Ludwig-Maximilians-Universität München
Institut für Informatik, Lehrstuhl für Programmier- und Modellierungssprachen
Fortgeschrittenenpraktika im Wintersemester 2005/2006

Protokollierung und statistische Auswertung von RDS/TMC Datenströmen

8. August 2006

Aufgabensteller: Prof. Dr. Hans Jürgen Ohlbach
Betreuer: Dipl.-Inform. Bernhard Lorenz
Bearbeiter: Markus Krieser,
Michael Buschmann

Inhaltsverzeichnis

1 Einleitung **5**

2 Einführung und Ziele **7**
2.1 Begriffsdefinition 7
2.2 Ausgangsposition des Projektes 11
2.3 Ziele der Arbeit 14

3 Datenspeicherung **17**
3.1 Entwurf der Datenbank 17
3.2 Parsen der Code Listen 21
3.3 Konfiguration der XML Ausgabe 23
3.4 Datenspeicherung durch Verwendung der XML Ausgabe 25
3.5 Hinzunahme eines 2. Empfängers 26
3.6 Verwendung der Datenbank als Quelle 27
3.7 Zusammenfassung 29

4 Statistische Auswertung **31**
4.1 Voraussetzungen 31
4.2 Erstellen des Frameworks 31
4.3 Konfigurieren des Frameworks 32
4.4 Datenbank-Verbindung 34
4.5 Testcases 35
4.5.1 Welches Ereignis tritt am Häufigsten auf? 35
4.5.2 Brutto- /Nettoeventanzahl der Nachrichten 36

5 Ausblick **39**

6 Abbildungsverzeichnis **41**

7 Literaturverzeichnis **43**

1 Einleitung

Das Projekt zur Protokollierung und Auswertung der Datenströme bei RDS/TMC Nachrichten knüpft an ein vorangegangenes Praktikum an. In diesem wurde ein System entwickelt, um Verkehrs- und Reiseinformationen (Traffic and travel information, TTI) zu beobachten, wobei diese von Verkehrslagen bis Wetterbedingungen eine Menge an Daten enthalten (vgl. [7]). Die Informationen sind in RDS/TMC Radiosignalen kodiert und wurden mit Hilfe des Systems in einen XML Strom umgewandelt, damit diese in einer strukturierten Form am Monitor ausgegeben werden können. Zusätzlich kam es zu einer Aufbereitung der Daten mit weiteren Informationen, wie den Zeitpunkt, wann eine Nachricht eingetroffen ist oder die Auflösung der verschiedenen Codes zu einer verständlichen Bedeutung. Zuletzt wurde ein kontinuierlicher Strom von aufbereiteten Nachrichten im XML Format am Monitor angezeigt. Was hier fehlt, ist eine dauerhafte Speicherung der Daten, um langfristige Aussagen über bestimmte Verkehrsvorkommnisse machen zu können. So wäre es möglich, über bestimmte Verkehrsknoten zu einer bestimmten Zeit immer einen Stau zu melden oder gewisse Regionen aufzuzeigen, die während des Berufsverkehrs staugefährdet sind. Diese Informationen wären zum Beispiel für eine kommerzielle Nutzung interessant, um auch solche Faktoren in eine Routenberechnung einfließen zu lassen. Daher wird im diesen Projekt ein Grundstein für die dauerhafte Speicherung dieser Informationen gelegt.

Die zugrunde liegende Arbeit teilt sich in drei Teile auf: Zuerst wird eine Erklärung von bisherigen und weiteren Begriffen vorgenommen, die für die Arbeit wichtig sind. Zusätzlich wird die bisherige Situation genau geklärt und wo sich die Anknüpfungspunkte befinden. Der zweite Teil behandelt die Sammlung der Daten in einer Datenbank. Dabei muss zuerst die Datenbank modelliert und darauf alle notwendigen Informationen gespeichert werden. Mit Hilfe von Codebeispielen soll das Verfahren veranschaulicht werden. Im letzten Teil wird es zwei Beispiele zur statistischen Auswertung der gesammelten Daten geben. Auch hier soll zuerst das Grundkonzept vorgestellt werden und wie die grafische Oberfläche bewerkstelligt wurde. Zusätzlich kann die vorgestellte Oberfläche leicht modifiziert werden, um beliebige weitere Auswertungsmöglichkeiten zu implementieren.

Die Aufgaben wurden folgendermaßen verteilt:

- Gemeinsam bearbeitet:
  - Modellierung der Datenbank
  - Parsen der Listen

  - Erstellung des Frameworks

- Markus Krieser:
  - Konfiguration der XML Ausgabe
  - Speicherung der Daten in die Datenbank
- Michael Buschmann:
  - Verwendung der Datenbank als Quelle
  - Konfigurieren des Frameworks
  - Erstellung der Testcases

2 Einführung und Ziele

In diesem Abschnitt werden notwendigen Begriffe geklärt und ein kurzer Ausblick zu RDS gegeben. Dann folgt eine genaue Beschreibung des vorangegangenen Praktikums, um die Datenbank-Verknüpfung in nächsten Teil besser zu verstehen. Als letztes werden die Ziele der Arbeit genau definiert, vor allem diese, die bisher keine Erwähnung fanden.

2.1 Begriffsdefinition

Der hier wichtigste Begriff ist in diesem Zusammenhang das RDS. Dies steht für "Radio Data System" und ermöglicht eine Übermittlung von Nicht-Audio Zusatzinformation beim Radio/Rundfunk. RDS wurde ungefähr 1983 von der European Broadcasting Union (kurz: EBU) konzipiert und ist um 1987 zur Marktreife gelangt (vgl. [8]). Das Radio-Data-System ist in der DIN EN 62106 standardisiert. RDS wird hauptsächlich in Autoradios verwendet, da es durch die Übertragung der "Alternative Frequencies" möglich ist, ohne Benutzereingriff automatisch die Frequenz zu wechseln und somit einem einmal eingestellten Programm zu folgen (vgl. [8]). Dies erspart das manuelle Suchen nach der neuen Frequenz, wenn das Fahrzeug auf der Fahrt den Sendebereich eines Senders verlässt. Die übertragenen Datenbits sind mit einer Datenrate von 1,1875 kHz auf einen 57 kHz Unterträger des Radiosignals moduliert. Jeweils 26 Bits bilden einen Block, der wiederum aus 16 Datenbits und 10 Prüfbits besteht. Mit Hilfe der Prüfbits können die Blockgrenzen und die Art des Blocks detektiert werden. Jeweils vier Blöcke (ABCD bzw. ABC'D) bilden eine RDS-Gruppe (vgl. [8]). In Block A wird immer die 16 Bit Sender-ID (program information, PI) übertragen. In Block B findet man den Programmtyp (PTY), einen Indikator für Verkehrsfunk (TP) und die RDS-Gruppennummer (Groupe Type, GT). Diese gibt Auskunft über die Verwendung der restlichen fünf Bits des Blocks B und der 32 Bits von Block C und D (vgl. [8]). Es existiert eine Reihe von RDS Gruppen, die für verschiedene zusätzliche Datendienste verwendet werden (vgl. [8]). Die wichtigste Gruppe für dieses Projekt ist die RDSGroup8A (siehe Abb. 2.1).

Die hier wichtige Datendienst, der bei RDS übermittelt wird, sind die sogenannten TMC Nachrichten. TMC steht für "Traffic Message Channel". Dabei handelt es sich um einen kostenlosen, digitalen Radio-Datendienst, der zur Übertragung von Verkehrsinformationen an ein geeignetes Empfangsgerät verwendet wird. Da das Signal ständig übertragen wird, ist der Benutzer nicht von den nur halbstündlich ausgestrahlten Verkehrshinweisen abhängig. Außerdem können wichtige Informationen, wie z. B. bei Geisterfahrern sofort weitergegeben werden. Moderne Navigationssysteme verwenden TMC zur automatischen Umfahrung von Verkehrsstaus und Behinderungen (dynamische Zielführung). Die RDS Gruppe, die für TMC von Bedeutung ist, ist die Gruppe 8A. In dieser werden

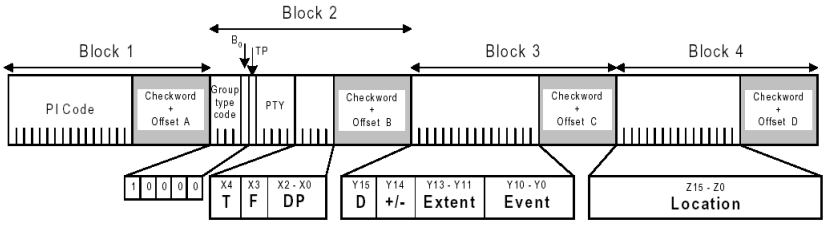

Figure 2 — RDS-TMC single-group full message structure

Key:

T = 0 indicates User message;
T = 1 indicates Tuning Information (or reserved for future use)

F = 0 indicates multi-group message;
F = 1 indicates single-group message

DP = Duration and Persistence values

D = 0 indicates no diversion is recommended
D = 1 indicates that drivers are advised to follow the indicated diversion

+/- = 0 indicates positive direction
+/- = 1 indicates negative direction

Abbildung 2.1: RDSGroup8A ([4], Seite 27)

die Verkehrsinformationen nach folgender Art beschrieben: Jede Verkehrsbeeinträchtigung wird als separate TMC-Meldung gesendet. Eine TMC-Meldung besteht aus einem Ereigniscode (Event Code, EC) sowie einem Lagecode (Location Code, LC), evtl. einer zusätzlichen zeitlichen Beschränkung (Verfallszeit). Nach Wegfall einer Störung wird diese Störungsmeldung durch eine weitere TMC-Meldung aufgehoben. Falls diese nicht empfangen wird (z. B. weil der Empfänger sich mittlerweile aus dem Empfangsbereich des Senders entfernt hat), wird die Störungsmeldung alternativ bei Erreichen der Verfallszeit durch den Empfänger gelöscht. Störungsmeldungen werden in gewissen Intervallen wiederholt, solange bis eine Aufhebungsmeldung kommt.

