1 Concepts 3
  1.1 Basic Concept ............................................ 3
  1.2 Components .............................................. 4
    1.2.1 Sources .................................................. 5
    1.2.2 Drains .................................................... 6
    1.2.3 Nodes ..................................................... 7
  1.3 Logging ...................................................... 7
  1.4 Plug-in ...................................................... 9

2 Installation 11
  2.1 Howto Install Java ......................................... 11
    2.1.1 Linux ..................................................... 11
    2.1.2 Windows (2000/XP) ...................................... 14
  2.2 L-DSMS ..................................................... 18

3 Examples 19
  3.1 Preliminaries ............................................... 19
    3.1.1 Path Information ........................................ 19
    3.1.2 Install Folder .......................................... 19
    3.1.3 Open a Console ......................................... 20
  3.2 Hello World ............................................... 21
  3.3 Filtering Strings ........................................... 22
  3.4 Filtering XML stream ....................................... 24

4 Managing L-DSMS with VISU-L-DSMS 29

5 Components 37
  5.1 Overview by packagename ................................. 37
    5.1.1 <ldsms core package>.generics ..................... 37
    5.1.2 <ldsms core package>.string ....................... 38
    5.1.3 <ldsms core package>.xml ........................... 38
  5.2 Core Components .......................................... 39
    5.2.1 Buffer ................................................. 39
    5.2.2 ByteArray2String ...................................... 40
    5.2.3 ByteArrayFileDrain .................................... 41
    5.2.4 ByteArrayFileSource .................................. 43
    5.2.5 ByteArraySocketDrain .................................. 44
    5.2.6 ByteArraySocketSource ................................ 45
5.2.7 Cast ................................................................. 47
5.2.8 ConsoleDrain .................................................... 48
5.2.9 Filter ............................................................ 49
5.2.10 ObjectSocketDrain ............................................. 50
5.2.11 ObjectSocketSource ........................................... 52
5.2.12 SpexNode ........................................................ 53
5.2.13 String2ByteArray .............................................. 54
5.2.14 StringFileDrain ............................................... 56
5.2.15 StringFileSource .............................................. 57
5.2.16 StringReplaceNode ............................................ 58
5.2.17 StringSocketDrain ............................................. 60
5.2.18 StringSocketSource ........................................... 61
5.2.19 StringTokenizerNode ......................................... 62
5.3 FilterConditions .................................................. 64
  5.3.1 AndCondition .................................................. 65
  5.3.2 FalseCondition ............................................... 65
  5.3.3 NotCondition ................................................ 65
  5.3.4 OrCondition .................................................. 65
  5.3.5 RDSGroupFilterCondition ................................... 66
  5.3.6 RegexCondition .............................................. 66
  5.3.7 RepeatedStringCondition ................................... 67
  5.3.8 TrueCondition .............................................. 67
5.4 Plugins ............................................................ 67

A Dependencies ....................................................... 69
B Configuration file .................................................. 71
C HowTo: Implement A Component .................................. 73
  C.1 Drains .......................................................... 73
  C.2 Sources ........................................................ 75
  C.3 Nodes .......................................................... 79
D Example Application ................................................. 81
  D.1 Overview by packagename ...................................... 81
    D.1.1 <ldms core package>.rds ................................ 81
    D.1.2 <ldms core package>.rds.easyway ....................... 81
    D.1.3 <ldms core package>.tmc ................................ 82
    D.1.4 <ldms core package>.xml ................................ 82
  D.2 Components .................................................. 83
    D.2.1 ByteArray2RDSBlock ..................................... 83
    D.2.2 ByteArray2RDSGroup ..................................... 84
    D.2.3 EasyWayFileSource ....................................... 86
    D.2.4 EasyWayRDSChunk2RDSBlock ............................... 87
    D.2.5 EasyWaySource ........................................... 89
    D.2.6 OTNDrain .................................................. 91
    D.2.7 RDSBlock2ByteArray ..................................... 93
    D.2.8 RDSBlock2RDSGroup ..................................... 95
    D.2.9 RDSGroup2ByteArray ..................................... 96
    D.2.10 RDSGroup2RDSGroup ..................................... 98
Introduction

The evolution in Information Technologies has led to more and more tools for data collection (e.g. electronic devices) and to decreasing complexity and costs of these tools. As a result of this evolution, more and more raw data is collected, that needs to be processed to result into valuable information. It may be filtered, transformed, cloned, accumulated or manipulated in another way and decisions have to be made, based on this data and the resulting information. The processing can’t always be done after the data has been fully collected, because in many cases the data is generated as an endless stream. This plenty of data can’t be processed by hand and software systems are needed to do this job automated. If the same scenario don’t need to be repeated over and over again, it won’t be efficient to create new software in a cost intensive and time lasting process to fit the needs. Not only because the main target is to process the data and not to implement software but because it won’t be used very often, maybe only once. Instead of creating new software for every scenario, a tool is needed, that is flexible enough to fulfill the changing requirements without taking much time to be readjusted for new scenarios.

All this can be done easily with L-DSMS. L-DSMS stands for ‘Local Data Stream Management System’ and is a system for managing and processing data streams. The basic concept is to break down the whole process of processing a data stream into smaller subprocesses, that can be managed by existing software components (processing nodes) and to combine these components to a chain of processing nodes. Some of them produce data (they are called Sources), other recieve data (they are called Drains) and some recieve data and produce new one, based on the input (they are called Nodes). L-DSMS connects these components with each other to a network of subprocesses. A Source can deliver data to one or several Drains and a Drain can recieve data from one or several Sources and a Node is a Drain (for a Node or Source) and a Source (for a Node or Drain) at the same time. The functionality of the system is the direct result of the Sources and Drains that are used and the way they are connected to each other. But instead of editing source code, all this configuration is made within a simple XML document, so that no programming skills are needed to change the configuration and therefore the behaviour of the system. This makes L-DSMS easy to use, very flexible, saves hardware resources and therefore minimizes the hardware requirements, because the system can always be configured in a way, that the requirements are fulfilled without any unnecessary functions.

A second application of L-DSMS may be to simulate data streams and therefore create a test environment for software that uses sensor data. There are several components, that can be used to simulate data producing sensors. So software testing is possible, even if the sensors aren’t available or special sce-
narios need to be tested, that can’t be realized with real sensors.

It is also possible to use L-DSMS as a library for your own Java applications, it was developed to be used as a standalone application out of the box. It can be executed on every platform that supports Java 5.0 or newer, which should include nearly all of the common computer systems.

This document was designed to give a step by step introduction to L-DSMS as well as to be a reference book for any questions regarding the use of L-DSMS. The first chapter explains the concepts behind L-DSMS and how it works. In the second chapter it is explained, how to install L-DSMS on a Linux or Windows system and in the third chapter there are several examples to show how to use L-DSMS. Chapter four shows how to manage L-DSMS using the graphic managing tool VISU-L-DSMS. Chapter five gives an overview for each component available in the L-DSMS core package and a detailed description for each of them. This chapter can be used as a reference during planning and managing new L-DSMS projects. Frequently asked questions (FAQ) are answered in the appendix.

Up-to-date informations, software downloads and further documentation for L-DSMS is located at the project homepage http://www.pms.ifi.lmu.de/reverse-wga1/ldsms/.
This chapter describes the concepts behind L-DSMS and explains how the system works. To understand, what kind of scenarios L-DSMS can be used for and how to create your own L-DSMS projects, it should be read carefully.

1.1 Basic Concept

L-DSMS could have been designed as a monolithic software, containing every necessary functionality. Such a system, however, can hardly be implemented because nobody knows every possible functionality somebody may ever need. Hence a concept is needed, that allows the user to enable or disable functions and allows him to extend L-DSMS easily with new functions, if necessary. A monolithic software doesn’t fulfill this condition and it would be difficult to extend and to maintain it. Users will find it very inefficient to analyze a full software system for creating a new functionality, if the logic of the functionality itself can be implemented in only a few lines of code.

A second approach is to distribute the complexity over different ‘micro components’. Every component provides only the necessary functionality for subproblems. It is therefore very small, easy to maintain, created within a few lines of code and highly specialized. But one component on its own is useless. To realize the required functionality, different components have to work together as a kind of production line (cf. Figure 1.1). Building up the required system by combining the necessary components is the basic functionality of L-DSMS itself.

At the very beginning of this production line is the raw input data, that is produced by any kind of data source (e.g. sensor). This input will be processed by the components one after another to be the final output after the last component has done its job. To change this process, components have to be replaced, removed or added, but it isn’t necessary to create a totally new system. This flexibility allows the user to deactivate unused functions and to run the system with minimal hardware requirements. Unlike in a real processing line, a component can have an arbitrary number of subsequent components, to allow parallel processing paths that can have a tree or even a graph structure. An advantage
of configurations with parallel processing paths is the increased stability of the system, because errors in one processing path don’t affect other paths.

L-DSMS was designed after the second approach and provides some core functionality to realize the concept of processing lines. The structure is automatically generated from a XML configuration document. This allows one to change the structure without the need of programming any line of code. The configuration document can be edited by every user with a text editor of its choice. More information about the configuration file, its layout and how to use it can be found in Section B.

1.2 Components

L-DSMS is based on a core system that is able to combine single components into a network of components. This is done based on the information in a configuration file. These connected components realize the real functionality. This section describes, what kind of component classes are available, how they work and what they are used for. A detailed description of each component in the L-DSMS core package is available in Chapter 5.
1.2. COMPONENTS

1.2.1 Sources

A source is always at the head of a production path and produces the data that needs to be processed. Each source produces this data in a different way, depending on its implementation. The L-DSMS core package contains sources that read data from files, sockets or external hardware (cf. Chapter 5). In most of the cases, a source receives its data from something external of L-DSMS (e.g. a sensor). Nevertheless, within the scope of L-DSMS we are recognizing a source as the data source without caring about where this data is originally from. This is the main difference to a node.

To provide additional information about the produced data, a source can create optional metainformation for each produced data package. The data together with its optional metadata forms the output of a source. This output would be useless, if no other component consumes it. Therefore, a source has to have at least one child component, that receives the output. Child components for a source can be an arbitrary number of drains or nodes (cf. Figure 1.3). Which drains or nodes are children for a source, is configured in the configuration file (cf. Section B). To connect a drain or node to a source, you have to consider that the output types (data and meta) of the source are compatible to all child component input types. This means, if Source $S_1$ produces data of type $t_1$ and metadata of type $t_2$, all connected drains and nodes have to accept data of type $t_1$ and metadata of type $t_2$. If not, L-DSMS produces an error during startup and may terminate. A drain or node always accepts data of type $t_1$ and metadata of type $t_2$ if its own data input type is $t_1$ or java.lang.Object and its own metadata input type is $t_2$ or java.lang.Object. Detailed descriptions about the input and output types for each component can be found in Chapter 5.

The configuration file contains, beside the relation between sources and their child components, some source specific properties. Two properties can be specified for all sources. The first property is the type itself. It is mandatory and has to be specified using the attribute class. The second property is optional and specifies the name of the source using the attribute name. If you specify the name, it has to be unique and no other component with the same name is allowed, because this name is used by drains or nodes to identify sources as additional parents. All other properties are source specific and are listed in the detailed description for each source. The ByteArrayFileSource (cf. Section 5.2.4), for example, has the additional attributes file, delay and repeat.

![Figure 1.3: source with one or more children](image-url)
1.2.2 Drains

A drain is a component, that consumes data and is always the tail of a production path. Within the scope of L-DSMS, we are recognising a drain as the final receiver of data, because it can’t have any child components inside of the system boundaries of L-DSMS. The data can be further processed by systems outside of L-DSMS. A SocketDrain, for example, forwards the incoming data to every process connected to this drain via a socket connection. This data doesn’t end in the SocketDrain but leaves the system boundaries of L-DSMS. The L-DSMS core package contains drains, that stores the incoming data to files, send them over socket connections or simply prints them to the console screen.

Drains can receive their data from one or more sources as well as nodes (cf. Figure 1.4). These connections between the drain and its parent components are configured in the configuration file. They are managed automatically by L-DSMS. As described in the previous section, a source can produce the data together with some corresponding metadata. Consequently, a drain can receive a data package as well as its corresponding metadata package, and both together are the input of the drain. To connect a drain with a source or node, the parent component has to produce output that can be processed by the drain. This means, that the data output type of the parent has to be compatible to the data input type of the drain and the metadata output type of the parent has to be compatible to the metadata input type of the drain. A drain always accepts data of type $t_1$ and metadata of type $t_2$ if its own data input type is $t_1$ or $\text{java.lang.Object}$ and its own metadata input type is $t_2$ or $\text{java.lang.Object}$. For a detailed description of the input and output types for each component take a look at Chapter 5.

Besides the relation between the drain and its parents the configuration file contains also specific properties. Two properties can be specified for all types. The first property is the type itself. Is mandatory and has to be specified using the attribute class. The second property is optional and specifies the names of additional sources using the attribute sourcerefs. If you specify sourcerefs, the names are used to specify the sources or nodes, identified by the names, as parent components for this drain. All other properties are drain specific and are listed in the detailed description for each drain. The ByteArrayFileDrain (cf. Section 5.2.3), for example, has the additional attribute file.

