A1-D6

Dynamic Data for Geospatial Reasoning – A Local Data Stream Management System (L-DSMS) and a Case Study with RDS-TMC

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/A1-D6/D/PU/a1
Responsible editors:	B. Lorenz
Reviewers:	H. J. Ohlbach, R. Backofen
Contributing participants:	Munich
Contributing workpackages:	A1
Contractual date of deliverable:	28 February 2006
Actual submission date:	8 March 2006

Abstract

In this deliverable we present two developments. The first one is a *Local Data Stream Management System* (L-DSMS) It is a general system for configuring and (locally) executing networks of processing nodes for data streams. Each such node receives data from one or several data sources, processes them in a certain way, and delivers the processed data to one or more data drains. A data drain can be the data source for the next processing node in the network, or it can be the end application in the whole processing chain. One of the components of L-DSMS is the SPEX XML filtering system. It processes XPath queries on a stream of XML data and can be used to extract interesting information from XML streams.

The second development is an application of L-DSMS for processing dynamic traffic information. The traffic information comes from RDS-TMC capable FM receivers. It is processed in several steps and then delivered to several application systems, including databases, statistical analysis systems, wayfinding algorithms, and in particular different graphical visualisation systems.

Keyword List

semantic web, dynamic data, data streams, xml streams, filtering, visualisation, traffic and travel information, radio data system, RDS, traffic message channel, TMC

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Dynamic Data for Geospatial Reasoning – A Local Data Stream Management System (L-DSMS) and a Case Study with RDS-TMC

Hans Jürgen Ohlbach, Bernhard Lorenz

Department of Computer Science, University of Munich
E-mail: {`ohlbach,lorenz`}`@pms.ifi.lmu.de`

8 March 2006

Abstract
In this deliverable we present two developments. The first one is a *Local Data Stream Management System* (L-DSMS) It is a general system for configuring and (locally) executing networks of processing nodes for data streams. Each such node receives data from one or several data sources, processes them in a certain way, and delivers the processed data to one or more data drains. A data drain can be the data source for the next processing node in the network, or it can be the end application in the whole processing chain. One of the components of L-DSMS is the SPEX XML filtering system. It processes XPath queries on a stream of XML data and can be used to extract interesting information from XML streams.

The second development is an application of L-DSMS for processing dynamic traffic information. The traffic information comes from RDS-TMC capable FM receivers. It is processed in several steps and then delivered to several application systems, including databases, statistical analysis systems, wayfinding algorithms, and in particular different graphical visualisation systems.

Keyword List
semantic web, dynamic data, data streams, xml streams, filtering, visualisation, traffic and travel information, radio data system, RDS, traffic message channel, TMC

Contents

1 Introduction

The REWERSE workpackage A1 is, among others, concerned with modelling, processing and reasoning with geospatial data. Geospatial data can be (more or less) static or dynamic. More or less static geospatial data are geospatial ontologies, road maps, addresses etc. Dynamic geospatial data are, in particular, traffic information, e.g. news about traffic jams, temporary construction sites, blocked roads, delays or changes in bus or train schedules etc.

Many practical applications of geospatial information processing systems are much more useful if dynamic information is taken into account. Navigation systems are a prominent example where dynamic information is most useful. 'Static' queries to (XML-) databases can, however, also benefit from dynamic information. A straighforward answer to a query like "where is the *closest* pharmacy" can be useless when the road to the closest pharmacy is blocked. As we have shown in [19], there are in fact many geospatial relations whose evaluation amounts to a path planning problems. Therefore one of the goals in WG A1 is to consider also dynamic information in geospatial information processing, and in particular in path planning problems.

Dynamic information usually comes as streams of data, and these data streams must be processed, usually in several steps, until they can be fed into the final application system.

In this deliverable we present two developments. The first one is a *Local Data Stream Management System* (L-DSMS). Data stream management systems are, for example, used in grids to control the flux of large amounts of data from the data sources, telescopes, for example, to world wide distributed computer centers. This is *not* an application for L-DSMS. L-DSMS is *local* in the sense that it facilitates the specification and construction of a single Java program which consists of a network of nodes for processing streams of data. Each such node receives data from one or several data sources, processes them in a certain way, and delivers the processed data to one or more data drains. A data drain can be the data source for the next processing node in the network, or it can be the end application in the whole processing chain. One of the components of L-DSMS is the SPEX XML–filtering system [21, 22]. It processes XPath [6] queries on a stream of XML data and can be used to extract interesting information from XML streams.

The L-DSMS network is configured by XML–files. They contain the list of nodes, and for each node its sources and drains. Each node corresponds to a Java class whose methods do the actual processing. The L-DSMS reads the configuration files, loads the corresponding Java classes and arranges them into the required network.

The second development is an application of the L-DSMS for processing dynamic traffic information. The traffic information comes from RDS-TMC receivers, is processed in several steps and then delivered to several application systems. The details are described in section 5.

2 Dynamic Data

The traditional way in which data is managed and made available for processing is via (relational) *Database Management Systems* (DBMS), which expect data to be put into the form of *persistent data sets*. Whereas for many applications this constitutes a suitable form of data storage, there exists a growing number of applications which require data to be treated and processed as a *continuous stream* [1]. Currently a number of different *Data Stream Management Systems* (DSMS) are developed to facilitate access to data streams.

If there are relatively infrequent and only small updates, querying stored data sets is a reasonable method for data access as opposed to cases, where there is a continuous stream of data which by existence constitutes updates inherently[1], or whenever data cannot be stored for processing.

There are many examples of data streams: news tickers, sensor networks, traffic monitoring, and usage logs, to name but a few.

2.1 Comparison between DSMS and DBMS

Traditional DBMSs operate on a basis of relatively static data. These static data can then be queried using a suitable query language (most commonly SQL), whereas at different times mostly different queries are processed. DSMSs in contrast store a number of rather static different (continuous) queries which are then processed against an incoming stream of data which is continuously changing. A comparison between different characteristics of DBMSs and DSMSs can be found in table 1. A more detailed review of DSMSs with an emphasis on application requirements, data models, continuous query languages, and query evaluation can be found in [8].

DBMS	**DSMS**
persistent data (relations)	transient data
random access	sequential access
single queries	continuous queries
(theoretically) infinite secondary memory	finite main memory
current state of data relevant	ordered access
relatively low and infrequent updates	continuous updates
processing speed not critical	processing speed critical

Table 1: Characteristics of DBMS and DSMS

2.2 Definition

Chiefly two types of data streams are distinguished in the literature: *tuple streams* and *message streams*. Tuple streams [4, 9] are streams of sequences of flat data items of the same length, like those used in relational database tables. In contrast to tupels, the single items contained in message streams are neither flat nor are they of the same size or structure, i.e. they can, for example, consist of tree like structures, whose size and nesting depth can vary [22].

2.2.1 Tuple Streams

Continuous streams from sensor networks, e.g. temperature or humidity sensors, are typically formatted as tuples, i.e. a certain number of numerical values transmitted as a sequence. Each sensor would generate data in the same form, the collection of which builds up the stream. Although these values could be stored in a relational database, sometimes it is not desirable or necessary to store them persistently – either because there are too many data, or because the data are only valuable for a short time and must be processed immediately. In some scenarios system engineers go to great lengths in order to reduce the amount of data to be transmitted (e.g. to reduce costs) while still retaining the net information produced. In Traffic and Travel

[1]Updates are tuples or messages simply appended to the data stream.