Die Meldung ist nach dem Alert C-Standard codiert (vgl. [6]). Dieser beinhaltet eine Liste mit ungefähr 1460 Ereignissen (vgl. [3]. Mit Hilfe dieser Liste kann die Meldung in eine vom Benutzer verständliche Form umgesetzt werden. Dabei wird nicht nur das Ereignis selber angezeigt, sondern es enthält auch zusätzliche Informationen, ob es sich z.B. um ein aktuelles Ereignis oder um eine Prognose handelt (siehe Abb.2.2).
Der 'Location Code' wird in der 'Location Code List' (LCL) national gepflegt. Jedes Land vergibt die Zahlen für die einzelnen Straßenabschnitte demnach selber. Die Verkehrsnetz in Deutschland ist zur Zeit in über 32000 Straßenabschnitte aufgeteilt (vgl. [3]. In der LCL sind zu dem jeweiligen Code weitere Informationen gespeichert (siehe Abb.2.3), z.B. welchen Code der folgende und vorherige Streckenabschnitt hat. Mit dieser Information lässt sich ein Stau über mehrere Streckenabschnitte definieren, da bei der

Line	Text CEN-English)	Text (German)	Text (Quantifier = Einzahl)	Text (Quantifier = Mehrzahl)	Code	N	Q	T	D	U	C	R
2												
3	**1. LEVEL OF SERVICE**	**1. Verkehrslage**										
4												
5	traffic problem	Verkehrsbehinderung			1			D	1	U	1	A5
6	stationary traffic	(L) Stau			101			D	1	U	1	A1
7	stationary traffic for 1 km	1 km Stau			102			D	1	U	1	A1
8	stationary traffic for 2 km	2 km Stau			103			D	1	U	1	A1
9	stationary traffic for 3 km	3 km Stau			129			D	1	U	1	A1
10	stationary traffic for 4 km	4 km Stau			104			D	1	U	1	A1
11	stationary traffic for 6 km	6 km Stau			105			D	1	U	1	A1
12	stationary traffic for 10 km	10 km Stau			106			D	1	U	1	A1
13	danger of stationary traffic	Staugefahr			130			D	1	U	1	A1
14	queuing traffic (with average speeds Q)	(L) stockender Verkehr			108		4	D	1	U	1	A2
15	queuing traffic for 1 km (with average speeds Q)	1 km stockender Verkehr			109		4	D	1	U	1	A2
16	queuing traffic for 2 km (with average speeds Q)	2 km stockender Verkehr			110		4	D	1	U	1	A2
17	queuing traffic for 3 km (with average speeds Q)	3 km stockender Verkehr			131		4	D	1	U	1	A2
18	queuing traffic for 4 km (with average speeds Q)	4 km stockender Verkehr			111		4	D	1	U	1	A2
19	queuing traffic for 6 km (with average speeds Q)	6 km stockender Verkehr			112		4	D	1	U	1	A2
20	queuing traffic for 10 km (with average speeds Q)	10 km stockender Verkehr			113		4	D	1	U	1	A2
21	danger of queuing traffic (with average speeds Q)	Gefahr von stockendem Verkehr			132		4	D	1	U	1	A2
22	long queues (with average speeds Q)	lange Staus			133		4	D	1	U	1	A7
23	slow traffic (with average speeds Q)	(L) dichter Verkehr			115		4	D	1	U	1	A3
24	slow traffic for 1 km (with average speeds Q)	1 km dichter Verkehr			116		4	D	1	U	1	A3
25	slow traffic for 2 km (with average speeds Q)	2 km dichter Verkehr			117		4	D	1	U	1	A3
26	slow traffic for 3 km (with average speeds Q)	3 km dichter Verkehr			134		4	D	1	U	1	A3
27	slow traffic for 4 km (with average speeds Q)	4 km dichter Verkehr			118		4	D	1	U	1	A3
28	slow traffic for 6 km (with average speeds Q)	6 km dichter Verkehr			119		4	D	1	U	1	A3
29	slow traffic for 10 km (with average speeds Q)	10 km dichter Verkehr			120		4	D	1	U	1	A3
30	heavy traffic (with average speeds Q)	(L) reger Verkehr			122		4	D	1		1	A4
31	traffic heavier than normal (with average speeds Q)	sehr viel dichterer Verkehr als normal			142		4	D	1		1	A1
32	traffic very much heavier than normal (with average speeds Q)	dichterer Verkehr als normal			143		4	D	1		1	A1
33	traffic flowing freely (with average speeds Q)	störungsfreier Verkehr			124		4	(D)	1		1	A5

Abbildung 2.2: Ausschnitt der Event List

Übertragung der TMC Nachricht, nur ein zusätzlicher Wert "Ausdehnung" notwendig ist.

Das Zusammenwirken von allen Komponenten, die beim "Traffic and Travel Information Broadcasting" mitwirken, ist in Abbildung 2.4 auf Seite 11 dargestellt.

Diese Abbildung zeigt die Schritte, wie die Verkehrsdaten beim Endverbraucher eintreffen, wobei zuerst Daten gesammelt werden müssen. Dies kann durch unterschiedliche Institutionen, wie der ADAC oder die Verkehrspolizei geschehen. Die Informationen aus den verschiedenen Quellen müssen in einer zentralen Stelle gesammelt werden (im Bild als 'Traffic Information Centre' dargestellt). Dann kommt RDS/TMC ins Spiel, mit dessen Verfahren die Daten zuerst kodiert und dann übertragen werden. Die Kodierung erfolgt mit Hilfe der RDS Gruppe 8A in TMC Nachrichten, die die oben beschriebenen Informationen (LC, EC, etc.) enthalten. Danach wird mit dem RDS und Broadcasting, z.B. durch die Sendeantenne eines Radiosenders, die Daten gesendet. Diese treffen beim Endnutzer (z.B. Autofahrer) ein und wenn dieser über das entsprechendes Empfangsgerät (z.B. ein RDS-fähiges Radio oder Navigationsgerät) verfügt, können die Informationen dekodiert und verarbeitet werden. Dies können einzeilige Anzeigen an Radiogeräten sein oder von Navigationsgeräten auf einem Bildschirm dargestellt und z.B. in eine Reiseroute mit eingeplant werden.
Die Säule "Future additional TTI service" zeigt zukünftige Möglichkeiten an Services. Dieses Projekt ist der "Private data collection" zuzuordnen, da dieser Punkt eine Sammlung von Daten beschreibt, die in der Navigation mit eine Rolle spielen kann. Dieses Projekt greift genau dies auf, denn es speichert die Daten permanent, um daraus statistische

LOCATIONCODE	TYPE	SUBTYPE	ROADNUMBER	ROADNAME	FIRST_NAME	SECOND_NAME	AREA_REFERENCE
1	A3	0			Deutschland		34196
2	A3	0			Frankreich		34196
3	A3	0			Belgien		34196
4	A3	0			Niederlande		34196
5	A3	0			Luxemburg		34196
6	A3	0			Schweiz		34196
7	A3	0			Tschechien		34196
8	A3	0			Polen		34196
9	A3	0			Österreich		34196
10	A3	0			Ungarn		34196
11	A3	0			Italien		34196
12	A3	0			Griechenland		34196
13	A3	0			Dänemark		34196
256	A7	0			Schleswig-Holstein		1
257	A7	0			Hamburg		1
258	A7	0			Niedersachsen		1
259	A7	0			Bremen		1
260	A7	0			Nordrhein-Westfalen		1
261	A7	0			Hessen		1
262	A7	0			Rheinland-Pfalz		1
263	A7	0			Baden-Württemberg		1
264	A7	0			Bayern		1
265	A7	0			Saarland		1
266	A7	0			Berlin		1
267	A7	0			Brandenburg		1
268	A7	0			Mecklenburg-Vorpommern		1
269	A7	0			Sachsen		1
270	A7	0			Sachsen-Anhalt		1
271	A7	0			Thüringen		1
272	A8	0			Arnsberg		260
273	A8	0			Braunschweig		258
274	A8	0			Chemnitz		269
275	A8	0			Reg.-Bez. Darmstadt		261
276	A8	0			Reg.-Bez. Dessau		270
277	A8	0			Detmold		260

Abbildung 2.3: Ausschnitt der Location List

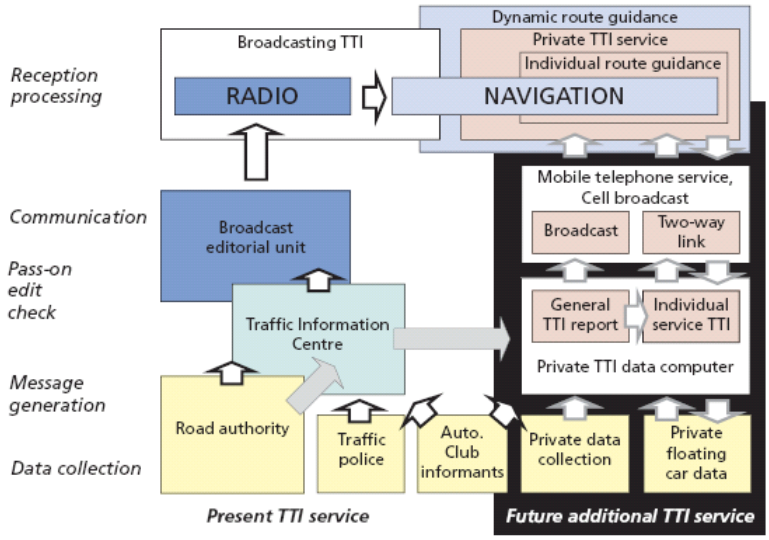

Abbildung 2.4: Die komplette TTI Umgebung: bisherige und zukünftige Dienste

Aussagen zu treffen.

Nach der konzeptionellen Vorstellung des RDS/TMC Systems, wird im nächsten Abschnitt das bisherige System, wie es im Praktikum modelliert und implementiert wurde, genau vorgestellt.

2.2 Ausgangsposition des Projektes

Bei der Modellierung des Systems galt es den eintreffenden Datenverkehr in eine menschlich lesbare Ausgabe umzuwandeln. Dazu waren mehrere Schritte und Umwandlungen notwendig. Mit Hilfe eines am seriellen Port (ttyS0) angeschlossenen RDS/TMC Empfängers können die Radio Signale empfangen werden. Aus diesem gesamten Datenstrom müssen die für hier wichtigen Informationen herausgefiltert werden. Dies geschieht mit Hilfe eines Servers, der zuerst den Binärstrom eines bestimmten Radiosenders (in diesem Fall auf der Frequenz 104,6) herausfilterte. In einem weiteren Schritt wurden die Gruppe 8A Blöcke des RDS ermittelt. In diesen Blöcken waren die TMC Nachrichten gespeichert. Unter Verwendung der offiziellen ISO Spezifikation ([4], [5] und [6]) konnten die Informationen der Blöcke zu einer TMC Nachricht umgewandelt werden. Es können mehrere TMC Nachrichten in einem RDS Block vorhanden sein. Für die weitere Verarbeitung muss aber jede TMC Nachricht einzeln vorliegen. Diese verschiedenen Schritte verarbeitet ein Server mit einer entsprechenden Konfiguration. In der Konfiguration wurde unter anderem die Radio Frequenz und Portnummer angegeben. Detaillierte Informationen zu der Serverkonfiguration findet sich im Abschnitt 3.3.

Dieser Server wurde nach dem Erzeuger/Verbraucher Modell entworfen. So 'verbraucht'

er einerseits den antreffenden Bitstrom am seriellen Port und nach Filterung 'erzeugt' dieser die TMC Nachrichten, die von einem weiteren Server als Verbraucher wieder genutzt werden können. Ein Beispiel, wie dies genau aussieht, wird in Abschnitt 3 gezeigt.

