. The first 5 bytes are "undefined", the only information carrier byte is the last one. This last byte must be read as 8 independent information bits.

The following table describes the 8 bits.

bit	description
1	not used
2	not used
3	device 1 alarm
4	device 2 alarm
5	device 3 alarm
6	device 1 connection
7	device 2 connection
8	device 3 connection

Table 3.1: Meaning of the different bits found in the protocol.

An alarm bit value 0 indicates an alarm-on state.
A connection bit value 0 indicates a connected device.
It should be noted that the protocol provided by the laboratory of electronic didn't match with their hardware. The state bits are actually reversed in the protocol. This has been easily verified since the controller has 6 lights to indicate the alarm and connection state of each of the four devices.

As we can see, with the current protocol, only three sensors can be connected. The hardware provided was a prototype. In the future, a byte could be used to indicate how many data bits are going to be transferred, as it is done in most of the actual data transmission protocols.

3.3 Integration in the OSGi architecture

Three sensors are not sufficient to monitor a house.In a real installation (outside the lab), more than a single SINE controller will be connected to the gateway. Following the OSGi architecture optic, the idea is to provide a *sine service* for each SINE controller. For this reason, some management of the serial port is needed. This serial port management must appear as a bundle that "distributes" the serial ports used by the sine services. Using that architecture, a *serial* bundle is first developed, followed by a *sine* bundle using it. As research has been done for a nice integration of the serial devices, we will develop what should have been a nice integration and what was possible due to hardware shortcomings of the prototype.

3.3.1 First approach

The first approach used the built-in *device* facility defined by the OSGi framework. This device facility allows automatic service registration to provide access to hardware (and other bundles). The service can be used to automatically start services that provide an access to some hardware. Applied to the serial port, the device facility requires:

1. A discovery of the device connection. This is done by using the CTX pin of the serial port. A device connected to the serial port will put the logical potential level of this pin to 1. Based on this potential level, the serial port controller can know if a device is connected.

2. A way to identify the device connected. This can be done by interacting with the device. For example, to discover a modem, we can send an "ATE0" message to the serial port. If the device answers with "OK", it's a modem.

This feature would have allowed us to detect and start automatically a SINE service when a controller is connected.
Two problems occurred. First, the controller doesn't use the CTX pin and therefore cannot be detected. A work around for this exists: we can bypass the device detection by polling the serial port with the message to identify the device.The second problem is the lack of answering mechanism on the controller. Therefore, the workaround couldn't be applied either.
The usage of the CTX pin and a simple message answering mechanism would improve the controller considerably.

3.3.2 Adopted solution

Considering the shortcomings, the only solution to manage the serial ports is to allocate them on demand. When a *sine* bundle is started, it asks the serial port management bundle to allocate the port (that means, put a lock on it). In this way, we don't have the automatic service registration, but at least, we can remotely manage the serial ports and if two bundles try to access the same port, a accurate exception is thrown.

3.4 *serialManagement* bundle

The purpose of this bundle is

1. to secure the access to the serial port by allowing prevention.

2. providing information concerning the usage of the serial ports.

3.4.1 Securing

The implementation of the JavaComm(Windows/Solaris)[8]/RXTX(Linux)[9] APIs prevents serial port access conflicts in two ways:

1. by allowing only one *SerialPortEventListener* to listen to the port. A *TooManyListenersException* is thrown when invoking the *addEventListener(...)* method on the serial port if there is already a listener on that port.

2. by refusing two programs to open the same serial port. A *PortAlreadyInUse* exception is thrown when invoking the *open* method on the serial port if the port is already opened.

Those two mechanisms are efficient, but do not fit to the management of the ports. The *serialManagement* bundle provides a way to get information on the serial ports before connecting to them.

3.4.2 Information gathering

The *serialManagement* bundle provides methods to know which ports are used but also to know which ports are available on the system.
For more information on the provided methods, refer to the API documentation (HTML format provided on the CD).
The following UML diagram shows the structure of the bundle. As we can see,

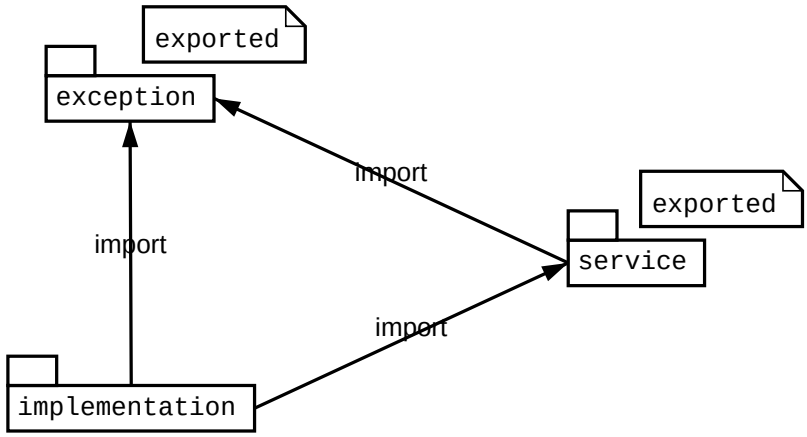

Figure 3.1: serialManagement bundle package structure.

there are three packages. The *service* and *implementation* packages are the ones described earlier. The packages noted as *exported* are the packages the bundle exports. The package *exception* contains the exception related to the serial ports management. This package is exported. This could seem a bit strange since, as

we have seen before, any package that is exported is **not updatable**. This is a drawback, but on the other hand, the client of the *clientManager* service must know about the exceptions, so the package **must** be exported. In this case, the defect is not so important since the exceptions are unlikely to change, therefore, the *package* will not require any updates. We could ask ourselves the question: Why aren't the events inside the service package then? This could perfectly be and will work. We chose to separate the events from the services to clarify the structure. The events are not services in the sense of the OSGi definition. The service is to offer a listening system through the *addListener* method, but not the events themselves.
The diagram 3.2 describes the *serialManagement* bundle.

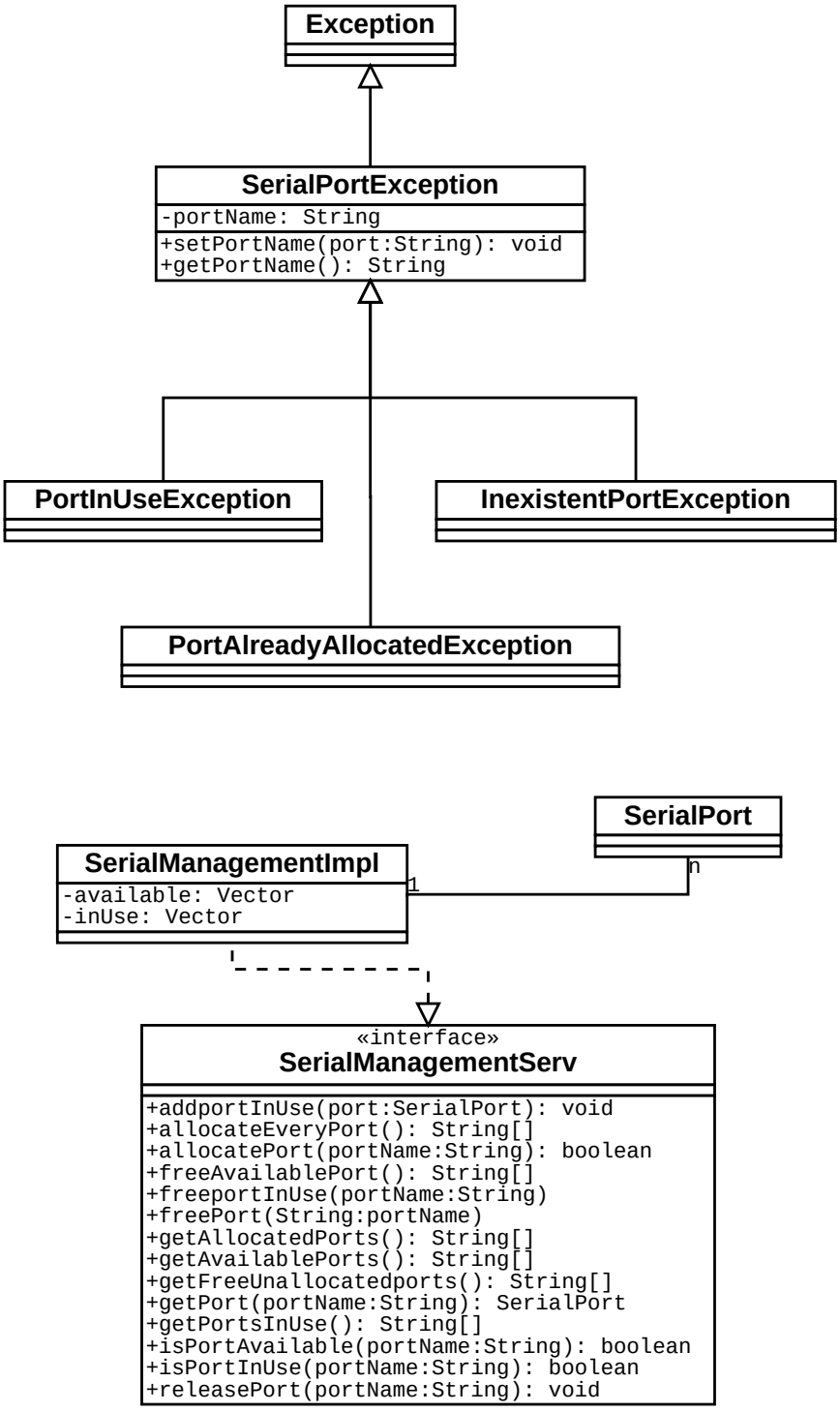

Figure 3.2: serialManagement bundle classes.

3.5 *sine* bundle

The goal of the *sine* bundle is to provide an event-driven way to know the status of a SINE network sensors. The *sine* bundle is responsible for reading the serial port. It translates the information sent by the controller into *SensorStatus* classes (actually, the *SensorStatus* is an interface implemented by a *SensorStatusImpl* class), compares the new status with the previous one and if a change has occurred, it fires a *statusChanged* method on every listener. Having this architecture, it becomes a lot easier to implement an application (bundle) that deals with SINE sensor changes in a further way (as a graphical interface for example).
Diagram 3.3 shows the *sine* bundle structure. For more complete information about the classes, refer to the API documentation. We see that there are four packages.

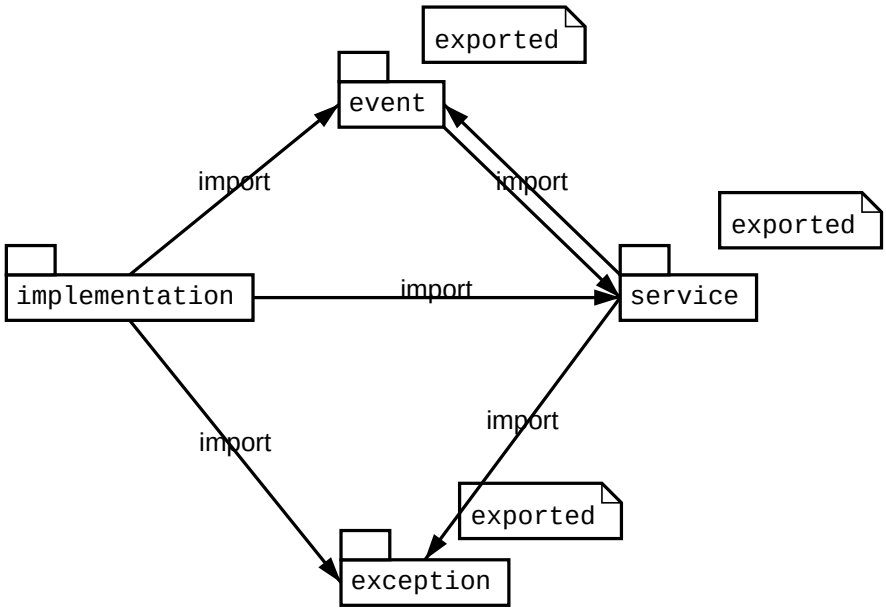

Figure 3.3: sine bundle package structure.

The *exception* package has been explained earlier. A new package *event* has been added. As the name implies, this package includes the events. The package is exported because the listeners (the sine clients) must know about the event. A said earlier, his makes the package not updatable. As the *SineEvent* transports information through a *SensorStatus*, which implementation can change, this is not a problem. We choose to add an event specific package to isolate the service from what happens *after* the service (the event firing-listening process happens after we get a reference to the *sine* service). Figure 3.4 describes the *sine* bundle classes.

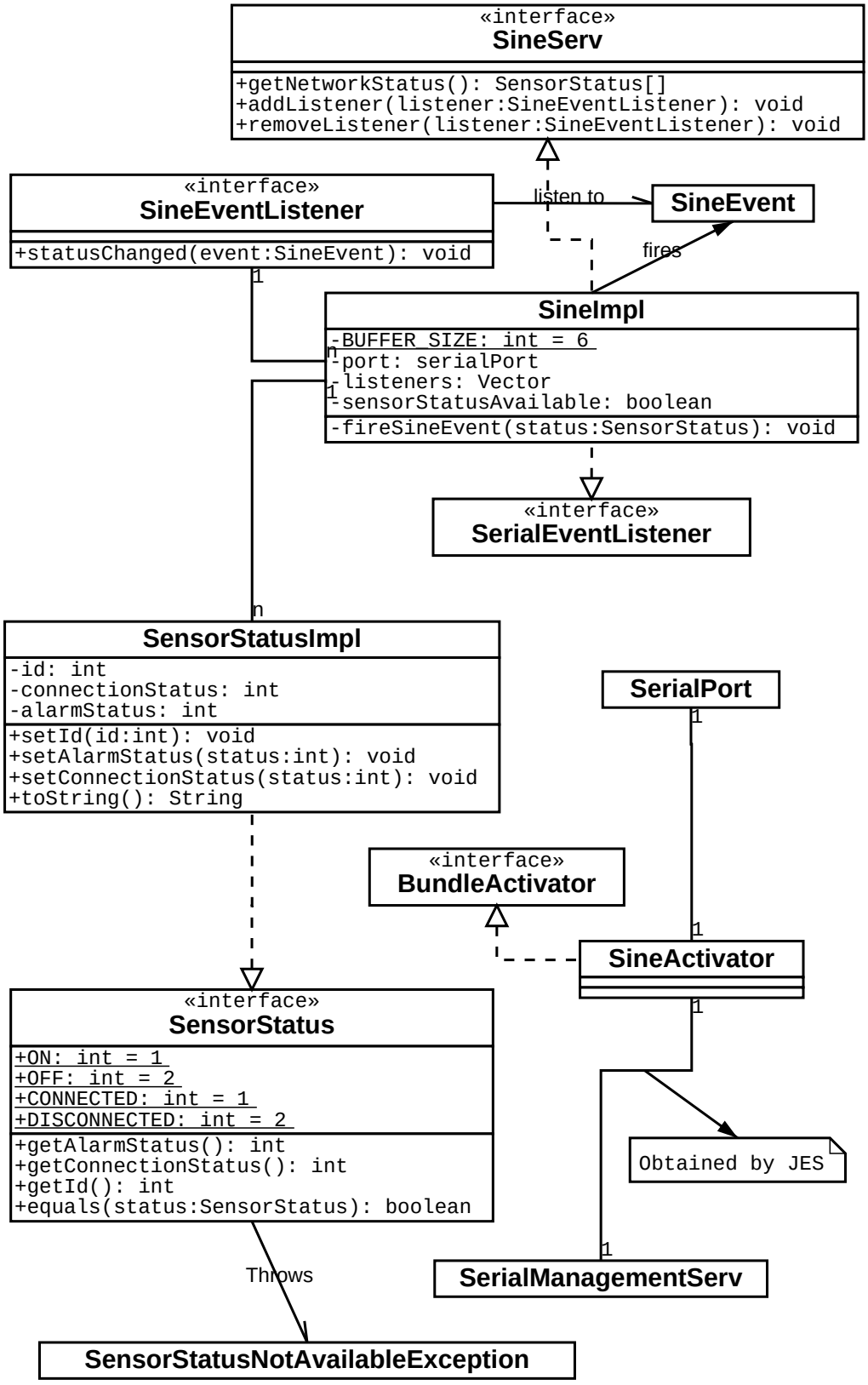

Figure 3.4: sine bundle classes.

3.5.1 Several controllers

Source detection

As said earlier, out of the lab, several controllers will probably be connected to the gateway. As the final goal is to have a system managing all the networks, at some point, a *single listener* will listen to all the sine controllers. This unique listener needs a way to identify the controller the information comes from and so, the different controllers must be identified separately.
The idea is to register a *SineService* for each controller with a parameter identifying the controller. The parameter is the serial port name the controller is connected to. In this way, further bundles, using the *SineService*, can identify which controller they are listening to by looking at the parameter.

New controller installation

Another requirement is to allow the installation of new controllers while the system is already running. This is possible using the update facility of the OSGi framework. The *SineActivator* class has a member containing the names of all the serial ports controllers are connected to. The activator registers a service for each of the controllers with a *description* parameter representing the port name. Having this variable (declared "final" in the code), adding a controller or changing a serial port is just a matter of changing the variable and updating the bundle. The following code snippet is extracted from the *SineActivator* class:

```
...
private final String[] portsToStart = {‘‘/dev/ttyS0’’};
...
for(int i = 0; i < portsToStart.length; i++) {
   ...
   // gets the SerialPort ports[i] from
   // the serialManagement service
   ...
   SineImpl sineImpl = new SineImpl(ports[i]);
   ...
   Properties props = new Properties();
   props.put("serial_port",portsToStart[i]);
   ctxt.registerService("driver.sine.service.SineServ",
                        sineImpl,props);
   ...
}
```

We keep in mind that updating a bundle for adding hardware doesn't look so

nice, but it's the only solution to avoid description files kept outside the architecture. As adding hardware shouldn't occur frequently, the solution is acceptable.

3.6 Conclusion

In this chapter, we have developed two bundles.

1. **serialManagement**. The purpose of this bundle is double:
   (a) Make the access to the serial port easier by providing a direct request to an already allocated port.
   (b) Prevent conflicts against the ports.
2. **sine**. The purpose of this bundle is to provide an event-driven access to the SINE network. The bundle reads the information received, compares it to the previously received. If a change occurs, it notifies the registered *SineEventListener*

Chapter 4

X10 Network

X10 is the name of the technology used by the devices to communicate with each others. In this chapter, we use the term *X10 network* for the whole set of devices forming a sensor/actuator network. The X10 network couldn't be integrated as wanted in the gateway because of a hardware architecture problem. This will be explained later. As, under certain circumstances, the X10 network could be used, we have chosen to develop the bundle in this paper anyway. The X10 products we had for testing, come from the *Marmitek* company. Therefore, the product names provided later refer to this company products and not to any X10 standards.

4.1 Network description

The X10 network is composed of modules, sensors or actuators (or both at once), to be plugged directly in a power supply wall plug. Each module has an address defined by a letter from A to P called the house code, and a number from 1 to 16 called the device code. Using the combination of those two codes, it becomes possible to address (to plug) up to 256 devices on one power-supply network. The X10 devices communicate with each others by sending short impulses sequences on the power supply lines[10]. Each transmission starts with the destination address. Each sensor reads the signal, and if its address is the destination address, it applies the command coded in the rest of the transmission.

4.1.1 Logic of the X10 network

Several ways are possible to provide a logic (e.g. switch light on when somebody is detected):

1. **Device build-in logic**. Some of the X10 sensors have a build-in capability to send a signal to a specified address when detecting changes. For example, the motion sensor (SM10) can send a signal to a pre-configured address when on both rising and falling edge of a detection.

2. **Controller logic**. A special device called the *controller* (SC2700) can be plugged on the power-supply network like any other device. The controller reads every signal sent on the network and sends signals to provide a logic. The controller must be pre-configured through a basic keyboard-screen interface.

3. **PC interface**. Another device (CM11) called the *PC interface* can be plugged on the power-supply network.[11] This device can be connected to a PC trough a serial link. This device has two functions:

   (a) Whenever it reads a signal on the power-supply network, it forwards it to the PC through the serial link.

   (b) A special *buffer* can be configured to contain some logic.

4.1.2 Computer usage

We must keep in mind that the finality of the job is to be able to provide some logic in the whole house, using different sensors and actuators, that can communicate with each others whatever the network backbone they use. What happens if we use a network specific controller? We will surely *miss* some informations. For example, let's say the X10 controller contains a basic logic that switch a light on when the motion detector goes off. A motion is detected, the light is switched on, but first maybe some other lights should be switched on or maybe some other events should take place if the light is switched on! We see that the only possible way to provide a home-wide control is to concentrate the whole logic in a unique entity. This entity is of course the gateway in our case.

4.2 Protocol

A protocol for coding the commands and addresses with impulses is available, but we are not interested in this since the only interaction with the X10 devices we have, is through the PC interface. We will therefore only look at the protocol used to communicate with the PC interface over the serial line.
It must be noted that the only available protocol description that could be found comes from the Marmitek company web site as a Microsoft Word© document. This protocol description is not complete. The protocol contains two kinds of transmission codes, *simple* and *extended* but only simple transmissions are described. Even for the simple transmissions, some reverse engineering has been applied on a Windows application from the Marmitek company to fully understand the protocol. [12]
As the protocol description is not complete and the X10 couldn't be integrated as wanted, we will not give a detailed description of the protocol.

4.3 Implementation

The implementation consists in two bundles. A first bundle, *X10Serial*, reads the serial port and extract information concerning the type of message sent by the PC interface. The *X10Serial* bundle forwards the information received in a event-driven way by firing *X10SerialEvent*. Another bundle, *X10RMIServer* listen to the *X10SerialEvents*, extract more information and fires *X10Events* which contains a more structured and detailed information. An RMI server bundle has been developed in a first time for the purpose of a demonstration of the remote accessibility of the X10 network information [13], [14].
Figure 4.1 describes the *X10Serial* bundle and the connection to the *X10RMIServer* bundle.

Diagram 4.2 describes the *X10RMIServer* bundle. The details about RMI (UnicastRemoteObject extension, RemoteExceptions) are not shown since it's not of primary interest here.

