Universität Stuttgart

IPVS

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis No. 3594

Solving the Content Routing Problem with Coupled Constraint Oriented Programs

Patrick Bosch

Course of Study:	Informatik
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Boris Koldehofe
Commenced:	6. November 2013
Completed:	8. May 2014
CR-Classification:	C.2.1, C.2.4, G.1.6

Abstract

Publish/Subscribe and especially content-based publish/subscribe are widely used for communication between distributed components. They offer efficient bandwidth usage through forwarding events only to subscribers which are interested in the content of the event. Forwarding is done by filters which identify the content of the event.

Previous implementations of such systems relied on a distributed set of brokers to match and forward the events accordingly. However, the advent of software-defined networking gave us the possibility to implement the system directly on the network layer. The overhead of the application layer can be prevented and the filtering process can be implemented more efficiently. Previously, the costs for filtering were a major disadvantage for content-based publish/subscribe systems. Such a new filtering approach with implementation on the network layer was presented in initial work and it was shown that line-rate performance could be achieved. For this approach the power of software-defined networking could be fully harnessed by implementing forwarding flows that handle the filtering as well as the forwarding. Although, this approach offers a possibility for a brokerless publish/subscribe system, the task to compute and optimise routes that connect publishers and subscribers is not solved adequately.

To understand the problem of route optimisation in such a system and the problems that are incorporated in it, this work divides the overall problem into multiple smaller problems and offers solutions for these smaller problems. A framework is presented and discussed with regard to achieving a complete solution which incorporates optimisation problems as well as transformations and mechanisms that ensure that each part is independent from the other parts. The performance as well as the results the framework produces are discussed after its presentation.

Acknowledgements

I would like to thank and acknowledge those that contributed to the completion of this thesis. Many thanks to Prof. Dr. Kurt Rothermel for giving me the opportunity to work on this thesis in the Department of Distributed Systems in a field that is very interesting.

I am grateful to my supervisor Dr. Boris Koldehofe for his support and patience throughout the time for this thesis. His help and guidance were an invaluable asset for the completion of this work. The meetings and discussions with him have contributed to developing the ideas for this thesis.

I express my gratitude to Franz Fabian who provided me with the necessary setup for the evaluation of this thesis. I also want to thank the department for the provided help.

Special thanks go to my friends Sukanya Bhowmik, Darshana Das, Naresh Nayak, Hua Ma and Siliang Xia for their support and motivation as well as help in solving problems and correcting my language.

At last I want to thank my family, especially my mother, for their continuous support throughout my studies as well as this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication and information exchange is more important than ever. The amount of data that is transferred increased drastically over the years. A lot of our systems that we daily use are distributed in their nature. To handle the delivery in such a system, which is mostly asynchronous and does not have a lot of dependencies, we use event notification systems like publish/subscribe. In such systems bandwidth optimisation is an important part to make them efficient.

The optimisation of bandwidth usage in publish/subscribe systems is an important topic in computer science. When we reduce the bandwidth, we can reduce the costs of the system, because we will need less hardware to achieve the same result or we can support more participants with the same amount of hardware. We can also use the topology more efficiently and deploy more applications. All this is possible with optimising the bandwidth usage, which means in our case reducing the number of events that are sent in the system. Along with the load on the switch, the individual delay from publisher to subscriber is an important aspect. Both should be optimised as well to achieve the best result.

With the rise of software-defined networking, we have many more possibilities to influence the routes, which messages use in a network. Therefore we do not need to rely on distributed path finding algorithms and can define routes in a centralised way so that content matters and the route is optimal for our use case. The controller as the central point has a global view of the entire network and can use this knowledge in the computation. The centralised controller and the global knowledge is very convenient if we have a way to encode information about the content of the message so that we can compare this information easily and use it together with the knowledge about the network to compute optimal paths that consider individual constraints.

