MASARYK UNIVERSITY
FACULTY OF INFORMATICS

Implementation of Open-Source Solver for Media Streams Planning Problem

BACHELOR'S THESIS

András Ürge

Brno, autumn 2013

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

András Ürge

Advisor: RNDr. Pavel Troubil

Acknowledgement

I would like to thank my supervisor RNDr. Pavel Troubil for his continuous support, guidance and valuable feedback which helped me to write this thesis.

Abstract

Advanced collaborative environments need to transfer multimedia data streams with bandwidths comparable to the capacities of network links. The problem of finding the right distribution paths with minimal transmission latency has been called the media streams planning problem (MSPP). The aim of this work is to present an implementation of a scheduler for the problem using a freely distributable mixed integer linear programming (MIP) solver. The performance of the implementation is described, and the results are compared with a scheduler that uses the proprietary MIP solver Gurobi. In overall Gurobi manages to plan the data routes faster, but the freely distributable MIP solver's performance is still sufficient for the use in typical collaborative environments.

Keywords

Media streams planning problem, mixed integer linear programming, CoUniverse

Contents

Chapter 1

Introduction

Modern computer networks allowing high-bandwidth data transmissions are becoming more and more common nowadays. The widespread of these networks enabled the development of many new collaborative applications requiring high-quality video for transmission. As an example we could mention remote lectures followed from different places or videoconferences of multiple users communicating in real-time. These kinds of applications usually need to transfer data streams from one or more sources to multiple receivers, and they very often require bandwidths comparable to the capacities of network links.

Managing collaborative environments and finding correct, latency-minimal distribution paths for high-bandwidth data streams is a non-trivial and complicated task. The applications have to establish communication in real-time and need to react to the network environment changes without loss of performance. For the automatization of these processes the framework CoUniverse was created. CoUniverse is a framework used for building real-time self-organizing collaborative environments that can work with high-bandwidth media applications on high-speed networks [1].

Due to the amount of transferred data, traditional routing mechanisms, like hop-by-hop routing, may not be sufficient to find an optimal or even correct solution. Therefore, we need more sophisticated methods to schedule media streams, which take into account transmission requirements as well as link capacities. The problem of choosing the right distribution paths in the network has been formally defined as the media streams planning problem (MSPP) [2]. The given network contains a set of data sources, which need to transmit their data to a predefined set of media applications. The MSPP can be then considered as a problem of finding a distribution tree for every data source, while the overall transmission latency of the transmitted streams stays as minimal as possible.

One of the possible approaches to find a solution for the MSPP is to use the methods of mixed integer linear programming (MIP) [2]. MIP is a linear programming optimization method in which some of the variables need to take integer values. It tries to minimize or maximize an objective function subject to some given constraints. These constraints are linear inequalities representing restrictions on the given problem. For example, in MSPP they could represent the limitations of the media applications and the properties of the network. The objective function expresses the goal of the problem, in our case it would be the minimization of the overall transmission latency. The constraints, the objective function and the variables together form an integer programming model.

A solver for the MSPP based on the integer programming model from the works [2, 3] has been already implemented in the CoUniverse. It uses a MIP proprietary solver called Gurobi to solve the created model. Unfortunately, Gurobi's license does not allow it to be freely distributable with the rest of the software.

The aim of this work was to implement a solver for the MSPP using a freely distributable MIP solver. First, I had to choose a suitable non-commercial MIP solver with the best possible performance. Afterwards, I had to make the necessary changes, so that the chosen MIP solver could be used for solving the MSPP problem instead of the commercial solver Gurobi. The final part of the work was to measure and analyse the performance of the implemented MSPP solver and compare the results with the performance of the actual implementation.

Chapter 2
Problem Description

In this chapter we are going to introduce the concept of media streams planning problem (MSPP), describe the structure of the network representation and define the basic notations, which will be used throughout the whole thesis.

2.1 Media Streams Planning Problem

Since traditional network routing technologies may lead to network congestion, we need different planning methods when dealing with high-bandwidth applications. The problem of finding the best routes for the transmission of high-bandwidth data streams has been called the MSPP [4]. Since a data source can have more than one destination for its data delivery, it is possible that multiple copies of the stream have to be created. Therefore the objective of the MSPP may be considered as computing a distribution tree for all of the data sources and their corresponding destinations. Usually the transmission of all of the streams takes place at the same time and from the point of enjoyability it is very important that the streams have the smallest possible latency. Another important requirement is that the planning should be done quickly and the scheduler should be able to react to the changes in the network in real-time.

2.2 Network Representation

For being able to describe the problem more precisely, we are going to introduce some formal notations on the basis of the given network. Actually by the term *network* we do not necessarily mean the real physical network. In previous approaches for finding a solution to the MSPP only the application level knowledge of the network was used, hiding the underlying physical network infrastructure, likewise in P2P applications [2]. In this kind of representation the links between the network nodes represent the end-to-end connections of the data processing applications. The main advantage of this principle is that the end users are able to use the planning methods even if no information about the actual physical network topology is available. The only problem with this kind of representation is that multiple end-to-end links may share the same physical network links, what can change the properties of the links in the application level. Especially the capacity of the given link can be influenced significantly, by transmitting data on another link which shares some of its physical links. This can lead to a network congestion, which needs to be detected. For that reason CoUniverse has to monitor the links periodically. If a congestion is detected, then the scheduler needs to invoke a replanning and send the data streams accordingly.

There are some situations, however, when the user might have some information about the underlying physical network topology. For example, he may know that during a communication where the participating sides reside on different continents, all of the streams need to be sent through a single sub-oceanic link. If this kind of knowledge is available, it is possible to use more advanced planning methods which could avoid possible congestions in the network and decrease the number of required replannings.

A new kind of approach for planning, which considers some part of the physical network to be available, has been already implemented in the CoUniverse [3]. This new approach combines the knowledge of the network on both levels, the one in the application level as well as the one in the physical level, into one network representation. This way we are able to represent networks even if there is no information about their underlying physical topology or if part of the topology is missing. Now we are going to describe the main entities of the network and introduce their formal definitions and notations.

2.2.1 Network Nodes and Links

Probably the most common way of describing a network topology is by a graph. A graph is defined by a set of vertices and a set of edges. The vertices represent the actual nodes in the network and the edges represent the corresponding links between the nodes. Since we consider a *partially known* network topology, we need three kinds of network nodes and two kinds of links. The set V_E represents the *endpoint network nodes*, which are the actual servers and desktop computers running the data processing media applications. The second type of network nodes are the *physical network nodes* represented by the set V_H. They include the entities on the lower layers of the ISO OSI model, like routers and switches. The last set V_U of the *unknown network nodes* represents the part of the network, where the actual physical network topology is not known.

Because between some of the nodes we may need to transmit data in both directions, we consider a directed graph with directed links. There are two kinds of links connecting the network nodes. One of them is represented by the set L_H of *physical links* containing all the existing connections on the network and link layers of the ISO OSI model. Basically they can connect any kinds of network nodes, but endpoint network nodes are typically not connected directly. We consider only one property of a physical link $h \in L_H$, its capacity, denoted as *cap(h)*. Usually there is no information about the latency of a physical link and also it is not easy to measure, so we do not consider it as a property.

The other set L_L of *logical links* on the other hand can connect only the endpoint network nodes. For a proper definition, we first need to define network interfaces. Let the set I be the set of all *network interfaces*. On each of the endpoint nodes $v \in V_E$ there is at least one interface. If the interface *i* is configured on the node v, we denote it by $i \in v$. Every interface has an important property, the transmission capacity *capI(i)*. The main purpose of the interfaces is to connect somehow the particular endpoint nodes on the application level of the network. This is achieved with the help of *subnetworks*. If an interface $i \in v$ belongs to a subnetwork *s*, then the node v has access to the physical subnetwork s. Each interface must belong to exactly one subnetwork. Two network nodes connected to the same subnetwork (through some of their interface) are considered mutually reachable.