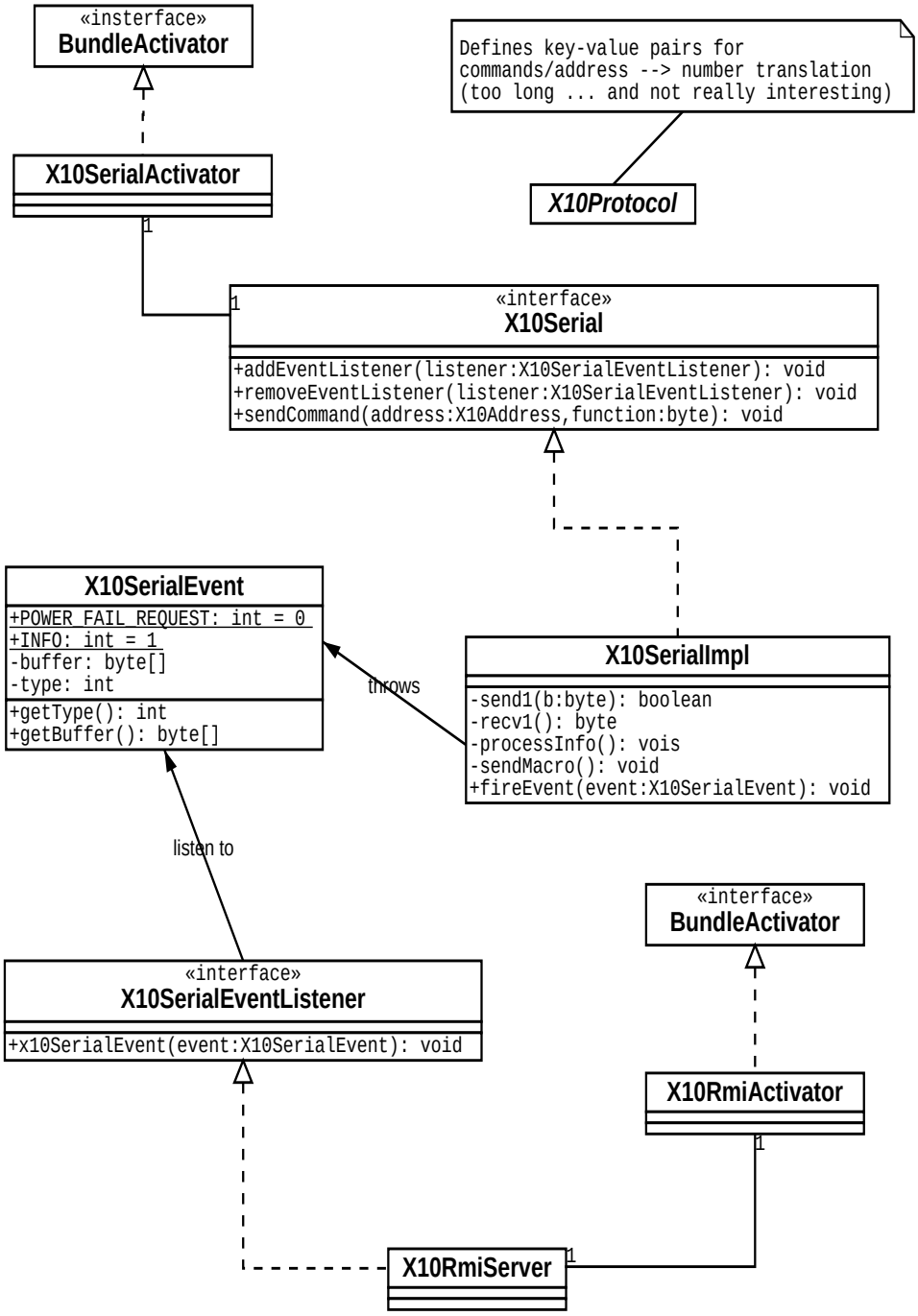

Figure 4.1: x10 bundle classes description.

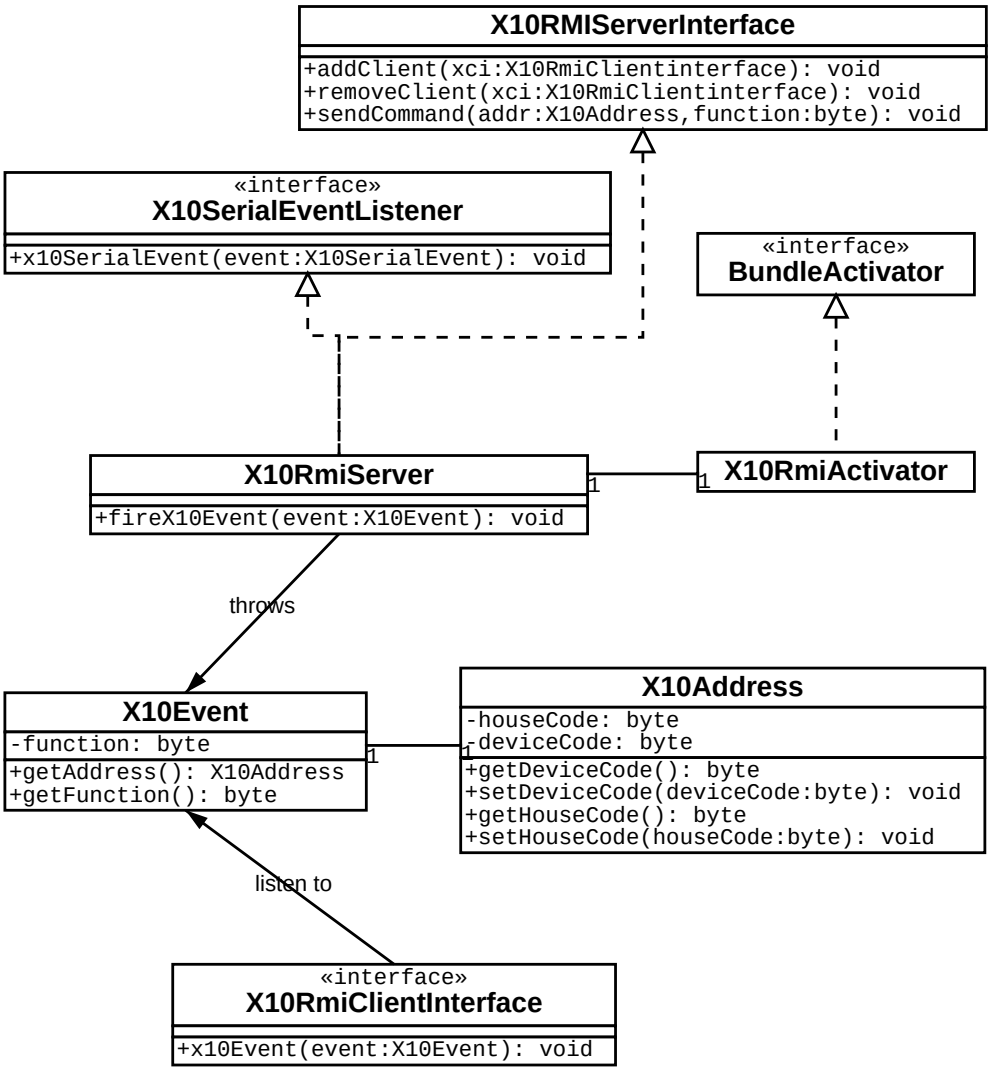

Figure 4.2: X10 RMI server bundle classesdescription.

The link between the x10 RMI server and the RMI clients has been built following the normal event interface. The *addClient* and *removeClient* plays the same role as the standard *addListener* and *removeListener* methods.
A RMI client has been developed for the demonstration.

4.4 Usability of the X10 network

So far, it looks like everything is perfect with the X10 network. Why then has this network been abandoned? The problem comes from the architecture of the PC interface. As we said, whenever the PC interface reads a signal on the power supply line, it forwards the information through the serial link.

4.4.1 How did we find the defect

We connected a motion sensor on the network. This standard X10 device sends a signal when detecting movement but also when it finishes detecting it. We say "finished it", because the signal comes right after the movement. This is the first part. The other part is that any transmission occurring when the PC interface wants to send data to the PC is canceled. Which means that if people are speaking in front of the motion sensor, the PC interface wants to continually send information through the serial link, canceling any other running transmission. In this situation, it is impossible to switch a light on since we would need a full transition to the PC interface.

4.5 Conclusion

We have chosen not to use the X10 network any further. This decision is based on the severe problem encountered while communicating with the PC interface. A solution could maybe have been found if the protocol description had been complete. The OSGi architecture allows to integrate easily the X10 network if a solution is found in the future.

Chapter 5

Linet Network

5.1 Network description

The Linet [15] (from Light Control Network) network consists in a controller(like a PC card) and nodes [16]. The network name is a bit confusing since it is not dedicated to light control. This name has just an historical reason. The nodes have entry ports where sensors and actuators are connected. Those sensors and actuators can be of any kind, as far as they provide an electrical signal that can be read by the nodes. A single controller can control up to 200 nodes. One of the advantages of the Linet network is that several nodes can be connected to the controller using a single pair of wires. The controller maintains a table of rules defining the interactions between sensors and actuators. This is another advantage since it avoids a voluminous logic wiring, especially if there are many devices and conditions. The rules can therefore be very complex and cost very few efforts. The wiring is even more reduced since the nodes can provide power supply to the sensors/actuators connected to them. The nodes are separated within groups[17]. Several kinds of groups are available.
The two following tables shortly define them:

Basic groups		
group type	**description**	**application example**
Toggle	Each node within a toggle group has a binary state, which is common to all nodes within the group.The state is inverted when there is a rising edge detected on a switch input on any of the nodes within the group.	lighting group
Analog Input	Each node contains an integrated 12-bit A/D-converter with internal, 1.25V reference. When in use, the converter performs the conversion and feeds the resulting 12-bit figure to the network.	temperature sensors
Dimmer	Each node within a dimmer group outputs a state, which is common to all nodes within the group. An input switch connected to any of the nodes may be used to control the output.	lighting dimmer groups
Data group	Each node is capable to receive and send binary data at constant rate. Data exchange with the network controller can be done with 8, 12 and 16 bits words. Data exchange group consists in two nodes exchanging data between themselves.	communication units

Table 5.1: Basic Linet devices groups.

Additional groups		
group type	**description**	**application example**
I/O-node	Toggle node is an individual I/O-device, connected to one input and one output.	lighting master switch
Lamp	Lamp nodes are used to control a load. They may be used as slave to dimmer or toggle groups.	principal lighting systems
Call	A call node puts its output to ON and generates a "call at gid" message on the controller. Once it receives the acknowledgment, it turns its output OFF.	alarm systems, hostess call systems
AD/state	The AD/state node is an analog input switch node.	combined humidity and temperature node
Control	control groups are used to replace thermostats when controlling temperature (or other magnitude). An input value coming from other sensors, applied to the control, is converted to centigrades.	heat control systems.

Table 5.2: Additional Linet devices groups.

Delay groups are also available. They allow to introduce a delay between sensor reaction and actuator effect.
The Linet controller and the gateway are connected through a standard Ethernet connection. This connection allows both network monitoring and network configuration. The same information is also available through a serial link, but is more difficult to implement and not easy to connect in a house (distance matters).

5.2 Protocol

The controller accepts IP/UDP connections and only transmits data on demand[18]. The controller receives and sends packets in binary format. Data entities are 8 or 16 bits unsigned integers. The same data structure is used to send and receive data to and from the controller, only one bit changes to indicate if the packet is a request or an answer, or to indicate if we want to update a node or read it. A transmission consists in a header that gives global information and in 200 *network status* packets describing each node of the network. Transmissions with less *network status* packets are possible to increase the available response rate of the controller, but, as in our case, only one computer, the gateway, is connected to the controller, it has little interest.

5.2.1 Packet description

Every transmitted packet starts with the following common header:

byte offset	**bytes**	**field**
0	1	protocol version
1	1	packet type, see below
2	2	flags, reserved

Table 5.3: Linet packet header.

The defined packet types (second byte) are defined in Table 5.4:

0	status request
1	status response
2	structure request
3	structure response

Table 5.4: Linet packet types.

The provided protocol description doesn't contain any further information about

the *structure* request and response.
The *network status* packets structure is defined in Table 5.5

byte offset	**bytes**	**field**
0	1	group type, see below
1	1	flags
2	2	group value

Table 5.5: Network status packet.

The flags field contains only one flag: if bit 0 is on, the request wants to change the group state to the value contained in the packet, if the flag bit 0 is off, the value field is ignored.
The group types are the ones defined earlier:

type code	**group type**	**value type**
0	NONE	none
1	TOGGLE	on/off
2	DIMMER	0=off, 1=on, 2=step, 3=step down
3	IOGROUP	1 bit I/O
4	XLAMP	1 bit I/O
5	LMON	1 bit I/O
6	XDELAY	16 bit delay value in milli sec
7	WCALL	waiter call
8	DATA EXCHANGE	not usable
9	DATA8	8 bit data value
10	DATA12	12 bit data value
11	DATA16	16 bit data value
12	AD/STATE	AD value (12 bit)
13	CONTROL	AD value (12 bit)

Table 5.6: Linet Group types.

5.3 Implementation

The implementation consists in a new bundle *linet*. The goal of the bundle is to provide both an event-driven access to the Linet network changes and a basic way to change the value of the nodes. The logic part (groups and which sensor is what) is kept outside the bundle. Some security is also provided. For example, an exception is thrown if one attempts to put a 16 bit value to a boolean valued node and multiple "client" bundles can modify the same controller (see below). Several

classes have been provided by the Linet company. The classes contain utilities to read and write UDP packets and more specifically, UDP packets to the Linet controller. With some add-ons and modifications, those classes could be used in the *Linet* bundle. To allow updates of the classes, some interfaces have been added. The same package structure as before has been used, it contains a *service*, an *implementation*, an *event* and an *exception* package.

5.3.1 Global functioning

We have seen that the controller only transfers data after a request is received. That means that to detect changes within the Linet network, we need to poll the controller every x seconds, and so is done. After receiving the response from the controller, the status of each node is checked and compared to the previous state. If any change appears, an event is fired containing the new node value. The *Linet* interface proposes therefore an *addListener* method to register a *LinetEventListener*. The *Linet* interface also proposes a *changeState* method that allows to modify any node of the network. The full network status (all the nodes) and one specific node status can also be obtained. The *Linet* interface is implemented by the *LinetImpl* class of the *implementation* package.

5.3.2 Classes

A first set of classes manages the low-level UDP connection. This set contains the *UDPListener*, *UDPPacket* and *UDPWriter* classes.
A second set of classes is used to transmit data types understood by the Linet controller. This set contains the *LinetPacket*, *LinetPacketEntry* and *LinetPacketHeader* classes. The *LinetPacketEntry* class is the data structure describing a single node. The *LinetPacket*, which contains an array of *LinetPacketEntry* is a data structure describing one whole Linet network. The *LinetPacketHeader*, which is also contained in the *LinetPacket* class, is used to mark a transmission as a request or an answer. Each of the Linet* classes provides a *getRawData* returning byte array representing the data structure that will be transmitted through the UDP connection via the UDPWriter *write*{*byte[] data*} method.
So far, the classes were those provided by the company. Some modifications have been brought to them. The classes defined later have been created from scratch for the integration in the OSGi architecture.
Whenever a change to one of the Linet node occurs, a *LinetEvent* is fired. This class carries the number and the new value of the altered node.
An interface *LinetNode* has been added to represent the *LinetPacketEntry* class. As we have seen, the *LinetPacketEntry* class has been created **before** the interface. This is certainly not the proper way of doing things in a general environment. But, in this case, the interface only exists to allow the *LinetPacketEntry* **not** to be exported in the OSGi framework. In this way, we are able to update the class.

The two UML class diagrams 5.1 and 5.2 describe the implementation of the bundle.(The two diagrams should be joined, but we chose to separate them for readability reasons)

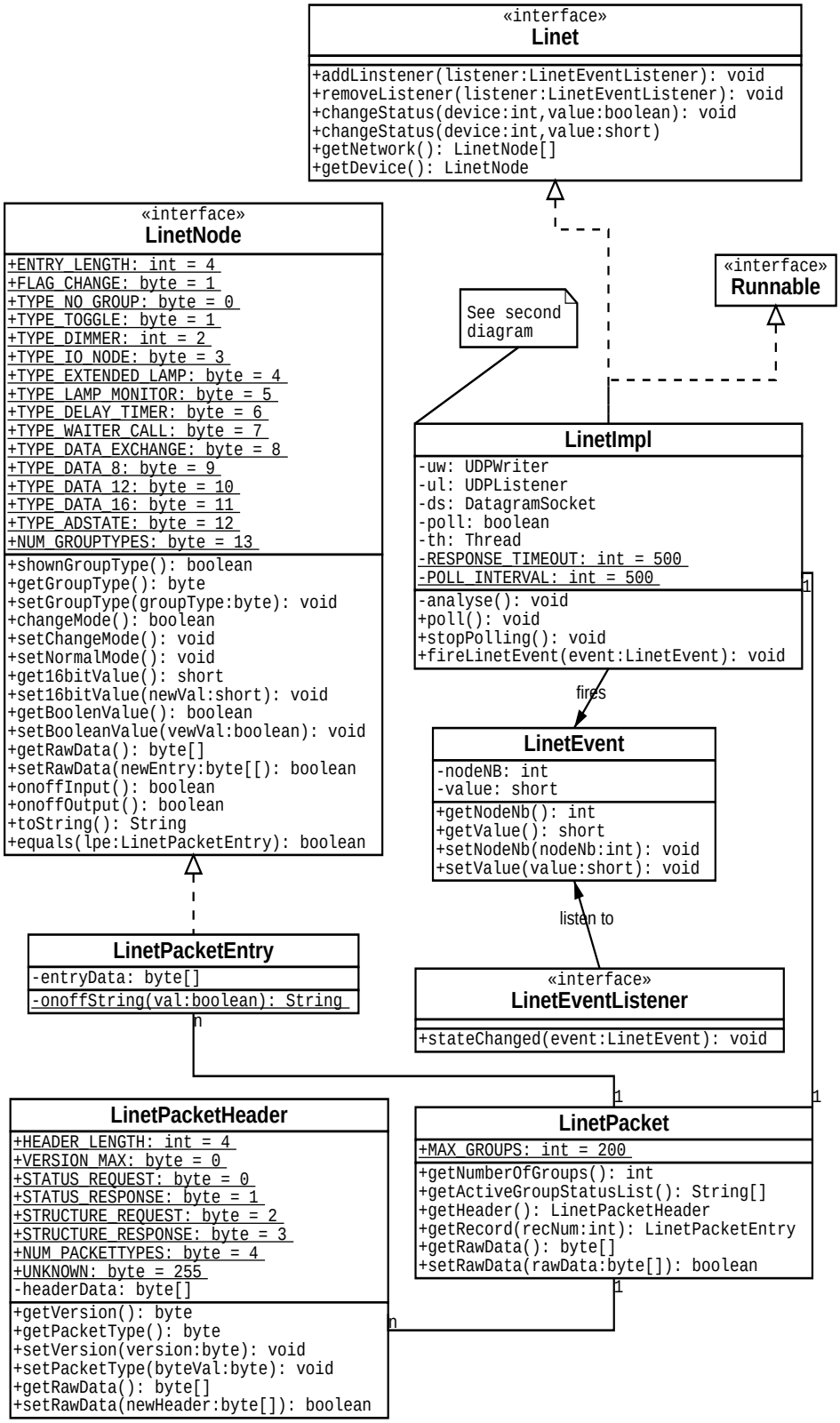

Figure 5.1: *linet* bundle classes.

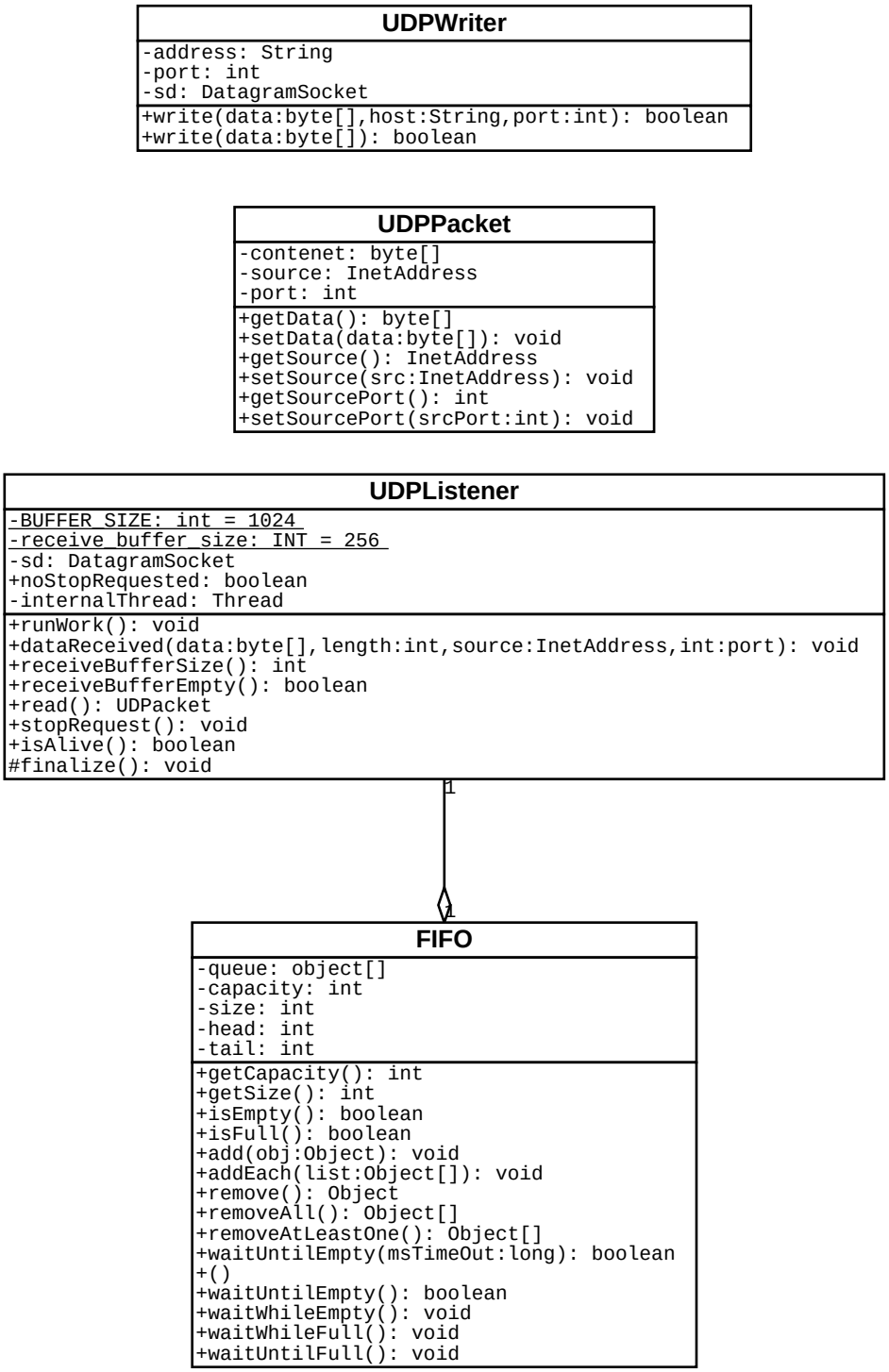

Figure 5.2: *linet* bundle UDP related classes.