Es müssen die einzelnen TMC Nachrichten mit entsprechenden Informationen aufgewertet werden, sodass eine informative Ausgabe zustande kommt. Ein zweiter Server verarbeitet hier die TMC Nachrichten und liest die Informationen, wie Event Code, Location Code und Zusatzinformationen aus. Anhand der zugehörigen Tabellen (vorliegend als .csv Dateien) können, neben der Übersetzung des Codes in eine verständliche Sprache, die in der Tabelle angegebenen Zusatzinformationen verwendet werden. All diese Daten werden anschließend in einem XML Format ausgegeben. Im vorliegenden XML Strom zeigt sich das Schema, wie eine aufgeschlüsselte Nachricht angezeigt wird:

```
<TMCMessage>

 <meta>
 <id>1</id>
        <timestamp>2006-03-12T10:56:37+0100</timestamp>
        <frequency>101.30</frequency>
 </meta>

 <primary>
       <event>
       <eventCode>394</eventCode>
       <eventName lang="de">Gefahr durch defekten LKW</eventName>
          <updateClass>
          <code>4</code>
          <name lang="de">Vorfälle</name>
          </updateClass>
         <nature>
         <code>information</code>
         <name lang="de">Information</name>
         </nature>
         <urgency>
         <code>urgent</code>
         <name lang="de">dringend</name>
         </urgency>
         <duration>
         <code>short</code>
         <until>2006-03-12T11:11:37+0100</until>
         </duration>
       </event>

 <location>
       <locationCode>1</locationCode>
       <locationName lang="de">Deutschland</locationName>
       <type>A3.0</type>
```

```
35        <typeName lang="de">Land</typeName>
            <location>
37          <locationCode>50167</locationCode>
            <type>L1.1</type>
39          <typeName lang="de">Autobahn</typeName>
            <roadNumber>A8</roadNumber>
41          <positiveEndName lang="de">Luxemburg</positiveEndName>
            <negativeEndName lang="de">Salzburg</negativeEndName>
43          <direction>unknown</direction>
              <location>
45            <locationCode>7210</locationCode>
              <type>L3.0</type>
47            <typeName lang="de">Segment 1. Ordnung</typeName>
              <roadNumber>A8</roadNumber>
49            <positiveEndName lang="de">München</positiveEndName>
              <negativeEndName lang="de">Salzburg</negativeEndName>
51            <direction>unknown</direction>
                <location>
53              <locationCode>12579</locationCode>
                <locationName lang="de">Irschenberg</locationName>
55              <type>P1.3</type>
                <typeName lang="de">Autobahnanbindung</typeName>
57              <direction>unknown</direction>
                </location>
59                <location>
                  <locationCode>12578</locationCode>
61                <locationName lang="de">Weyarn</locationName>
                  <type>P1.3</type>
63                <typeName lang="de">Autobahnanbindung</typeName>
                  <direction>unknown</direction>
65                </location>
              </location>
67          </location>
          </location>
69  </primary>

71 </TMCMessage>
```

Zuerst wird im `meta` - Bock eine fortlaufende Nummer vergeben und die Eingangszeit der Nachricht angezeigt. Danach wird zuerst der Event Code aufgeschlüsselt (hier im Beispiel 394 zu "Gefahr durch defekten LKW"). Darauf folgt der Ort, an dem dieses Event eingetreten ist. Dabei wird dieser in einer Baumstruktur aufgeschlüsselt, um die verschiedenen Stufen zu sehen (Deutschland -> Autobahn A8 -> Segment 1.Ordnung von München nach Salzburg -> Irschenberg). Weitere Informationen sind hier auch ersichtlich (z.B. dringender Vorfall 'urgent'). Betrifft dieses Problem mehrere Abschnitte, folgen daraufhin mehre Location Codes. Wie in diesem Beispiel betrifft es nicht nur den Location Code 12579 ("Irschenberg") sondern auch den Abschnitt 12578 ("Weyarn"). Mit den zugehörigen Close-Tags wird die XML Nachricht geschlossen und es folgt die

nächste TMC Nachricht.

Für detailliertere Informationen über das bisherige Projekt soll einerseits die Readme der Vorgruppe (zu finden in deren Projektverzeichnis), die verwendeten Java Klassen und die ISO Standards (vgl. [4], [5] und [6]) dienen. Eine grundlegende Beschreibung des Projektes findet sich unter [7].

Deutlich hat sich gezeigt, das zwar die Informationen anschaulich präsentiert werden, aber bei einer Flut von Nachrichten, die fast im Sekundentakt über den Bildschirm laufen, keine langfristigen Aussagen getroffen werden können. Auch durch die stetige Wiederholung der Nachrichten, geht schnell die Übersicht verloren.

Mit diesem diesem Projekt wird daran angeknüpft. Daraus ergeben sich die im nächsten Abschnitt definierten Ziele für diese Projektarbeit.

2.3 Ziele der Arbeit

Hier werden die Ziele und Überlegungen aufgezählt, wie sie in den nächsten beiden Kapiteln erklärt und definiert werden.

1. **Datenmodellierung**: Um die Nachrichten zu speichern, muss eine für diesen Zweck geeignete Datenbank entworfen werden (siehe Abschnitt 3.1).

2. **Datenspeicherung**: Dabei muss auf das vorherige Projekt aufgebaut werden. Ein weiterer 'Verbraucher' nimmt die Daten und zeigt sie nicht als XML Ausgabe, sondern legt diese entsprechend in die Datenbank ab. Hier sind mehrere Punkte (siehe Abschnitt 3.2 bis 3.4) zu beachten.

3. **mehrere Datenquellen**: Sobald die Daten von einem Empfänger erfolgreich in die Datenbank gespeichert werden, sollen die Informationen eines 2. Radiosenders gespeichert werden. Dadurch können in späteren Auswertungen Ähnlichkeiten oder Abweichungen zwischen den Sendern festgestellt werden (siehe Abschnitt 3.5).

4. **künstliche Datenquelle**: Es soll die Möglichkeit geben, das die gefüllte Datenbank als Sender fungiert. So kann ein Startzeitpunkt gewählt werden und die Datenbank schickt daraufhin die Nachrichten in richtiger Reihenfolge und Abstand als XML Ausgabe. So können Tests unternommen werden, auch wenn an dem entsprechenden Rechner kein Empfänger installiert ist (siehe Abschnitt 3.6).

5. **Statistische Auswertung** der gesammelten Daten: Nachdem die Daten über einen Zeitraum gesammelt wurden, werden diese entsprechend ausgewertet. Dazu wird eine Oberfläche erstellt, die auch eine leichte Erweiterung von anderen Auswertungsfällen (Uses Cases) ermöglicht (siehe Abschnitt 4).

Welche Umwandlungen von den einzelnen Komponenten gemacht werden, verdeutlicht Abbildung 2.5. Hier ist das gesamte Konzept grafisch dargestellt, wobei der obere Teil

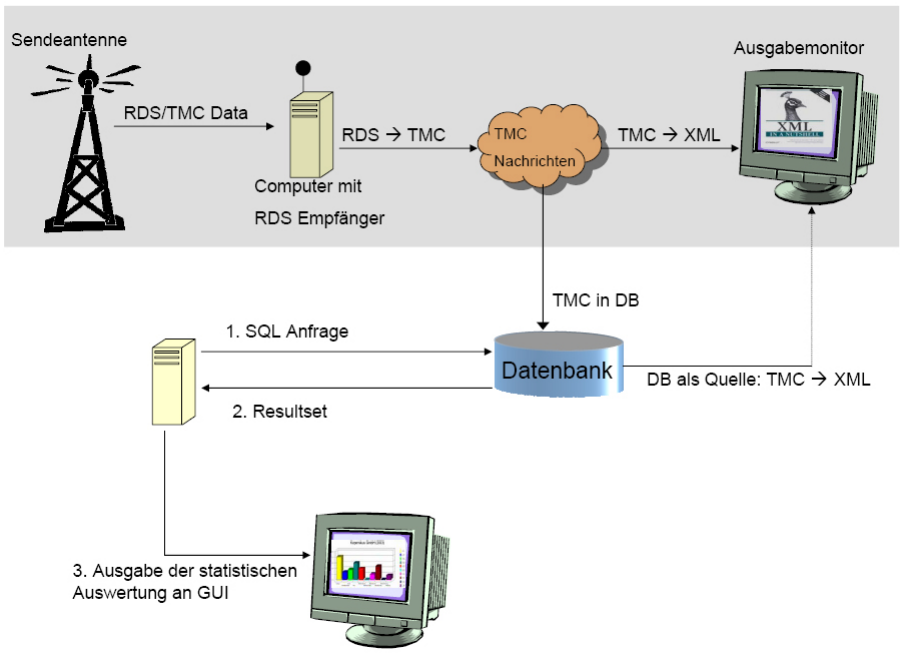

Abbildung 2.5: Kompletter Datenfluss und Umwandlung des gesamten Projektes

(grau unterlegt) das vorherige Projekt zeigt. Die Erweiterung durch die Datenbank und der statistischen Ausgabe ist Gegenstand dieses Projektes (unterer Teil des Bildes).

3 Datenspeicherung

Dieses Kapitel beschreibt die Umsetzung der Ziele 1. bis 4. der Arbeit. Dabei wird im ersten Schritt die Datenbank entworfen, dann folgt das Ablegen der beiden Listen für Event- und Locationcode in die Datenbank. Als nächstes wird die XML Ausgabe erweitert und verändert, so dass im folgenden Schritt der Nachrichtenspeicherung in die Datenbank nichts mehr im Wege steht. Damit ist die permanente Speicherung der Ereignisse abgeschlossen. Als eine zusätzliche Option gibt es die Möglichkeit die Datenbank als Quelle zu verwenden, um einen Sender zu simulieren. Als letztes wird auf die auftauchenden Probleme während des Projektes eingegangen.

3.1 Entwurf der Datenbank

Beim Entwurf der Datenbank waren folgende Überlegungen wichtig: Welche Informationen müssen gespeichert, wie sollen sie gespeichert werden und wie viele Tabellen werden benötigt.
Wie im Codebeispiel der XML Ausgabe zu sehen war, wurden die Informationen so weit aufbereitet, das diese anschaulich und lesbar waren. Die Datenbank soll nur die Codes speichern,d.h. ein Minimum an Daten, wobei keine Informationen verloren gehen. Dies sind neben 'PI Code' (die eindeutige Sendernummer), 'Event Code', 'Location Code', 'Direction' (Boolean über eine mögliche Umleitungsangabe im optionalen Anhang), 'Duration' (Dauer des Events), 'Direction' (in welche Richtung das Problem besteht) und 'Extend' (über wie viele Streckenabschnitte sich das Problem ausdehnt). Zusätzlich gibt es einen optionalen Anhang, der weitere Informationen speichern kann. Für die optionalen Daten wurde eine eigene Tabelle erstellt, um mögliche Auswertungen leichter erstellen zu können. So kann ausgewertet werden, welcher Sender diese Möglichkeit überhaupt nutzt und welche Informationen zusätzlich mitgeschickt werden. Bei den beobachtenden Sendern wurde die Verwendung von optionalen Daten kaum genutzt, außer wenn durch 'Extend' mehrere Streckenabschnitte betroffen waren, wurden die Location Codes zusätzlich im optionalen Anhang mitgeschickt. Ob andere Sender mehr optionale Information anhängen oder ob dies in anderen Länder mehr verwendet wird, kann hier nicht gesagt werden. Doch die Möglichkeit zur Untersuchung dieser Inhalte ist mit dieser Modellierung gegeben.
In Abbildung 3.1 ist die Tabelle für eine Nachricht ('message') zu sehen. Neben den enthaltenen Informationen der TMC Nachricht, wurden zusätzliche Informationen gespeichert, die nicht direkt aus der TMC Nachricht kommen. So müssen folgende Informationen generiert werden: Die fortlaufende Nummer ('message_nr') der Nachrichten, der Zeitstempel ('timestamp'), wann eine Nachricht eingetroffen ist, aus welchem Land die Nachricht kommt ('country') und die Sendefrequenz ('frequency_id'). Die Sendefrequenz