Information systems (TTI, see section 2.3.1), for example, a huge number of sensors on highway bridges need to transmit traffic information to a central component. Instead of transmitting data in fixed intervals, different mechanisms are employed to send only data when something "interesting" is happening (e.g. if there is a change in traffic flow, etc.).

2.2.2 Message Streams

Items in a news ticker cannot be treated in the same way. Although it is technically possible to produce a tuple of strings, there are far more advanced methods available. One possibility is the use of XML and an appropriate schema. The schema defines the form of a news item so that a receiving device can understand the structure. In this case, a suitable schema could be that a news item has to consist of a title, a location, a source, one or more paragraphs of text, and – optionally – one or more images. This way, the news items can be adapted in size and struture to their content while still conforming to their schema. Receiving devices can decode and display the items depending on the target media.

A more comprehensive discussion of data streams is not in the scope of this paper, it can be found, for example, in [3] or [21].

2.3 Examples

The following examples may not be the most typical to illustrate the two prevailing types of data streams, but they serve well to illustrate the significant impact dynamic data have on geospatial reasoning processes.

2.3.1 Traffic Information

Traffic and traveller information (TTI) [26] as an important part of Intelligent Transport Systems (ITS) [17, 7] serves to provide to the traveller real-time information about traffic and travel conditions, for example, schedules, extraordinary events and incidents, delays, possibly weather conditions and more. Fig. 1 shows a schematic overview (from [14], p.159).

The aim of TTI is to support the efficient use of network infrastructure, i.e. to improve network capacity and reduce travel time, delays, and fuel consumption, by incorporating TTI into the routing process of travellers. This way a route can be better optimised to different criteria, such as minimal overall travel time, minimal cost, or other, offering a much broader range of (dynamic) parameters[2] in order to facilitate more complex cost-functions and, subsequently, more accurate route planning.

The influence of TTI on travel behaviour (e.g. the optimisation of travel time) can be divided into three main categories [26]:

- **Spatial**: Optimise travel time by using different routes which allow for the quickest travel because of speed limits, number of traffic lights, current and predicted congestion, etc. (note the important distinction between *quickest* and *shortest* route).
- **Temporal**: Temporal adaptation works similar to spatial adaptation. Instead of shifting spatially, travel is optimised by temporal shifts, such as avoiding peak times by conducting the travel before or after the so-called "rush hours". Temporal and spatial adaptation are the most common methods, a combination of both is also possible.

[2]Static attributes of, for example, a street segment include *length* or *number_of_lanes*, while *current_congestion* or *road_surface_condition* constitue dynamic attributes – the latter being derived from TTI.

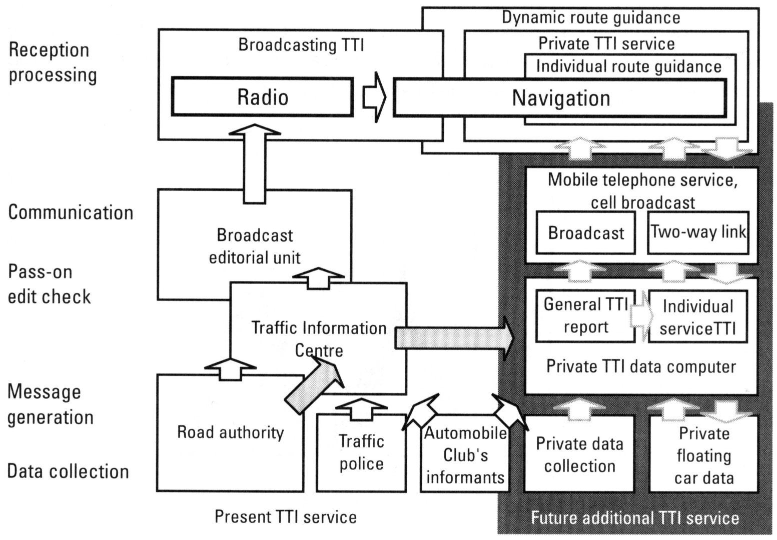


Figure 1: Projected scenario for TTI service provision

- **Modal**: Modal adaptation means using one or more alternative modes of transportation (or combinations thereof) for travel. A typical example is the combination of a private vehicle and public transport by train, for example, travelling with a car to a city-near park-and-ride station, then taking a train or subway to the final destination near the city center.

Generally, one has to distinguish between *pre-trip* information and *en-route* information [26]. The former is less dynamic in nature and is important for the planning process of a certain route (e.g. distances, flight schedules, congestion statistics, etc.). The latter is highly dynamic up-to-date information which concerns an already planned route insofar as it provides information (e.g. cancelled flights, recently developed traffic congestions, etc.) that would have had impact on the a priori planning process. It usually leads to an alteration of the route and/or renewed planning.

Chrobok et al. [5] mention another categorisation which consists of three types of traffic data: *historical*, *current*, and *predicted*. Statistical analysis is based on historical data in order to provide information about the state of the traffic network at previous periods in time. Current data is up-to-date information about the current state of the traffic network, possibly fed into and enriched by on-line simulations. Predictive data is generated by different means (simulation, statistical analysis, etc.) and aims at influencing travel behaviour.

TTI can be broadcast by a number of different means and services. One widely used method is the use of the Traffic Message Channel (TMC) via the Radio Data System (RDS) which serves as part of the foundation of the protoype described in this article. A more in-depth discussion of RDS-TMC can be found in section 4.2. In addition to the more classic ways to broadcast TTI (e.g. spoken messages or RDS-TMC) a number of alternative media have become available in the recent past, such as DAB, DVB, Internet, GSM, etc., for an overview see for example [15].

2.3.2 Weather Information

Weather forecasts generate a very similar type of data stream as TTI. Before weather forecasts are created, typically by simulation, a huge amount of raw weather data from highly distributed sensors are to be processed, refined and fed into a simulation. These raw data usually consist of a pair of values, one holding the value of interest (e.g. 980 hPa), the other holding a time stamp of the measurement. If values are transmitted in regular intervals (e.g. 1 minute), one single time stamp can be transmit together with a row of values, intermediate time stamps being extrapolated.

As opposed to the different formats and structures messages can have, tuples of values are much less complex in nature and can be treated nearly on a value by value basis. They normally come in regular (usually very short) intervals, and they always conform to one given (uncomplex) format.

2.3.3 Schedule Updates

Wherever there are scheduled services in operation, there is the possibility of deviations from the underlying schedules. Well known examples in public transport are busses, trains, subways, etc. Deviations from the regular schedule are usually expressed as delays, which constitute temporary updates to an otherwise valid schedule. In this lies the ambivalence of this example. Because they are static, the primary data are usually kept in traditional data base systems, e.g. as tables. In these tables, one could find, for example, the information that the train *ICE882* from Munich to Hamburg leaves Munich on weekdays at 07:44 and passes by Hannover at 12:36. Whenever there is a delay which impacts the regular schedule siginifcantly (i.e. a cancellation in the worst-case) an update must be available to inform customers about this fact. One possibility for this would be a continuous data stream containing schedule updates. Systems providing information about the schedule could subscribe to such a stream and keep a temporary record of updates until their expiration date. Users accessing the service would then be informed not only about the scheduled times, but – more or less – about the expected actual times.