Now we can describe, how the logical links interconnect the endpoint network nodes. Each logical link is defined as a directed end-to-end link which connects two interfaces belonging to the same subnetwork. These interfaces are not allowed to be configured at the same node. Although logical links formally connect the interfaces, we usually speak about them as they connect nodes. A logical link connects the nodes v_1, v_2 if and only if it connects the interfaces i_1, i_2 and $i_1 \in v_1 \wedge i_2 \in v_2$. We denote the connected nodes of the link *l* as *begin(l)* and *end(l)*, depending on the direction of the link. Also every logical link *l* has two important attributes. The first is the maximum capacity, denoted as *cap(l)*, which is determined by the capacity of the endpoint interfaces of the link or by network monitoring. The other attribute is the transfer latency of the link, denoted as *latency(l)*. The latencies of the links are determined only by the help of network monitoring.

It is important to mention that the set of logical and physical links are related to each other. Each physical link *h* is related to a set *logical(h)* which contains logical links that use the link *h* during their data transmissions. Similarly, each logical link *l* is related to a set *physical(l)* which contains physical links representing the path between the endpoint nodes of the link *l*.

2.2.2 Media Applications

Media applications are the applications running on the endpoint network nodes. All of the applications need to processes a given *media stream*. Let *S* be the set of all streams in the given network topology. Every stream $s \in S$ has a bandwidth required for the transmission of the stream, denoted as *bw(s)*.

We can distinguish three kinds of media applications. The *producer* $p \in P$ creates a data stream and sends it somewhere further in the network. The *distributor* $d \in D$ receives the data, creates multiple copies of it and transmits the copies to some other network applications. It is also possible for the distributors, to not create any copies of the received data, but just simply forward the received stream directly to one of the other applications. Finally, the *consumer* $c \in C$ just receives the stream from the network. All of these applications are capable of processing at most one data stream, so none of them can produce, distribute or consume more than one stream. Each stream s is produced by a producer, denoted as $producer(s) \in P$ and is delivered to set of consumers $consumers(s) \subseteq C$. It is important that the stream has to be produced by exactly one producer and it needs to have at least one consumer.

For every application $a \in P \cup D \cup C$ we define the set $in_H(a)$, which is a set of all physical links that end in the endpoint node which runs the application *a*. The set $out_H(a)$ is defined similarly for the outgoing physical network links.

There are some restrictions regarding what kinds of applications can run together on a given network node. The first rule is, if there is a distributor on the given node, then no other applications can run there. Consumers and producers are allowed to run on the same node, but all of them have to process a different media stream. This means, that if a consumer receives a stream *s*, then no other media application on the same node is allowed to produce or consume the same stream *s*. Although more media applications are allowed to reside on a single network node, it is not typical, in practise usually every application runs on a different network node due to their high requirements on computational resources.

Despite the fact that multiple copies of a given stream can be created, all of the packets need to be sent from a producer to a single consumer along the same path. Therefore, distributors are not allowed to split the stream to smaller streams and send them through different network links. If it was allowed, then packet reordering could occur on the receiving node, which could have negative affection on the performance.

Now we can describe the MSPP a little bit more precisely. Every stream $s \in S$ needs to be transferred from a producer to its consumers by a distribution tree having the producer in the root, the consumers at the leaves and the distributors in the internal nodes of the tree. Each distribution tree has to satisfy some restrictions, depending on the type of the used media applications and on the capacities of the network links inside the network. Also, the transmission of every media stream has to occur at the same time and their overall transmission latency needs to be as minimal as possible.

Chapter 3

Mixed Integer Linear Programming

3.1 Linear Programming

Linear programming (LP) is a mathematical method that is used to solve optimization problems. Its aim is to find an optimal solution to a linear *objective function* under a given set of linear constraints. By finding an optimal solution we usually mean to find the minimum or maximum value of the objective function. There are many practical problems that can be solved by LP techniques. Some of the most typical are planning problems, like farm planning or food manufacturing, telecommunication tasks, and production deployment problems [5].

The main elements of every linear program are:

- **Decision variables.** At the beginning their value is not known. The aim of LP is to find values for them which will provide optimal solution for the objective function.

- **Objective function.** A linear function that combines the variables to express our goal. Its value is required to be minimized or maximized.

- **Constraints** are linear inequalities combined from the decision variables. They represent the restrictions that the solution of the optimization problem is required to satisfy.

- **Variable bounds** determine the intervals in which the variables are allowed to take a value.

More formally a linear programming model with m variables and n constraints can be written as:

$$
\begin{array}{lll}
\text{Minimize} & \displaystyle\sum_{j=1}^{m} c_j x_j & \\
\text{subject to} & \displaystyle\sum_{j=1}^{m} a_{ij} x_j \leq b_i, & i\text{=}1...n \\
 & l_j \leq x_j \leq u_j, & j\text{=}1...m
\end{array}
$$

where $x_1, x_2, ..., x_m$ are decision variables, $c_1, c_2, ..., c_m$ are objective coefficients, $a_{11}, a_{12}, ..., a_{nm}$ are constraint coefficients, $l_1, l_2, ..., l_m$ are lower bounds of variables and $u_1, u_2, ..., u_m$ are upper bounds of variables. Note that in case we would like to maximize the objective function it would be enough to negate the value of all of the objective coefficients c_j.

A *feasible solution* of a LP problem is a valuation of the decision variables which satisfies all the linear constraints, while the variables remain within their bounds. An *optimal solution* is a feasible solution that minimizes or maximizes the value of the objective function. An optimal solution, however, does not necessarily exist. The problem can contain a set of constraints that can not be satisfied at the same time. Some unbounded variables may also cause that for any existing feasible solution another feasible solution can be found which further improves the objective function.

Many algorithms exist that are capable to solve LP problems in a reasonable amount of time. As an example we can mention the *primal simplex*, *dual simplex*, or various *interior point methods* [6]. Probably the most used of them are the variants of the *simplex method*, but the actual performance of the algorithms heavily depends on the type of the given problem [7].

Some of the variables in LP may require to take integer values. In this case we talk about *mixed integer linear programming* (MIP). If all of the variables have to be integers, then the problem is called *integer programming*. As we can see in [6], this seemingly minor change makes the whole problem much more difficult to solve. LP problems can have a totally different solution if some of their variables are changed to integers. There are many methods for solving MIP problems, but their performance is highly dependent on the type of the particular problem [6].

3.2 Integer Programming Model

Now we are going to introduce the integer programming model of the MSPP. It is important to note that this model has already been proposed and all the information about it is processed from [3]. The model considers a partially known network topology. It consists of an objective function, some linear constraints and two kinds of integer variables.

3.2.1 Elimination process

First of all, it is important to note that not all of the network links inside the network are used to create the actual integer programming model. There are many links which are unusable, and they are eliminated from the network before creating the model. A network link is considered unusable if it has too small capacity for the transmission of any of the media streams, or if it has inappropriate media application on some of its endpoint nodes. For example a link with a consumer on its beginning node, or with a producer on its ending node, will be eliminated. Therefore, when we talk about the sets L_H and L_L below, we consider only the links which remained inside the graph after the elimination process.