Feld	Typ	Attribute	Null	Standard	Extra
message_nr	bigint(20)		Nein		auto_increment
pi_code	varchar(10)		Nein		
eventcode	int(11)		Nein	0	
primaryloc	int(11)		Nein	0	
direction	tinyint(1)		Ja	*NULL*	
extent	int(11)		Nein	0	
diversion	tinyint(1)		Nein	0	
timestamp	timestamp(14)		Ja	*NULL*	
country	varchar(5)		Nein		
frequency_id	varchar(20)		Nein	0	
duration	int(11)		Nein	0	

Abbildung 3.1: struktureller Aufbau der Tabelle "message"

Feld	Typ	Attribute	Null	Standard	Extra
optional_nr	bigint(20)		Nein		auto_increment
message_nr	bigint(20)		Nein	0	
label	int(11)		Nein	0	
content	varchar(16)		Nein		

Abbildung 3.2: struktureller Aufbau der Tabelle "optional"

darf nicht mit dem PI Code verwechselt werden, da die 'frequency_id' nicht eindeutig ist.
Die Tabelle für die optionalen Daten ist in Abbildung 3.2 zu sehen. Auch hier gibt es eine fortlaufende Nummer ('optional_nr'), die 'message_nr', um die Verbindung zwischen Nachricht und der möglichen Zusatzinformation zu gewährleisten und die beiden Felder, in der die eigentlichen Daten gespeichert sind ('label' und 'content'). Der ISO Standard definiert 16 Labels (siehe [4] auf Seite 16), die die jeweiligen Zusatzinformationen definieren. Der Inhalt richtet sich dementsprechend danach. In diesem Projekt wurde vor allem das Label 9 verwendet, das zusätzlich betroffene Streckenabschnitte angibt, falls in der Nachricht ein Wert für 'Extend' vergeben war.

Mit diesen beiden Tabellen war der Entwurf der TMC Nachrichten beendet, aber es waren zusätzliche Tabellen anzulegen. So wurden die Tabellen für die Location und Event Codes zusätzlich in der Datenbank gespeichert. Diese lagen bisher nur als csv-Dateien vor. Vor dem Import müssen zuerst die entsprechenden Tabellen angelegt werden. Diese Tabellen wurden exakt so angelegt, wie auch die Spalten in den Dateien waren. So können

Feld	Typ	Attribute	Null	Standard	Extra
line	int(11)		Nein	0	
code	int(11)		Nein	0	
n	char(2)		Nein		
q	int(11)		Nein	0	
t	char(2)		Nein		
d	int(11)		Nein	0	
u	char(2)		Nein		
c	int(11)		Nein	0	
r	varchar(10)		Nein		

Abbildung 3.3: struktureller Aufbau der Tabelle "events"

später alle Informationen die zur Aufbereitung der TMC Nachrichten notwendig sind, wieder miteinander verknüpft und ausgegeben werden. In Abbildung 3.3 und 3.4 sind die angelegten Tabellen zu sehen.

Des Weiteren fehlen zwei Tabellen. Die Tabelle 'duration', in der aus dem Wert der TMC Nachricht beim Feld 'Duration' und dem Eintrag im Eventcode (Spalte 't') die entsprechende Dauer des Events bestimmen lässt. Diese Tabelle wurde per Hand eingetragen, da diese nur in der Spezifikation (siehe [4] auf Seite 12 f.) vorliegt. Die Angabe der Dauer eines Ereignisses wurde bei den betrachtenden Sendern nicht verwendet und wurde entweder auf 0 ("no explicit duration to be given") oder gar nicht gesetzt. So konnte auch hier keine Aussage darüber getroffen werden, was zu dem damaligen Planungszeitpunkt aber nicht ersichtlich war. Trotzdem besteht die Möglichkeit, falls andere Sender dies nutzen, es zu verwenden und Auswertungen darüber zu erstellen.

Als letztes wurde die Tabelle 'language' angelegt, in der die Übersetzungen der Event Codes gespeichert sind. So war in der vorliegenden Event-Liste sowohl das technische Englisch (CEN) als auch die deutsche Übersetzung gespeichert. Für die spätere Verwendung der Datenbank als Quelle kann in einem zusätzlichen Parameter festgelegt werden, welche Sprache gewünscht ist. Dies kann für andere Sprachen beliebig erweitert werden, wenn die entsprechende Liste vorliegt. Mit einer geringen Modifikation des Parsers im nächsten Abschnitt kann dies leicht automatisiert werden.

Dies wurde nur für die Event Codes gemacht, weil diese Liste für alle Länder, die dieses System verwenden, gleich ist. Die Location Tabelle hingegen wird, wie schon oben beschrieben, von den jeweiligen Ländern selbst erstellt und verwaltet. So bezieht sich ein bestimmter Location Code in Deutschland auf einen anderen Sreckenabschnitt als in anderen Ländern. Die Listen liegen nur in der jeweiligen Landessprache vor. Diese

Feld	Typ	Attribute	Null	Standard	Extra
locationcode	int(5)	UNSIGNED	Nein	0	
country	varchar(5)		Nein	de	
type	varchar(20)		Nein		
subtype	int(5)		Nein	0	
roadnumber	varchar(10)		Nein		
roadname	varchar(50)		Nein		
first_name	varchar(50)		Nein		
second_name	varchar(50)		Nein		
area_reference	int(5)		Nein	0	
linear_reference	int(5)		Nein	0	
negative_offset	int(5)		Nein	0	
positive_offset	int(5)		Nein	0	
urban	tinyint(1)		Ja	*NULL*	
intersectioncode	int(5)		Nein	0	
in_positive	tinyint(1)		Ja	*NULL*	
out_positive	tinyint(1)		Ja	*NULL*	
in_negative	tinyint(1)		Ja	*NULL*	
out_negative	tinyint(1)		Ja	*NULL*	
present_positive	tinyint(1)		Ja	*NULL*	
present_negative	tinyint(1)		Ja	*NULL*	
exit_number	varchar(5)		Nein		
diversion_positive	varchar(10)		Nein		
diversion_negative	varchar(10)		Nein		
veraendert	tinyint(1)		Ja	*NULL*	
tern	tinyint(1)		Ja	*NULL*	
netzknoten_nr	bigint(20)		Nein	0	
netzknoten2_nr	bigint(20)		Nein	0	
station	int(11)		Nein	0	
x_coord	bigint(20)		Nein	0	
y_coord	bigint(20)		Nein	0	
poldir	varchar(50)		Nein		
admin_county	varchar(5)		Nein		
actuality	varchar(20)		Nein		
activated	tinyint(1)		Ja	*NULL*	
tested	tinyint(1)		Ja	*NULL*	
special1	varchar(50)		Nein		
special2	varchar(50)		Nein		
special3	varchar(50)		Nein		
special4	varchar(50)		Nein		
special5	varchar(50)		Nein		
special6	varchar(50)		Nein		
special7	varchar(50)		Nein		
special8	varchar(50)		Nein		
special9	varchar(50)		Nein		

Abbildung 3.4: struktureller Aufbau der Tabelle "location"

Aufteilung der Sprachen ist sinnvoll, da es sich bei den Streckenabschnitten oft um Eigennamen handelt, wie Ortsausfahrten. So könnte der Nutzer die Event Codes in eine für ihn verständliche Sprache umsetzen und kann daran erkennen, an welchen Orten Probleme auftauchen.

Die Datenbankadministration wurde mit der grafischen Web-Oberfläche gelöst (siehe [2]). In dieser konnten neue Tabellen angelegt und für die jeweilige Tabelle die notwendigen Felder zugewiesen werden. Die unterstrichenen Felder in den Abbildungen 3.1 bis 3.4 bilden den Primärschlüssel. An der Spalte 'Typ' ist ersichtlich, was für ein Attribut gespeichert wird. Bis auf die Tabelle 'duration' werden alle Tabellen mit dem entwickelten Programm automatisch gefüllt. Dies wird in den nächsten Abschnitten genau erklärt.

3.2 Parsen der Code Listen

Beim Parsen der Listen geht es darum, automatisiert, die Listen für die Codes in die Datenbank zu speichern. Dies erfolgt in drei Schritten: Mit der Klasse 'CSVParser' wird der eigentliche Parser konstruiert. Dieser liest die gewünschten Werte aus den csv-Files aus. Im folgenden Code-Beispiel soll dies für die Event Liste gezeigt werden. Das Verfahren funktioniert bei der Location Liste nach dem selben Prinzip.

```
1
public void parseEventFile(String file) {
3       BufferedReader bfReader = null;
        try {
5           bfReader = new BufferedReader(new FileReader(file));
        } catch (FileNotFoundException fnfe) {
7           System.out.println("File not found: " +file);
            return;
9       }

11      String line;
        try {
13          bfReader.readLine();
            while ((line = bfReader.readLine()) != null) {
15              addEvent(line);
            }
17      } catch (IOException ioe) {
            ioe.printStackTrace();
19      }
    }
```

Diese Methode bekommt die csv-Datei als Parameter und der `bfReader` versucht die Werte auszulesen und in der Methode `addEvent` in ein String Array zu schreiben. In dieser Methode werden die einzelnen Felder der Tabelle nach und nach definiert. Hier für die Event Liste:

```
private void addEvent(String line) {
        line = cleanup(line);
        String[] elements = line.split(";");

        Event event = new Event();
        try {
            event.setLine(createInteger(elements[0]));
            event.setCen(elements[1]);
            event.setDe(elements[2]);
            event.setQez(elements[3]);
            event.setQmz(elements[4]);
            event.setCode(createInteger(elements[5]));
            event.setN(elements[6]);
            event.setQ(createInteger(elements[7]));
            event.setT(elements[8]);
            event.setD(createInteger(elements[9]));
            event.setU(elements[10]);
            event.setC(createInteger(elements[11]));
            event.setR(elements[12]);
        } catch (ArrayIndexOutOfBoundsException ex) {
            exceptions++;
        }
        getEvents().add(event);

    }
```

In der Klasse `Event` wird der SQL Befehl INSERT für den Import vorbereitet. Es werden zuerst die Spalten festgelegt und dann die entsprechenden Werte. Hier wird mit Hilfe des Stringbuffers eine SQL Anfrage der Felder und der Werte gebildet. Zu beachten ist, das hier die Sprache für Deutsch und dem technischen Englisch (CEN) getrennt wird, da diese Informationen in eine eigene Tabelle gespeichert werden. So werden drei Stringbuffer konstruiert: Einer für die Eventcodes mit deren Werten und jeweils einer für die entsprechende Übersetzung (DE bzw. CEN). 'Qez' und 'Qmz' sind teilweise verwendete Qualifier, die eine Aussage über z.B. die Länge des Staus machen. Auch diese liegen in verschiedenen Sprachen vor und werden dementsprechend bei den Stringbuffern hinzugefügt.