3 The Local Data Stream Management System (L-DSMS)

The architecture of L-DSMS consists of two general framework components and several integrated components comprising the server: the `NetworkBroker` in the package `sttiprak.network` and a number of generic producers/consumers in the package `sttiprak.generics`.

The `NetworkBroker` initialises and starts its components in the order of the stream flow, i.e. from sources to drains. After operation, these are stopped in the opposite order, so that a drain is stopped before the source it is attached to is stopped. By components (i.e. drains and sources) the internal network of producers and consumers is meant, which *consume* the data stream at one point, filter or transform it, and *produce* a modified output stream.

The package `sttiprak.generics` provides generic producers/consumers, such as those facilitating connectivity via TCP/IP (regarding both input and output), reading from and writing to files, console output, etc. All further components implement concrete producers/consumers which are necessary for the TMC XML transformation prototype and not part of the general infrastructure.

As defined by the system architecture and as depicted in Fig. 2, a server can consist of a number of different combinations of components, i.e. *sources*, *nodes* and *drains*. Databases (DBMS) and DBMS sources respectively are used to enrich the data stream with data from traditional databases by either a direct connection to the DBMS or by providing an artificial data stream which is originating at a DBMS source connected to a DBMS.

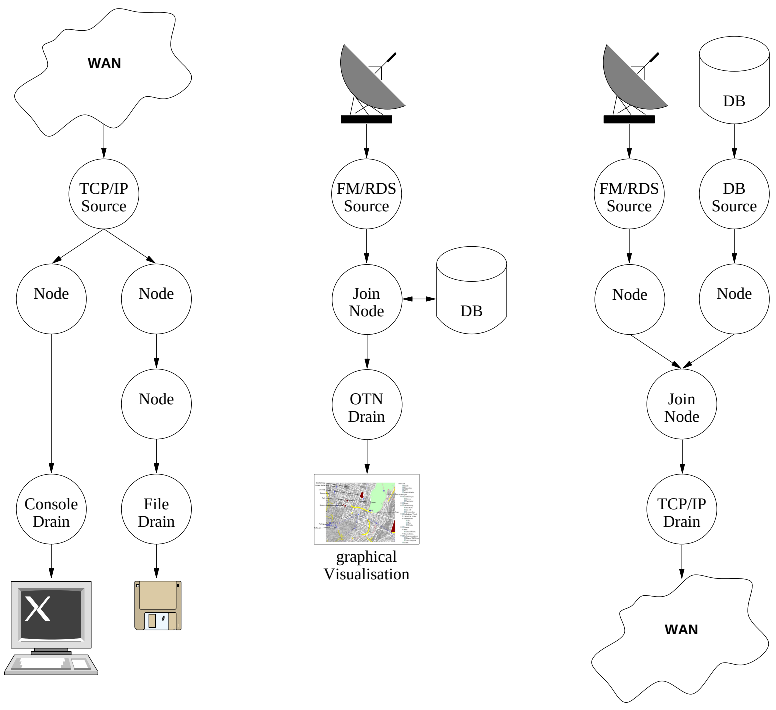


Figure 2: Examples of server configurations

- **Sources** can produce a data stream in a number of different ways, for example, by
  - reading from a file
  - connecting to a suitable service via TCP/IP
  - connecting to a DBMS
  - directly connecting to some data producing device, e.g. receiver hardware.

  Since different applications require different and individual implementations, these are only some examples.

- **Nodes** serve the purpose of filtering and transforming the data stream and providing input for further nodes or drains. It is possible to concatenate a number of nodes in a row for more complex filtering mechanisms.

- **Drains**, much like sources, consume data, for example, by:

  - writing to a file
  - offering the data stream as a TCP/IP service
  - inserting records into tables stored in a DBMS
  - providing output on a system console

The topology of a server, i.e. the structure of sources, nodes and drains, as well as their order and specific configuration parameters is kept in special configuration files. See section 3.2 for more detail on this issue. All the classes used to build a server are located in the package `sttiprak.network`.

Corresponding to the above mentioned components, the framework provides a number of generic classes, which implement the necessary functionality. These are classes which are indepentend from the individual use of the framework and which can be subclassed to be adapted for specific applications.

3.1 Generic Components

The purpose of generic components is to provide basic I/O and filtering functionality within the framework. The current implementation includes the following components:

- **Socket Source / Drain**

  Probably the most important I/O mechanisms are network sockets, since they facilitate the flexible distribution of components over a number of machines which are connected by a network. On the input side, data sources can be accessed via a fully qualified domain name and a port number, while on the output side the transformed data stream is made available to other consumers to be accessed at a configurable local port.

  Apart from a connectionless mode (`SocketDrain` and `SocketSource`), there exist so called "polite" counterparts (`PoliteSocketDrain` and `PoliteSocketSource`) which require a formal opening and closing of a connection, as opposed to just terminating the consuming process. This enables the server to keep track of who is listening to its service.

- **File Source / Drain**

  File sources and drains exist primarily for testing and simulation purposes. Instead of requiring a receiver set ready to operate or any other data source, previously logged data can be read from a file and be made available as a data stream. This also allows for reproducing otherwise unique data streams, which are lost after they have been passed through the system.

  A special variant of the `FileDrain` is the `ConsoleDrain` which directs the textual output directly to a terminal.

- **Filter Nodes**

  Querying of data streams can be achieved through the use of filters. A number of nodes, each containing a certain single filter configuration, can be combined in a network of nodes to realise the desired filter options. A sample filter configuration can be found in the appendix in section A.1.2.

For technical reasons, sources and drains operating on files or sockets are available in two variants, those processing *ByteArrays* and those processing *Strings*.

3.2 Configuration

Configurations are also part of the generic framework. In order to demonstrate the internal structures of configurations and their practical use, the following illustration is done in anticipation of the concrete use of the prototype with RDS-TMC streams. See section 5 for details on this matter.

A server configuration is kept in an XML configuration file which describes the topology of the network of sources, nodes and drains, and their respective configurations.

A node is added to the network by including a node class and providing the necessary configuration data, for example:

```
<node class="xml.TMCMessage2Xml" prettyPrint="true" />
```

Child nodes can be added by using the nesting syntax of XML. The following example means that node 1 directs its outout to node 2.

```
<node class="A" name="1">
    <node class="B" name="2"/>
</node>
```

More complex networks (other than trees) can be constructed by incorporating references into nodes or drains. The attribute `sourcerefs` contains a reference to the providing node/-source:

```
<node class="A" name="1">
  ...
</node>
  ...
<node class="B" name="2" sourcerefs="1"/>
```

This example defines the same relation like the one before, i.e. that node 2 receives its data from node 1.