The Linet related exceptions, *LinetDeviceException, NotOnOffDeviceException, NotValueDeviceException* are not described in a diagram for evident place reasons. Those exceptions provide some security concerning assignments of value to the devices.

5.3.3 Multiple clients

What happens when several "clients" send information to the same controller? Due to the mechanism used to send information to it, no conflict will arise. The changes do not occur instantaneously after the change has been requested, it will occur at the next poll (see protocol description). When analyzing the response to this poll, the bundle will find the change to the specified device and will notify the registered *LinetEventListener.*

5.3.4 Several controllers

As a single Linet controller can control up to 200 sensors or actuators, it is not probable that we would need more than one controller. Anyway, with little effort, the solution used for the *sine* bundle can be applied here. For reminding, the idea is to add a string-array member to the *LinetActivator* class. The string-array contains a descriptions for each controller we want to connect. In the case of the SINE controllers, the description was the name of the serial port the controller is connected to. In the case of the Linet controllers, the description is the network address and the port of the controller.

5.4 Conclusion

In this chapter, we have described a *linet* bundle. The purpose of this bundle is to provide both an event-driven access to the information provided by the Linet devices and a way to write information to the linet actuators. The bundle also allows to read the current available network information at any time.

Part II

Programmable Logic

Chapter 6

Introduction

6.1 Definition and requirements

The programmable logic system is a service (in the OSGi sense of the word) that provides the *automation* in the house. The logic system must be able to provide this automation on every network (Linet and Sine for the moment). This service maintains the logic between sensors and actuators. The system involves several requirements:

1. **Network independence**. As said in the work introduction, the system must control every network. But more than that, the user should be able to incorporate sensors and actuators in the logic without worrying about which network the devices are part of. For example, a sensor of the SINE network can provoke a light connected to the Linet network to be switched on.

2. **Updatability**. Two different cases must be treated.

   (a) *Logic.* The logic itself should be easily updated. New rules must be easily added and existing ones must be easily modified or erased. The house owner should be able to update the logic by himself, without any external help.

   (b) *Hardware.* It must be possible to add or replace components of any network and to incorporate them into the logic system. As hardware updates will occur more rarely than logic updates, and as the task involves work from external persons, this case is not as important as the first one and it is acceptable that deeper changes could be involved. Anyway, the whole program cannot be rebuilt if new hardware is added.

3. **Human friendliness**. This is related to the updatability. An intuitive interface must be provided to update the system more easily. Sensors and actuators of the networks should be easily identified. A unique identifier like those found in databases does not fit anymore and a “human friendly name”

must be associated with every device. For example, kitchen-sink-light is more intuitive than dev037.

4. **System update**. This update consists in the ability to update the system as a program, in order to add features or to make it more efficient.

6.2 Different parts of the system

6.2.1 Several levels

The system will be divided in several levels. Each level provides a facility used by the next level. In the order of increasing abstraction, the facilities to implement are:

- **Network independence**. This level provides a *standard* way to access the different networks sensors and actuators through a unique access point. The level defines both a way to access and set sensors and actuators value but also a way to reference them. The *value-change* events coming from the driver level are forwarded in a standard way to the upper level. This level uses the driver bundles developed in the *Drivers* part of this work.

- **User-friendly names**. This level allows to use human friendly names to reference the sensors and actuators. The level also maintains more detailed information about each device and secure the access to their value.

- **The logic-management module**. At each change in the network sensors or actuators, this level will check the logic and apply any pre-programmed automatic change.

- **User interface**. This level consists in a user interface that allows to update the automation logic.

The diagram 6.1 shows the relationship between the different levels.
We choose to separate the tasks as much as possible. Of course, all those levels could be incorporated in one big bundle, but separating the tasks in different bundles allows a more efficient *system update* afterwards. Also, separating the whole task into different services allows more freedom to connect new bundles in order to extend the functionalities.
We put the network independence as the first level because it directly decreases the code needed by the following levels (the code to deal with each specific network is reduced to one unique code).
The following two levels are less obvious. By using the *user-friendly names* before the *logic-management* level, we impose this latter level to deal with device names readable by humans. Those names do not bring any advantage to the level, at least on the programming point of view. So why are the levels in this order? To answer this question, we must look a bit further in the logic implementation. The idea is to

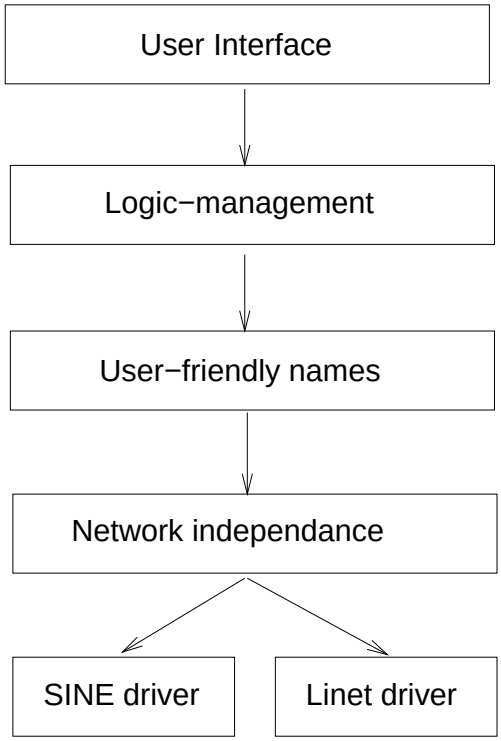

Figure 6.1: The different levels integration.

use XML files to define the logic. The logic-management reads rules defined in the XML file and, whenever any element of the networks changes, it checks the rules to find a suitable one. All of this will be explained in further details in chapter 9. To come back to the levels order choice, let's see what happens if the "user-friendly names" level comes up to the "logic-management" one. In this case, the XML file defining the logic is not human readable anymore since the device names it uses must be defined with network and device ID. Having the "logic-management" level as the third one allows to keep a friendly XML file. The chosen order also reduces the difficulty of dealing with different networks IDs.
The interface level should be as loosely coupled to the previous level as possible. To achieve this, the interface is chosen not to contain any responsibilities but "printing to the screen".
In this second part, each chapter will describe a level.

Chapter 7

Network independence

7.1 Goals

The goal of this layer is multiple:

1. Providing a uniform access to the value of the sensors and actuators.
2. Providing a uniform reference to the sensors and actuators themselves.

7.1.1 Uniform value access

We want to be able to access every device value of any kind of network in the same way. The access must be provided to *read*, *write* and *listen to events*. By *writing* we mean assigning a value to an actuator. By *reading* we mean asking directly the value of a sensor or of an actuator. This *reading* is only done during the initialization. *Event listening* is what drives the whole logic process. It consists in listening to changes in the sensors values. Reading and writing methods can be described by an interface. Standard event listening can be achieved through a common event describing every sensors changes.
Uniform value access involves a reduction of the information provided by the devices to a "less common divisor". This means for example that the device "presence" information provided by the SINE network is not accessible anymore, since this feature is not provided by the Linet network. The value assignment exceptions provided by the Linet driver service are not available as well, since they cannot be applied to the SINE network. However, back to the "presence" information of the SINE devices, it can be used by other bundles, in parallel to the standardization bundle. It is in fact the purpose of the OSGi architecture.
Diagram 7.1 illustrates an example of such a parallel architecture.

In this diagram, we see that a completely independent bundle is responsible for alerting the house-owner if a SINE device goes down. At the driver level, all the

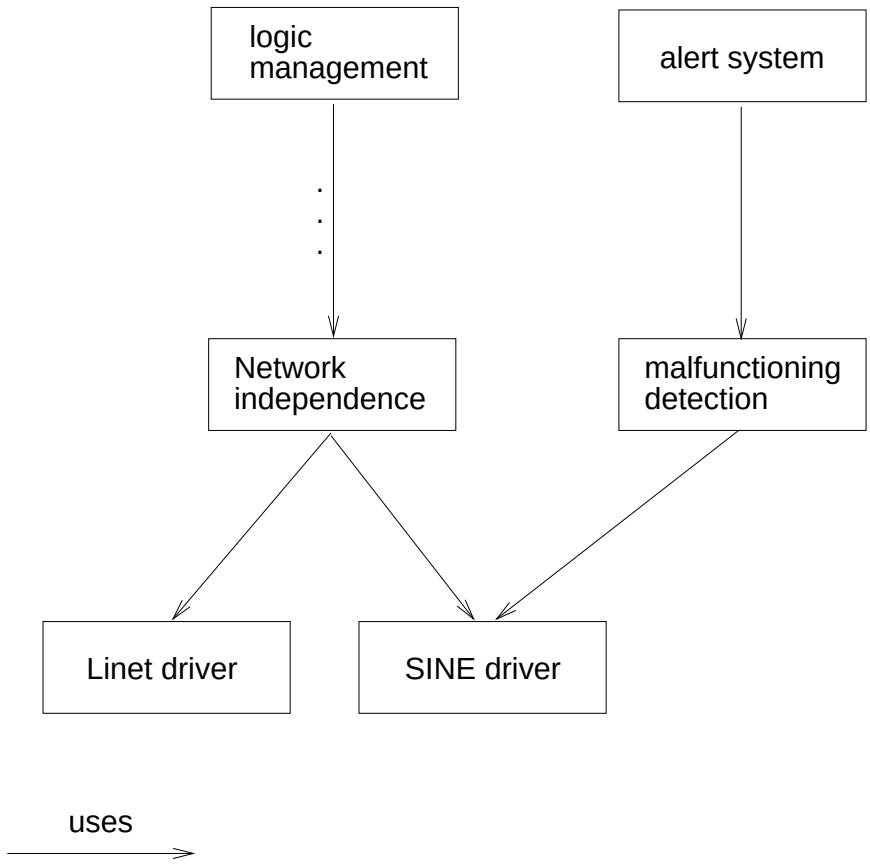

Figure 7.1: Using bundles in parallel to treat independently different information.

information is available. Upper the *Network independence* level, only the value of the devices is accessible. The *malfunction detection* bundle directly listens to the *SineEvents* which are fired at any device status changes, thus also when a device goes wrong. Using this kind of architecture, no information is lost and the system becomes more modular and easier to manage.

7.1.2 Uniform reference

Every device, sensor or actuator, can be accessed through the drivers developed earlier in this document. But so far, accessing a sensor from a SINE network is different from accessing a sensor from a Linet network. We want a standard, and unique access to each device. We have seen that one *driver* service is registered for each controller with a unique description. For example, a *SineService* is registered for every SINE controller with the serial port name the controller is connected to, as a description. Using this description; it is possible to reference uniquely any device from any network with the following information:

- network type.
- network description.
- a unique integer on that network.

Thus, for example, the second sensor from the SINE network "connected" to the first serial port can be accessed by:

network type	SINE
network description(serial port for SINE)	/dev/ttyS0
unique integer	2

Table 7.1: Device unique identification.

7.2 Implementation

This level is split into two sub-levels. A first sub-level consists in a *Network Standardizer* service that standardizes each network. A second sub-level provides a unique access-point to all the networks. One *Network Standardizer* service is provided for each *driver* service. A different implementation is required for every type of network but a unique service(in the OSGI definition) is provided. Each service *Network Standardizer* service is registered with the *network type* and *network description* seen above as properties. The second sub-level uses that information to dispatch the incoming value requests and assignments from the next level.
Figure 7.2 describes the sub-levels and the link to the driver level.

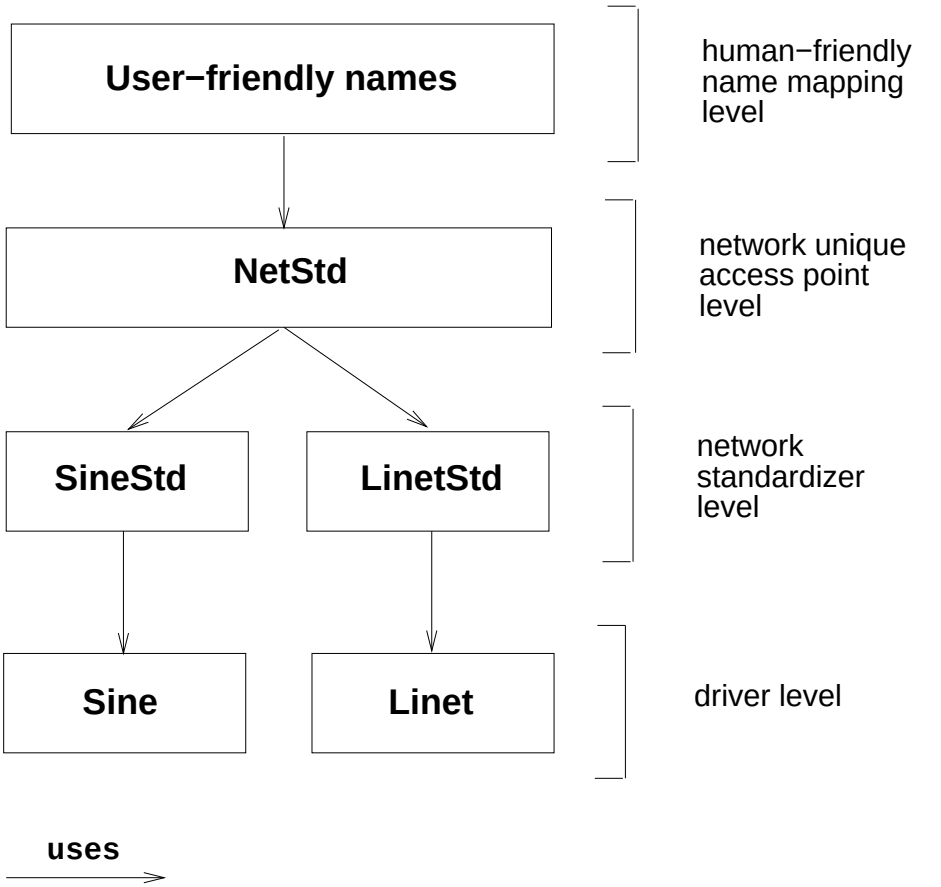

Figure 7.2: network-independence sub-levels architecture.

7.2.1 Structure of the sub-levels

The package structure of the first sub-level bundle slightly changes from the one we have seen before. The events are here of particular importance. The event package wonderfully takes place in the exported packages since events provide the standard interface to every sensor changes.
As it has been said earlier, a different implementation is needed for every type of network. Here, we will develop two bundles, one for the SINE network and one for the Linet network.
Figure 7.3 shows the package structure of the first sub-level. The structure of the second sub-level is the same as the one previously seen for the driver bundles.

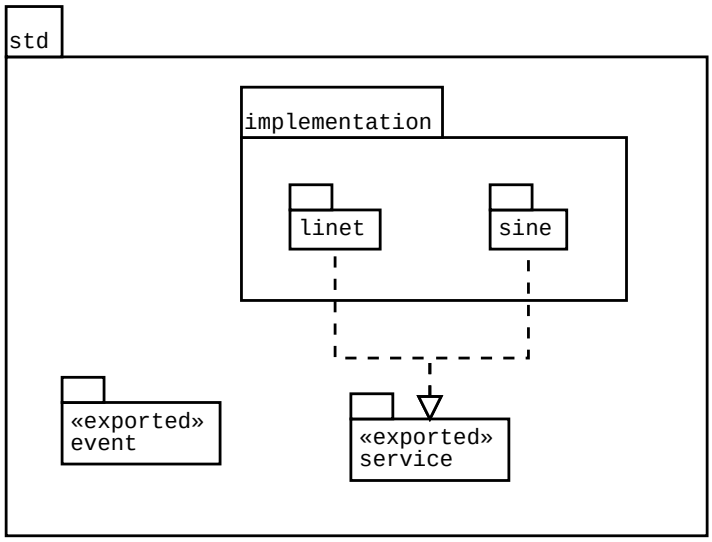

Figure 7.3: first sub-level package architecture.

7.2.2 First sub-level bundle

The first sub-level is composed of two bundles, one for the SINE network and another for the Linet network. The two bundles will be named *sineStd* and *linetStd*. Each of those two bundles contains a reference to a unique interface which defines the first sub-level service. As explained earlier, a service is registered for each network. For example, if two SINE controllers are connected to the first two serial ports, two services will be registered with a description parameter set to the serial port name. The bundle activators are responsible for starting the services. Using the previous example, the activator of the *sineStd* bundle will register two services, one with a description pair *net-description=/dev/ttys0* and another with a description pair *net-description=/dev/ttys1*.

Service

There are two kinds of accesses to the devices' value: a query/set access and an event-driven access.

Query access The query access is directly defined in the interface that describes the service. This interface, named *StandardNet* declares simple *get* and *set* methods that can be used to get and put a value from/to the sensors and actuators. Both *get* and *set* methods are defined twice, once for boolean devices and once for *short* (value-ranged) devices. Those two formats cover the kind of value used by every device of any network. The set methods must be declared "*synchronized*" in the code, to resolve concurrency.

The way the devices are accessed by the upper level depends o some preconfiguration. Without this configuration, it is not possible to know if a device is a On-Off device or a value-ranged device. It is therefore the responsibility of some upper level to ensure the right kind of value is assigned to any device.

Event access This kind of access is the most usual one. A single event *DeviceEvent* is used for every device's value change. This event contains the identification parameters of the device the value has changed, the old value and the new value. It is the responsibility of the listener to interpret the value. The interface mentioned before also defines the standard *addListener* and *removeListener* methods.
The complete *StandardNet* interface can be found in appendix.

7.2.3 Second sub-level bundle

The second sub-level gets a reference to all the *StandardNet* services registered on the framework. When the upper level wants to access a device, it does it by specifying the network type, the network description and the device number. Based on this information, the second sub-level forwards the request to the real device through the right service reference.

Service

The second sub-level bundle, which is named *homeStd*, register the *NetAccess* interface(service). This interface defines the same methods as the first sub-level, but the device is now referenced with the three parameters: *network type*, *network description* and *device number* instead of only the device number.

Device access The accesses to the device are the same as the ones found in the previous level. The only difference is the way the devices are referenced. The event *NetworkEvent* corresponds to the *DeviceEvent* from the previous sub-level. The *NetworkEvent* references the changed device by the three parameters exposed before. The listener interface *NetworkEventListener* corresponds to the *DeviceEventListener*. The two interfaces define the same *valueChanged* method, but the argument is now a *NetworkEvent* instead of a *DeviceEvent*.

7.2.4 Event notification

This subsection applies to both sub-levels.
There are two circumstances under which a device or a network event can be fired:

1. Whenever the lower level has fired a *NetworkEvent*.

2. When an upper level successfully assigns a value to a device. This is necessary for the next levels to stay up-to-date. Of course, the upper level that

changed the device value will be notified of the changes it just made and must deal with it. The only way to avoid this unnecessary notification would be to compare the identity of the listener and the value assigner. This is not possible since they will generally be different classes.

7.3 conclusion

In this chapter, a two bundles have been designed. Because of their close functioning, the two bundles are part of the same sub-level. The first sub-level bundle provides a common interface to access the different networks devices. The second sub-level bundle provides a unique access point to the devices of every network. With the small disadvantage of an unnecessary event notification, both bundles are designed so they can be used safely by several bundles.

Chapter 8

User friendliness

8.1 Goal

The goal of this level is triple:

1. Providing a human-friendly name for every sensors and actuators.
2. Providing some control over the way the value of the sensors and actuators are accessed.
3. Providing some description of the sensors and actuators to be used by the logic editor.

8.1.1 Human-friendly names

The aim is to allow access to any sensor with an intuitive name instead of a computer-friendly identifier. The names should be modifiable without too many efforts. More device names should also be possible to add.

8.1.2 Control

The goal is to check the validity of the value the upper level wants to assign or get to/from the sensors and actuators. Two kinds of error should be caught:

1. Wrong value type assignment or asking. For example, an upper level tries to assign a *short* value to an on-off device which can only take boolean values.
2. Actuator/Sensor confusion. This arises when an upper level tries to assign a value to a sensor.

8.1.3 Devices description

We want to have a more extended description than the name associated to any devices. This requirements come in view of the next level, where devices will be

incorporated into rules while the end-user will not necessarily remember the details of a sensor or actuator. The description should be close to the device-id/friendly-name pair for matters of management. This aspect is thus highly tied up with the first one.

8.2 XML implementation

8.2.1 Names and description

To associate a device id with a user-friendly name and a description, in such a way that devices can be added or modified, the first thing that comes to mind is a database. In this case, we found that the usage of a database had two real defects. First the database doesn't fit in the OSGi framework, which means that some special bundles or other external tools (web applications for example) are needed to administrate the database. It should be better if the whole system was kept inside the OSGi architecture. Secondly, a database engine as MySQL requires a lot of resources just to run, even before managing the single table we need [19]. For those two reasons, we found the XML technology more appropriate [20], [21]. To take again a previous example, the second sensor of the SINE network connected to the serial port */dev/ttyS0* can be defined by the following XML code snippet:

```
<device>
    <id network_type=''SINE''
        network_description=''/dev/ttyS0''
        device_number=''2''/>
    <name> kitchen_fire_alarm </name>
    <description>
        The fire sensor fixed in the kitchen ceiling
    </description>
</device>
```

As there are not to many sensors and actuators, the devices descriptions can be kept in memory after it has been parsed. A special bundle must be written to allow the XML file modification and the automatic update of the friendly-naming bundle. Once again, we can take profit of the parallel bundle development of the OSGi architecture.
Keeping an XML file on the gateway raises one question: *What happens if a crash-disk occurs?* Several solutions exist. A first could be to simply have a backup disk on the gateway. However, the gateway should better be a "thin client" if possible (both from place and price requirements). So this solution has an obvious drawback. Another solution could be to always keep the last file version on a separate server running *somewhere*. No disk is needed anymore and the gateway (by definition) has access to Internet servers. That solution is however not perfect either: it could happen that the local file is corrupted and the INTERNET connection is down. This is the worse case because the "friendly name" level will not be able to

initialize and, without this level, the upper level will be stuck with friendly names not linked with the devices. In this case, the whole system would go down. This situation is as bad as rare and we consider that the probability of it to happen is extremely low. To summarize, no perfect solution has been found but the later one seems more than acceptable. If, one day, thousand devices came to be connected, the solution of the database should be adopted.

8.2.2 Control

The control can be realized easily if, in the previous XML device description, we add the two fields *type* and *valued*. The first one can take the values *sensor* and *actuator*, the second *boolean* and *ranged*. In case the *valued* is set to *ranged*, the upper and lower limits as well as the increment can be provided.

8.2.3 The XML file structure

The following XML schema [22] defines the configuration file structure:
Using this schema and a parser providing the *validation* feature like Xerces2 from

Apache [23], we can validate the XML file before applying its content.

8.3 Bundle implementation

The bundle of this layer is called *naming*.