3.2.2 Streamlinks

Streamlinks represent the actual integer variables in the model. As we mentioned before, there are two types of variables. The first of them are the *physical streamlinks* $y_{s,h}$ for every pair of stream $s \in S$ and physical link $h \in L_H$. A physical stream link $y_{s,h}$ represents the number of copies of the stream s sent through the physical link h. The maximum value it can take is the number of consumers of the stream s, but only if link h has an appropriate bandwidth for such a transmission. Otherwise, it is the maximum number of copies of the stream that can be transmitted through the link at the same time. More formally:

$$0 \leq y_{s,h} \leq \min\left(\|consumers(s)\|, \left\lfloor \frac{cap(h)}{bw(s)} \right\rfloor \right) \qquad \forall s \in S \; \forall h \in L_H$$

The other type of variables are the *logical streamlinks*. For each pair of stream $s \in S$ and logical link $l \in L_L$ there is a logical streamlink $x_{s,l}$. Its value determines whether the link l transmits the stream s. Since logical links are allowed to transmit only one stream at a time, logical streamlinks can have only two possible values: one or zero.

3.2.3 Constraints

As we already mentioned in Chapter 2, there is a relation between logical and physical links. For every stream s transmitted through the physical link h has to exist a logical link $l \in logical(h)$ which carries the same stream s. The constraint (3.1) defines this relation.

$$y_{s,h} = \sum_{l \in logical(h)} x_{s,l} \qquad \forall s \in S \; \forall h \in L_H \tag{3.1}$$

The total bandwidths of the streams transmitted through a given link can not exceed the capacity of the link. We assume that the capacities of the logical links can be measured by network monitoring. On the contrary physical link capacities do not necessarily have to be available. Let $L'_H \subseteq L_H$ be the set of physical links with a known capacity. Then, the constraints (3.2) and (3.3) guarantee that the capacities of the links are not exceeded by the transmitted streams.

$$\sum_{s \in S} bw(s) \cdot y_{s,h} \leq cap(h) \qquad \forall h \in L'_H \tag{3.2}$$

$$\sum_{s \in S} bw(s) \cdot x_{s,l} \leq cap(l) \qquad \forall l \in L_L \tag{3.3}$$

Every producer is required to create a single stream and send it to a single media application. Therefore, only one of its outgoing physical links transmits the created stream (3.4).

$$\sum_{h \in out_H(p)} y_{s,h} = 1 \qquad \forall s \in S, p \in P : p = producer(s) \tag{3.4}$$

Since a stream is required to be sent from a producer to a consumer along a single path, every consumer has to receive its required stream only by one of its incoming physical link (3.5).

$$\sum_{h \in in_H(c)} y_{s,h} = 1 \qquad \forall c \in C \; \forall s \in S : c = consumers(s) \tag{3.5}$$

Distributors similarly to consumers can receive only one stream and only over one link. However, it may happen that they do not receive any stream at all (3.6).

$$\sum_{s \in S} \sum_{h \in in_H(d)} y_{s,h} \leq 1 \qquad \forall d \in D \tag{3.6}$$

The constraints (3.7) and (3.8) guarantee that the distributors forward a stream only if they also receive it. The number of forwarded streams have to be greater than the number of received streams (3.7). Therefore, the distributor must forward every incoming stream. Constraint (3.8) states that a distributor receiving the stream s, can forward at most $\|consumers(s)\|$ copies of it, but it can not forward any streams that are not received.

$$\sum_{h \in in_H(d)} y_{s,h} \leq \sum_{h \in out_H(d)} y_{s,h} \qquad \forall s \in S \; \forall d \in D \tag{3.7}$$

$$\|consumers(s)\| \cdot \sum_{h \in in_H(d)} y_{s,h} \geq \sum_{h \in out_H(d)} y_{s,h} \qquad \forall s \in S \; \forall d \in D \tag{3.8}$$

Physical and unknown network nodes are allowed only to forward incoming streams. Every instance of a received stream has to be forwarded further (3.9).

$$\sum_{h \in in_H(v)} y_{s,h} = \sum_{h \in out_H(v)} y_{s,h} \qquad \forall s \in S \; \forall v \in V_H \cup V_U \tag{3.9}$$

Note that the constraints representing the limitations of the media applications contain only physical streamlinks. Since there is a much lower amount of physical links than logical ones, the time required to find a solution for the problem may be decreased this way.

3.2.4 Cycles

The constraints presented above do not forbid the existence of a cycle in the solution. It may happen that a cycle appears between some distributors which do not receive any stream from other media applications. The approach used in this work does not prevent the cycles in the solution, but instead eliminates them only when they appear. Every time an optimal solution is found to the problem it is checked whether it contains any cycles. If it does, a constraint is added to the model for every single cycle, which prevents the existence of the given cycle in the solution. Afterwards the solution is discarded and the solving continues from the point before when the solution was found. This approach has been shown much more efficient than including all the cycle elimination constraints in the base model [4].

3.2.5 Objective Function

Since the main objective of the MSPP is to reduce the transmission latency of the streams, we define the objective function as a sum of latencies of all the active logical streamlinks.

$$\min \sum_{s \in S} \sum_{l \in L_L} x_{s,l} \cdot latency(l)$$

Chapter 4

Free MIP Solvers

4.1 Available Solvers

The first part of this work was to choose an appropriate freely distributable MIP solver for the implementation of the MSPP solver. Since CoUniverse is written in Java, one of the main requirements of the solver was to have a suitable Java library. Otherwise, the most significant factor in the selection was the performance of the solvers. On the web page [8] we can find a comprehensive list of the best known MIP solvers. There are four non-commercial ones between them with an available Java library and an appropriate performance: lp_solve, GLPK, CBC and SYMPHONY. In the next section we give a brief overview about their main properties. All the information is processed from the official websites of the solvers [9, 10, 11, 12].

4.1.1 lp_solve

- **License:** GNU lesser general public license
- **Version:** 5.5.2.0
- **Website:** `http://lpsolve.sourceforge.net/5.5/`
- **Language:** C

Lp_solve is a free linear and integer programming solver. It contains full source, examples and manuals. It does not limit its model size and can accept user defined input files. There are plenty of interfaces through which lp_solve can be called, like Java, .NET, AMPL, MATLAB, O-Matrix, Scilab, Octave, and others.

4.1.2 GLPK

- **License:** GNU General Public License
- **Version:** 4.51
- **Website:** `http://www.gnu.org/software/glpk/glpk.html`
- **Language:** C

GLPK (GNU Linear Programming Kit) is a package organized in the form of a C callable library. It contains a standalone LP and MIP solver. It can be used through a large number of community built interfaces.

4.1.3 CBC

- **License:** Eclipse Public License
- **Version:** 2.8
- **Website:** `https://projects.coin-or.org/Cbc`
- **Language:** C++

The software CBC is an open-source integer programming solver distributed under the COIN-OR project. It uses many other software products from the same project for additional functionality, like the linear programming solver Clp or the Coin Cut Generation Library. It can be used as a callable library or as a standalone solver.

4.1.4 SYMPHONY

- **License:** Eclipse Public License
- **Version:** 5.5
- **Website:** `https://projects.coin-or.org/SYMPHONY`
- **Language:** C

SYMPHONY is an open-source MIP solver. Similarly to CBC it is distributed under the COIN-OR project and built from many other COIN components. It has some unique advanced features, like being able to solve MIP problems with two objective functions or to warm start the solution procedure, which allows to set an advanced starting point for MIP optimization.

4.2 Performance Analysis

An important part of the thesis was to analyse and compare the performance of the above described MIP solvers. For that I used integer programming models generated by the CoUniverse. These models represent some typical network topologies used in high-bandwidth data transmissions.

4.2.1 Testing Topologies

For being able to describe the structure of the above-mentioned testing topologies, we need to introduce the term of a *network site*. A network site represents the geographical collocation between the nodes of the given network. Typically one site contains several network nodes, which represent the computers connected to the network. As an example we can mention the devices responsible for the generation or reception of the required data streams. The nodes can run any kind of media application, the only restriction is that none of the consumers can consume any streams produced by the producers inside the same site.