The description of a topology must start with sources and end with drains. For example:

```
<source class="S">
    <node class="N">
        <drain class="D"/>
    </node>
</source>
```

The specification of a topology for a complete server could be:

```
<server>
    <logging level="INFO">
    </logging>
    <services>
        <network>
            <source class="S">
                <node class="N">
                    <drain class="D"/>
                </node>
            </source>
        </network>
    </services>
</server>
```

Naturally, the attributes for configuration are different depending on the respective node type. An overview can be found in the configuratin schema in the appendix in section A.2.2, some example configurations in sections A.1.2 to A.1.4.

Although L-DSMS generates Java programs which work locally on one machine, it is no problem to use it for processing streamed data in a network of computers. A socket data drain of machine A can be linked to a socket data source at machine B, where another instance of L-DSMS is installed. This way one can distribute different components over a network of different machines.

3.3 The SPEX Filter Mechanisms

SPEX is an XML filter system for streamed data developed by the Munich group. It uses pushdown transducer networks for evaluating regular XPath expressions with qualifiers over streamed XML-data. The SPEX approach enjoys the following attractive features. The translation of a regular path expression with qualifiers into a transducer network takes a time linear in the size of the input expression. The evaluation can be performed in one pass over the data stream. The memory space needed for an evaluation is at most quadratic in the nesting depth of the XML–document. The computational power needed by each transducer of a SPEX network is within the 1-DPDT class.

In this article, we use SPEX as a means to filter the produced XML stream by inserting one or more SPEX nodes into the system architecture (see section 5.1) and by supplying suitable XPath [6] queries.

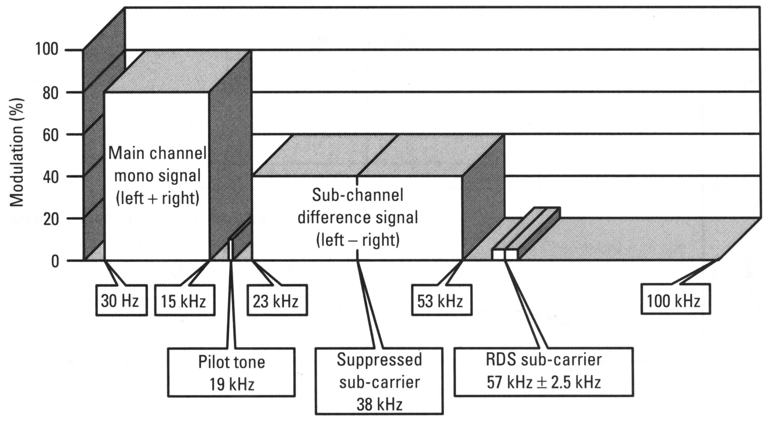
SPEX was originally designed to handle infinite and deeply nested XML data streams. The application of SPEX in the prototype presented in this article serves mainly to illustrate the possibilities of integration of existing stream processing software. The low complexity and low bandwidth characteristics of TTI does not necessitate such a powerful stream processor. Further information about SPEX can be found in [21, 22].

4 RDS / TMC Traffic Information

In this article we take one of the RDS services, the Traffic Message Channel (TMC) as an example for a stream of traffic information to be incorporated into a dynamic routing application.

4.1 RDS

The Radio Data System (RDS) [14] is a narrow bandwidth data transmission channel for VH-F/FM broadcasting. RDS supports data transmission alongside (see Fig. 3) sound broadcasts and facilitates services which are based on sending a small amount of digital data to a great number of users. It was developed in the 1970s and early 1980s and is now implemented all over Western Europe, several Central and East European countries, in parts of Asia Pacific, South Africa and (using the slightly different [24] Radio Broadcast Data System (RBDS) [23] standard) the United States. Rather recent additions to RDS are RDS-TMC (see next section) and Open Data Applications (ODA) (not discussed here, see [14], chapter 9, instead).



Modulation (%)
100
80
60
40
20
0
Main channel mono signal (left + right)
Sub-channel difference signal (left – right)
30 Hz
15 kHz
23 kHz
53 kHz
100 kHz
Pilot tone 19 kHz
Suppressed sub-carrier 38 kHz
RDS sub-carrier 57 kHz ± 2.5 kHz

Figure 3: Spectrum of a pilot-tone stereo multiplex signal with RDS

Basic features of RDS include, among others, the following information features, tuning aids, and programme-related features. Because this list only serves to illustrate the basic ideas and functionalities of RDS, it is not exhaustive.

- **Information Features**
  - Clock Time (CT): The current time and date can be transmitted in type $4A$ groups by the radio stations to keep receivers' internal clocks within an accuracy of ± 0.1 seconds of a certain reference time (e.g. DCF (77.5) kHz in Germany or MSF (60kHz) in England).
  - Enhanced Other Networks (EON): Especially valuable for larger broadcast networks, EON information, transmitted in type 14 groups, allows the update of a number of features for programme services other than the currently tuned service. This includes for example AF, PIN, PS, PTY, TA (described below).
- **Tuning Aids** (all of type $0A$ group)
  - Programme Identification (PI): This identifier is not intended for display, but for identifying identical broadcasts on different frequencies. If reception quality is decreasing, and if they are equipped with a secondary fm tuner, RDS receivers can search for broadcasts on other frequencies with indentical PI code which offer better reception quality (of the very same programme).
  - Programme Service Name (PS): This contains the static 8 character identifier to be displayed to the user.
  - Traffic Programme and Traffic Announcement (TP/TA): These flags indicate the availability of spoken traffic announcements on the currently tuned station (when used with EON also other stations). This enables the receiver to increase the volume

and to stop CD or cassette playback whenever spoken announcements are transmitted.

  – Alternative Frequencies (AF): This feature provides alternative frequencies for the currently tuned station in order to optimize reception quality.

- **Programme-Related Features**
  – Programme Type (PTY): A list of 29 standardised choices describing the broadcast programme enable the user to set the receiver to a certain programme type (e.g. news) and therefore not choose a specific radio station, but a certain type of broadcast.
  – Radio Text (RT): Text messages of up to 64 characters can be coded and broadcast by the radio text feature. Although many receivers, especially mobile ones, feature displays with less than 64 characters.

4.2 TMC

TMC [11, 12, 13] was mainly developed in the years from 1984 to 1997 by a number of european companies and institutions, under the leadership of the European Broadcast Union (EBU), in order to broadcast TTI messages on VHF/FM broadcast transmissions using RDS [14]. TMC is one of several RDS features and services, although, compared to some rather simple features such as tuning aids (PI, PS, TP) or programme-related features (PTY, RT, PIN), it is one of the most complex standards within RDS.