Nachdem die Daten vorbereitet und ausgelesen sind, müssen diese in die Datenbank geschrieben werden. Dies passiert mit der Klasse `DBInsert`. In dieser wird die Datenbank-Verbindung mit den bekannten Konstrukten hergestellt. Es folgt die Methode `insertEvent`:

```
public void insertEvent() {

        ArrayList events = parser.getEvents();
        for (Iterator eventIter = events.iterator();
```

```
5           eventIter.hasNext(); ) {

7               Event event = (Event) eventIter.next();

9               StringBuffer sb = new StringBuffer();
                sb.append("INSERT INTO events (");
11              sb.append(event.getSQLColumnsEvents() +") ");
                sb.append("VALUES (");
13              sb.append(event.getSQLValuesEvents() +")");

15              sql = sb.toString();
                System.out.println("Executing: " +sql);
17
                //In Datenbank "klopfen"
19              try {
                    Statement stat = conn.createStatement();
21                  stat.execute(sql);
                    stat.close();
23              } catch (SQLException sqle) {
                    sqle.printStackTrace();
25              }
            }
27  }
```

Der SQL Befehl für die Datenbank wird aus den einzelnen Teilen zusammengesetzt und im try-Konstrukt ausgeführt. Nach dem selben Verfahren wird die Tabelle `language` gefüllt. Auch die LCL wird nach dem beschriebenen Verfahren ausgelesen und in die Datenbank gespeichert.

Nachdem beide Listen in der Datenbank gespeichert sind, sind alle permanenten Daten gespeichert. Im nächsten Schritt werden die antreffenden TMC Nachrichten dauerhaft gespeichert. Bevor dies aber geschehen kann, muss die XML Ausgabe und teilweise die vorliegenden Klassen modifiziert werden, da an dieser Ausgabe angeknüpft wird.

3.3 Konfiguration der XML Ausgabe

Bei der bisherigen Ausgabe erschienen nicht alle notwendigen Daten im XML Strom. Zusätzlich wurden manche Informationen zur besseren Lesbarkeit weiterverarbeitet und können somit nicht in die Datenbank gespeichert werden. Dies geschah zum Beispiel bei der 'Duration' (Dauer eines Events) dessen eintreffender Code gleich in die entsprechende Zeit umgerechnet wurde. Somit ging die ursprüngliche Information der TMC Nachricht verloren. Daher sind zwei Schritte notwendig, um alle Informationen zu erhalten: Erstens müssen einige Klassen erweitert werden, damit die ursprünglichen Codes bis zur XML Ausgabe erhalten bleiben. Zusätzlich muss in der XML Ausgabe (siehe unter Punkt 2.2) ein weiterer Tag erscheinen, der die Ursprungsdaten (raw information) ausgibt.

Bei der Umwandlung der eintreffenden Signale (`ByteArray2RDSGroup`) musste nichts verändert werden, da hier alle Informationen erhalten waren. Beim nächsten Schritt in eine TMC Nachricht (`RDSGroup2TMCMessage`) gingen wichtige Informationen verloren. Daher sind in dieser Klasse Erweiterungen eingefügt worden und mit 'modified' kommentiert. Die Informationen werden im entsprechenden Block der TMC Nachricht gespeichert. Somit können im folgenden Schritt die Informationen (z.B. 'frequency-id', 'error-codes', 'duration-and-persistance') ausgelesen werden, die bisher unvollständig oder nicht vorhanden waren.

Im letzten Schritt zur Ausgabe des XML Stroms (`TMCMessage2XML`) wird der zusätzliche Tag `raw` eingerichtet, der nur die übertragenen Codes der Nachricht auflistet. Dazu folgt ein Codebeispiel aus der Klasse `TMCMessage2XML.java`:

```
1  private void writeRaw(TMCMessage message, TMCMessageMetaInfo meta)
   {
3      this.buffer.startElement(RAW_DATA);

5  //optional message content
     if (message.getRawData().getOmcs() != null) {
7     for(OptionalMessageContent omc : message.getRawData().getOmcs()) {
      this.buffer.startElement(OPTIONAL);
9
      this.buffer.addTextElement(LABEL, Integer.toString(omc.getLabel()));
11    this.buffer.addTextElement(VALUE, Integer.toString(omc.getValue()));

13    this.buffer.endElement();
                }
15          }
```

Hier werden mit einer get-Methode die zuvor gespeicherten Daten aus der Nachricht herausgelesen. Als Ausgabe erscheint jeweils das Label und der dazugehörige Wert. Am Ende der bisherigen Ausgabe taucht der zusätzlicher Tag `raw` auf, der diese Informationen folgendermaßen auflistet:

```
1  <raw>
              <picode>D311</picode>
3             <eventCode>108</eventCode>
              <locationCode>12981</locationCode>
5             <directionAndExtent>0</directionAndExtent>
              <timestamp>Wed Nov 26 09:58:15 CEST 2005</timestamp>
7             <frequency>104.60</frequency>
              <duration>0</duration>
9             <diversion>0</diversion>
   </raw>
```

Diese Funktion ist optional und kann in der `rawxml.config.xml` beim Starten des Servers eingeschalten werden (`printRaw=''true''`). An der bisherigen Konfigurations-

datei gibt es sonst keine Änderung. Hier ein Auszug aus der Datei `rawxml.config.xml`:

```
<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" printRaw="true">
     <drain class="generics.ConsoleDrain"/>
      <drain class="generics.StringSocketDrain" port="9778"/>
   </node>
 </node>
 </node>
</node>
```

Nach dieser Modifikation sind alle wichtigen Informationen zur Verarbeitung vorhanden. Im nächsten Punkt werden mit Hilfe der Ausgabe die Informationen in der Datenbank abgelegt.

3.4 Datenspeicherung durch Verwendung der XML Ausgabe

Zur Speicherung der Ereignisse in die Datenbank, wird ein weiterer 'Verbraucher' (`Drain`) benötigt, der die XML Ausgabe verwendet. Dies übernimmt die Klasse `DBDrain`, in der alle notwendigen Aufgaben koordiniert werden. Dabei werden mit einem XML Parser die notwendigen Informationen aus dem XML Strom und der Konfigurationsdatei heraus gelesen. Angaben zur Datenbank (Hostname, Portnummer, Benutzer etc.) befinden sich in der Konfigurationsdatei `databasedrain.config.xml`. Die ausgelesenen Werte werden im nächsten Schritt an den `MessageHandler` übergeben. Dieser kümmert sich um die Verbindung zur Datenbank (Starten und Beenden) und konstruiert mit Hilfe der Klasse `TMCMessage` die SQL Anweisung in der Methode `writeToDatabase(TMCMessage message)`:

```
 private void writeToDatabase(TMCMessage message) {
        LOGGER.info("Writing to database");
        try {
            Statement stmt = conn.createStatement();
            LOGGER.info(message.getMessageSQL());
            stmt.execute(message.getMessageSQL());

            ResultSet rs = stmt.getGeneratedKeys();
            rs.next();
            int key = rs.getInt(1);

            for (String optionalStatement :
            message.getOptionalContentMessages(key)) {
                LOGGER.info(optionalStatement);
                stmt.execute(optionalStatement);
            }
```

```
18              stmt.close();
            } catch (SQLException sqle) {
20              sqle.printStackTrace();
            }
22      }
```

Hier wird zuerst eine Datenbankverbindung hergestellt (`conn.createStatement()`) und im nächsten Schritt der konstruierte SQL Befehl ausgeführt. Zusätzlich gibt es eine Bildschirmausgabe, die auflistet, ob und welcher SQL Befehl verwendet wurde. Dazu ein kurzes Beispiel:

```
44957 parsing XML to database
2 45030 Writing to database
45033
4 INSERT INTO message (pi_code, eventcode, primaryloc, direction,
extent, diversion, timestamp, country, frequency_id, duration)
6 VALUES('D311', '704', '12962', '-1', '2', '0', '2005-11-26
10:28:08.0', 'de', '91.65', '-1')
8 45037 INSERT INTO optional
(message_nr, label, content) VALUES('57454', '9', '107')
10 45039
INSERT INTO optional (message_nr, label, content) VALUES('57454',
12 '1', '2')
```

In den Zeilen 8 - 12 sind zwei zusätzliche SQL Befehle zu sehen. Diese speichern die Werte für den optionalen Inhalt in die entsprechende Tabelle.

Wie oben schon erwähnt, muss die Konfigurationsdatei für den TMCServer erweitert werden, damit die `DBDrain` auch verwendet wird. Dabei muss ein zusätzlicher `node` in der Datei angegeben werden (aus `databasedrain.config.xml`):
`<drain class=''generics.DbDrain'' host=''localhost'' port=''3306''`
`database=''tmcstat'' user=''---'' password=''---'' country=''de''/>`
Mit dieser Erweiterung werden die ankommenden TMC Nachrichten in der Datenbank abgelegt.

3.5 Hinzunahme eines 2. Empfängers

Der bisherige Empfänger befindet sich am seriellen Port `ttyS0`. Der zweite Empfänger wird am selben Rechner am seriellen Port `ttyS14` angeschlossen. Die Konfigurationen ändern sich nur geringfügig, da nur in den entsprechenden Konfigurationsdateien (`server2.config.xml`) der Port geändert werden muss. Es handelt sich bei dem Empfänger um einen baugleichen Typ des ersten Gerätes und unterstützt somit dieselben Funktionen. Das Hauptproblem war der Empfang mit einer leistungsstarken Antenne. Dieses Gerät sollte einen anderen TMC-fähigen Radiosender empfangen, damit bei der

statistischen Auswertung ein möglicher Vergleich von eintreffenden Ereignissen aufgezeigt werden kann. Zu Beginn hat der Empfänger aber nur denselben Sender (Frequenz 104.60) ausgegeben. Nach einigen Konfigurationen mit einer Antenne konnte der Sender auf die Frequenz 101.30 eingestellt werden.

Somit stand dem parallelen Betrieb nichts mehr im Wege. Es gibt jeweils zwei Server die einen XML Strom, wie in Punkt 3.3 beschrieben, am Monitor ausgeben. Zusätzlich können über unterschiedliche Ports (anzugeben in den Konfigurationsdateien der Server) beide Empfänger in die Datenbank schreiben. Dabei kam es zu Problemen, die im Punkt 3.7 genauer beschrieben werden.

3.6 Verwendung der Datenbank als Quelle

Diese Option dient zur Nutzung einer XML Ausgabe, unabhängig von einem angeschlossenen Sender. Damit ein Ereignis in seinen ursprünglichen Zustand zurückgewandelt wird, müssen die Schritte vom Eintreffen der Nachricht bis zur XML Ausgabe rückwärts vollzogen werden.
Dies zeigt sich in der veränderten Konfigurationsdatei:

```
<server>
  <services>
    <network>
      <source class="generics.DatabaseSource" host="localhost"
      port="3306" database="tmcstat" user="---" password="---"
      country="de" pi_code="D75B"
      startdate="15:25:00 30.10.2005" enddate="15:26:00 30.10.2005">
        <node class="database.Database2StatTMCMessage">
          <node class="database.StatTMCMessage2RDSGroup8A">
            <node class="tmc.RDSGroup2TMCMessage"
             locationDB="LocationList_de.csv"
            eventDB="EventList_en_de.csv" lang="de_DE">
            <node class="xml.TMCMessage2Xml" prettyPrint="true"
             printRaw="true">
             <drain class="generics.ConsoleDrain"/>
            </node>
          </node>
        </node>
      </node>
     </source>
    </network>
  </services>
</server>
```

Zu Beginn wird unter `source` nicht mehr wie bisher die Klasse und der entsprechende Server angegeben, sondern die zu verwendende Datenbank. Als zusätzliche Option gibt es zwei Möglichkeiten, welche TMC Nachrichten später ausgegeben werden sollen. Einer-

seits kann ein Startdatum (`startdate`) und Enddatum (`enddate`) angegeben werden, bei dem überprüft wird, ob zu diesem Zeitpunkt Ereignisse in der Datenbank vorhanden sind. Liegen keine Ereignisse vor, wird mit einer entsprechenden Ausgabe abgebrochen.

