HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Extraction and Analysis of State Information in Peer-to-Peer Networks

Bachelor Thesis

by

Kolja Salewski

born in
Essen

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich Heine University Düsseldorf

October 2014

Supervisors:
Jun.-Prof. Dr.-Ing. Kalman Graffi
Prof. Dr. Stefan Conrad

Abstract

This bachelor thesis is about extending the peer-to-peer (p2p) simulator *PeerfactSim.KOM* [Pee14] with a database connection in order to extract detailed information concerning created or removed nodes and connections as well as sent and received messages. The measured data should be used to visualize the particular overlay network or parts of it in a user-configurable way in order to expand the evaluation possibilities. Although *PeerfactSim.KOM* already offers visualization features, its capabilities in this area are limited as it is mainly a simulator. Examinations outside the application can be performed by plotting generated data files but as the latter predominantly contain statistic values it is hardly possible to retrieve information relating to a certain node or time window. Due to the proprietary file format of *PeerfactSim.KOM* recordings there is no way of visualizing them without the actual program.

I focused on the problem of subsequent analysis of executed simulations. With the published version of the simulator it is not possible to query data about specific nodes, connections or messages of visualizations created in the past. Therefore one of the main aims of my thesis was to find a user-friendly and deeply configurable way of analyzing run simulations. I achieved this goal by automatically collecting all necessary information in an adequately structured database and offering a tool to convert the measured data to compatible inputs for already existing visualization software. Thereby it is now possible to textually and visually evaluate simulations executed in the past. As at least one of the visualization tools used offers various non-proprietary output formats users are from now on able to open and change a graphical overlay representation with several applications.

Acknowledgments

First of all I would like to express my deep gratitude to Jun.-Prof. Dr.-Ing. Kalman Graffi and Prof. Dr. Stefan Conrad, who have made it possible for me to write this thesis.

Furthermore special thanks go to my advisor Tobias Amft, whose invaluable ideas, suggestions and weekly meetings helped me a lot. This bachelor thesis would not exist without him.

Advice given by Marlis Hombergs has been a great help. Her perfectionism and ambition were very much appreciated.

Last but not least I wish to thank my whole family and all of my friends for their inexhaustible regard and patience during the last months.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The motivation and structure of this bachelor thesis are introduced below. Section 1.1 presents the former and Section 1.2 the latter. Necessary background information is given, the current situation as well as the targeted one are described.

1.1 Motivation

Peer-to-peer networks are at least partially decentralized architectures and usually consist of equally privileged nodes. Therefore typically any instance taking part in the former offers but also uses services (like file sharing, voice over IP, instant messaging or computational capacity). This is the main difference to client-server architectures, in which one or more servers provide resources as a central unit and clients request them. One of the advantages of peer-to-peer networks over conventional client-server architectures is the scalability due to the growth of resources like unused computational power, free storage capacity or available bandwidth with each additional peer. As the shared contents often exist multiple times, the data is kept in a redundant form. The biggest challenge of the so-called overlays which realize different types of peer-to-peer networks is the self-organization in order to manage the join and leave process of participants but also to assign data, tasks and resources to them. The use of simulators for peer-to-peer networks and distributed systems enables the user to evaluate different overlay architectures and protocols. By visualizing the generated events it is possible to get an overview and keep track of changes without manually analyzing a huge amount of logged actions. Although the simulator *PeerfactSim.KOM* already offers a lot of possibilities a user could miss the option to select single events or small time windows which are part of very large simulations. A possibility of configuring the optical appearance or of using different layouts positioning the nodes and their connections based on various algorithms would also be an advantage. With the help of the latter it could be possible to recognize specific structures such as rings formed by the particular overlay without adding fixed position information to each node.

1.2 Structure

Part of this bachelor thesis consisted of finding a reasonable database structure to automatically collect all information needed for further analysis and visualization in the background while generating a new simulation run. The main challenge was to implement a tool to visualize the measured data with the help of existing programs. For this purpose *db2vis* (database to visualization) was designed and developed, a software converting the database entries to visualization commands used by tools *Graphviz* [Gra14] and *UbiGraph* [Ubi14b].

In Chapter 2 the software used for simulation, data storage and visualization is introduced.

Chapter 3 describes the database and software design. The generated database tables and their columns are characterized before presenting the way *db2vis* is designed to be configured, to log and visualize.

Afterwards the implementation is described in Chapter 4. This comprises a documentation of the extensions made in *PeerfactSim.KOM*. The different parts of *db2vis* such as configuration, logging and visualization are the main topics.

Chapter 5 evaluates the features and limitations of *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*. The visualization results of the three applications are compared before discussing the challenge of appropriate visualization. Furthermore the scalability of *PeerfactSim.KOM*, *db2vis* and *Graphviz* referring to overlay size is analyzed.

Finally Chapter 6 summarizes the results, discusses open questions and provides ideas for possible future work on *db2vis* and its database input.

Chapter 2

Related Work

In this chapter the software used for simulation (Section 2.1), data storage (Section 2.2) and visualization (Section 2.3) is presented. The aims and features of each program as well as the particular version used during this thesis are stated.

2.1 Simulation

2.1.1 *PeerfactSim.KOM*

PeerfactSim.KOM [Gra11] is an open source software for simulating potentially large distributed/peer-to-peer networks. It has been developed at the *Technical University (TU) Darmstadt* [Tec14] and further extended as well as maintained by the *University of Paderborn (UPB)* [Uni14] and the *Heinrich Heine University Düsseldorf (HHU)* [Hei14]. The simulator contains various layers, of which the peer-to-peer overlay layer covers structured, unstructured and information dissemination p2p overlays. Thanks to the filters offered inside the visualization window of *PeerfactSim.KOM* the user is able to show just the information he or she is interested in. Different metrics can be shown in order to compare properties of the nodes and connections. As all events follow a time line it is possible to move forward and backward inside the whole specified simulation time.
The relevant visualization events consist of added/changed/removed nodes, added/removed edges and sent messages. Each of these six events is defined in its own class. With each event taking place an according object is created by calling the particular constructor. The latter expects a visual overlay node respectively edge as argument in case of an added/removed node or edge. To change a node the attribute object and its changed attributes need to be given as constructor parameters. The constructor of a message sent event demands the message type as well as the visual overlay nodes sending respectively receiving the message.

Each visual overlay node object contains information about its attributes such as net ID, overlay ID, location, topological/schematic position, overlay and document counter. Visual overlay edge objects at least consist of the two visual overlay nodes connected and optionally of additional attributes like message type, payload et cetera.
All statements about *PeerfactSim.KOM* refer to the *Community Edition* released on 2013/10/17.

2.2 Data Storage

2.2.1 *MySQL*

MySQL [Ora14e] is a relational database management system (RDBMS) originally engineered by *MySQL AB* [Saf14] (acquired by *Sun Microsystems* [Ora14f], in turn acquired by *Oracle Corporation* [Ora14a]). The community edition used for this thesis is a freely downloadable version of the open source database. Databases and their tables can be created and accessed via several connectors like *Java Database Connectivity* (*JDBC*) [Ora14b], an application programming interface (API) for *Java* [Ora14c]. The *JDBC* driver for *MySQL* is *MySQL Connector/J* [Ora14d].
MySQL Server 5.6.21.1 and *MySQL Connector/J* 5.1.33 are the latest versions used in this thesis.

2.3 Visualization

This section introduces the visualization software used, *Graphviz* in Subsection 2.3.1 and *UbiGraph* in Subsection 2.3.2. Inputs and outputs are shown for both tools. The former are desired to be automatically generated by *db2vis* for any given simulation database created by *PeerfactSim.KOM*.

2.3.1 *Graphviz*

Graphviz [EGK$^+$02] (short for "Graph Visualization Software") is an open source program for visualizing objects as well as their relations and has been initiated by *AT&T Labs Research* [Lab14]. It offers several layouts, each using a different algorithm for positioning the elements to be shown. Inputs are taken by commands in a graph description language and can be passed in the form of a text file given as parameter. Outputs can be textual or graphical, while various file formats are supported. *Graphviz* produces static two-dimensional graphs and provides numerous attributes to customize visualization. In this bachelor thesis version 2.38 of the stable release for *Windows* [Mic14] and *Ubuntu* [Can14] 13.10 x64 was used.

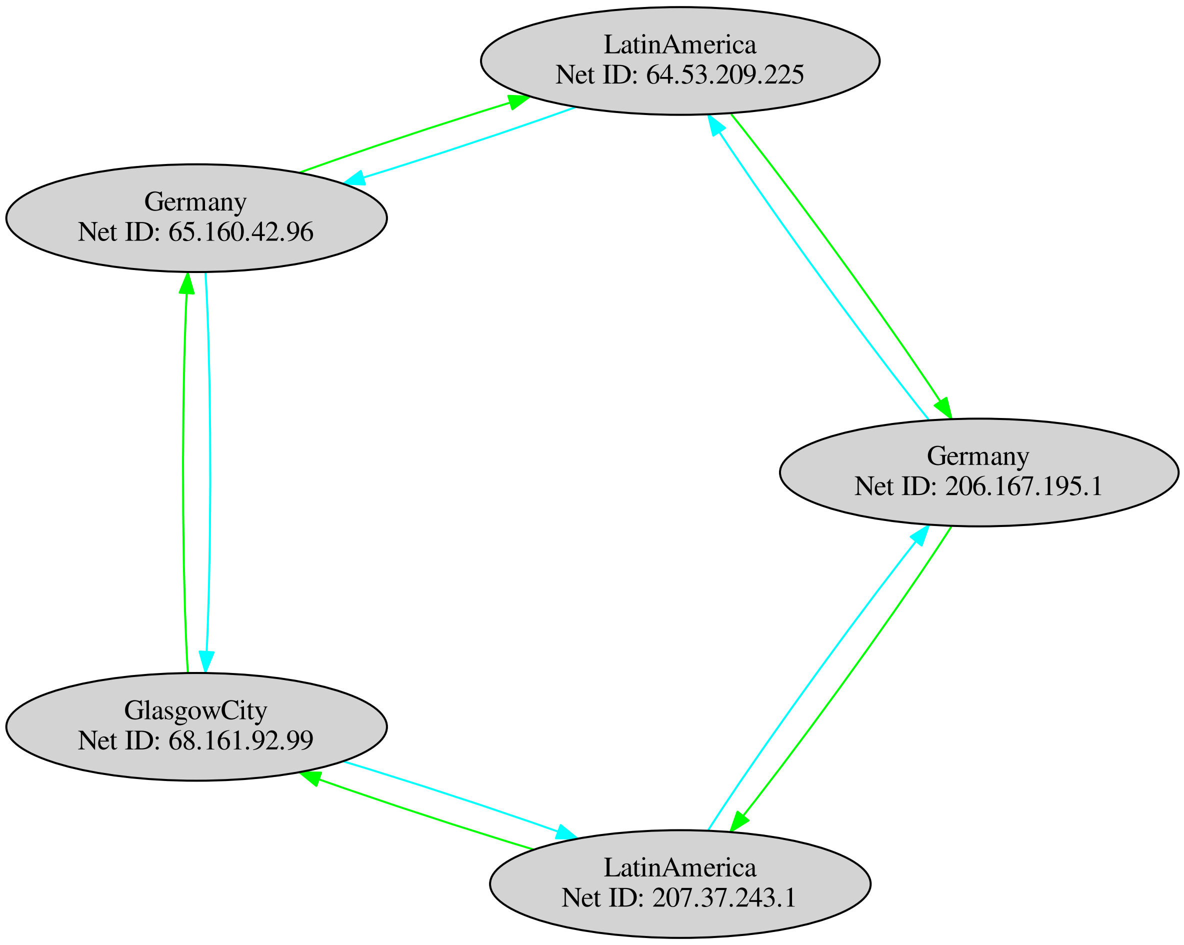
In Listing 2.1 one can find the content of a *gv* file that contains the textual *Graphviz* representation of a so-called *Chord Ring* with five nodes and ten edges. The hexadecimal RGB triple *00ffff* represents the color cyan, *00ff00* stands for green. In this example the former is used for so-called *predecessor* edges and the latter for *successor* ones. Subsubsection *Graphviz* of Subsection 3.2.3 provides additional information about *gv* files.

Listing 2.1: Example of a *gv* File

```
// <graph type> <graph name> {
digraph G {
// graph attributes
// <attribute> = <value>
outputorder = edgesfirst

// node attributes
// node [<attribute> = <value>]
node [style = filled]

// nodes
// <node> [<attribute> = <value>]
"65.160.42.96"  [label = "Germany\nNet ID: 65.160.42.96"        ]
"68.161.92.99"  [label = "GlasgowCity\nNet ID: 68.161.92.99"    ]
"207.37.243.1"  [label = "LatinAmerica\nNet ID: 207.37.243.1"   ]
"206.167.195.1" [label = "Germany\nNet ID: 206.167.195.1"       ]
"64.53.209.225" [label = "LatinAmerica\nNet ID: 64.53.209.225"  ]

// edges
// <from node> <edge type> <to node> [<attribute> = <value>]
"65.160.42.96"  -> "68.161.92.99"     [color = "#00ffff"]
"207.37.243.1"  -> "68.161.92.99"     [color = "#00ff00"]
"68.161.92.99"  -> "207.37.243.1"     [color = "#00ffff"]
"206.167.195.1" -> "207.37.243.1"     [color = "#00ff00"]
"207.37.243.1"  -> "206.167.195.1"    [color = "#00ffff"]
"64.53.209.225" -> "206.167.195.1"    [color = "#00ff00"]
"206.167.195.1" -> "64.53.209.225"    [color = "#00ffff"]
"65.160.42.96"  -> "64.53.209.225"    [color = "#00ff00"]
"64.53.209.225" -> "65.160.42.96"     [color = "#00ffff"]
"68.161.92.99"  -> "65.160.42.96"     [color = "#00ff00"]
}
```

As one can see the graph and therefore also the edges are meant to be directed, all node shapes are to be filled and each one is supposed to be labeled with its location and net ID. Rendering this *gv* file as PDF with the circular *Graphviz* layout *circo* produces the vector graphic shown in Figure 2.1.

Figure 2.1: *Chord Ring* Visualized by *Graphviz* Layout *circo*



2.3.2 *UbiGraph*

UbiGraph [Vel07] is a free application for visualizing dynamic graphs and is offered by *Ubiety Lab, Inc.* [Ubi14a]. It talks to various programming languages, which enables direct communication. Established standards, libraries and APIs are used for cross-language, rendering and parallelism. Various vertex and edge attributes are supported. *UbiGraph* draws interactive three-dimensional graphs.
The version used for this thesis carries the number 0.2.4 and was developed for *Ubuntu* 8.04 x64.

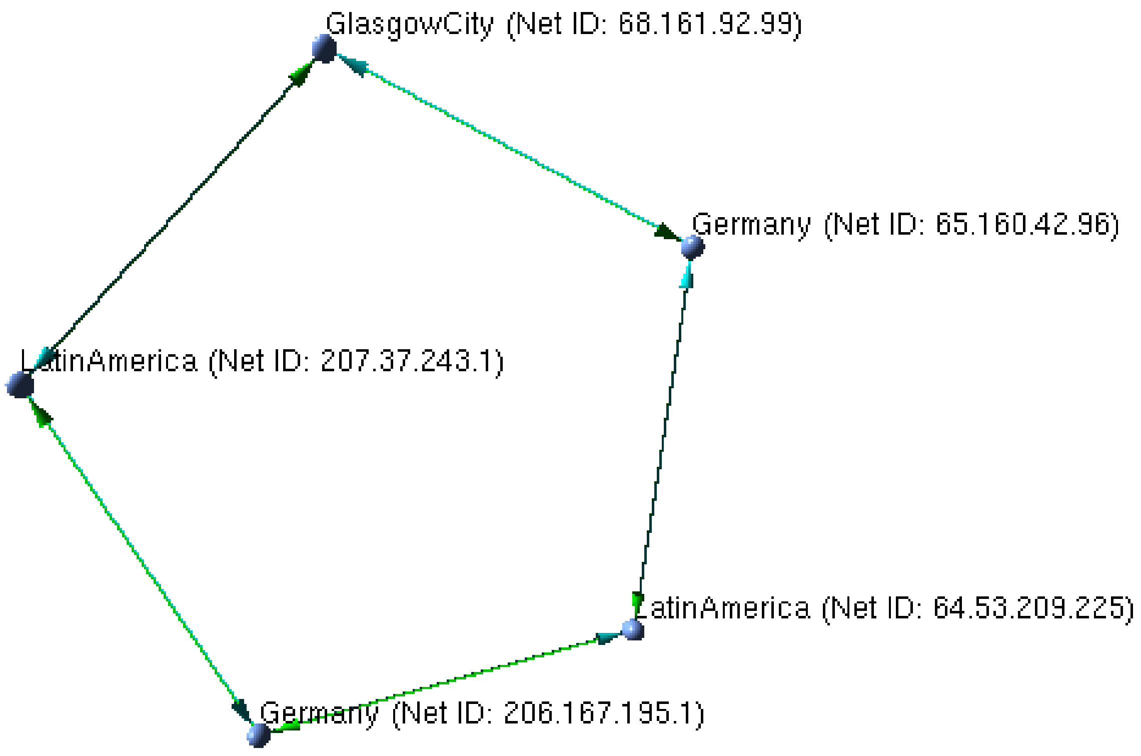
In Listing 2.2 one can find the *UbiGraph* commands in *Java* [Gos00], which represent exactly the same *Chord Ring* as listed in Listing 2.1 and as shown in Figure 2.1.

