



NeSSI²

A graphic element consisting of a green and white globe with a red square on its surface, positioned to the right of the 'NeSSI²' text.

Tutorial

Version 2.0.0-beta-2



CC ACT
Agent Core Technologies



CC COG
Cognitive Architectures



CC IRML
Information Retrieval
& Machine Learning



CC NEMO
Network & Mobility



CC NGS
Next Generation Services



CC SEC
Security

Editors:

Karsten Bsufka and Rainer Bye

Contributors:

Arik Messerman

Katja Luther

Stephan Schmidt

Joël Chinnow

Dennis Grunewald

Thomas Hauschild

Sebastian Linkiewicz

Thorsten Rimkus

Jakob Strafer

DAI-Labor der Technischen Universität Berlin

Prof. Dr.-Ing. habil. Sahin Albayrak

Technische Universität Berlin / DAI-Labor

Institut für Wirtschaftsinformatik und Quantitative Methoden

Fachgebiet Agententechnologien in betrieblichen Anwendungen und der Telekommunikation

Sekretariat TEL 14

Ernst-Reuter-Platz 7

10587 Berlin

Telefon: (030) 314 74000

Telefax: (030) 314 74003

E-mail: Sekretariat@dai-labor.de

WWW: <http://www.dai-labor.de>

Contents

List of Figures	v
1 Introduction to the tutorial example	1
2 Setting up your development environment	3
2.1 Setting up Maven 2	3
2.2 Creating a Maven 2 project for a <i>NeSSi²</i> plug-in	4
3 Implementing a <i>NeSSi²</i> Device	5
4 Creating a <i>NeSSi²</i> application	7
4.1 A very simple application	7
4.2 Defining an echo client application	7
4.3 An echo client application	8
4.4 An echo server application	10
4.5 Event recording	13

List of Figures

2.1	settings.xml for Maven 2	3
2.2	Example pom.xml for a <i>NeSSi²</i> plug-in project	4
3.1	A simple <i>NeSSi²</i> device implementation	5
4.1	Skeleton for a <i>NeSSi²</i> application	7
4.2	Application Definition , part 1	8
4.3	Application Definition , part 2	9
4.4	start method of an application	10
4.5	handleEvent method of an application	11
4.6	EchoServer application class	11
4.7	start method for echo server application	12
4.8	receive method implementation	12
4.9	doEcho method implementation	13
4.10	Echo server application logic	13
4.11	Echo sent event	14
4.12	receive method implementation with recording functionality	15
4.13	Recorder configuration for recorder defined receive method	16

1 Introduction to the tutorial example

In this *NeSSi*² tutorial we develop a server that answer echo request from various client application distributed to various sub-networks within in a larger network.

The goal of this tutorial is to introduce the basic steps required to extend *NeSSi*² with additional applications, protocols and devices, also we will present how to record simulation events to a database.

This tutorial is structured as follows. Section one describes setting up a development environment using Maven 2¹ and Eclipse². In the second section we will develop a device, that letter will host the echo client and server applications. The third and forth section describe the actual development of an echo client and server. Finally the fifth chapter describes how to create and run a simulation with the echo clients and servers.

This tutorial assumes a basic knowledge of Java, Maven 2 and Eclipse, please refer to the documentation of those products for further information.

¹<http://maven.apache.org>

²<http://www.eclipse.org>

2 Setting up your development environment

To develop a *NeSSi²* application you need at least Java 6. Furthermore, for this tutorial we will use Maven 2.0.9 and Eclipse 3.4. If you do not have those versions yet, download¹ and install them.