Two kinds of communication scenarios were used for the testing. The first one is the *1:n* scenario. It represents a situation where one of the sites transmits a data stream to all of the remaining sites and also receives a stream from each of them. All of the sites contain a consumer for every incoming stream, a producer, and a distributor. It is a typical representation of remote lecturing.

The second kind of scenario is denoted as *m:n*. It is a classical representation of videoconferencing, where all of the participants communicate with each other. Every site transmits a stream to all of the others and receives a stream from each of them at the same time.

Both of these communication scenarios can be further divided to two types of testing topologies based on the available knowledge of the underlying physical network. The first one is a topology with *detailed* knowledge of the underlying physical network, where we have complete information about the lower network levels. It is based on the real connection of European and North American networks (Figure 4.1).

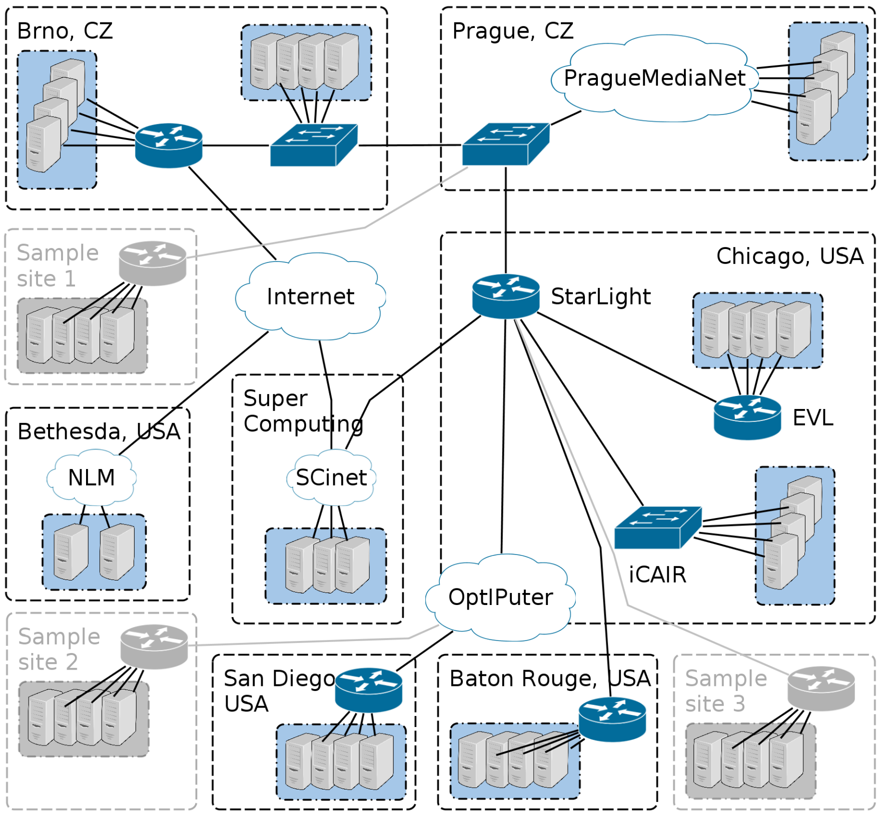

Figure 4.1: Example of a network with 9 sites and detailed knowledge of the underlying physical topology

The second kind of topology is with *missing* knowledge of the underlying physical network. Here no information is available about the lower network layers. Each site is modelled with the help of an unknown network node, which is connected to all of the endpoint nodes of the given site. There is also one unknown network node which interconnects all of the sites in the network. An example with 4 sites can be found in Figure 4.2.

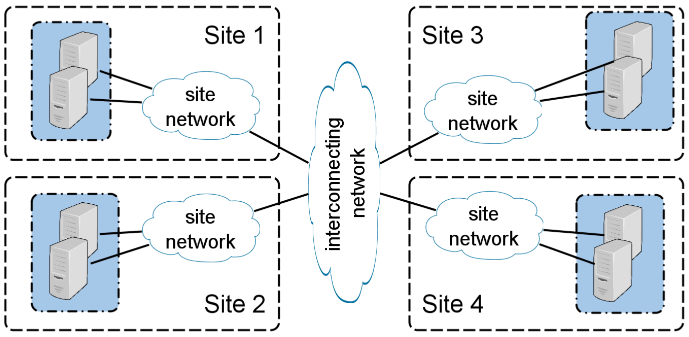

Figure 4.2: Example of a network with 4 sites and missing knowledge of the underlying physical topology

Knowledge	Scheme	Sites	Logical level		Physical level	
			$\|V_E\|$	$\|L_L\|$	$\|V_H \cup V_U\|$	$\|L_H\|$
Detailed	1:n	4	14	76	23	92
		6	22	186	23	102
		8	30	344	23	112
		10	38	550	23	122
		12	46	804	23	132
		14	54	1106	23	142
		16	62	1456	23	152
	m:n	4	20	124	23	98
		6	42	426	23	122
		8	72	1016	23	154
		10	110	1990	23	194

Table 4.1: Parameters of testing topologies with detailed knowledge of the underlying physical network

Knowledge	Scheme	Sites	Logical level		Physical level	
			$\|\|V_E\|\|$	$\|\|L_L\|\|$	$\|\|V_H \cup V_U\|\|$	$\|\|L_H\|\|$
Missing	1:n	4	14	76	5	26
		6	22	186	7	40
		8	30	344	9	54
		10	38	550	11	68
		12	46	804	13	82
		14	54	1106	15	96
		16	62	1456	17	110
	m:n	4	20	124	5	32
		6	42	426	7	60
		8	72	1016	9	96
		10	110	1990	11	194

Table 4.2: Parameters of testing topologies with missing knowledge of the underlying physical network

The number of nodes and links in the testing topologies with detailed knowledge of the physical network can be found in Table 4.1, and with missing knowledge of the physical network in Table 4.2. The numbers of the links are shown after the elimination process, which we have already described in Section 3.2.1.

4.2.2 Testing Methodology

The performance of the MIP solvers was measured on all of the above described testing topologies. For every topology an integer programming model was created, which were used as an input for the MIP solvers. Every model was solved 8 times, but only during the last 5 of them was the actual time of solving the model measured. This way we could be sure that the integer programming model was already loaded to the memory and the results were not affected by searching for the file on the hard disk.

All of the measurements were run on a PC equipped with Intel® Core^{TM}2 Duo CPU T6600 @ 2.20GHz dual-core processor, 3GB RAM, running linux 3.5.0-24-generic and Ubuntu 12.04 LTS. The analysed freely distributable MIP solvers were lp_solve 5.5.2.0, GLPK 4.51, CBC 2.8 and SYMPHONY 5.5.

4.2.3 Results

The results of the tests can be seen in the graphs below. The represented values are in milliseconds and were calculated as the average of the 5 measured times for all of the testing topologies. Since the performance of the freely distributable MIP solvers is far behind the commercial ones, the size of the testing topologies was chosen small enough so that all of the analysed solvers could solve them in a reasonable amount of time.