Listing 2.2: Example of a *UbiGraph* Command Set in *Java*

```
// create graph
UbigraphClient graph = new UbigraphClient();

// set default vertex style attributes
// graph.setVertexStyleAttribute(<style ID>, <attribute>, <value>);
graph.setVertexStyleAttribute(0, "shape", "sphere"     );
graph.setVertexStyleAttribute(0, "size" , "0.1"        );

// create edge styles and set their attributes
// graph.newEdgeStyle(<style ID>, <passed style ID>);
// graph.setEdgeStyleAttribute(<style ID>, <attribute>, <value>);
// predecessor edge
graph.newEdgeStyle(1, 0);
graph.setEdgeStyleAttribute(1, "color"          , "#00ffff"    );
graph.setEdgeStyleAttribute(1, "arrow_length"   , "0.5"        );
graph.setEdgeStyleAttribute(1, "arrow"          , "true"       );
graph.setEdgeStyleAttribute(1, "arrow_position" , "1.0"        );
graph.setEdgeStyleAttribute(1, "arrow_radius"   , "0.5"        );
// successor edge
graph.newEdgeStyle(2, 0);
graph.setEdgeStyleAttribute(2, "color"          , "#00ff00"    );
graph.setEdgeStyleAttribute(2, "arrow_length"   , "0.5"        );
graph.setEdgeStyleAttribute(2, "arrow"          , "true"       );
graph.setEdgeStyleAttribute(2, "arrow_position" , "1.0"        );
graph.setEdgeStyleAttribute(2, "arrow_radius"   , "0.5"        );

// create vertexes and set their attributes
// graph.newVertex(<vertex ID>);
// graph.setVertexAttribute(<vertex ID>, <attribute>, <value>);
// node 65.160.42.96
graph.newVertex(10);
graph.setVertexAttribute(10, "label", "Germany (Net ID: 65.160.42.96)");
// node 68.161.92.99
graph.newVertex(20);
graph.setVertexAttribute(20, "label", "GlasgowCity (Net ID: 68.161.92.99)
    ");
// node 207.37.243.1
graph.newVertex(30);
graph.setVertexAttribute(30, "label", "LatinAmerica (Net ID:
    207.37.243.1)");
// node 206.167.195.1
graph.newVertex(40);
graph.setVertexAttribute(40, "label", "Germany (Net ID: 206.167.195.1)");
// node 64.53.209.225
graph.newVertex(50);
graph.setVertexAttribute(50, "label", "LatinAmerica (Net ID:
    64.53.209.225)");

// create edges and change their style
// graph.newEdge(<edge ID>, <from vertex ID>, <to vertex ID>);
// graph.changeEdgeStyle(<edge ID>, <style ID>);
// predecessor edge from 65.160.42.96 to 68.161.92.99
graph.newEdge(100, 10, 20);
graph.changeEdgeStyle(100, 1);
// successor edge from 207.37.243.1 to 68.161.92.99
graph.newEdge(200, 30, 20);
graph.changeEdgeStyle(200, 2);
// predecessor edge from 68.161.92.99 to 207.37.243.1
graph.newEdge(300, 20, 30);
graph.changeEdgeStyle(300, 1);
// successor edge from 206.167.195.1 to 207.37.243.1
graph.newEdge(400, 40, 30);
graph.changeEdgeStyle(400, 2);
// predecessor edge from 207.37.243.1 to 206.167.195.1
graph.newEdge(500, 30, 40);
graph.changeEdgeStyle(500, 1);
// successor edge from 64.53.209.225 to 206.167.195.1
graph.newEdge(600, 50, 40);
graph.changeEdgeStyle(600, 2);
// predecessor edge from 206.167.195.1 to 64.53.209.225
graph.newEdge(700, 40, 50);
graph.changeEdgeStyle(700, 1);
// successor edge from 65.160.42.96 to 64.53.209.225
graph.newEdge(800, 10, 50);
graph.changeEdgeStyle(800, 2);
// predecessor edge from 64.53.209.225 to 65.160.42.96
graph.newEdge(900, 50, 10);
graph.changeEdgeStyle(900, 1);
// successor edge from 68.161.92.99 to 65.160.42.96
graph.newEdge(1000, 20, 10);
graph.changeEdgeStyle(1000, 2);
```

Executing these *UbiGraph* commands in *Java* produces the graph shown in Figure 2.2.

Figure 2.2: *Chord Ring* Visualized by *UbiGraph*


GlasgowCity (Net ID: 68.161.92.99)
Germany (Net ID: 65.160.42.96)
LatinAmerica (Net ID: 207.37.243.1)
LatinAmerica (Net ID: 64.53.209.225)
Germany (Net ID: 206.167.195.1)

Chapter 3

Design

This chapter gives an overview of the design decisions made. Just as a reminder: The intention of extending *PeerfactSim.KOM* with a database connection is to store state information collected during simulation. Hereby a textual examination both manually and automatically is proposed to be possible. Consequently the purpose of *db2vis* is to use the measured data for visualizing overlays in a configurable way with the help of the visualization tools *Graphviz* and *UbiGraph*.
In Section 3.1 the database naming convention and table structures are described, Section 3.2 details the design of the software configuration, logging and visualization. The criteria of each decision were usability, performance and compatibility. As there were often several alternatives, the reasons for the one chosen are presented.

3.1 Database

The database design including its naming convention and table structures is presented in Subsections 3.1.1 and 3.1.2, respectively. In each simulation run executed by *PeerfactSim.KOM* a new database on the locally hosted *MySQL* [WA02] server is created to separate the measured data between the different simulations.

3.1.1 Name

In order to achieve a usability as intuitive as possible the database name is similar to the output folder name generated by *PeerfactSim.KOM* (<timestamp>__<overlay>__seed_<seed_no.>). The only difference is the symbol '_' instead of '-' inside the timestamp due to *MySQL* database naming conventions, as can be seen in Table 3.1.

Table 3.1: Example of Database/Folder Naming Convention

Name	**Used by**
2014-09-18_15-51-36__chord__seed_5342083169705674899	*PeerfactSim.KOM*
2014_09_18_15_51_36__chord__seed_5342083169705674899	Database

3.1.2 Tables

Inside the database three tables are created named *node*, *edge* and *message*. This separation reduces table size and redundant data. Each table is designed to contain as much data as necessary but as little as possible to gain all relevant information while minimizing the performance impact. As a result table *node* does not contain any information like the number of neighbors or a counter for sent and received messages as these factors can already be retrieved by appropriate queries of the other two tables. As another consequence of this database design tables *edge* and *message* do not save any other node data than the affected net IDs. The allowed length of the particular field value is designed to contain exactly the necessary information - not more, not less. Fields containing net IDs for example allow up to 15 characters: Three dots plus at most twelve numerals. The type of field considers the desired content in order to force correct input and improve storage efficiency as well as usefulness for queries. A good example is the *time* field, which expects inputs compatible with *MySQL* data type *time(3)* in order to prevent not or falsely interpretable time formats and to be able to sort the table chronologically ascending or descending, to compare different time values et cetera.
Although *MySQL* is a relational DBMS as already mentioned in Subsection 2.2.1, its relational features do not have to be used as they are sometimes neither necessary nor suitable. One of these cases is the database connection between *PeerfactSim.KOM* and *db2vis*. As the database is supposed to not only contain the latest overlay state but also all previous ones in order to be able to visualize the overlay process and not only its final condition, all three tables potentially contain more than one entry representing the same node, edge or message. Changes as well as removals of nodes and edges must not be logged by simply updating or deleting the particular table entries but by inserting additional ones. Even messages which of course cannot be changed or removed once they are sent have to cause a new table entry each time they are sent - even if sender, receiver and type are identical as not only the latest occurrence is supposed to be visualized but also each previous one. In consequence none of the tables contains any primary or foreign key as actually no field holds exclusively unique values but potentially also duplicates. Due to the insertion of the particular simulation time in each table entry the latter can be distinguished without any difficulty even if all other columns are identical. This is for example the case if the same node or edge is added, then removed and finally added again. Due to the fact that entries are inserted by *PeerfactSim.KOM* during the simulation run they are automatically ordered by simulation time.

Below one can find the structure of all three database tables. Column **Field** of each description table contains the field names, **Type** the particular *MySQL* data type, **Null** specifies if it is allowed to insert a new entry in that database table without giving the particular field any value or the value (not string) *null* which stands for nothing. At last column **Description** explains the intention of that field.

Database Table *node*

Table 3.2 describes database table *node*. Fields *overlayId* and *doc_count* allow *null* as some overlay implementations within *PeerfactSim.KOM* do not provide information about these node attributes.

Table 3.2: Description of Database Table *node*

Field	**Type**	**Null**	**Description**
time	time(3)	No	Current simulation time (<hhh>:<mm>:<ss>.<fff>)
netId	varchar(15)	No	Net ID (text, max. 15 characters)
overlayId	varchar(19)	Yes	Overlay ID (text, max. 19 characters)
location	varchar(12)	No	Location (text, max. 12 characters)
topological_pos	varchar(22)	No	Topological position (text, max. 22 characters)
schematic_pos	varchar(25)	No	Schematic position (text, max. 25 characters)
overlay	varchar(10)	No	Used overlays (text, max. 10 characters)
overlay_raw	varchar(23)	No	Overlays, class names (text, max. 23 characters)
doc_count	int(11)	Yes	Number of documents (integer, max. 11 numerals)
action	varchar(7)	No	Action executed (*added*/*changed*/*removed* expected)

Table 3.3 represents a node with its attributes that was added after ten seconds of simulation.

Table 3.3: Example of Database Table *node*

time	***netId***	***overlayId***	***location***	***topological_pos***
00:00:10.000	64.70.178.2	97e9b..26134	Germany	(1296.3049,1318.0181)
schematic_pos	***overlay***	***overlay_raw***	***doc_count***	***action***
(0.27848208,0.16693869)	[Chord]	[ChordNode]	0	added

Database Table *edge*

Table 3.4 describes database table *edge*.

Table 3.4: Description of Database Table *edge*

Field	**Type**	**Null**	**Description**
time	time(3)	No	Current simulation time (<hhh>:<mm>:<ss>.<fff>)
nodeA	varchar(15)	No	Net ID of node A (text, max. 15 characters)
nodeB	varchar(15)	No	Net ID of node B (text, max. 15 characters)
type	varchar(22)	No	Type (text, max. 22 characters)
action	varchar(7)	No	Action executed (*added*/*removed* expected)

In Table 3.5 one can find an example of a so-called *successor* edge between the nodes 64.70.178.2 and 209.180.6.41 that was added at 1 m 467 ms.

Table 3.5: Example of Database Table *edge*

time	***nodeA***	***nodeB***	***type***	***action***
00:01:00.467	64.70.178.2	209.180.6.41	successor	added

Database Table *message*

Table 3.6 describes database table *message*.

Table 3.6: Description of Database Table *message*

Field	**Type**	**Null**	**Description**
time	time(3)	No	Current simulation time (<hhh>:<mm>:<ss>.<fff>)
nodeA	varchar(15)	No	Net ID of node A (text, max. 15 characters)
nodeB	varchar(15)	No	Net ID of node B (text, max. 15 characters)
type	varchar(24)	No	Type (text, max. 24 characters)

An example of a so-called *retrieve successor* message sent by node 64.70.178.2 to node 209.180.6.41 at 1 m 530 ms can be found in Table 3.7.

Table 3.7: Example of Database Table *message*

time	***nodeA***	***nodeB***	***type***
00:01:00.530	64.70.178.2	209.180.6.41	RetrieveSuccessorMsg

3.2 *db2vis* - Database to Visualization

In this section the design of the *db2vis* configuration (Subsection 3.2.1), logging (Subsection 3.2.2) and visualization (Subsection 3.2.3) is detailed.
db2vis is written in *Java* as is *PeerfactSim.KOM* for three main reasons:

1. Harmonize the way of database access between the two applications
2. Make extensions which are necessary in both programs as simple as possible
3. Be able to talk to *UbiGraph*

db2vis uses three classes, out of which class *Db2Vis* contains the main method. The other ones are *Node* and *GvLayout* which represent an overlay node and a *Graphviz* layout, respectively, each containing specific attributes. This way an unlimited number of according objects can be created as well as differentiated.
To successfully run the program the user has to specify the complete path to the configuration file (including filename) to be used:

run.<script file extension> <user configuration path>

Example:
run.sh config/chord.cfg

As a consequence of this users are able to create multiple configuration files and switch between them when executing *db2vis*.

3.2.1 Configuration

The program first reads the user configuration file, checks its syntax and saves its attributes and values. The configuration is designed hierarchically in order to simplify recognizing which part the particular setting belongs to. The file offers three main sections, namely *general*, *graphviz* and *ubigraph*. These sections start with a headline consisting of *###* and the particular section name:

### <section>

Example:
### general

All subsections until the next section start with ## and a convenient title:

## <title>

Example: *## connection colors & names*

If additional information is needed to understand the expected syntax of the particular attribute or if an example is provided there is at least one subsubsection line beginning with # followed by an explanation or example:

# <explanation/example>

Example:
# format: hex rgb triple, blank, comma- & blank-separated names

The actual inputs given by the user are expected in the line(s) below the respective (sub)subsection. If an attribute as well as at least one value should be given, the typed line has to have the following syntax:

<attribute> <value1>, <value2>, ...

Example:
00ff00 successor

In case no attribute is estimated the value(s) is/are expected to be (as above) simply typed one after another separated by a comma and a blank:

<value1>, <value2>, ...

Example:
successor, predecessor

The examples provided above together with some additional lines could look like Listing 3.1 in the user configuration file.

Listing 3.1: Example Part of *db2vis* User Configuration File

```
### general
## connection colors & names
# format: hex rgb triple, blank, comma- & blank-separated names
# example (without #):
# 0a1b2c connection1, connection2
00ff00 successor
00ffff predecessor
c0c0c0 finger
```

As the user has the possibility to configure if a log file should be created and if *Graphviz*, *UbiGraph* or both tools should be used for visualization, these elemental functionalities are presented next.

3.2.2 Logging

In case of activated logging a new folder *outputs* is created in the directory from which *db2vis* is run. Furthermore *db2vis* creates a subfolder *logging* within the output folder. In this logging folder another subdirectory is created and named like the output folder name of *PeerfactSim.KOM* (see Table 3.1):

outputs/logging/<timestamp>__<overlay>__seed_<seed_no.>/

Example:
outputs/logging/2014-09-18_15-51-36__chord__seed_0/

A new log file whose name consists of the current system timestamp followed by *.log* is created within the folder of the particular simulation each time it is visualized:

<system timestamp>.log

Example:
2014-09-19_13-03-21.235.log

Each line of the log file consists of the current system timestamp followed by the performed action:

<system timestamp> <action>

Example:
2014-09-19 13:03:24.330 connected to database 2014_09_18_15_51_36__chord__seed_0

In case anything went wrong suitable warning and error messages are written into the log file as well as the system console. They begin with *warning:* or *error:*, respectively, and are highlighted by an empty line above and below. While warnings do not force the program to exit, errors do.

Example:
2014-09-21 15:55:50.626 warning: could not remove UbiGraph predecessor edge from 69.21.148.41 to 69.21.148.41 at simulation time 000:01:00.930 because self-edges are not yet allowed by UbiGraph.

As all events are logged at the time of occurrence the log lines are chronologically ordered. At the end of the log users are informed if *db2vis* terminated successfully to detect an unexpected termination.

Example:
2014-09-21 15:56:34.920 finished visualization

3.2.3 Visualization

If both visualization tools are specified to be used, the program begins with *Graphviz* and then executes the *UbiGraph* commands. Independently of the tool used *db2vis* then starts to read all three database tables in chronological order by reading line after line of each table and comparing their time fields. The earliest event is visualized first. Each time *db2vis* reads an added node in the database, a new object of class *Node* is created.

Graphviz

The *Graphviz* part of the program creates a subfolder named *outputs* in the directory from which *db2vis* is run if it does not already exist due to activated logging. In this output folder another subdirectory is created and named like the output folder name of *PeerfactSim.KOM* (see Table 3.1). Finally a last folder is created named by the particular system timestamp. Thus the output folder name does not change if the same simulation is visualized multiple times, whereas its subfolder is only used once because of the new system time. So the whole path to the output files has the following structure:

outputs/<timestamp>__<overlay>__seed_<seed_no.>/<system timestamp>/

Example:
outputs/2014-09-18_15-51-36__chord__seed_0/2014-09-19_13-03-21.235/

Each user-configured layout produces its own outputs by creating files in the following format:

<simulation time>_<extension>_<layout>.gv

<simulation time> is the particular time of the database entry read at the moment of file creation. *<extension>* is used to be able to differentiate between more than one event at the same simulation time if the user configured *db2vis* to create more than one output file. *<layout>* contains the particular user-specified layout name.

Example:
000-02-56.294_0_dot.gv

Each *gv* file starts with either *graph G {* for an undirected graph or *digraph G {* for a directed one depending on the layout chosen by the user. Configured graph attributes are listed next and follow this principle:

<attribute> = <value>

Example:
outputorder = edgesfirst

Below that the configured node attributes are presented in the following way:

node [<attribute1> = <value1>, <attribute2> = <value2>, ...]

Example:
node [style = filled, shape = circle]

Then the actual edges between nodes follow. An edge always consists of two nodes called nodeA and nodeB as well as the type of edge, namely – for undirected edges or -> for directed ones. Additionally multiple edge attributes can be given according to the configuration. A whole edge line is structured as follows:

"<nodeA>" <edgeType> "<nodeB>" [<attribute1> = <value1>, <attribute2> = <value2>, ...];

Example:
"209.180.6.41" -> "64.70.178.2" [color = "#0000ff", style = "dashed", label = "JoinReply"];

After all edge lines the node-specific labels with user-activated metrics are shown:

"<node>" [label = "<metric1>\n<metric2>\n ..."];

Example:
"210.208.37.62" [label = "Germany\nNet ID: 210.208.37.62"];

The terminate symbol *\n* is interpreted as a new line by *Graphviz*. Therefore each metric is shown in its own line within the node shape.

The *gv* files can be rendered by the particular *Graphviz* layout (see Listing 2.1 respectively Figure 2.1).

Users are able to let *db2vis* create scripts (batch file for *Windows* and/or shell script for *Unix* [The14]) and also to let them be executed automatically to generate the user-specified output formats. The scripts are named as follows:

render_layouts.<script file extension>

<script file extension> is either *bat* (batch file for *Windows*) or *sh* (shell script for *Unix*). A created shell script starts with the lines of Listing 3.2.

Listing 3.2: Begin of Shell Script for *Unix*

```
#!/bin/bash

echo "Please wait until the shell script was executed. Warnings and
    errors are displayed in case anything went wrong. The bash prompt is
     shown when the execution has finished. Closing this window before
    stops the script."

set -v # echo on
```

The first line is the so-called *shebang* and specifies which shell should be used to interpret the script - in this case the *bash* [Fre14] shell. Then a user information follows and the echo is activated to show all following commands in a terminal in order to enable the user to track the script process. Below that the *Graphviz* commands are listed:

<layout> -T<output format> <input file> -o <output file>

Example:
dot -Tpng 000-02-56.294_0_dot.gv -o 000-02-56.294_0_dot.png

At the end of the file another user information is provided as shown in Listing 3.3.

Listing 3.3: End of Shell Script for *Unix*

```
echo "Please check above if the shell script was executed successfully.
    Warnings and errors are displayed in case anything went wrong."
```

The batch file for *Windows* begins similarly to the shell script for *Unix* as one can see in Listing 3.4.

Listing 3.4: Begin of Batch File for *Windows*

```
@echo. & @echo "Please wait until the batch file was executed. Warnings
    and errors are displayed in case anything went wrong. The cmd
   prompt is shown when the execution has finished. Closing this window
    before stops the script."
```

After using the same *Graphviz* commands as in the shell script for *Unix* the batch file for *Windows* also ends similarly as can be seen from Listing 3.5.

Listing 3.5: End of Batch File for *Windows*

```
@echo. & @echo "Please check above if the batch file was executed
   successfully. Warnings and errors are displayed in case anything
   went wrong."
```

Generated output files are named similarly to the *gv* files:

<simulation time>_<extension>_<layout>.<output format>

Example:
000-02-56.294_0_dot.png

UbiGraph

In contrast to the *Graphviz* functionality no files are created by *UbiGraph*. Instead a *UbiGraph* server has to be started and at least one client should connect to it. This client is shown as a window containing the current graph. If the user did not start the server himself *db2vis* does so. Afterwards a new client is established and connected to the running server. Then *db2vis* creates a new graph and inserts all nodes and edges during runtime. The graph changes dynamically so that one can keep track of events as they happen.

Chapter 4

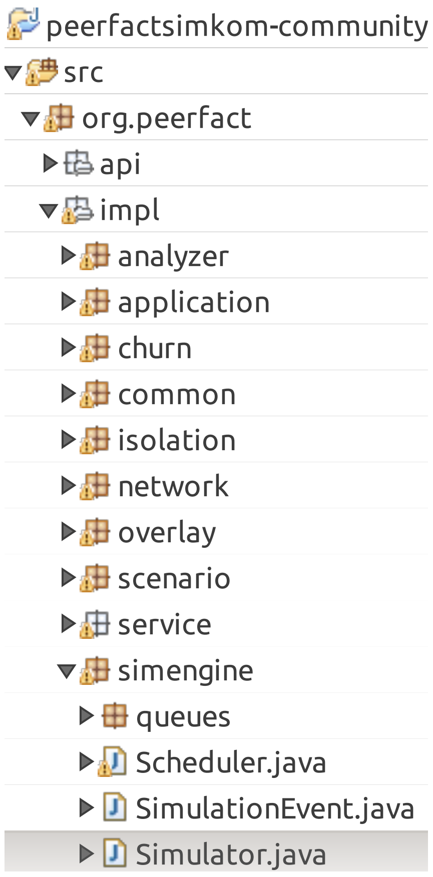
Implementation

The implementation had to be done in two parts - extending *PeerfactSim.KOM* as stated in Section 4.1 on the one hand and developing *db2vis* as described in Section 4.2 on the other hand. The database is the interface between both: *PeerfactSim.KOM* creates it and its tables *node*, *edge* and *message* but also inserts the according entries, *db2vis* reads and visualizes the data.