2.1 Setting up Maven 2

To use Maven 2 for developing a *NeSSi²* extension you must make sure you have access to the *NeSSi²* Maven repository. You can either add this repository to your pom.xml (see below) or to your Maven 2 settings.xml.

```
1 <settings>
2   <profiles>
3     <profile>
4       <id>nessi2</id>
5       <repositories>
6         <repository>
7           <id>nessi2</id>
8           <url>
9             http://lehre.dai-labor.de:8080/archiva/repository/nessi2
10            </url>
11          <snapshots>
12            <enabled>>false</enabled>
13          </snapshots>
14          <releases>
15            <enabled>>true</enabled>
16          </releases>
17        </repository>
18      </repositories>
19    </profile>
20  </profiles>
21  <activeProfiles>
22    <activeProfile>nessi2</activeProfile>
23  </activeProfiles>
24 </settings>
```

Figure 2.1: settings.xml for Maven 2 with *NeSSi²* repository entry

¹<http://java.sun.com>
<http://maven.apache.org>
<http://www.eclipse.org>

2.2 Creating a Maven 2 project for a *NeSSi²* plug-in

The project description for *NeSSi²* plug-in project is relativ simple, Figure 2.2 shows a minimal pom.xml file for a *NeSSi²* plug-in.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project
3   xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6   http://maven.apache.org/maven-v4_0_0.xsd"
7 >
8   <modelVersion>4.0.0</modelVersion>
9
10  <groupId>foo.bar</groupId>
11  <artifactId>myPlugin</artifactId>
12  <version>1.0.0</version>
13
14  <dependencies>
15    <dependency>
16      <groupId>de.dailab.nessi2.protocols</groupId>
17      <artifactId>nessi2-protocols-core</artifactId>
18      <version>nov08</version>
19    </dependency>
20    <dependency>
21      <groupId>de.dailab</groupId>
22      <artifactId>sad</artifactId>
23      <version>1.0.0</version>
24    </dependency>
25  </dependencies>
26
27  <build>
28    <plugins>
29      <plugin>
30        <artifactId>maven-compiler-plugin</artifactId>
31        <configuration>
32          <source>1.6</source>
33          <target>1.6</target>
34        </configuration>
35      </plugin>
36    </plugins>
37  </build>
38 </project>
```

Figure 2.2: Example pom.xml for a *NeSSi²* plug-in project

With this basic POM file we can start developing our plugin. So start Eclipse and import the POM as a Maven project² and your are ready to create a device.

²To do this you need a Maven 2 Eclipse plugin, which is available here: <http://m2eclipse.codehaus.org>

3 Implementing a *NeSSi²* Device

Figure 3.1 presents the simplest possible implementation of a *NeSSi²* device. In *NeSSi²* devices serve two purposes:

1. To model the attributes and behavior of networked devices.
2. To represent devices that always execute some specific applications, e.g. a server computer with a web server application running on it.

```
1 package de.dailab.nessi.tutorial;
2
3 import de.dailab.insin.netshield.api.annotations.*;
4 import de.dailab.insin.netshield.api.applications.*;
5
6 @Device(caption = "TutorialDevice",
7         description = "Example Device for NeSSi2 tutorial",
8         largeImage = "icons/Computer.png",
9         smallImage = "icons/Compuer16x16.png")
10 public class TutorialDevice extends AbstractDevice {
11
12     private static final long serialVersionUID = 8372324679984736143L;
13
14     public TutorialDevice() {
15         // Nothing to do here
16     }
17
18     @Override
19     protected void initialize() {
20         // Again nothing to do
21     }
22 }
```

Figure 3.1: A simple *NeSSi²* device implementation

A device must have a default constructor, otherwise the *NeSSi²* API cannot make instances of the device.

The annotation `@Device` is required for every device. The information contained in this annotation are used by the *NeSSi²* user interface to visualize a device and provide hints about it. The four elements of this annotation are:

1. `caption` Caption for the device when displayed in user interface.
2. `description` Short description, which will be used by the user interface for tool tips.

3. `smallImage` A small icon for device. This icon will be displayed in a network editor palette and should have a size of 16 by 16 pixels.
4. `largeImage` A larger version of a device icon. This icon will be displayed within the network editor and should have a size of 64 by 64 pixels.

*NeSSI*² devices can also have device specific attributes, which can be fixed and only will displayed in a properties or can be changeable by applications attached to the device. In both cases initial values can be set during the initialization of the device, this will be done within in the `initialize` method. To enable the user interface to access these attributes, they have to be registered by annotations. The details for this registration are described in the application development section. Applications and devices use the same mechanism for registering attributes.

4 Creating a *NeSSi²* application

A *NeSSi²* consist of two part an application definition and the actual application implementation.

4.1 A very simple application

Figure 4.1 shows the simplest possible *NeSSi²* application. Unfortunately, this application also does nothing useful.