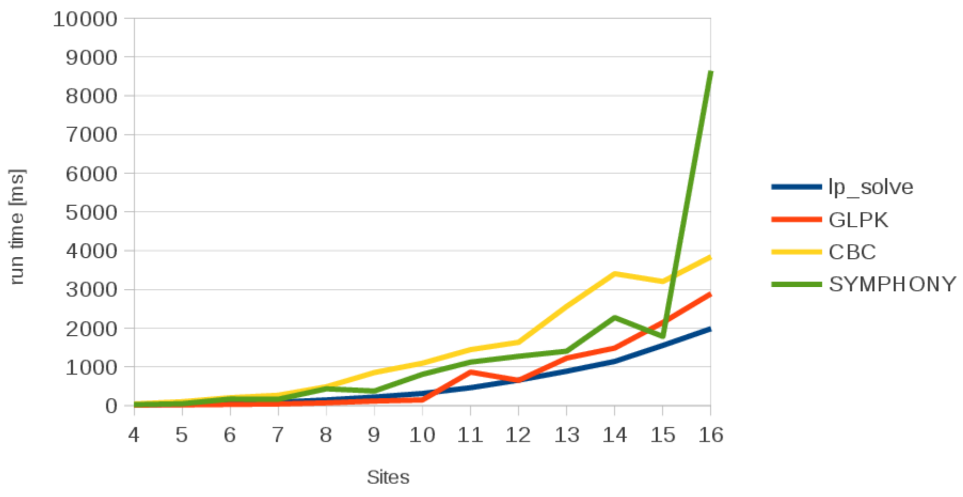

Figure 4.3: Detailed 1:n topology

The graph in Figure 4.3 shows the result for the 1:n topology with a detailed knowledge of the underlying physical network. The solver lp_solve achieved the best performance in this kind of topology closely followed by GLPK. The performance of SYMPHONY was quite close to them on the smaller instances, but was much slower on the biggest instance with 16 sites.

The results for the m:n topology with a detailed knowledge of the physical network can be seen at the graph in Figure 4.4. The solver GLPK achieved the best performance on all of the testing instances, except the last one, where CBC was ahead of it by more than 4 seconds.

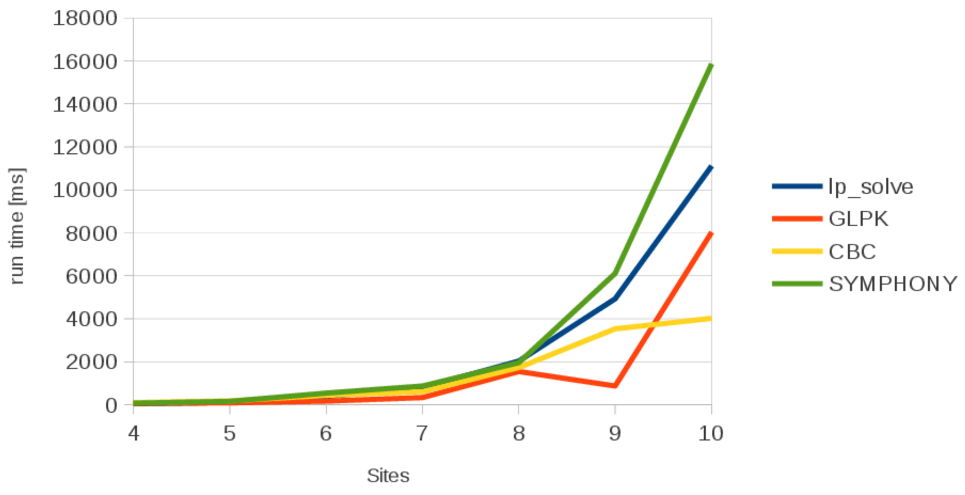

Figure 4.4: Detailed m:n topology

The results for the 1:n topology with missing knowledge are at the graph in Figure 4.5. The fastest solver on this kind of topologies was GLPK, closely followed by lp_solve. The results of SYMPHONY and CBC were similar to each other, one of them being sometimes a little slower or little faster than the other.

At last, the results for the m:n topology with missing knowledge of the physical network can be seen at the graph in Figure 4.6. Just like in the previous case, the best results were achieved by the solver GLPK, with CBC closely behind it. The worst results were achieved by SYMPHONY, which was far behind the other solvers on the bigger testing instances.

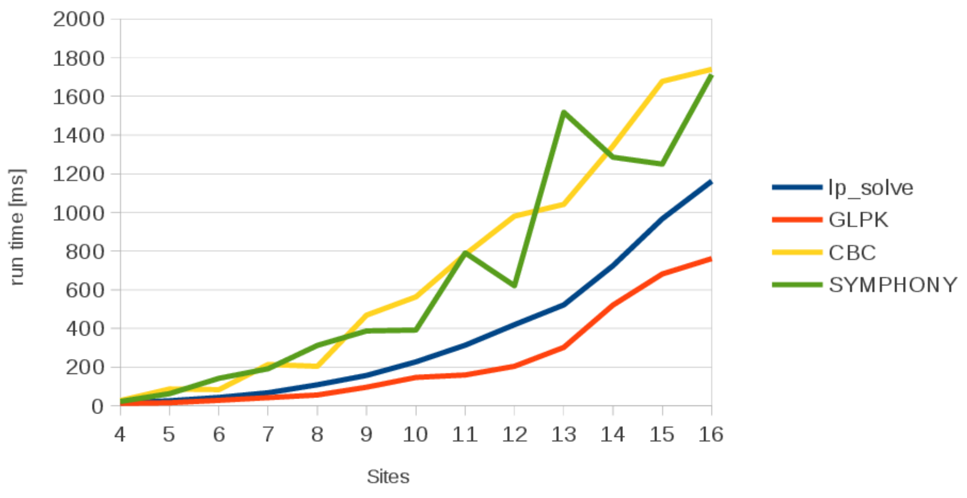

Figure 4.5: Missing 1:n topology

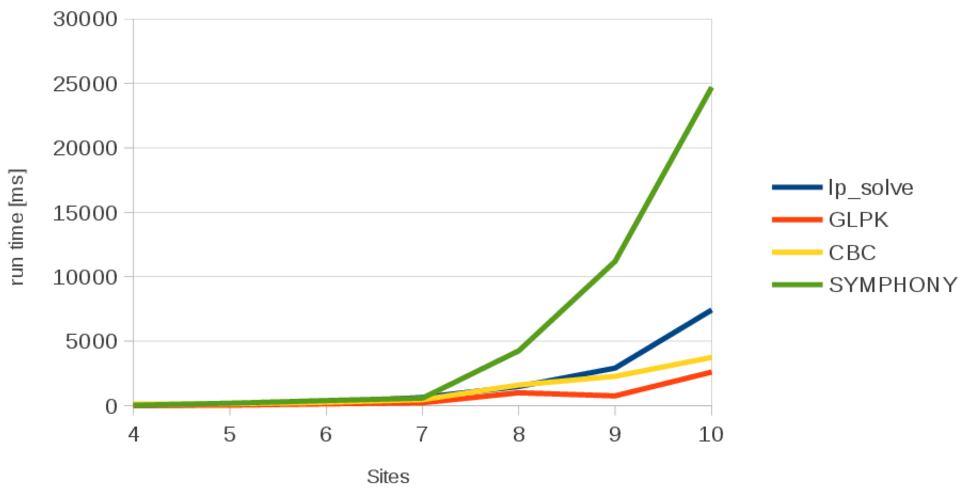

Figure 4.6: Missing m:n topology

Note that the type of the testing topologies had a big influence on the results of the tests and on the behaviour of the analysed solvers. For example, lp_solve performed very well on both of the 1:n communication scenarios, but was much slower on the m:n scenarios. On the contrary, CBC achieved very good results on the m:n scenarios, while being a little bit slower on the 1:n ones. GLPK had the best performance on both of the testing topologies with missing knowledge of the physical network and also achieved very good results on the other ones. On the other hand, the performance of SYMPHONY was probably the worst, it was much slower than the other solvers on the bigger testing instances.

Considering everything I have decided to choose the solver GLPK for the implementation of the MSPP solver. In overall its performance was better than the performance of the other solvers. Even in the testing topologies where it was not the fastest, it was just a little bit behind the best solver on the given topology.