4.1 *PeerfactSim.KOM*

The source files of *PeerfactSim.KOM* are structured hierarchically in the form of various packages as one can see in Figure 4.1.

Figure 4.1: Hierarchical Position of *Java* Class *Simulator*



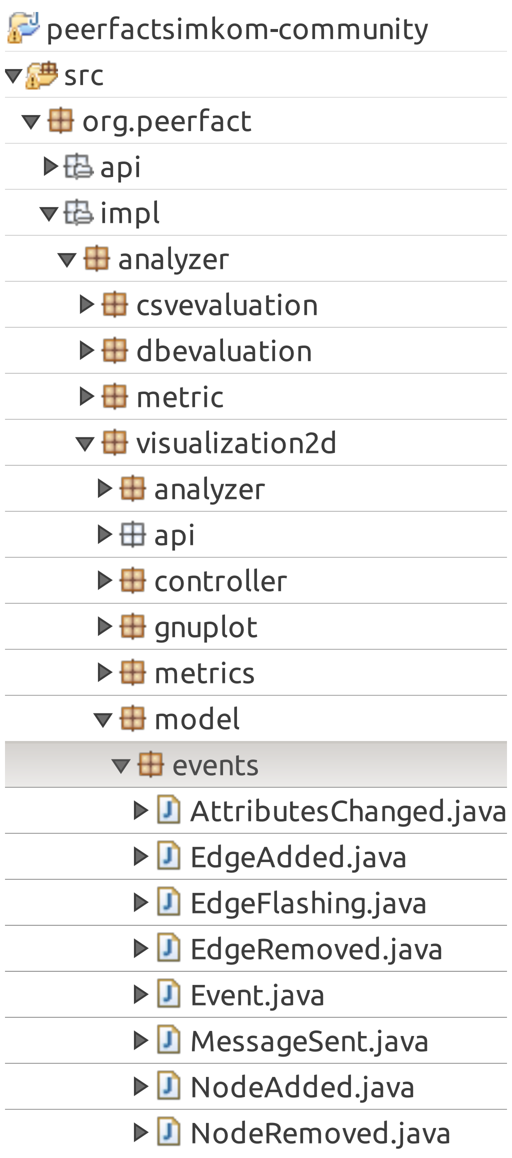
Class *Simulator* loads the *JDBC* driver and connects with its help to the locally hosted *MySQL* database server. Then it creates the database and the three tables *node*, *edge* and *message*. Listing 4.1 exemplarily shows the creation of database table *edge* according to Table 3.4.

Listing 4.1: Creation of Database Table *edge*

```
stmt.executeUpdate("CREATE TABLE edge ("
        + "time TIME(3) NOT NULL, "
        + "nodeA VARCHAR(15) NOT NULL, "
        + "nodeB VARCHAR(15) NOT NULL, "
        + "type VARCHAR(22) NOT NULL, "
        + "action VARCHAR(7) NOT NULL)");
```

The six classes used for inserting data are by name *NodeAdded*, *AttributesChanged*, *NodeRemoved*, *EdgeAdded*, *EdgeRemoved* and *MessageSent*. They are part of package *events* as can be seen from Figure 4.2.

Figure 4.2: Hierarchical Position of *Java* Package *events*



```
peerfactsimkom-community
▼ src
  ▼ org.peerfact
    ▶ api
    ▼ impl
      ▼ analyzer
        ▶ csvevaluation
        ▶ dbevaluation
        ▶ metric
        ▼ visualization2d
          ▶ analyzer
          ▶ api
          ▶ controller
          ▶ gnuplot
          ▶ metrics
          ▼ model
            ▼ events
              ▶ AttributesChanged.java
              ▶ EdgeAdded.java
              ▶ EdgeFlashing.java
              ▶ EdgeRemoved.java
              ▶ Event.java
              ▶ MessageSent.java
              ▶ NodeAdded.java
              ▶ NodeRemoved.java
```

Each time an object of one of these classes is created, the according constructor is called and an entry in the particular database table inserted. This happens during simulation after selecting a launch configuration. The code for inserting an added edge into the database can be seen from Listing 4.2. It needs to be mentioned that *time*, *nodeA*, *nodeB* and *type* are variables containing the actual values.

Listing 4.2: Insert into Database Table *edge*

```
Simulator.stmt.executeUpdate("INSERT edge VALUES ('"
            + time + "', '"
            + nodeA + "', '"
            + nodeB + "', '"
            + type + "', '"
            + "added" + "')");
```

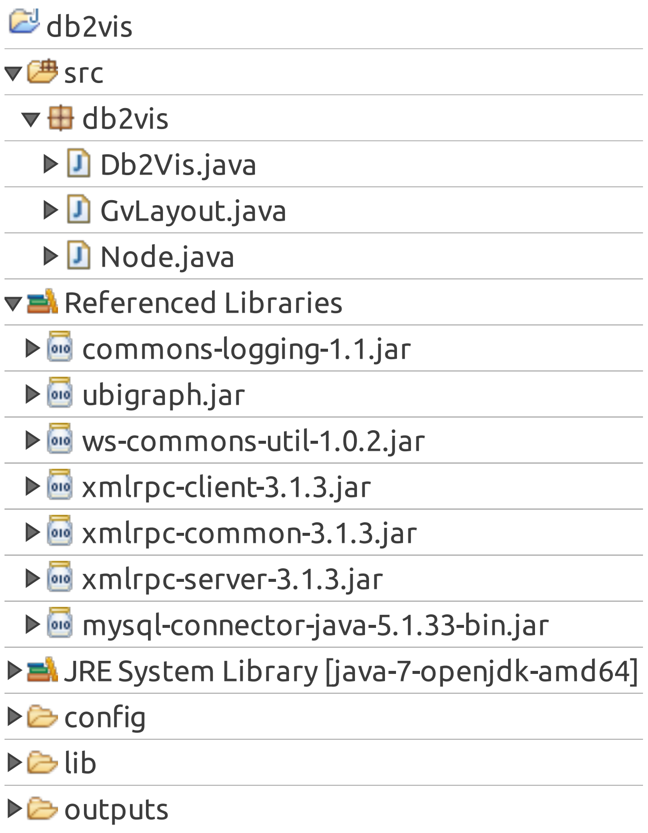
4.2 *db2vis* - Database to Visualization

This section presents the implementation of the software *db2vis* that was developed during this bachelor thesis. Subsection 4.2.1 starts with the structure, followed by the configuration in 4.2.2, the user data in 4.2.3, the database configuration in 4.2.4, the logging in 4.2.5 and eventually the visualization in 4.2.6.

4.2.1 Structure

As one can see in Figure 4.3 the *Eclipse* [Ecl14] project *db2vis* consists of the identically named package containing the three Classes *Db2Vis*, *GvLayout* and *Node*. Several libraries are referenced to communicate with *UbiGraph* and *MySQL* servers.

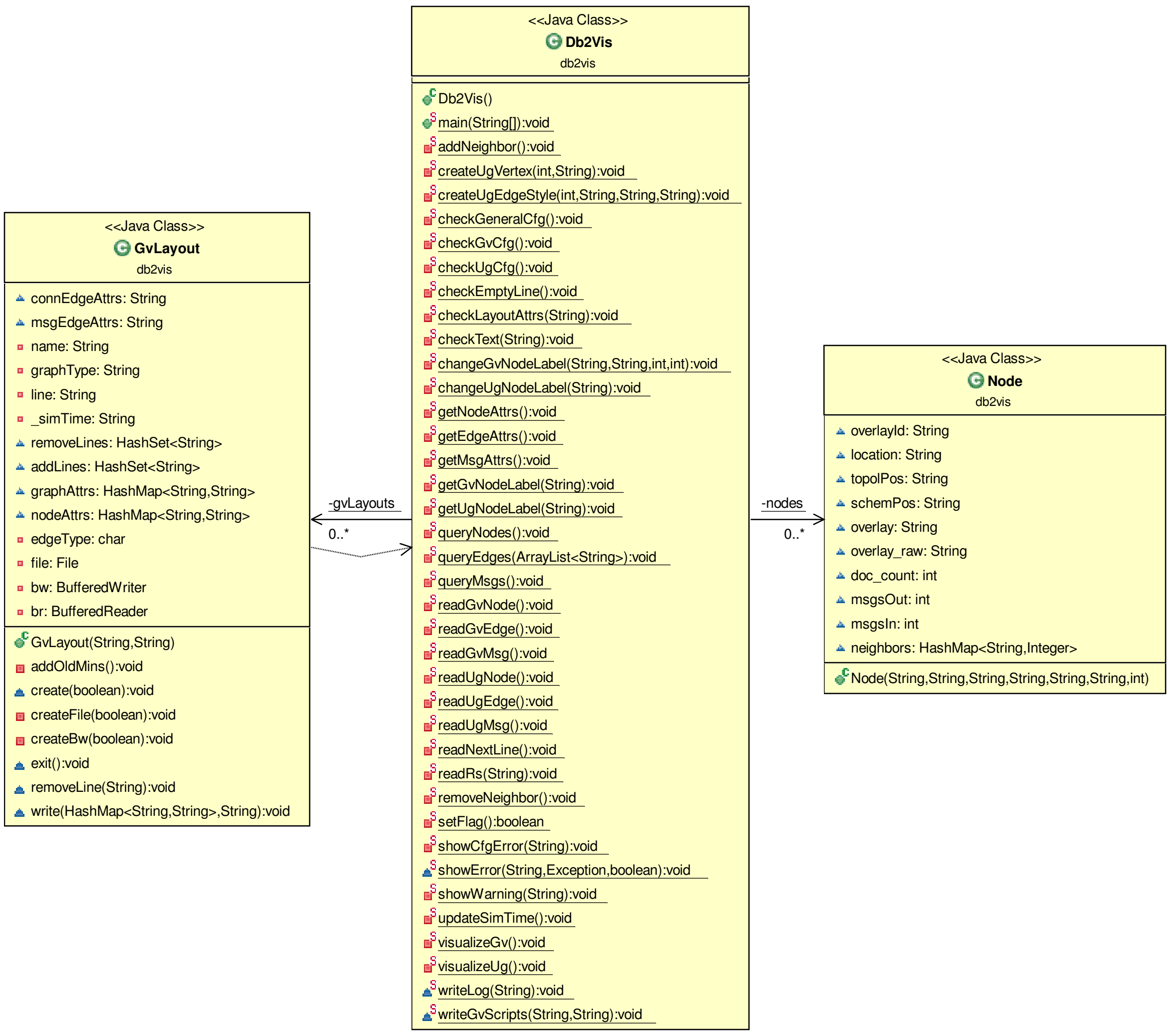
Figure 4.3: Project Structure of Program *db2vis*



```
db2vis
▼ src
  ▼ db2vis
    ▶ Db2Vis.java
    ▶ GvLayout.java
    ▶ Node.java
▼ Referenced Libraries
  ▶ commons-logging-1.1.jar
  ▶ ubigraph.jar
  ▶ ws-commons-util-1.0.2.jar
  ▶ xmlrpc-client-3.1.3.jar
  ▶ xmlrpc-common-3.1.3.jar
  ▶ xmlrpc-server-3.1.3.jar
  ▶ mysql-connector-java-5.1.33-bin.jar
▶ JRE System Library [java-7-openjdk-amd64]
▶ config
▶ lib
▶ outputs
```

Figure 4.4 shows the non-static attributes as well as all provided methods of each class.

Figure 4.4: UML Class Diagram of Program *db2vis*



As can be seen from Figure 4.4 class *Db2Vis* contains the main method and provides a lot of other methods to reduce redundancy on the one hand and to structure and modularize its source code on the other hand. Methods exclusively used for *Graphviz* or *UbiGraph* contain *Gv* respectively *Ug* in their identifiers. The abbreviation *Attrs* in the identifiers of affected methods and variables represents *Attributes*, similarly to that *Msg* means *Message*, *Rs* denotes *Result set*, *doc_count* stands for *document counter*, *topol* represents *topological*, *schem* means *schematic*, *Pos* denotes *Position* and *Cfg* stands for *Configuration*.

4.2.2 Configuration

Corresponding to the acronyms explained above the methods *checkGeneralCfg*, *checkGvCfg* and *checkUgCfg* screen the three sections of the user configuration file (see Subsection 3.2.1). While doing so they use the methods *checkEmptyLine*, *checkLayoutAttrs*, *checkText*, *readNextLine*, *setFlag* and *showCfgError*. In Table 4.1 the extensive possibilities of configuration can be found.

Table 4.1: Configuration Settings of Program *db2vis*

Setting	**Attributes/Values**
General	
Database server	IP/Hostname of the machine hosting the *MySQL* server
Database name	Any database generated by *PeerfactSim.KOM*
Create log file containing detailed information about internal program behavior	*yes* / *no*
Visualization tools	*graphviz* / *ubigraph* / *both*
Desired beginning time of the simulation	*<hhh>:<mm>:<ss>.<fff>*, leading zeros in each case and fractional seconds are optional, maximum: *838:59:59.999*
Desired ending time of the simulation	*<hhh>:<mm>:<ss>.<fff>*, leading zeros in each case and fractional seconds are optional, maximum: *838:59:59.999*
Show hosts' net ID	*yes* / *no*
Show hosts' overlay ID	*yes* / *no*
Show hosts' topological position	*yes* / *no*
Show hosts' schematic position	*yes* / *no*
Show hosts' used overlays	*yes* / *no*
Show hosts' overlays, class names	*yes* / *no*
Show hosts' number of documents	*yes* / *no*
Show hosts' number of neighbors	*yes* / *no*; Only shown connection edges are considered
Show hosts' number of sent messages	*yes* / *no*; Ignored if no shown message edges specified; Only shown message edges are considered
Show hosts' number of received messages	*yes* / *no*; Ignored if no shown message edges specified; Only shown message edges are considered

Connection edges to be shown	Comma- and blank-separated names; Optional if message edges to be shown are specified
Message edges to be shown	Comma- and blank-separated names; Optional if connection edges to be shown are specified
Show message edge labels containing their particular type	*yes* / *no*
Connection colors and names	Hexadecimal RGB triple, blank, comma- and blank-separated names
Message colors and names	Hexadecimal RGB triple, blank, comma- and blank-separated names
Graphviz	
Layouts	All layouts listed at `http://www.graphviz.org/Documentation.php`
Layout-specific attributes	All attributes listed at `http://www.graphviz.org/content/attrs`; Optional
Output file formats	All output formats listed at `http://www.graphviz.org/content/output-formats`
Generate new *gv* and output file every *x* minutes	Positive integer; Optional
Generate new *gv* and output file when new information about specified edge type(s) has occurred	Comma- and blank-separated names; Optional
Generate new *gv* and output file when specified action has occurred	*added* / *removed*; Ignored if no edge type is specified above
Should the generated *gv* and output files contain only the particular simulation minute or also all previous ones?	*particular* / *all*; Ignored if only one *gv* file
Generate *Windows* batch file for *Graphviz*	*yes* / *no*
Generate *Unix* shell script for *Graphviz*	*yes* / *no*
Run the generated *Windows* batch file for *Graphviz*	*yes* / *no*; Ignored if not generated
Run the generated *Unix* shell script for *Graphviz*	*yes* / *no*; Ignored if not generated

UbiGraph	
Complete path to binary file *ubigraph_server*	Relative or absolute path
Factor accelerating edge visualization	Integer
Factor accelerating message visualization	Integer
Vertex attributes	All vertex attributes listed at `http://www.ubietylab.net/ubigraph/content/Docs/index.html#vertexattributes`; Optional
Connection edge attributes	All edge attributes listed at `http://www.ubietylab.net/ubigraph/content/Docs/index.html#edgeattributes`; Optional
Message edge attributes	All edge attributes listed at `http://www.ubietylab.net/ubigraph/content/Docs/index.html#edgeattributes`; Optional

The support of all *Graphviz* layouts and output formats as well as all attributes of *Graphviz* and *UbiGraph* was explicitly not realized by including each of them in *db2vis*. That would be a huge effort and not future-proof.
Instead a new object of class *GvLayout* is created for each user-configured *Graphviz* layout and added to the hash map *gvLayouts* for later reference. Since the according constructor demands the name and graph type (directed/undirected) of the particular layout and both details have to be declared by the user, *db2vis* does not limit the supported layouts. The graph type can be used to determine the edge type in the layout-specific *gv* file thanks to the fact that each *gvLayout* contains references to its own files. Similarly the layout names as well as the user-specified output formats are just forwarded to the generated script files that specify which layout should read and write which files.
The different attributes and values are supported by considering not their semantics but common syntax. Each pair of attribute and value is passed on to *Graphviz* and *UbiGraph*, respectively, in the format expected by the particular visualization tool. Therefore *db2vis* itself interprets neither attributes nor values.

Due to the limitation of the *MySQL* data type *TIME(3)* the desired beginning or ending time of the simulation must not surmount *838:59:59.999*.

4.2.3 User Data

If the configuration was successfully checked the user is asked for his or her username and password of the database specified in the configuration file. The system console is used if available in order to hide the entered password. Otherwise (for example when running *db2vis* in *Eclipse* [Dau04]) a scanner is necessary to read the password, which cannot mask or hide it. In that case the user is warned about this security risk in the password prompt.

4.2.4 Database Connection

After reading the necessary credentials the program loads the *JDBC* driver and connects via *JDBC* to the given *MySQL* database. This API is provided by package *java.sql*. The binary for *MySQL Connector/J* (*mysql-connector-java-5.1.33-bin.jar*) is referenced by *db2vis* as one can see in Figure 4.3.