![Figure 1.4: drain with one or more parents](image-url)
1.3 LOGGING

1.2.3 Nodes

Nodes are a combination of sources and drains and can be located at every possible position within a production line. They receive data from one or more components, process them and forward the results to an arbitrary number of subsequent components. Every computation, that is done between a single source and all drains at the end of the processing line is done by a certain number of nodes. Consequently, because nodes are sources as well as drains, they inherit the properties of both of them. To connect a source (or node) with a node, you have to ensure, that the type of the outgoing data as well as the type of the outgoing metadata is compatible to the input types of this node. The same holds when connecting a node with a drain (or node). Nodes are configured the same way as sources and drains. Besides node specific properties, they all have the non-optional class attribute that is used to define the type. They can have a name (like a source), that is specified using the attribute name and additional sources (like a drain), that are specified using the sourcerefs attribute.

Each node available in the L-DSMS core package processes the data in a different way and therefore there is no general statement about the outcome of nodes. Some of them produces data with the same type as the input and others produces data of a totally different type. They even do not always produce output for every input they have received, but discard the received data if it does not fulfil their conditions (e.g. a filter).

![Diagram of nodes with an arbitrary number of parents and children](image)

Figure 1.5: Node with an arbitrary number of parents and children

1.3 Logging

L-DSMS runs as a server process without any user interaction. To ensure, that the system works correctly, the system administrators need to monitor it. Therefore L-DSMS can print logging messages to the console screen to inform the system administrator about internal events. These messages fall into the categories:

- INFO
- WARN
These categories are hierarchically ordered, beginning with INFO as the lowest and ends with DEBUG as the highest category. You can assign a logging category to each component separately, as well as a default logging level which is used for each component without an explicitly specified category.

INFO is the lowest level category and therefore contains no other categories. Info messages are used to inform the system administrator about usual and non-critical events (e.g. system started, component loaded, client connected/disconnected).

WARN messages are used to inform the system administrator about unusual and non-critical events. Using the WARN category includes all messages assigned to the INFO category (e.g. client is no more reachable, end of file reached).

ERROR messages are used to inform the system administrator about unusual and maybe critical events. The ERROR category includes all messages assigned to the INFO and WARN categories (e.g. illegal input data, host unreachable).

DEBUG messages are used especially by developers. They are used to inform a developer about what’s going on and to help him tracing errors and bugs. The DEBUG category includes all messages assigned to the INFO, WARN or ERROR category.
1.4 Plug-in

L-DSMS has a plug-in mechanism to enrich the core functionality of L-DSMS without the necessity to change any line of code of L-DSMS. The plug-ins can be hooked into L-DSMS just by adding them into the configuration file. This makes it possible to enable and disable functions as needed.

Figure 1.7: plug-ins hooked to the L-DSMS core
This chapter explains, how to set up L-DSMS on your Linux or Windows system. L-DSMS is written in Java. So it needs a working J2SE™ Runtime Environment (JRE) (version ≥ 5.0). Setting up a working JRE is described in the first section. The installation and configuration process of L-DSMS itself is described in the second section.

2.1 How to Install Java

This section describes, how to set up a running JRE 5.0 at your system. If you want to install L-DSMS on a Linux system, please proceed with 2.1.1. If you want to install L-DSMS on a Windows system, please proceed with 2.1.2.

2.1.1 Linux

check JRE version

If you think, there is already a JRE installed, please perform the following steps, to check if the version of the installed JRE fits our needs, otherwise skip this part. To run L-DSMS you need a version greater or equal to 1.5 (5.0 is a synonym for 1.5).

1. Open a console window (e.g. press ‘Alt’+‘F2’, type ‘xterm’ (cf. figure 2.1) and press ‘Enter’).

2. Type `java -version` in the console window and press Enter.

If the result looks like shown in figure 2.2, a JRE is already installed and correctly configured. In our case, the current version is 1.5.0_11, but every version ≥ 1.5 should work as well and you can skip this section and proceed with section 2.2.

If the result looks like shown in figure 2.3, no JRE is installed or your JRE is not configured correctly and you have to proceed with the following install instructions.
CHAPTER 2. INSTALLATION

Figure 2.1: run application

Figure 2.2: java -version (successful)

Figure 2.3: java -version (unsuccessful)
2.1. HOWTO INSTALL JAVA

Download the JRE offline installation file

Go to http://java.sun.com/javase/downloads/index_jdk5.jsp to download the latest release of JRE 5.0. To download the latest release of JRE instead, go to http://java.sun.com/javase/downloads/index.jsp. Press the download button for Java Runtime Environment (JRE) 5.0 (cf. figure 2.4) and follow the download instructions to save the file (e.g. at /Downloads/Java/).

![Java Runtime Environment (JRE) 5.0 Update 14](http://java.sun.com/javase/downloads/index_jdk5.jsp)

Figure 2.4: Download JRE 5.0

Run the installer

After downloading the installation package, open the folder where the file is located. Execute the installation package and follow the installation instructions.

Configure path environment

To use the JRE, you need to tell your system, where to find the JRE. If you run a command within the console (e.g. java -version), the system tries to find this executable in the current folder. If the executable isn’t located in the current folder, the system tries to find the executable in every folder that is listed in the PATH variable. So we need to add the JRE installation folder to the PATH variable. To set the PATH variable, follow these steps:

1. Open a console window (e.g. press ‘Alt’+‘F2’, type ‘xterm’ (cf. figure 2.1) and press ‘Enter’).
2. Type “vi ~/.bashrc” and press ‘Enter’.
3. Press ‘i’ to change to the ‘Insert’ mode.
4. Insert “export JAVA_HOME=\<install folder\>” where \<install folder\> is your JRE installation folter (e.g. ‘/Program Files/Java/jre1.5.0_11’) in a new line.
5. Search for the line, that starts with “export PATH” and append “:JAVA_HOME” to this line.
6. Press ‘Esc’ to stop the ‘Insert’ mode.
7. Type ‘:’ to switch to the ‘Command’ mode.
8. Press ‘x’ and press ‘Enter’ to save the changes and close the file.
9. Type ‘.’ ‘/bashrc’ to load the changes.
2.1.2 Windows (2000/XP)

check JRE version

If you think, there is already a JRE installed, please perform the following steps, to check if the version of the installed JRE fits our needs, otherwise skip this part. To run L-DSMS you need a version greater or equal to 1.5 (5.0 is a synonym for 1.5).

1. Click the start button (cf. figure 2.5).
2. Select “Run...” from the menu.
3. Type `cmd` in the window (cf. figure 2.6) and click OK. This opens a command prompt window.
4. Type `java -version` in the command prompt window and press Enter.

If the result looks like shown in figure 2.7, a JRE is already installed and correctly configured. In our case, the current version is 1.5.0_11, but every version ≥ 1.5 should work as well and you can skip this section and proceed with section 2.2.

If the result looks like shown in figure 2.8, no JRE is installed or your JRE is not configured correctly and you have to proceed with the following install instructions.

Download the JRE offline installation file

Go to [http://java.sun.com/javase/downloads/index_jdk5.jsp](http://java.sun.com/javase/downloads/index_jdk5.jsp) to download the latest release of JRE 5.0. To download the latest release of JRE instead,
2.1. HOWTO INSTALL JAVA

Figure 2.6: Run

![Image of Run window]

Figure 2.7: Version

![Image of Command Prompt]

go to http://java.sun.com/javase/downloads/index.jsp. Press the download button for Java Runtime Environment (JRE) 5.0 (cf. figure 2.9) and follow the download instructions to save the file (e.g. at C:\Downloads\Java\).

Run the installer

After downloading the installation package, open the folder where the file is located. Execute the installation package and follow the installation instructions.

Configure path environment

To use the JRE, you need to tell your system, where to find the JRE. If you run a command within a command prompt window (e.g. java -version), the system tries to find this executable in the current folder. If the executable isn’t
located in the current folder, the system tries to find the executable in every
folder that is listed in the PATH variable. So we need to add the JRE installation
folder to the PATH variable. To set the PATH variable, follow these steps:

1. Click the start button.
2. Open the ‘Control Panel’.
3. Go to System, select the ‘Advanced’ tab and click the ‘Environment Variables’ button (cf. figure 2.10).
4. In the region ‘System Variables’ click the ‘New’ button.
5. Set ‘Variable Name’ to ‘JAVA_HOME’ and ‘Variable Value’ to your JRE installation folder (e.g. “C:\Program Files\Java\jre1.5.0_11”, cf. figure 2.11) and click the ‘OK’ button (cf. figure 2.13).
6. In the region ‘System Variables’ select “Path” and click the ‘Edit’ button.
7. Add “;%JAVA_HOME%\bin” at the end of the ‘Variable value’ field (cf. figure 2.12) and click the ‘OK’ button (cf. figure 2.14).
2.1. HOW TO INSTALL JAVA

Figure 2.13: JAVA_HOME configured

Figure 2.14: JAVA_HOME and PATH configured
2.2 L-DSMS

This section describes step-by-step, how to set up L-DSMS at your system assuming that a working JS2ETM Runtime Environment is already available.

1. Download the latest release from the L-DSMS project page http://www.pms.ifi.lmu.de/reverse-vga1/ldsms/.

2. Unzip the file to your program folder (e.g. /Program Files/ or C:\Program Files\).

3. You now should have a new folder ‘ldsms’ in your program folder, we will refer to this folder using the <LDSMS_HOME> as a symbolic representation. This folder should contain
   - config (folder)
   - lib (folder)
   - ldsms.jar (file)

4. Download all additional libraries, listed in section A and save them in <LDSMS_HOME>/lib.

5. Test your installation with the Hello World example (section 3.2).
3.1 Preliminaries

This section contains general informations for you, that are useful in the following examples. To increase the clearness of the example descriptions, these informations are collected in the current subsections and will be referenced within the example descriptions as needed.

3.1.1 Path Information

All paths are given in the Linux syntax (with a slash ‘/’ as separator). Windows user need to replace the slash (‘/’) by a backslash (‘\’) and add a hard drive label (e.g. C:\). The first listing shows a path in Linux syntax.

Listing 3.1: Path for Linux users

/Program Files/ldsms/

The second listing shows the (semantic) same path for Windows users.

Listing 3.2: Path for Windows users

C:\\Program Files\ldsms\

3.1.2 Install Folder

In every example, the paths are relative to the L-DSMS installation folder (see section 2.2). Instead of writing the full path, we use ‘<LDSMS_HOME>’ as a symbolic representation for the installation folder. Assume, you installed L-DSMS in ‘/Program Files/ldsms/’. The path

Listing 3.3: absolute path

/Program Files/ldsms/examples/hello_world/

will be written as

Listing 3.4: relative path

<LDSMS_HOME>/examples/hello_world/
3.1.3 Open a Console

**Linux**

There are many ways to open a console in Linux, depending on different criteria like the installed window manager or the installed software. The following descriptions are only an extract of these numerous ways:

- general solution: press ‘Alt+F2’
- for Gnome users: click on ‘Applications’ at your menu bar and select ‘Run Application...’
- for KDE users: click on the KDE logo at your menu bar and select ‘Run command’

This opens a text prompt, that asks for the name of the program to execute. Type either ‘xterm’ or ‘konsole’ and press ‘Run’.

**Windows**

1. Click the start button (cf. figure 3.1).
2. Select “Run...” from the menu.
3. Type `cmd` in the window (cf. figure 3.2) and click OK. This opens a command prompt window (console)
3.2. **HELLO WORLD**

This example illustrates a very basic configuration, that prints “Hello World” onto the screen. Please follow the below mentioned steps, to set up the ‘Hello World’ example.

- Create a folder `<LDSMS_HOME>/examples/hello_world/` and open it.
- Create a file `input.txt` within this folder and open it with your text editor (e.g. `vi` for Linux, Notepad for Windows).
- Write “HELLO WORLD” into this file. Save and close it.
- Create a file `config.xml` within this folder and open it with your text editor.
- Insert the content from listing 3.5 (without the line numbers) into this file. Save and close it.
- Open a console and change to `<LDSMS_HOME>` (cf. section 3.1.3).
- Type “java -jar ldsms.jar examples/hello_world/config.xml” and press ‘ENTER’ (cf. listing 3.6).
- Press ‘Strg’+‘C’ to stop L-DSMS, after ‘HELLO WORLD’ has been printed.

