FAKULTÄT FÜR !NFORMATIK

Scalimero

TUTORIAL

ausgeführt von

Gabriel A. Grill and Alexander C. Steiner

am:
Institut für rechnergestützte Automation

Betreuer: Ao.Univ.Prof. Dr. Wolfgang Kastner

Wien, August 25, 2010

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Contents

Chapter 1

Introduction

Because of the increasing poularity of home automation, the desire to develop applications for controlling your house became more and more important. Calimero is a collection of Java APIs that together form a foundation for building such applications in EIB/KNX installations. Detailed knowledge of the protocol is not required. On the top of that Calimero only requires J2ME enviorments, which enables use on embedded platforms. It is an open source project and was developed by the Institute of Computer Aided Automation of the Technical University of Vienna.

EIBnet/IP allows you to communicate with components of the widely spreaded EIB/KNX standard by tunnelling over IP Networks. The protocol is published in the KNX Handbook and in the European standard EN 13321-2:2006 (Open Data Communication in Building Automation, Controls and Building Management - Home and Building Electronic Systems - Part 2: KNXnet/IP Communication), available from European standardization organizations. Note: EN 13321-2 is useless without information on the KNX/EIB control network specific data structures (cEMI, DPTs); these should be defined in EN 50090 (Home and Building Electronic Systems).

Since development of Calimero many great projects are using it:

- **KNX@Home**(http://knxathome.fh-deggendorf.de/)
- **BASys**(http://sourceforge.net/projects/basys/)
- **KNXnet/IP Wireshark dissector**(http://knxnetipdissect.sourceforge.net/)
- **CONECT**(http://sourceforge.net/projects/conect/)
- **EIB Home Server**(http://eibcontrol.sourceforge.net/)
- **LEIBnix**(http://leibnix.sourceforge.net/Wikka/HomePage)
- **Sombrero**(http://grill.github.com/sombrero/)
- **Scalimero**(http://grill.github.com/SCalimero/)

Chapter 2

Installation

First, download the calimero package from http://www.auto.tuwien.ac.at/downloads/calimero-all-2.0a4.zip and unpack it to a detination of your choice. It contains several other archives, including 4 Java Archives (.jar), the source code of all of them and documentation of tools and the library. The core library is calimero-2.0a4.jar, this is the file you want to have on the classpath of all your calimero projects. Calimero-gui and calimero-tools are graphical and command line tools to test calimero and KNX, and to gain a better understanding of calimero by looking at their source code. The calimero-rxtx package contains an optional way of serial port access, which is not needed in most circumstances.

The important thing is to put calimero-2.0a4.jar on your classpath. How to add files to your classpath depends on your development environment and should easily be found in the corresponding manual or home page. It is a good idea to unzip the calimero-2.0a3.zip to a place where you can access the documentation comfortably. It is also useful to get familiar with the command line tools and especially their source code, because it covers some important use cases in a concise way.

Chapter 3

DSL

The scalimero DSL is an easy interface for accessing KNX devices and meant to be used in the Scala interpreter. It features convenient classes and a lot of implicit conversions to make the code look good while being able to check type bounds.

3.1 Example

To use the DSL,

```
import tuwien.auto.scalimero.dsl._
```

Then create a network and open it using

```
Network("10.0.0.5") open
```

The IP address specified here is the KNX router used to access the network. Then, we need to create some devices.

```
val lA = Lamp("1/1/0")
val lB = Lamp("1/1/1")
```

Then we can turn the lamps on and off.

```
lA turn on
lA turn off
```

More general:

```
lB send true
lB send false
```

We can also read from the devices.

```
val b = lA.read
```

Note that b is of type `Boolean`, it gets converted from KNX data automatically.

You can also subscribe to events:

```
lA.eventSubscribe(on) {
  println("lA has been turned on")
}
```

If you want to be notified for every write on the device, subscribe a write callback:

```
lA.writeSubscribe{
  newstatus : Boolean =>
  println("lA status: " + newstatus)
}
```

3.2 Network

The `Network` class and companion object in the `tuwien.auto.scalimero.connection` package are the access point to the KNX network. To declare a `Network`, use the companion objects apply method:

```
Network("knxrouter")
```

Replace `"knxrouter"` with a correct KNX router IP address or DNS name. The default medium is twisted pair 1 (9600 bit/s). If you need a different medium for a particular `Network`, use the optional medium parameter:

```
Network("knxrouter", TPSettings.TP0)
```