8.3.1 Interface

The bundle provides a service defined by the interface *DeviceNaming*.
A first set of methods defined in this interface are used by the logic editor. Those methods provides the list of specific devices, for example, the list of all sensors, the list of the on-off actuators, ... This set of methods also defines request methods, as *isDeviceOnOff(String deviceName)* and the description information request *getDeviceInfo(String device)*.
A second set of methods is provide to set and get the values of the devices. Those methods looks as the one of the previous level, except for:

1. They take a String parameter to reference the device.

2. They throw exception if a wrong kind of value is assigned to a device.

These methods must be declared *synchronized* in the implementation to resolve concurrency. A third set of methods concern the event listener registering and unregistering. The two standard methods *addListener* and *removeListener* are therefore defined.

8.3.2 Exceptions

To provide a control over the kind of values assigned or asked, the value assignment and request methods throw exceptions. Three exceptions are defined:

1. **NotBooleanDeviceException**. This exception is thrown when a *short* value is asked or assigned to a on-off device.
2. **NotRangedValueDeviceException**. This exception is thrown when a *boolean* value is asked or assigned to a ranged-value device.
3. **NotActuatorException**. This exception is thrown when a value is assigned to a sensor(which is read-only by definition).

Those three exceptions just contains the "faulty" device name. As they have the same structure, they all extend a common exception *DeviceException.*

8.3.3 Events

Events are thrown to notify the next level that a device value has changed. An event can be fired under to same circumstances than for the previous level:

1. Whenever the lower level has fired an *NetworkEvent.*
2. When an upper level successfully assigns a value to a device. The same remark than for the previous level applies here. See page 59 for further details)

The events fired to the upper level, called *DeviceEvent,* are the same as the ones sent to this level, except that they contain a *String* to reference the device. The event listener interface is the same as the one of the previous level, it defines a *valueChanged* methods with a reference argument to a *DeviceEvent* instead of a *NetEvent.*

8.4 Conclusion

In this chapter, we have designed a bundle that fulfill two functions: to provide a human-friendly naming of the devices and to secure the access to the devices value.
With the small disadvantage of an unnecessary event notification, the bundle is designed so that several "client" bundles can securely use its service.

Chapter 9

Logic-management level

This level is separated into two services. A first service applies the logic to the networks. A second service provides utilities to edit the logic.

9.1 Logic representation

Before entering into the details of the two services, it is first useful to see how the logic will be represented. We chose to use the XML technology to keep the logic. An XML file stored on the gateway contains a set of rules which describe the automation throughout the different networks.

9.1.1 Rules

The rules consist in pairs input-output. The input represents the network state conditions that will result in the changes described by the output. The XML file contains therefore a sequence of rules, written like follows:

```
<rule>
    <in>
        conditions come here
    </in>
    <out>
        changes come here
    </out>
</rule>
```

The state conditions consist in conditions over the status of sensors and actuators. They are expressed with comparison operators as =, <, >, ... Those state conditions can be joined by logicaloperators *AND* and *OR* and can be negated with the sign "*!*".
The changes are XML simple (in the XML meaning) elements containing the new

value of the actuators.
For example, the following XML code snippet defines the following automatic change: *When the front door motion sensor detects a movement and the day light sensor is lower than 10, switch the front door light on.*

```
<rule>
    <in>
        <and>
            <condition sensor_name=''front-door-motion-sensor''
                       state=''on''/>
            <condition sensor_name=''day-light-sensor''
                       state=''<10''/>
        </and>
    </in>

    <out>
        <change actuator_name=''front-door-light''
                new_state=''on''/>
    </out>
</rule>
```

Delays can also be defined in the XML files as parameter of the *change* elements. For example, if in the previous example we wanted to switch the light on after 3 seconds, the output can be rewritten as follows:

```
    ...
    <out>
        <change actuator_name=''front-door-light''
                new_state=''on''
                delay=''3''/>
    </out>
    ...
```

The new state can be defined both as an absolute value or as a value calculated from the previous one. In the latter case, the *new_state* is more like a simple rule. An example of this could be a dimmer emulation, where the value of an actuator can be increased or decreased by a unit. The XML file contains human-friendly device names since it uses the previously developed level to translate them.
Having the logic defined in an XML file, a service to update the file and a service to apply the rules, updating the rules is a simple matter of updating the XML file and restarting the "rules applicator" bundle. In these circumstances, dividing rules application and update into two services is the best tactic.

9.2 Rules applicator bundle

This bundle is called *RuleApplicator.* It provides the logic of the system. The bundle doesn't provide any service.

9.2.1 Initialization

The bundle initialization follows these steps:

1. It parses the XML file that describes the rules.
2. It builds an internal representation of the rules and build a hash-table containing each device found in the condition part of a rule. The hash-table entry key is the device name, and a structure containing the device value (not known so far) and a vector of rules reference in which the device appears in the condition part.
3. It asks the previous layer for the current value of each device of the hash-table.
4. It registers itself as a listener to the previous level.
5. It checks the rules to see if any full condition part is met. In this case, it operates the changes described in the rule. If any of the device found in the *out* part of the rule is also part of the *in* part of a rule, this step is done again.

The goal of keeping a hash-table with the device name and value allows to minimize the number of device value requests to the previous level. The value of the devices are only asked at initialization, since everything after is done through event notifications. When an event is received, the rules can be checked without asking the device values to the previous level. Another advantage of having a hash-table is to fasten the search for changed rules. If a device referenced in an event matched a key from the hash-table, we can directly know which rules must be checked.

9.2.2 Run-time operations

After it is initialized, the bundle is waked up by the events fired by the lower level. The hash-table is checked for any key matching with the device the value has changed. If such a match is found, every rule the device is part of is checked. If a rule condition is entirely matched, the changes listed in the rules are applied by sending *set* requests to the lower level. The newly changed devices are compared to the hash-table keys and if a match is found, the operation is repeated with the new device. This procedure allows a change to fire another change and so on, which makes the rules construction easier.

9.2.3 Rules representation

The rules are separated into two parts, one corresponding to the *in* part and the other to the *out*.

out part

The out part is quite easy to implement, since it can be represented as a list of structures containing the name of the device and the new value or a reference to a simple rule to apply (see dimmer example above).

In part

The condition part of the rules can be represented as a tree. The nodes of the tree can be logical operators or device value comparison operators. The leaves are either the device value to check or the values to compare the device values with.
Figure 9.1 shows the tree corresponding to the previous example: *When the front door motion sensor detects a movement and the day light sensor is lower than 10, switch the front door light on.* Once we have built the tree structure, the full

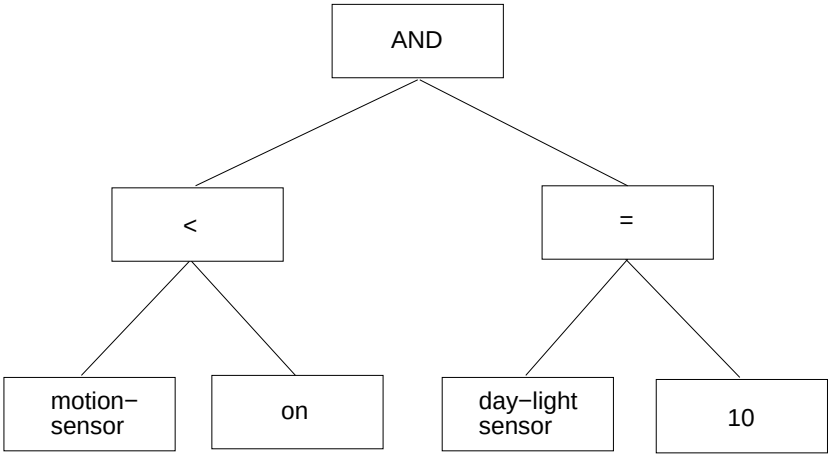

Figure 9.1: The tree representation of the rule.

condition of a rule is easily evaluated to true or false.

9.3 Rules updater bundle

This bundle is called *ruleUpdator*. It doesn't register any service either since it contains its own interface. This bundle first set of methods defined in the *Device-Naming* service. Gathering information about the devices, an interface can be built

so that constructing rules is just a matter of click-an-drag components, logical operators and comparators. The principal idea is to provide an RMI server interface, so that the user-interface can run anywhere.

9.4 Logic updates

The gestion of the logic updates is solved by the built-in OSGi framework facility. In the *BundleContext* interface, the OSGi framework provides methods to get a reference to another installed bundle. This reference consists in a *Bundle* interface. The *Bundle* interface, on its side, provides methods for updating the bundle it refers to.
When the *rule-updator* bundle has committed changes, it gets a reference to the *rule-applicator* bundle and updates it. The *rule-applicator* will then stop and start again, reading the new version of the XML file.
The logic update can be done at run-time and requires only a short interruption of the logic management (the time necessary to stop, start and initialize the bundle).

9.5 Conclusion

In this chapter, we have designed a bundle that applies the logic to the sensors and actuators. The built-in blocks of the logic are rules. Thoe rules are composed of two parts: a first one defines the conditions to trigger the value changes to the devices; the second defines the changes to apply. The rules are easily represented using the XML technology. Using the OSGi update facility, modifications to the logic can be done at run-time, with a minimal interruption of service.

Chapter 10

Conclusion

10.1 Development

The aim of this work was to integrate different networks of sensors and actuators and to develop an architecture for a programmable logic system that uses those networks. It was part of a larger project consisting in building a system that could help disabled people to live comfortably and safely in their own house.
Drivers for the different networks and the basis of a programmable house-wide logic were developed.

10.1.1 Drivers

An analysis has been made for each of the three networks which integration in the system was planned: SINE, X10 and Linet.
The SINE and the Linet networks have been successfully integrated. For each of them, we have developed a driver that gives access to the entire information provided.
The X10 network hasn't been integrated as wanted because of hardware shortcomings. A driver has been developed, but it's usability is limited to X10 networks involving no sensor that can be triggered repeatedly. Should such a device be integrated, the access to the network could be blocked as long as the device sends signals. Therefore, we have chosen not to use the X10 network any further. Even-driven access to the devices state changes were developed to make communication between bundles clean and economic and to reduce bundle coupling.
In order to facilitate future updates, we have used the rather new OSGi architecture.

10.1.2 Programmable logic system

A programmable logic system has been designed. Its architecture uses several levels and provides a great flexibility in terms of updates and expansions. Indeed, using the facilities provided by the OSGi framework, the implementation of each level can be updated independently. Expansions to the system are also easy since

each layer of the system has been designed so that several bundles can use them concurrently.
A level of the architecture ensures the networks independence. It allows to use a common interface to access devices from any network.
Another level, the user-friendliness, consolidates the architecture with a management interface.

10.1.3 Integration in the TerveTaas project

So far this development has been made in parallel with other network specific logic systems. For evident efficiency reasons in the logic management, these parallel threads should be merged and the specific logic of the X10 and Linet networks which are redundant should be dropped. The risk of not doing so is that the place where a logic rule is defined could become difficult to localize.

10.2 Future improvements

Security aspects could be improved. For instance, a notification to the set of bundles should take place in case of unregistering of any entity.
The most interesting development would be to implement the programmable logic in th whole system. Another interesting development could be to integrate more networks, especially Bluetooth which is used to communicate with Rolo.

Part III

Appendices

Appendix A

serialManagement bundle

A.1 Exception

A.1.1 InexistentPortException.java

```
package driver.serialManagement.exception;


/**
 * Thrown when trying to allocate a inexistent serial port.
 *
 * @author Pierre Moermans
 */

public class InexistentPortException extends SerialPortException
{
    /**
     * Default constructor
     *
     * @param portName  the name of the serial port
     */
    public InexistentPortException(String portName) {
        super(portName);
    }
}
```

A.1.2 PortAlreadyAllocatedException.java

```
package driver.serialManagement.exception;

/**
 * Thrown when trying to allocated a port already allocated.
 *
 * @author Pierre Moermans
 */

public class PortAlreadyAllocatedException extends SerialPortException
{
    /**
     * Default constructor
     *
```

```
     * @param portName  the name of the serial port
     */
    public PortAlreadyAllocatedException(String portName) {
        super(portName);
    }
}
```

A.1.3 PortInUseException.java

```
package driver.serialManagement.exception;

/**
 * Thrown when trying to release (close) a port in use.
 *
 * @author Pierre Moermans
 */

public class PortInUseException extends SerialPortException
{
    /**
     * Default constructor
     *
     * @param portName  the name of the serial port
     */
    public PortInUseException(String portName) {
        super(portName);
    }


}
```

A.1.4 SerialPortException.java

```
package driver.serialManagement.exception;

/**
 * Exception thrown concerning a serial port.
 *
 * @author Pierre Moermans
 */

public class SerialPortException extends Exception
{
    /**
     * The name of the serial port that was attempted to be closed.
     *
     * @serial
     */
    private String portName;

    /**
     * Sets the name of the serial port that was tried to be closed.
     *
     * @param   portName    the name of the serial port.
     */
    public void setPortName(String portName)
    {
        this.portName = portName;
```

```
    }

    /**
     * Returns the name of theserial port that was tried to be closed.
     *
     * @return  the name of the serial port.
     */
    public String getPortName()
    {
        return portName;
    }

    /**
     * Default constructor
     *
     * @param portName  the name of the serial port
     */
    public SerialPortException(String portname) {
        setPortName(portName);
    }
}
```

A.2 Service

A.2.1 SerialManagementServ.java

```
package driver.serialManagement.service;

import gnu.io.SerialPort;
import driver.serialManagement.exception.*;

/**
 * The <code><b>SerialServ</b></code> interface declares the
 * <code>serial</code> service interface.
 *
 * @author Pierre Moermans
 */

public interface SerialManagementServ
{
    /**
     * Returns the serial port of name <code>portName</code>.
     *
     * The <code><b>getPort</b></code> method returns a Serialport
     * instance representing the serial port of name <code>portName</code>.
     *
     * @param   portName    the name of the serial port
     * @return              a SerialPort instance representing the serial port.
     * @throws  PortInUseException if the port <code>portName</code> is in use.
     * @throws  InexistentPortException     if the port <code>portName</code>
     *                                      doesn't exist.
     */
    SerialPort getPort(String portName) throws PortInUseException,
        InexistentPortException;

    /**
     * Returns the name of all the allocated serial ports.
     *
     * The <code><b>getAvailablePorts</b></code> method returns a
```

```
     * <code>String</code> array containing the name of all the allocated
     * serial ports. The allocated ports are both the ports in use and the
     * ports allocated but not in use.
     *
     * @return              a <code>String[]</code> containing the name of all
     *                      the allocated serial ports.
     *                      <code>NULL</code> if no port is allocated.
     */
    String[] getAllocatedPorts();

    /**
     * Returns the name of all the serial ports allocated but not in use.
     *
     * The <code><b>getPortsInUse</b></code> method returns a
     * <code>String</code> array containing the name of all the serial ports
     * currently allocated but not in use(by another bundle).
     *
     * @return              a <code>String[]</code> containing the name of all
     *                      the serial ports currently allocated but not in use.
     *                      <code>NULL</code> if no ports are allocated.
     */
    String[] getAvailablePorts();

    /**
     * Returns the name of all the serial ports in use.
     *
     * The <code><b>getPortsInUse</b></code> method returns a
     * <code>String</code> array containing the name of all the serial ports
     * currently in use.
     *
     * @return              a <code>String[]</code> array containing the name
     *                      of all the serial ports currently in use.
     *                      <code>NULL</code> if there is no available ports.
     */
    String[] getPortsInUse();

    /**
     * Returns the name of the unallocated, free serial ports.
     *
     * The <code><b>getFreeUnallocatedPorts</b></code> method return a
     * <code>String</code> array containing the name of all the
     * <b>free</b>, unallocated serial ports.
     *
     * @return  a <code>String</code> array containing the name of all the
     *          free, unallocated serial ports.
     *          <code>NULL</code> if there is no free, unallocated serial ports.
     */
    String[] getFreeUnallocatedPorts();

    /**
     * Returns true if the serial port <code>portName</code> is currently
     * in use.
     *
     * The <code><b>isPortInUse</b></code> method returns <code>true</code>
     * if the SerialPort <code>portName</code> is currently in use.
     *
     * @param   portName    the name of the serial port.
     * @return              <code>true</code> if the serial port is in use,
     *                      <code>false</code> otherwise.
     */
    boolean isPortInUse(String portName);
```

```
/**
 * Returns true if the serial port <code>portName</code> is currently
 * available.
 *
 * The <code><b>isPortAvailable</b></code> method returns <code>true</code>
 * if the SerialPort <code>portName</code> is currently available.
 *
 * @param   portName    the name of the serial port.
 * @return              <code>true</code> if the serial port is available,
 *                      <code>false</code> otherwise.
 */
boolean isPortAvailable(String portName);

/**
 * Forces the release of the serial port <code>portName</code>.
 *
 * The <code><b>freePort</b></code> method closes the serial port
 * <code>portName</code>. If the serial port <code>portName</code>
 * is not in the available ports nor in the "is use" ports, the method
 * does nothing.
 *
 * @param   portName    the name of the serial port
 * @throws  PortinUseException  when the serial port <code>portName</code>
 *                      is currently in use.
 * @throws InexistentPortException when <code>portName</code> doesn't exist.
 */
void freePort(String portName) throws
    driver.serialManagement.exception.PortInUseException,
    InexistentPortException;

/**
 * Forces the release of all the available (not in use) serial ports.
 *
 * The <code><b>freeAvailablePort</b></code> method closes all the
 * available (not in use) serial ports.
 *
 *@return   a <code>String</code> array containing the names of the ports
 *          that have been closed.
 *          <code>NULL</code> if no door has been closed.
 */
String[] freeAvailablePorts();


/**
 * Makes the serial port available again.
 *
 * The <code><b>releasePort</b></code> method makes the serial port
 * <code>portName</code> available again for other futur usage.
 *
 * @param   portName the name of the serial port
 * @throws  InexistentPortException     if the released port is not in use.
 */
void releasePort(String portName) throws InexistentPortException;


/**
 * Allocates the serial port <code>portName</code>.
 *
 * The <code><b>allocatePort</b></code> method tries to allocate
 * (put a lock) on the serial port the name is <code>portName</code>
 *
 * @param   portName    the name of the serial port
```

```
     * @return              <code>true</code> if the port has been allocated
     *                      <code>false</code> otherwise.
     * @throws  PortAlreadyAllocatedException   when the port is already
     *                                          allocated.
     * @throws  InexistentPortException     when <code>portName</code> doesn't
     *                                      exist.
     */
    boolean allocatePort(String portName) throws PortAlreadyAllocatedException,
        InexistentPortException;

    /**
     * Allocates all possible serial port.
     *
     * The <code><b>allocateEveryPort</b></code> method tries to allocate every
     * unallocated, free serial port.
     *
     * @return  String[]    a <code>String</code> array containing the name
     *                      of the newly allocated serial ports.
     *                      <code>NULL</code> if no serial port has been
     *                      allocated.
     */
    String[] allocateEveryPort();

    /**
     * Force the release of a port actually in use.
     *
     * The <code><b>releaseInUsePort</b></code> method should only be used
     * in extreme case when a bundle has failed and doesn't release a port.
     *
     * @param   portName    the serial port name
     * @throws InexistentPortException if the serial port <code>portName</code>
     *                                 is not actually in use.
     */
    void freeInUsePort(String portName)
        throws InexistentPortException;

    /**
     * Add a serial port to the port-in-use list.
     *
     * The <code><b>addPortInUse</b></code> adds a serial port to the in-use
     * list. This method should be called when a port is already allocated by
     * a program that detects the registration of the SerialManagement service.
     *
     * @param   port    the serial port to add.
     */
    void addPortInUse(SerialPort port);
}
```