Die andere Möglichkeit ist die Festlegung einer bestimmten Anzahl von Nachrichten, die ausgegeben werden sollen (`maxmessages`="beliebiger Integerwert"). Bei der Ausgabe erscheinen die entsprechenden Ereignisse, beginnend bei dem ersten, das in der Datenbank gespeichert ist.

Für die Umwandlung der Werte zur ursprünglichen Nachricht sind zwei Schritte notwendig. Zuerst erfolgt eine Speicherung der Daten in eine `StatTMCMessage` in der Klasse `Database2StatTMCMessage`. Dabei wird das zurückgegebene Resultset aus der Datenbank in die ursprünglichen Werte aufgeteilt und entsprechend gespeichert. Auch der optionale Inhalt, falls vorhanden, wird an das Ereignis wieder angebunden.

Im zweiten Schritt wird eine `RDSGroup8A` erzeugt, wie sie nach Auslesen des Bitstroms und der Umwandlung in Blöcke vorliegt. In der Klasse `StatTMCMessage2RDSGroup8A` übernimmt das die Methode `createMessage`:

```
1  private RDSGroup8A createMessage(TMCMessage data, String meta) {
2
3      RDSGroup8A group = new RDSGroup8A();
4
5      group.setAlertPlusOrTuningInfo(false);
6      group.setEventCode(data.getPrimaryEvent());
7      group.setLocationCode(data.getPrimaryLocation());
8      group.setDurationAndPersistence(data.getDuration());
9      group.setDirectionAndExtent(calculateDirectionAndExtent(data));
10     group.setSenderID(data.getPi_code());
11
12
13         if (addAdditionalContents(data, group)) {
14             LOGGER.info("Additional content found!");
15
16             group.setCompletelyLinked(true);
17             group.setFirstGroup(true);
18             group.setContinuityIndex(2);
19             group.setSingleGroupFullMessage(false);
20         }
21         else {
22             LOGGER.info("No additional content found!");
23             group.setFirstGroup(true);
24             group.setSingleGroupFullMessage(true);
25         }
26         return group;
27     }
```

Es erfolgt die Speicherung in die festgelegten Werte, wie es in den Blöcken (siehe frühere Abbildung 2.1) definiert ist.

Befindet sich das Ereignis in dieser ursprünglichen Form, kann wieder mit dem herkömmlichen Verbraucher darauf zugegriffen werden (`RDSGroup2TMCMessage`). Im weiteren Schritt erfolgt die XML Ausgabe, an der nichts geändert werden muss. Wenn gewünscht, könnte auch hier eine andere Datenbank als Verbraucher angebunden werden.

Zwei Punkte sind bei der Ausgabe zu beachten. Damit die Nachrichten bei der Ausgabe nicht den gespeicherten Zeitpunkt (`timestamp`) auflisten, sondern den Aktuellen, wird dieser bei der Ausgabe überschrieben. Dies simuliert einen Radiosender, der gerade erst die Nachricht verschickt hat.

Ein weiterer Punkt muss beim Auslesen aus der Datenbank beachtet werden. Die Ereignisse sollen nicht alle am Stück ausgegeben werden, sondern es muss die Zeitdifferenz zwischen zwei gespeicherten Ereignissen beachtet werden. Während dieser Zeit darf keine weitere Nachricht ausgelesen werden. So wird bei der Rückgabe des `resultset` der Zeitstempel der aktuellen und der folgenden Nachricht voneinander abgezogen. Der Ausgabeprozess wird mit der ermittelten Zeitspanne schlafen gelegt.

3.7 Zusammenfassung

Die in Kapitel 3 vorgestellten Lösungen ermöglichen eine dauerhafte Speicherung von TMC Daten. Aus diesen lassen sich längerfristige Prognosen und Informationen ermitteln. In Kapitel 4 werden zwei mögliche Auswertungen vorgestellt, die sich mit den zugrunde liegenden Daten berechnen lassen. Bevor die statistische Auswertung vorgestellt wird, werden zuerst die Hauptprobleme beim Ablegen in die Datenbank dargelegt.

Ein Problem warr der Server (zur Umwandlung der Binärsignale in TMC Nachrichten), der nicht immer kontinuierlich lief und so kam es einige Male vor, das der Serverprozess abgestürzt ist. Dies trat vor allem auf, wenn die beiden Server, die parallel in die Datenbank geschrieben haben, liefen. Ob die genaue Ursache am Rechner oder an der eingerichteten Datenbank liegt, kann nicht gesagt werden. Zeitweise blieb beim simultanen Schreiben einer der Schreibe-Prozesse hängen, aber nach einiger Zeit lief dieser ohne Fremdeinwirkung wieder weiter. Ob in der Zeit keine neuen TMC Nachrichten eingetroffen sind oder der Prozessor ausgelastet war, konnte nicht ermittelt werden. Auch war längeres Schreiben, bei nur einem Empfänger, in die Datenbank nicht möglich, da auch dieser nach ein paar Stunden bis zu ein paar Tagen automatisch stoppte. Vermutlich gibt es zwischen dem Empfänger und der Datenbank immer wieder Synchronisationsprobleme, sodass ein Prozess in einen Timeout läuft und hängen bleibt. Hier hat sich der Prozess nicht wieder regeneriert und weitergearbeitet.

4 Statistische Auswertung

4.1 Voraussetzungen

Für die statistische Auswertung ist ein grafischer Client mit direktem Zugriff auf die Datenbank erstellt worden. Als Programmiersprache wurde hierfür Java in der Version 1.5 verwendet. Die Verbindung zur Datenbank wurde per JDBC und dem MySQL Connector/J (Version 3.1) aufgebaut. Als Entwicklungsumgebung kam Eclipse in der Version 3.1 zum Einsatz. Für die Erstellung der grafischen Oberfläche wurde die komponentenorientiere Java-Swing-API eingesetzt. Diese ist fester Bestandteil der Java-Runtime-Enviroment (JRE) und daher ohne Installation zusätzlicher (plattformabhängiger) Bibliotheken einsetzbar. Diese Plattformunabhängigkeit und bestehende Integration in JAVA gab in der Planungsphase den Ausschlag zugunsten von Swing, trotz der gegenüber anderen, moderneren Grafikframeworks wie GTK oder SWT wesentlich schlechteren Performanz. Für die Darstellung von statistischen Grafiken und Charts wurde die freie Java-Bibliothek jFreeChart eingesetzt. jFreeChart zeichnet sich durch eine einfache API, vielseitige Einsatzmöglichkeiten (Torten-, Balken- und Linendiagramme etc.) und eine hohe optische Qualität (3D-Effekte, Zoom und animierte Diagramme) aus.

4.2 Erstellen des Frameworks

Das Framework zur statistischen Auswertung der TMC-Nachrichten ist nach dem Model-View-Controller-Pattern (MVC) aufgebaut. Es enthält demnach die Packages
`controller` `model`, `view`, sowie `testcases`, wo Testcase-spezifische Klassen (eine Klasse pro Testcase) liegen. Das Package `view` besteht im Wesentlichen nur aus der Klasse MainFrame. Es handelt sich hierbei um das Hauptfenster des statistischen Auswertungsprogramm, dem so genannten Evaluator. Das Fenster ist in zwei horizontale JPanels gegliedert. Im linken JPanel werden die Namen aller verfügbarer Testcases in einer scrollbaren Liste angezeigt. Wählt man einen Testcase durch einfach Mausclick aus, so erscheint im rechten JPanel eine Vorschau für den ausgewählten Testcase. Dort können für den Testcase relevante Einstellungen vorgenommen werden (z.B. Start- und Endzeitpunkt der Auswertung). Mit einem Mausclick auf den Button `Auswerten` wird die statistische Auswertung mit den gegebenen Einstellungen durchgeführt. Das Package `model` enthält alle Klassen die für die interne Programstruktur notwendig sind. Die Klasse `Event` entspricht der JAVA-Bean-Konvention (getter/setter-Methoden für Instanzvariablen und parameterloser Konstruktor) und enthält Eventcode, Identifier und Anzahl der so genannten Events (TMC-Meldungen). Ein Event-Bean für Stau könnte z.B. einen Eventcode=101 und Identifier="Stau wegen Unfall" haben. Mehrere solcher Events werden in so ge-

nannten `EventGroups` zusammengefasst. Eine `EventGroup` beinhaltet eine Liste der in ihr gespeicherten Events. Darüber hinaus enthält jede EventGroup alle Eventcodes der Nachrichten, für die die jeweilige Gruppe zuständig ist. Diese Eventcodes können sowohl einzelne diskrete Werte sein (10, 15, 380) als auch mehrere Eventcode-Bereiche (z.B. 10-15, 300-302). Durch einen Identifier können die EventGroups benannt werden. Wichtige EventGroups sind beispielsweise die EventGroup mit Identifier 'Meldung aufgehoben', welche 37 unzusammenhängende Eventcodes beinhaltet, oder die EventGroup mit Identifier 'Stau', welche sowohl einzelne Eventcodes als auch mehrere Eventcode-Bereiche umfasst. Wichtige `EventGroups` können bereits vor dem Programmstart definiert werden. Andere `EventGroups` werden zur Laufzeit des Programms dynamisch erzeugt. Die Klasse `EventGrouper` übernimmt die Einteilung der einzelnen Events zu EventGroups. Ein EventGrouper speichert eine Liste von `EventGroups` und erstellt bei Bedarf dynamisch neue `EventGroups`, die nur jeweils eine Kapselung eines einzelnen Events darstellen. Wenn von einem Testcase TMC-Nachrichten evaluiert werden müssen, so benutzt dieser Testcase zumeist den EventGrouper. Der Testcase speichert die Nachrichten die er aus der Datenbank liest mit Hilfe des EventGrouper ab und kann aus diesem später die erzeugten EventGroup-Objekte zur weiteren Evaluierung abrufen. Zuletzt enthält das `model`-package die abstrakte Klasse Testcase, wovon alle Testcases erben. Diese Klasse enthält die Grundfunktionaliäten die jeder Testcase beinhalten muss (z.B. die Erzeugung des Testcase-Listenelements). Im Package `controller` sind alle Klassen zu finden, die für den Programmablauf wichtig sind. Die Klasse Controller enthält zunächst die main-Methode, welcher der Einstiegspunkt in das Evaluator-Programm ist. Bei Programmstart wird hier die Konfigurationsdatei gelesen und entsprechend umgesetzt. Darüber hinaus kümmert sich diese Klasse um das Zusammenspiel der einzelnen Komponenten (beispielsweise zwischen dem `MainFrame` und der Datenbank).