You can set this globally for all `Network`s by assigning to the companion object's `defaultMedium` field:

```
Network.defaultMedium = TPSettings.TP0
Network("knxrouter")
```

All devices will use the last `Network` created by default. If you want to create the `Network`s seperately from the devices, you can pass a code block to the `Network`s, and every device created within will use the specific `Network`. It gets clearer in an example:

```
val newnet = Network("knxrouter")
val oldnet = Network("knxrouter-old", TPSettings.TP0)

newnet {
  Lamp("1/1/1")
  Lamp("1/1/2")
}
oldnet {
  Lamp("1/2/3")
  Lamp("1/2/5")
}
```

While this was introduced to resemble a markup language, it may be difficult to access the devices from the outside.

If you want to update a particular device (and trigger its callbacks), you can send a read request to the KNX device. It will answer with its current value and the `Network` will relais that update to the device objects:

```
net.readRequest("1/1/1")
```

Note that this call is asynchronous and can (and should) be done from the device object, if available, using its `readRequest` method.

You can also subscribe to events on the KNX network. This is seldom needed, however, as devices do so automatically and messages received from the `Network` lack type information. Still, it can be useful if you need to implement a seperate device handling mechanism:

```
val net = Network("knxrouter")
val act = actor {
  case e : WriteEvent => ...
  case e : ProcessEvent => ...
}
net subscribe act
```

The actor will start to receive `tuwien.auto.scalimero.WriteEvent`s and `tuwien.auto.calimero.process.ProcessEvent`s. `WriteEvent`s occur when a scalimero device writes something to the network while `ProcessEvent`s indicate that something on the network (e.g. a switch) sent an update. In most cases, however, it is easier to subscribe to devices themselves.

3.3 Devices

Devices are abstractions for the physical devices connected to the KXN bus. In most cases, the preconfigured devices in `tuwien.auto.scalimero.device.preconf` should suffice (for information on how to create your own devices and advanced funcionality, see later chapters). As stated above, devices automatically subscribe to the last created `Network`, so it is a bad idea to create a device without creating a `Network` first. To create a device, use one of the constructor objects in `preconf`:

```
1 val lamp = Lamp("1/1/1")
2 val switch = Switch("1/1/2")
3 val dimmer = Dimmer("1/1/3")
4 val rollerBlind = RollerBlind("1/1/4")
5 val temperature = Temperature("1/1/5")
```

`Lamp` and `Switch` use boolean values or the more descriptive `on` and `off` aliases, while `Dimmer` and `RollerBlind` use percent values (= `Int`) from 0 to 100 and `Temperature` uses float values (unit depends on your thermostat).

To set values, use the send method, or `Lamp`'s and `Switch`'s more natural `turn` method.

```
1 lamp turn on
2 switch turn off
3 dimmer send 100
4 rollerBlind send 0
5 temperature send 19
```

To read values, use the read method. It always is of the appropriate type.

You can also subscribe callbacks. There are two types of callbacks, event callbacks and write callbacks. Event callbacks register to a specific type of event and execute whenever that event occurs while write callbacks get called every time the state of a device is updated.

```
1 lamp.eventSubscribe(on) {
2   println("lA has been turned on")
3 }
4
5 dimmer writeSubscribe {
6   newVal : Int =>
7   println("new dimmer value:" newVal)
8 }
```

The subscription methods return `WriteCallback` or `EventCallback` objects, which can be used to change the event code or detach(unsubscribe) the event.

```
val wc = dimmer writeSubscribe {
  newVal : Int =>
  println("new dimmer value:" newVal)
}

def loggit(l : Logger) {
  wc update {
    newVal : Int =>
    l.info("new dimmer value:" newVal)
  }
}

def shutdown {
  wc.detach
  //which is the same as
  //dimmer writeUnsubscribe wc
}
```

Chapter 4

Devices

4.1 Overview

In Calimero KNX devices were represented through the `Datapoint` class. Instances of this class could be used to send and receive messages by passing it as a parameter to a `ProcessCommunicator`.

In Scalimero a more decentralized approach was chosen. Devices were introduced.

Devices have the following members(constructor parameter):

- **A KNX address** - Name: destAddr

  Can be passed as a String or GroupAddress object to the constructor of Device.

- **DPType** - Name: dpt

  `DPType` is a wrapper class for `DPT`, which means a instance of DPType or DPT can be passed as constructor parameter.