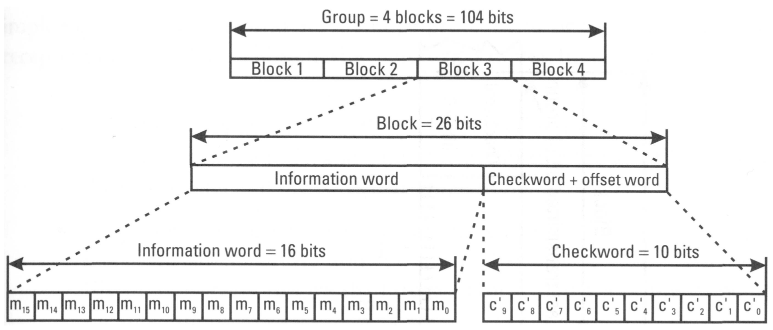

Group = 4 blocks = 104 bits
Block 1
Block 2
Block 3
Block 4
Block = 26 bits
Information word
Checkword + offset word
Information word = 16 bits
Checkword = 10 bits
m_{15} m_{14} m_{13} m_{12} m_{11} m_{10} m_9 m_8 m_7 m_6 m_5 m_4 m_3 m_2 m_1 m_0
c'_9 c'_8 c'_7 c'_6 c'_5 c'_4 c'_3 c'_2 c'_1 c'_0

Figure 4: Structure of RDS baseband coding

The main advantages of digitally broadcast RDS-TMC messages over spoken traffic announcements are:

- **Ansynchronous reception**: Users need not be listening at the correct time to the correct radio station in order to receive information. This especially important for individual traffic, since onboard systems must not interfere with the drivers' ability to concentrate on the traffic.
- **Caching mechanisms**: Messages are stored in a client device and can be queried any time. Life cycle management ensures that outdated messages are erased from the memory.

- **Filtering mechanisms**: Several mechanisms exist for filtering out unwanted content, for example, by event type, current location, or projected path. Furthermore, short repetition cycles combined with duplicate elimination facilitate timely broadcast of information[3].

- **Language independence**: The binary coded messages rely on the client devices' abilities to generate human understandable messages. This may require increased device intelligence but also facilitates the use of different languages.

- **Message density**: With 1187.5 bps the RDS bandwidth is comparably narrow from a current viewpoint. Although only some 300 messages can be transmitted per hour [14], this displays significant advantages over spoken messages. If the information of each message could be conveyed by an average of 15 seconds of spoken text, the same number of messages would still produce the unrealistic amount of about 75 minutes of announcement time.

- **Navigation assistance**: By incorporating digital traffic announcements into car navigation systems, the task or navigation and route planning could be substantially improved.

Especially the last point is of great importance for intelligent transport systems which need to take into account current traffic siituations as well as statistical data and data from simulations in order to refine and further optimise the movement of goods and passengers in more and more complex scenarios.

In October 2004 the two major providers of digital map data, Tele Atlas [25] and NAVTEQ [18] announced future collaboration [16] on the standardisation of traffic codes for digital maps for the United States, which will be based on the European RDS-TMC Alert-C specifications.

5 TMC to XML Transformation Prototype

The availability of TTI for any device or application is depending on the acessibility of some source for this type of information. Car navigation systems can, for example, be connected to an RDS compatible radio receiver. As sales have risen and RDS radios have become quite common, marginal costs for the necessary hardware components have been declining which further opens the market to other devices and fields. GPS receivers for mobile solutions sometimes also contain an RDS compatible FM receiver, although this is not as common.

However, in cases where there is no FM receiver at hand, an alternative source for TTI must be found. Routing applications running on standard PCs (which usually don't have built in FM receivers), for example, belong to this category. One possible solution is to substitute radio transmission and receiver hardware with an internet connection and a web service, which has been the objective of a project described in this section. The main goal of the project was to provide the following functionality:

- **FM Receiver**

  This is the first of only two hardware components of the system. In our testbed there are currently two RDS capable FM receivers attached via serial ports to a server. They can be tuned to different radio stations and can be set to provide a raw binary RDS data stream

[3]Rather optimistic refresh cycles of 15 minutes would lead to an average latency of 7.5 minutes, whereas the typical RDS-TMC cycles of 120 seconds result in only 60 seconds of latency.

at the serial port. By design, the FM receiver shall neither block the frequencies from 15 to 23kHz nor above 53kHz, since this is where RDS data is transmitted (see Fig. 3).

- **RDS Decoder**

  This second hardware component produces the raw RDS bit stream by isolating and decoding the signals around 57kHz at a rate of 1187.5 bits per second. This stream is directly delivered to the receiver's serial port, which is connected to the server machine.

- **TMC Decoder**

  The first task entirely realised in software is the decoding of RDS groups from the raw data stream. RDS groups consist of 4 data blocks which contain 26 bits each. Of these 104 bits ($4 * 26$), 40 bits (10 in each block) are used for error correction, which leaves a net payload of 16 bits per block or 64 bits per group (see Fig. 4). TMC messages contain the group id *8A*.

- **XML Stream Generator**

  At this stage, the raw TMC data are transformed to XML corresponding to a customized schema (see section 5). Furthermore, the data are enriched with the contents of the *Event Code List* (ECL) and *Location Code List* (LCL). This enables devices which cannot access these code lists to nevertheless display the textual contents of TMC messages, instead of rather cryptic raw binary data. Being part of the core of the system, this process is described in detail in section 5.1.

- **SPEX Filter Mechanisms**

  In contrast to classic querying of relational data, which produces a result set designed to meet the users demands, the processing of data streams needs other mechanisms to query or filter the incoming data. We use a system called SPEX [21, 22], which has been developed by a former member of the Munich team, Dan Olteanu, as a powerful filtering mechanism for our system. This way, the stream optionally passes one or more nodes which filter according to certain criteria to produce a suitable output stream. More detail about this can be found in section 5.1, SPEX is described in detail in [21].

- **Configuration Component**

  In order to facilitate easy (re-)configuration of the different components, the networks of nodes which the stream passes through is configured entirely via a single XML file. In this file the respective sources and drains, as well as (filter-) nodes and the necessary parameters are specified.

- **Visualisation Component**

  Apart from the textual output of TMC messages, which strongly resemble the usual spoken announcements on the radio, graphical output in form of symbols on a map display is also provided. Easily implemented on different digital map systems, we show the basic procedure of how to integrate these graphical messages with an SVG-based map system rendered in a conventional internet browser window. Strictly speaking, this component is not part of the TMC to XML transformation prototype, nevertheless similar output mechanisms like the one provided here would logically be the consumers of the provided data stream.

5.1 System Architecture

5.2 Custom Components

The generic components (see section 3.1) are used and extended to provide the basic functionality for the processing of RDS-TMC data. This section describes the individual adaptation of each component to its role in the processing of TMC data streams.

As the system offers a very flexible way to combine and integrate different nodes and the individual structure is very much depending on the respective application of the system, we can only provide some examples for illustration purposes.

Fig. 2 on page 6 shows two configurations involving RDS-TMC sources. The structure in the middle shows a sample configuration involving a graphical output of DB-enriched RDS-TMC data. Here, current traffic messages are displayed in combination with statistical congestion information coming from a tertiary source through an attached DBMS. The example on the right of fig. 2 shows a different configuration. The purpose of this structure is the combination of two sources into a joint stream , e.g. enriching RDS-TMC data with data from a DB-stored weather report system. The joint stream is then published for applications on the web, accesible through a TCP/IP drain.

5.2.1 Radio Data System Component (RDS)

As a receiver, we use a device called *Easyway Light* available from the *Institut für Rundfunktechnik (IRT)* [10]. It is designed for testing purposes and provides the following connectors: aerial antenna, audio equipment, DC power, and RS-232 serial port. The following classes can be found in `sttiprak.rds` and `sttiprak.rds.easyway` respectively.