Listing 3.5: examples/hello_world/config.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
```
CHAPTER 3. EXAMPLES

Listing 3.6: run the ‘Hello World’ example

```java
java −jar ldsms.jar examples/hello_world/config.xml
```

Explanation: In the first two steps a text file as input for L-DSMS has been created. In the third step, the configuration file for L-DSMS has been created. In line 7 in this configuration file, a StringFileSource (see section 5.2.15) has been specified. This FileSource reads every data from ‘examples/hello_world/input.txt’ (specified by the file attribute).

In line 8, a ConsoleDrain (see section 5.2.8) has been specified. A ConsoleDrain prints every input to the console. Because the ConsoleDrain is a child element of StringFileSource, every output of the StringFileSource (here the data from the text file) is passed to the input of the ConsoleDrain. This simple example shows how to print the content of the text file directly to the console.

3.3 Filtering Strings
This example illustrates a configuration, that uses a regular expression, to filter “Hello World” from a text file. Please follow the below-mentioned steps, to set up this example.

- Create a folder `<LDSMS_HOME>/examples/regex` and open it.
- Create a file `input.txt` within this folder and open it with your text editor (e.g. vi for Linux, Notepad for Windows).
- Insert the content from listing 3.7 (with the line breaks) into this file. Save and close it.
- Create a file `config.xml` within this folder and open it with your text editor.
- Insert the content from listing 3.8 (without the line numbers) into this file. Save and close it.
- Open a console and change to `<LDSMS_HOME>` (cf. section 3.1.3).
- Type “java -jar ldsms.jar examples/regex/config.xml” and press ‘ENTER’ (cf. listing 3.6).
- Press ‘Strg’+’C’ to stop L-DSMS, after ‘Hello world.’ has been printed.

Listing 3.7: examples/regex/input.txt
```
Hello
Kitty
is
loved
by
people
all
over
the
world.
```

Listing 3.8: examples/regex/config.xml
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
   <logging level="INFO" />
   <services>
      <network>
         <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/filter/input.txt"/>
         <node class="de.lmu.ifi.pms.ldsms.generics.Filter"/>
         <condition class="de.lmu.ifi.pms.ldsms.generics.RegexCondition" data="Hello|world.*"/>
         <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
      </network>
   </services>
</server>
```
CHAPTER 3. EXAMPLES

Listing 3.9: run the ‘Regex’ example

```
java -jar ldsms.jar examples/hello_world/config.xml0 [main]
    INFO de.lmu.ifi.pms.ldsms.network.Server - Configuring Server...
238 [main] INFO de.lmu.ifi.pms.ldsms.network.Server - Initializing Server...
238 [main] INFO de.lmu.ifi.pms.ldsms.network.Server - Starting Server...
    Hello world.
1081 [Server stopp thread] INFO de.lmu.ifi.pms.ldsms.network.Server - Stopping Server...
```

Explanation: The first step was to create a text file as input for L-DSMS. In the next step, the configuration file for L-DSMS has been created. In line 7 in this configuration file, a StringFileSource (see section 5.2.15) has been specified. This FileSource reads every data from ‘examples/regex/input.txt’ (specified by the file attribute).

In line 8 a Filter has been specified. This Filter has a RegexCondition that checks if the incoming data matches the specified regular expression (specified by the data attribute).

At line 9, a ConsoleDrain (see section 5.2.8) is specified. A ConsoleDrain will print every input to the console. Because the ConsoleDrain is a child element of Filter, every output of it (the filtered data) will be passed to the input of the ConsoleDrain.

### 3.4 Filtering XML stream

This example illustrates a configuration, that filters XML data using SpexNode.
• Create a folder `<LDSMS_HOME>/examples/xml/` and open it.

• Create a file `input.xml` within this folder and open it with your text editor (e.g. vi for Linux, Notepad for Windows).

• Insert the content from listing 3.10 (without the line numbers) into this file. Save and close it.

• Create a file `config.xml` within this folder and open it with your text editor.

• Insert the content from listing 3.11 (without the line numbers) into this file. Save and close it.

• Open a console and change to `<LDSMS_HOME>` (cf. section 3.1.3).

• Type “java -jar ldsms.jar examples/xml/config.xml” and press ‘ENTER’ (cf. listing 3.12).

• Press ‘Strg’+‘C’ to stop L-DSMS, after the result has been printed.

Listing 3.10: examples/xml/input.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<cdlist>
  <cd>
    <name>Yellow Submarine</name>
    <artist>The Beatles</artist>
    <tracklist>
      <track id="1">Yellow Submarine</track>
      <track id="2">Love You Too</track>
      <track id="3">Eleanor Rigby</track>
      <track id="4">Lucy In The Sky With Diamons</track>
      <track id="5">Hey Bulldog</track>
      <track id="6">Think For Yourself</track>
      <track id="7">All Together Now</track>
      <track id="8">Sargent Pepper Lonely Hearts Club Band</track>
      <track id="9">With A Little Help For My Friends</track>
      <track id="10">Baby You Are A Richman</track>
      <track id="11">Only A Northern Song</track>
      <track id="12">All You Need Is Love</track>
      <track id="13">When I’m Sixty Four</track>
      <track id="14">Nowhere Man</track>
      <track id="15">It’s All Too Much</track>
    </tracklist>
  </cd>
  <cd>
    <name>Greatest Hits</name>
    <artist>The Police</artist>
    <tracklist>
      <track id="1">Roxanne</track>
    </tracklist>
  </cd>
</cdlist>
```
CHAPTER 3. EXAMPLES

Listing 3.11: examples/xml/config.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO" />
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/xml/input.xml" repeat="true">
      </node>
    </network>
  </services>
</server>
```

Listing 3.12: run the ‘SpexNode’ example

```
java -jar ldsms.jar examples/hello_world/config.xml0 [main]
INFO de.lmu.ifi.pms.ldsms.network.Server - Configuring Server...
238 [main] INFO de.lmu.ifi.pms.ldsms.network.Server - Initializing Server...
```
3.4. FILTERING XML STREAM

Explanation: In the first step a XML document as input for L-DSMS has been created. The next step has created a configuration file for L-DSMS. In line 7 in this configuration file a StringFileSource (see section 5.2.15) has been specified. This StringFileSource reads every data from the XML document ‘examples/xml/input.txt’ (specified by the file attribute).

In line 8 a SpexNode has been specified. This SpexNode filters the incoming XML data, using the XPath expression from the xpath attribute.

In line 9 a ConsoleDrain (see section 5.2.8) has been specified. A ConsoleDrain prints every input to the console. Because the ConsoleDrain is a child element of SpexNode, every output of it (the filtered XML data) is passed to the input of the ConsoleDrain.
VISU-L-DSMS is the graphical user interface for L-DSMS that was developed to ease the management of L-DSMS. It can manage instances of L-DSMS located at the same host as VISU-L-DSMS as well as instances located on remote hosts. This allows to manage running instances of L-DSMS from a single management workstation, using the available network infrastructure. The following paragraphs explain how to use VISU-L-DSMS to manage running L-DSMS instances.

Install VISU-L-DSMS

VISU-L-DSMS is based on L-DSMS, so there has to be a fully functional L-DSMS installation on the system first (cf. Chapter 2), the rest is as easy as installing L-DSMS. To install VISU-L-DSMS

- go to the L-DSMS project webpage [http://www.pms.ifi.lmu.de/reverse-wga1/ldsms],
- download the latest visu-l-dsms.zip and
- extract this archive to your L-DSMS installation folder.

Advice:
To manage a L-DSMS instance with VISU-L-DSMS, it needs to be started with the RMIPlugin (cf. Section 5.4) activated.

Start VISU-L-DSMS

Linux as well as Windows users can start VISU-L-DSMS by simply executing a script that is shipped with VISU-L-DSMS. This script executes all necessary instructions and the user does not have to care about the details.

Linux user: go to <LDSMS_HOME> and execute visu-l-dsms.sh
Windows user: go to <LDSMS_HOME> and execute visu-l-dsms.cmd

Executing the script opens the VISU-L-DSMS main window (cf. Figure 4.1). This main window contains three areas and one menu panel. At the left side is the graphic representation of all components that are used and their relationship to each other. We call it the **Network View**. Each component is represented as a node (colored symbol) and their relationships to each other by edges (black lines).

At the right side is the **Capturing View**. This area contains two subwindows. The upper one is used to show the incoming data of a component selected in the network area, and the lower one is used to show the outgoing data of a component (the same or another one).

In the bottom area, there are two tabs that provide additional informations about the selected components of the network window (the **Overview Tab**) and their properties (the **Properties tab**).

These three areas are empty until VISU-L-DSMS is connected to a running L-DSMS instance. This is explained in the following paragraph.

**Connect VISU-L-DSMS to an instance of L-DSMS**

To manage an instance of L-DSMS, a connection between VISU-L-DSMS and this instance has to be established first. This is done by selecting the **Connect** item from the **Server** menu (cf. Figure 4.2). A connection dialog will be opened (cf. Figure 4.3) where the necessary connection parameters for the L-DSMS instance have to be specified. The connection parameters are:

- hostname (optional, default = localhost)
- port (optional, default = 1099)
- name of the instance (not optional, no default value)

The L-DSMS instance name was configured as a parameter of the RMIPlugin (cf. Section 5.4). If you don’t know this name, please refere to the configuration
file of the instance of your interest or contact your local L-DSMS administrator.

The Overview Tab
The overview tab (cf. Figure 4.5) shows the names of all used components in a tree structure with sources being the parents and their drains being the children (recursive). Each component is listed only once, even if it has more than one source. In this case, it is only listed once as a child of the first of its sources. If a component is selected in the network area, its representation in the overview tab will be selected also. The same counts vice versa. So if a component is selected in the overview tab, it’s representation in the network area will be selected also. This gives a compact overview about the network structure while the names of the components are listed in the overview tab. To select more than one component, hold down the CTRL button.

The Properties Tab
The properties tab (cf. Figure 4.6) is used to display and edit the properties for the last component that has been selected. If a component is selected in the network area, the properties tab updates itself automatically and displays the properties for this component. Each property is presented as either
CHAPTER 4. MANAGING L-DSMS WITH VISU-L-DSMS

Figure 4.5: overview tab

- a text field,
- a list or
- a check box.

Figure 4.6: properties register

Not all properties are editable, therefore some values can be edited and others not. Which property is editable, and which is not depends on the selected component. Editing values does not affect the L-DSMS instance until the changes have been saved (cf. Figure 4.7). The reload button (cf. Figure 4.8) can be used to revoke changes before they have been saved.

The Capturing Area

The capturing area is used for monitoring the incoming and outgoing data of the components. To capture data, select one or more components, either in the network or the overview area, open the node menu and select start listening (cf. Figure 4.9). This opens a tab for the incoming data in the upper subwindow and a tab for the outgoing data in the lower subwindow for each selected component.
Figure 4.7: save property changes

Figure 4.8: reload properties

Figure 4.9: open capture window for selected component

Figure 4.10: capture component input/output
Each tab has the same functionality to manage the capturing process. The buffer size field (cf. Figure 4.11) defines the number of characters that can be captured, until the oldest data has to be deleted to free space for new data. The clear button (cf. Figure 4.12) can be used to delete all captured data from the textarea.

The filter field can be used, to filter the data, that fulfills the given regular expression. Only the data matching the regular expression is displayed and the rest is hidden. No data is lost but the user gets a better overview with filtering. After deactivating the filter, all captured data is visible again.

The pause button (cf. Figure 4.15) can be used to pause the capturing. No more data will be captured while the capture window is in pause mode, but the captured data stays visible. In pause mode, the pause button changes its appearance (cf. Figure 4.16) and can be used to restart the capturing.

If a capture window isn’t needed anymore, the close button (cf. Figure 4.17) can be used to close it.
Figure 4.14: capture window with active filter

Figure 4.15: pause capturing

Figure 4.16: continue capturing

Figure 4.17: close capture window
5.1 Overview by packagename

This section gives you a brief overview over the components available in L-DSMS. This makes it possible to search for components by function. The detailed descriptions for each component are available in Section 5.2. The exact subsection and page are listed along with each component.

Remark:
The current package name for L-DSMS core is ‘de.lmu.ifi.pms.ldsms’. This may change in the future, therefore the package names are listed as \texttt{<ldsms core package>}.\texttt{<subpackage>}.  

5.1.1 \texttt{<ldsms core package>}.generics

This package provides a core functionality for L-DSMS.
5.1.2  `<ldsms core package>-.string`

This package provides components for processing text data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegexCondition</td>
<td>Condition</td>
<td>5.3.6</td>
<td>66</td>
</tr>
<tr>
<td>RepeatedStringCondition</td>
<td>Condition</td>
<td>5.3.7</td>
<td>67</td>
</tr>
<tr>
<td>StringReplaceNode</td>
<td>Node</td>
<td>5.2.16</td>
<td>58</td>
</tr>
<tr>
<td>StringTokenizerNode</td>
<td>Node</td>
<td>5.2.19</td>
<td>62</td>
</tr>
</tbody>
</table>

5.1.3  `<ldsms core package>-.xml`

This package provides components for processing XML data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SpexNode</td>
<td>Node</td>
<td>5.2.12</td>
<td>53</td>
</tr>
</tbody>
</table>
5.2 Core Components

This section contains an alphabetic overview with a detailed description of each component, in the L-DSMS core package. For Sources, the data/metadata input types are listed. For Drains, the data/metadata output types are listed. For Nodes, the data/metadata input and output types are listed. If it is possible to parametrize the input or output types no types are listed. By parametrizing a class, you can specify the input or output types in the configuration file as needed.