4.2.5 Logging

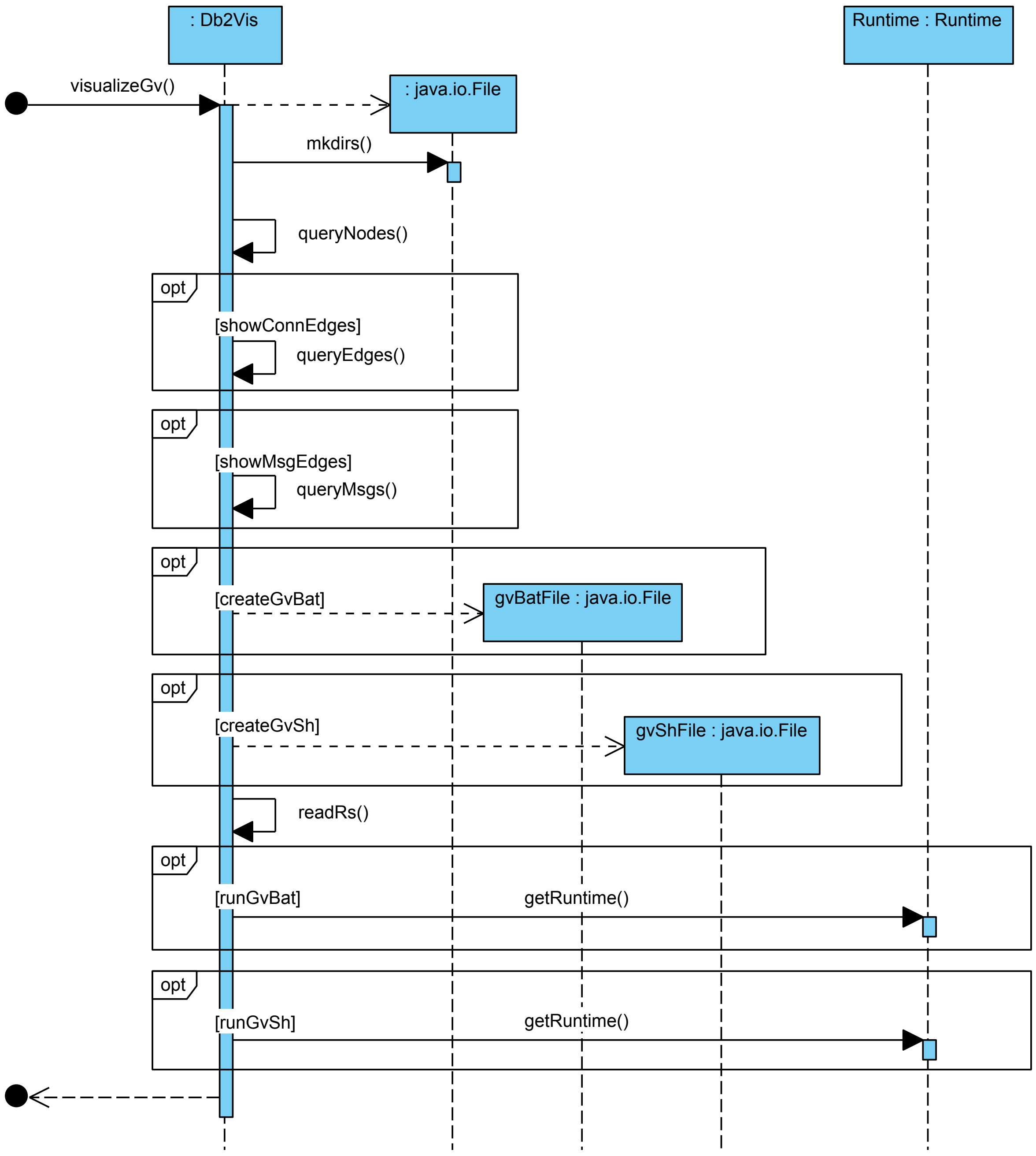
In case of a successfully established connection the log directory and file are created if the logging feature is enabled by the user (see Subsection 3.2.2). In that case all relevant events like input/output operations or warning/error messages are logged from this moment on by calling method *writeLog*.

4.2.6 Visualization

If both visualization tools are specified to be used, *db2vis* starts with *Graphviz* by calling method *visualizeGv* and then executes *UbiGraph* commands by applying *visualizeUg*. In case of only one tool to be used the method of the other one is skipped.

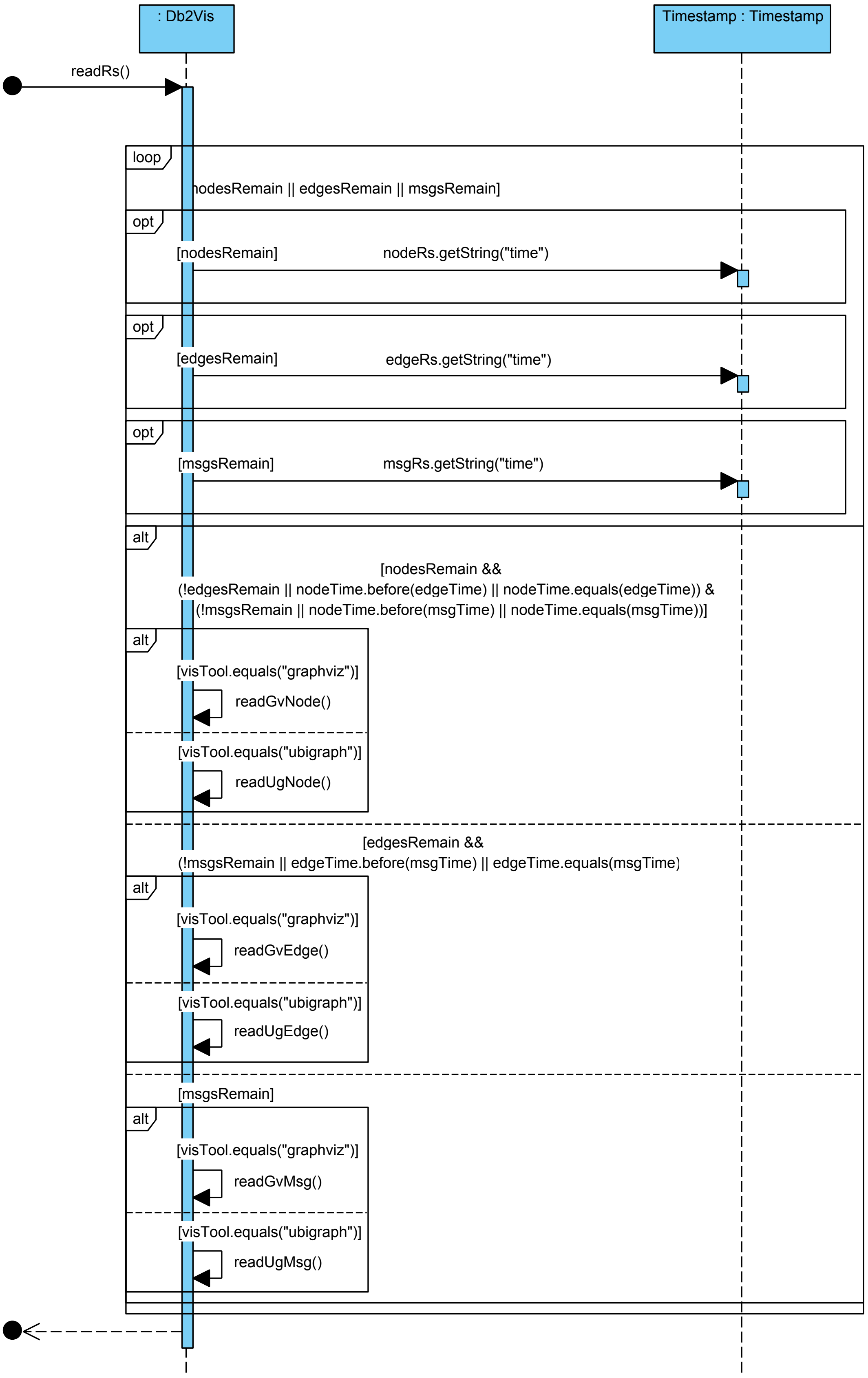
Graphviz

Figure 4.5: UML Sequence Diagram of *Java* Method *visualizeGv*



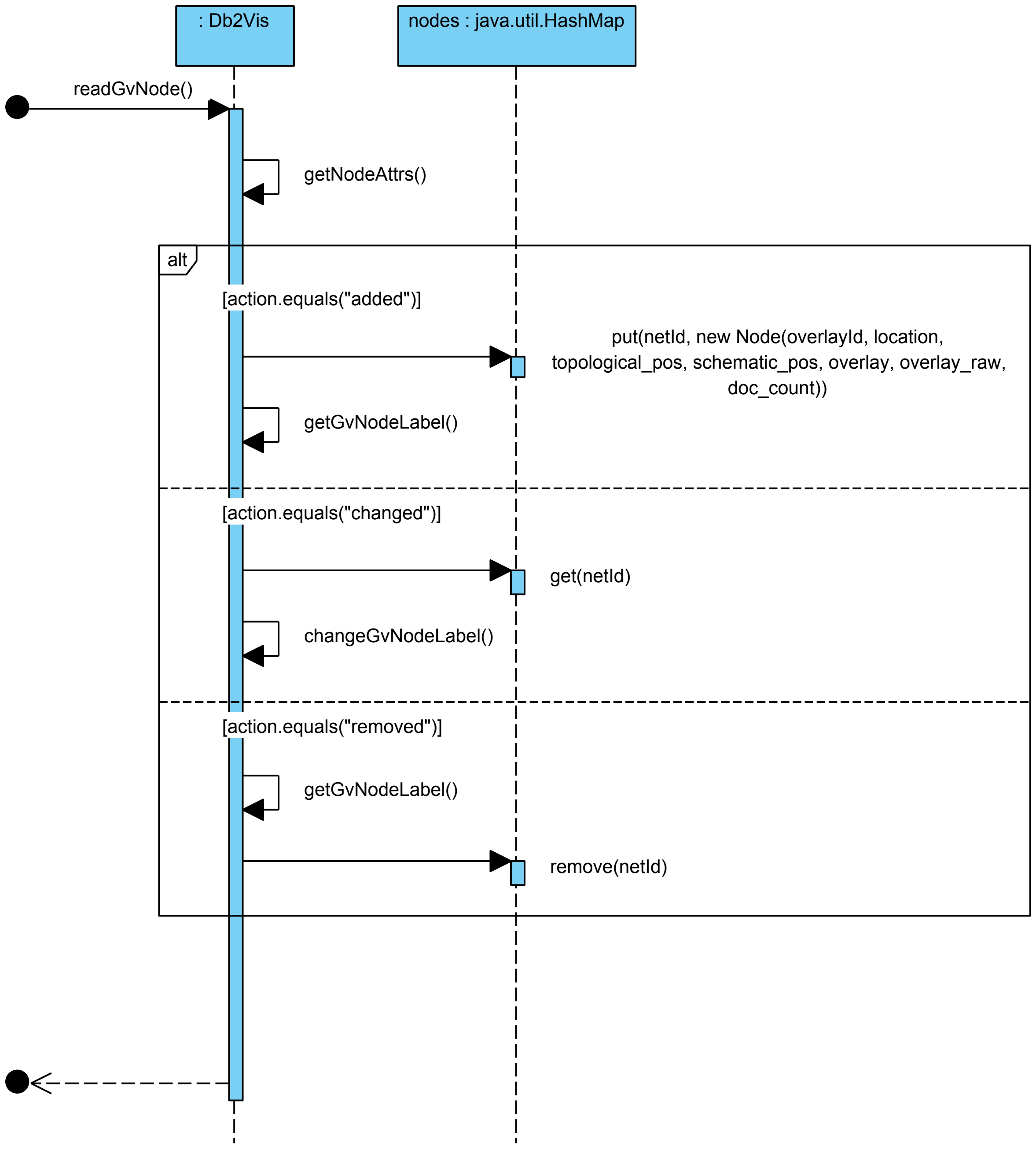
At the beginning of *visualizeGv* (see Figure 4.5) the output directory including its subfolders is created (see Subsection 3.2.3). After that the necessary nodes, edges and messages are queried by using the methods *queryNodes* / *queryEdges* / *queryMsgs*. To reduce the performance impact due to *MySQL* interaction only the edge and message types specified by the user (see Table 4.1) are queried. If the user did not specify either edges or messages the respective table is not queried at all to further improve the performance. Analogously only the time window between the desired beginning and ending time of the simulation (see Table 4.1) is queried.
In case of activated script generation the according script files are created next (see Subsubsection *Graphviz* of Subsection 3.2.3). Due to the potentially time-consuming *gv* file creation process, the user is briefly informed by the console message *Started creating Graphviz files...*
When that is done, the result sets are read in chronological order by executing *readRs*.

Figure 4.6: UML Sequence Diagram of *Java* Method *readRs*


: Db2Vis
Timestamp : Timestamp
readRs()
loop
nodesRemain || edgesRemain || msgsRemain]
opt
[nodesRemain]
nodeRs.getString("time")
opt
[edgesRemain]
edgeRs.getString("time")
opt
[msgsRemain]
msgRs.getString("time")
alt
[nodesRemain &&
(!edgesRemain || nodeTime.before(edgeTime) || nodeTime.equals(edgeTime)) &
(!msgsRemain || nodeTime.before(msgTime) || nodeTime.equals(msgTime))]
alt
[visTool.equals("graphviz")]
readGvNode()
[visTool.equals("ubigraph")]
readUgNode()
[edgesRemain &&
(!msgsRemain || edgeTime.before(msgTime) || edgeTime.equals(msgTime)
alt
[visTool.equals("graphviz")]
readGvEdge()
[visTool.equals("ubigraph")]
readUgEdge()
[msgsRemain]
alt
[visTool.equals("graphviz")]
readGvMsg()
[visTool.equals("ubigraph")]
readUgMsg()

As can be seen from Figure4.6 this method starts to read all three database tables in chronological order by reading line after line of each table and comparing their time fields. The earliest event is visualized first by calling either *readGvNode*, *readGvEdge* or *readGvMsg*.

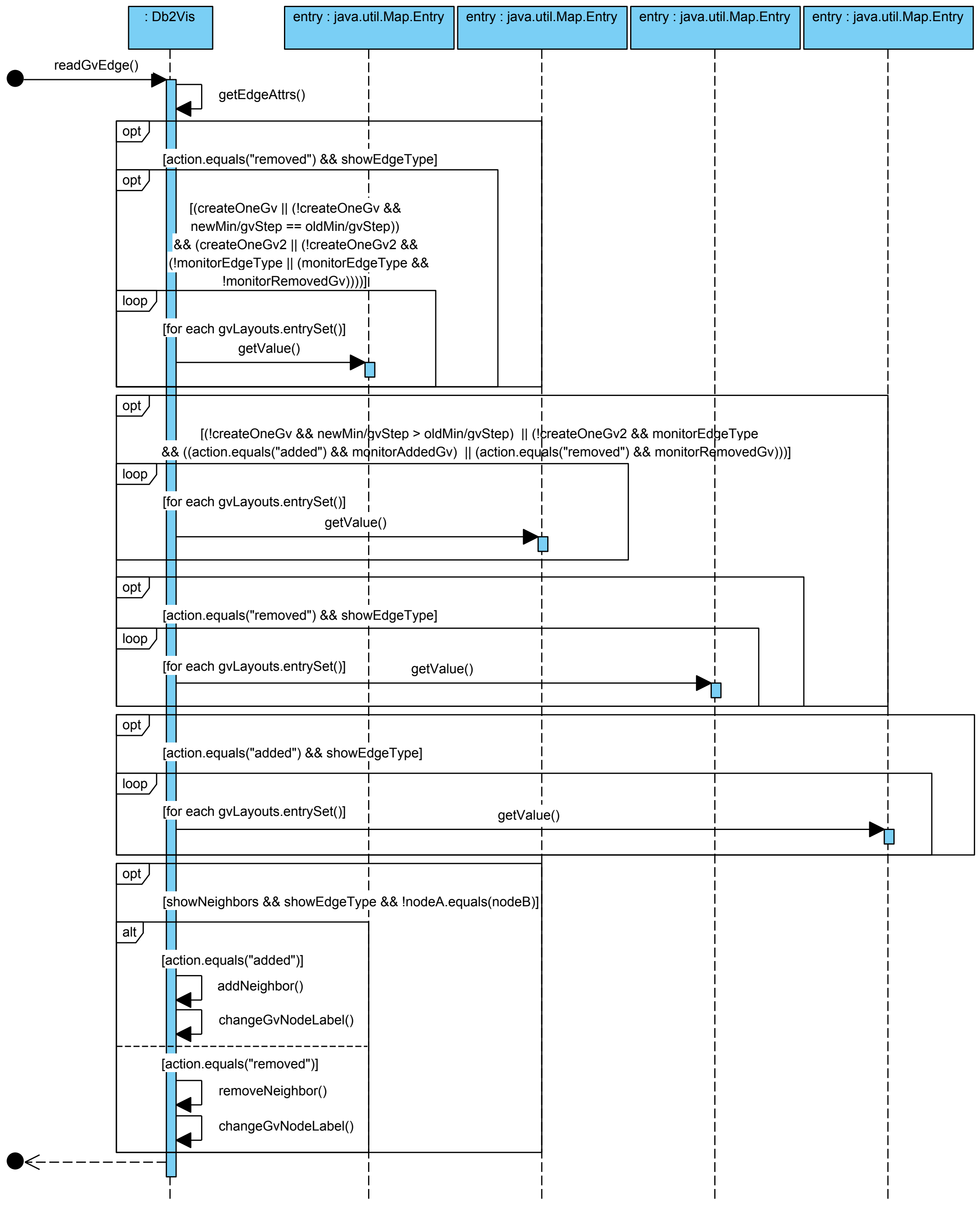
Figure 4.7: UML Sequence Diagram of *Java* Method *readGvNode*


: Db2Vis
nodes : java.util.HashMap
readGvNode()
getNodeAttrs()
alt
[action.equals("added")]
put(netId, new Node(overlayId, location, topological_pos, schematic_pos, overlay, overlay_raw, doc_count))
getGvNodeLabel()
[action.equals("changed")]
get(netId)
changeGvNodeLabel()
[action.equals("removed")]
getGvNodeLabel()
remove(netId)

readGvNode (see Figure 4.7) first reads the node attributes by applying *getNodeAttrs*. One of three possible strings in table field *action* is expected, namely *added*, *changed* or *removed* (see Table 3.2).

In case of *added* a new object of class *Node* is created by using the according constructor with all other field values of the particular database entry as arguments. To keep a reference to this object it is added to the hash map *nodes*. With the help of *getGvNodeLabel* the label containing all user-activated attributes for the respective node is constructed. By adding the built *Graphviz* line to the hash set *addLines* and removing it from the hash set *removeLines* of each user-given layout the node-specific attributes are saved for later insert into the current *gv* file.
If the *action* field equals *changed* the affected value of the particular node is updated. Subsequently the label is changed with the help of *changeGvNodeLabel*.
The third and last possible event is *removed*, which revokes the *added* operation by adding the according *Graphviz* line to the hash set *removeLines* and removing it from the hash set *addLines* as well as the hash map *nodes*.