- **Name**[optional] - Name: name

  The name is a String an will be given to the DataPoint instance encapsulated in the Device class.

- **Network**[optional] - Name: net

  If this parameter is not given, the value of Network.default, which contains a reference to last created Network instance, will be taken.

Device traits:

- **TDevice** - Every device inherits this trait.
- **TCommandDevice** - Every device with the send method inherits this trait.
- **TStateDevice** - Every device with the read method inherits this trait.

The `Device` base class is an Actor, which means it can send and receive messages. Because of this ability it is possible to send to the device actor `ProcessEvent`s or `WriteEvent`s to trigger subscribed events.

Subclasses of Device:

- **CommandDevice**

  This class hasn't got any state, which means it can only be used for sending messages.

  send(d: DataPointValue)

  `DataPointValue` is a wrapper class for the primitive types specified through the `DPType`. It's possible to pass a instance of `DataPointValue` or primitive type value.

- **StateDevice**

  This class has the ability to send and read messages. Even though the name of the class is `StateDevice`, no state is stored in this class. It has got two methods to read values from the KNX Network:

  - **read**: PrimitveType

    Returns a value with the primitive type chosen through the `DPType`. If it's not possible to get a message, an Exception is thrown.

  - **readOption**: Option[PrimitiveType]

    Returns a value with the primitive type chosen through the `DPType` encapsulated in an Option.

4.2 Device in the REPL

It is advised to use the base Device classes only for the purpose of testing or quick scripting. In most cases it is more rewarding to create a specialzied Device class and

let it inherit from `StateDevice` or `CommandDevice`. How this can be done exactly will be explained in the next chapter.

To use it,

```
import tuwien.auto.scalimero
import device._
import device.dtype._
import connection._
```

Then create a network and open it. Now the creation of the devices can begin.

```
val d1 = Device("1/1/1", Boolean.SWITCH)
```

To shorten the qualification of the `DPType` just

```
import device.dtype.Boolean._
```

Now all Boolean `DPType`s are in scope an it is possible to write

```
val d2 = Device("1/1/0", SWITCH)
```

By writing the above code the `apply` method of the companion object Device is called and instantiates a `StateDevice` with the given parameters.

To send messages

```
d1 send true
```

or

```
d2 send on
```

To read messages

```
try{
    if(d1 read)
        println("Lamp was turned on")
    else
        println("Lamp was turned off")
}catch {
    case e => println("No value was received, because of an Exception")
```

or

```
d1 readOption match {
    case Some(value) =>
        if(value)
            println("Lamp was turned on")
        else
            println("Lamp was turned off")
    case _ => println("No value was received, because of an Exception")
}
```

The Device base classes can only subscribe `WriteCallback`s.

4.3 SimpleDevice

The class `SimpleDevice` was created because of the need for an quick to write abstract with less overhead. This is accomplished by removing the Actor functionality. This class has the same read and send methods like `StateDevice` and some additional low level sending methods:

- **readRequest** - sends a read request
- **write**(value: PrimitiveType) - sends a value of primitve type
- **write**(value: Array[Byte]) - sends a value of type byte array
- **write**(value: String) - sends a value of type String(the format depends on `DPT`)

4.4 Grouping of Devices

Grouping of Devices can be done on different levels:

- **low level** - through `WriteCallback`s
- **mid level** - through events
- **high level** - through:
  - **GroupDevice**

    `GroupDevice` inherits from `HashSet` and because of that it can be filled with Devices. With the `send` method a message can be sent to all Devices in the

GroupDevice. The `setMaster` method enables you to set a master device. When the state of master device changes a message, containing the new state, is sent to all devices in the GroupDevice. Through the `addProxyFunction` method it is possible set a function, which will be evaluated, before the new received state will be forwarded to the other devices. The Result of this function will then be sent instead of original received value.

```
1 val gd = GroupDevice(d1, d2)
2
3 //all lights on
4 gd send on
5 //or
6 gd map {_ send on}
7
8 gd setMaster d1
9
10 //d2 will also be turned off automatically
11 d1 send off
```

– **MultipleAddressDevice**

This class functions as an abstraction for a devices with a separate reading and writing address.

```
1 val mad1 = new MultipleAddressDevice(d1, d2)
2 //or
3 val mad2 = new MultipleAddressDevice("1/1/1", "1/1/0", Switch)
```

Chapter 5

How to develop Devices

5.1 Getting started

5.2 Events

Chapter 6

Conclusion and Outlook

Bla Bla