5.2.1 Buffer

| Package: | de.lmu.ifi.pms.ldsms.generics |
| Type: | Node |
| Input data type: | parametrizable by attribute ‘data’ |
| Input meta type: | parametrizable by attribute ‘meta’ |
| Output data type: | same as input data |
| Output meta type: | same as input meta |
| Dependencies: | no additional external library needed |
| Abstract: | A Buffer caches information until it can be delivered to underlying consumers (drains). It can be used to cache data until one ore more drains are able to consume it. The recived content is sent according to the first in first out principle (FIFO). This means, if the data $D_1$ was recived before data $D_2$, data $D_1$ is sent before data $D_2$. |

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>String</td>
<td>No</td>
<td>Specifies the data input/output type.</td>
<td></td>
</tr>
<tr>
<td>meta</td>
<td>String</td>
<td>No</td>
<td>Specifies the meta input/output type.</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
<td></td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
<td></td>
</tr>
</tbody>
</table>

Example:

Listing 5.1: Example

```xml
<?xml version="2.0" encoding="ISO-8859-1" ?>
```
CHAPTER 5. COMPONENTS

This example creates an instance of Buffer, that caches every incoming data of type String and incoming metadata of type Object. New data is appended to the end of the cache. If at least one consumer (drain) is connected to the StringSocketDrain (at port 6543), the cache is emptied beginning with its head. To test the Buffer (and receive the cached data), open a telnet connection to the StringSocketDrain (e.g. ‘telnet localhost 6543’).

5.2.2 ByteArray2String

Package: de.lmu.ifi.pms.ldsms.generics
Type: Node
Input data type: byte array (byte[])
Input meta type: byte array (byte[])
Output data type: java.lang.String
Output meta type: java.lang.String
Dependencies: no additional external library needed
Abstract: A ByteArray2String node transforms incoming byte arrays into strings, using the specified encoding format. It can be used, to connect a source with a drain, where the source produces byte arrays but the drain is expecting strings.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoding</td>
<td>String</td>
<td>ISO-8859-1</td>
<td>Yes</td>
<td>Specifies the encoding format used to transform a byte array into a string.</td>
</tr>
</tbody>
</table>
### 5.2. CORE COMPONENTS

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

**Example:**

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">  
    <logger name="de.lmu.ifi.pms.ldsms.generics.ByteArray2String" level="DEBUG" />
  </logging>
  <services>
    <source class="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" file="examples/hello_world/input.bin">  
      <node class="de.lmu.ifi.pms.ldsms.generics.ByteArray2String" encoding="ISO-8859-1">  
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
      </node>
    </source>
  </services>
</server>
```

This example reads the binary data from ‘examples/hello_world/input.bin’ and forwards it to an instance of ByteArray2String, where it is converted from ByteArrays to Strings, using the encoding ‘ISO-8859-1’. The instance of ConsoleDrain is used to print the result to the console.

#### 5.2.3 ByteArrayFileDrain

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Drain  
**Input data type:** byte array (byte[])  
**Input meta type:** byte array (byte[])  
**Dependencies:** no additional external library needed
CHAPTER 5. COMPONENTS

Abstract: A ByteArrayFileDrain drain writes the incoming byte arrays into the specified file. The byte arrays are written the stream as follows:

- the first four bytes describe the length of the data array as an integer, let it be \( n \).
- the next \( n \) bytes make up the data array.
- the next four bytes describe the length of the metainfo array as an integer, let it be \( m \).
- the next \( m \) bytes make up the metainfo array.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the path of the destination file.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separted list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing 5.3: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.iqms.gens.ByteArrayFileDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.iqms.gens.StringFileSource" file="examples/hello_world/input.txt">
        <node class="de.lmu.ifi.pms.iqms.gens.String2ByteArray">
          <drain class="de.lmu.ifi.pms.iqms.gens.ByteArrayFileDrain" file="examples/hello_world/output.bin" />
        </node>
      </source>
    </network>
  </services>
</server>
```
This example reads the data from `examples/hello_world/input.txt` and forwards it to an instance of String2ByteArray, where it is converted from Strings to byte arrays. These byte arrays are forwarded to an instance of ByteArrayFileDrain, that writes the incoming data to `examples/hello_world/output.bin`. For examining the result, open the output file.

### 5.2.4 ByteArrayFileSource

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Source  
**Output data type:** byte array (byte[])  
**Output meta type:** byte array (byte[])  
**Dependencies:** no external library needed  
**Abstract:** ByteArrayFileSource produces binary data by reading it from a filestream. The data is received in the form of byte arrays. The byte arrays are read from the stream as follows:

- the first four bytes describe the length of the data array as an integer, let it be n.
- the next n bytes make up the data array.
- the next four bytes describe the length of the metainfo array as an integer, let it be m.
- the next m bytes make up the metainfo array.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>Integer</td>
<td>100</td>
<td>Yes</td>
<td>Specifies a delay (in ms). After reading a part of the file, the thread sleeps for the specified time, before the next part is read.</td>
</tr>
<tr>
<td>file</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the path of the source file.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>repeat</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, whether the reading should restart after the end of file was reached.</td>
</tr>
</tbody>
</table>
Example:

Listing 5.4: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" level="DEBUG"/>
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" file="examples/hello_world/input.bin" delay="1000" repeat="true">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
      </source>
    </network>
  </services>
</server>
```

This example reads the binary data from ‘examples/hello_world/input.bin’ with a delay of 1 second. If, during reading, the end of file is reached, ByteArrayFileSource repeats the reading at the begin of the file. The instance of ConsoleDrain is used to print the data to the console.

### 5.2.5 ByteArraySocketDrain

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Drain  
**Input data type:** byte array (byte[])  
**Input meta type:** byte array (byte[])  
**Dependencies:** no additional external library needed  
**Abstract:** A ByteArraySocketDrain drain writes the incoming byte arrays to every client, connected at the specified port. The byte arrays are written to the stream as follows:

- the first four bytes describe the length of the data array as an integer, let it be n.
- the next n bytes make up the data array.
- the next four bytes describe the length of the metainfo array as an integer, let it be m.
- the next m bytes make up the metainfo array.

**Attributes:**
### 5.2. CORE COMPONENTS

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>period</td>
<td>Integer</td>
<td>1</td>
<td>Yes</td>
<td>Specifies the interval in seconds, the drain repeats looking for locked up client connections.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td></td>
<td>No</td>
<td>Specifies the port at which the SocketDrain accepts client connections.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

#### Example:

Listing 5.5: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" file="examples/hello_world/input.bin">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ByteArraySocketDrain" port="6543" />
      </source>
    </network>
  </services>
</server>
```

This example reads the binary data from ‘examples/hello_world/input.bin’ and forwards it to ByteArraySocketDrain. ByteArraySocketDrain sends the input to every client connected at port 6543. The example for ByteArraySocketSource (cf. subsection 5.2.6) can be used, to test this example.

### 5.2.6 ByteArraySocketSource

Package: de.lmu.ifi.pms.ldsms.generics
Type: Source
Output data type: byte array (byte[])
Output meta type: byte array (byte[])
Dependencies: no additional external library needed
**Abstract:**

ByteArraySocketSource produces binary data by reading it from a socket connection. The data is received in the form of byte arrays. The byte arrays are read from the stream as follows:

- the first four bytes describe the length of the data array as an integer, let it be n.
- the next n bytes make up the data array.
- the next four bytes describe the length of the metainfo array as an integer, let it be m.
- the next m bytes make up the metainfo array.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-Name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the hostname for the socket connection.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>No</td>
<td></td>
<td>Specifies the portnumber for the socket connection.</td>
</tr>
<tr>
<td>retry</td>
<td>Integer</td>
<td>1000</td>
<td>Yes</td>
<td>Specifies the interval for reconnect attempts, if the connection is broken.</td>
</tr>
</tbody>
</table>

**Example:**

Listing 5.6: Example

```xml
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.ByteArraySocketSource" level="DEBUG" />
  </logging>
  <services>
    <source class="de.lmu.ifi.pms.ldsms.generics.ByteArraySocketSource" host="localhost" port="6543">
      <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
    </source>
  </services>
</server>
```
This example receives the binary data from localhost (port 6543). The instance of ConsoleDrain is used to print the data to the console. You can use this configuration, to test the example for ByteArraySocketDrain (cf. subsection 5.2.5).

### 5.2.7 Cast

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Node  
**Input data type:** java.lang.Object  
**Input meta type:** java.lang.Object  
**Output data type:** parametrizable by attribute ‘data’  
**Output meta type:** parametrizable by attribute ‘meta’  
**Dependencies:** no additional external library needed  
**Abstract:**

A Cast node casts the input to the specified type and filters out any incompatible data or meta objects. It can be used to connect a source with a drain, where the output types of the source are different but compatible to the input types of the drain. This works if for data and metadata one of the following cases (doesn’t need to be the same) matches:

1. The input type can be casted to the output type. Therefore, it is a subtype of the output type.
2. The output type is either String, Boolean, Integer, Double or Float and the input can be transformed to this type.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the data output type.</td>
</tr>
<tr>
<td>meta</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the metadata output type.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>
Example:

Listing 5.7: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.Cast" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/numbers/input.txt">
        <node class="de.lmu.ifi.pms.ldsms.generics.Cast" data="Integer" meta="Object">
          <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
        </node>
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of Cast, that forwards every incoming data and metadata after converting the data to Integer.

### 5.2.8 ConsoleDrain

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Drain  
**Input data type:** java.lang.Object  
**Input meta type:** java.lang.Object  
**Dependencies:** no additional external library needed  
**Abstract:** A ConsoleDrain prints every incoming data without any formatting to the console.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-Name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>printData</td>
<td>Boolean</td>
<td>true</td>
<td>Yes</td>
<td>Specifies, if the data should be printed to the console.</td>
</tr>
<tr>
<td>printMeta</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, if the metadata should be printed to the console.</td>
</tr>
</tbody>
</table>
### 5.2. Core Components

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

**Example:**

Listing 5.8: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/hello_world/input.txt">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" />
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of StringFileSource, that reads from examples/hello_world/input.txt. The strings read from the file, are printed to the console by the ConsoleDrain.

### 5.2.9 Filter

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Node  
**Input data type:** depends on selected conditions  
**Input meta type:** depends on selected conditions  
**Output data type:** same as input data  
**Output meta type:** same as input meta  
**Dependencies:** no additional external library needed  
**Abstract:** A Filter tests whether the data and metainfo fullfills a certain condition and passes it to its drains only if the condition is fullfilled. Otherwise, the data and metainfo is omitted. Section 5.3 describes the filter, filter conditions and how to use them in detail.

**Attributes:**

...
### 5.2.10 ObjectSocketDrain

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Drain  
**Input data type:** java.lang.Object

This example reads the data from ‘examples/filter/input.txt’ and forwards it to an instance of Filter. This Filter contains only one condition, the RegexCondition (cf. subsection 5.3.6) that drops each line, that doesn’t contain either ‘Hello’ or ‘world’. The instance of ConsoleDrain is used to print the result to the console.
### Input meta type:
java.lang.Object

### Dependencies:
no additional external library needed

### Abstract:
An ObjectSocketDrain writes the incoming data and metadata into the output stream of a socket (in this order). The incoming objects must implement the java.io.Serializable interface. If no client is connected, the data is dropped.

### Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>period</td>
<td>Integer</td>
<td>1</td>
<td>Yes</td>
<td>Specifies the interval in seconds, the drain repeats looking for locked up client connections.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td></td>
<td>No</td>
<td>Specifies the port at which the SocketDrain accepts client connections.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

### Example:

Listing 5.10: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.ObjectSocketDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="test/Hello_World.txt" interval="0" repeat="true" forwardEOL="true">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ObjectSocketDrain" port="6543" />
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of ObjectSocketDrain, that recieves its incoming data and metadata from a StringFileSource and forwards everything to each client connected at port 6543.
5.2.11 ObjectSocketSource

Package: de.lmu.ifi.pms.ldsms.generics
Type: Source
Output data type: parametrizable by attribute ‘data’
Output meta type: parametrizable by attribute ‘meta’
Dependencies: no additional external library needed
Abstract:
ObjectSocketSource produces data by reading objects from a socket connection. The objects are read from the stream as follows:

- at first, the data object is read and casted to the specified data output type
- after this the metadata object is read and casted to the specified metadata output type

The ObjectSocketSource tries to cast the incoming data and metadata to their specified types. If either the incoming data or metadata could not be casted, both is dropped.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the data output type.</td>
</tr>
<tr>
<td>meta</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the metadata output type.</td>
</tr>
<tr>
<td>host</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the hostname for the socket connection.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>No</td>
<td></td>
<td>Specifies the portnumber for the socket connection.</td>
</tr>
<tr>
<td>retry</td>
<td>Integer</td>
<td>1000</td>
<td>Yes</td>
<td>Specifies the interval for reconnect attempts, if the connection is broken.</td>
</tr>
</tbody>
</table>

Example:

Listing 5.11: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging_level="INFO"/>
</server>
```
This example creates an instance of ObjectSocketSource, that builds up a connection to localhost at port 6543. Every data object received, is casted to a String.

### 5.2.12 SpexNode

**Package:** de.lmu.ifi.pms.ldsms.xml  
**Type:** Node  
**Input data type:** java.lang.String  
**Input meta type:** java.lang.Object  
**Output data type:** java.lang.String  
**Output meta type:** java.lang.Void  
**Dependencies:**

- **name:** **SPEX:** XPath Evaluation against XML Streams  
- **version:** Spex 1.0 (or higher)  

**Abstract:** Filters data from a XML stream, using the de.lmu.ifi.pms.spex.main.SpexProcessor and a XPath expression.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>
### Attribute-Name: xpath

**Type:** String  
**Default:** No  
**Optional:** Yes  
**Description:** The XPath expression for querying the XML stream.

#### Example:

Listing 5.12: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.xml.SpexNode"
      level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/xml/TMCMessage.xml" repeat="true">
        <node class="xml.SpexNode" xpath="/descendant::TMCMessage[descendant::eventName/child::text()='Unfall']"/>
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
      </node>
    </network>
  </services>
</server>
```

This example reads XML data from examples/xml/TMCMessage.xml and filters all the events of type ‘Unfall’ using SpexNode. The instance of ConsoleDrain is used to print the result to the console.