4.3 Konfigurieren des Frameworks

Wie bereits erwähnt ist das Evaluator-Programm (teilweise) konfigurier- bzw. erweiterbar. Beim Start des Programms wird zunächst die Konfigurations-Datei config.xml ausgewertet. Dazu kommt die freie JAVA-Bibliothek jConfig in der Version 2.8.5 zum Einsatz. jConfig bietet eine einfache API und kann (im Gegensatz zu herkömmlichen properties) auch hierarchische Beziehungen (z.B. mit Kategorien) abbilden. Konfiguriert werden können die Verbindungsdaten zur Datenbank, die Namen und entsprechenden Klassen der einzelnen Testcases und die EventGroups, zu denen einzelne Events zusammengefaßt werden können. Zur Konfiguration der Testcases wird zu Beginn des Programms folgende Methode ausgeführt:

```
1 private void buildTestcases() {
    for (String testCase : configuration.getCategoryNames()) {
3       Category category = configuration.getCategory(testCase);
    try {
5       if (!category.getCategoryName().equals("general") &&
        !category.getCategoryName().startsWith("group") &&
```

```
7           !category.getCategoryName().equals("connection")) {
            Testcase newCase =(Testcase)
9           Class.forName(category.getProperty("class")).newInstance();
            newCase.setTitle(category.getProperty("title"));
11          newCase.setController(this);

13          testcases.add(newCase);
               }
15            } catch (Exception e) {
                e.printStackTrace()     }
17      }
    }
```

Zunächst werden alle Kategorien aus der Konfigurations-Datei ausgelesen. Danach wird jede Kategorie, solange sie nicht `general` oder `connection` heißt oder mit `group` beginnt, als Testcase verarbeitet:

```
<category name="Testcase1">
2     <property name="class" value="testcases.Testcase1"/>
      <property name="title" value="Testcase1"/>
4  </category>
```

In der Konfigurations-Datei besteht jeder Testcase aus genau zwei Properties, dem Title, der dem Testcase seinen Namen gibt, und dem class-Property, was aussagt welche Klasse den Testcase repräsentiert. Durch den Aufruf `Class.forName(category.getProperty(''class'')).newInstance();` wird ein neues Object als Testcase instanziiert und kann der Liste der Testcases im Controller hinzugefügt werden. Tritt hierbei ein Fehler auf, beispielsweise weil die Konfigurationsdatei fehlerhaft oder die referenzierte Klasse nicht im classpath zu finden ist, wird der Stack-Trace der geworfenen Exception ausgegeben. Analog zu der Konfiguration der Testcases erfolgt die Initialisierung der EventGroups. Hierzu wird bei Programmstart folgende Methode aufgerufen:

```
private void configureGrouper() {
2     for (String testCase : configuration.getCategoryNames()) {
          Category category = configuration.getCategory(testCase);
4         try {
            if (category.getCategoryName().startsWith("group")) {
6             String identifier = category.getProperty("identifier");
              String range = category.getProperty("range");
8             String single = category.getProperty("single");
              int[] singleInts = null;
10            if (single != null) {
                 String[] singleStrings = single.split(",");
12               singleInts = new int[singleStrings.length];
                 for (int i = 0; i < singleStrings.length; i++) {
14                   singleInts[i] = Integer.parseInt(singleStrings[i]);
```

```
                    }
16                }

18      this.eventGrouper.addEventGroup(identifier, range, singleInts);
                }
20          } catch (Exception e) {
                e.printStackTrace();
22          }
        }
24  }
```

Dabei werden alle Kategorien die mit `group` beginnen ausgelesen und jeweils in eine EventGroup umgewandelt. Dazu werden drei Bestandteile aus der Konfigurationsdatei ausgelesen, `identifier`, der die EventGroup benamt, `range`, eine Verkettung von Event-Bereichen und `single`, eine Verkettung einzelner Events.

```
<category name="group3">
2     <property name="identifier" value="Stau"/>
      <property name="range" value="101-106,108-113,115-120,131-134"/>
4     <property name="single" value="1"/>
</category>
```

Auf bei der Erstellung der EventGroups kann ein Fehler in der Konfigurationsdatei zu einer Exception führen, die abgefangen und ausgegeben wird.

4.4 Datenbank-Verbindung

Wie bereits erwähnt ist die Datenbankverbindung ebenfalls konfigurierbar und erfolgt per Java Database Connectivity (JDBC). Um den Aufbau/Abbau der Verbindung kümmert sich die Klasse `DatabaseConnector` aus dem Package `controller`. Der entsprechende Abschnitt in der Konfigurationdatei sieht folgendermassen aus:

```
1 <category name="connection">
      <property name="host" value="localhost"/>
3     <property name="port" value="3306"/>
      <property name="database" value="tmcstat"/>
5     <property name="user" value="tmcstat"/>
      <property name="pw" value="secret"/>
7 </category>
```

Es müssen alle relevanten Daten für den Aufbau der Datenbankverbindung angegeben werden, darunter auch Username und Passwort im Klartext. Im gegenwärtigen Zustand des Programms können nur MySQL-Datenbanken angesprochen werden, durch geringfügige Erweiterung und Installation eines passenden JDBC-Treibers könnten aber auch andere Datenbanken (z.B. ORACLE) angesprochen werden. Die Klasse Database-Connector verarbeitet auch alle möglichen SQL-Statements die von anderen Programm-

teilen via Controller abgegeben werden können. Aus Perfomanzgründen wird z.B. die Abfrage nach dem Aufhebungscode für bestimme Events als PreparedStatement angelegt:

```
cancelStmt = connection.prepareStatement("SELECT cancelcode FROM
cancelmessages WHERE eventcode=?"); public int getCancelCode(int
eventcode) {
    try {
        cancelStmt.setInt(1, eventcode);
        ResultSet rs = cancelStmt.executeQuery();
        if (rs.next()) {
            return rs.getInt("cancelcode");
        }
    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
    return -1;
}
```

4.5 Testcases

Das modellierte Evaluator-Framework ist um weitere Testcases erweiterbar. Dafür muss lediglich eine Testcase-Klasse erstellt werden, die von der abstrakten Klasse `Testcase` erbt, und ein Eintrag in die Konfigurationsdatei gemacht werden. Durch die Vererbung von Testcase haben die neu erstellten Testcases automatisch über den Controller Zugriff auf die Datenbank und auf die EventGroups. Im Folgenden werden zwei Testcases exemplarisch erläutert:

4.5.1 Welches Ereignis tritt am Häufigsten auf?

Dieser statistische Testcase evaluiert, welche Events (im angegebenen Zeitraum) am Häufigsten auftreten und stellt das Ergebnis in einem Tortendiagramm dar. Als Parameter zur Einschränkung der Suche kann - durch die Eingabe von Start- und Endzeitpunkt - ein Zeitfenster definiert werden, innerhalb dessen die Events stattgefunden haben müssen. Diese Auswahloptionen sind in Abbildung 4.1 zu sehen.

Die SQL-Abfrage für den beschriebenen Testcase sieht folgendermassen aus:

```
StringBuffer sql = new StringBuffer(); sql.append("SELECT COUNT(*)
AS anzahl, eventcode, text FROM message JOIN language ON
eventcode=code WHERE language.country='de' ");  if
(startDate!=null) {
    sql.append("AND message.timestamp>'" + startDate + "' ");
} if (endDate != null) {
    sql.append("AND message.timestamp<'" + endDate + "' ");
} sql.append("GROUP BY eventcode ORDER BY anzahl DESC");
```

Abbildung 4.1: Screenshot des Vorschau-Fensters für Testcase1

Durch diese Abfrage (ausgeführt von der Klasse `DatabaseConnector` via der Klasse `Controller`) erhält man eine Tabelle mit Anzahl, dem Eventcode und der lokalisierte textuelle Beschreibung nach der Anzahl absteigend sortiert. Sofern Start- oder Endzeitpunkt angegeben wurden, werden diese als Kriterien in die Suche aufgenommen. Die Ergebnistabelle der Anfrage muss in ein für JFreeChart (vgl. [1]) passendes Datenmodell überführt werden, in diesem Fall repräsentiert durch die Klasse `org.jfree.data.general.DefaultPieDataset`. Bevor dies geschieht müssen zunächst die einzelnen Events zu (den in der Konfigurationsdatei vordefinierten) `EventGroups` zusammengefasst werden. Dazu wird das gesamte `ResultSet` dem `EventGrouper` übergeben, der zeilenweise die Events zu EventGroups umwandelt. Später werden die fünf EventGroups mit der größten Event-Anzahl herausgesucht und in das oben bereits erwähnte `DefaultPieDataSet` umgewandelt (siehe Abbildung 4.2). Eine sinnvolle Auswertung ist hier nur mit vordefinierten EventGroups möglich. So haben z.B. Staus je nach Länge auch unterschiedliche Eventcodes. Für die statistische Auswertung ist die Länge aber meist relativ unwichtig, so daß alle Staus unterschiedlicher Länge zu einer `EventGroup` "Stau" zusammengefasst werden.

4.5.2 Brutto- /Nettoeventanzahl der Nachrichten

Der zweite exemplarisch vorgestellte und implementierte Testcase wertet den zeitlichen Verlauf des Verhältnisses von Brutto- zu Nettonachrichten aus. Um auch neu hinzugekommene Empfänger ständig auf dem Laufenden zu halten, ist es notwendig daß die TMC-Nachrichten nicht nur einmal bei Auftreten des Ereignisses gesendet werden, sondern danach periodisch wiederholt werden.Ursprünglich waren die Nachrichten so konzipiert, daß sie einen Timestamp und eine Gültigkeitsdauer erhalten, nach derer sie automatisch verfallen. Während des Praktikums stellte sich jedoch heraus, daß kaum ein Sender die Nachrichten mit einer Gültigkeitsdauer versieht. Stattdessen senden die Sender so genannte Aufhebungs-Nachrichten. Sobald ein bestimmtes Ereignis an einer

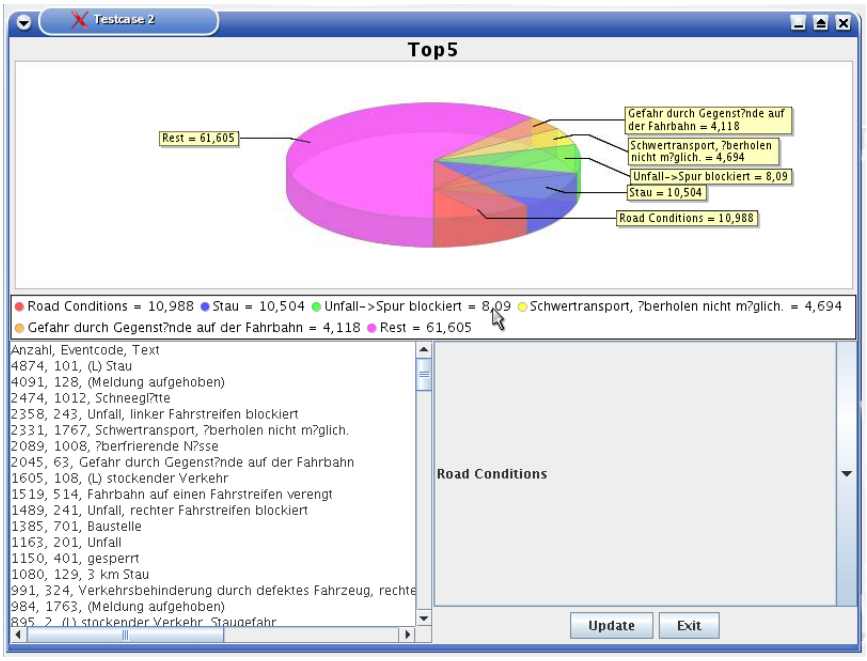

Abbildung 4.2: Screenshot des Ergebnisses für Testcase 1 - Pie Chart

bestimmten Position gemeldet wurde bleibt es bestehen, bis es explizit durch eine dieser Aufhebungs-Nachrichten invalidiert wird. Dieser Testcase wertet also das Verhältnis von neuen Nachrichten (netto) zu allen Nachrichten (brutto), die nur wiederholt werden, aus. Auch dieser Testcase wird mit einem Zeitfenster parametrisiert (siehe Abbildung 4.3).