```
1 package de.dailab.nessi.tutorial;
2
3 import de.dailab.insin.netshield.api.applications.
4 AbstractApplication;
5
6 public class EchoClient extends AbstractApplication {
7
8     public EchoClient() {
9
10    }
11 }
```

Figure 4.1: Skeleton for a *NeSSi²* application

But before we can add some functionality to this application. We need an application definition.

4.2 Defining an echo client application

Application definitions are used to globally register static information about an application. This includes:

- A name for the application.
- A short description for an application.
- Attributes and attribute types for an application.

See Figures 4.2 and 4.3 for the echo client application definition. This application definition allows to specify to which echo server, echo clients application should establish a connection. **Note:** The default value `127.0.0.1` will not be resolved by

```
1 package de.dailab.nessi.tutorial;
2
3 import de.dailab.insin.netshield.api.annotations.*
4 import de.dailab.insin.netshield.api.applications.AppDef;
5
6 @ApplicationDefinition(caption = "Echo client",
7     description = "A client for an echo service")
8 public class EchoClientDef extends AppDef {
9
10     private static final String APP_NAME = "Echo client";
11
12     private static final long serialVersionUID =
13         -5343696563064518406L;
14
```

Figure 4.2: Skeleton for a *NeSSi²* application definition, part one

NeSSi² to the localhost or its IP address. It will cause an error, stating that the IP address is unknown.

`@ApplicationDefinition` in Figure 4.2 is used for global registration of a name and a description for an application.

If an application has attributes, it needs to register them with the `@Attribute` annotation, see Figure 4.3. The `@Attribute` annotation has two parameters: `caption` that will be shown at the graphical user interface and `fieldType` that will be used in the GUI to show and edit the attribute. By default `fieldType` is set to `FieldType.String`. Currently `FieldType.String`, `FieldType.Integer` and `FieldType.Boolean` are available.

Caution: Attribute values are not set for individual applications, but for all applications that have the same application definition.

In Figure 4.1 there is no reference to an application definition. In the next section we will have a look on how an application can access the values defined in an application definition.

4.3 An echo client application

Now that we have the echo client application definition, we can start to implement an echo client, that uses the echo service.

An application steps through several phases in its lifetime:

1. Construction
2. Initialization: If your requires special initialization, it must override the method `init`. **Attention:** You MUST call the super `init` method within your `init` implementation.
3. Start: If you requires a customized start code, it must override the method `start`, see Figure 4.4. The `start` method also allows you access to the application defi-

```

1  @Attribute(caption = "IP address of the echo server")
2  private String echoServerIp="127.0.0.1";
3
4  @Attribute(caption = "Port if the echo server",
5            fieldType = FieldType.Integer)
6  private int echoServerPort=7;
7
8  public EchoClientDef() {
9      this.setApplication(EchoClient.class.getCanonicalName());
10     this.setName(APP_NAME);
11 }
12
13 public final String getEchoServerIp() {
14     return this.echoServerIp;
15 }
16
17 public final void setEchoServerIp(String ipAddr) {
18     this.echoServerIp = ipAddr;
19 }
20
21 public final int getEchoServerPort() {
22     return this.echoServerPort;
23 }
24
25 public final void setEchoServerPort(int port) {
26     this.echoServerPort = port;
27 }
28 }

```

Figure 4.3: Skeleton for a *NeSSi²* application definition, part two

inition for this application. **Attention:** You MUST call the super `start` method within your `start` implementation.

4. Shutdown: If you want to clean up your application during the application shutdown phase, you override the method `shutdown`. As a convenience, *NeSSi²* offers a method restarting applications. **Attention:** You MUST call the super `shutdown/restart` method within your `shutdown/restart` implementation.

The `start` method is just responsible for the starting of the application, the runtime behaviour of the application is determined by the events the application receives and how it reacts to them. An important role the `start` method plays in this context is that it tells *NeSSi²* when to deliver the first tick event to the application (see line 20 – 21).

The first event type our application receives are time (tick¹) events, sent by *NeSSi²* to an application. To handle those events an application needs to override the method `handleEvent`, see Figure 4.5.

¹A tick is the smallest possible time interval in *NeSSi²*. The actual time length depends on the simulation and simulation mode.