One such possibility and the corresponding publish/subscribe system is presented in [34]. This system shall be used as the basis of our work. Filters are used to encode and identify the content of the message and to compare it with other messages. Another feature of this system is the implementation of the broker logic into the network itself and therefore no more brokers are needed. This reduces the overhead since brokers are normally located on the application layer. Software-defined networking is used to implement routes in a network topology. These switches present the broker logic and ensure the correct delivery of the messages to the subscribers. The controller computes these routes and the result should be optimal in consideration of bandwidth, delay, and load-balancing

The computation of these routes is not an easy task and can be quite time consuming, like route computation in general. A heuristic approach that solves this problem is presented in [41]. This approach focuses on dynamic adaption of the system to newly joined participants. The time of the integration of these participants into the active system is a major aspect. However, this approach has the drawback that the solutions are not optimal. We will not consider heuristics in our approach and will instead focus on the quality of the solution and not the dynamics of the system, which means we will take more time for the computation. Of course, we will keep the dynamic in mind and will consider it in the tests afterwards and look for parameters that allow a trade-off between time of the computation and quality of the solution. Nevertheless, our focus stays on the quality of the solution, which should be optimal in respect to the mentioned aspects.

To achieve our aim of optimising the routes, we will employ methods for constraint oriented programming. For this purpose, we will identify several optimisation problems that can be formulated as constraint oriented programs. Furthermore, we will identify mechanisms and transformations, which are necessary, for completing our goal to optimise the routes. All these components will be organised in a framework which we will present and analyse. The framework will be assembled out of components, which are exchangeable, and will be therefore highly dynamic. Further, the framework consists of a surrounding program in which the components are embedded. This makes it easy to use the system for different optimisation goals. We only need to change a few components and we are ready to compute a completely different optimisation. We will present one or more solutions for each problem, mechanism or transformation and also their combinations. The framework will consist of one of these solutions which fits our purpose the most.

Structure

The structure of the remaining thesis is organised as follows:

Chapter 2 on page 4 presents the background to understand publish/subscribe systems, SDN and constraint oriented programming. The fundamental parts and behaviours are explained to give a further understanding of the topic.

In chapter 3 on page 11 we will take a look at the related work and the approaches they took in realising optimisation in a publish/subscribe system. We will also regard fractional solutions that solve a part of the problem we want to solve.

Chapter 4 on page 15 is used to identify different optimisation problems, the mechanisms needed, and the transformations that are required. Each problem will be formally described so that we can find solutions for it easily.

In chapter 5 on page 24 several solutions and mechanisms are presented which solve the problems that were presented in the chapter before. The solutions will be analysed and their compatibility with our goals shall be tested.

Chapter 6 on page 50 introduces the framework and explains the collaboration between the different components.

In chapter 7 on page 61 the results of the framework will be presented. The tests will vary in several parameters and we will check if there is a good setting where the trade-off between time consumption and quality is passable.

Chapter 8 on page 77 will sum the thesis up and gives an outlook won the possible future work in context of this thesis.

Chapter 2

Background and Problem Formulation

2.1 Background

There are three important areas to present before we can go further into details. These three areas are publish/subscribe systems, the topic of software-defined networking and constrained oriented programming. Publish/subscribe systems, as well as constraint oriented programming, especially linear programming, are two well-established topics and there exists a lot of literature and research which covers these topics. SDN on the other side is relatively new and a lot of research is going on in this area currently because it gives many possibilities, which we will be covered in this chapter.

2.1.1 Publish/Subscribe

Publish/Subscribe systems are widely established systems where the participants are loosely coupled. They are in contrast to remote procedure calls where there is a lot of dependency between the two participants. Loose coupling as opposed to strong coupling means that both participants have no knowledge about each other and no dependencies. In publish/subscribe systems this is normally realised through the broker system. Publishers as well as subscribers only communicate with the broker system to advertise their events or to subscribe to events. The broker system then forwards the events to the correct subscribers. There is no direct communication between the publisher and the subscriber. The function of the broker system is explained in detail later on. Loosely coupled systems have the advantage that the participants do not need to have knowledge about each other. This is useful when the components of the system often change or if they should be independent by design. If one of the endpoints, publisher or subscriber, changes than there is no need to change the other endpoint. In strongly coupled systems, it would be different. If one endpoint changes the other one needs to be changed too because the dependencies are higher and the communication is generally directly between both participants.

In publish/subscribe systems there exist two kinds of participants with distinct roles. Both types have their roles. Additionally there exists a facilitator whose role is very important.

Publisher The first type of participant is the message source, which produces events, the publisher. It advertises what kind of events it produces to the broker system. It can also unadvertise events if it does not produce them anymore. After the advertisement, the publisher starts to produce messages and sends them to the broker system, which forwards them, further.