### 5.2.13 String2ByteArray

- **Package:** de.lmu.ifi.pms.ldsms.generics
- **Type:** Node
- **Input data type:** java.lang.String
- **Input meta type:** java.lang.String
- **Output data type:** byte array (byte[])
- **Output meta type:** byte array (byte[])
- **Dependencies:** no additional external library needed
5.2. CORE COMPONENTS

Abstract: String2ByteArray takes as input Strings and forwards them as byte arrays. The parameter ‘encoding’ can be used to specify the encoding format (e.g. US-ASCII, UTF-8, UTF-16BE, UTF-16LE, UTF-16). More information about valid encoding formats can be found in the specification of java.nio.charset.Charset at http://java.sun.com/j2se/1.5.0/docs/api/java.nio.charset/Charset.html.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-Name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encoding</td>
<td>String</td>
<td>ISO-8859-1</td>
<td>Yes</td>
<td>Specifies which encoding format should be used to transform the incoming Strings into byte arrays.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing 5.13: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.String2ByteArray" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/hello_world/input.txt" interval="0" repeat="true" forwardEOL="true">
        <node class="de.lmu.ifi.pms.ldsms.generics.String2ByteArray">
          <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" />
        </node>
      </source>
    </network>
  </services>
</server>
```
This example creates an instance of String2ByteArray, that receives its incoming data and metadata from a StringFileSource. After encoding the incoming strings using ISO-8859-1, everything is forwarded to the ConsoleDrain. The ConsoleDrain prints every incoming data to the console.

### 5.2.14 StringFileDrain

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Drain  
**Input data type:** java.lang.String  
**Input meta type:** java.lang.Object  
**Dependencies:** no additional external library needed  
**Abstract:** A StringFileDrain writes the incoming data strings into the specified file using UTF-8 encoding. The metadata is not written into the specified file. The metadata is only used to indicate that the corresponding data is a GoodbyeMessage. This data won’t be written immediately but at last to the specified file, when the StringFileDrain stops.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the path of the destination file.</td>
</tr>
<tr>
<td>sourcerels</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

**Example:**

```
Listing 5.14: Example

<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.StringFileDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/hello.world/" />
    </network>
  </services>
</server>
```
This example reads the data from ‘examples/hello_world/input.txt’ and copies it to ‘examples/hello_world/output.txt’ using StringFileDrain.

### 5.2.15 StringFileSource

**Package:** de.lmu.ifi.pms.ldsms.generics  
**Type:** Source  
**Output data type:** java.lang.String  
**Output meta type:** java.lang.Void  
**Dependencies:** no external library needed  
**Abstract:** StringFileSource reads a file as a list of strings. It does not send any metainfo information, but merely the content of the file line by line. It also sends the ‘\n’ newline delimiter.

#### Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>Integer</td>
<td>100</td>
<td>Yes</td>
<td>Specifies a delay (in ms). After reading a part of the file, the thread sleeps for the specified time, before the next part is read.</td>
</tr>
<tr>
<td>file</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the path of the source file.</td>
</tr>
<tr>
<td>forwardEOL</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, if line terminators is eliminated (forwardEOL=false) or forwarded to the drains. forwardEOL is only used, when ignoreEOL is set to false.</td>
</tr>
<tr>
<td>ignoreEOL</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, if each line is sent seperatly (ignoreEOL=false), or as much characters as possible should be send.</td>
</tr>
</tbody>
</table>
### Attribute-name

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>repeat</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, whether the reading should restart after the end of file was reached.</td>
</tr>
</tbody>
</table>

#### Example:

**Listing 5.15: Example**

```xml
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.StringFileSource" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/hello_world/input.txt" interval="0" repeat="true">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" />
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of StringFileSource, that reads from examples/hello_world/input.txt (relative to the L-DSMS startup location) with a delay of 1 second. The strings read from the file, are printed to the console by the ConsoleDrain. If the end of file is reached during reading, the StringFileSource repeats the reading at the beginning of the file.

#### 5.2.16 StringReplaceNode

- **Package:** de.lmu.ifi.pms.ldsms.string
- **Type:** Source
- **Output data type:** java.lang.String
- **Output meta type:** java.lang.String
- **Dependencies:** no external library needed
Abstract: The StringReplaceNode can be used to do basic text transformations on an input stream. It is configured using a regular expression in the format `<match ex.> / <replace ex.>`.

The match expression specifies, which parts should be replaced.

The replace expression specifies, how the matched parts should be replaced.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>regex</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the match and the replace expression in the format <code>&lt;match ex.&gt; / &lt;replace ex.&gt;</code></td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing 5.16: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
    <logging level="INFO">
        <logger name="de.lmu.ifi.pms.ldsms.string.
            StringReplaceNode" level="DEBUG" />
    </logging>
    <services>
        <network>
            <source class="de.lmu.ifi.pms.ldsms.generics.
                StringFileSource" file="examples/regex/input.txt
                " ignoreEOL="true">
                <node class="de.lmu.ifi.pms.ldsms.string.
                    StringReplaceNode" regex="(\w*)(e|o)(\w*)(o|e)
                    (\w*)/$1$4$3$2$5">
                <drain class="de.lmu.ifi.pms.ldsms.generics.
                    ConsoleDrain" linebreak="false" />
            </node>
        </network>
    </services>
```
This example reads the data from 'examples/regex/input.txt' using String-FileDrain and changes the order of 'e' and 'o' within each word. The instances of ConsoleDrain are used to show the difference between the unchanged and the changed data and produce an output similar to the following one:

```
75 [main] DEBUG de.lmu.ifi.pms.ldsms.string.
    StringReplaceNode — Match expression set to (\w*)(e|o)
                       (\w*)(o|e)(\w*).
75 [main] DEBUG de.lmu.ifi.pms.ldsms.string.
    StringReplaceNode — Replace expression set to $1$4$3$2$5.
102 [main] INFO de.lmu.ifi.pms.ldsms.network.Server — Initializing Server...
103 [main] INFO de.lmu.ifi.pms.ldsms.network.Server — Starting Server...
Holle Kitty is levod by peeplo all evor the world.
Hello Kitty is loved by people all over the world.
```

### 5.2.17 StringSocketDrain

<table>
<thead>
<tr>
<th>Package:</th>
<th>de.lmu.ifi.pms.ldsms.generics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Drain</td>
</tr>
<tr>
<td>Input data type:</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>Input meta type:</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>Dependencies:</td>
<td>no additional external library needed</td>
</tr>
<tr>
<td>Abstract:</td>
<td>A StringSocketDrain drain writes the incoming byte strings to every client, connected at the specified port.</td>
</tr>
</tbody>
</table>

#### Attributes:

<table>
<thead>
<tr>
<th>Attribute-</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>period</td>
<td>Integer</td>
<td>1</td>
<td>Yes</td>
<td>Specifies the interval in seconds, the drain repeats looking for locked up client connections.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>No</td>
<td></td>
<td>Specifies the port at which the SocketDrain accepts client connections.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>
Example:

Listing 5.17: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.StringSocketDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="test/Hello_World.txt"
        interval="0" repeat="true" forwardEOL="true">
        <drain class="de.lmu.ifi.pms.ldsms.generics.StringSocketDrain" port="6543" />
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of StringSocketDrain, that recieves its incoming data and metadata from a StringFileSource. The incoming strings are sent to every client connected at port 6543.

### 5.2.18 StringSocketSource

<table>
<thead>
<tr>
<th>Package:</th>
<th>de.lmu.ifi.pms.ldsms.generics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Source</td>
</tr>
<tr>
<td>Output data type:</td>
<td>java.lang.String</td>
</tr>
<tr>
<td>Output meta type:</td>
<td>java.lang.Object</td>
</tr>
<tr>
<td>Dependencies:</td>
<td>no additional external library needed</td>
</tr>
<tr>
<td>Abstract:</td>
<td>StringSocketSource recieves the data from a socket connection as Strings.</td>
</tr>
</tbody>
</table>

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the hostname for the socket connection.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>port</td>
<td>Integer</td>
<td>No</td>
<td></td>
<td>Specifies the portnumber for the socket connection.</td>
</tr>
</tbody>
</table>
### Attribute-Name | Type | Default | Optional | Description
--- | --- | --- | --- | ---
retry | Integer | 1000 | Yes | Specifies the interval for reconnect attempts, if the connection is broken.

Example:

Listing 5.18: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.generics.StringSocketSource" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringSocketSource" host="alice" port="6543"
        retry="5000">
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" />
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of StringSocketSource, that builds up a socket connection to host alice at port 6543. If the connection fails, reconnect attempts are repeated every 5 seconds.

#### 5.2.19 StringTokenizerNode

**Package:** de.lmu.ifi.pms.ldsms.string  
**Type:** Source  
**Output data type:** java.lang.String  
**Output meta type:** java.lang.String  
**Dependencies:** no external library needed
5.2. CORE COMPONENTS

Abstract:
This node can be used, to break a large string (e.g. produced by an instance of StringFileSource) into smaller pieces (tokens). Every part of the input, that matches the given regular expression, is interpreted as a delimiter. Each token is forwarded separately, but the delimiters are eliminated.

There are three different policies to handle the metadata:

- drop the metadata
  → metaPolicy="drop"
- repeat it with the first token
  → metaPolicy="repeatOnce"
- repeat it with every token
  → metaPolicy="repeat"

Attributes:

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metaPolicy</td>
<td>String</td>
<td>drop</td>
<td>Yes</td>
<td>Specifies, which forward policy should be used. Valid values are ‘drop’, ‘repeatOnce’ and ‘repeat’.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>regex</td>
<td>String</td>
<td></td>
<td>No</td>
<td>Specifies the expression, that matches the delimiters.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing 5.19: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.string.StringTokenizerNode" level="DEBUG" />
  </logging>
</server>
```
CHAPTER 5. COMPONENTS

This example reads the data from ‘examples/regex/input.txt’ using String-FileDrain and breaks it into smaller pieces at every whitespace. The instances of ConsoleDrain are used to show the difference between the unsplitted and the splitted data and produce an output similar to the following one:

```
0 [main] INFO de.lmu.ifi.pms.ldsms.network.Server – Configuring Server...
76 [main] DEBUG de.lmu.ifi.pms.ldsms.string.StringTokenizerNode – Regex set to \s.
80 [main] DEBUG de.lmu.ifi.pms.ldsms.string.StringTokenizerNode – Metadata will be dropped.
106 [main] INFO de.lmu.ifi.pms.ldsms.network.Server – Initializing Server...
107 [main] INFO de.lmu.ifi.pms.ldsms.network.Server – Starting Server...
Hello Kitty
is loved
by people
all over
the world.
Hello Kitty is loved by people all over the world.
```

5.3 FilterConditions

FilterConditions are used by Filter nodes to verify, that the incoming data fulfills the required conditions. The Filter nodes ask the FilterConditions, if
they would discard or accept a given input. If the input is forwarded or not, is based on these information.

5.3.1 AndCondition

Package: de.lmu.ifi.pms.lsms.generics
Output data type: depends on the types of the children
Output meta type: depends on the types of the children
Dependencies: no additional external library needed
Abstract: AndCondition provides a method for combining several conditions by a boolean and. The request is forwarded to each of these conditions. The AndCondition accepts the input only, if all subconditions accept the input.

5.3.2 FalseCondition

Package: de.lmu.ifi.pms.lsms.generics
Output data type: java.lang.Object
Output meta type: java.lang.Object
Dependencies: no additional external library needed
Abstract: A false condition always returns false.

5.3.3 NotCondition

Package: de.lmu.ifi.pms.lsms.generics
Output data type: same as the child condition
Output meta type: same as the child condition
Dependencies: no additional external library needed
Abstract: A NotCondition negates the result of it’s child condition. So the real checking is done by the child condition and NotCondition only forwards the request.