Chapter 5

Implementation

The aim of this work was to implement a scheduler for the MSPP which would use a non-commercial MIP solver instead of the actually used proprietary MIP solver Gurobi. For that I could freely use the implementation of the original scheduler and all the other components in CoUniverse. My work thereby mostly consisted of making the necessary changes so that the solver GLPK could be used during the solving process instead of the solver Gurobi. In the sections below I am going to describe the structure of the MSPP scheduler and afterwards the methods and processes I have used during the implementation of my work.

5.1 Scheduler

As it was already described in the previous chapters, the scheduler for the MSPP is the component responsible for planning the routes inside the given network during actual data transmissions. It was developed in the Java programming language like the rest of the CoUniverse framework. The base class describing the required functionality of a scheduler is the abstract class `MatchMaker`. Every scheduler should be implemented within a class inheriting from it. `MatchMaker` declares three base methods: `doMatch`, `getPlan` and `getMapVizualization`. These methods serve for launching the planning process, retrieving the plans for the data transmission, and for generating the visualisation of the network environment. From the point of this work the most important is the `doMatch` method, which creates the actual integer programming model and solves it via the MIP solver.

The scheduler using Gurobi has been implemented inside the class `MatchMakerPKTGurobi`. The `doMatch` method of the class can be divided to some different phases. The first one is the elimination process, which has been already described in Section 3.2.1. In this phase all of the unnecessary links are eliminated from the network topology. These include the ones with insufficient capacity for data transmissions or links with an inappropriate media application on some of their endpoint nodes. Furthermore, since media applications are aggregated into network sites, all of the links between the media applications inside the same site can be eliminated. By the elimination process we can decrease the number of decision variables inside the integer model and this way improve the time needed to find a solution.

During the next phase of the `doMatch` method the individual decision variables, constraints and objective function of the integer programming model are created. However, we do not necessarily have to create all of the constraints in the model. There is a class called `MatchMakerConfig`, which contains information about the configuration of the scheduler. In this class we can explicitly specify which of the constraints should be created.

The last phase only consists of solving the integer programming model by the MIP solver Gurobi. A new constraint may be added to the model if a cycle is found in the solution during the solving process, like it was already described in Section 3.2.4.

5.2 Moving to GLPK

Since GLPK provides only a library written in C for the MIP solver, my first task was to find a suitable Java interface for it. There are two publicly available Java interfaces for GLPK: the project GLPK for Java [1] and the GLPK 4.8 Java Interface [2]. Both of them were generated from GLPK's default C library through the Java Native In-

1. http://glpk-java.sourceforge.net/
2. http://bjoern.dapnet.de/glpk/

terface, which enables to integrate non-Java language libraries into Java applications. Because of that they declare the same methods as the C library of GLPK and provide the same level of functionality. However, I have discovered an error in the GLPK 4.8 Java Interface because of what it is not capable of solving MIP problems correctly. There is a method called `setColKind`, in which we can set whether a decision variable should be of continuous or integer type. Nevertheless, in both of the cases the variable becomes of continuous type and there is no way to change it to integer. For that reason I have decided to use the interface GLPK for Java, where all of the methods I have needed worked correctly.

After choosing the Java interface, my work continued by comparing it to the Gurobi Java interface [3] and looking for the differences in their functionalities. Since GLPK for Java was only generated from the default C library, most of its functionality resides in a single class called `GLPK`. On the other hand, the Gurobi Java interface contains plenty of classes which represent the components of the MIP problem. As an example we can mention the classes representing decision variables, constraints or the integer programming model. These classes enable to easily create and modify MIP problems. Plenty of their methods enable to do things which are achievable only by calling multiple methods of the GLPK for Java library. Also, when using GLPK for Java we need to deal with things which are kept hidden in the Gurobi Java interface, like allocating memory before the creation of a constraint, or initializing the control parameters of the solver before the optimization. Therefore, I have decided to create my own classes which would encapsulate the GLPK for Java interface and provide the same functionality and method declarations as the Gurobi Java interface.

5.2.1 Implemented Classes

I have created three different classes: `GLPKVar`, `GLPKLinExpr` and `GLPKModel`. The class `GLPKVar` represents the decision variables of

3. http://www.gurobi.com/documentation/5.1/reference-manual/node254#sec:Java

the integer programming model. Every variable needs to be associated with a GLPKModel object. Typically a variable object is created by adding a variable to the model (using GLPKModel.addVar), rather than by calling its constructor. Every variable contains three attributes: the already mentioned model, a name, and the column of the variable. The columns represent the order of the variables inside the model, and serve for identification purposes. The only methods of the class are for getting the values of the above mentioned attributes.

The main purpose of the decision variable objects is to create the instances of the GLPKLinExpr class. An instance of GLPKLinExpr contains a list of coefficient-variable pairs representing a linear expression. A coefficient-variable pair is called a term. Usually linear expressions are built by starting with an empty linear expression and adding terms to it afterwards. Terms can be added individually with the addTerm or in groups with the addTerms method. Also, with the help of the multAdd method we can add a constant multiple of one linear expression into another. Upon completion, the invoking linear expression is equal the sum of itself and constant times the argument linear expression. The last method of the class is the size, which returns the number of terms of the linear expression.

The last class, GLPKModel, encapsulates a glp_prob object of the GLPK for Java interface. The glp_prob object is a so-called *problem object*, which represents the particular integer programming model. The instances of the GLPKModel serve to create decision variables and constraints, and to add them to the problem object. Variables can be added by the addVar method. The method takes five parameters: name, lower bound, upper bound, objective coefficient, and type. The bounds determine the interval in which the variable is allowed to take a value, the objective coefficient is the coefficient value of the variable in the objective function, and the type determines whether the variable should be of continuous, integer or binary type. All of these values are set by calling the appropriate methods of the glp_prob object after the creation of the particular variable. If we would like to modify some of them later, we can do so by calling the `setVarName`, `setVarType`, `setVarObjCoeff` or `setVarBounds` methods.

The constraints of the model are created by calling the `addConstr` method. It has multiple overloaded versions, each of them required to specify a left-hand side value, a right-hand side value and a relation between the values of the two sides. The types of both of the values can be either a decision variable or a linear expression. The right-hand side value can be also a constant. The relation between the two sides can have three possible values: equal, less than or equal, and greater than or equal. At first a single linear expression is created, by subtracting the right-hand side value from the left-hand side, leaving only a constant on the right side. After that the created linear expression together with the right-hand side constant are passed to the `glp_prob` object to create the actual constraint.

After I have implemented the above-mentioned classes, I have modified the `MatchMakerPKTGurobi` class by replacing all of the methods which call the MIP solver Gurobi by the methods of my classes. This way I have created the `MatchMakerPKTGLPK` class, which is a scheduler for the MSPP using GLPK as its MIP solver.

5.3 Bugs

During my work on the scheduler I have managed to find an error in the `MatchMakerPKTGurobi` class concerning the creation of the constraint (3.8). This constraint should ensure that if a distributor d receives a stream s it creates at most $\|consumers(s)\|$ copies of it. In the implementation the left-hand side value of the constraint was multiplied by the value $\|out_H(d)\|$ instead of $\|consumers(s)\|$. However, this is not correct because if the value of $\|out_H(d)\|$ was smaller than $\|consumers(s)\|$, the distributor would not be able to create a copy of the stream for all of its consumers.

CoUniverse contains also classes which can create network topologies for testing purposes (see Section 6.1). I have found the next error in the creation process of these networks. As it was already stated in Section 2.2.1, every logical link l is related to a set of physical links, which represent the path between the endpoint nodes of the link l. However, during the creation of the network topologies the logical link l was associated with the physical links representing the path from $end(l)$ to $begin(l)$. Since we consider a directed graph, the associated physical links should be the ones forming the path from $begin(l)$ to $end(l)$.