The components, which read data from the device are named accordingly. Data is read directly from the device by `rds.easyway.EasyWaySource`. Further transformations are done in `rds.EasyWayRDSChunk2RDSBlock`, which generates RDS blocks, and `rds.RDSBlock2RDSGroup`, which produces RDS groups from the blocks (see Fig. 4). Intermittendly multi group messages are connected using `rds.RDSGroup8AMultiGroupLinker`.

Furthermore, there exist some filtering functions, which pertain directly to the raw RDS data and are therefore handled at this early stage (before the main filtering facilities can be applied). `rds.RDSGroupCorrectnessCondition` indentifies and drops incorrectly received groups. Certain *types* of groups can be filtered by `rds.RDSGroupFilterCondition`, and `rds.RDSGroup-8ARepetionConditon` only lets groups pass, which have been transmitted at least twice .

5.2.2 Traffic Message Channel Component (TMC)

The class `tmc.RDSGroup2TMCMessage` takes an 8A group and transforms it into an internal representation of a TMC message. In this step, the data is transformed into a more easily manageable format and is enriched by the corresponding data from the ECL and LCL tables. Whereas from the ECL only the textual equivalent of the event code and its parameters are taken, the location code is enriched by hierarchical information. This means that in addition to an event code, e.g. '12733' and its textual name 'A9 Pfaffenhofen' (part of highway number 9), also all its parent's codes and names in the hierarchy are included, i.e. '602, Pfaffenhofen a. d. Ilm' (name of the region), '294, Oberbayern' (Upper Bavaria), '264, Bayern' (Bavaria), and '1, Deutschland' (Germany, see also table 2). All the corresponding classes can be found in the package `sttiprak.tmc`.

Code	No.	Name	RefA	RefL	Off+	Off-	Exit
1	–	**Deutschland**	–	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
263	–	Baden Württemberg	1	–	–	–	–
264	–	**Bayern**	**1**	–	–	–	–
265	–	Saarland	1	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
293	–	Niederbayern	264	–	–	–	–
294	–	**Oberbayern**	**264**	–	–	–	–
295	–	Oberfranken	264	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
601	–	Peine	273	–	–	–	–
602	–	**Pfaffenhofen a. d. Ilm**	**294**	–	–	–	–
603	–	Pforzheim	284	–	–	–	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
12732	A9	Allershausen	418	7219	12733	12731	67
12733	**A9**	**Pfaffenhofen**	**602**	7219	12734	12732	66
12734	A9	In der Holledau	602	7219	12735	12733	–
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2: Extract from the German Location Code List

The reason for enriching the data (and thereby arguably introducing redundancy) is to enable devices without access to, or enough storage capacity for the ECL/LCL to process and display the entire relevant information. A device may, for example, need to filter out all messages not regarding a certain region. Including the enriched location information this is a simple task: drop any messages not containing the area code. Without this information, each location code must be hierarchically backtracked to its regional code in order to decide whether to drop the package or not.

5.2.3 Transformation into XML

The data in a TMC message are then transformed into a continuous XML stream by `xml.TMC-Message2Xml` contained in the package `sttiprak.xml`. A reverse transformation can be done by `xml.Xml2TMCMessage`, XML–filtering using XPath queries can be done in the node `SpexNode` (cf. sec. 5).

5.2.4 Visualisation Component (OTN)

As mentioned earlier, a proprietary visualisation component based on SVG is used to display traffic messges on a map [20]. The system requires an XML file containing the current state of the TMC channel, in valid OTN syntax [2]. The current state of the TMC channel means the set of *currently valid* messages which are tracked and kept up-to-date by `tmc.TMCMessageManagement` and sent to the child nodes in regular intervals. Currently, `OTNDrain` is the only node designed to receive such data and generate the necessary OTN file. These components can be found in the package `sttiprak.tmc`.

6 Impact on Reasoning Process

In the recent decades navigation systems have come a long way from industrial applications in aviation and navigation to the current substantial rise of demand in personal, mobile, and car navigation. In this transition to a widely used technology the main requirements also have to be adjusted in different aspects. One of the most important factors in future applications will be the adaptation to the individual context in which route planning takes place. Adaptation to context includes not only the time of day, means of transportation used, network structures and other overall constraints, but also, and more importantly, dynamic factors such as current traffic and weather situation (including complex effects of congestions, delays in scheduled operations, etc.) and the overall optimisation of many thousands of individual routing requests.

For some time now traffic information has played a rather minor role in routing applications due to the scarse availability and suboptimal quality of the provided data. Available data underly latency and volatility issues, which current *reactive* systems struggle to cope with. Reactive systems only track traffic data and, for example, record the occurence of increased traffic density, changes in average speed, or traffic jams in general. In the future, these systems will be perfected by means of statistics, simulation and prediction of traffic information to improve the quality of data and facilitate their timely delivery, within a *proactive* system. In some research projects even car-to-car communication (ad-hoc networking) is developed to generate and transmit e.g. data about road surface conditions to following vehicles within mere milliseconds, to insure the flow of information for the purpose of early warning and to trigger appropriate reactions, such as speed reduction or collision avoidance.

But even in cases when data of acceptable quality (i.e. accurate and up-to-date data) can be incorporated into the routing process, there is still a need for sophisticated techniques to really make use of these volatile informations, before the situation changes, which it constantly does.

6.1 Traffic Information

The main goals of incorporating traffic information into routing applications are to optimise travel by (1) avoiding blocked or congested road segments and (2) providing a form of load balancing between alternative route segments.

1. In the pre-trip stage, current traffic messages are incorporated into route processing so as to avoid problematic road segments altogether. The result is an optimised route, which does not include road segments for which significant incidents, such as traffic jams, heavy road works, accidents, etc., have already been reported.

   En-route, the navigation unit constantly processes incoming traffic messages and checks them for relevance regarding the planned route. This can be divided into two separate and independent tasks. One task deals exclusively with the road segments which are part of the current route. Should a new incident be reported for such a segment the system must decide whether to change the current route, depending on the severity and possible alternative subroutes. Major traffic jams on highways can sometimes be avoided by using secondary roads. The other task deals with already know incidents, regardless of whether they concern road segments which are part of the current route or not. Should a formerly reported incident be cancelled (e.g. traffic jam has dissolved, accident has been cleared, etc.) or reduced in severity (e.g. 10 minutes waiting time in front of a tunnel instead of 2 hours), it might be advisable to initiate re-planning in cases when the road segement in question would have been part of the initial planning, but had been disregarded because of the reported incident.

2. While there exists the possibility of quasi-equal distribution by statistical means, this measure can only be realised in an accurate way in centralised environments . It is based on the idea that a significant number of similar route requests are occuring within the same area, for example, daily commute into and out of major cities. In many cases of similar starting points and destinations, routing applications would propose similar, if not in part identical routes. In each individual case this is exactly the desired behaviour, but the overall performance of the network infrastructure would not be optimal, since there is no load balancing between alternative routes with nearly identical cost. In a decentralised system, individual routing applications could choose the resulting route from the top three or 5 best results at random in order to insure that not all similar cases lead to identical routes. In a centralised system, the resulting routes could be equally distributed over the network infrastructure in order to avoid subsequent congestion on individual road segments (i.e. already at the planning stage).