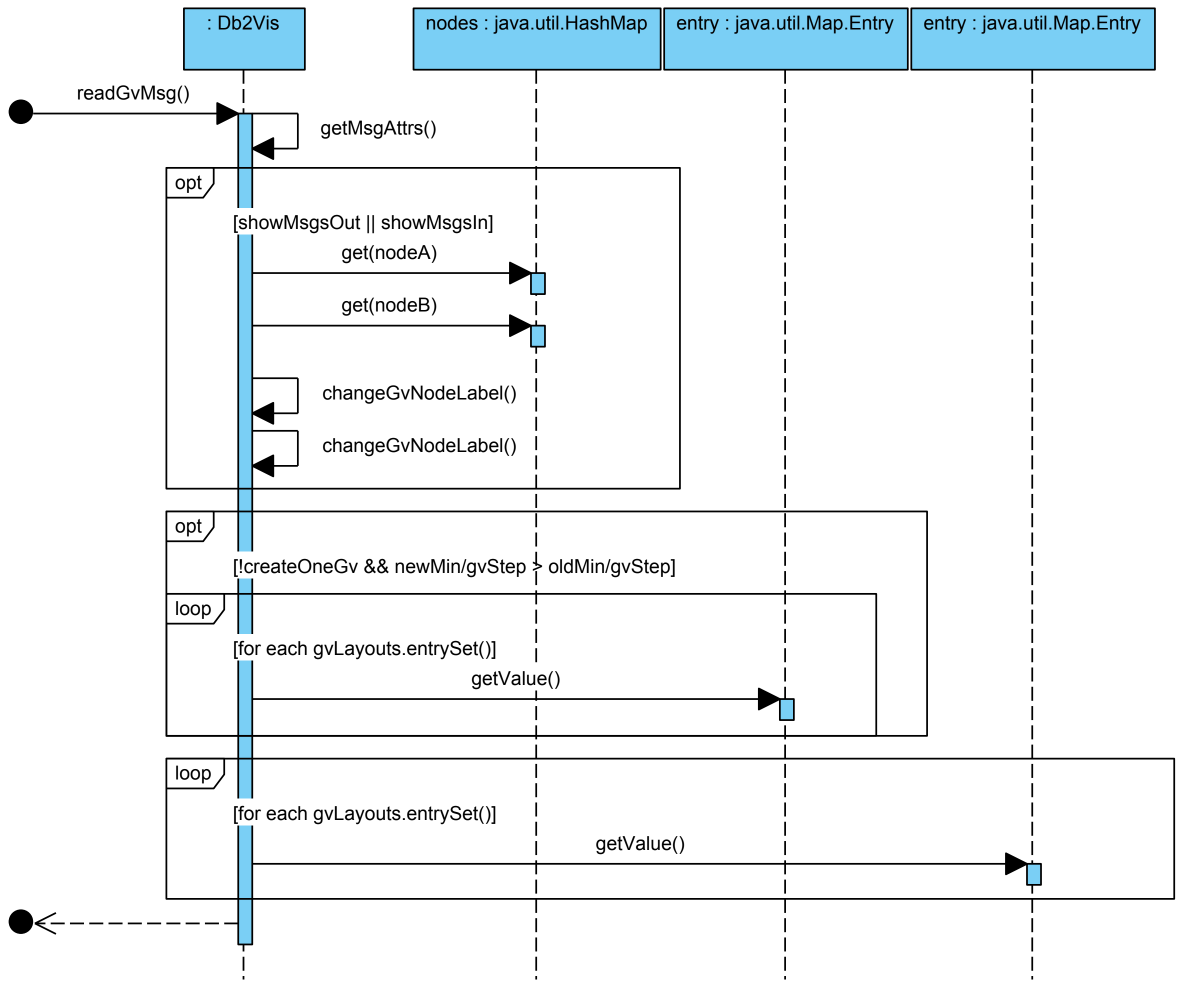
Figure 4.8: UML Sequence Diagram of *Java* Method *readGvEdge*


: Db2Vis
entry : java.util.Map.Entry
entry : java.util.Map.Entry
entry : java.util.Map.Entry
entry : java.util.Map.Entry
readGvEdge()
getEdgeAttrs()
opt
[action.equals("removed") && showEdgeType]
opt
[(createOneGv || (!createOneGv && newMin/gvStep == oldMin/gvStep)) && (createOneGv2 || (!createOneGv2 && (!monitorEdgeType || (monitorEdgeType && !monitorRemovedGv))))]
loop
[for each gvLayouts.entrySet()]
getValue()
opt
[(!createOneGv && newMin/gvStep > oldMin/gvStep) || (!createOneGv2 && monitorEdgeType && ((action.equals("added") && monitorAddedGv) || (action.equals("removed") && monitorRemovedGv)))]
loop
[for each gvLayouts.entrySet()]
getValue()
opt
[action.equals("removed") && showEdgeType]
loop
[for each gvLayouts.entrySet()]
getValue()
opt
[action.equals("added") && showEdgeType]
loop
[for each gvLayouts.entrySet()]
getValue()
opt
[showNeighbors && showEdgeType && !nodeA.equals(nodeB)]
alt
[action.equals("added")]
addNeighbor()
changeGvNodeLabel()
[action.equals("removed")]
removeNeighbor()
changeGvNodeLabel()

Method *readGvEdge* gains the edge attributes via *getEdgeAttrs* as can be seen from Figure 4.8. Expected values for *action* are either *added* or *removed* (see Table 3.4).

added can cause different operations depending on the configuration file. For each edge type the user can define if it is to be shown, produce a new *gv* file when a specified action occurs (so-called monitored connection edges) or even both (see Table 4.1). In case of an edge to be shown it is written into the current *gv* file by executing method *write* of each layout in the hash map *gvLayouts*. If the edge type and action are expected to finish the current *gv* file and generate a new one, method *create* of all defined layouts is called with its parameter set to *true* in order to not only create a new file but also close the old one (in contrast to the first file created with this method). If the user has enabled the counting of neighbors and the examined edge is meant to be shown, method *addNeighbor* is called and the node's label changed with the help of *changeGvNodeLabel*.
The other possible action is *removed* and also differentiates between shown and monitored edges. If the edge type is shown and the action not supposed to be monitored, the according *Graphviz* line is marked to be removed in the current *gv* file by calling *removeLine* of each specified layout. In the case of monitoring removed edges of the particular type, similarly to the monitored edges added a new *gv* file is created. If the edge is a shown one, it is removed from the new (but not old) file in order to detect the difference when comparing both time windows. Analogously to the *added* action the neighbor counter is updated if it and the particular edge are to be shown, but this time by applying *removeNeighbor* before changing the node label with *changeGvNodeLabel*.
Apart from the configuration of shown edges as well as monitored edge types and actions, the user can configure a time interval, after which a new *gv* file should be created (see Table 4.1). For that purpose the program compares the previous simulation minute at which a new file was created with the current one. If it is at least as long ago as the defined step-width the next file begins.

Figure 4.9: UML Sequence Diagram of *Java* Method *readGvMsg*


: Db2Vis
nodes : java.util.HashMap
entry : java.util.Map.Entry
entry : java.util.Map.Entry
readGvMsg()
getMsgAttrs()
opt
[showMsgsOut || showMsgsIn]
get(nodeA)
get(nodeB)
changeGvNodeLabel()
changeGvNodeLabel()
opt
[!createOneGv && newMin/gvStep > oldMin/gvStep]
loop
[for each gvLayouts.entrySet()]
getValue()
loop
[for each gvLayouts.entrySet()]
getValue()

In case method *readGvMsg* (see Figure 4.9) is called, the message attributes are read with the help of *getMsgAttrs*. After that *msgsOut* of the sender and *msgsIn* of the receiver are incremented and their labels updated if the user configured these message counters to be shown (see Table 4.1). Similarly to method *readGvEdge* described above a new *gv* file is created each time the user-configured time interval has passed. If this happens, the occurred message is written into the new file, otherwise into the old one.

After reading the whole result set and writing all necessary text lines into the *gv* files and scripts, another brief user information is provided: *Finished creating Graphviz files*. Eventually the scripts can optionally be started to render the generated *gv* files (see Table 4.1).

UbiGraph

visualizeUg first clears the *nodes* hash map if *Graphviz* has used it before. Analogously to *visualizeGv* the three database tables are queried with *queryNodes*, *queryEdges* and *queryMsgs*, the latter two in dependence of the user configuration (see Table 4.1). As before the creation of *gv* files the user gets a short status message when the *UbiGraph* visualization begins. Then the *UbiGraph* server is started and a new graph created. Setting the user-configured vertex attributes (see Table 4.1) and reading the result set via *readRs* are the subsequent steps. As detailed at the beginning of Subsubsection *Graphviz* this method reads all three database tables in chronological order by reading line after line of each table and comparing their time fields. This time the methods *readUgNode*, *readUgEdge* and *readUgMsg* are used to examine the particular table entry.

Like *readGvNode* method *readUgNode* first requests the node attributes by using *getNodeAttrs*. Since the methods for *Graphviz* and *UbiGraph* use the same three database tables, *readUgNode* also expects *added*, *changed* or *removed* in the *action* field of the *node* table.
Also analogously to *readGvNode* the particular node is added to the *nodes* hash map if the action demands this. In addition method *createUgVertex* is called to create the *UbiGraph* vertex after building the node label with the help of *getUgNodeLabel*.
In case of a table entry marked as changed the according node attribute and vertex label are updated.
The removal of a node has to be done in two steps - first by removing its vertex from the graph and second by removing it from the *nodes* hash map.

readUgEdge of course first calls *getEdgeAttrs* and also expects the particular edge to be either added or removed. In contrast to *readGvEdge* method *readUgNode* can optionally slow down the visualization by pausing the main thread for the same time span as the difference between the simulation time of the previous and current event divided by the factor given in the user configuration file (see Table 4.1). Whereas that would not make sense while writing the *Graphviz* lines into the according files, it helps the user keep track of dynamic graph changes during the *UbiGraph* visualization.
In case of an edge to be added it is created in the graph and styled on the basis of the user-defined connection edge attributes (see Table 4.1). For both purposes *UbiGraph* provides according methods. Removing an edge is as simple as it seems: Just call the correspondent method provided by *UbiGraph*. If the node's number of neighbors is to be shown it is updated by executing *removeNeighbor* and *changeUgNodeLabel*.

Method *readUgMsg* begins with retrieving the message attributes via *getMsgAttrs* as one might have already expected. As in *readGvMsg* the message counters are updated if necessary. Afterwards a new message edge is drawn and styled by suitable *UbiGraph* methods as defined in the message edge attributes of the user configuration file (see Table 4.1). The time of message edge visualization can be extended by the same principle as used in *readUgEdge*, but with its own factor as one can see in Table 4.1. When the sleeping thread wakes up the message edge is removed, therefore it is only temporary in contrast to the connection edges drawn by *readUgEdge*. This behavior is similar to that of *PeerfactSim.KOM*, where messages produce so-called flashing edges.

To inform the user about the finish of the *UbiGraph* visualization a final short information message is provided.

Chapter 5

Evaluation

This chapter compares *PeerfactSim.KOM* with *Graphviz* and *UbiGraph* in Section 5.1 on the one hand and with *db2vis* in Section 5.2 on the other hand. The challenge of appropriate visualization is discussed in Section 5.3, whereas Section 5.4 examines the scalability referring to the overlay size.

5.1 Comparison of Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

In this section the visualization capabilities of *PeerfactSim.KOM* (Subsection 5.1.1), *Graphviz* (Subsection 5.1.2) and *UbiGraph* (Subsection 5.1.3) are compared. Both the features and limitations are considered. Subsection 5.1.4 offers facts and figures about the three tools.
As *PeerfactSim.KOM* mainly is a simulator and not primarily a visualization tool, it offers fewer graphical possibilities than *Graphviz* and *UbiGraph*. On the other hand the latter two of course do not provide any simulation functionalities. Therefore the three programs have different aims, which needs to be considered.

5.1.1 *PeerfactSim.KOM*

Features

PeerfactSim.KOM [GLS$^+$10] handles the most important visualization features like independently filtering single or multiple message and edge types as well as node- and edge-specific metrics. Edges and metrics are mostly shown in different colors in order to distinguish them.

A legend informs users about the mapping. A very useful feature is the possibility of rewinding or fast-forwarding the visualization in configurable steps and moving within the simulation time with a horizontal scroll bar. Users can specify the play speed, zoom in and out, switch between a schematic and topological view, set the font size, save and load recordings and let the nodes be sized relatively to different metrics. Connections are drawn as solid permanent edges, messages as dashed temporary ones.

Limitations

As quite a few different message and edge types use the same color and can only be mapped in the source code, there is room for improvement. The stroke type of the edges (like solid, dashed, dotted, ...) cannot be configured by the user. Users are not able to jump to a specific moment of the simulation by entering a concrete time. Recordings can only be saved in the proprietary file format *peerfact* and therefore be opened exclusively by *PeerfactSim.KOM*. Apart from several generated statistics which can be plotted by *gnuplot* [Tho14], there is no textual output. Therefore it is not possible for example to search for specific events or list each neighbor of a certain node, all messages between two nodes and so on.

5.1.2 *Graphviz*

Features

Graphviz [EGK+04] features lots of graph, node and edge attributes, several layouts as well as various node and arrow shapes. As the software is only as good as its inputs the results depend on the commands given, in this case by *db2vis*. In contrast to *PeerfactSim.KOM* any of 16.7 million colors can be separately assigned to any edge and message type in the user configuration file. The shape of nodes, connection and message edges can also be configured by the user. Another benefit of *Graphviz* in combination with *db2vis* is the possibility of saving the graphs in different output formats. This guarantees compatibility and portability. As all outputs base upon text inputs, simulations or parts of them can be analyzed by text-based scripts. Due to the fact that *Graphviz* does not only support its own output formats like *dot*, but also popular ones for vector and raster graphics like *svg* and *png*, respectively, as well as the *Portable Document Format* (*PDF*) the diagrams can be viewed by any compatible program. In consequence a lot of features like zooming, turning et cetera no longer depend on *Graphviz* or *db2vis*.

Limitations

A shortfall of *Graphviz* (at least in combination with *db2vis*) is that unlike *PeerfactSim.KOM* there is no legend containing the mapping of edge types to colors beneath the generated drawings. Therefore users have to either activate the labeling of edges or create a legend on their own. Usually this is only a disadvantage if the author of the configuration file is not the (only) one who views the diagrams as the edge types and colors are manually mapped and the user therefore should be aware of the mapping. Since there is no legend, the node labels have to contain not only the metric values but also their names in order to distinguish them. In addition the different metrics are presented with the same font color, as each node holds just one label and no possibility could be found to use different font colors within the same node label.
A problem connected with the static graph design is that events cannot be visualized temporarily. This affects the so-called flashing edges which are used for message visualization. In order to make them disappear shortly after their occurrence two new *gv* files would have to be created each time a message is sent - one containing the edge and another minus the edge. This way a huge number of files would be produced. In order to avoid such storage consumption messages like connections are permanently visualized. Thanks to the possibility of creating a new output file each time a configurable time interval has passed or specific events have occurred, messages can at least be illustrated exclusively in the diagrams created since the moment of transmission.

5.1.3 *UbiGraph*

Features

UbiGraph enables the user to view the particular graph from different perspectives due to its three-dimensional presentation. This makes it possible to recognize plastic formations formed by the particular overlay. As the graph is dynamic, all objects like nodes and edges can be removed without much effort which makes it possible to visualize messages temporarily. Various vertex and edge attributes are provided.

Limitations

Often some nodes or edges are not visible as other ones hide them. Probably caused by the alpha status of *UbiGraph* a few attributes like the edge width have no effect. While working on this thesis no way could be found to break lines in node labels. Therefore all node attributes have to be listed one after another in a single line, separated by semicolons.

Like in *Graphviz* neither a legend nor different colors within the same node label seem to be supported by *UbiGraph*. As there is no possibility of saving a graph, it needs to be regenerated each time the user wants to visualize it.
In contrast to *Graphviz* there seems to be no way of changing the color of the black background. Sometimes this makes it hard to detect dark-colored temporary message edges and of course impossible to see any black ones. The *UbiGraph* server is not yet ported to *Windows*.
Some attributes exclude each other even if they should not. A good example is the simultaneous activation of the message attributes *spline* and *arrow*, which only produces the same undirected splines as it would if arrows were not enabled. Without activated splines, the *arrow* flag causes directed edges as expected. Multiple arrows between the same two nodes overlap each other.
As the latest *UbiGraph* version was released 2008, the software appears to be no longer maintained.

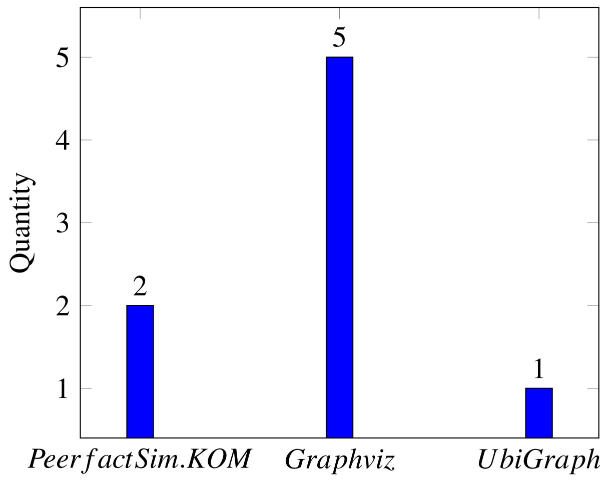
5.1.4 Facts and Figures

This subsection contains on the one hand quantitative comparisons of the layouts, node and edge attributes as well as output formats offered by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*. On the other hand the quality of different *Graphviz* layouts is presented.
Furthermore the appearance of identical overlays visualized by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph* is compared between the three programs.

Quantity of Layouts in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

Figure 5.1 is a quantitative comparison of the layouts offered by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*.

Figure 5.1: Quantity of Layouts in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*



PeerfactSim.KOM provides a schematic and topological view. Users of *Graphviz* can choose between the layouts *circo*, *dot*, *fdp*, *neato* and *twopi*. *UbiGraph* only offers the default arrangement.

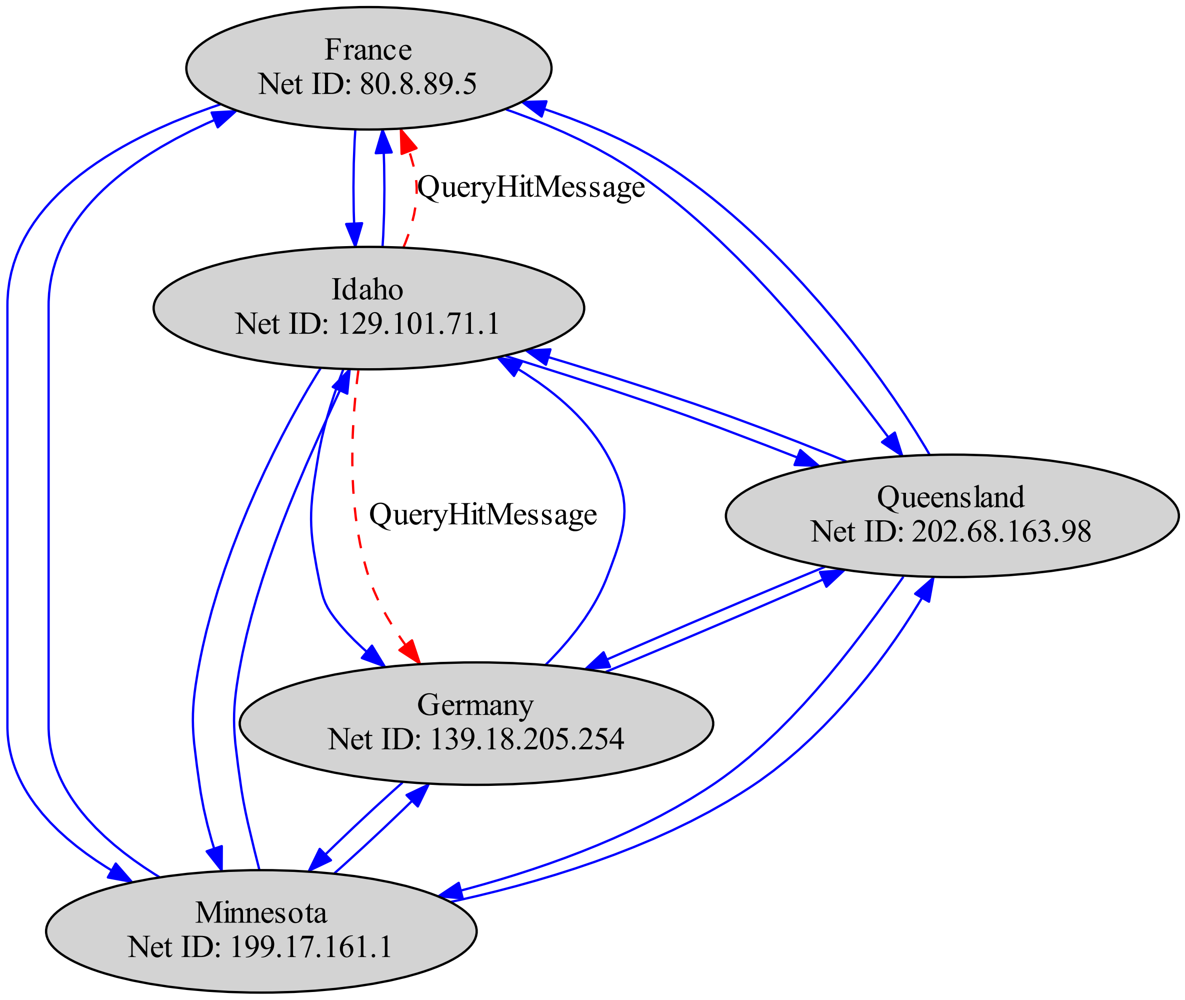
Quality of Layouts in Program *Graphviz*

One can find a *Gnutella* 0.4 overlay with five nodes, 18 connections and two messages in Figures 5.2 to 5.6. It is visualized by all five *Graphviz* layouts in the form of gray filled node shapes labeled with the particular node location and net ID, blue solid connection edges and red dashed message edges labeled with their type.