A.3 Implementation

A.3.1 SerialManagementImpl.java

```
package driver.serialManagement.implementation;

import driver.serialManagement.service.SerialManagementServ;
import driver.serialManagement.exception.*;
import java.util.*;
import gnu.io.SerialPort;
import gnu.io.CommPortIdentifier;
```

```
/**
 * The <code><b>SerialImpl</b></code> provides the implementation of the
 * <code>serial</code> service.
 *
 * By default, when the class is instanciated, it will allocate (put a lock)
 * on all the serial ports it can.
 *
 * @author Pierre Moermans
 */

public class SerialManagementImpl implements SerialManagementServ
{
    private Vector inUse, available;

    /**
     * Returns the name of all the serial ports allocated but not in use.
     *
     * The <code><b>getPortsInUse</b></code> method returns a
     * <code>String</code> array containing the name of all the serial ports
     * currently allocated but not in use(by another bundle).
     *
     * @return  a <code>String[]</code> containing the name of all
     *          the serial ports currently allocated but not in use.
     *          <code>NULL</code> if no ports are allocated.
     */
    public String[] getAvailablePorts() {
        if (available.size() == 0) return null;
        String[] portNames = new String[available.size()];
        SerialPort port;
        for (int i = 0; i < portNames.length; i++) {
            port = (SerialPort) available.elementAt(i);
            portNames[i] = port.getName();
        }
        return portNames;
    }

    /**
     * Returns true if the serial port <code>portName</code> is currently
     * available.
     *
     * The <code><b>isPortAvailable</b></code> method returns <code>true</code>
     * if the SerialPort <code>portName</code> is currently available.
     *
     * @param   portName    the name of the serial port.
     * @return              <code>true</code> if the serial port is available,
     *                      <code>false</code> otherwise.
     */
    public boolean isPortAvailable(String portName){
        if (available.size() == 0) return false;
        SerialPort port;
        for (int i = 0; i < available.size(); i++) {
            port = (SerialPort) available.elementAt(i);
            if (portName.equals(port.getName())) return true;
        }
        return false;
    }

    /**
     * Returns the name of all the allocated serial ports.
     *
     * The <code><b>getAvailablePorts</b></code> method returns a
```

```
     * <code>String</code> array containing the name of all the allocated
     * serial ports. The allocated ports are both the ports in use and the
     * ports allocated but not in use.
     *
     * @return  a <code>String[]</code> containing the name of all
     *          the allocated serial ports.
     *          <code>NULL</code> if no port is allocated.
     */
    public String[] getAllocatedPorts() {
        String[] portNames = new String[inUse.size() + available.size()];
        if (portNames.length == 0) return null;
        int j = 0;
        SerialPort port;
        for (int i = 0; i < inUse.size(); i++) {
            port = (SerialPort) inUse.elementAt(i);
            portNames[j] = port.getName();
            j++;
        }
        for (int i = 0; i < available.size(); i++) {
            port = (SerialPort) available.elementAt(i);
            portNames[j] = port.getName();
            j++;
        }
        return portNames;
    }

    /**
     * Forces the release of the serial port <code>portName</code>.
     *
     * The <code><b>freePort</b></code> method closes the serial port
     * <code>portName</code>. If the serial port <code>portName</code>
     * is not in the available ports nor in the "is use" ports, the method
     * does nothing.
     *
     * @param   portName    the name of the serial port
     * @throws  PortinUseException  when the serial port <code>portName</code>
     *                      is currently in use.
     * @throws InexistentPortException when <code>portName</code> doesn't exist.
     */
    public void freePort(String portName) throws PortInUseException,
        InexistentPortException {
        if (isPortInUse(portName))
            throw new PortInUseException(portName);
        if (! isPortAvailable(portName))
            throw new InexistentPortException(portName);
        SerialPort port;
        for (int i = 0; i < available.size(); i++) {
            port = (SerialPort) available.elementAt(i);
            if (port.getName().equals(portName)) {
                available.removeElementAt(i);
                port.close();
            }
        }
    }

    /**
     * Makes the serial port available again.
     *
     * The <code><b>releasePort</b></code> method makes the serial port
     * <code>portName</code> available again for other futur usage.
     *
     * @param   portName the name of the serial port
```

```
     * @throws  InexistentPortException     if the released port is not in use.
     */
    public void releasePort(String portName) throws InexistentPortException {
        if (! isPortInUse(portName))
            throw new InexistentPortException(portName);
        SerialPort port = null;
        for (int i = 0; i < inUse.size(); i++) {
            port = (SerialPort) inUse.elementAt(i);
            if (port.getName().equals(portName)) {
                inUse.removeElementAt(i);
                available.addElement(port);
            }
        }
    }

    /**
     * Returns the name of all the serial ports in use.
     *
     * The <code><b>getPortsInUse</b></code> method returns a
     * <code>String</code> array containing the name of all the serial ports
     * currently in use.
     *
     * @return  a <code>String[]</code> array containing the name
     *          of all the serial ports currently in use.
     *          <code>NULL</code> if there is no available ports.
     */
    public String[] getPortsInUse() {
        if (inUse.size() == 0) return null;
        String[] portNames = new String[inUse.size()];
        SerialPort port;
        for (int i = 0; i < inUse.size(); i++) {
            port = (SerialPort) inUse.elementAt(i);
            portNames[i] = port.getName();
        }
        return portNames;
    }

    /**
     * Returns the name of the unallocated, free serial ports.
     *
     * The <code><b>getFreeUnallocatedPorts</b></code> method return a
     * <code>String</code> array containing the name of all the
     * <b>free</b>, unallocated serial ports.
     *
     * @return  a <code>String</code> array containing the name of all the
     *          free, unallocated serial ports.
     *          <code>NULL</code> if there is no free, unallocated serial ports.
     */
    public String[] getFreeUnallocatedPorts() {
        Vector vect = new Vector(5);
        Enumeration enum = CommPortIdentifier.getPortIdentifiers();
        String portName;
        while (enum.hasMoreElements())
        {
            CommPortIdentifier element =
                (CommPortIdentifier) enum.nextElement();
            if (element.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                portName = element.getName();
                if (! isPortAllocated(portName)) vect.add(portName);
            }
        }
        if (vect.size() == 0) return null;
```

```
        String[] portNames = new String[vect.size()];
        for (int i = 0; i < vect.size(); i++) {
            portNames[i] = (String) vect.elementAt(i);
        }
        return portNames;
    }

    /**
     * Allocates all possible serial port.
     *
     * The <code><b>allocateEveryPort</b></code> method tries to allocate every
     * unallocated, free serial port.
     *
     * @return  String[]    a <code>String</code> array containing the name
     *                      of the newly allocated serial ports.
     *                      <code>NULL</code> if no serial port has been
     *                      allocated.
     */
    public String[] allocateEveryPort() {
        Enumeration enum = CommPortIdentifier.getPortIdentifiers();
        if (! enum.hasMoreElements()) return null;
        Vector vect = new Vector();
        SerialPort port;
        while (enum.hasMoreElements()) {
            CommPortIdentifier element =
                (CommPortIdentifier) enum.nextElement();
            if (element.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                try {
                    port = (SerialPort) element.open("Serial Manager", 2000);
                    available.add(port);
                    vect.add(port.getName());
                } catch (gnu.io.PortInUseException e) {
                    // do nothing
                }
            }
        }
        String[] portNames = new String[vect.size()];
        for (int i = 0; i < vect.size(); i++) {
            portNames[i] = (String) vect.elementAt(i);
        }
        return portNames;
    }

    /**
     * Forces the release of all the available (not in use) serial ports.
     *
     * The <code><b>freeAvailablePort</b></code> method closes all the
     * available (not in use) serial ports.
     *
     * @return  a <code>String</code> array containing the names of the ports
     *          that have been closed.
     *          <code>NULL</code> if no door has been closed.
     */
    public String[] freeAvailablePorts() {
        if (available.size() == 0) return null;
        String[] portNames = new String[available.size()];
        SerialPort port;
        for (int i = 0; i < available.size(); i++) {
            port = (SerialPort) available.elementAt(i);
            port.close();
            portNames[i] = port.getName();
        }
```

```
        available.removeAllElements();
        return portNames;
    }

    /**
     * Returns the serial port of name <code>portName</code>.
     *
     * The <code><b>getPort</b></code> method returns a Serialport
     * instance representing the serial port of name <code>portName</code>.
     *
     * @param   portName    the name of the serial port
     * @return              a SerialPort instance representing the serial port
     * @throws  PortInUseException if the port <code>portName</code> is in use.
     * @throws  InexistentPortException     if the port <code>portName</code>
     *                                      doesn't exist.
     */
    public SerialPort getPort(String portName) throws PortInUseException,
        InexistentPortException {
        if (isPortInUse(portName)) throw new PortInUseException(portName);
        if (! isPortAvailable(portName))
            throw new InexistentPortException(portName);
        SerialPort port = null;
        for (int i = 0; i < available.size(); i++) {
            port = (SerialPort) available.elementAt(i);
            if (portName.equals(port.getName())) break;
        }
        if (port == null) System.out.println("SerialManagement: port is null");
        available.remove(port);
        inUse.add(port);
        return port;
    }

    /**
     * Allocates the serial port <code>portName</code>.
     *
     * The <code><b>allocatePort</b></code> method tries to allocate
     * (put a lock) on the serial port the name is <code>portName</code>
     *
     * @param   portName    the name of the serial port
     * @throws  PortAlreadyAllocatedException   when the port is already
     *                                          allocated.
     * @throws  InexistentPortException     when <code>portName</code> doesn't
     *                                      exist.
     */
    public boolean allocatePort(String portName)
        throws PortAlreadyAllocatedException, InexistentPortException {
        if (isPortAllocated(portName))
            throw new PortAlreadyAllocatedException(portName);
        Enumeration enum = CommPortIdentifier.getPortIdentifiers();
        if (! enum.hasMoreElements())
            throw new InexistentPortException(portName);
        CommPortIdentifier cpi = null;
        SerialPort port;
        while (enum.hasMoreElements()) {
            CommPortIdentifier element =
                (CommPortIdentifier) enum.nextElement();
            if (element.getPortType() == CommPortIdentifier.PORT_SERIAL)
                if (element.getName().equals(portName)) {
                    cpi = element;
                    break;
                }
        }
```

```
        if (cpi == null)
            throw new InexistentPortException(portName);
        try {
            port = (SerialPort) cpi.open("Serial Manager", 2000);
            available.add(port);
        } catch (gnu.io.PortInUseException e) {
            return false;
        }
        return true;
    }

    /**
     * Returns true if the serial port <code>portName</code> is currently
     * in use.
     *
     * The <code><b>isPortInUse</b></code> method returns <code>true</code>
     * if the SerialPort <code>portName</code> is currently in use.
     *
     * @param   portName    the name of the serial port.
     * @return              <code>true</code> if the serial port is in use,
     *                      <code>false</code> otherwise.
     */
    public boolean isPortInUse(String portName) {
        SerialPort port;
        for (int i = 0; i < inUse.size(); i++) {
            port = (SerialPort) inUse.elementAt(i);
            if (portName.equals(port.getName())) return true;
        }
        return false;
    }

    /**
     * Returns true if the port <code>portName</code> is allocated.
     *
     * The <code><b>isPortAllocated</b></code> method returns <code>true</code>
     * if the SerialPort <code>portName</code> is available or in use.
     * @param   portName    the name of the serial port.
     * @return              <code>true</code> if the serial port is in use,
     *                      <code>false</code> otherwise.
     */
    public boolean isPortAllocated(String portName) {
        if (isPortInUse(portName) | isPortAvailable(portName)) return true;
        return false;
    }

    /**
     * Force the release of a port actually in use.
     *
     * The <code><b>releaseInUsePort</b></code> method should only be used
     * in extreme case, when a bundle has failed and doesn't release a port.
     *
     * @param   portName    the serial port name
     * @throws InexistentPortException if the serial port <code>portName</code>
     *                                  is not actually in use.
     */
    public void freeInUsePort(String portName)
        throws InexistentPortException {
        if (! isPortInUse(portName))
            throw new InexistentPortException(portName);
        SerialPort port = null;
        // we are sure we'll find the port
        for (int i = 0; i < inUse.size(); i++) {
```

```
            port = (SerialPort) inUse.elementAt(i);
            if (portName.equals(port.getName())) break;
        }
        port.close();
    }

    /**
     * Add a serial port to the port-in-use list.
     *
     * The <code><b>addPortInUse</b></code> adds a serial port to the in-use
     * list. This method should be called when a port is already allocated by
     * a program that detects the registration of the SerialManagement service.
     *
     * @param   port    the serial port to add.
     */
    public void addPortInUse(SerialPort port) {
        inUse.add(port);
    }

    /**
     * Default constructor
     */
    public SerialManagementImpl() {
        inUse = new Vector(4);
        available = new Vector(4);
    }
}
```

A.4 SerialManagementActiv.java (Activator)

```
package driver.serialManagement;

import driver.serialManagement.implementation.SerialManagementImpl;
import driver.serialManagement.service.SerialManagementServ;
import org.osgi.framework.*;

/**
 * The <code><b>SerialActiv</b></code> class is the activator of the
 * <code>serial</code> bundle.
 * The <code>SerialActiv</code> register a service <code>SerialServ</code>
 * that provides an easier access and management of the serial port.
 *
 * @author Pierre Moermans
 */

public class SerialManagementActiv implements BundleActivator
{
    private ServiceRegistration serviceReg;
    private SerialManagementImpl service;

    public void start(BundleContext ctxt) {
        service = new SerialManagementImpl();
        service.allocateEveryPort();
        serviceReg =
            ctxt.registerService(
            "driver.serialManagement.service.SerialManagementServ",
            service, null);
        String[] ports = service.getFreeUnallocatedPorts();
        System.out.print("unallocated serial ports:");
```

```
        if (ports == null)
            System.out.println(" none");
        else {
            System.out.println();
            for (int i = 0; i < ports.length; i++) {
                System.out.println("\t" + ports[i]);
            }
        }
        ports = service.getAvailablePorts();
        System.out.print("available serial ports:");
        if (ports == null)
            System.out.println(" none");
        else {
            System.out.println();
            for (int i = 0; i < ports.length; i++) {
                System.out.println("\t" + ports[i]);
            }
            System.out.println("serialManagement bundle started");
        }
    }

    public void stop(BundleContext ctxt) {
        // releases all the serial ports (removes lock files)
        service.freeAvailablePorts();
        // unregistering is done automatically
        System.out.println("serialManagement bundle stopped");
    }
}
```

A.5 Manifest

```
Export-Package: driver.serialManagement.service, driver.serialManagement.exception
Bundle-Activator: driver.serialManagement.SerialManagementActiv
```

Appendix B

SINE driver bundle *sine*

B.1 Events

B.1.1 SineEvent.java

```
package driver.sine.event;

import driver.sine.service.SensorStatus;

/**
 * The <code>SineEvent</code> is the event sent when a change occurs to any of
 * the sine network sensors.
 *
 * @author Pierre Moermans
 */

public class SineEvent extends Throwable
{
    private SensorStatus sensorStatus;

    /**
     * Set the sensor status
     */
    public void setSensorStatus(SensorStatus sensorStatus) {
        this.sensorStatus = sensorStatus;
    }

    /**
     * Returns the sensor status
     */
    public SensorStatus getSensorStatus() {
        return sensorStatus;
    }

    /**
     * Default constructor
     */
    public SineEvent(SensorStatus status) {
        setSensorStatus(status);
    }
}
```

B.1.2 SineEventListener.java

```
package driver.sine.event;

/**
 * The SineEventListener class defines the interface for sine change event.
 *
 * @author Pierre Moermans
 */

public interface SineEventListener
{
    /**
     * invoked whenever a sine sensor status changes
     */
    void statusChanged(SineEvent event);
}
```

B.2 Exceptions

B.2.1 SensorStatusNotAvailableException.java

```
package driver.sine.exception;

/**
 * The <code><b>SensorStatusNotAvailableException</b></code> exception
 * is thrown when the sensor status is requested without having been
 * initialized.
 *
 * @author Pierre Moermans
 */

public class SensorStatusNotAvailableException extends Exception
{

}
```

B.3 Service

B.3.1 SensorStatus.java

```
package driver.sine.service;


/**
 * Describes the available requests to a SINE sensor
 *
 * @author Pierre Moermans
 */


public interface SensorStatus
{
    /**
     * arlarm status on
     *
```

```
 * @serial
 */
static final int ON = 1;

/**
 * alarm status off
 *
 * @serial
 */
static final int OFF = 2;

/**
 * sensor status connected
 *
 * @serial
 */
static final int CONNECTED = 1;

/**
 * sensor status disconnected
 *
 * @serial
 */
static final int DISCONNECTED = 2;

/**
 * Returns the alarm status.
 *
 * The <code><b>getAlarmStatus</b></code> method returns the alarm status,
 * that is to say, <code>ON</code> id the alarm went off, <code>OFF</code>
 * otherwise.
 *
 * @return  int     the status of the sensor's alarm.
 */
int getAlarmStatus();

/**
 * Returns the connection status.
 *
 * The <code><b>getConnectionStatus</b></code> method returns the
 * connection status of the sensor, that is to say,
 * <code>CONNECTED</code> if the sensor is connected,
 * <code>DISCONNECTED</code> otherise.
 *
 * @return  int     the status of the sensor's connection.
 */
int getConnectionStatus();

/**
 * Returns the id of the sensor.
 *
 * Returns the id of the sensor as an <code>int</code>.
 *
 * @return  int     the id of the sensor.
 */
int getId();

/**
 * Compare to another sensor status
 *
 * The <code><b>equals</b></code> method return true if the sensor
 * status equals the <code>status</code>
```

```
     *
     * @param   status  a SensorStatus to compare
     * @return  boolean     return <code>true</code> if the two sensors status
     *                      are equals, <code>false</code> otherwise.
     */
    boolean equals(SensorStatus status);
}
```

B.3.2 SineServ.java

```
package driver.sine.service;
import driver.sine.event.*;
import driver.sine.exception.SensorStatusNotAvailableException;


/**
 * Describe the available requests to a SINE network.
 *
 * @author Pierre Moermans
 */

public interface SineServ
{
    /**
     * Returns the status of all the SINE sensors.
     *
     * the <code><b>getNetworkStatus</b></code> method returns a array of
     * <code>SensorStatusInterface</code> describing the state of each sensor
     * of the network.
     *
     * @return  SensorStatusInterface[]     the status of each sensor of the
     *                                      network.
     * @throws  SensorStatusNotAvailableException   if the network's
     *                  sensors status is requested but not available.
     *                  This could happen if the SINE controller is not
     *                  connected or if the request arrives while the SINE
     *                  controller is initializing (which takes a while).
     */
    public SensorStatus[] getNetworkStatus()
        throws SensorStatusNotAvailableException;

    /**
     * Add a listener for sine changes events.
     *
     * the <code><b>addListener</b></code> method adds a sine change listener.
     */
    public void addListener(SineEventListener listener);

    /**
     * Removes a sine change listener.
     *
     * The <code><b>removeListner</b></code> method removes a sine change
     * listener.
     */
    public void removeListener(SineEventListener listener);
}
```

B.4 Implementation

B.4.1 SineImpl.java

```
package driver.sine.implementation;

import driver.sine.service.*;
import driver.sine.event.*;
import driver.sine.exception.SensorStatusNotAvailableException;
import gnu.io.*;
import java.util.Vector;
import java.io.*;


/**
 * The <code><b>SineImpl</b></code> class is the implementation of
 * the <code>SineServ</code> interface.
 * This class is responsible of reading the controller through the serial port
 * and to translate the informations into the correscponding data structure.
 *
 * @author Pierre Moermans
 */

public class SineImpl implements SineServ, SerialPortEventListener
{
    private static final int BUFFER_SIZE = 6;
    private SerialPort port;
    private Vector listeners;
    private boolean sensorStatusAvailable;

    // this is what should change in a "real" version of the protocol
    private SensorStatusImpl[] sensors;
    /**
     * Standard constructor.
     *
     * @param   portName    the name of the serial port the SINE controller
     *                      is connected to.
     */
    public SineImpl(SerialPort port) {
        this.port = port;
        initializeSensorStatus();
        listeners = new Vector(4);
        initSerialPort();
    }

    /**
     * Initialize the port for transmition
     */
    private void initSerialPort() {
        try {
            port.setSerialPortParams(9600,
                                     SerialPort.DATABITS_8,
                                     SerialPort.STOPBITS_1,
                                     SerialPort.PARITY_NONE);
        }
        catch (UnsupportedCommOperationException ex) {
            System.out.println("Unsupported Comm Operation ... error");
        }
        //
        // THE FOLLOWING CODE GENERATES AN ERROR WHILE UPDATING
        // THE BUNDLE (NOT THE FIRST TIME
        // NO WORK AROUND HAS BEEN FOUND SO FAR
```

```
        //
        try {
            port.notifyOnDataAvailable(true);
        }
        catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    /**
     * Initialize the sensor status
     */
    private void initializeSensorStatus() {
        sensors = new SensorStatusImpl[3];
        for (int i = 0; i < sensors.length; i++) {
            sensors[i] = new SensorStatusImpl(i);
        }
        sensorStatusAvailable = false;
    }

    /**
     * Returns the status of all the SINE sensors.
     *
     * the <code><b>getNetworkStatus</b></code> method returns a array of
     * <code>SensorStatusInterface</code> describing the state of each sensor
     * of the network.
     *
     * @return  SensorStatusInterface[]     the status of each sensor of the
     *                                      network.
     */
    public SensorStatus[] getNetworkStatus()
        throws SensorStatusNotAvailableException {
        if (! sensorStatusAvailable)
            throw new SensorStatusNotAvailableException();
        return sensors;
    }

    /**
     * Removes a sine change listener.
     *
     * The <code><b>removeListner</b></code> method removes a sine change
     * listener.
     */
    public void removeListener(SineEventListener listener) {
        listeners.remove(listener);
    }

    /**
     * Add a listener for sine changes events.
     *
     * the <code><b>addListener</b></code> method adds a sine change listener.
     */
    public void addListener(SineEventListener listener) {
        listeners.add(listener);
    }


    /**
     * Manage everything related to the data reading.
     */
    public void serialEvent(SerialPortEvent evt) {
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
```

```
// acutally, this is the only expected event on the serial port.
byte[] buffer = new byte[BUFFER_SIZE];
// make backup of the old status
SensorStatusImpl[] oldSensors = new SensorStatusImpl[3];
for (int i = 0; i < sensors.length; i++) {
    oldSensors[i] = new SensorStatusImpl(sensors[i].getId());
    oldSensors[i].setAlarmStatus(sensors[i].getAlarmStatus());
    oldSensors[i].setConnectionStatus(
        sensors[i].getConnectionStatus());
}
// lets read the serial port
try {
    InputStream in = port.getInputStream();
    while (in.available()>0) {
        in.read(buffer);
    }
} catch (IOException e) {
    System.out.println("Error occured while reading serial port");
    // do nothing
}
// now we have the bytes, especialy the last one
// let's analyse the content.

// all the information is in the last byte.


// let's print a bit ...
System.out.println("byte is " + Integer.toBinaryString((int) buffer[5]));

int status;
status = (int) (buffer[5] & 128);
if (status == 0)
    sensors[2].setConnectionStatus(SensorStatus.CONNECTED);
else
    sensors[2].setConnectionStatus(SensorStatus.DISCONNECTED);
status = (int) (buffer[5] & 64);
if (status == 0)
    sensors[1].setConnectionStatus(SensorStatus.CONNECTED);
else
    sensors[1].setConnectionStatus(SensorStatus.DISCONNECTED);
status = (int) (buffer[5] & 32);
if (status == 0)
    sensors[0].setConnectionStatus(SensorStatus.CONNECTED);
else
    sensors[0].setConnectionStatus(SensorStatus.DISCONNECTED);
status = (int) (buffer[5] & 16);
if (status == 0)
    sensors[2].setAlarmStatus(SensorStatus.ON);
else
    sensors[2].setAlarmStatus(SensorStatus.OFF);
status = (int) (buffer[5] & 8);
if (status == 0)
    sensors[1].setAlarmStatus(SensorStatus.ON);
else
    sensors[1].setAlarmStatus(SensorStatus.OFF);
status = (int) (buffer[5] & 4);
if (status == 0)
    sensors[0].setAlarmStatus(SensorStatus.ON);
else
    sensors[0].setAlarmStatus(SensorStatus.OFF);

// we have the new sensor status
```

```
            // if this is the first info we get from the SINE controller
            // (the sensorStatus weren't yet available), then fire an event
            // otherwise,
            // we compare each of them to the corresponding old status
            // for each sensor status change, fire a SineEvent
            if (! sensorStatusAvailable) {
                for (int i = 0; i < sensors.length; i++)
                    fireSineEvent(sensors[i]);
            }
            else {
                for (int i = 0; i < sensors.length; i++) {
                    if (! oldSensors[i].equals(sensors[i])) {
                        fireSineEvent(sensors[i]);
                    }
                }
            }
            for (int i = 0; i < sensors.length; i++) {
                System.out.println(sensors[i].toString());
            }
        }
    }

    /**
     * invoke the statusChanged method of every listener;
     */
    private void fireSineEvent(SensorStatus status) {
        SineEvent event = new SineEvent(status);
        SineEventListener listener;
        synchronized(listeners) {
            for (int i = 0; i < listeners.size(); i++) {
                listener = (SineEventListener) listeners.elementAt(i);
                listener.statusChanged(event);
            }
        }
    }
}
```