Subscriber The second type is the subscriber, the event consumer. The subscriber subscribes to certain events that it wants to receive. Of course, if it does not want to receive these events anymore it can also unsubscribe from them. When an event, that fulfils the condition of the subscription, arrives at the broker system, the broker system notifies the subscriber and delivers the event. The subscriber then consumes the event.

Broker System In addition to these two participants, there is also a so-called broker system. This system ensures the delivery of the messages and manages the advertisements and subscriptions. All messages from the publishers are delivered to the broker system and there the messages are examined and forwarded to the correct subscribers. How this is done depends on the kind of publish/subscribe system but one way is to do this via a list of subscribers and their subscriptions. The new events are compared with the list and all matching events will be delivered to the appropriate subscribers.

The broker system is the heart of a publish/subscribe system. All the logic of the system is embedded in the broker system. For this reason there are several approaches to increase the performance and scalability of the broker system because the load can be very high [24, 10, 45, 42, 56, 57].

Furthermore there are two main categories of publish/subscribe systems. These are topic-based and content-based systems. The difference between these two is the way in which they categorise the messages.

Topic-based In the topic-based approach, several topics are exposed to which the subscribers can subscribe. These topics are static, predefined and it is possible that subscribers get messages, which they do not want but the topic provides. Topics can be seen as logical channels. Publishers are the ones who decide to which topic their events are published. To avoid getting unwanted messages, there exist so-called sub-topics. The sub-topics divide topics into further, smaller parts. This is helpful insofar that subscribers get fewer messages in which they are not interested but cannot prevent it completely because the topics and sub-topics are still static. The advantage of this way is that we do not need much computational power for the broker system. They can simply forward it to all subscribers which have registered for the topic. The disadvantage is of course the overhead of messages and the false messages that the subscribers get.

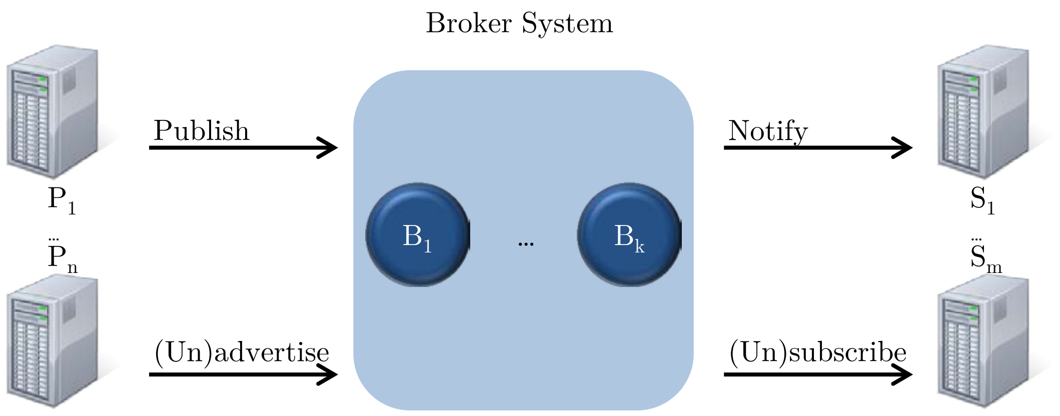


Figure 2.1: Organisation of a Pub/Sub system

Content-based The content-based approach on the other hand filters according to the content or attributes of the messages. A subscriber can specify which content he wants to receive and only gets this content. Every combination of publication and subscription can be seen as an individual logical channel, which means that the filtering is done in a much more precise way. On the other hand, the broker system needs to compare the content of each event with every subscription and if there is a match it is forwarded. It depends on the algorithm for the filtering if this method is time consuming or not but the granularity is of course much higher than in topic-based approaches and the number of false positives is drastically reduced if not eliminated. The granularity of the solution also depends on the algorithm used and the time for the computation. To get good results many optimised filtering techniques were introduced to lower the time of the filtering and make these systems more scalable, like the one in [24]. The advantage in this approach is the granularity and the possibility for highly customised subscriptions but this comes at the expense of time.