It has to be mentioned that even the same layout can produce very different arrangements depending on how many and which nodes are connected with each other.

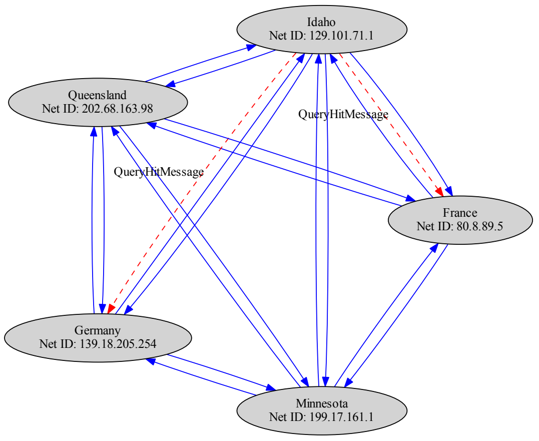
dot is a hierarchical layout with directed edges:

Figure 5.2: *Gnutella* 0.4 Overlay Visualized by *Graphviz* Layout *dot*



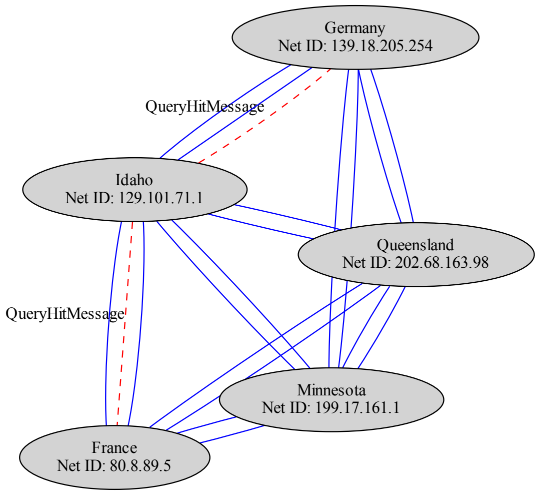
circo is a circular layout after *Six* and *Tollis* with directed edges:

Figure 5.3: *Gnutella* 0.4 Overlay Visualized by *Graphviz* Layout *circo*



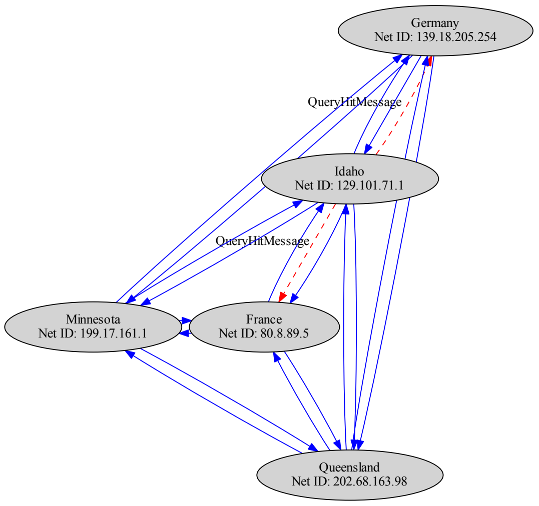
neato is a spring model layout using the *Kamada-Kawai-algorithm* with undirected edges:

Figure 5.4: *Gnutella* 0.4 Overlay Visualized by *Graphviz* Layout *neato*



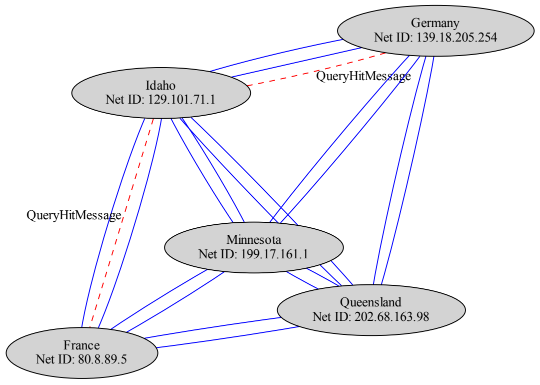
twopi is a radial layout after *Graham Wills* with directed edges:

Figure 5.5: *Gnutella* 0.4 Overlay Visualized by *Graphviz* Layout *twopi*



fdp is a spring model layout using the *Fruchterman-Reingold-heuristic* with undirected edges:

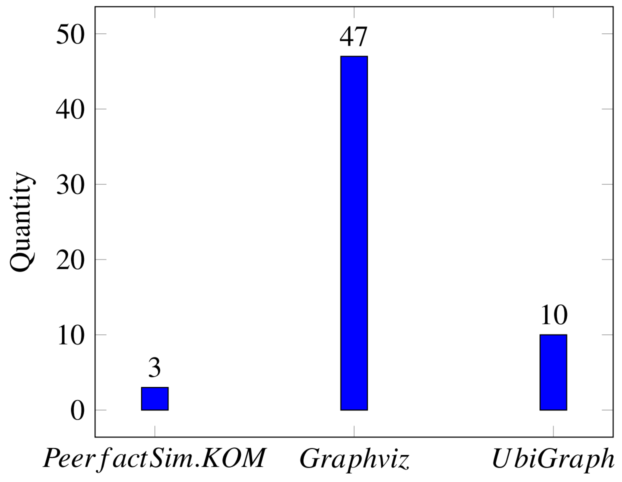
Figure 5.6: *Gnutella* 0.4 Overlay Visualized by *Graphviz* Layout *fdp*



Quantity of Node Attributes in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

The number of node attributes provided by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph* can be compared in Figure 5.7.

Figure 5.7: Quantity of Node Attributes in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

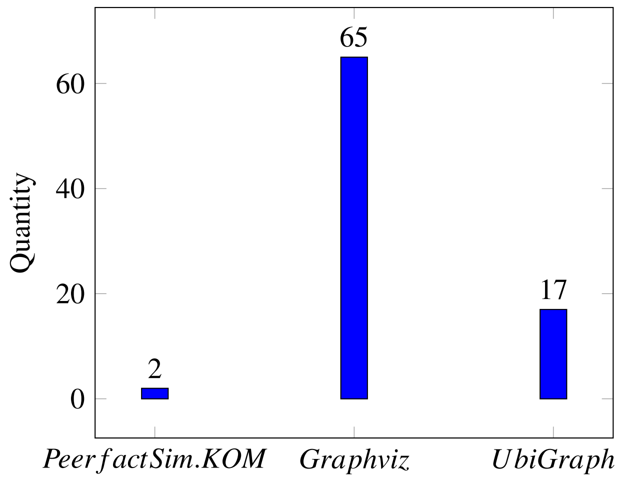


PeerfactSim.KOM provides labels, configurable font size and dynamic node size. Users of *Graphviz* can configure color, font color/size/name, height, label, width, shape and many other node attributes. In *UbiGraph* the color, font color/size/family, label, size, shape and a few other vertex attributes can be specified.

Quantity of Edge Attributes in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

In Figure 5.8 one can compare the quantity of edge attributes offered by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*.

Figure 5.8: Quantity of Edge Attributes in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*

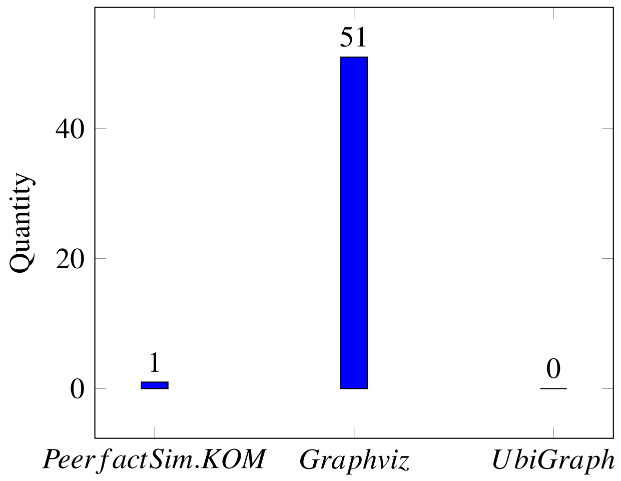


PeerfactSim.KOM provides labels and dynamic size. Users of *Graphviz* can configure color, font color/size/name, label, pen width and many other edge attributes. In *UbiGraph* the color, font color/size/family, label, width and several other edge attributes can be specified.

Quantity of Output Formats in Programs ***PeerfactSim.KOM***, ***Graphviz*** and ***UbiGraph***

In Figure 5.9 a quantitative comparison of the output formats offered by *PeerfactSim.KOM*, *Graphviz* and *UbiGraph* can be found.

Figure 5.9: Quantity of Output Formats in Programs *PeerfactSim.KOM*, *Graphviz* and *UbiGraph*



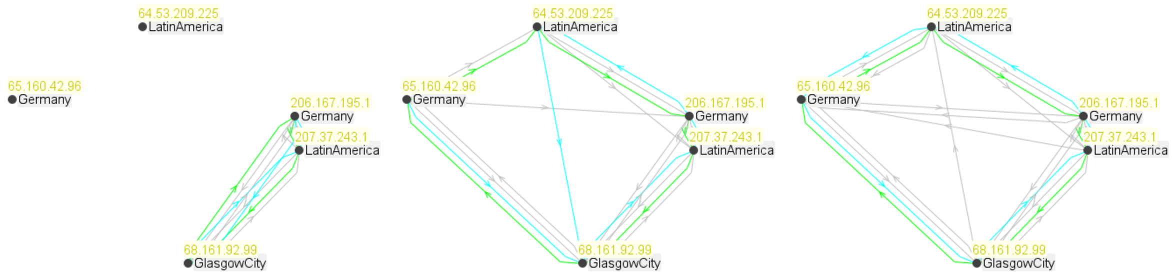
PeerfactSim.KOM exclusively provides its own file format *peerfact*. Users of *Graphviz* can generate *bmp*, *gif*, *jpg*, *pdf*, *png*, *svg*, *tif* and many other output files. In *UbiGraph* the user cannot export/save the visualization.

Same Overlays Visualized by Programs ***PeerfactSim.KOM***, ***Graphviz*** and ***UbiGraph***

Figures 5.10 to 5.12 compare the same *Chord* setup with shown *predecessor* (cyan), *successor* (green) and *finger* (grey) edges between *PeerfactSim.KOM*, *Graphviz* and *UbiGraph* at simulation minutes 4, 9 and 14. The same *Gnutella* 0.4 setup with shown *routing table entry* (grey) edges at simulation minutes 1, 3 and 7 is compared in Figures 5.13 to 5.15. Figures 5.16 to 5.18 compare the same *Pastry* setup with shown *routing table entry* (orange), *leaf set entry* (yellow) and *neighborhood set entry* (grey) edges at seconds 1, 2 and 3 of simulation minute 8. Each overlay contains five nodes and was simulated for 60 minutes while selecting the most relevant situations for visualization comparison.

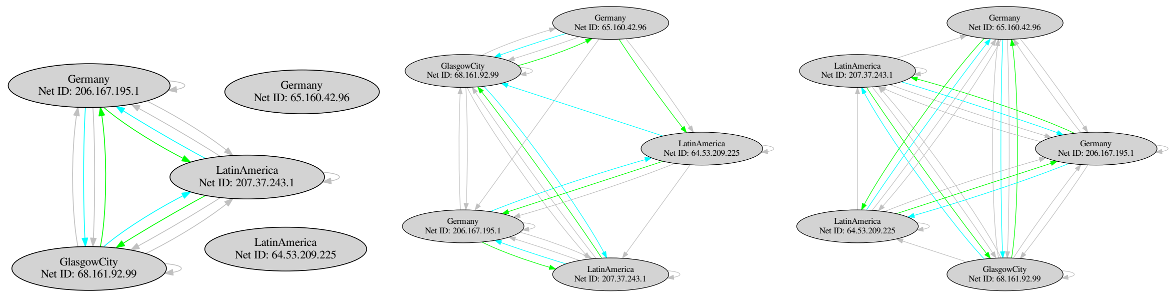
All three programs are of course capable of visualizing much bigger overlays, but these would be even less readable in this document. The shown visualizations are only examples as the same overlay can be visualized by the same tool in many different ways thanks to various settings. In addition it needs to be considered that the *UbiGraph* perspective can be changed within the application. Actual visualizations of *PeerfactSim.KOM*, *Graphviz* and *UbiGraph* could be zoomed in and out. Due to the limited respectively non-existing visualization export features of *PeerfactSim.KOM* and *UbiGraph*, their graphs had to be saved in the form of snapshots which causes rasterization and therefore quality loss. The non-changeable black background of the *UbiGraph* window (stated in Subsubsection Limitations of Subsection 5.1.3) as well as the white font color have been inverted within the snapshots.

Figure 5.10: *Chord* Overlay Visualized by *PeerfactSim.KOM*



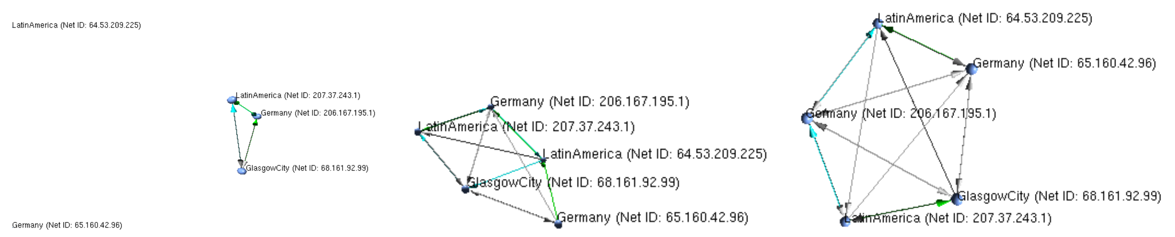
(a) Simulation Minute 4 (b) Simulation Minute 9 (c) Simulation Minute 14

Figure 5.11: *Chord* Overlay Visualized by *Graphviz*



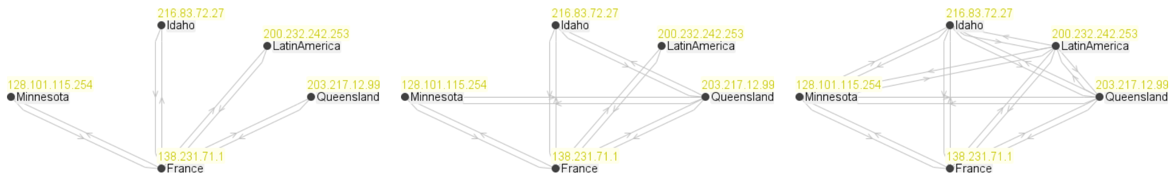
(a) Simulation Minute 4 (b) Simulation Minute 9 (c) Simulation Minute 14

Figure 5.12: *Chord* Overlay Visualized by *UbiGraph*


LatinAmerica (Net ID: 64.53.209.225)
LatinAmerica (Net ID: 207.37.243.1)
Germany (Net ID: 206.167.195.1)
GlasgowCity (Net ID: 68.161.92.99)
Germany (Net ID: 65.160.42.96)

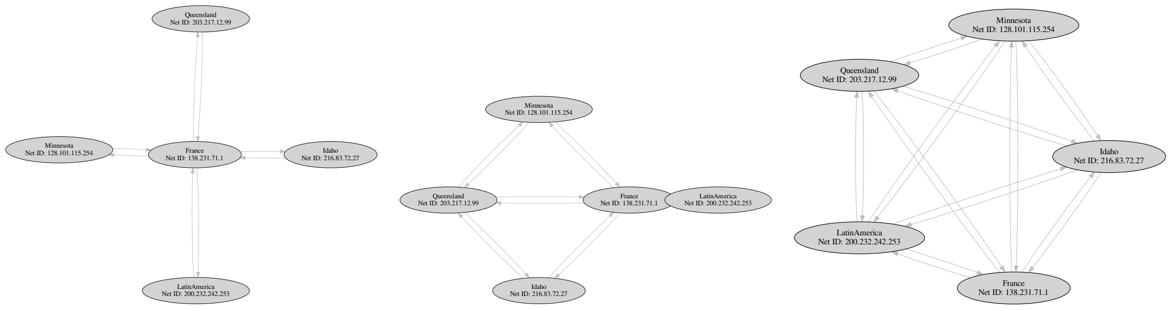
(a) Simulation Minute 4 (b) Simulation Minute 9 (c) Simulation Minute 14

Figure 5.13: *Gnutella* 0.4 Overlay Visualized by *PeerfactSim.KOM*



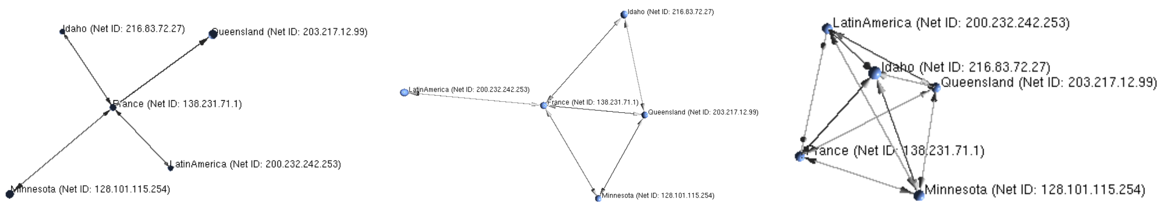
(a) Simulation Minute 1 (b) Simulation Minute 3 (c) Simulation Minute 7

Figure 5.14: *Gnutella* 0.4 Overlay Visualized by *Graphviz*



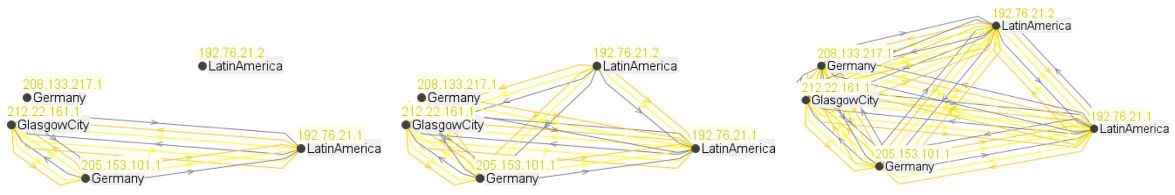
(a) Simulation Minute 1 (b) Simulation Minute 3 (c) Simulation Minute 7

Figure 5.15: *Gnutella* 0.4 Overlay Visualized by *UbiGraph*


Idaho (Net ID: 216.83.72.27)
Queensland (Net ID: 203.217.12.99)
France (Net ID: 138.231.71.1)
LatinAmerica (Net ID: 200.232.242.253)
Minnesota (Net ID: 128.101.115.254)