```
1 @Override
2 public boolean start(AppDef ad) {
3
4     EchoClientDef echoClientAppDef;
5     if(ad instanceof EchoClientDef) {
6         echoClientAppDef = (EchoClientDef) ad;
7     } else {
8         return false;
9     }
10
11     this.echoServerIp = echoClientAppDef.getEchoServerIp();
12     this.echoServerPort = echoClientAppDef.getEchoServerPort();
13
14     TickedEvent e = new TickedEvent(this, 10);
15     getLayerHandler().addEvent(e);
16
17     return true;
18 }
```

Figure 4.4: start method of an application

This implementation sends an UDP packet with an echo request to an echo server (see line 5 – 13). After that it tells *NeSSI*² to inform it, when ten ticks have past in the simulation. After that a new echo request will be created.

This concludes the description of a simple echo application. Next we look at the echo server implementation and how packets are received and events are logged to the database.

4.4 An echo server application

In this section we skip a detailed look at the echo server application class and just highlight the differences to the echo client application. For the complete source code we refer you to the tutorial example source code. The main difference is, that our echo server application extends another class, see Figure 4.6.

The method `handleDetectionResults` (see line 16 – 18) will not be needed for this example. The intended purpose of the class `AbstractPromiscuousModeApplication` was to realize applications for detecting malware, attacks and intrusions. The actual detection mechanisms could report its result with an event which would be handled by this method.

The use of `AbstractPromiscuousModeApplication` also requires a slightly different implementation of the `start`, see 4.7, here it is important to call the `super.start` and check the result, before your own implementation.

What we will need is the method `receive`. This method ensures that your application receives every (!) packet that is received by one of the network interfaces attached to the device your application is running on. Since the implementation is a bit more complex we omitted the code from Figure 4.6. The full implementation for this method can be seen in Figure 4.8.

```

1 @Override
2 public void handleEvent(TickedEvent e) {
3     super.handleEvent(e);
4
5     IPAddress addr =
6         IpFactory.eINSTANCE.createIPv4Address(echoServerIp);
7
8     deviceLayerHandler.getTransportLayer().createUDPPacket(
9         1234,
10        echoServerPort,
11        addr,
12        "Hello World".getBytes()
13    );
14
15    e.setTick(e.getTick()+10);
16    this.deviceLayerHandler.addEvent(e);
17 }

```

Figure 4.5: `handleEvent` method of an application

```

1 package de.dailab.nessi.tutorial;
2
3 import de.dailab.insin.netshield.api.applications.
4 AbstractPromiscuousModeApplication;
5 import de.dailab.insin.netshield.ip.IPacket;
6 import de.dailab.insin.netshield.ip.NetworkInterface;
7 import de.dailab.sad.detection.unit.DetectionResultEvent;
8
9 public class EchoServer
10     extends AbstractPromiscuousModeApplication {
11
12     @Override
13     public void receive(IPacket p, NetworkInterface ni) {
14     }
15
16     @Override
17     public void handleDetectionResults(DetectionResultEvent arg0) {
18     }
19 }

```

Figure 4.6: `EchoServer` application class

The `receive` method retrieves IP and UDP headers from the received packet. Only if both are present it checks, if the received UDP packet was actually an echo request.

If an UDP packet is an echo request is determined by comparing the destination port with the defined echo server port², see line 9 – 17.

If an echo request has been found it will be processed by the `doEcho` method, see

²This port is again defined in an echo server application definition and retrieved within `start` method of the echo server application.