5.3.4 OrCondition

Package: de.lmu.ifi.pms.lsms.generics
Output data type: depends on the types of the children
Output meta type: depends on the types of the children
Dependencies: no additional external library needed
Abstract: OrCondition provides a method for combining several conditions by a boolean or. The request is forwarded to each of these conditions. The OrCondition accepts the input, if at least one subcondition accepts the input.
5.3.5  RDSGroupFilterCondition

Package:  de.lmu.ifi.pms.ldsms.generics
Output data type:  de.lmu.ifi.pms.ldsms.rds.RDSGroup
Output meta type:  de.lmu.ifi.pms.ldsms.rds.RDSGroupMetInfo
Dependencies:  no additional external library needed
Abstract:  RDSGroupFilterCondition provides methods for the filtering of RDS-TMC messages, based on their group identifier.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the group identifier of groups, that should be accepted.</td>
</tr>
</tbody>
</table>

5.3.6  RegexCondition

Package:  de.lmu.ifi.pms.ldsms.string
Output data type:  java.lang.String
Output meta type:  java.lang.String
Dependencies:  no additional external library needed
Abstract:  RegexCondition provides a method for filtering strings, using regular expressions. RegexCondition only accepts the input, if the input data matches the regular expression for data and the input metadata matches the regular expression for metadata.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies the regular expression, the incoming data has to match to be accepted. If no regular expression is specified for data, every incoming data is accepted.</td>
</tr>
<tr>
<td>meta</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies the regular expression, the incoming metadata has to match to be accepted. If no regular expression is specified for metadata, every incoming metadata is accepted.</td>
</tr>
</tbody>
</table>
5.4 Plugins

The L-DSMS core functionality is to manage and process data streams by a chain of processing nodes. Other functions can be added as plugins, using the L-DSMS plugin mechanism. The L-DSMS plugin mechanism is configured, like everything else, with the configuration file (cf. Section B). Therefore, adding and removing plugins is as easy as configuring the components and L-DSMS can be executed with a minimal set of functions, preserving resources.

RMIPlugin is the plugin, that is shipped with the L-DSMS core package. Using this plugin, enables L-DSMS to be managed by VISU-L-DSMS.

Example:

Listing 5.20: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO" />
  <plugin-list>
    <plugin-prefix>de.lmu.ifi.pms.ldsms.plugin</plugin-prefix>
    <plugin-prefix>de.lmu.ifi.pms.ldsms.network</plugin-prefix>
    <plugin class="de.lmu.ifi.pms.ldsms.plugin.RMIPlugin"
      registry-name="LDSMSServer" security-policy="" server.policy="" stub-class="de.lmu.ifi.pms.ldsms.remote.ConfigurationFacadeRemoteAdapter" rmi-codebase="file:/LDSMS_HOME/ldsms.jar file:/LDSMS_HOME/lib/avalon-framework-4.2.0.jar"/>
  </plugin-list>
</server>
```
</plugin-list>
<services>
  <network>
    ...
  </network>
</services>
</server>
This section lists all additional Java libraries with their download locations, that L-DSMS needs to work properly. Libraries, that are not necessary for the core functionality, but needed by optional components, are listed within the component documentation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Version</th>
<th>Vendor</th>
<th>Download Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log4j</td>
<td>≥ 1.2</td>
<td>Apache Software Foundation</td>
<td><a href="http://logging.apache.org/log4j/1.2/index.html">http://logging.apache.org/log4j/1.2/index.html</a></td>
</tr>
<tr>
<td>Commons Configuration</td>
<td>≥ 1.0.4</td>
<td>Apache Software Foundation</td>
<td><a href="http://commons.apache.org/configuration/">http://commons.apache.org/configuration/</a></td>
</tr>
</tbody>
</table>

Table A.1: External Libraries
All configuration informations for L-DSMS are in an external configuration file, that has to be a valid XML document. L-DSMS will use that configuration file during start-up to configure itself. You can specify:

- the plugins, L-DSMS should use
- the logging level for L-DSMS in general and for each specific component
- the network structure (the used components and their relationship to each other)

The following Document Type Definition (DTD) illustrates the structure, the configuration file has to follow.

```xml
<!ELEMENT server (logging, plugin-list?, services)>
<!ELEMENT logging (logger*)>  
<!ATTLIST logging level (OFF|FATAL|ERROR|WARN|INFO|DEBUG|ALL) #REQUIRED>
<!ELEMENT logger EMPTY>
<!ATTLIST logger name CDATA #REQUIRED>
<!ATTLIST logger level (OFF|FATAL|ERROR|WARN|INFO|DEBUG|ALL) #REQUIRED>
<!ELEMENT plugin-list (plugin-prefix*, plugin*)>
<!ELEMENT plugin-prefix (#PCDATA)>
<!ELEMENT plugin ANY>  
<!ATTLIST plugin class CDATA #REQUIRED>
<!ELEMENT services (network)>
<!ELEMENT network (source+, node*)>
<!ATTLIST source class CDATA #REQUIRED>
<!ATTLIST source name ID #IMPLIED>
<!ATTLIST node class CDATA #REQUIRED>
<!ATTLIST node name ID #IMPLIED>
```

For an example, take a look at chapter 3 (Examples).
HowTo: Implement A Component

C

This chapter describes, how to implement a new component. Section C.1 describes how to implement a new Drain. Section C.2 describes how to implement a new Source and Section C.3 describes how to implement a new Node.

C.1 Drains

Every component, that should be handled as a Drain by L-DSMS, has to implement, either direct or indirect, the interface `<ldsms core package>.network.Drain` (cf. Figure C.1).

A detailed description for every method is located within the JavaDoc at the L-DSMS project page (http://www.pms.ifi.lmu.de/reverse-wga1/ldsms/).

The abstract class ‘DrainImpl’ (cf. Figure C.2) already implements the interface ‘Drain’. To ease the implementation of a Drain, it should be used as the superclass for new Drains.

Drains that need additional attributes from the configuration file, additionally have to implement the interface

org.apache.avalon.framework.configuration.Configurable

This ensures, that the configuration informations from the configuration file are passed to the Drain.

Drains that need to be started or stopped, additionally have to implement the interface

org.apache.avalon.framework.activity.Stoppable

This ensures, that the server starts the Drain after all configuration has been done and that the server stops the Drain if the system is supposed to terminate. This is useful, if additionally threads are used or streams have to be opened and closed.
APPENDIX C. HOWTO: IMPLEMENT A COMPONENT

Figure C.1: ‘Drain’ interface

Figure C.2: Abstract class ‘DrainImpl’
package test;

import de.lmu.ifi.pms.ldsms.network.DrainImpl;

public class PrintLnDrain extends DrainImpl<String, String> {
    public PrintLnDrain() {
        super(String.class, String.class);
    }

    public boolean isReady() {
        return true;
    }

    public void consume(String data, String meta) {
        System.out.println(String.format("Data: %s", data));
        System.out.println(String.format("Meta: %s", meta));
    }

    public String getDescription() {
        return "A simple drain that prints the input to the console."
    }

    public String getShortDescription() {
        return "PrintLnDrain";
    }

    public String toString() {
        return "PrintLnDrain";
    }
}

C.2 Sources

Every component, that should be handled as a Source by L-DSMS, has to implement, either direct or indirect, the interface \texttt{ldsms core package}.network.Source (cf. Figure C.3).

A detailed description for every method is located within the JavaDoc at the L-DSMS project page (\url{http://www.pms.ifi.lmu.de/rewerse-wga1/ldsms/}).

The abstract class \texttt{SourceImpl} (cf. Figure C.4) already implements the interface \texttt{Source}. To ease the implementation of a Source, it should be used as the superclass for new Sources.
Figure C.3: ‘Source’ interface
C.2. SOURCES

Figure C.4: Abstract class ‘SourceImpl’

```java
+ SourceImpl()
+ SourceImpl(Class<Data>, Class<Meta>)
+ addDrain(Configuration, String): Drain<Data, Meta>
+ addDrain(Drain<Data, Meta>)
+ configure(Configuration)
+ drainList(): List<Drain<Data, Meta>>
+ getDescription(): String
+ getName(): String
+ getOutputDataType(): Class<Data>
+ getOutputMetaType(): Class<Meta>
+ getProperties(): List<Property>
+ getShortDescription(): String
+ isCreated(): boolean
+ isNotSending(): boolean
+ isSending(): boolean
+ isStarted(): boolean
+ isStopped(): boolean
+ listDrains(StringBuffer, String)
+ removeDrain(Drain<Data, Meta>)
+ setProperty(Property)
+ start()
+ stop()
+ update(Observable, Object)
# canSend(): boolean
# configureType(Configuration)
# send(Data, Meta)
# startSending()
# stopSending()
```
package test;

import de.lmu.ifi.pms.ldsms.network.SourceImpl;

public class HelloSource extends SourceImpl<String, String> {

    private Thread helloThread;

    public HelloSource() {
        super(String.class, String.class);
    }

    public void start() throws Exception{
        super.start();
        startSending();
    }

    public void startSending() {
        super.startSending();
        try {
            helloThread = new Thread() {
                public void run() {
                    while (!isStopped()) {
                        try {
                            send("Hello", null);
                            Thread.sleep(5000);
                        } catch (InterruptedException ie){
                            break;
                        }
                    }
                };
            helloThread.start();
        } catch (Exception e) {}
    }

    public String getDescription() {
        return "A simple Source that sends 'Hello' every 5 seconds."
    }

    public String getShortDescription() {
        return "HelloSource";
    }

    public String toString() {
        return "HelloSource";
    }
}
C.3 Nodes

Nodes are a combination of a Drain and a Source. Every component, that should be handled as a Node (Source and Drain) by L-DSMS, has to implement, either direct or indirect, the interfaces `<lsm package>.network.Drain` (cf. Figure C.1) and `<lsm package>.network.Source` (cf. Figure C.3).

The abstract class ‘Node’ (cf. Figure C.5) already implements both interfaces. To ease the implementation of Nodes, it should be used as the superclass for new Nodes.

Figure C.5: Abstract class ‘Node’
Listing C.3: A simple Node example
D.1 Overview by packagename

This section gives you a brief overview over the components available in L-DSMS. This makes it possible to search for components by function. The detailed descriptions for each component are available in Section D.2. The exact subsection and page are listed along with each component.

Remark:
The current package name for L-DSMS core is ‘de.lmu.ifi.pms.ldsms’. This may change in the future, therefore the package names are listed as <ldsms core package>.<subpackage>.

D.1.1 <ldsms core package>.rds

This package provides components for processing RDS data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteArray2RDSBlock</td>
<td>Node</td>
<td>D.2.1</td>
<td>83</td>
</tr>
<tr>
<td>ByteArray2RDSGroup</td>
<td>Node</td>
<td>D.2.2</td>
<td>84</td>
</tr>
<tr>
<td>RDSBlock2ByteArray</td>
<td>Node</td>
<td>D.2.7</td>
<td>93</td>
</tr>
<tr>
<td>RDSBlock2RDSGroup</td>
<td>Node</td>
<td>D.2.8</td>
<td>95</td>
</tr>
<tr>
<td>RDSGroup2ByteArray</td>
<td>Node</td>
<td>D.2.9</td>
<td>96</td>
</tr>
<tr>
<td>RDSGroup2RDSGroup</td>
<td>Node</td>
<td>D.2.10</td>
<td>98</td>
</tr>
<tr>
<td>RDSGroup2RDSGroup3A</td>
<td>Node</td>
<td>D.2.11</td>
<td>100</td>
</tr>
<tr>
<td>RDSGroup8AMultiGroupLinker</td>
<td>Node</td>
<td>D.2.13</td>
<td>103</td>
</tr>
</tbody>
</table>

D.1.2 <ldsms core package>.rds.easyway

This package provides components for processing data produced by an EasyWay RDS receiver.
## D.1.3 `<ldsms core package>.tmc`

This package provides components for processing TMC data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>OTNDrain</td>
<td>Drain</td>
<td>D.2.6</td>
<td>91</td>
</tr>
<tr>
<td>RDSGroup2TMCMessage</td>
<td>Node</td>
<td>D.2.12</td>
<td>101</td>
</tr>
<tr>
<td>TMCMessageManagement</td>
<td>Node</td>
<td>D.2.15</td>
<td>107</td>
</tr>
</tbody>
</table>

## D.1.4 `<ldsms core package>.xml`

This package provides components for processing XML data.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMCMessage2XML</td>
<td>Node</td>
<td>D.2.14</td>
<td>105</td>
</tr>
<tr>
<td>XML2TMCMessage</td>
<td>Node</td>
<td>D.2.16</td>
<td>108</td>
</tr>
</tbody>
</table>
D.2. Components

D.2.1 ByteArray2RDSBlock

Package: de.lmu.ifi.pms.ldsms.rds
Type: Node
Input data type: byte array (byte[])
Input meta type: byte array (byte[])
Output data type: de.lmu.ifi.pms.ldsms.rds.RDSBlock
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSBlockMetaInfo
Dependencies: no additional external library needed
Abstract: Reads a RDSBlock and its Metainformation from two byte arrays (byte[]). The first array (with a length of three) represents the data and is interpreted as follows:

- data byte 0: Block id
- data byte 1: Block high byte
- data byte 2: Block low byte

The second array (with a length of six) represents the metadata and is interpreted as follows:

- meta byte 0-3: Frequency
- meta byte 4: number of Errors
- meta byte 5: bit 0 wether all errors are corrected

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.ByteArray2RDSBlock" level="DEBUG" />
  </logging>
</server>
```
This example reads the binary data from ‘examples/rds/RDSBlock.bin’ and forwards it to an instance of ByteArray2RDSBlock, where it is converted from ByteArrays to RDSBlocks. The instance of ConsoleDrain is used to print the result to the console.