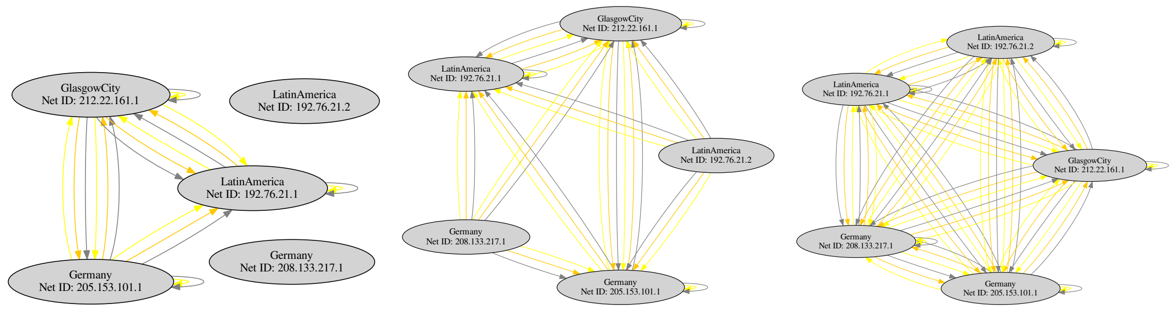
(a) Simulation Minute 1 (b) Simulation Minute 3 (c) Simulation Minute 7

Figure 5.16: *Pastry* Overlay Visualized by *PeerfactSim.KOM*



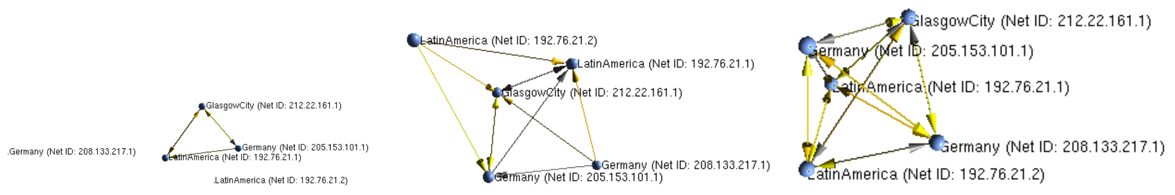
(a) Simulation Minute 8, Second 1 (b) Simulation Minute 8, Second 2 (c) Simulation Minute 8, Second 3

Figure 5.17: *Pastry* Overlay Visualized by *Graphviz*



(a) Simulation Minute 8, Second 1 (b) Simulation Minute 8, Second 2 (c) Simulation Minute 8, Second 3

Figure 5.18: *Pastry* Overlay Visualized by *UbiGraph*


GlasgowCity (Net ID: 212.22.161.1)
Germany (Net ID: 205.153.101.1)
LatinAmerica (Net ID: 192.76.21.1)
Germany (Net ID: 208.133.217.1)
LatinAmerica (Net ID: 192.76.21.2)

(a) Simulation Minute 8, Second 1 (b) Simulation Minute 8, Second 2 (c) Simulation Minute 8, Second 3

5.2 Comparison of Programs *PeerfactSim.KOM* and *db2vis*

As can be seen in Table 5.1 *PeerfactSim.KOM* and *db2vis* without the specific features for *Graphviz* and *UbiGraph* have comparable visualization capabilities.

Table 5.1: Comparison of Programs *PeerfactSim.KOM* and *db2vis*

Feature	***PeerfactSim.KOM***	***db2vis***
Show net IDs	Yes	Yes
Show overlay IDs	Yes	Yes
Show used overlays	Yes	Yes
Show overlays, class names	Yes	Yes
Show number of documents	Yes	Yes
Show number of neighbors	Yes	Yes
Show number of sent messages	No	Yes
Show number of received messages	No	Yes
Show number of messages per second	Yes	No
Show message edge labels	Yes	Yes
Filter for connection edges	Yes	Yes
Configurable connection edge colors	No	Yes
Filter for message edges	Yes	Yes
Configurable message edge colors	No	Yes
Scenario statistics	Yes	No
Dynamic node size	Yes	No
Dynamic edge size	Yes	No
Sum	13/17	13/17

5.3 Appropriate Visualization

The goal of any visualization is to present as much information as necessary but as little as possible. Related to peer-to-peer overlays it means visualizing many nodes, edges and attributes while keeping the graphics relatively compact and readable.
One challenge for example is to avoid overlapping objects in order to still recognize each individual one of them. Methods to bypass or at least reduce the amount of crossing edges are curves/splines instead of straight lines but also algorithms calculating optimal node positioning. *Graphviz* and *UbiGraph* offer both possibilities, users of *PeerfactSim.KOM* can at least switch between the two views.

As each *Graphviz* layout uses a specific algorithm, the user has several alternatives. Though *UbiGraph* users cannot influence the arrangement of the objects, they have the possibility of changing the point of view in the three-dimensional space to detect hidden nodes or edges.
An important factor in using the available drawing space as efficiently as possible is the minimization of redundant information. Therefore at best the node attributes are formatted in different colors while presenting their mapping in a single legend instead of mentioning the same attribute names in each node label. A similar method should be used to distinguish different edge types. *PeerfactSim.KOM* achieves that while still offering the possibility of labeling messages type-dependently due to their short appearance in the form of flashing edges. As already mentioned in the sections about the limitations of *Graphviz* and *UbiGraph* these tools do not seem to offer a straightforward way of showing a legend within the visualization or of using different font colors in the same label.

5.4 Scalability

Subsection 5.4.1 examines the performance impact on *PeerfactSim.KOM* by the database connection. The performance and disk space usage of *db2vis* are analyzed in Subsection 5.4.2, the ones of *Graphviz* in Subsection 5.4.3. As the *UbiGraph* performance mainly depends on the one of *db2vis*, the standalone performance of the former cannot be measured. Due to the fact that *UbiGraph* does not generate any output files, its disk space usage is limited to the actual program size and does not depend on inputs given.

All performance tests have been run on the two hardware configurations described in Table 5.2.

Table 5.2: Hardware Configurations Used for Performance Tests

Number	**CPU**	**RAM**	**SSD**
1	*Intel Core i7-3517U* (2 cores @ up to 3.0 GHz, 4 Threads)	4 GB DDR3 PC3-12800 CL11 (1,600 MHz, 25.6 GB/s)	*ADATA XM11*
2	*Intel Core i7-3770* (4 cores @ up to 3.9 GHz, 8 Threads)	8 GB DDR3 PC3-17000 CL9 (2,133 MHz, 34 GB/s)	*Samsung SSD 830*

5.4.1 *PeerfactSim.KOM*

As the particular database as well as its three tables are created and filled during the actual simulation, the database connection only influences the computation time before the visualization of *PeerfactSim.KOM*. Therefore the reaction time while using any visualization function is not affected.

To measure the performance impact during computation 24 simulations have been executed, each of the four overlays *Chord*, *Gnutella* 0.4, *Pastry* and *Parallel Pastry* for three times on two different hardware configurations. In Table 5.3 one can see the performance impact on the first machine. The relatively huge number of events in overlays *Pastry* and *Parallel Pastry* is caused by a bug of *PeerfactSim.KOM* that produces multiple *EdgeRemoved* events of edges which have been already removed.

Table 5.3: Performance Impact on Program *PeerfactSim.KOM* by Database Connection, Ex. 1

Measurement	**Without DB**	**With DB**	**Factor**
Chord, 11 Nodes, 15,766 events			
1	8 s	18 s	2.25
2	8 s	17 s	2.125
3	8 s	17 s	2.125
Average	8 s	17 s 333 ms	2.166
Gnutella 0.4, 14 Nodes, 15,108 events			
1	10 s	18 s	1.8
2	10 s	18 s	1.8
3	10 s	17 s	1.7
Average	10 s	17 s 666 ms	1.766
Pastry, 32 Nodes, 78,923 events			
1	6 s	51 s	8.5
2	6 s	51 s	8.5
3	6 s	51 s	8.5
Average	6 s	51 s	8.5
Parallel Pastry, 32 Nodes, 194,038 events			
1	6 s	1 m 58 s	19.66
2	6 s	1 m 58 s	19.66
3	7 s	1 m 58 s	16.86
Average	6 s 666 ms	1 m 58 s	17.7

Dividing the particular number of events by the average computation time when using the database connection results at each of the four overlay simulations in between round about 850 and 1650 database inserts per second.

One can see the correlation between the number of events (not nodes) and the impact of the database connection due to the fact that each event is inserted into the database.

The performance impact on the second machine can be seen in Table 5.4.

Table 5.4: Performance Impact on Program *PeerfactSim.KOM* by Database Connection, Ex. 2

Measurement	**Without DB**	**With DB**	**Factor**
Chord, 11 Nodes, 15,766 events			
1	4 s	55 s	13.75
2	4 s	54 s	13.5
3	4 s	53 s	13.25
Average	4 s	54 s	13.5
Gnutella 0.4, 14 Nodes, 15,108 events			
1	6 s	54 s	9
2	5 s	53 s	10.6
3	5 s	51 s	10.2
Average	5 s 333 ms	52 s 666 ms	9.875
Pastry, 32 Nodes, 78,923 events			
1	3 s	3 m 55 s	78.33
2	3 s	3 m 57 s	79
3	3 s	3 m 58 s	79.33
Average	3 s	3 m 56 s 666 ms	78.88
Parallel Pastry, 32 Nodes, 194,038 events			
1	4 s	9 m 26 s	141.5
2	3 s	9 m 30 s	190
3	3 s	9 m 29 s	189.66
Average	3 s 333 ms	9 m 28 s 333 ms	170.5

Dividing the particular number of events by the average computation time when using the database connection results at each of the four overlay simulations in between round about 285 and 340 database inserts per second.

The HDD/SSD used has to be capable of as many write operations per second as possible, as this is the limiting factor of the database performance at least in both hardware configurations tested. As can be seen from the two tables above, the CPU and RAM mainly affect the computation time without the database connection. Due to the better SSD performance of the first hardware configuration the impact by the database connection during simulations is much smaller than in the second one.
Other systems of course can have different bottlenecks - more than ever if *PeerfactSim.KOM* and the database do not run on the same machine. Then not only the two machines can hold limiting components but also the connection between them.

5.4.2 *db2vis* - Database to Visualization

Performance

Tables 5.5 (hardware configuration 1) and 5.6 (hardware configuration 2) compare the *db2vis* execution time used for creating logs, *gv* files and scripts for the *Graphviz* layouts *dot*, *neato*, *fdp*, *circo* and *twopi* of the four overlays shown in Tables 5.3 and 5.4. Each run was measured twice in order to exclude unusual punctual workload caused by other processes. Only a subset of all simulated and stored edge/message types was configured to be queried and visualized to avoid overloaded graphs.

Table 5.5: Execution Time of Program *db2vis*, Example 1

Overlay	**Nodes**	**Events**	**Measurement**	**Execution Time (in ms)**
Chord	11	176	1	391
			2	313
Gnutella 0.4	14	546	1	562
			2	468
Pastry	32	76,870	1	5,581
			2	5,456
Parallel Pastry	32	189,718	1	11,467
			2	11,290

Table 5.6: Execution Time of Program *db2vis*, Example 2

Overlay	**Nodes**	**Events**	**Measurement**	**Execution Time (in ms)**
Chord	11	176	1	313
			2	168
Gnutella 0.4	14	546	1	1,331
			2	296
Pastry	32	76,870	1	4,114
			2	6,037
Parallel Pastry	32	189,718	1	8,288
			2	8,383

As in Tables 5.3 and 5.4 the performance mainly depends on the number of events (not nodes). The simple reason is that *PeerfactSim.KOM* inserts all events into the database and *db2vis* queries each (user-specified) one of them.

The execution time of *db2vis* in combination with *UbiGraph* does not need to be measured for two main reasons:

1. As *db2vis* communicates directly with *UbiGraph* the performance of the former depends mainly on the performance of the latter - in contrast to the *Graphviz* functionality of *db2vis*, which prepares the *gv* files (and optionally the scripts) without any interaction with *Graphviz*. Instead *Graphviz* is executed afterwards and works on its own.

2. The purpose of *UbiGraph* is visualizing changes of a graph dynamically and in a reasonable speed to keep track of them. Therefore *UbiGraph* should not be used for getting the final state of a certain overlay as fast as possible. According to that users of *db2vis* can define the *UbiGraph* visualization speed, even separately for connection and message edges.

Storage

Table 5.7 compares the size of the *gv* and log files generated by *db2vis* for the *Graphviz* layouts *dot*, *neato*, *fdp*, *circo* and *twopi* of the four overlays shown in Tables 5.3 and 5.4.

Table 5.7: Size of Files Generated by Program *db2vis*

Overlay	**Nodes**	**Events**	**Average File Size (in KB)**		
			Script	*gv*	Log
Chord	11	176	3	10	273
Gnutella 0.4	14	546	3	28	677
Pastry	32	76,870	3	53	65,827
Parallel Pastry	32	189,718	3	58	159,872

As one can see the size of the generated script files does not depend on the number of nodes or events. Instead the relevant factors for their size are the number of *Graphviz* layouts, output formats and *gv* files as for each combination a script line has to be written. Due to the fact that all three parameters are identical in all four overlays shown in Table 5.7, the file size of the scripts does not change either. Each script contains 35 *Graphviz* commands: five layouts multiplied by seven output formats multiplied by one *gv* file. Should one configure *db2vis* to create a new *gv* file every *x* minutes or each time a specific event has occurred (see Table 4.1) the number of *gv* files would be increased.

5.4.3 *Graphviz*

Performance

Tables 5.8 (hardware configuration 1) and 5.9 (hardware configuration 2) compare the creation time of seven popular output file formats distributed over the five *Graphviz* layouts and separated by the four overlays simulated above. Within each overlay the layouts and output formats are sorted by average creation time in ascending order. Each layout output was measured twice in order to exclude unusual punctual workload caused by other processes.

The following attributes have been used:
Graph: Output edges first, scale in case of overlapping node shapes
Nodes: Filled shapes, label with location and net ID
Connection edges: Solid
Message edges: Dashed, label with message type

Table 5.8: Creation Time Comparison of Output Formats Offered by Program *Graphviz*, Ex. 1

Layout	**Measurement**	**Creation Time (in s)**						
		Vector Graphics		Raster Graphics				
		pdf	*svg*	*jpg*	*bmp*	*tif*	*gif*	*png*
Chord, 11 nodes, 85 connection edges, 40 message edges								
fdp	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
neato	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
twopi	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
circo	1	0	0	0	1	0	0	1
	2	0	0	0	0	0	0	1
dot	1	0	0	0	0	1	1	1
	2	0	0	0	0	1	1	1

Gnutella 0.4, 14 nodes, 166 connection edges, 176 message edges								
fdp	1	0	0	0	0	0	0	1
	2	0	0	0	0	0	0	1
neato	1	0	0	0	0	0	0	1
	2	0	0	0	0	0	0	1
twopi	1	0	0	0	0	0	0	1
	2	0	0	0	0	0	0	1
circo	1	0	0	1	1	1	1	1
	2	0	0	1	1	1	1	1
dot	1	3	4	5	5	5	6	7
	2	4	3	5	5	5	6	7
Pastry, 32 nodes, 630 connection edges, 229 message edges								
fdp	1	1	1	1	1	1	1	2
	2	1	1	1	1	1	1	2
neato	1	0	0	1	1	1	1	3
	2	0	0	1	1	1	1	3
twopi	1	0	0	1	2	2	2	4
	2	0	0	1	1	2	2	4
circo	1	34	34	36	36	37	37	39
	2	35	35	36	36	37	37	39
dot	1	83	86	92	99	97	95	103
	2	88	87	96	91	89	99	103
Parallel Pastry, 32 nodes, 772 connection edges, 96 message edges								
fdp	1	1	1	1	1	1	1	2
	2	1	1	1	1	1	1	2
neato	1	0	0	1	1	1	1	2
	2	0	0	1	1	1	1	2
twopi	1	0	0	1	1	1	2	3
	2	0	0	1	1	1	2	3
circo	1	18	18	19	20	20	20	22
	2	18	18	19	19	20	20	22
dot	1	108	102	117	114	112	113	120
	2	104	108	111	113	108	110	121

Table 5.9: Creation Time Comparison of Output Formats Offered by Program *Graphviz*, Ex. 2

Layout	**Measurement**	**Creation Time (in s)**						
		Vector Graphics		Raster Graphics				
		pdf	*svg*	*jpg*	*bmp*	*tif*	*gif*	*png*
Chord, 11 nodes, 85 connection edges, 40 message edges								
fdp	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
neato	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
twopi	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
circo	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
dot	1	0	0	0	0	0	0	1
	2	0	0	0	0	0	0	1
Gnutella 0.4, 14 nodes, 166 connection edges, 176 message edges								
fdp	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
neato	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
twopi	1	0	0	0	0	0	0	0
	2	0	0	0	0	0	0	0
circo	1	0	0	0	1	0	1	1
	2	0	0	0	1	0	1	1
dot	1	2	2	3	3	3	4	5
	2	3	3	3	3	4	4	5
Pastry, 32 nodes, 630 connection edges, 229 message edges								
fdp	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
neato	1	0	0	1	1	1	1	2
	2	0	0	1	1	1	1	2
twopi	1	0	0	1	1	1	1	3
	2	0	0	1	1	1	1	3
circo	1	26	26	27	27	28	28	30
	2	26	26	28	28	27	28	30
dot	1	50	50	51	58	53	58	57
	2	55	46	62	57	51	55	61

Parallel Pastry, 32 nodes, 772 connection edges, 96 message edges								
fdp	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
neato	1	0	0	1	1	1	1	2
	2	0	0	1	1	1	1	2
twopi	1	0	0	1	1	1	1	2
	2	0	0	1	1	1	1	2
circo	1	14	14	15	15	15	15	16
	2	14	13	14	15	15	15	16
dot	1	55	61	66	60	71	66	69
	2	63	73	71	67	67	74	75

As the creation time does not only consist of writing the data onto the HDD/SSD but also of computing the particular layout, both the output file formats and especially the layouts differ in some cases considerably in creation time within the same overlay.

Storage

The size of all generated output files can be compared in Table 5.10.