```
1 @Override
2 public boolean start(AppDef ad) {
3     if (!super.start(ad)) {
4         return false;
5     }
6
7     if(ad instanceof EchoServerDef) {
8         this).echoServerPort =
9             ((EchoServerDef) ad).getEchoServerPort();
10    } else {
11        return false;
12    }
13    return true;
14 }
```

Figure 4.7: start method for echo server application

```
1 @Override
2 public void receive(IPacket p, NetworkInterface ni) {
3
4     if(!(p instanceof Packet)) {
5         return;
6     }
7     Packet packet = (Packet) p;
8
9     IPv4Header ipHeader =
10        (IPv4Header) packet.findHeader(IPv4Header.class);
11    UDPHeader udpHeader =
12        (UDPHeader) packet.findHeader(UDPHeader.class);
13
14    if((ipHeader==null) || (udpHeader==null)) {
15        return;
16    }
17    if(echoServerPort == udpHeader.getDestinationPort()) {
18        doEcho(new String(packet.getPayload()),
19            ipHeader.getSourceAddress(), udpHeader.getSourcePort());
20    }
21 }
```

Figure 4.8: receive method implementation

line 18 – 19 and Figure reffig:doEcho. The doEcho method receives the payload, the source IP address and the source port of the the received echo request packet

Note: It is considered good practice to separate the application logic and and application integration in two separate classes. The application integration call handles all communication/interaction with *NeSSi²* and application logic class, should not have any *NeSSi²* knowledge and only implement the logic of an application. See Figure 4.10 for echo server application logic.

Finally we will take a look on how the echo client and server can log events to the database, so that activities can be visualized and analyzed with *NeSSi²*.

```

1 private void doEcho(String msg, IPAddress addr, int port) {
2     String echoMsg = this.echo.echo(msg);
3     deviceLayerHandler.getTransportLayer().
4         createUDPPacket(echoServerPort, port, addr, echoMsg.getBytes());
5 }

```

Figure 4.9: doEcho method implementation

```

1 package de.dailab.nessi.tutorial;
2
3 public class Echo {
4
5     public String echo(String msg) {
6         if(msg == null) {
7             return "";
8         } else {
9             return new StringBuilder(msg).reverse().toString();
10        }
11    }
12 }

```

Figure 4.10: Echo server application logic

4.5 Event recording

*NeSSi*² offers a flexible way of configuring events for recording to database. The first step is to decide what we want to record to the database. For this example we want to record the following:

1. Packets sent by a device.
2. Number of Echo requests received by an echo server.
3. Number of Echo replies sent by an echo server.

The first event is easy to record, because *NeSSi*² offers out of the box support for recording this type of event. For the other two we need to write an event class. In Figure 4.11 we did this for sent echo replies³. Note that we specified to record these events with component-dependent integer values (line 30 – 33).

In the second step we need to identify for the later two of those three event types the locations in the code where we want to record the appropriate events (Remember: The first event type is supported out of the box).

The EchoSentEvent would be best recorded after line 19 within the code snippet from Figure 4.8. For the actual recording we need a recorder which can be accessed via the layer handler which we saved within the start method in the private field `deviceLayerHandler`. The recorder provides the method `record` which is quite self-explanatory.

³The example for received echo requests is omitted, because it is almost the same, except for the class name and the label text.