B.5 Activator SineActivator.java

```
package driver.sine;

import org.osgi.framework.*;
import driver.serialManagement.service.*;
import driver.serialManagement.exception.*;
import driver.sine.implementation.SineImpl;
import gnu.io.SerialPort;
import java.util.Properties;

/**
 * The <code><b>SineActivator</b></code> class is the activator of the
 * sine bundle.
 *
 * @author Pierre Moermans
 */

public class SineActivator implements BundleActivator
{
    private SineImpl sineImpl;
    private SerialManagementServ sms;
```

```
private SerialPort[] ports;

// this must be update if several controllers are needed
private final String[] portsToStart = {"/dev/ttyS0"};

public void start(BundleContext ctxt) {
    try {
        ServiceReference sr =
            ctxt.getServiceReference(
            "driver.serialManagement.service.SerialManagementServ");
        sms = (SerialManagementServ) ctxt.getService(sr);

        ports = new SerialPort[portsToStart.length];
        for (int i = 0; i < ports.length; i++)
        {
            ports[i] = null;
            if (! sms.isPortAvailable(portsToStart[i])) {
                System.out.println(
                    "serial port -" + portsToStart[i] + "- not available.");
                continue;
            }
            try {
                sms.allocatePort(portsToStart[i]);
            }
            catch (InexistentPortException e) {
            }
            catch (PortAlreadyAllocatedException e) {
            }
            try {
                ports[i] = sms.getPort(portsToStart[i]);
            } catch (PortInUseException e) {
                System.out.println(
                    "error: serial port " +
                        portsToStart[i] +
                        " already in use.");
            } catch (InexistentPortException e) {
                System.out.println(
                    "error: serial port " +
                        portsToStart[i] +
                        " doesn't exist");
            }
            sineImpl = new SineImpl(ports[i]);
            try {
                ports[i].addEventListener(sineImpl);
            } catch (java.util.TooManyListenersException e) {
                System.out.println("too many listeners");
            }
            Properties props = new Properties();
            props.put("serial port",portsToStart[i]);
            ctxt.registerService(
                "driver.sine.service.SineServ",sineImpl,props);
            System.out.println("sine bundle started");
        }
    }
    catch (Exception ex) { ex.printStackTrace(); }
}

public void stop(BundleContext ctxt) {
    for (int i = 0; i < ports.length; i++)
    {
        ports[i].removeEventListener();
        //release the serial port
```

```
            try {
                sms.releasePort(ports[i].getName());
                try {
                    sms.freePort(ports[i].getName());
                } catch (PortInUseException e) {
                } catch (InexistentPortException e) {
                }
            } catch (InexistentPortException e) {
                // do nothing
                System.out.println("InexistentPort: " +
                                   ports[i].getName());
            }
        }
        System.out.println("sine bundle stopped");
    }
}
```

B.6 Manifest

```
Bundle-Activator: driver.sine.SineActivator
Import-Package: driver.serialManagement.service, driver.serialManagement.exception
Export-Package: driver.sine.service, driver.sine.exception, driver.sine.event
```

Appendix C

Demonstration bundles*sinedemo*

C.1 SineDemo.java (Activator)

```
package driver.sinedemo;

import org.osgi.framework.*;
import driver.sine.service.SineServ;


/**
 * The class <code><b>SineDemo</b></code> is a simple user interace
 * the only purpose is to show the usage of the <code>sine</code> bundle.
 *
 * @author Pierre Moermans
 */

public class SineDemo implements BundleActivator
{
        private SineServ ss;
        private SineDemoFrame sdf;
        public void start(BundleContext ctxt) {
                ServiceReference sr = ctxt.getServiceReference("sine.service.SineServ");
                ss = (SineServ) ctxt.getService(sr);
                sdf = new SineDemoFrame(ss);
                sdf.setVisible(true);
        }

        public void stop(BundleContext ctxt) {
                sdf.setVisible(false);
                sdf = null;
                ss.removeListener(sdf);
        }
}
```

C.2 SineDemoFrame.java

```
package driver.sinedemo;
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import driver.sine.service.*;
import driver.sine.event.*;

/**
 * The <code><b>SineDemoFrame</b></code> is the main frame of the sine demo.
 *
 * @author Pierre Moermans
 */

public class SineDemoFrame extends JFrame implements SineEventListener
{
        private SineServ sineServ;
        private JLabel[] connectionLabels;
        private JLabel[] alarmLabels;

        public SineDemoFrame(SineServ sineServ) {
                super("Sine demo");
                this.sineServ = sineServ;
                sineServ.addListener(this);
                buildInterface();
                setSize(400,300);
        }

        private void buildInterface() {
                Container container = getContentPane();
                container.setLayout(new GridLayout(6,3));
                connectionLabels = new JLabel[3];
                alarmLabels = new JLabel[3];
                for (int i = 0; i < connectionLabels.length; i++) {
                        connectionLabels[i] = new JLabel();
                        alarmLabels[i] = new JLabel();
                }
                for (int i = 0; i < connectionLabels.length; i++) {
                        container.add(new JLabel("sensor " + i));
                        container.add(new JLabel("connection"));
                        container.add(connectionLabels[i]);
                        container.add(new JLabel());
                        container.add(new JLabel("alarm"));
                        container.add(alarmLabels[i]);
                }
        }

        /**
         * invoked whenever a sine sensor status changes
         */
        public void statusChanged(driver.sine.event.SineEvent event) {
                SensorStatus status = event.getSensorStatus();
                int id = status.getId();
                if (status.getAlarmStatus() == SensorStatus.ON)
                        alarmLabels[id].setText("ON");
                else
                        alarmLabels[id].setText("OFF");
                if (status.getConnectionStatus() == SensorStatus.CONNECTED)
                        connectionLabels[id].setText("CONNETED");
                else
                        connectionLabels[id].setText("DISCONNECTED");
        }
```

```
}
```

C.3 Manifest

```
Bundle-Activator: driver.sinedemo.SineDemo
import-package: driver.sine.service, driver.sine.exception, driver.sine.event
```

Appendix D

Linet driver bundle *linet*

D.1 Events

D.1.1 LinetEvent.java

```
package driver.linet.event;

/**
 * LinetEvent.java
 *
 * @author Pierre Moermans
 */

public class LinetEvent extends Throwable
{
    private int nodeNb;
    private short value;

    public void setValue(short value) {
        this.value = value;
    }

    public short getValue() {
        return value;
    }

    public void setNodeNb(int nodeNb) {
        this.nodeNb = nodeNb;
    }

    public int getNodeNb() {
        return nodeNb;
    }

    public LinetEvent(int number, short value) {
        setNodeNb(number);
        setValue(value);
    }

}
```

D.1.2 LinetEventListener.java

```
package driver.linet.event;

import driver.linet.event.LinetEvent;
/**
 * LinetEventListener.java
 *
 * @author Pierre Moermans
 */

public interface LinetEventListener
{
    void stateChanged(LinetEvent event);
}
```

D.2 Exceptions

D.2.1 LinetDeviceException.java

```
package driver.linet.exception;

/**
 * LinetDeviceException.java
 *
 * @author Pierre Moermans
 */

public class LinetDeviceException extends Exception
{
    private int device;

    public void setDevice(int device) {
        this.device = device;
    }

    public int getDevice() {
        return device;
    }

    public LinetDeviceException(int device) {
        setDevice(device);
    }
}
```

D.2.2 NotOnOffDeviceException.java

```
package driver.linet.exception;

/**
 * NotOnOffDeviceException.java
 *
 * @author Pierre Moermans
 */

public class NotOnOffDeviceException extends LinetDeviceException
{
```

```
    public NotOnOffDeviceException(int device) {
        super(device);
    }
}
```

D.2.3 NotValueDeviceException.java

```
package driver.linet.exception;

/**
 * NotValueDeviceException.java
 *
 * @author Pierre Moermans
 */

public class NotValueDeviceException extends LinetDeviceException
{
    public NotValueDeviceException(int device) {
        super(device);
    }
}
```

D.3 Services

D.3.1 Linet.java

```
package driver.linet.service;
import driver.linet.event.*;
import driver.linet.exception.*;

/**
 * The <code><b>Linet</b></code> interface describes the access point
 * to the <code><b>linet</b></code> service.
 *
 * @author Pierre Moermans
 */

public interface Linet
{
    /**
     * Adds a LinetEvent listener.
     *
     * @param   listener    A linet event listener.
     */
    void addListener(LinetEventListener listener);

    /**
     * Removes a LinetEvent listener.
     *
     * @param   listener    A linet event listener.
     */
    void removeListener(LinetEventListener listener);

    /**
     * changes the status of a linet ON-OFF device.
     *
     * @param   device  the number of the ON-OFF device to modify
```

```
     * @param   value   the ON/OFF value
     */
    void changeStatus(int device, boolean value)
        throws NotOnOffDeviceException;

    /**
     * changes the status of a continuous value device.
     *
     * @param   device  the number of the device to modify
     * @param   value   the value to assign to the device
     */
    void changeStatus(int device_nb, short value)
        throws NotValueDeviceException;

    /**
     * returns all the devices of the Linet network.
     */
    LinetNode[] getNetwork();

    /**
     * return sthe specified device.
     */
    LinetNode getDevice(int device);
}
```

D.3.2 LinetNode.java

```
package driver.linet.service;

import driver.linet.implementation.LinetPacketEntry;

/**
 * LinetEntry.java
 *
 * @author Pierre Moermans
 */

public interface LinetNode
{
    public static final int ENTRY_LENGTH = 4;
    public static final byte FLAG_CHANGE = 1;

    //  Group types

    public static final byte TYPE_NO_GROUP      = 0;
    public static final byte TYPE_TOGGLE        = 1;
    public static final byte TYPE_DIMMER        = 2;
    public static final byte TYPE_IO_NODE       = 3;
    public static final byte TYPE_EXTENDED_LAMP = 4;
    public static final byte TYPE_LAMP_MONITOR  = 5;
    public static final byte TYPE_DELAY_TIMER   = 6;
    public static final byte TYPE_WAITER_CALL   = 7;
    public static final byte TYPE_DATA_EXCHANGE = 8;
    public static final byte TYPE_DATA_8        = 9;
    public static final byte TYPE_DATA_12       = 10;
    public static final byte TYPE_DATA_16       = 11;
    public static final byte TYPE_ADSTATE       = 12;

    public static final byte NUM_GROUPTYPES     = 13; // keep updated!
```

```
/**
 * return true if the node is to be shown.
 */
boolean shownGroupType();

/**
 * return the group type.
 */
byte getGroupType();

/**
 * set the group type.
 * return true if the ...
 */
boolean setGroupType(byte byteVal);

/**
 * return true if the node is to be updated.
 */
boolean changeMode();

/**
 * set the node update mode to "change".
 */
void setChangeMode();

/**
 * set the node update mode to "no change".
 * this mode is a simple request then ...
 */
void setNormalMode();

/**
 * return the node value.
 */
short get16bitValue();

/**
 * set the node value.
 */
void set16bitValue(short newVal);

/**
 * return the node boolean value.
 */
boolean getBooleanValue();

/**
 * set the node value as a boolean
 */
void setBooleanValue(boolean newVal);

/**
 * get the "row" data of the node
 */
byte[] getRawData();

/**
 * set the node raw data.
 * return true if the newEntry has the right length.
 */
```

```
    boolean setRawData(byte[] newEntry);

    /**
     * return true if the node is a on/off input node
     */
    boolean onoffInput();

    /**
     * return true if the node is a on/off output node
     */
    boolean onoffOutput();

    /**
     * return the String representation of te node
     */
    String toString();

    /**
     * returns true if the two nodes are equal
     */
    boolean equals(LinetPacketEntry lpe);
}
```

D.4 Implementation

D.4.1 FIFO.java

```
package driver.linet.implementation;

/**
 * ObjectFIFO.java
 *
 * @author Paul Hyde - Adaptation by Pierre Moermans
 */

public class FIFO
{
    private Object[] queue;
    private int capacity;
    private int size;
    private int head;
    private int tail;

    public FIFO(int cap)
    {
        capacity = ( cap > 0 ) ? cap : 1; // at least 1
        queue = new Object[capacity];

        head = 0;
        tail = 0;
        size = 0;
    }

    public int getCapacity()
    {
        return capacity;
    }

    public synchronized int getSize()
```

```
{
    return size;
}

public synchronized boolean isEmpty()
{
    return size == 0;
}

public synchronized boolean isFull()
{
    return size == capacity;
}

public synchronized void add(Object obj)
    throws InterruptedException
{
    waitWhileFull();

    queue[head] = obj;
    head = (head + 1) % capacity;
    size++;

    notifyAll(); // let any waiting threads know about change
}

public synchronized void addEach(Object[] list)
    throws InterruptedException
{

    //
    // You might want to code a more efficient
    // implementation here ... (see ByteFIFO.java)
    //

    for (int i = 0; i < list.length; i++)
        add(list[i]);
}

public synchronized Object remove()
    throws InterruptedException
{

    waitWhileEmpty();

    Object obj = queue[tail];
    queue[tail] = null;        // announce garbage

    tail = (tail + 1) % capacity;
    size--;

    notifyAll();     // let any waiting threads know about change
    return obj;
}

public synchronized Object[] removeAll()
    throws InterruptedException
{

    //
    // You might want to code a more efficient
    // implementation here ... (see ByteFIFO.java)
```

```
        //

        Object[] list = new Object[size]; // use the current size

        for (int i = 0; i < list.length; i++)
            list[i] = remove();

        // if FIFO was empty, a zero-length array is returned
        return list;
    }

    public synchronized Object[] removeAtLeastOne()
        throws InterruptedException
    {
        waitWhileEmpty(); // wait for a least one to be in FIFO
        return removeAll();
    }

    public synchronized boolean waitUntilEmpty(long msTimeout)
        throws InterruptedException
    {
        if (msTimeout == 0L)
        {
            waitUntilEmpty();  // use other method
            return true;
        }

        // wait only for the specified amount of time

        long endTime = System.currentTimeMillis() + msTimeout;
        long msRemaining = msTimeout;

        while (!isEmpty() && msRemaining > 0L)
        {
            wait(msRemaining);
            msRemaining = endTime - System.currentTimeMillis();
        }

        // May have timed out, or may have met condition,
        // calc return value.
        return isEmpty();
    }

    public synchronized void waitUntilEmpty()
        throws InterruptedException
    {
        while (!isEmpty())
            wait();
    }

    public synchronized void waitWhileEmpty()
        throws InterruptedException
    {
        while (isEmpty())
            wait();
    }

    public synchronized void waitUntilFull()
        throws InterruptedException
    {
        while (!isFull())
            wait();
```

```
    }

    public synchronized void waitWhileFull()
        throws InterruptedException
    {
        while (isFull())
            wait();
    }
}
```

D.4.2 LinetImpl.java

```
package driver.linet.implementation;

import driver.linet.service.*;
import driver.linet.event.*;
import driver.linet.exception.*;

import java.net.*;
import java.util.Vector;

/**
 * LinetImpl.java
 *
 * @author Pierre Moermans
 */

public class LinetImpl implements Linet, Runnable
{
    private static final int RESPONSE_TIMEOUT = 5000; // in msec
    private static final int POLL_INTERVAL = 500;  // in msec
    private UDPWriter uw;
    private UDPListener ul;
    private LinetPacket lp;
    private DatagramSocket ds;
    private boolean poll;
    private Thread th;

    private Vector listeners;

    public LinetImpl(String address, int port) {
        try {
            ds = new DatagramSocket();
        } catch (SocketException e) {
            System.out.println(
                "LinetImpl: unable to open Datagram socket");
        }
        uw = new UDPWriter(ds, address , port);
        ul = new UDPListener(ds);
        lp = new LinetPacket();
        listeners = new Vector(4);
    }

    public void run() {
        LinetPacket lpnew = new LinetPacket();
        while (poll) {
            System.out.println("polling ...");
            lp.getHeader().setPacketType(
                LinetPacketHeader.STATUS_REQUEST);
            if (uw.write(lp.getRawData())) {
```

```
            // we have written to the net
            long deadline =
                System.currentTimeMillis() +
                RESPONSE_TIMEOUT;
            while (ul.receiveBufferEmpty() &&
                   System.currentTimeMillis() < deadline)
                ;  // wait
            if (!ul.receiveBufferEmpty())
            {
                lpnew.setRawData(ul.read().getData());
                if (lpnew.getHeader().getPacketType() ==
                    LinetPacketHeader.STATUS_RESPONSE) {
                    // here comes the interesting code !!!
                    analyse(lpnew);
                }
            }
        }
        try {
            Thread.sleep(POLL_INTERVAL);
        } catch (InterruptedException e) {
            // do nothing
        }
    }
}

/**
 *
 */
private void analyse(LinetPacket lpnew) {
    LinetPacketEntry lpeold, lpenew;
    for (int i = 0; i < LinetPacket.MAX_GROUPS; i++) {
        lpeold = lp.getRecord(i);
        lpenew = lpnew.getRecord(i);
        if (! lpenew.equals(lpeold))
            fireLinetEvent(
                new LinetEvent(i, lpenew.get16bitValue()));
    }
    lp.setRawData(lpnew.getRawData());
}

/**
 * starts to poll the linet controller
 */
public void poll() {
    poll = true;
    th = new Thread(this);
    th.start();
}

/**
 * stop polling the linet controller
 */
public void stopPolling() {
    poll = false;
}

/**
 * Removes a LinetEvent listener.
 *
 * @param   listener    A linet event listener.
 */
public void removeListener(LinetEventListener listener) {
```

```
        listeners.remove(listener);
    }

    /**
     * changes the status of a linet ON-OFF device.
     *
     * @param   device  the number of the ON-OFF device to modify
     * @param   value   the ON/OFF value
     */
    public void changeStatus(int device, boolean value)
        throws NotOnOffDeviceException {
        LinetPacketEntry lpe = lp.getRecord(device);
        if (! lpe.onoffInput())
            throw new NotOnOffDeviceException(device);
        lpe.setBooleanValue(value);
    }

    /**
     * changes the status of a continuous value device.
     *
     * @param   device  the number of the device to modify
     * @param   value   the value to assign to the device
     */
    public void changeStatus(int device_nb, short value)
        throws NotValueDeviceException {
        LinetPacketEntry lpe = lp.getRecord(device_nb);
        if (lpe.onoffInput())
            throw new NotValueDeviceException(device_nb);
        lpe.set16bitValue(value);
    }

    /**
     * Adds a LinetEvent listener.
     *
     * @param   listener    A linet event listener.
     */
    public void addListener(LinetEventListener listener) {
        listeners.add(listener);
    }

    /**
     * return the specified device.
     */
    public LinetNode getDevice(int device) {
        return (LinetNode) lp.getRecord(device);
    }

    /**
     * returns all the devices of the Linet network.
     */
    public LinetNode[] getNetwork() {
        LinetNode[] nodes = new LinetNode[LinetPacket.MAX_GROUPS];
        for (int i = 0; i < nodes.length; i++) {
            nodes[i] = (LinetNode)lp.getRecord(i);
        }
        return nodes;
    }

    /**
     * sends an event to every listener
     */
    private void fireLinetEvent(LinetEvent event) {
```

```
        LinetEventListener listener;
        System.out.println("firing event");
        for (int i = 0; i < listeners.size(); i++) {
            listener = (LinetEventListener) listeners.elementAt(i);
            listener.stateChanged(event);
        }
    }
}
```

D.4.3 LinetPacket.java

```
package driver.linet.implementation;

/**
 * LinetPacket.java
 *
 * @author Pierre Moermans
 *
 */

public class LinetPacket
{
    public static final int MAX_GROUPS = 200;

    public static final int FRAME_LENGTH =
    LinetPacketEntry.ENTRY_LENGTH * MAX_GROUPS +
    LinetPacketHeader.HEADER_LENGTH;

    private LinetPacketHeader   header = new LinetPacketHeader();
    private LinetPacketEntry[] entries = new LinetPacketEntry[MAX_GROUPS];

    //  Constructor - no arguments

    public LinetPacket()
    {
        for (int i = 0; i < MAX_GROUPS; i++)
            entries[i] = new LinetPacketEntry();
    }

    public int getNumberOfGroups() {
        int iCount = 0;
        for (int i = 0; i < MAX_GROUPS; i++ )
            if (getRecord(i).shownGroupType())
                iCount++;
        return iCount;
    }

    // Return a String array presentation of active groups

    public String[] getActiveGroupStatusList() {
        int i;
        int iCount = 0;
        String[] list = new String[getNumberOfGroups()];
        for (i = 0; i < MAX_GROUPS; i++)
        {
            if (getRecord(i).shownGroupType())
                list[iCount++] = (i + 1) + " - " + getRecord(i).toString();
        }
        return list;
    }
```

```
// Return the packet header

public LinetPacketHeader getHeader() {
    return header;
}


// Return a record by index

public LinetPacketEntry getRecord(int recNum)
{
    return entries[recNum];
}


// Return the whole 804 byte packet as a byte array

public byte[] getRawData()
{
    byte[] rawdata = new byte[FRAME_LENGTH];
    byte[] temp;
    int i, j;
    int index = 0;

    temp = header.getRawData();

    for (i = 0; i < LinetPacketHeader.HEADER_LENGTH; i++)
        rawdata[index++] = temp[i];

    for (i = 0; i < MAX_GROUPS; i++)
    {
        temp = entries[i].getRawData();

        for (j = 0; j < LinetPacketEntry.ENTRY_LENGTH; j++)
            rawdata[index++] = temp[j];
    }

    return rawdata;
}


// Set header and all nodes from a byte array

public boolean setRawData(byte[] rawData)
{
    byte[] temp;
    int i, j;
    int index = 0;

    if (rawData.length != FRAME_LENGTH)
        return false;

    temp = new byte[LinetPacketHeader.HEADER_LENGTH];

    for (i = 0; i < LinetPacketHeader.HEADER_LENGTH; i++)
        temp[index++] = rawData[i];

    header.setRawData(temp);

    for (i = 0; i < MAX_GROUPS; i++)
```

```
        {
            temp = new byte[LinetPacketEntry.ENTRY_LENGTH];

            for (j = 0; j < LinetPacketEntry.ENTRY_LENGTH; j++)
                temp[j] = rawData[index++];

            getRecord(i).setRawData(temp);
        }

        return true;
    }
}
```