The last error I have managed to find considers the creation of a network topology with detailed knowledge based on the real connection of European and North American networks (see Chapter 4, Figure 4.1). Some of the associations between logical and physical links were not defined correctly, and because of that topologies with more than 8 sites could not be solved.

Chapter 6

Evaluation

This chapter is about the performance analysis of the two implemented schedulers, the one using GLPK as its backend solver and the one using Gurobi. In the sections below I am going to describe the methodology of the testing and afterwards the results of the measurements.

6.1 Testing Tools

CoUniverse already contains all the necessary tools for the performance testing of the `MatchMakerPKTGurobi` class. The main component responsible for the testing of the scheduler is the class `MatchMakerPKTGurobiEvaluationTest`. Since CoUniverse can be used in different kinds of networks and collaborative environments, `MatchMakerPKTGurobiEvaluationTest` can create four types of testing topologies, each of them representing a different kind of scenario. For further information see Section 4.2.1, where all of these topologies were described in detail.

Testing topologies with detailed knowledge of the underlying physical network can be created by calling the `createNetworkTopology` method of the class, and topologies with missing knowledge by the `createUnknownNetworkTopology` method. Both of the methods take parameters which specify the number of sites and the communication scenario (1:n, m:n) of the created network topology. The parameters of the actual testing topologies that were used in the testing can be found in Table 6.1 and Table 6.2.

Knowledge	Scheme	Sites	Logical level $\|V_E\|$	Logical level $\|L_L\|$	Physical level $\|V_H \cup V_U\|$	Physical level $\|L_H\|$
Detailed	1:n	2	6	14	23	82
		4	14	76	23	92
		6	22	186	23	102
		8	30	344	23	112
		10	38	550	23	122
		13	50	949	23	137
	m:n	2	6	14	23	82
		3	12	51	23	89
		4	20	124	23	98
		6	42	426	23	122
		7	56	679	23	137

Table 6.1: Parameters of testing topologies with detailed knowledge of the underlying physical network

Knowledge	Scheme	Sites	Logical level $\|V_E\|$	Logical level $\|L_L\|$	Physical level $\|V_H \cup V_U\|$	Physical level $\|L_H\|$
Missing	1:n	2	6	14	3	12
		4	14	76	5	26
		8	30	344	9	54
		10	38	550	11	68
		14	54	1106	15	96
		18	70	1854	19	124
	m:n	2	6	14	3	12
		3	12	51	4	21
		4	20	124	5	32
		6	42	426	7	60
		8	72	1016	9	96

Table 6.2: Parameters of testing topologies with missing knowledge of the underlying physical network

The next method of the class is the `testEvaluateMatchMaker`. It calls the previously described `createNetworkTopology` method to create a testing topology with detailed knowledge, and solves it with an instance of the `MatchMakerPKTGurobi` class. The method `testEvaluateMatchMakerUnknown` works analogously, with the only difference of creating a topology with missing knowledge by calling the `createUnknownNetworkTopology` method.

And finally, the method `testMultiRunEvaluateMatchMaker` controls the whole process of the testing with the help of the previously described methods. It is the only public method of the class. It controls the creation of the testing topologies, decides what kind of topologies to create, determines the number of sites inside the topologies, and repeats all the measurements a certain amount of times. It creates also an instance of the `MatchMakerConfig` class, where we can specify which of the constraints should be created and what kind of cycle elimination to use in the integer programming model.

For being able to test also the performance of the MSPP scheduler which uses GLPK as its backend solver, I have created the class `MatchMakerPKTGlpkEvaluationTest`. It differs from the previously described class only in one thing, it creates an instance of the `MatchMakerPKTGlpk` class to solve the testing topologies instead of an instance of `MatchMakerPKTGurobi`.

6.2 Testing Methodology

Two values were measured during the testing process, the time of the preparation phase and of the solving phase. The preparation phase includes the elimination process (see Section 3.2.1), where all of the unnecessary network links are eliminated from the network, and the creation of the decision variables, constraints and objective function of the model. The solving phase includes the actual call of the MIP solver to solve the integer programming model. Since the cycles are eliminated during the solving process (see Section 3.2.4), the time needed for their elimination is also counted to this phase.

The actual tests were performed on all of the testing topologies shown in Table 6.1 and Table 6.2. Every topology was tested 20 times, but only the last 5 measurements were taken into account. This way the results were much less influenced by the startup delay of the Just-in-time compilation and by the influence of the Java garbage collector. The test were run on a PC equipped with Intel® Core^{TM}2 Duo CPU T6600 @ 2.20GHz dual-core processor, 3GB RAM, running linux 3.5.0-24-generic and Ubuntu 12.04 LTS. The used MIP solvers for the testing were GLPK 4.51 and Gurobi 5.5.

6.3 Results

The measured results of the tests can be found in Table 6.3 and in Table 6.4. All of the represented values are shown in milliseconds and were calculated as the average of the last five measured values of the twenty performed runs. The tests were performed also on bigger testing topologies, but increasing the number of the sites on any of the represented topologies just by one value causes a huge rise in the results of GLPK. For the performance of Gurobi on bigger testing instances you can see the work [3].

Topology	Sites	Preparation		Solving		Total	
		GLPK	Gurobi	GLPK	Gurobi	GLPK	Gurobi
Detailed 1:n	2	30	35	13	20	43	55
	4	25	43	21	49	46	92
	6	91	46	114	107	205	153
	8	52	48	361	229	413	277
	10	86	100	549	410	635	510
	13	192	244	4789	554	4981	798
Detailed m:n	2	6	12	3	24	9	36
	3	7	16	7	22	14	38
	4	15	38	83	43	98	81
	6	49	88	1136	148	1185	236
	7	127	108	1746	220	1873	328

Table 6.3: Results measured on testing topologies with detailed knowledge of the underlying physical network

Topology	Sites	Preparation		Solving		Total	
		GLPK	Gurobi	GLPK	Gurobi	GLPK	Gurobi
Missing 1:n	2	2	3	1	6	3	9
	4	19	38	21	28	40	66
	8	41	54	141	164	182	218
	10	67	87	373	363	440	450
	14	185	210	1597	565	1782	775
	18	424	444	3246	817	3670	1261
Missing m:n	2	7	2	4	1	11	3
	3	52	8	43	35	95	43
	4	27	29	41	67	68	96
	6	44	79	540	127	584	206
	8	123	154	4572	436	4695	590

Table 6.4: Results measured on testing topologies with missing knowledge of the underlying physical network

The results of the performance tests with additional error bars displayed can be found in the Figures 6.1 and 6.2. They display the overall performance of the schedulers, that is the sum of the preparation time and solving time. The biggest error in the case of GLPK was 811 milliseconds, while in the case of Gurobi 227.

The aim of these tests were to compare the performance of the two schedulers and to find out for how big testing instances can GLPK be used. If we want to provide real time communication for the users, the scheduler should be able to plan the data transmissions within the limit of five seconds. From the results we can see that my implementation is sufficient to plan data transmissions up to 7 sites in m:n communication scenarios and up to 13 in 1:n ones. For testing topologies with missing knowledge of the underlying physical network the scheduler managed to find the solutions a little bit faster and was capable to solve also some of the bigger testing instances. Notice, that the values of the preparation phase does not increase linearly on some of the smaller topologies as it would be expected. It is probably due to the influence of the garbage collector, which could not be eliminated entirely.