6.2 Schedule Updates

As outlined in section 2.3.3 schedule updates hold information critical to a routing application similar to TTI. When planning a route involving a sequence of connections between scheduled services, e.g. subway, regional train, airplane, etc., then the transitions between independently operated systems can be critical. Delays on one of the involved sections could easily render the whole route useless. If, for example, a delay of a train leads to missing a subsequent flight. In some scenarios this is somewhat less critical due to high frequency of individual services (e.g. subways operating every 5 minutes). In general, longer intervals between services (e.g. one flight per day or week) and longer overall travel time, possibly including more critical transistions, lead to situations where taking into account possible delays (allowing for idle periods and therefore longer travel time as a trade off) is becoming increasingly important.

The more complex interdependencies of scheduled operations present an interesting field for further research. These dynamics cannot simply be incorporated into classic cost functions and therefore a different, specially suited approach must be developed. This approach should facilitate multi modal processing of heterogeneous network structures while still being compatible to the graph based framework of classic routing applications.

7 Summary

The Flexible Data Stream Management System (L-DSMS), which includes the SPEX XML–filtering system, is a general purpose Java framework which can be used to configure and execute networks of processing nodes for manipulating data streams. The system will be made available as an open source package.

The particular application to processing TMC traffic information is also described in this deliverable. The TMC messages are recieved by special radio receivers, turned into XML streams, filtered by the SPEX system, and directed to several application systems. As one of the next steps we plan to combine this dynamic information with static geospatial information for advanced geospatial information processing in the Semantic Web context.

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

[1] S. Babu and J. Widom. Continuous Queries over Data Streams. *SIGMOD Record*, 30(3):109–120, 2001.

[2] H. J. O. Bernhard Lorenz and L. Yang. Ontology of transportation networks. REWERSE Deliverable A1-D4, Department of Computer Science, University of Munich, August 2005.

[3] F. Bry, T. Furche, and D. Olteanu. Datenströme. *Informatik Spektrum*, 27(2), 2004.

[4] A. Bulut and A. Singh. SWAT: Hierarchical stream summarization in large networks. In *Proc. of IEEE International Conference on Data Engineering*, pages 303–314, 2003.

[5] R. Chrobok, O. Kaumann, J. Wahle, and M. Schreckenberg. Three Categories Of Traffic Data: Historical, Current, And Predictive.

[6] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. Recommendation, W3C – World Wide Web Consortium, 1999.
http://www.w3.org/TR/xpath.

[7] H. Dia. Towards sustainable transportation - the intelligent transportation systems approach. In *Proceedings: Shaping the Sustainable Millenium.* Queensland University of Technology, 2000.

[8] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[9] T. S. Group. STREAM: The Stanford Stream Data Manager. In *IEEE Data Engineering Bulletin*, volume 26,1, `http://www-db.stanford.edu/stream`, 2003.

[10] Institut für Rundfunktechnik, IRT, `http://www.irt.de/`. *Forschungs- und Entwicklungsinstitut der öffentlich-rechtlichen Rundfunkanstalten in Deutschland*, 2005.

[11] ISO Standard 14819-1: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 1: Coding protocol for Radio Data System – Traffic Message Channel (RDS-TMC) using ALERT-C. `http://www.iso.org`, May 2003.

[12] ISO Standard 14819-2: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 2: Event and information codes for Radio Data System – Traffic Message Channel (RDS-TMC). `http://www.iso.org`, May 2003.

[13] ISO/TS Standard 14819-3: Traffic and Traveller Information (TTI) – TTI messages via traffic message coding – Part 3: Location referencing for ALERT-C. `http://www.iso.org`, June 2000.

[14] D. Kopitz and B. Marks. *RDS: The Radio Data System.* Artech House Publishers, 1998.

[15] D. Kopitz and B. Marks. Traffic and Travel Information broadcasting - protocols for the 21st century. *EBU Technical Review*, pages 4–12, 1999.

[16] Real-Time Travel Information – More Services ... with the Right Equipment. `http://www.locationintelligence.net/articles/559.html`, April 2004.

[17] T. Mast. Introduction to ITS. In *Human Factors in Intelligent Transportation Systems*, Mahwah, New Jersey, 1998. Lawrence Erlbaum Associates Inc.

[18] NAVTEQ, `http://www.navteq.com`. *Provider of digital map data*, 2005.

[19] H. J. Ohlbach and B. Lorenz. Geospatial reasoning: Basic concepts and theory. REWERSE Deliverable A1-D2, Department of Computer Science, University of Munich, March 2005.

[20] H. J. Ohlbach and B. Lorenz. Ontology driven visualisation of maps with svg. REWERSE Deliverable A1-D5, Department of Computer Science, University of Munich, September 2005.

[21] D. Olteanu. *Evaluation of XPath Queries against XML Streams.* Dissertation/Ph.D. thesis, Institute of Computer Science, University of Munich, 2005. PhD Thesis, Institute for Informatics, University of Munich, 2005.

[22] D. Olteanu, T. Furche, and F. Bry. An efficient single-pass query evaluator for XML data streams. In *SAC*, pages 627–631, 2004.

[23] Specification of the radio broadcast data system. `ftp://ftp.rds.org.uk/pub/acrobat/rbds1998.pdf`, April 1998.

[24] RBDS versus RDS – What are the differences and how can receivers cope with both systems? `ftp://ftp.rds.org.uk/pub/acrobat/rbds_vs_rds.pdf`, April 1998.

[25] Tele Atlas, `http://www.teleatlas.com`. *Provider of digital map data*, 2005.

[26] J. Wahle. *Information in Intelligent Transportation Systems.* Dissertation/Ph.D. thesis, University Duisburg Essen, 2002.

A Selected code excerpts

A.1 Configuration Files

A.1.1 Server configuration

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

<server>
  <logging level="INFO"/>
  <services>
      <network>
    <source class="rds.easyway.EasyWaySource" device="/dev/ttyS0"
        frequency="107.2">
      <node class="rds.easyway.EasyWayRDSChunk2RDSBlock">
        <node class="rds.RDSBlock2ByteArray">
          <drain class="generics.ByteArraySocketDrain" port="9777"
              period="1"/>
        </node>
      </node>
    </source>
        </network>
    </services>
</server>
```

A.1.2 Filter mechanism configuration

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

<server>
  <logging level="INFO">
  </logging>
  <services>
      <network>
        <source class="generics.StringSocketSource" host="localhost"
            port="9778">
          <node class="xml.SpexNode" xpath="/descendant::TMCMessage[
              descendant::locationName/child::text()='Bayern']">
            <drain class="generics.ConsoleDrain"/>
          </node>
        </source>
        </network>
    </services>
</server>
```