D.4.4 LinetPacketEntry.java

```
package driver.linet.implementation;

import driver.linet.service.*;

/**
 * LinetDeviceImpl.java
 *
 * @author Pierre Moermans
 */

public class LinetPacketEntry implements LinetNode
{


    private byte[] entryData = new byte[ENTRY_LENGTH];


    //  Return true for packet types to show on list

    public boolean shownGroupType()
    {
        switch (getGroupType())
        {
            case TYPE_TOGGLE:
            case TYPE_DIMMER:
            case TYPE_IO_NODE:
            case TYPE_LAMP_MONITOR:
            case TYPE_DELAY_TIMER:
            case TYPE_DATA_8:
            case TYPE_DATA_12:
            case TYPE_DATA_16:
            case TYPE_ADSTATE:
                return true;
        }

        return false;
    }


    // Methods for handling group types

    public byte getGroupType()
    {
        return entryData[0];
    }
```

```
public boolean setGroupType(byte byteVal)
{
    if (byteVal >= NUM_GROUPTYPES)
        return false;

    entryData[0] = byteVal;
    return true;
}


// Change mode flag handling methods

public boolean changeMode()
{
    return (entryData[1] & FLAG_CHANGE) != 0;
}

public void setChangeMode()
{
    entryData[1] = (byte)((int)entryData[1] | (int)FLAG_CHANGE);
}

public void setNormalMode()
{
    entryData[1] = (byte)((int)entryData[1] & ~(int)FLAG_CHANGE);
}


// Methods for reading and setting group values

public short get16bitValue()
{
    return (short)(entryData[2] * 0x100 + (entryData[3] & 0xFF));
}

public void set16bitValue(short newVal)
{
    entryData[2] = (byte)(newVal >> 8);
    entryData[3] = (byte)(newVal & 0xFF);
}

public boolean getBooleanValue()
{
    return get16bitValue() != 0;
}

public void setBooleanValue(boolean newVal)
{
    set16bitValue((short)(newVal ? 1 : 0));
}


// Raw byte array data handling methods

public byte[] getRawData()
{
    return entryData;
}

public boolean setRawData(byte[] newEntry)
{
```

```
        if  (newEntry.length != ENTRY_LENGTH)
            return false;

        entryData = newEntry;
        return true;
    }


    //  Select groups with on/off inputs

    public boolean onoffInput()
    {
        switch (getGroupType())
        {
            case TYPE_TOGGLE:
            case TYPE_DIMMER:
            case TYPE_IO_NODE:
            case TYPE_EXTENDED_LAMP:
            case TYPE_LAMP_MONITOR:
            case TYPE_DELAY_TIMER:
                return true;
        }

        return false;
    }


    //  Select groups with on/off outputs

    public boolean onoffOutput()
    {
        switch (getGroupType())
        {
            case TYPE_TOGGLE:
            case TYPE_DIMMER:
            case TYPE_IO_NODE:
            case TYPE_EXTENDED_LAMP:
            case TYPE_LAMP_MONITOR:
                return true;
        }

        return false;
    }


    // String representation of the group types

    private static String onoffString(boolean val)
    {
        if  (val)
            return "on";

        return "off";
    }


    public String toString()
    {
        String[] groupTypeNames =
        {"no group", "toggle", "dimmer", "I/O node", "extended lamp",
        "lamp monitor", "delay timer", "waiter call", "data exchange",
        "data8", "data12", "data16", "AD/state"
```

```
        };

        String ret = groupTypeNames[getGroupType()] + " - ";

        if (onoffOutput())
            return ret + onoffString(get16bitValue() != 0);

        return ret + get16bitValue();
    }

    public boolean equals(LinetPacketEntry lpe) {
        if ((getGroupType() != lpe.getGroupType()) ||
            (get16bitValue() != lpe.get16bitValue()))
            return false;
        return true;
    }
}
```

D.4.5 LinetPacketHeader.java

```
package driver.linet.implementation;

/**
 * LinetPacketHeader.java
 *
 * @author Pierre Moermans
 */

public class LinetPacketHeader
{
    public static final int HEADER_LENGTH = 4;
    public static final byte VERSION_MAX = 0;

    // Packet types

    public static final byte STATUS_REQUEST     = 0;
    public static final byte STATUS_RESPONSE    = 1;
    public static final byte STRUCTURE_REQUEST  = 2;
    public static final byte STRUCTURE_RESPONSE = 3;

    public static final byte NUM_PACKETTYPES    = 4;

    public static final byte UNKNOWN = (byte) 255;

    private byte[] headerData = {VERSION_MAX, UNKNOWN, 0, 0};

    //  Data access methods

    public byte getVersion()
    {
        return headerData[0];
    }

    public byte getPacketType()
    {
        return headerData[1];
    }

    public boolean setVersion(byte byteVal)
    {
```

```
        if (byteVal > VERSION_MAX)
            return false;

        headerData[0] = byteVal;
        return true;
    }

    public boolean setPacketType( byte byteVal )
    {
        if (byteVal >= NUM_PACKETTYPES)
            return false;

        headerData[1] = byteVal;
        return true;
    }

    //  Raw byte string handling methods

    public byte[] getRawData()
    {
        return headerData;
    }

    public boolean setRawData(byte[] newHeader)
    {
        if (newHeader.length != HEADER_LENGTH)
            return false;

        headerData = newHeader;
        return true;
    }
}
```

D.4.6 UDPListener.java

```
package driver.linet.implementation;

import java.net.*;
import java.io.*;

/**
 * UDPListener.java
 *
 * @author Tieto-Laurila Oy, Lasse Laurila 2000
 *
 */

public class UDPListener
{
    private final int BUFFER_SIZE = 1024;
    private final int RECEIVE_BUFFER_SIZE = 256;

    private DatagramSocket sd = null;

    private volatile boolean noStopRequested;
    private Thread internalThread;

    private FIFO receiveBuffer = new FIFO(RECEIVE_BUFFER_SIZE);
```

```
public UDPListener(DatagramSocket my_sd)
{
    sd = my_sd;
    noStopRequested = true;

    Runnable r = new Runnable()
    {
        public void run()
        {
            try
            {
                runWork();
            }
            catch (Exception e)
            {
            }
        }
    };

    internalThread = new Thread(r);
    internalThread.start();
}


public void runWork()
{
    DatagramPacket packet = null;

    try
    {
        while (noStopRequested)
        {
            // create a datagram packet to receive message in

            byte[] buffer = new byte[BUFFER_SIZE];
            packet = new DatagramPacket(buffer, BUFFER_SIZE);

            sd.receive(packet);  // wait for message from client

            // forward data to dataReceived() method

            dataReceived(packet.getData(), packet.getLength(),
                         packet.getAddress(), packet.getPort());
        }

    }
    catch (Exception e)
    {
        /*  This (omitted) exception will occur when close() method
         is called from stopRequest() method. Not very pretty,
         but as far as I know, this is the only way to break out
         of an blocked I/O state... */
    }
}


private void dataReceived(byte[] data, int length, InetAddress source,
                          int port)
{
    byte[] d = new byte[length];

    for (int i = 0; i < length; i++)
```

```
            d[i] = data[i];

        UDPPacket p = new UDPPacket(d, source, port);

        try
        {
            if (!receiveBuffer.isFull())  // drop overflow
                receiveBuffer.add(p);
        }
        catch (InterruptedException e)
        {
            stopRequest();
        }
    }


    public int receiveBufferSize()
    {
        return receiveBuffer.getSize();
    }


    public boolean receiveBufferEmpty()
    {
        return receiveBuffer.isEmpty();
    }


    public UDPPacket read()
    {
        UDPPacket ret = null;

        try
        {
            ret = (UDPPacket)receiveBuffer.remove();
        }
        catch (InterruptedException e)
        {
            stopRequest();
        }

        return ret;
    }


    public void stopRequest()
    {
        noStopRequested = false;
        internalThread.interrupt();

        sd.close();
        sd = null;
    }


    public boolean isAlive()
    {
        return internalThread.isAlive();
    }


    protected void finalize()
```

```
    {
        stopRequest();
    }
}
```

D.4.7 UDPPacket.java

```
package driver.linet.implementation;

import java.net.*;

/**
 * UDPPacket.java
 *
 * @author Tieto-Laurila Oy, Lasse Laurila 2000
 */

public class UDPPacket
{
    private byte[] contents;
    private InetAddress source;
    private int port;

    public UDPPacket(byte[] data, InetAddress src, int srcPort)
    {
        contents = data;
        source = src;
        port = srcPort;
    }

    public byte[] getData()
    {
        return contents;
    }

    public void setData(byte[] data)
    {
        contents = data;
    }

    public InetAddress getSource()
    {
        return source;
    }

    public void setSource(InetAddress src)
    {
        source = src;
    }

    public int getSourcePort()
    {
        return port;
    }

    public void setSourcePort(int srcPort)
    {
        port = srcPort;
    }
}
```

D.4.8 UDPWriter.java

```
package driver.linet.implementation;

import java.net.*;

/**
 * UDPWriter.java
 *
 * @author Tieto-Laurila Oy, Lasse Laurila 2000
 */

public class UDPWriter
{
    private String addr = "localhost";
    private int port = 0;
    private DatagramSocket sd = null;

    public UDPWriter(DatagramSocket sd)
        {
        this.sd = sd;
        }

    public UDPWriter(DatagramSocket sd, String host)
        {
        this.sd = sd;
        addr = host;
        }

    public UDPWriter(DatagramSocket sd, String host, int port)
        {
        this.sd = sd;
        addr = host;
        this.port = port;
        }


    public boolean write(byte[] data, String host, int port)
        {
        /* final int BUFFER_SIZE = 1024; */

        try
            {
            byte[] buffer = new byte[data.length];

            for (int i = 0; i < data.length; i++)
                buffer[i] = data[i];

            sd.send(new DatagramPacket(buffer, buffer.length,
                                       InetAddress.getByName(host), port));

            return true;
            }
        catch (Exception e)
            {
            return false;
            }
        }
```

```
    public boolean write(byte[] data)
        {
        return write(data, addr, port);
        }
}
```

D.5 Activator LinetActivator.java

```
package driver.linet;

import org.osgi.framework.*;
import driver.linet.implementation.*;
import driver.linet.service.*;
import java.util.Properties;

/**
 * LinetActivator.java
 *
 * @author Pierre Moermans
 */

public class LinetActivator implements BundleActivator
{
    private LinetImpl[] linets;

    // to be updated if several controllers are needed
    private String[] addrs = {"130.233.152.193"};
    private int[] ports = {1313};

    public void start(BundleContext ctxt) {
        System.out.println("starting Linet bundle");
        linets = new LinetImpl[addrs.length];
        for (int i = 0; i < addrs.length; i++)
        {
            linets[i] = new LinetImpl(addrs[i], ports[i]);
            linets[i].poll();
            Properties props = new Properties();
            props.put("net address", addrs[i]);
            props.put("net port", new Integer(ports[i]));
            ctxt.registerService(
                "driver.linet.service.Linet", linets[i], props);
        }
        System.out.println("bundle started");
    }

    public void stop(BundleContext ctxt) {
        System.out.println("stopping linet bundle");
        for (int i = 0; i < addrs.length; i++)
        {
            linets[i].stopPolling();
            linets[i] = null;
        }
        System.out.println("bundle stopped");
    }
}
```

D.6 Manifest

```
Bundle-Activator: driver.linet.LinetActivator
Export-Package: driver.linet.service, driver.linet.exception, driver.linet.event
```

Appendix E

X10 network bundle *x10Serial*

It must be noted that the X10 network has been the first one to be developped. As explained earlier, this network has been abandonned. The code of the serialManagement bundle has completely changed since that development. As the network hasen't been used further, the code hasn't been kept up-to-date and presents incompatibilities. The package structure of this bundle is completely different from the other ones and doesn't present any sub-packages.

E.1 X10Address.java

```
package driver.x10;

import java.io.Serializable;

public class X10Address implements Serializable {

    private byte houseCode;
    private byte deviceCode;

    public X10Address(byte houseCode, byte deviceCode) {
    this.houseCode = houseCode;
    this.deviceCode = deviceCode;
    }

    public byte getHouseCode() {return houseCode;}

    public byte getDeviceCode() {return deviceCode;}

    public void setHouseCode(byte houseCode) {this.houseCode = houseCode;}

    public void setDeviceCode(byte deviceCode) {this.deviceCode = deviceCode;}
}
```

E.2 X10Protocol.java

```
package driver.x10;

public abstract class X10Protocol
{
    public static final byte HOUSE_A = 96;
    public static final byte HOUSE_B = -32;
    public static final byte HOUSE_C = 32;
    public static final byte HOUSE_D = -96;
    public static final byte HOUSE_E = 16;
    public static final byte HOUSE_F = -112;
    public static final byte HOUSE_G = 80;
    public static final byte HOUSE_H = -48;
    public static final byte HOUSE_I = 112;
    public static final byte HOUSE_J = -16;
    public static final byte HOUSE_K = 48;
    public static final byte HOUSE_L = -80;
    public static final byte HOUSE_M = 0;
    public static final byte HOUSE_N = -128;
    public static final byte HOUSE_O = 64;
    public static final byte HOUSE_P = -64;

    public static final byte DEVICE_1 = 6;
    public static final byte DEVICE_2 = 14;
    public static final byte DEVICE_3 = 2;
    public static final byte DEVICE_4 = 10;
    public static final byte DEVICE_5 = 1;
    public static final byte DEVICE_6 = 9;
    public static final byte DEVICE_7 = 5;
    public static final byte DEVICE_8 = 13;
    public static final byte DEVICE_9 = 7;
    public static final byte DEVICE_10 = 15;
    public static final byte DEVICE_11 = 3;
    public static final byte DEVICE_12 = 11;
    public static final byte DEVICE_13 = 0;
    public static final byte DEVICE_14 = 8;
    public static final byte DEVICE_15 = 4;
    public static final byte DEVICE_16 = 12;

    public static final byte ALL_UNIT_OFF = 0;
    public static final byte ALL_LIGHTS_ON = 1;
    public static final byte ON = 2;
    public static final byte OFF = 3;
    public static final byte DIM = 4;
    public static final byte BRIGHT = 5;
    public static final byte ALL_LIGHTS_OFF = 6;
    public static final byte EXTENDED_CODE = 7;
    public static final byte HAIL_REQUEST = 8;
    public static final byte HAIL_ACKNOWLEDGE = 9;
    public static final byte PRE_SET_DIM_1 = 10;
    public static final byte PRE_SET_DIM_2 = 11;
    public static final byte EXTENDED_DATA_TRANSFERT = 12;
    public static final byte STATUS_ON = 13;
    public static final byte STATUS_OFF = 14;
    public static final byte STATUS_REQUEST = 15;

    public static final byte UPLOAD_REQUEST = 90;
    public static final byte UPLOAD_READY = -61;

    public static final byte POWER_FAIL_REQUEST = -91;
    public static final byte POWER_FAIL_READY = -5;
```

```
    public static final byte CM11_READY = 85;

    public static final byte TRANS_CORRECT = 0;
    public static final byte SEND_ADDRESS = 4;
    public static final byte SEND_FUNCTION = 6;
}
```

E.3 X10Serial.java

```
package driver.x10;

import gnu.io.*;
import driver.x10.*;

public interface X10Serial
{
    public void addEventListener(X10SerialEventListener l);

    public void removeEventListener(X10SerialEventListener l);

    public void sendCommand(X10Address addr, byte function);
}
```

E.4 X10SerialImpl.java

```
package driver.x10;

import java.io.*;
import gnu.io.*;
import driver.device.serial.*;
import java.util.*;
import util.base.*;


public class X10SerialImpl implements X10Serial, SerialPortEventListener
{
    private static final int UNKNOWN = 0;
    private static final int MACRO_REQUEST = 1;
    private static final int INFO_UPLOAD = 2;

    private SerialPort port;
    private InputStream input = null;;
    private OutputStream output = null;
    // stores the bytes received from the serial port
    private byte inputBuffer[];

    private int readStatus;
    private boolean acknowledged;
    private byte ack;

    private BaseConverter bc;
    private Vector listeners;

    public X10SerialImpl(SerialService ss) {
    System.setSecurityManager(null);
    this.port = ss.getPort();

    // initialize the serial port
```

```
try {
    port.setSerialPortParams(4800,SerialPort.DATABITS_8,
                             SerialPort.STOPBITS_1,
                             SerialPort.PARITY_NONE);
} catch (UnsupportedCommOperationException  e) {
    System.out.println(
        "X10Serial: Unable to set port parameters");
    return;
} // end of try-catch

try {
    input = port.getInputStream();
    output = port.getOutputStream();
} catch ( IOException e) {
    System.out.println(
        "X10Serial: unable to get streams from serial port");
    // there should be a way to
    // unregister the service right away
} // end of try-catch
ss.addEventListener(this);
// the CM11 never transmits more than 10 bytes at a time
inputBuffer = new byte[10];
bc = new BaseConverter();
listeners = new Vector(4);
acknowledged = false;
}



// in a first time (and because of time shortness,
// i'll not worry about input/output concurrency
// but it has to be done later since there is no way
// to send information to the CM11 while it's trying
// to upload its buffer to the computer

public synchronized void serialEvent(SerialPortEvent event) {
System.out.println("X10Serial: received serial event");
try {
    while ( input.available() > 0) {
    //System.out.println("X10Serial: byte(s) available");
    int nbBytes = input.read(inputBuffer);

    // in a first time, directly process the bytes
    // after, two theads should be implemented (producer-consumer)

    //System.out.println("X10Serial: received " + nbBytes + " bytes");
    if ( nbBytes == 1 ) {
        // we initiate a macro-request or an information uploading
        if ( inputBuffer[0] == X10Protocol.POWER_FAIL_REQUEST) {
        System.out.println("X10Serial: Power-fail Request");
        readStatus = MACRO_REQUEST;
        sendMacro();
        System.out.println("X10Serial: Sent macro");
        }
        else if ( inputBuffer[0] == X10Protocol.UPLOAD_REQUEST ) {
        System.out.println("X10Serial: Upload Request");
        readStatus = INFO_UPLOAD;
        // the next read, we now that it is an info uploading
        send1(X10Protocol.UPLOAD_READY);
        // the next event will read the data
        } // end of if - else
        else if ( inputBuffer[0] == X10Protocol.CM11_READY) {
```

```
            // the CM11 is ready (just do nothing)
            System.out.println("CM11 ready");
            } // end of if ()
            else  {
            // we receive a checksum
            ack = inputBuffer[0];
            acknowledged = true;
            } // end of else

        }
        else {
            // we have several bytes so it should be a info uploading
            if ( readStatus == INFO_UPLOAD ) {
            System.out.println("X10Serial: receiving info");
            processInfo(inputBuffer);
            readStatus = UNKNOWN;
            } // end of if ()
            else {
            System.out.println("X10Serial: receiving unknown bytes!");
            } // end of if ()else

        } // end of if - else
        } // end of while ()
    } catch ( IOException e) {
        System.out.println(
            "X10Serial: An error occured while reading");
    } // end of try-catch
    }


    private  boolean send1(byte b) {
    byte buf[] = new byte[1];
    buf[0] = b;
    try {
        output.write(b);
    } catch ( IOException e) {
        System.out.println(
            "X10Serial: An error occured while writing single byte: "
                + bc.byte2binaryString(b));
        return false;
    } // end of try-catch
    return true;
    }

    private  boolean send(byte[] buf) {
    try {
        output.write(buf);
    } catch ( IOException e) {
        System.out.println(
            "X10Serial: An error occured while writing bytes. ");
        return false;
    } // end of try-catch
    return true;
    }

    private  byte recv1() {
    byte[] buf = new byte[1];
    try {
        input.read(buf);
    } catch ( IOException e) {
        e.printStackTrace();
    } // end of try-catch
```

```
return buf[0];
}


private  void processInfo(byte[] buffer) {
// here we manage the stuff
X10SerialEvent event =
        new X10SerialEvent(X10SerialEvent.INFO, buffer);
fireEvent(event);
}


// the sendMacro method is still a trick for the moment !
public  void sendMacro() {
byte[] macroBuffer = new byte[7];
    macroBuffer[0] = -101;
    macroBuffer[1] = 57;
    macroBuffer[2] = 7;
    macroBuffer[3] = 1;
    macroBuffer[4] = 44;
    macroBuffer[5] = 16;
    macroBuffer[6] = 96;
    try
        {
            output.write(macroBuffer);
        }
    catch (IOException e) {}
    return;
}

public  void sendCommand(X10Address addr, byte function) {
send1((byte)0);
try {
    Thread.sleep(500);
} catch ( InterruptedException e) {

} // end of try-catch

    for ( int cnt = 0; cnt < 3; cnt ++ ) {
    System.out.println("X10Serial: sending command");
    boolean transmitionCorrect = false;
    byte[] byteAddress = new byte[2];
    byteAddress[0] = X10Protocol.SEND_ADDRESS;
    byteAddress[1] = (byte) (addr.getHouseCode() +
                                addr.getDeviceCode());
    System.out.println("X10Serial: sending "
                           + bc.byte2binaryString(
                               byteAddress[0]));
    System.out.println("X10Serial: sending "
                           + bc.byte2binaryString(
                               byteAddress[1]));
    byte cksum = (byte) (byteAddress[0] +
                             byteAddress[1]);
    int counter = 0;
    while ( (! transmitionCorrect) && counter < 4) {
    System.out.println("X10Serial: sending address");
    port.notifyOnDataAvailable(false);
    send(byteAddress);
    ack = recv1();
    if ( ack != cksum ) {
        System.out.println("X10Serial: checksum error");
        counter++;
```

```
            continue;
        } // end of if ()
        send1(X10Protocol.TRANS_CORRECT);
        port.notifyOnDataAvailable(true);
        transmitionCorrect = true;
        } // end of while ()

        transmitionCorrect = false;
        byte[] byteFunction = new byte[2];
        byteFunction[0] = X10Protocol.SEND_FUNCTION;
        byteFunction[1] = (byte) (addr.getHouseCode() + function);
        System.out.println("X10Serial: sending " +
                           bc.byte2binaryString(
                               byteFunction[0]));
        System.out.println("X10Serial: sending " +
                           bc.byte2binaryString(
                               byteFunction[1]));
        cksum = (byte) (byteFunction[0] + byteFunction[1]);
        while ( ! transmitionCorrect ) {
        System.out.println("X10Serial: sending function");
        port.notifyOnDataAvailable(false);
        send(byteFunction);
        ack = recv1();
        if ( ack != cksum ) {
            System.out.println("X10Serial: checksum error");
            counter++;
            continue;
        } // end of if ()
        send1(X10Protocol.TRANS_CORRECT);
        port.notifyOnDataAvailable(true);
        transmitionCorrect = true;
        } // end of while ()
        System.out.println("X10Serial: command sent");
        try {
        Thread.sleep(1500);
        } catch ( InterruptedException e) {

        } // end of try-catch

        } // end of for ()
    }


    public void addEventListener(X10SerialEventListener l) {
    System.out.println("X10Serial: adding client");
    listeners.addElement(l);
    }

    public void removeEventListener(X10SerialEventListener l) {
    listeners.removeElement(l);
    }

    private  void fireEvent(X10SerialEvent event) {
    Enumeration enum = listeners.elements();
    while (enum.hasMoreElements()) {
        //      System.out.println("X10Serial: sending event");
  ((X10SerialEventListener)enum.nextElement()).x10SerialEvent(event);
    } // end of while ()

    }
}
```