Für die Auswertung werden alle Events in der Datenbank im angegebenen Zeitfenster aufsteigend geordnet nach Eingangsdatum ausgewertet. Über jedes dieser Events wird iteriert. Für jedes Event muss entschieden werden ob es eine Netto- oder eine Brutto-Nachricht ist. Dazu wird ausgehend vom aktuellen Event in der ausgegebenen Eventliste zurückgegangen. Wird dieselbe Nachricht noch einmal gefunden, so wird das Ereignis als Brutto-Event eingestuft, weil man davon ausgehen kann, das das vorliegende Ereignis nur eine Wiederholung ist. Wird eine Aufhebungs-Nachricht für das dieses entdeckt, wird die Nachricht als Netto-Nachricht eingestuft, weil das gemeldete Ereignis offensichtlich nach der Aufhebung neu gemeldet wurde. Sobald eines der beiden o.g. Kriterien eintritt ist das Event klassifiziert und das nächste Event wird untersucht. Sollte auch mit allen vorherigen Nachrichten der Eventliste keine Entscheidung zu treffen sein, so wird die Nachricht als Netto-Nachricht eingestuft, weil nicht nachzuweisen ist, dass die Nachricht nur eine Wiederholung darstellt. Nachdem alle Nachrichten des eingegebenen Zeitfensters fertig ausgewertet sind wird das Ergebnis in ein für JFreeChart (vgl. [1]) passendes Model überführt, in diesem Fall repräsentiert durch die Klasse org.jfree.data.category.CategoryDataset. Dieses wird als Flächengraph, wie in Abbildung 4.4 zu sehen, ausgegeben.

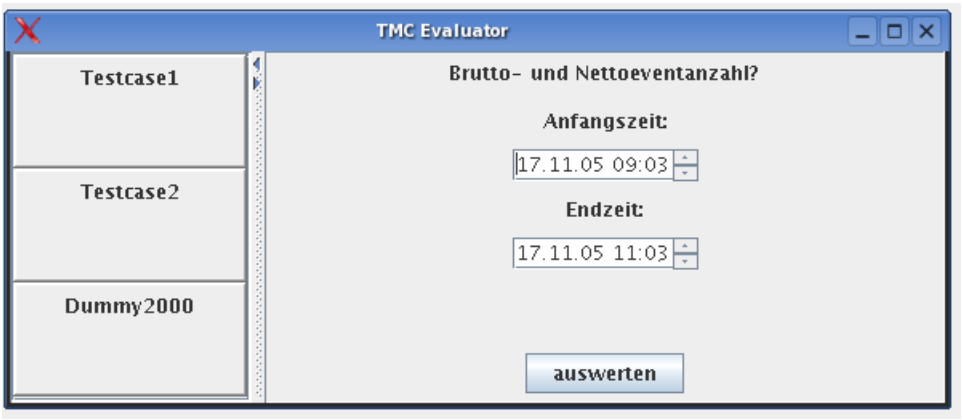

Abbildung 4.3: Screenshot des Vorschau-Fenster für Testcase2

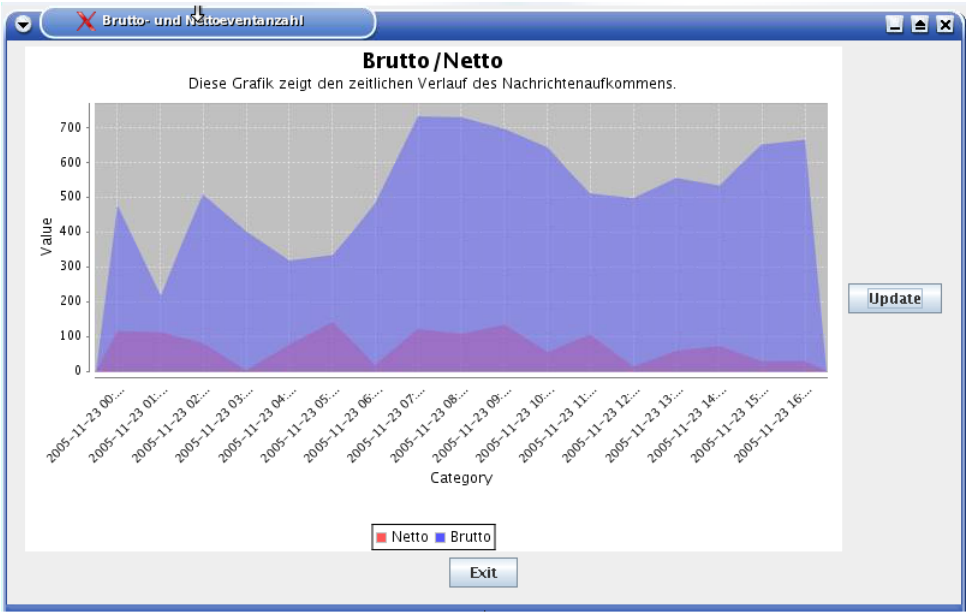

Abbildung 4.4: Screenshot des Ergebnisses für Testcase 2 - Verlaufsdiagramm

5 Ausblick

Mit diesem Projekt ist eine dauerhafte Speicherung vom Ereignissen und dessen Auswertung möglich. In Kapitel 3 wird das Ablegen der Daten genau erklärt und welche Probleme auftraten. In Kapitel 4 werden zwei Möglichkeiten vorgestellt, welche Ergebnisse daraus gezogen werden können. Dies ist nur ein kleiner Teil der aus den permanenten Daten gewonnen werden kann. Folgende Fragen und Probleme konnten innerhalb der Projektzeit nicht beantwortet werden:

- **Paralleler Betrieb mehrerer Empfänger:** Im parallelen Betrieb tauchen weitere Fragestellungen auf. Nach welchem Schema ein Sender Nachrichten ausstrahlt und mit welcher Verzögerung die Nachrichten empfangen werden könnte starken Unterschieden unterliegen. Beispielsweise ist auch die Reihenfolge eintreffender Nachrichten zu untersuchen, insbesondere das zeitversetzte Eintreffen identischer Nachrichten und Aufhebungsnachrichten. Idealerweise sollten alle redundanten Nachrichten verworfen werden. Im Datenbank- bzw. Archivierungsbetrieb würden andererseits sämtliche Nachrichten benötigt, ebenso für die statistische Auswertung. Auch ist zu untersuchen welche statistischen Auswertungen bei Empfang von Nachrichten mehrerer Sender sinnvoll erscheinen.

- **Untersuchung von Zusatzinformationen:** Jeder Sender hat die Möglichkeit zusätzliche Informationen zu liefern. Dies kann z.B. eine Ausweichroute bei Stau oder Hinweise auf Geschwindigkeitsbegrenzungen sein. Welcher Radiosender diese Möglichkeit nutzt und seinen Hörern einen Mehrwert durch Informationen gibt, wäre ein weiterer Punkt den es zu untersuchen gilt.

- **Prognosen über Staubildung:** Ein weiterer interessanter Punkt ist die Speicherung der Daten über einen längeren Zeitraum, um daraus Verkehrsprognosen zu ziehen. Möglicherweise lassen sich aus den Daten regionale Brennpunkte ermitteln, die den Autofahrer vor diesen warnen. Besonders ortsunkundige Fahrer könnten z.B. frühzeitig über staugefährdete Straßen und Autobahnkreuze beim Feierabendverkehr gewarnt werden.

- **Lösung technischer Probleme:** Hierbei sind zwei Punkte zu behandeln. Erstens gibt es Probleme beim parallelen Speichern von zwei Servern in die Datenbank. Zweitens ist die grafische Darstellung der Daten sehr langsam. Gerade das erste Problem muss gelöst werden, da sonst keine Datenspeicherung über einen längeren

Zeitpunkt erfolgen kann. Das Problem war nicht reproduzierbar und daher kam es innerhalb der Projektzeit zu keiner Lösung. Eine Beschleunigung der grafischen Darstellung wäre hilfreich, um schneller die gewünschten Auswertungen darzustellen.

- **Ausbau und Erweiterung des Frameworks:** Die zusätzlich überlegten Auswertungen können in das erstellte Framework eingebunden werden. Um zusätzliche Auswertungen einfach einzubinden, wurde ein System entwickelt, das leicht neue `Testcases` anbindet (siehe Absatz 4.5). Mit Hilfe des JFreeChart Paketes (vgl. [1]) gibt es unzählige Möglichkeiten die Daten zu repräsentieren. Damit können unterschiedlichste Überlegungen auf verschiedenste Weise repräsentiert werden. Mit Hilfe von Java kann das Auswahlfenster wie gewünscht umstrukturiert werden.

6 Abbildungsverzeichnis

2.1 RDSGroup8A ([4], Seite 27) 8
2.2 Ausschnitt der Event List 9
2.3 Ausschnitt der Location List 10
2.4 Die komplette TTI Umgebung: bisherige und zukünftige Dienste 11
2.5 Kompletter Datenfluss und Umwandlung des gesamten Projektes 15

3.1 struktureller Aufbau der Tabelle "message" 18
3.2 struktureller Aufbau der Tabelle "optional" 18
3.3 struktureller Aufbau der Tabelle "events" 19
3.4 struktureller Aufbau der Tabelle "location" 20

4.1 Screenshot des Vorschau-Fensters für Testcase1 36
4.2 Screenshot des Ergebnisses für Testcase 1 - Pie Chart 37
4.3 Screenshot des Vorschau-Fenster für Testcase2 38
4.4 Screenshot des Ergebnisses für Testcase 2 - Verlaufsdiagramm 38

7 Literaturverzeichnis

[1] *JFreeChart*. http://www.jfree.org/jfreechart/, June 2006.

[2] *The phpMyAdmin Project*. http://www.phpmyadmin.net, June 2006.

[3] Bundesanstalt für Straßenwesen. www.bast.de, 2006.

[4] 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.

[5] 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.

[6] 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.

[7] Hans Jürgen Ohlbach and Bernhard Lorenz. Dynamic Data for Geospatial Reasoning - A Local Data Stream Management System (L-DSMS) and a Case Study with RDS-TMC. Deliverable A1-D6, Institute for Informatics, Ludwig-Maximilians-Universität München, 2006.

[8] Wikipedia. *RDS - Radio Data System*. http://de.wikipedia.org/wiki/Radio_Data_System, 2006.