2.1.2 Software-Defined Networking

The advent of Software-defined networking dates back a few years. It brought new ideas to distributed environments and gives more control over the network. Control over the network means that we can control how switches behave and how they forward messages.

The main idea is to separate the logic, calculating the forwarding tables, from the forwarding of the messages. We can also call it the separation between the control plane and the data plane. For this, we have a controller that is connected to every switch via the control plane. Through this connection, the controller can manipulate the forwarding rules of the switches. The switches do not compute their own rules anymore, but it is possible to have a hybrid switch where there are forwarding rules, flows, set by the controller and classic ones, which are calculated in a distributed way. The communication over the control plane is done via a

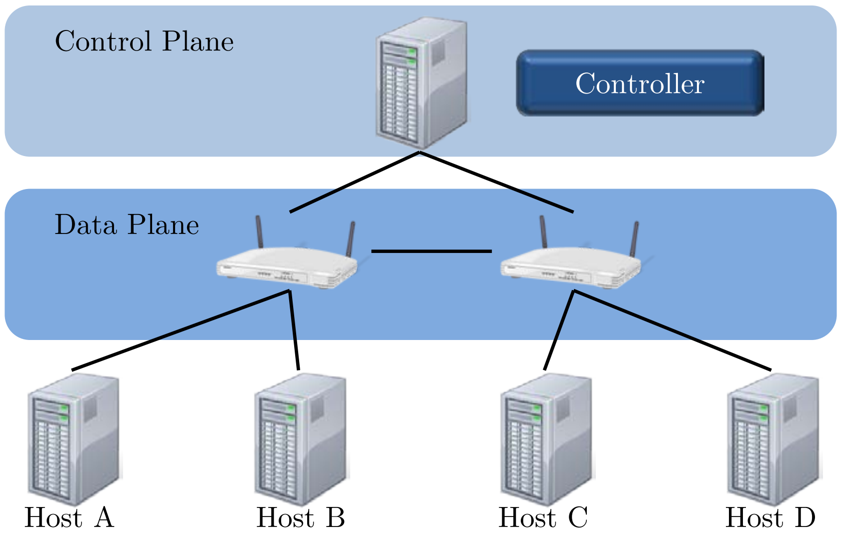


Figure 2.2: Organisation of SDN

protocol, in most cases OpenFlow. This protocol allows us to manipulate nearly anything on the switch and is an abstraction for the manipulation process.

The data plane also has some modifications with regard to "normal" switches. The switch can modify all the meta data and the matching of a packet is not limited to the IP address or the MAC address anymore. Additionally, some OpenFlow specific mechanisms give the user further possibilities like setting the destination of a routing entry to another routing entry and have multiple steps before the packet leaves the switch.

The benefit of this setting is the control over all the forwarding tables. We can install specific routes with specific filters so that a participant receives a specific message. In our particular case that means we can forward the messages from a publisher to a subscriber and filter out those messages which the subscriber does not want on the way. The filtering of the messages is done at the same time as the forwarding but it needs a special representation of the content of the messages to which we will come later. The result of this constellation is a higher effective data rate because we have lower number of false positives and can transmit more data that is relevant. In addition, we do not need a broker system anymore, we only need to calculate the routes once, and the filtering is done passively not actively like with the broker system.

2.1.3 Constraint Oriented Programming

Constraint oriented programming includes a wide area of optimisation domains. One such is linear programming, which is a well-established area. But constraint oriented programming is not limited to linear programming. A great variety of problems can be formulated and solved

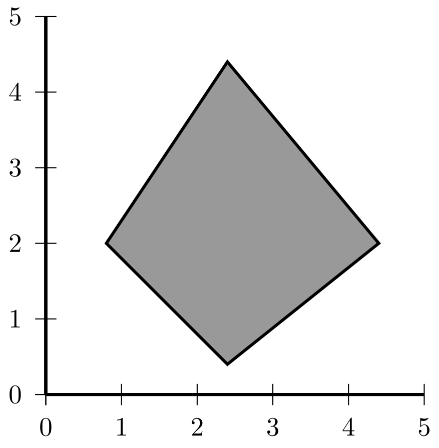


Figure 2.3: Sample space given through constraints

through constraint oriented programming. They are divided into domains; the non-linear domain and finite domain are two examples.