E.5 X10SerialActivator.java

```
package driver.x10;

import org.osgi.framework.*;
import java.util.*;
import driver.device.serial.*;
import java.io.*;

public class X10SerialActivator implements BundleActivator
{
    X10Serial xserial;
    ServiceReference[] ref = null;


    public void start(BundleContext ctxt) {
    System.out.println("Starting X10Serial");
    try {
        ref = ctxt.getServiceReferences(
                "hut.device.serial.SerialService",
                "(port=/dev/ttyS1)");
    } catch ( InvalidSyntaxException e) {
        System.out.println("A syntaxic error occured");
        return;
    } // end of try-catch

    if ( ref == null ) {
        // by now, it is the responsability of the
        // manager to restart the service later.
        System.out.println(
            "There is no SerialService registered for port /dev/ttyS0");
        System.out.println("Launch this service and restart this on after");
    } // end of if ()
    SerialService ss = (SerialService) ctxt.getService(ref[0]);
    // there is only one SerialService with port /dev/ttyS0

    // Registering the SerialComm service
    Properties props = new Properties();
    props.setProperty("description", "X10Serial service");
    xserial = new X10SerialImpl(ss);
    ctxt.registerService("hut.x10.X10Serial", xserial, props);
    }

    public void stop(BundleContext ctxt) {
    System.out.println("Stopping X10Serial service");
    }
}
```

E.6 X10Address.java

```
package driver.x10;

import java.io.Serializable;

public class X10Address implements Serializable {
```

```
    private byte houseCode;
    private byte deviceCode;

    public X10Address(byte houseCode, byte deviceCode) {
    this.houseCode = houseCode;
    this.deviceCode = deviceCode;
    }

    public byte getHouseCode() {return houseCode;}

    public byte getDeviceCode() {return deviceCode;}

    public void setHouseCode(byte houseCode) {this.houseCode = houseCode;}

    public void setDeviceCode(byte deviceCode) {this.deviceCode = deviceCode;}
}
```

E.7 X10Serialvent.java

```
package driver.x10;

public class X10SerialEvent
{
    public static final int POWER_FAIL_REQUEST = 0;
    public static final int INFO = 1;

    private byte[] buffer;
    private int type;

    public X10SerialEvent(int type, byte[] buffer) {
    this.type = type;
    this.buffer = buffer;
    }

    public int getType() { return type; }

    public byte[] getBuffer() { return buffer; }
}
```

E.8 X10SerialEventListener.java

```
package driver.x10;

public interface X10SerialEventListener
{
    public void x10SerialEvent(X10SerialEvent event);
}
```

E.9 X10SerialManifest.java

```
Bundle-Activator: driver.x10.X10SerialActivator
Import-Package: driver.device.serial
Export-Package: driver.x10
```

Appendix F

X10 RMI server bundle

F.1 X10RmiServerInterface.java

```
package driver.x10;

import java.rmi.*;

public interface X10RmiServerInterface extends Remote {
    public void addClient(X10RmiClientInterface xci)
        throws RemoteException;

    public void removeClient(X10RmiClientInterface xci)
        throws RemoteException;

    public void sendCommand(X10Address addr, byte function)
        throws RemoteException;
}
```

F.2 X10RmiClientInterface.java

```
package driver.x10;

import java.rmi.*;

public interface X10RmiClientInterface
    extends Remote {

    public void x10Event(X10Event event)
        throws RemoteException;

}
```

F.3 X10RmiServer.java

```
package driver.x10;

import java.rmi.*;
import java.rmi.server.*;

import java.util.*;
```

```
import util.base.BaseConverter;


public class X10RmiServer
    extends UnicastRemoteObject
    implements X10SerialEventListener,
    X10RmiServerInterface
{
    public static final String SERVERNAME =
        "X10RmiServer";
    private Vector clients;
    private X10Serial sc;
    private BaseConverter bc;

    private boolean receivedAddress = false;
    private X10Address addr;

    public X10RmiServer(X10Serial sc) throws RemoteException{
        clients = new Vector(4);
        this.sc = sc;
        this.sc.addEventListener(this);
        bc = new BaseConverter();
    }

    public synchronized void x10SerialEvent(X10SerialEvent event) {

        // We have received information so the steps are:
        //    - read how many bytes are meaningfull
        //      (value of first byte) (SKIPPED)
        //      We suppose there will only be 3 bytes meaningfull
        //       as we treat the information right away
        //    - the second byte is just says the third one
        //      is an address and the forth one is the data (SKIPPED)
        //    - read the address
        //    - read the data

        // Read the device-code
        byte[] buffer = event.getBuffer();
        if ( buffer[0] == 2 ) {
            // we have an address or a command
            // depending on the second byte
            if ( buffer[1] == 0) {
                // we have an address
                receivedAddress = true;
                //  15  = 00001111
                //  -16 = 11110000
                byte devCode = (byte) (buffer[2]&((byte)15));
                byte houseCode = (byte) (buffer[2]&((byte)-16));
                addr = new X10Address(houseCode, devCode);
            } // end of if ()

            else {
                // we have a function
                // if we have already received an address
                // process it, otherwise, drop everything
                if ( receivedAddress ) {
                    // get function
                    byte function = (byte) (buffer[2]&((byte)15));
                    // send event to all clients
                    receivedAddress = false;
                    fireX10Event(new X10Event(addr,function));
                } // end of if ()
```

```
            } // end of if ()else
        } // end of if ()
        else if ( buffer[0] == 3 ) {
            System.out.println("Got a 3 bytes meaningfull");
            // let's assume we have an address
            // followed by a function
            // ...
        } // end of if ()
    }


    private void fireX10Event(X10Event event) {
        Enumeration enum = clients.elements();
        X10RmiClientInterface srci = null;
        while ( enum.hasMoreElements()) {
            try {
                srci = (X10RmiClientInterface) enum.nextElement();
                srci.x10Event(event);
            } catch ( RemoteException e) {
                clients.removeElement(srci);
                System.out.println("SineRmiServer: client removed");
                e.printStackTrace();
            } // end of try-catch
        } // end of while ()

    }


    public void addClient(X10RmiClientInterface xci)
        throws RemoteException {
        clients.addElement(xci);
    }

    public void removeClient(X10RmiClientInterface xci)
        throws RemoteException {
        clients.removeElement(xci);
    }

    public void sendCommand(X10Address addr, byte function) {
        sc.sendCommand(addr, function);
    }
}
```

F.4 X10Event.java

```
package driver.x10;

import java.io.Serializable;

public class X10Event implements Serializable
{
    private X10Address address;
    private byte function;

    public X10Event(X10Address address, byte function) {
    this.address = address;
    this.function = function;
    }

    public X10Address getAddress() {return address;}
```

```
    public byte getFunction() {return function;}
}
```

F.5 X10RmiActivator.java

```
package driver.x10;

import driver.device.serial.*;
import org.osgi.framework.*;
import java.rmi.*;
import java.rmi.server.*;

public class X10RmiActivator implements BundleActivator
{
    X10RmiServer rmiServer;

    public void start(BundleContext ctxt) {
    try {
        System.out.println("starting X10 RMI Server");
        ServiceReference ref =
                ctxt.getServiceReference(
                "hut.x10.X10Serial");
        if (ref == null) {
        System.out.println(
                    "the rmi server  isn't registered");
        throw new Exception(
                    "no x10 serial service found");
        }
        X10Serial sc = (X10Serial) ctxt.getService(ref);
        rmiServer = new X10RmiServer(sc);

        Naming.rebind(X10RmiServer.SERVERNAME, rmiServer);
        System.out.println("X10RmiServer ready.");

    } catch ( Exception e) {
        System.out.println(
            "Binding of X10 rmi server failed.");
        e.printStackTrace();
    } // end of try-catch

    }

    public void stop(BundleContext ctxt) {
    try {
        System.out.println(
                "Stopping X10 rmi server.");
        Naming.unbind(X10RmiServer.SERVERNAME);
    } catch ( Exception e) {
        System.out.println(
            "Error occured while unbinding X10 rmi server.");
        e.printStackTrace();
    } // end of try-catch
    }
}
```

F.6 Manifest

```
Bundle-Activator: driver.x10.X10RmiActivator
Import-Package: driver.x10
Export-Package: driver.x10
```

Appendix G

Network independance bundles

G.1 First sub-level: interfaces and events

G.1.1 DeviceEvent.java

```
/**
 * DeviceEvent.java
 *
 * @author Peirre Moermans
 */

package stdNet.event;

public class DeviceEvent
{
    private int deviceNumber;
    private short oldValue;
    private short newValue;
    private boolean onOffDevice;

    public void setOldValue(short oldValue)
    {
        this.oldValue = oldValue;
    }

    public short getOldValue()
    {
        return oldValue;
    }

    public void setNewValue(short newValue)
    {
        this.newValue = newValue;
    }

    public short getNewValue()
    {
        return newValue;
    }
```

```
    public void setOnOffDevice(boolean onOffDevice)
    {
        this.onOffDevice = onOffDevice;
    }

    public boolean isOnOffDevice()
    {
        return onOffDevice;
    }

    public void setDeviceNumber(int deviceNumber)
    {
        this.deviceNumber = deviceNumber;
    }

    public int getDeviceNumber()
    {
        return deviceNumber;
    }


    public DeviceEvent(int deviceNumber,
                       boolean oldValue,
                       boolean newValue)
    {
        this.onOffDevice = true;
        if (oldValue)
            this.oldValue = 1;
        else this.oldValue = 0;
        if (newValue)
            this.newValue = 1;
        else this.newValue = 0;
        this.deviceNumber = deviceNumber;
    }

    public DeviceEvent(int deviceNumber,
                       short oldValue,
                       short newValue)
    {
        this.onOffDevice = false;
        this.oldValue = oldValue;
        this.newValue = newValue;
        this.deviceNumber = deviceNumber;
    }
}
```

G.1.2 DeviceEventListener.java

```
package stdNet.event;

/**
 * DeviceEventListener.java
 *
 * @author Pierre Moermans
 */

public interface DeviceEventListener
{
    /**
```

```
     * This method is called whenever a device value has changed.
     *
     * @param event An event containing the device number,
     *              the old and the new value.
     */
    public void valueChanged(DeviceEvent event);
}
```

G.1.3 StandardNet.java

```
package stdNet.service;

/**
 * StandardNet.java
 *
 * @author Pierre Moermans
 */

/**
 * This interface defines how to access a network device.
 */

import stdNet.event.*;

public interface StandardNet
{
    /**
     * Set the boolean <code>value</code> to a ON-OFF actor.
     *
     * @param device the device number on the network.
     * @param value the <code>boolean</code> value to assign.
     */
    public void setBooleanValue(int device, boolean value);

    /**
     * Set the short <code>value</code> to a ranged-value actor.
     *
     * @param device the device number on the network.
     * @param value the <code>short</code> value to assign.
     */
    public void setShortValue(int device, short value);

    /**
     * Get the boolean value from a on-off sensor/actor.
     *
     * @param device the number of the device on the network.
     * @return the boolean value of the device.
     */
    public boolean getBooleanValue(int device);

    /**
     * Get the short value from a ranged-value device.
     *
     * @param device the number of the device on the network.
     * @return the short value of the device.
     */
    public short getShortValue(int device);

    /**
```

```
     * register a listener.
     *
     * @param listener The listener to register.
     */
    public void addListener(DeviceEventListener listener);

    /**
     * unregister a listener.
     *
     * @param listener The listener to unregister.
     */
    public void removeListener(DeviceEventListener listener);

}
```

G.2 Second sub-level: interface and events

G.2.1 NetworkEvent.java

```
package homeStd.event;

/**
 * DeviceEvent.java
 *
 * @author Pierre Moermans
 */

import homeStd.service.DeviceID;

public class NetworkEvent
{
    private short oldValue;
    private short newValue;
    private boolean onOffDevice;

    private DeviceID devId;

    public void setOldValue(short oldValue)
    {
        this.oldValue = oldValue;
    }

    public short getOldValue()
    {
        return oldValue;
    }

    public void setNewValue(short newValue)
    {
        this.newValue = newValue;
    }

    public short getNewValue()
    {
        return newValue;
    }

    public void setOnOffDevice(boolean onOffDevice)
    {
```

```
        this.onOffDevice = onOffDevice;
    }

    public boolean isOnOffDevice()
    {
        return onOffDevice;
    }

    public void setDevId(DeviceID devId)
    {
        this.devId = devId;
    }

    public DeviceID getDevId()
    {
        return devId;
    }


    public NetworkEvent(DeviceID devId,
                        boolean oldValue,
                        boolean newValue)
    {
        this.onOffDevice = true;
        if (oldValue)
            this.oldValue = 1;
        else this.oldValue = 0;
        if (newValue)
            this.newValue = 1;
        else this.newValue = 0;
        this.devId = devId;
    }

    public NetworkEvent(DeviceID devId,
                        short oldValue,
                        short newValue)
    {
        this.onOffDevice = false;
        this.oldValue = oldValue;
        this.newValue = newValue;
    }
}
```

G.2.2 DeviceEventListener.java

```
package homeStd.event;

/**
 * DeviceEventListener.java
 *
 * @author Pierre Moermans
 */

public interface NetworkEventListener
{
    /**
     * This method is called whenever a device value has changed.
     *
     * @param event An event containing the device reference,
     *              the old and the new value.
```

```
     */
    public void valueChanged(NetworkEvent event);
}
```

G.2.3 StandardNet.java

```
package homeStd.service;

/**
 * StandardNet.java
 *
 * @author Pierre Moermans
 */

import homeStd.event.*;

/**
 * This interface defines the uniform access to any devices of any network.
 */

public interface NetAccess
{
    /**
     *  Set the boolean <code>value</code> to a ON-OFF actor.
     *
     * @param netType a string containing the name of a net type.
     * @param description a paramater differentiating the same kind of nets.
     * @param device the number of the device on the net.
     * @param value the <code>boolean</code> value to assign.
     */
    public void setBooleanValue(String netType,
                                String description,
                                int device,
                                boolean value);

    /**
     * Set the short <code>value</code> to a ranged-value actor.
     *
     * @param netType a string containing the name of a net type.
     * @param description a paramater differentiating the same kind of nets.
     * @param device the number of the device on the net.
     * @param value the <code>short</code> value to assign.
     */
    public void setShortValue(String netType,
                              String description,
                              int device,
                              short value);

    /**
     * Get the boolean value from an ON-OFF sensor/actor.
     *
     * @param netType a string containing the name of a net type.
     * @param description a paramater differentiating the same kind of nets.
     * @param device the number of the device on the net.
     * @return the boolean value of the device.
     */
    public boolean getBooleanValue(String netType,
                                   String description,
                                   int device);
```

```
    /**
     * Get the short value from an ranged-value sensor/actor.
     *
     * @param netType a string containing the name of a net type.
     * @param description a paramater differentiating the same kind of nets.
     * @param device the number of the device on the net.
     */
    public short getShortValue(String netType,
                               String descr,
                               int deviceiption);

    /**
     * register a listener.
     *
     * @param listener The listener to register.
     */
    public void addListener(NetworkEventListener listener);

    /**
     * unregister a listener.
     *
     * @param listener The listener to unregister.
     */
    public void removeListener(NetworkEventListener listener);
}
```

Appendix H

Friendly naming bundle

H.1 Interface

```
package friendlyNaming.service;

/**
 * DeviceNaming.java
 *
 * @author Pierre Moermans
 */

import friendlyNaming.exception.*;
import friendlyNaming.event.*;

public interface DeviceNaming
{

    /*
     *
     * information gathering methods
     *
     */

    /**
     * Returns all the sensor names.
     */
    public String[] getSensorNames();

    /**
     * Returns all the actors names.
     */
    public String[] getActorNames();

    /**
     * Return all boolean sensors.
     */
    public String[] getBooleanSensorNames();

    /**
     * Return all ranged-value sensors.
     */
    public String[] getRangedValueSensorNames();
```

```
/**
 * Return all boolean actors.
 */
public String[] getBooleanActorNames();

/**
 * Return all ranged-value actors.
 */
public String[] getRangedValueActorNames();

/**
 * Return information about the device.
 */
public String getDeviceInfo(String device);

/**
 * Return true if the device is an On-Off device.
 */
public boolean isDeviceOnOff(String device);

/**
 * Return true if te device is an value-ranged device.
 */
public boolean isDEviceRangedValue(String device);



/*
 *
 * value related methods.
 *
 */

/**
 * Assign a boolean value to a device.
 */
public void setBooleanValue(String device, boolean value)
    throws NotOnOffDeviceException, NotActuatorException;

/**
 * Assign a short value to a device.
 */
public void setShortValue(String device, short value)
    throws NotRangedValueDeviceException, NotActuatorException;

/**
 * Get the boolean value from a device.
 */
public boolean getBooleanValue(String device)
    throws NotOnOffDeviceException;

/**
 * Get a short value from a device.
 */
public short getShortValue(String device)
    throws NotRangedValueDeviceException;




/*
```

```
     *
     * event listening related methods.
     *
     */

    /**
     * Register a listener.
     */
    public void addListener(DeviceEventListener listener);

    /*
     * Unregister a listener.
     */
    public void removeListener(DeviceEventListener listener);
}
```

H.2 Events

H.2.1 DeviceEvent.java

```
package friendlyNaming.event;

/**
 * DeviceEvent.java
 *
 * @author Pierre Moermans
 */

public class DeviceEvent
{
    private boolean onOffDevice;
    private int oldValue, newValue;
    private String deviceName;

    public void setDeviceName(String deviceName)
    {
        this.deviceName = deviceName;
    }

    public String getDeviceName()
    {
        return deviceName;
    }

    public void setOnOffDevice(boolean onOffDevice)
    {
        this.onOffDevice = onOffDevice;
    }

    public boolean isOnOffDevice()
    {
        return onOffDevice;
    }

    public void setOldValue(int oldValue)
    {
        this.oldValue = oldValue;
    }
```

```
    public int getOldValue()
    {
        return oldValue;
    }

    public void setNewValue(int newValue)
    {
        this.newValue = newValue;
    }

    public int getNewValue()
    {
        return newValue;
    }

    public DeviceEvent(String deviceName,
                       boolean oldValue,
                       boolean newValue)
    {
        this.onOffDevice = true;
        if (oldValue)
            this.oldValue = 1;
        else this.oldValue = 0;
        if (newValue)
            this.newValue = 1;
        else this.newValue = 0;
        this.deviceName = deviceName;
    }

    public DeviceEvent(String deviceName,
                       short oldValue,
                       short newValue)
    {
        this.onOffDevice = false;
        this.oldValue = oldValue;
        this.newValue = newValue;
        this.deviceName = deviceName;
    }
}
```

H.2.2 DeviceEventListener.java

```
package friendlyNaming.event;

/**
 * DeviceEventListener.java
 *
 * @author Pierre Moermans
 */

public interface DeviceEventListener
{
    /**
     * This method is called whenever a device value has changed.
     *
     * @param event An event containing the device number,
     *              the old and the new value.
     */
```

```
    public void valueChanged(DeviceEvent event);
}
```

H.3 Exceptions

H.3.1 DeviceException.java

```
package friendlyNaming.exception;

/**
 * DeviceException.java
 *
 * @author Pierre Moermans
 */

public class DeviceException extends Throwable
{
    private String deviceName;

    public void setDeviceName(String deviceName)
    {
        this.deviceName = deviceName;
    }

    public String getDeviceName()
    {
        return deviceName;
    }

    public DeviceException(String deviceName) {
        setDeviceName(deviceName);
    }
}
```

H.3.2 NotActuatorException.java

```
package friendlyNaming.exception;

/**
 * NotActorException.java
 *
 * @author Pierre Moermans
 */

public class NotActuatorException extends DeviceException
{
    public NotActuatorException(String deviceName) {
        super(deviceName);
    }
}
```

H.3.3 NotOnOffDeviceException.java

```
package friendlyNaming.exception;

/**
 * NotBooleanDeviceException.java
 *
 * @author Pierre Moermans
 */

public class NotOnOffDeviceException
    extends DeviceException
{
    public NotOnOffDeviceException(String deviceName) {
        super(deviceName);
    }
}
```

H.3.4 NotRangedValueDeviceException.java

```
package friendlyNaming.exception;

/**
 * NotRangedValueDeviceException.java
 *
 * @author Pierre Moermans
 */

public class NotRangedValueDeviceException
    extends DeviceException
{
    public NotRangedValueDeviceException(
        String deviceName) {
        super(deviceName);
    }
}
```

List of Figures

List of Tables

Bibliography

[1] http://www.osgi.org. web page. Website of the OSGi organization.

[2] Kirk Chen. *Programming Open Service Gateways with Java Embedded Server Technolgy*. Addison-Wesley, 2001.

[3] Brent A. Miller and Chatschik Bisdikian. *Bluetooth Revealed*. Prentice Hall PTR, 2 edition, 2002.

[4] Sing Li. *Proffessional Jini*. Wrox, 2000.

[5] Anne Thomas. http://wwws.sun.com/software/embeddedserver/whitepapers/whitepaper1.html. Sun Microsystem JES whitepaper, october 1998.

[6] Robert Mines. http://wwws.sun.com/software/embeddedserver/whitepapers/whitepaper2.html. Sun Microsystem JES whitepaper.

[7] Sun MicroSystems. http://wwws.sun.com/software/embeddedserver/whitepapers/dot.com.home.pdf. Sun Microsystem JES whitepaper.

[8] Sun MicroSystems. http://java.sun.com/products/javacomm/. web page, 1998. Sun Microsystem JavaComm web page.

[9] Keane Jarvi. www.rxtx.org. web page, 2002. RXTX home page. Downloads are found in the *download* directory.

[10] Phil Kingery. Digital x-10, which one should i use, part xiii (preamble). http://www.hometoys.com/htinews/feb99/articles/kingery/kingery13.htm.

[11] Computer interface cm11. http://www.marmitek.com/nl_site/pdf/8945.pdf.

[12] *CM11 communication protocol description (not complete)*. http://www.marmitek.com/nl_site/software/PROTOCOL.DOC.

[13] William Grosso. *Java RMI*. O'Reilly, january 2002.

[14] Bill McCarty and Luke Cassady-Dorion. *Java Distributed Objects*. SAMS, 1999.

[15] http://www.linetnetwork.com/. Linet network company home page.

[16] Linet, product description. http://www.linet-network.com/assets/download/gendescr.pdf.

[17] Linet. *Configuration Manual*, 2002. http://www.linet-network.com/assets/download/linconf.pdf.

[18] Linet. *Software specification V6.3*, 2002. http://www.linet-network.com/assets/download/licsw6v3.pdf.

[19] Leon Atkinson. *Core MySQL*. Prentice Hall PTR, 2002.

[20] David Hunter. *Beginning XML*. Wrox, 2000.

[21] Steven Brodhead; Andrei Cioroina; James Hart; Eric Jung Mohammad Akif and Dave Writz. *Java XML Programmer's Reference*. Wrox, july 2001.

[22] Eric van der Vlist. *XML Schema*. OŔeilly, 2002.

[23] http://xml.apache.org/xerces2-j/index.html. Xerces2 for java home page.