```
1 package de.dailab.insin.netshield.sample.echo;
2
3 import de.dailab.insin.netshield.logging.types.LoggingTypeInterface;
4
5 public class EchoSentEvent
6     implements LoggingTypeInterface {
7
8     private static final String label = "Echos Sent";
9     private static final int id =
10         EchoSentEvent.class.getCanonicalName().hashCode();
11     private static EchoSentEvent instance = null;
12
13     public static final EchoSentEvent getInstance() {
14         if(EchoSentEvent.instance == null) {
15             EchoSentEvent.instance = new EchoSentEvent();
16         }
17         return EchoSentEvent.instance;
18     }
19
20     @Override
21     public int getID() {
22         return id;
23     }
24
25     @Override
26     public String getLabel() {
27         return label;
28     }
29
30     @Override
31     public RecordType getRecordType() {
32         return RecordType.INTEGER;
33     }
34 }
```

Figure 4.11: Echo sent event

Figure 4.12 shows the code from Figure 4.8 with added recording functionality.

We want to record the point in time (i.e. the tick) the event occurred. We get this from the layer handler (line 21 – 22). Since we chose the component-dependent type of integer logs, we also need to pass information about the affected component. The easiest way to do so is to get the device from the layer handler and pass it to the constructor of the `NetworkComponent` class, just as shown in line 23 – 24). To bundle up the information, we simply create an `IntegerLogEntry` with the tick, the component, the type descriptor that is instantiated by its `getInstance` method, and finally the value that should be recorded for this event (which is constantly 1 in this case), as in line 28 – 29. This `IntegerLogEntry` is then passed to the `record` method of the appropriate recorder that we get via the layer handler's `getRecorder` method. **Note** that the class name of the *written event* is passed and **NOT** the name of the application class!

The decision to record this not just as plain device dependent events but as integer

```

1 @Override
2 public void receive(IPacket p, NetworkInterface ni) {
3
4     if(!(p instanceof Packet)) {
5         return;
6     }
7     Packet packet = (Packet) p;
8
9     IPv4Header ipHeader =
10         (IPv4Header) packet.findHeader(IPv4Header.class);
11     UDPHeader udpHeader =
12         (UDPHeader) packet.findHeader(UDPHeader.class);
13
14     if((ipHeader==null) || (udpHeader==null)) {
15         return;
16     }
17     if(echoServerPort == udpHeader.getDestinationPort()) {
18         doEcho(new String(packet.getPayload()),
19             ipHeader.getSourceAddress(), udpHeader.getSourcePort());
20
21         final int tick =
22             (int) deviceLayerHandler.getCurrentTick();
23         final NetworkComponent device =
24             new NetworkComponent(deviceLayerHandler.getMyDevice());
25         final LoggingTypeInterface echoSentEvent =
26             EchoSentEvent.getInstance();
27         final IntegerLogEntry valLogEntry =
28             new IntegerLogEntry(tick, device, echoSentEvent, 1);
29
30         final IRecorder recorder = this.deviceLayerHandler
31             .getRecorder(echoSentEvent.getCanonicalName());
32         recorder.record(valLogEntry);
33     }
34 }

```

Figure 4.12: receive method implementation with recording functionality (line 21 – 32)

events (with the constant value 1) pays off when we have a look at the *NeSSI*² GUI (graphical user interface). The Statistics view of the Network Simulation perspective is able to plot the activities that were recorded in this way. We also may choose to set the charts cumulative to see the total number of echos sent until a certain point in time.

Before we are able to start a simulation that records the identified events, we have to check the configuration of the log4j framework. When we create a new session for a network, we have to choose one of the default configurations which is then copied to the created session. We need to edit the configuration and check, that either the DBAppender class or the DBIntegerLogger is active for the EchoSentEvent class. Since log4j logger configuration is hierarchical, the easiest way to make sure that EchoSentEvent is configured properly is to add the following section to the configuration file:

Make sure the DBAppender is configured within the file and refers to the class *de.dailab.insin.netshield.logging.types.DBAppender*.

```
1 <logger name="de.dailab.insin.netshield.sample.echo.EchoSentEvent "  
2   additivity="false">  
3   <appender-ref ref="DBAppender"/>  
4 </logger>
```

Figure 4.13: Recorder configuration for recorder defined `receive` method