The goal in optimisation is to minimise or maximise a certain objective function. Usually this function is restricted by constraints, which limit the possible sample space. In linear programming, these constraints are linear equality and inequality equations. A sample example would be:

$$\text{Min} \quad Cx \tag{2.1}$$

$$\text{Subject to} \quad Bx \leq a \tag{2.2}$$

$$x \geq 0 \tag{2.3}$$

In this case we try to minimise the objective function C subject to x and under the constraints that x is greater than zero and Bx is lesser than a. That means we are trying to find a value for x that holds those constraints and is minimal for the function C. The two constraints have the task to limit the sample space and form the constraint set. This constraint set spans a sub space in the sample space where possible solutions can occur as shown in figure 2.3.

Although there are an infinite number of feasible solutions in this sample space, the turning points can only occur on the vertices or corners of our constraint set. That means we only need to check those points to get a feasible and optimal solution. The challenge lies therefore to find the global optimum for the objective function. However, there is a method that solves this problem; it is the so-called simplex method. The Simplex Method is a method to find the optimal solution in a rapid manner and makes this kind of programming very efficient. To this point in time there are more methods than the Simplex Method, that are even more optimised and are not limited to only linear programming.

If the variables can only assume integer values than the discipline is called integer linear programming or only integer programming. This makes it more difficult to find the optimal solution because the solution needs to be an integer value and the optimal one could be a real number. One cannot simply take the nearest integer value and take this as the optimal solution but needs different methods.

There also exist other methods to solve non-linear programs or simply only constraint oriented programs. It is similar for other domains than the linear one. We formulate constraints via equations and logical expressions and have an objective function, which we want to minimise. Their exist methods, based on satisfiability modulo theories, to solve these programs. For all domains exist solvers, which can solve the domain efficiently. But not all domains can be solved similar efficient. It is desirable to formulate problems as linear programs, but that is not always possible. Some problems cannot be formulated as one such program. Although, we will try to formulate the problems as linear programs, we may not succeed every time and need to formulate it in another domain or as a mixed constraint oriented program, which means that we use several domains.

2.2 Problem Formulation

Our output network topology is a directed multi graph $G(V, E)$ where V represents the switches and E the edges between these switches. All the edges have a weight w and an identifier f, a filter. The weight can change over time and so can the filters and hence the edges. Between two vertices v_1 and v_2 there exist as many edges as there are filters. The amount of filters that are needed is dependent on the computation of the routes. A filter represents information about the content of events, advertisements, and subscriptions.

This topology is the basis of a content-based publish/subscribe system in which we want to optimise routes. The system, which is the basis of this work, is presented in [34]. Information about the content is coded in filters and these filters are mapped to IP addresses, which can be used to forward the events. The logic of distributing the events, which was previously incorporated into the brokers, is now implemented in the topology itself. The filters and routes define the logic and forward the events to the subscribers accordingly. The controller takes over the role of the broker system by computing the routes, which means creating the routing logic.

Obviously, we do not have only one publisher and one subscriber in such a system but multiple publishers and multiple subscribers. The subscription of a subscriber can include events of many publishers, not always the complete advertisement, but a part of it. The subscriptions of subscribers and the advertisements of publishers can and will overlap. The overlap of subscriptions and advertisements can be used to reduce messages in the network if they use the same edges. This is not always possible, for example, when two subscribers are far away from each other. However, there are many cases of spatial distance, where it is possible to reduce messages.

In this given system, we want to compute routes so that subscribers are connected to the publishers from whom they want to receive messages. This in turn means that there needs to be a path between each publisher and subscriber where there is a subscription. The overall bandwidth usage of these paths should be optimal in respect to individual and average delay, and load balancing on switches. This implies that the weight w on an edge cannot exceed a certain threshold and the overall weight for the system should be minimal. This means we are minimising the weight of the topology under the constraints that there is a maximum individual delay and no switch is over loaded. The bandwidth usage of an edge is represented by the weight. To achieve this, as a first step, we need to make sure that no more than the requested content arrives at the subscribers. This can be done through well-set filters so that they filter out unnecessary messages and reduce false positives. Nevertheless, it is not enough to ensure the correct delivery of messages; our original goal is the reduction of the load of the system. Therefore, we encourage the sharing of paths between similar filters. This is also done through minimising the weight, because a shared path has usually lesser weight. Furthermore, we will limit the length of the individual paths so that we can limit the individual delay. Additionally, the load should be distributed so that no switch is overloaded. This can be done by restricting the number of flows on the switches and the weights on the edges, which can be manipulated to reflect the situation on the switches.