A.1.3 Raw XML configuration

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

<server>
  <logging level="INFO">
  </logging>
  <services>
      <network>
        <source class="generics.ByteArraySocketSource" host="nihiru.pms.
            ifi.lmu.de" port="9777">
          <node class="rds.ByteArray2RDSBlock">
            <node class="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.RDSGroup2TMCMessage"
                              locationDB="LocationList_de.csv" eventDB="
                              EventList_en_de.csv" lang="de_DE">
                            <node class="xml.TMCMessage2Xml" prettyPrint
                                ="true">
                              <drain class="generics.ConsoleDrain"/>
                              <drain class="generics.StringSocketDrain"
                                  port="9778"/>
                            </node>
                        </node>
                    </node>
                  </node>
            </node>
          </node>
          </node>
        </source>
        </network>
    </services>
</server>
```

A.1.4 Visualisation configuration

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

<server>
  <logging level="INFO">
  </logging>
    <services>
        <network>
```

```
            <source class="generics.ByteArraySocketSource" name="
                SocketSource" host="nihiru.pms.ifi.lmu.de" port="9777">
              <node class="rds.ByteArray2RDSBlock">
              <node class="rds.RDSBlock2RDSGroup">
              <node class="generics.Filter">
                <condition class="generics.AndCondition">
                  <condition class="rds.RDSGroupFilterCondition" group="
                      8A"/>
                  <condition class="rds.RDSGroupCorrectnessCondition"/>
                </condition>
                  <node class="rds.RDSGroup2RDSGroup">
                        <drain class="HumanreadableRDSGroupDrain" name="
                            HumanReadableDrain"/>
                </node>
                  </node>
                <node class="rds.RDSGroup2RDSGroup">
                      <drain class="visualization.RDSTestFrameDrain"
                          locationDB="src/tmcdatabases/LocationList_de.
                          csv" locationNameDB="de.lmu.ifi.pms.sttiprak.
                          tmc.LocationNames" eventDB="src/tmcdatabases/
                          EventList_en_de.csv"/>
                 </node>
                </node>
              </node>
            </source>
        </network>
    </services>
</server>
```

A.2 Schemas

A.2.1 RelaxNG-Schema for Configurations

```
grammar {
  start =
    element server {
      #
      #  Defines the logging levels
      #
      element logging {
        #
        #  Defines the global logging level
        #
        attribute level { "DEBUG" | "INFO" | "WARNING" | "ERROR" | "
            FATAL" },

        #
        #  Defines the logging level for a single class.
        #
        element logger {
          attribute class { text },
          attribute level { "DEBUG" | "INFO" | "WARNING" | "ERROR" | "
              FATAL" }
        }?
      }
      element services {
      element network {
        element source {
          nodeConfig,
          nodeOrDrain*
        }+
      }
    }

  nodeConfig = attribute class { text }
    attribute * { text } # any further Attributes may be specified here.

  nodeOrDrain = ( element node  { nodeConfig, nodeOrDrain* }
          | element drain { nodeConfig, nodeOrDrain* } )
}
```

A.2.2 RelaxNG-Schema for XML-TMC stream

```
grammar {

#
#   Defines the stream.
#
```

```
start =
    element TMCStream {
        element TMCMessage {
            #
            #   The meta element contains additional information
               about the
            #   TMCMessage.
            #
            element meta{
                element id{ xsd:integer },
                element timestamp{ text }, # should be: xsd:dateTime
                element frequency{ xsd:float },
                element sender{ text }?
            },
            #
            #   The primary element of the TMCMessage will mostly be
                the
            #   only one in the TMCMessage element. It contains the
               basic
            #   information contained in every TMCMessage. All other
                elements
            #   are optional and only used if the data is provided
               by the sender.
            #
            element primary {
                eventElement+,
                multiLocationElement+
            },
            #
            #   References to the initial cause of the problem
               described in the primary
            #   element (such as a queue on route A caused by an
               accident on route B)
            #
            element crosslink {
                eventElement,
                singleLocationElement
            }?,
            #
            #   Describes a diversion hint. Sometimes a diversion is
                specific for certain
            #   destinations, sometimes global.
            #   Sometimes a detailed diversion advice is given,
               sometimes not.
            #
            element diversion {
                element advicedRoute {
                    multiLocationElement+
                }?,
                destinationElement*
            }*
            }*
        }

#
#   Describes an event. An event may affect certain drivers to
#   a destination only.
#
    eventElement = (knownEvent | unknownEvent)

    unknownEvent =
      element event {
        element unknownEventCode{ xsd:int }
      }

    knownEvent =
        element event {
          element eventCode { xsd:int },
            element eventName { multilanguage }+,
            element updateClass
            {
                element code{ xsd:int },
                element name{ multilanguage }+
            },
            element nature
            {
                element code{ "information" | "forecast" | "silent" },
                element name{ multilanguage }+
            },
            element urgency
            {
                element code{ "extremely␣urgent" | "urgent" | "normal"
                    },
                element name{ multilanguage }+
            },
            element quantifier
            {
                element code{ xsd:int },
                element name{ multilanguage }+,
                element value{ xsd:decimal }
            }?,
            element duration
            {
                element code{ "short" | "longer" },
                element startTime{ text }?, #  should be: xsd:dateTime
                ( element stopTime{ text } #  should be: xsd:dateTime
            | element until{ text } #  should be: xsd:dateTime
            )
            },
            #
            #   Describes a speed limit advice for drivers to specific
                locations.
            #
            element speedAdvice
            {
                element limit {
                 attribute unit { text },
                 xsd:int
               },
                destinationElement?
            }?,
            element lengthOfRouteAffected{ (xsd:int | "more␣than␣100␣km"
                ) }?,
            supplementaryElement*,
            destinationElement?
        }

    #
    # Describes a supplementary information element. An information
    # may affect only given for drivers to a specific destination.
    #
    supplementaryElement = knownSI | unknownSI

    unknownSI =
      element supplementaryInformation {
        element unknownCode { xsd:int }
      }

    knownSI =
        element supplementaryInformation {
            element code { xsd:int },
            element name { multilanguage }*,
            destinationElement?
        }

    #
    # A destination describes a location to which a diver may
    # pass by or drive to.
    #
    destinationElement =
        element destination
        {
            singleLocationElement
        }

    #
    # A multiLocationElement describes a nested location tree structure,
    # a location may have more than one children.
    #
    multiLocationElement =
      ( knownMLE
      | unknownLocation
      | element unknownContinuation { empty }
```

```
        )

     unknownLocation =
       element location
       {
         element unknownLocationCode{ xsd:int }
       }

     knownMLE =
         element location
         {
             location.id,
             (   (linearLocationContent?, multiLocationElement*)
             |   pointLocationContent
             )
         }

#
#   A singleLocationElement describes a nested location path structure,
#   a location may have at most one child.
#
     singleLocationElement = (knownSLE | unknownLocation)

     knownSLE =
         element location
         {
             location.id,
             (   (linearLocationContent?, singleLocationElement)
             |   pointLocationContent
             )
         }

    location.id =
         element locationCode{ xsd:int },
         element locationName{ multilanguage }*,
         element type{ text },
         element typeName{ multilanguage }*

    linearLocationContent =
         element roadNumber{ text },
         element positiveEndName{ multilanguage }*,
         element negativeEndName{ multilanguage }*,
         element direction{ "unknown" | "positive" | "negative" | "both"
             }

     pointLocationContent =
         element direction{ "unknown" | "positive" | "negative" | "both"
             }

     multilanguage =
         attribute lang{ text },
            text
}
```