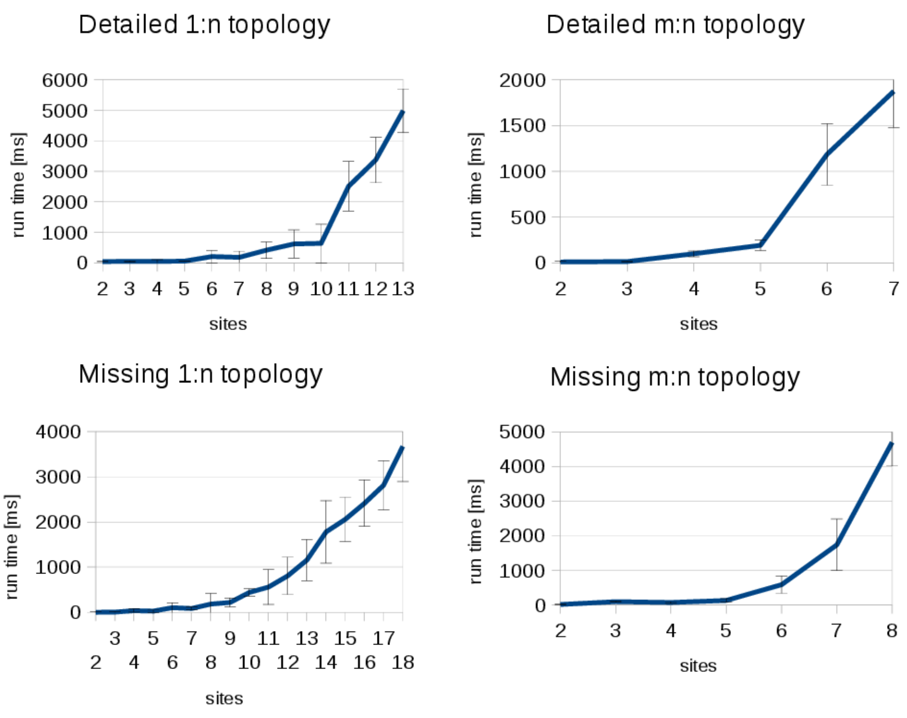

Figure 6.1: Results of the overall performance of GLPK

When comparing the performance of the schedulers Gurobi managed to plan the data transmissions much faster than GLPK. However, this result is not surprising, considering that commercial solvers have much better performance as non-commercial ones. In the preparation phase, however, GLPK managed to produce very similar results as Gurobi. It is the solving phase where GLPK's performance stays behind and becomes the bottleneck of the planning. GLPK was able to outperform Gurobi only on some of the smallest topologies. On the other hand, the scheduler using GLPK has adequate performance for being used in typical communication scenarios, and more importantly it can be used in the distribution of the CoUniverse framework.

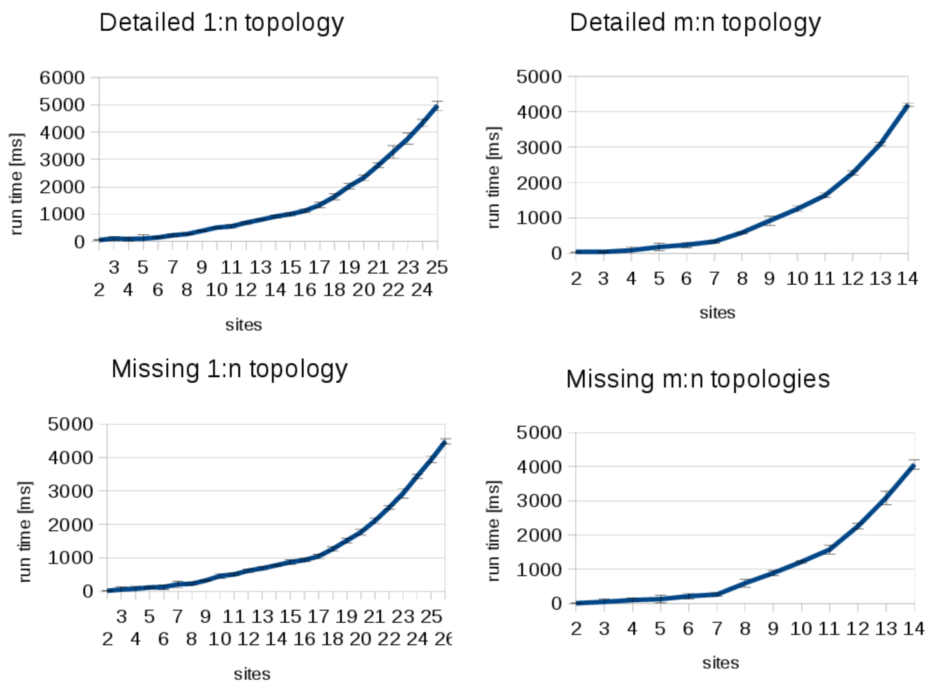

Figure 6.2: Results of the overall performance of Gurobi

Chapter 7

Conclusions

In this thesis I have presented the implementation of a scheduler for the MSPP using a non-commercial MIP solver. The first part of my work consisted of analysing the performance of the non-commercial MIP solvers lp_solve, GLPK, CBC and SYMPHONY. I have tested the solvers on integer programming models representing typical types of collaborative environments. The performance of GLPK proved to be the best in most of the situations, and therefore I have decided to use it for the implementation.

Before starting the actual implementation, I needed to get familiar with the source codes of CoUniverse and to understand the implementation of the scheduler that uses Gurobi. After I understood its functioning, I was able to implement a scheduler for the MSPP using GLPK as its MIP solver.

The last part of my work was to analyse the performance of the implemented scheduler and to compare it with the performance of the scheduler using Gurobi. I have measured two values during the tests, the preparation time and the solving time. As it could have been expected, GLPK's overall performance was not as fast as Gurobi's. However, the results of the preparation time of both of the solvers were on the same level, what can be considered a good result.

Despite its weaker performance, the scheduler using GLPK can be still used during typical data transmissions, and most importantly it can be freely distributed with the rest of the CoUniverse framework, while Gurobi can be used only for academical purposes.

Bibliography

[1] Miloš Liška and Petr Holub. Couniverse: Framework for building self-organizing collaborative environments using extreme-bandwidth media applications. In *The Eighth International Conference on Networks ICN 2009*, pages 259–265, Cancún, Mexico, 2009.

[2] Pavel Troubil and Hana Rudová. Integer programming for media streams planning problem. In *Sixth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10) – Selected Papers*, pages 116–123, Dagstuhl, Germany, 2011.

[3] Pavel Troubil, Hana Rudová, and Petr Holub. Media streams planning with transcoding. In *12th IEEE International Symposium on Network Computing and Applications (NCA 2013)*, pages 41–48, USA, 2013. IEEE.

[4] Pavel Troubil and Hana Rudová. Integer linear programming models for media streams planning. *Lecture Notes in Management Science*.

[5] IBM. Linear programming and cplex optimizer. `http://www-01.ibm.com/software/commerce/optimization/linear-programming/`. [Online].

[6] Robert Bosch. Integer programming. In *Search Methodologies*, pages 69–95, 2005.

[7] V. Chvátal. *Linear Programming*. A Series of books in the mathematical sciences. Freeman & Company, 1983.

[8] Decision tree for optimization software. `http://plato.asu.edu/guide.html`. [Online].

[9] Introduction to lp_solve 5.5.2.0. `http://lpsolve.sourceforge.net/5.5/`. [Online].

[10] Glpk (gnu linear programming kit). `http://www.gnu.org/software/glpk/`. [Online].

[11] Cbc home page. https://projects.coin-or.org/Cbc. [Online].

[12] Symphony home page. https://projects.coin-or.org/SYMPHONY. [Online].