The goal of this work is the optimisation of the topology and associated problems. We need to reduce messages, load on switches and delay of connections on the complete network while maintaining the connectivity between publishers and subscribers. To achieve this we will introduce a framework with different components. Every component can solve a sub problem of the whole task and thus presenting us with a solution for the whole optimisation.

Chapter 3

Related Work

Optimisation and specifically the domain of linear programming are fields that have existed for a long time already. Algorithms like calculating the shortest path between two nodes in a graph are used in examples for linear programming like in [20] or [54], where the path is a set of several connected edges where the first one is the source and the last one is the sink. These examples are widely spread and often used. But mostly only the path between two nodes is considered. Our goal is to calculate a spanning tree from one publisher to multiple subscribers and merge these trees according to the filters so that we have one big route in the network where individual routes are identified by filters. One such approach is presented in [41]. This approach has its focus on dynamism and has therefore a greedy heuristic to cope with the changes that can happen in the network and which the algorithm needs to take care of. It also needs to ensure that subscribers and publishers are connected. As soon as a publisher joins, a tree for this publisher will be computed, if there is none with the specified filter, and all subscribers are added to this tree. This algorithm can deal with dynamism pretty well, but over time the trees can degenerate. A tree is calculated only at the beginning. If another participant joins that fits a tree it will be inserted into that tree. That entails the problem that the tree can differ from a good solution over time in aspect of the length of the individual paths and a new tree would be more beneficial. This heuristic approach may produce more than necessary false positives in this case because the optimal routes are not used. It seems a better idea to compute the tree again occasionally and adapt the filters to the new routes.

There are also multiple other solutions which construct trees and all have some advantages but also some disadvantages that make them unsuitable for our purpose. Minimum spanning trees, used in the heuristic approach, and Steiner trees are two well-known approaches. Minimum spanning tree algorithms are presented in [36] and in [46]. These are the mostly known basic algorithms. But there also exist new algorithms that are optimised like the one in [18].

A minimum spanning tree is a tree, which connects all vertices in a graph $G(V, E)$ with the minimum overall weight. For this every edge has a weight and the optimal solution according to the weight is chosen. The number of edges to connect every vertex is $|V - 1|$ and therefore the number of edges in a minimum spanning tree is the same. The disadvantage in this solution is the possibility of a very long individual delay. There is no upper bound on the maximum delay between two vertices because in the worst case there are all $|V-1|$ edges between two vertices. This is the case if the minimum spanning tree is a chain of vertices.

Something similar is the case for Steiner tree algorithms. As with minimum spanning trees, there are multiple algorithms, even some for integer and linear programming [3, 4, 50, 51]. Again, the length of a path between two vertices is not limited by any bound. We can get the same problem as with the minimum spanning tree, where some paths are considerably longer than the others. This should be avoided. The individual path length of every path should be around the same value and therefore limited. With this, we can be sure that the events will get delivered around the same point in time.

The approach in [32] tackles the problem of the path length. It uses a minimum spanning tree and a shortest path tree to choose the best path for every vertex and can guarantee a certain length and a certain weight of the overall tree. But the drawback is the formulation for only one root. Normally, this is sufficient and a tree only has one root, but in our case we would need more than one root, because we have more than one publisher. Although this approach is interesting, it would not be sufficient. We would need to extend it so that we can have more than one root and we also need to formulate it as a constraint oriented program, which can be problematic as this algorithm uses iterations to achieve its goal.

There exist implemented publish/subscribe systems, some of them use one of the techniques described before, some have their own concept. A good overview of such systems is given in [39] and [5]. We will take a look at some of them but we must mention that none of these systems is designed to work together with SDN. We will concentrate on systems deploying content-based routing. Some of the systems have a broker-based system architecture and some rely on a peer-to-peer architecture. We do not have a broker based system anymore. The logic of the brokers was transferred to the network itself. The flows on each switch implement the logic of the forwarding of the messages to the correct subscriber and the controller prepares the logic by computing the routes. Even so, we can use some of the knowledge of these systems to build our own framework.