D.2.2 ByteArray2RDSGroup

<table>
<thead>
<tr>
<th>Package:</th>
<th>de.lmu.ifi.pms.ldsms.rds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Node</td>
</tr>
<tr>
<td>Input data type:</td>
<td>byte array (byte[])</td>
</tr>
<tr>
<td>Input meta type:</td>
<td>byte array (byte[])</td>
</tr>
<tr>
<td>Output data type:</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroup</td>
</tr>
<tr>
<td>Output meta type:</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo</td>
</tr>
<tr>
<td>Dependencies:</td>
<td>no additional external library needed</td>
</tr>
<tr>
<td>Abstract:</td>
<td>Reads a RDSGroup and its Metainformation from two byte arrays (byte[][]). The first array (with a length of 13) represents the data and is interpreted as follows:</td>
</tr>
</tbody>
</table>
D.2. COMPONENTS

- data byte 0: Block A id
- data byte 1: Block A high byte
- data byte 2: Block A low byte
- data byte 3: Block B id
- data byte 4: Block B high byte
- data byte 5: Block B low byte
- data byte 6: Block C id
- data byte 7: Block C high byte
- data byte 8: Block C low byte
- data byte 9: Block D id
- data byte 10: Block D high byte
- data byte 11: Block D low byte

The second array (with a length of six) represents the metadata and is interpreted as follows:

- meta byte 0-3: Frequency
- meta byte 4: number of Errors
- meta byte 5: bit 0 wether all errors are corrected

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

**Example:**

Listing D.2: Example

```xml
<xml version="1.0" encoding="ISO-8859-1" ?>
```
This example reads the binary data from ‘examples/rds/RDSGroup.bin’ and forwards it to an instance of ByteArray2RDSGroup, where it is converted from ByteArrays to RDSGroups. The instance of ConsoleDrain is used to print the result to the console.

### D.2.3 EasyWayFileSource

**Package:** de.lmu.ifi.pms.ldsms.rds.easyway  
**Type:** Source  
**Output data type:** de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunk  
**Output meta type:** de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunkMetaInfo  
**Dependencies:** no additional external library needed  
**Abstract:** Emulates an EasyWaySource by reading the RDS-TMC raw data from a source file.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>String</td>
<td></td>
<td>No</td>
<td>Specifies the path of the source file.</td>
</tr>
<tr>
<td>frequency</td>
<td>Integer</td>
<td></td>
<td>No</td>
<td>Specifies a delay (in ms). After reading a part of the file, the thread sleeps for the specified time, before the next part is read.</td>
</tr>
</tbody>
</table>
### D.2. COMPONENTS

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>repeat</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Specifies, whether the reading should restart after the end of file was reached.</td>
</tr>
<tr>
<td>version</td>
<td>String</td>
<td></td>
<td>No</td>
<td>Specifies the version of the attached simulated Easyway light RDS receiver.</td>
</tr>
</tbody>
</table>

**Example:**

Listing D.3: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayFileSource" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayFileSource" file="examples/easyway/input.bin" interval="100" frequency="95.1" version="1.0"> 
        <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain" />
      </source>
    </network>
  </services>
</server>
```

This example simulates an EasyWay light RDS reciever of version ‘1.0’, that is tuned to the frequency 95.1Mhz. The data is read from ‘examples/easyway/input.bin’ every 100ms. The instance of ConsoleDrain is used to print the data to the console.

### D.2.4 EasyWayRDSChunk2RDSBlock

- **Package:** de.lmu.ifi.pms.ldsms.rds.easyway
- **Type:** Source
- **Input data type:** EasyWayRDSChunk
- **Input meta type:** EasyWayRDSChunkMetaInfo
Output data type: de.lmu.ifi.pms.ldsms.rds.RDSBlock
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSBlockMetaInfo
Dependencies: no additional external library needed
Abstract: Converts incoming EasyWayRDSChunk into RDS-Block. This is done by creating a new RDSBlock using the EasyWayRDSChunk block id and the EasyWayRDSChunk current high and low byte. The EasyWayRDSChunk block id is transformed into a RDS-Block id as follows:

- 0x00 \(\rightarrow\) (byte)'a'
- 0x20 \(\rightarrow\) (byte)'b'
- 0x40 \(\rightarrow\) (byte)'c'
- 0x60 \(\rightarrow\) (byte)'d'
- 0x80 \(\rightarrow\) (byte)'C'
- 0xA0 \(\rightarrow\) (byte)'e'
- 0xC0 \(\rightarrow\) (byte)'E'
- 0xE0 \(\rightarrow\) (byte)'I'

The EasyWayRDSChunk current high and low byte are copied without any transformations. The metainformation is forwarded unchanged.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.4: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO"/>
```
This example simulates an EasyWay light RDS receiver of version ‘1.0’, that is tuned to the frequency 95.1Mhz. The data is read from ‘examples/easyway/input.bin’ every 100ms and transformed from EasyWayRDSChunk to RDSBlock. The instance of ConsoleDrain is used to print the data to the console.

D.2.5 EasyWaySource

Package: de.lmu.ifi.pms.ldsms.rds.easyway
Type: Source
Output data type: de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunk
Output meta type: de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunkMetaInfo
Abstract: Reads RDS data from an EasyWay Light RDS-TMC receiver and produces EasyWayRDSChunk. An EasyWayRDSChunk encapsulates a chunk (block) of raw RDS data, built of exactly seven bytes.
APPENDIX D. EXAMPLE APPLICATION

Byte 0: [BL2, BL1, BL0, SNYC, DOFL, RSTD, ELB1, ELB0]
Byte 1: [M15, M14, M13, M12, M11, M10, M09, M08]
   Current HIGH byte
Byte 2: [M07, M06, M05, M04, M03, M02, M01, M00]
   Current LOW byte
Byte 3: [P15, P14, P13, P12, P11, P10, P09, P08]
   Previous HIGH byte
Byte 4: [P07, P06, P05, P04, P03, P02, P01, P00]
   Previous LOW byte
Byte 5: [BEC5, BEC4, BEC3, BEC2, BEC1, BEC0, EPB1, EPB0]
Byte 6: [BP2, BP1, BP0, undefined, SQI3, SQI2, SQI1, SQI0]

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>device</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the device name of the RDS-TMC reciever within the system (e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/dev/ttyS0).</td>
</tr>
<tr>
<td>dx</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Not used.</td>
</tr>
<tr>
<td>frequency</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Specifies the frequency (in MHz), the RDS-TMC reciever is atuned (e.g.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>98.1).</td>
</tr>
<tr>
<td>mono</td>
<td>Boolean</td>
<td>false</td>
<td>Yes</td>
<td>Indicates, whether the signal should be received in mono.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>component.</td>
</tr>
<tr>
<td>searchLevel</td>
<td>Integer</td>
<td>15</td>
<td>Yes</td>
<td>Specifies the sensibility for the signal strength of automatic searched</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>frequencies.</td>
</tr>
</tbody>
</table>
## D.2. COMPONENTS

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
</table>
| search         | String | down    | Yes      | Indicates, if the next available frequency should be automatically searched. Valid values are ‘up’ and ‘down’.

### Example:

Listing D.5: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWaySource" level="DEBUG"/>
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWaySource" device="/dev/ttyS0" frequency="95.1">
        <node class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
      </source>
    </network>
  </services>
</server>
```

This example creates an instance of EasyWaySource, that reads the data from an EasyWay Light RDS-TMC reciever, which is registered under ‘/dev/ttyS0’ and tuned to 95.1MHz. The instance of ConsoleDrain is used to print the data to the console.

### D.2.6 OTNDrain

Package: de.lmu.ifi.pms.ldsms.tmc
Type: Drain
Input data type: java.util.List&lt;TMCMessage&gt;
Input meta type: java.util.List
Dependencies: no additional external library needed
Abstract: Creates an OWL representation of the given data and writes it to ‘Ontologie/ transformation/TMCEvents.owl’ in the specified base dir. The metainformation is not used.
Attributes:
D.2. COMPONENTS

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dir</td>
<td>String</td>
<td>No</td>
<td></td>
<td>The base directory. This directory is used to find the ontologie file ‘Ontologie/Transformation/TMCEvents.owl’</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.6: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.OTNDrain" level="DEBUG" />
  </logging>
  <services>
    <network>
    </network>
  </services>
</server>
```

D.2.7 RDSBlock2ByteArray

Package: de.lmu.ifi.pms.ldsms.rds
Type: Node
Input data type: de.lmu.ifi.pms.ldsms.rds.RDSBlock
Input meta type: de.lmu.ifi.pms.ldsms.rds.RDSBlockMetaInfo
Output data type: byte array (byte[])
Output meta type: byte array (byte[])
Dependencies: no additional external library needed
Abstract:  
Transforms a RDSBlock and its Metainformation into two byte arrays (byte[]). The first array (with a length of three) represents the data and its structure is as follows:

- data byte 0: Block id
- data byte 1: Block high byte
- data byte 2: Block low byte

The second array (with a length of six) represents the metadata and its structure is as follows:

- meta byte 0-3: Frequency
- meta byte 4: number of Errors
- meta byte 5: bit 0 wether all errors are corrected

Attributes:

<table>
<thead>
<tr>
<th>Attribute-Name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.7: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.RDSBlock2ByteArray" level="DEBUG"/>
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWaySource" device="/dev/ttyS0" frequency="95.1"/>
      <node class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunk2RDSBlock"/>
```
This example creates an instance of EasyWaySource, that reads the data from an EasyWay Light RDS-TMC receiver, which is registered under ‘/dev/ttyS0’ and tuned to 95.1MHz. The data is first grouped to RDSBlocks. These RDSBlocks are transformed to byte arrays, to be stored to ‘examples/rds/RDSBlock.bin’ as binary data.

D.2.8 RDSBlock2RDSGroup

Package: de.lmu.ifi.pms.ldsms.rds
Type: Node
Input data type: de.lmu.ifi.pms.ldsms.rds.RDSBlock
Input meta type: de.lmu.ifi.pms.ldsms.rds.RDSBlockMetaInfo
Output data type: de.lmu.ifi.pms.ldsms.rds.RDSGroup
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo
Dependencies: no additional external library needed
Abstract: Collects single RDSBlock and RDSBlockMetaInfo elements, groups them to a RDSGroup and RDSGroupMetaInfo and passes both together to the children.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.8: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
```
This example creates an instance of EasyWaySource, that reads the data from an EasyWay Light RDS-TMC receiver, which is registered under '/dev/ttyS0' and tuned to 95.1MHz. The data is first grouped to RDSBlocks and then to RDSGroups. The instance of ConsoleDrain is used to print the result to the console.

### D.2.9 RDSGroup2ByteArray

<table>
<thead>
<tr>
<th>Package:</th>
<th>de.lmu.ifi.pms.ldsms.rds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type:</td>
<td>Node</td>
</tr>
<tr>
<td>Input data type:</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroup</td>
</tr>
<tr>
<td>Input meta type:</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo</td>
</tr>
<tr>
<td>Output data type:</td>
<td>byte array (byte[])</td>
</tr>
<tr>
<td>Output meta type:</td>
<td>byte array (byte[])</td>
</tr>
<tr>
<td>Dependencies:</td>
<td>no additional external library needed</td>
</tr>
</tbody>
</table>
Abstract: Transforms a RDSGroup and its meta information into two byte arrays (byte[]). The first array (with a length of 13) represents the data and its structure is as follows:

- data byte 0: Block A id
- data byte 1: Block A high byte
- data byte 2: Block A low byte
- data byte 3: Block B id
- data byte 4: Block B high byte
- data byte 5: Block B low byte
- data byte 6: Block C id
- data byte 7: Block C high byte
- data byte 8: Block C low byte
- data byte 9: Block D id
- data byte 10: Block D high byte
- data byte 11: Block D low byte

The second array (with a length of six) represents the metadata and is interpreted as follows:

- meta byte 0-3: Frequency
- meta byte 4: number of Errors
- meta byte 5: bit 0 wether all errors are corrected

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:
This example creates an instance of EasyWaySource, that reads the data from an EasyWay Light RDS-TMC receiver, which is registered under ‘/dev/ttyS0’ and tuned to 95.1MHz. The data is first grouped to RDSBlocks and then to RDSGroups. These RDSGroups are transformed to byte arrays, to be stored to ‘examples/rds/RDSGroup.bin’ as binary data.