Table 5.10: Size Comparison of Output Formats Offered by Program *Graphviz*

Layout	**Measurement**	**File Size (in KB)**						
		Vector Graphics		Raster Graphics				
		pdf	*svg*	*gif*	*jpg*	*png*	*tif*	*bmp*
Chord, 11 nodes, 85 connection edges, 40 message edges								
neato	1	32	41	97	160	355	352	3,744
	2	32	41	97	160	355	352	3,744
fdp	1	32	41	112	186	386	407	4,574
	2	33	41	117	172	471	323	5,582
twopi	1	36	56	117	195	444	448	4,127
	2	36	56	117	195	444	448	4,127
circo	1	36	56	221	354	821	806	9,071
	2	36	56	221	354	821	806	9,071
dot	1	42	70	253	458	857	881	20,785
	2	42	70	251	472	804	884	20,282

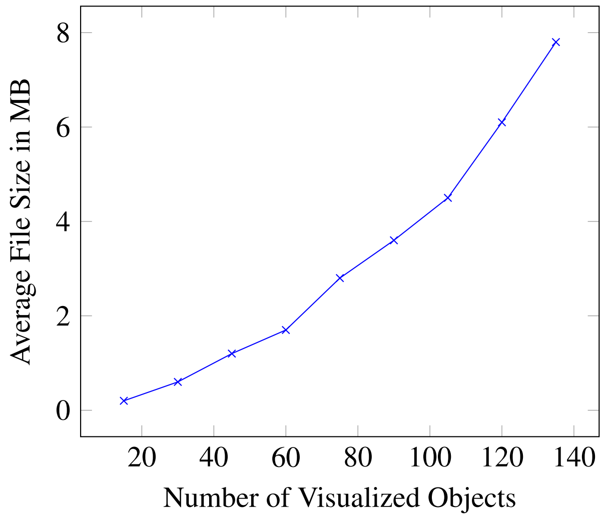
Gnutella 0.4, 14 nodes, 166 connection edges, 176 message edges								
fdp	1	40	115	292	500	883	914	4,139
	2	40	115	263	336	736	667	4,649
neato	1	40	115	394	513	1,239	1,201	7,883
	2	40	115	394	513	1,239	1,201	7,883
twopi	1	49	154	424	513	1,319	1,292	8,241
	2	49	154	424	513	1,319	1,292	8,241
circo	1	50	154	631	856	2,075	1,984	13,078
	2	50	154	631	856	2,075	1,984	13,078
dot	1	84	236	1,270	2,418	4,013	3,762	81,818
	2	88	240	1,322	2,365	3,983	3,940	78,471
Pastry, 32 nodes, 630 connection edges, 229 message edges								
fdp	1	56	236	627	698	2,089	2,613	13,826
	2	56	236	593	679	2,107	2,411	14,818
neato	1	57	236	1,464	1,673	5,267	6,163	38,429
	2	57	236	1,464	1,673	5,267	6,163	38,429
twopi	1	82	336	1,936	2,154	7,058	8,500	34,618
	2	82	336	1,936	2,154	7,058	8,500	34,618
circo	1	82	336	2,786	3,232	10,101	12,132	61,840
	2	82	336	2,786	3,232	10,101	12,132	61,840
dot	1	177	541	4,814	8,130	17,140	20,897	306,124
	2	168	537	4,661	8,075	18,161	20,992	299,606
Parallel Pastry, 32 nodes, 772 connection edges, 96 message edges								
fdp	1	58	254	661	791	2,299	2,760	15,298
	2	58	254	661	800	2,204	2,535	14,271
neato	1	59	254	1,381	1,935	5,150	5,893	42,744
	2	59	254	1,381	1,935	5,150	5,893	42,744
twopi	1	84	353	1,839	2,341	6,986	8,340	35,475
	2	84	353	1,839	2,341	6,986	8,340	35,475
circo	1	84	353	2,262	3,168	8,367	10,241	61,387
	2	84	353	2,262	3,168	8,367	10,241	61,387
dot	1	176	575	4,510	9,039	16,545	19,996	356,650
	2	191	579	4,526	9,037	17,035	20,463	352,379

As one can see the file size varies highly within the same overlay, not only between the output formats but also between the layouts. It stands out that unlike the other three overlays the *Chord* one tested is most efficiently visualized by *Graphviz* layout *neato*. Apart from that the nondeterministic file size of the *fdp* and *dot* outputs attracts attention. The order of disk space usage between the output formats *png* and *tif* does not only differ between the overlays and their layouts, but also between both measurements within the same layout of the same overlay as can be seen in the *fdp* row of the *Chord* overlay.

As vector graphics typically can be saved much more efficiently than raster ones and offer lossless scalability as well as searchable/copyable text fields, they are most suitable for the visualizing purpose of *db2vis* in combination with *Graphviz*. In addition *svg* files can be analyzed or customized textually. It has to be mentioned that although the file size between vector and raster graphics differs highly within the same layout of the same overlay, generating both graphic types takes nearly the same time as can be seen in Tables 5.8 and 5.9. Most of the creation time is spent by computing the particular layout and this does not depend on the file format.

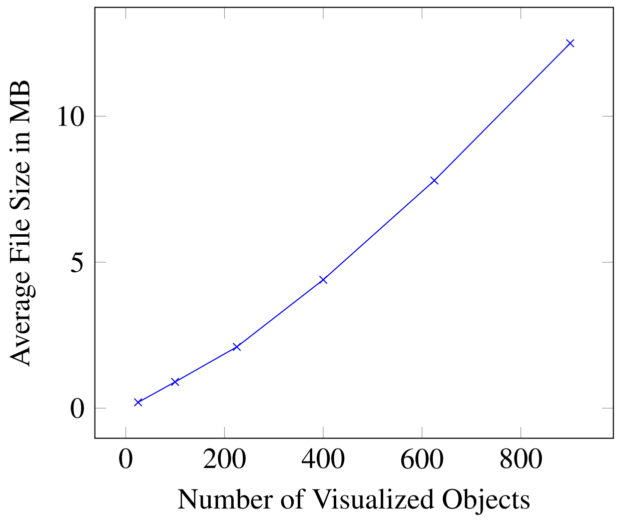
One can see in Figures 5.19, 5.20 and 5.21 that the average file size of the *Graphviz* output formats usually scales super-linearly in relation to the number of visualized objects. The latter was increased several times within the particular overlay. Each time the graph was visualized as *pdf*, *svg*, *gif*, *jpg*, *png*, *tif* and *bmp* by the *Graphviz* layouts *fdp*, *neato*, *dot*, *twopi* and *circo*. The average size of all generated files was calculated separately for each visualization run.

Figure 5.19: File Size of *Graphviz* Output Formats in Relation to Number of Objects, Example 1



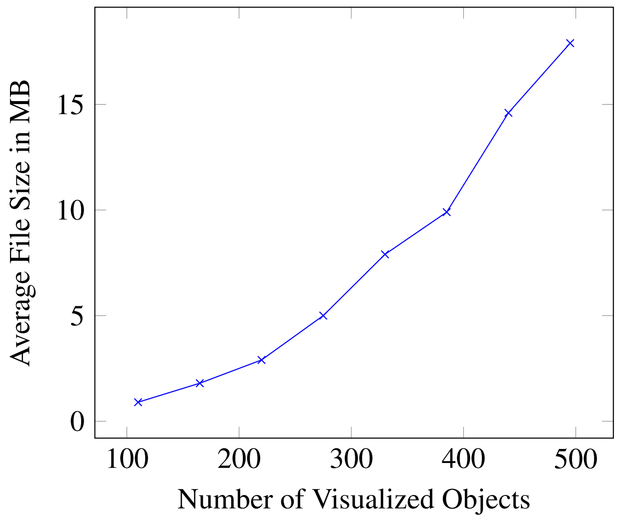
Nine *Chord* rings with their nodes as well as *successor* and *predecessor* edges were visualized. Each visualization contained n nodes, $2n$ edges and therefore $3n$ objects. n started with 5 and was increased by 5 each run.

Figure 5.20: File Size of *Graphviz* Output Formats in Relation to Number of Objects, Example 2



Six *Gnutella* 0.4 overlays with their nodes and *routing table entry* edges were visualized. Each visualization contained n nodes, $n^2 - n$ edges and therefore n^2 objects. n started with 5 and was increased by 5 each run.

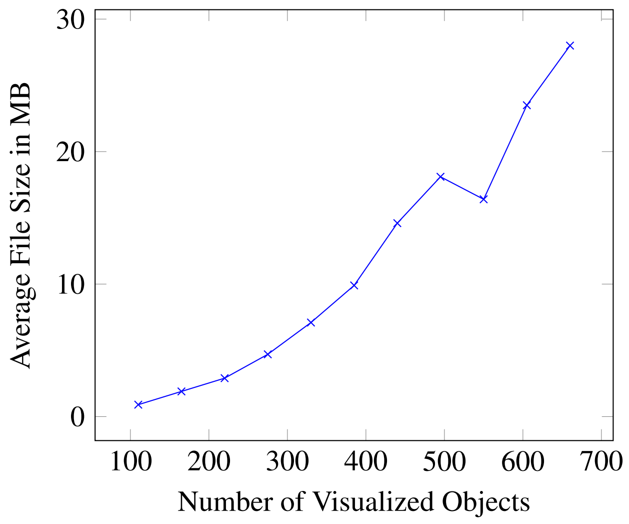
Figure 5.21: File Size of *Graphviz* Output Formats in Relation to Number of Objects, Example 3



Eight *Pastry* overlays with their nodes and *leaf set entry* edges were visualized. Each visualization contained n nodes, $10n$ edges and therefore $11n$ objects. n started with 10 and was increased by 5 each run.

As can be seen from Figure 5.22 sometimes a higher number of visualized objects can even produce smaller graphics depending on the particular node arrangement. The latter influences the compactness of the graph. The nondeterministic file size caused by the *Graphviz* layouts *fdp* and *dot* as described further above is explicitly not responsible for this exception. On the one hand the difference of 62 MB between the overall (not average) output size of the eighth and ninth run would be too big, on the other hand only the *twopi* outputs were affected due to its specific algorithm. Visualizing the ninth simulation a second time produces the same effect.

Figure 5.22: File Size of *Graphviz* Output Formats in Relation to Number of Objects, Example 4



Eleven *Parallel Pastry* overlays with their nodes and *leaf set entry* edges were visualized. Each visualization contained n nodes, $10n$ edges and therefore $11n$ objects. n started with 10 and was increased by 5 each run.

Chapter 6

Conclusions

In this chapter the conclusions of this thesis are presented. Section 6.1 summarizes the results, open questions are discussed in Section 6.2 and eventually Section 6.3 provides ideas for possible future work on *db2vis* and its database input.

6.1 Results

In summary, one can say that extending *PeerfactSim.KOM* with a database connection on the one hand helps textually analyzing the simulation but on the other hand slows down the process of computing the simulation (see Subsection 5.4.1).
Once the events are inserted into the database, they can be visualized as often and as customized as desired with the help of *db2vis*. Due to its extensive configuration possibilities users can define required time windows of the simulation, select node metrics and edges to be shown et cetera (see Table 4.1).

Thanks to the support of both visualization tools *Graphviz* and *UbiGraph* it is now possible to set a lot of graph, node and edge attributes in order to improve the usefulness of the created diagrams. As the two programs offer very different kinds of visualization, the same simulation can be examined from various perspectives with just one execution of *db2vis*.

6.2 Open Questions

Although support for fractional seconds in the *MySQL* data type *TIME* has already been introduced in version 5.6.4 released on 2011/12/20, method *getTime* of package *com.mysql.jdbc.ResultSetImpl* still cuts them off when the *time* column is given as argument. Using method *getString* of the same package throws the *SQL* exception "Bad format for Time".
The only possible workaround in *db2vis* was to cast the *time* field as binary in the *SELECT* query before reading it with *getString*. In contrast creating the database table field *time TIME(3)* and inserting times with fractional seconds out of *PeerfactSim.KOM* was possible without any problems.

Since *java.sql.Time* objects in contrast to *java.sql.Timestamp* ones still do not support fractional seconds, the latter ones had to be used in *db2vis*. As the *Timestamp* constructor does not only expect a time but also a date, each object was created with the dummy date 1970-1-1. In each output of *db2vis* only the actual simulation time is shown, since it is not a time of day, but a time span.

In rare cases *tif* files generated by *Graphviz* are empty. This phenomenon occurred for example when the number of objects in the evaluation described in Figure 5.20 was further increased to 1,225 or more while using *Graphviz* layout *dot*. Surprisingly increasing the number of objects in the evaluation described in Figure 5.21 to 550 while using *Graphviz* layout *twopi* caused the same problem, whereas increasing it further to 605 objects solved it.

6.3 Future Work

To reduce the size of log files when visualizing huge simulations with the help of *db2vis* (see Table 5.7) different levels of logging could be provided. Levels only logging warning and/or error messages would be suitable as well as ones which only log specific events or exclusively *Graphviz* / *UbiGraph* actions.

Overlay-specific node attributes like the number of so-called *fingers* when using *Chord* or the number of so-called *leaves* and *RT neighbors* when using *Pastry* would also be an improvement of *db2vis*. To realize such a feature the database design would have to be changed as there are no according fields in the *node* table yet (see Table 3.2). This could not be implemented by simply adding a fixed number of columns as each overlay contains a different amount of specific node attributes. In consequence *db2vis* would also have to act flexibly according to the particular overlay used.

In order to interact with *db2vis* a graphical user interface (GUI) could be offered. With its help the user should at least be able to pause/stop the creation of *gv* files as well as the visualization within *UbiGraph*. In addition the possibility of rewinding and fast-forwarding the latter interactively as offered by *PeerfactSim.KOM* (see Subsection 5.1.1) could be an option.

If still no way can be found to place a legend within the *Graphviz* and *UbiGraph* visualization (see Subsections 5.1.2 and 5.1.3), *db2vis* could generate one in the form of a separate graphics file.

To satisfy the specific needs of storing overlays a graph database could be used as their data structure consists of nodes, edges and attributes - instead of tables as is the case in classic relational databases. Graph databases provide specialized algorithms to simplify complex queries. They offer for example algorithms for finding all direct and indirect neighbors of a node, calculating the shortest path between two nodes or identifying hot spots of exceptionally high-meshed graph regions. *Neo4j* [Neo14] would probably be suitable as it is open source, written in *Java* and therefore cross-platform-compatible.

To improve the *Graphviz* performance in case of generating multiple output formats (see Subsubsection *Performance* of Subsection 5.4.3), it could be investigated if they can be passed as arguments of one single command line instead of using a new command for each output format (see Subsubsection *Graphviz* of Subsection 3.2.3). Of course it would have to be analyzed if this avoids computing the same layout again and again for each output format as is the case at the moment.
Another approach to accelerating the output process of *Graphviz* could be parallelizing it by creating and executing multiple script files. The latter could be for example separated by layouts. Multiprocessor respectively multi-core systems would benefit from this as they no longer have to sequentially compute the different layouts.

Eventually *db2vis* might support additional visualization tools in order to provide further alternatives to *Graphviz* and *UbiGraph*.

Bibliography

[Can14] CANONICAL: *The leading OS for PC, tablet, phone and cloud | Ubuntu*. http://www.ubuntu.com/, 2014.

[Dau04] DAUM, Berthold: Java-Entwicklung mit Eclipse 3. In: *Heidelberg: dpunkt. verlag* (2004).

[Ecl14] ECLIPSE FOUNDATION, INC.: *Eclipse - The Eclipse Foundation open source community website*. http://www.eclipse.org/, 2014.

[EGK+02] ELLSON, John; GANSNER, Emden; KOUTSOFIOS, Lefteris; NORTH, StephenC.; WOODHULL, Gordon: Graphviz— Open Source Graph Drawing Tools. Version: 2002. http://dx.doi.org/10.1007/3-540-45848-4_57. In: MUTZEL, Petra (Hrsg.); JÜNGER, Michael (Hrsg.); LEIPERT, Sebastian (Hrsg.): *Graph Drawing* Bd. 2265. Springer Berlin Heidelberg, 2002. DOI 10.1007/3–540–45848–4_57. ISBN 978–3–540–43309–5, 483-484.

[EGK+04] ELLSON, John; GANSNER, EmdenR.; KOUTSOFIOS, Eleftherios; NORTH, StephenC.; WOODHULL, Gordon: Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools. Version: 2004. http://dx.doi.org/10.1007/978-3-642-18638-7_6. In: JÜNGER, Michael (Hrsg.); MUTZEL, Petra (Hrsg.): *Graph Drawing Software*. Springer Berlin Heidelberg, 2004 (Mathematics and Visualization). DOI 10.1007/978–3–642–18638–7_6. ISBN 978–3–642–62214–4, 127-148.

[Fre14] FREE SOFTWARE FOUNDATION: *Bash - GNU Project - Free Software Foundation*. https://www.gnu.org/software/bash/, 2014.

[GLS+10] GROSS, C.; LEHN, M.; STINGL, D.; KOVACEVIC, A.; BUCHMANN, A.; STEINMETZ, R.: Towards a Common Interface for Overlay Network Simulators. In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, 2010. ISSN 1521–9097, S. 59–66.

[Gos00] GOSLING, James: *The Java language specification*. Addison-Wesley Professional, 2000

[Gra11] GRAFFI, K.: PeerfactSim.KOM: A P2P system simulator; Experiences and lessons learned. In: *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*, 2011. ISSN 2161–3559, S. 154–155.

[Gra14] GRAPHVIZ: *Graphviz | Graphviz - Graph Visualization Software*. `http://www.graphviz.org/`, 2014.

[Hei14] HEINRICH-HEINE-UNIVERSITÄT DÜSSELDORF: *Universität Düsseldorf: Startseite*. `http://www.uni-duesseldorf.de/`, 2014.

[Lab14] LABS, AT&T INC.: *AT&T Labs Fosters Innovative Technology | AT&T Labs*. `http://www.att.com/labs/`, 2014.

[Mic14] MICROSOFT CORPORATION: *Windows - Microsoft Windows*. `http://windows.microsoft.com/de-DE/windows/home`, 2014.

[Neo14] NEO TECHNOLOGY: *Neo4j - The World's Leading Graph Database*. `http://www.neo4j.org/`, 2014.

[Ora14a] ORACLE: *Oracle | Hardware and Software, Engineered to Work Together*. `http://www.oracle.com/`, 2014.

[Ora14b] ORACLE CORPORATION: *Java SE Technologies - Database*. `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`, 2014.

[Ora14c] ORACLE CORPORATION: *Java Software | Oracle*. `https://www.oracle.com/java/index.html`, 2014.

[Ora14d] ORACLE CORPORATION: *MySQL :: MySQL Connector/J Developer Guide*. `https://dev.mysql.com/doc/connector-j/en/`, 2014.

[Ora14e] ORACLE CORPORATION: *MySQL :: The world's most popular open source database*. `http://www.mysql.com`, 2014.

[Ora14f] ORACLE CORPORATION: *Oracle and Sun Microsystems | Strategic Acquisitions*. `http://www.oracle.com/us/sun/`, 2014.

[Pee14] PEERFACTSIM.KOM: *PeerfactSIM.KOM*. `https://sites.google.com/site/peerfactsimkom/`, 2014.

[Saf14] SAFARI BOOKS ONLINE: *What Is MySQL AB? - MySQL Reference Man-*

ual. https://www.safaribooksonline.com/library/view/mysql-reference-manual/0596002653/ch01s03.html, 2014.

[Tec14] TECHNISCHE UNIVERSITÄT DARMSTADT: *Home – Technische Universität Darmstadt*. http://www.tu-darmstadt.de/, 2014.

[The14] THE OPEN GROUP: *The UNIX System, UNIX System*. http://www.unix.org/, 2014.

[Tho14] THOMAS WILLIAMS, COLIN KELLEY, RUSSELL LANG, DAVE KOTZ, JOHN CAMPBELL, GERSHON ELBER, ALEXANDER WOO ET AL.: *gnuplot homepage*. http://www.gnuplot.info/, 2014.

[Ubi14a] UBIETY LAB, INC.: *Ubiety Lab, Inc.* http://ubietylab.net/, 2014.

[Ubi14b] UBIETY LAB, INC.: *Ubigraph: Free dynamic graph visualization software*. http://ubietylab.net/ubigraph/index.html, 2014.

[Uni14] UNIVERSITÄT PADERBORN: *Universität Paderborn*. http://www.uni-paderborn.de/, 2014.

[Vel07] VELDHUIZEN, Todd: *UbiGraph: Free dynamic graph visualization software*. 2007.

[WA02] WIDENIUS, Michael; AXMARK, David: *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.", 2002

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 21. Oktober 2014 Kolja Salewski

This DVD contains:

- A *pdf* version of this bachelor thesis
- All LaTeXand graphic files that have been used, as well as the corresponding scripts
- The source code of the software that was created during the bachelor thesis
- The measurement data that was created during the evaluation
- The referenced websites and papers
- A manual for *db2vis* describing its requirements, installation, configuration, execution and outputs
- The software required for *db2vis*