D.2.10 RDSGroup2RDSGroup

<table>
<thead>
<tr>
<th>Package</th>
<th>de.lmu.ifi.pms.ldsms.rds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Node</td>
</tr>
<tr>
<td>Input data type</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroup</td>
</tr>
<tr>
<td>Input meta type</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo</td>
</tr>
<tr>
<td>Output data type</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroup</td>
</tr>
<tr>
<td>Output meta type</td>
<td>de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo</td>
</tr>
<tr>
<td>Dependencies</td>
<td>no additional external library needed</td>
</tr>
</tbody>
</table>
Abstract: Tries to convert the given RDSGroup into one of the following RDS groups:

- RDSGroup0A
- RDSGroup2A
- RDSGroup3A
- RDSGroup4A
- RDSGroup8A

Data that can’t be transformed into one of the mentioned groups, is dropped.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.10: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
    <logging level="INFO">
        <logger name="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWaySource" level="DEBUG" />
    </logging>
    <services>
        <network>
            <source class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWaySource" device="/dev/ttyS0" frequency="95.1">
                <node class="de.lmu.ifi.pms.ldsms.rds.easyway.RDSBlock2RDSGroup">
                    <node class="de.lmu.ifi.pms.ldsms.rds.RDSGroup2RDSGroup">
                        <node class="de.lmu.ifi.pms.ldsms.rds.RDSGroup">
```
This example creates an instance of EasyWaySource, that reads the data from an EasyWay Light RDS-TMC receiver, which is registered under ‘/dev/ttyS0’ and tuned to 95.1MHz. The data is first grouped to RDSBlocks and then to RDSGroups. The instance of ConsoleDrain is used to print the result to the console.

### D.2.11 RDSGroup2RDSGroup3A

**Package:** de.lmu.ifi.pms.ldsms.rds  
**Type:** Node  
**Input data type:** de.lmu.ifi.pms.ldsms.rds.RDSGroup  
**Input meta type:** de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo  
**Output data type:** de.lmu.ifi.pms.ldsms.rds.RDSGroup3A  
**Output meta type:** de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo  
**Dependencies:** no additional external library needed  
**Abstract:** If the given RDSGroup is a RDSGroup3A or can be converted into a RDSGroup3A and the application group type is 16 (0x10), the input is sent to all connected consumers as a new RDSGroup3A, otherwise it is dropped.

**Attributes:**

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

**Example:**

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
```
This example does exactly the same as described in abstract. The instance of ConsoleDrain is used to print the result to the console.

D.2.12  RDSGroup2TMCMessage

Package:  de.lmu.ifi.pms.ldsms.rds
Type:  Node
Input data type:  de.lmu.ifi.pms.ldsms.rds.RDSGroup
Input meta type:  de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo
Output data type:  de.lmu.ifi.pms.ldsms.tmc.TMCMessage
Output meta type:  de.lmu.ifi.pms.ldsms.tmc.TMCMessageMetaInfo
Dependencies:  no additional external library needed
Abstract: This class maps the content of a RDSGroup8A into a convenient data structure that also reflects the "multiple information blocks" substructure of a RDS-Group8A Multigroup. The Data structure reflects the semantic separation of single elements in the additional content and the scopes of the single optional message content elements. Basically, the content is split into scopes by separator elements, but there is an additional scoping rule for destinations, events and quantifiers.

The following rules for scoping apply:

- The elements that may occur only once in a RDSMultiGroup are considered as having a global scope for the whole group.
- An event is considered as having a scope that reaches from the event to the next separator or event, whichever comes first. This scope determines the association of quantifiers to events.
- The destination element has a scope that reaches to the next destination element or separator, whichever comes first. If a diversion element precedes the destination, the scope of the destination ends after the diversion element.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventDB</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Name (and path) of the file, that stores the event list as a CSV table. This table is used, to transform the event codes into a user readable representation.</td>
</tr>
<tr>
<td>locales</td>
<td>String</td>
<td>de</td>
<td>Yes</td>
<td>A comma separated list of locales (e.g. ‘de’ for german, ‘en’ for english) that represent the languages, the event code should be translated.</td>
</tr>
<tr>
<td>locationDB</td>
<td>String</td>
<td>No</td>
<td></td>
<td>Name (and path) of the file, that stores the location list as a CSV table. This table is used, to transform the location codes into a user readable representation.</td>
</tr>
</tbody>
</table>
### Attribute-Description Table

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

### Example:

```
Listing D.12: Example

<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.RDSGroup2TMCMessage" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" file="examples/rds/RDSGroup.bin">
        <node class="de.lmu.ifi.pms.ldsms.rds.ByteArray2RDSGroup">
          <node class="de.lmu.ifi.pms.ldsms.rds.RDSGroup2TMCMessage">
            <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
          </node>
        </node>
      </source>
    </network>
  </services>
</server>
```

This example does exactly the same as described in abstract. The instance of ConsoleDrain is used to print the result to the console.

### D.2.13 RDSGroup8AMultiGroupLinker

- **Package:** de.lmu.ifi.pms.ldsms.rds
- **Type:** Node
- **Input data type:** de.lmu.ifi.pms.ldsms.rds.RDSGroup
- **Input meta type:** de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo
- **Output data type:** de.lmu.ifi.pms.ldsms.rds.RDSGroup
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSGroupMetaInfo

Dependencies: no additional external library needed

Abstract: RDSGroup8AMultiGroupLinker provides methods for linking RDS-TMC multi group messages together. Data, that is not of type RDSGroup8A, is forwarded without any modifications. Data, that is of type RDSGroup8A, is linked to the current first RDSGroup8A. If the given RDSGroup8A is a first group itself, the old one is sent and the given one is abrogated.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma seperated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.13: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>

<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.rds.RDSGroup2RDSGroup" level="DEBUG" />
  </logging>
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.ByteArrayFileSource" file="examples/rds/RDSGroup.bin">
        <node class="de.lmu.ifi.pms.ldsms.rds.ByteArray2RDSGroup"/>
      </source>
      <source class="de.lmu.ifi.pms.ldsms.rds.RDSGroup2RDSGroup">
        <node class="de.lmu.ifi.pms.ldsms.rds.RDSGroup2RDSGroup"/>
      </source>
      <source class="de.lmu.ifi.pms.ldsms.rds.RDSGroup8AMultiGroupLinker">
        <node class="de.lmu.ifi.pms.ldsms.rds.RDSGroup8AMultiGroupLinker"/>
      </source>
      <drain class="de.lmu.ifi.pms.ldsms.generics.ConsoleDrain"/>
    </network>
  </services>
</server>
```
This example reads the data from examples/rds/RDSGroup.bin, creates RDS-Groups and links multi group messages together. The instance of ConsoleDrain is used to print the result to the console.

### D.2.14 TMCMessage2Xml

**Package:** de.lmu.ifi.pms.ldsms.xml  
**Type:** Node  
**Input data type:** de.lmu.ifi.pms.ldsms.tmc.TMCMessage  
**Input meta type:** de.lmu.ifi.pms.ldsms.tmc.TMCMessageMetaInfo  
**Output data type:** java.lang.String  
**Output meta type:** de.lmu.ifi.pms.ldsms.network.SalutationMetaInfo  
**Dependencies:** no additional external library needed  
**Abstract:** Creates XML elements to represent the data and metadata in a platform independent data structure.

#### Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>prettyPrint</td>
<td>boolean</td>
<td>false</td>
<td>Yes</td>
<td>If pretty print is enabled, tabs and newlines are used to make the logical structure visible to humans. Each XML element starts at a new line and is indented relative to its parent element.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
<tr>
<td>xpath</td>
<td>String</td>
<td></td>
<td>No</td>
<td>The XPath expression for querying the XML stream.</td>
</tr>
</tbody>
</table>

#### Example:

```
Listing D.14: Example

<?xml version="1.0" encoding="ISO-8859-1" ?>
```
<server>
<logging level="INFO" />
<services>
<network>
<source_class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayFileSource" file="examples/easyway/input.bin" interval="100" frequency="95.1" version="1.0">
<node_class="de.lmu.ifi.pms.ldsms.rds.easyway.EasyWayRDSChunk2RDSBlock"/>
<node_class="de.lmu.ifi.pms.ldsms.rds.RDSBlock2RDSGroup">
<node_class="generics.Filter">
<and>
<condition_class="rds.RDSGroupCorrectnessCondition"/>
<condition_class="rds.RDSGroup8ARepetitionCondition"/>
</and>
</node_class="rds.RDSGroup2RDSGroup">
<node_class="rds.RDSGroup8AMultiGroupLinker">
<node_class="tmc.RDSGroup2TMCMessages" locationDB="tmc_databases/LocationList_de.csv" eventDB="tmc_databases/EventList_en_de.csv" locales="de.DE">
<node_class="xml.TMCMessages2Xml" prettyPrint="true">
<drain_class="generics.StringFileDrain" file="examples/xml/TMCMessages.xml"></drain_class> 
</node_class>
</node_class>
</node_class>
</node_class>
</node_class>
</source_class>
</network>
</services>
</server>

This example simulates an EasyWay light RDS reciever of version ‘1.0’, that is tuned to the frequency 95.1Mhz. The data is read from ‘examples/easyway/input.bin’ every 100ms. It is processed step-by-step to a TMCMessages. The TMCMessages are transformed to a XML format, using TMCMessages2XML. The instance of StringFileDrain is used to save the output to ‘examples/xml/TMCMessages.xml’.
D.2.15 TMCMessageManagement

Package: de.lmu.ifi.pms.ldsms.xml
Type: Node
Input data type: de.lmu.ifi.pms.ldsms.tmc.TMCMessage
Input meta type: de.lmu.ifi.pms.ldsms.tmc.TMCMessageMetaInfo
Output data type: java.lang.String
Output meta type: de.lmu.ifi.pms.ldsms.network.SalutationMetaInfo
Dependencies: no additional external library needed
Abstract: Manages the TMCMessages and keeps an actual state of the TMC channel, i.e. caches all currently valid TMCMessages. This state is sent to subsequent nodes in a certain delay, which is specified with the delay attribute.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>Integer</td>
<td>15000</td>
<td>Yes</td>
<td>Specifies the time interval (in milliseconds) the current state is sent repeatedly to subsequent nodes.</td>
</tr>
<tr>
<td>name</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td></td>
<td>Yes</td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

Listing D.15: Example

```xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO">
    <logger name="de.lmu.ifi.pms.ldsms.xml.TMCMessageManagement" level="DEBUG" />
  </logging>
  <services>
    <network/>
  </services>
</server>
```
**D.2.16 Xml2TMCMessage**

Package: de.lmu.ifi.pms.ldsms.xml  
Type: Node  
Input data type: java.lang.String  
Input meta type: java.lang.Object  
Output data type: de.lmu.ifi.pms.ldsms.tmc.TMCMessage  
Output meta type: de.lmu.ifi.pms.ldsms.tmc.TMCMessageMetaInfo  

Dependencies:

- name: Apache Commons Digester  
  version: Digester 1.6 (or higher)  
  url: [http://commons.apache.org/digester/](http://commons.apache.org/digester/)

- name: Apache Crimson  
  version: Crimson 1.1 (or higher)  

Abstract: XML2TMCMessage provides methods for transforming the XML representation of a RDS-TMC message into a TMCMessage object.

Attributes:

<table>
<thead>
<tr>
<th>Attribute-name</th>
<th>Type</th>
<th>Default</th>
<th>Optional</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>Specifies a unique identifier within the current L-DSMS instance for this component.</td>
</tr>
<tr>
<td>sourcerefs</td>
<td>String</td>
<td>Yes</td>
<td></td>
<td>A comma separated list of identifiers from additional sources, that aren’t direct parents.</td>
</tr>
</tbody>
</table>

Example:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<server>
  <logging level="INFO" />
  <services>
    <network>
      <source class="de.lmu.ifi.pms.ldsms.generics.StringFileSource" file="examples/xml/TMCMessage.xml">
```
This example reads the TMC messages from a XML document and uses Xml2TMCMessage to transform it back to TMCMessage. The instance of ConsoleDrain is used to print the result to the console.

D.3 FilterConditions

FilterConditions are used by Filter nodes to verify, that the incoming data fulfills the required conditions. The Filter nodes ask the FilterConditions, if they would discard or accept a given input. If the input is forwarded or not, is based on these information.

D.3.1 RDSGroup8ARepititionCondition

Package: de.lmu.ifi.pms.ldsms.rds
Output data type: de.lmu.ifi.pms.ldsms.rds.RDSGroup
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSGroupMetInfo
Dependencies: no additional external library needed
Abstract: RDSGroup8ARepititionCondition caches the last two received rds groups and returns true for an rds group if it is contained in its internal cache and if the group wasn’t the last received. This condition implements the validation through repetition that the RDS specification requires (a group is only sent if it was immediately repeated at least once).

D.3.2 RDSGroupCorrectnessCondition

Package: de.lmu.ifi.pms.ldsms.rds
Output data type: de.lmu.ifi.pms.ldsms.rds.RDSGroup
Output meta type: de.lmu.ifi.pms.ldsms.rds.RDSGroupMetInfo
Dependencies: no additional external library needed
Abstract: RDSGroup8ACorrectnessCondition filters out RDS groups containing incorrect blocks. A RDS group contains incorrect blocks, if the RDSGroupMetaInfo indicates, that there are uncorrected errors.