3.1 LIPSIN

Our first candidate is LIPSIN. Although it is a topic-based approach it has quite a few similarities with our approach in terms of the layer on which it operates. The idea, the same as in SDN, is to utilise the network layer directly and manipulate the forwarding tables so that we can prevent the overhead of the application layer. Similar to it, it also contains a control plane and a data plane where the control plane has the functionality to maintain connectivity between participants and establish this connectivity. The data plane takes care of the forwarding of the messages as well as transport functions. But different to SDN, this approach is completely based on the publish/subscribe paradigm which means that we would need a completely different architecture to deploy it. We have two phases in this system. The first is the so called *recursive bootstrapping* where the control plane on the switch discovers the underlying topology and communicates with other entities to get a global view of the network. The second one is the forwarding where link IDs and Bloom Filters, to encode those IDs, are used to realise the forwarding. A message is forwarded by a forwarding tree where the message has information about the links it needs to pass in the header. The Bloom Filters are used to create the headers by encoding the link IDs. The matching itself is then a simple AND operation at the switch.

Although this approach can achieve good performance it has the disadvantage of expressiveness by using a topic-based approach. Additionally, the Bloom Filters can produce false positives which can lead to unneeded traffic consumption. But this approach also shows what is possible by using the network layer with respect to performance.

3.2 SIENA

Our second candidate is SIENA [13]. Basically it is an approach to the same topic as Gryphon, a broker and content based publish/subscribe system. The difference is the matching algorithm for the subscriptions. The approach to the event distribution is similar to the one from Gryphon. The goal is to reduce overhead, that means not all events are forwarded, no flooding takes place. Events are only forwarded to areas where subscribers are present. The brokers forward the event to its neighbours, when they have a subscription. This is determined by filter matching. A subscription is forwarded from subscribers to publishers but only when the broker does not have a subscription that already includes the new subscription. The matching is done with so called binary decision diagrams [12]. These diagrams are the representation of a Boolean function from the subscription and the filters emanate from them. Subscriptions can share sub-expressions and these sub-expressions make the whole system efficient. These sub-expressions are only compared once. But the binary decision diagrams are also limited in their expressiveness and you cannot have too many attributes.

3.3 Hermes

The third candidate, Hermes, is more than a simple publish/subscribe system [44, 45]. It is a middleware that also includes security and type checking. But the logic is, like with the other two systems, on the application layer. A broker network is used to forward the events accordingly. At first, a rendezvous vertex is negotiated to which the advertisements and subscriptions are routed. The publications will be routed on the reverse path from this rendezvous vertex to the subscribers, but it is not necessary to route all the publications through the rendezvous vertex. The vertex is only necessary for the exchange of advertisements and subscriptions. By this the routes are computed in a distributed way and do not need a global view. Also publications without new information are not forwarded. Additionally the subscriptions can have a filter for filtering out the messages the subscribers do not want from the publication. The forwarding is done in the same manner as without filters. The whole approach is on the application layer with multiple layers itself which add to the overhead but bring additional benefits like security and failure tolerance. The subscription language is designed to work well with programming language objects. One can map objects directly to the language.

3.4 Meghdoot

The fourth candidate is Meghdoot, a system which is brokerless and operates on a peer-to-peer basis [27]. Obviously, this system has a completely different approach to forward the messages, because we have no brokers anymore. The approach they use is from a different work named CAN [48, 61], the so called distributed hash tables. The subscription space is divided into a logical $2n$ space where each n is an attribute. The peers maintain information about their zone and neighbouring zones with the appropriate IP address. One peer in each zone is responsible for the zone. The information about the zones is used for the routing. For every event an event point is computed and the event is forwarded to the peer responsible for the zone in which the event point relates to. This makes a broker system unnecessary and ensures the delivery in a distributed way. The peers do not need any global knowledge but without that it is not sure if the optimal solution is reached.

If we want to achieve efficient use of bandwidth, a low average delay and individual delay and load balancing with an optimal solution, we cannot use any of the presented systems. They cannot guarantee all of our goals, so we need to formulate a new solution. This solution is the topic of this work and shall be presented in the following chapters.

Chapter 4

Modelling the elements in a publish/subscribe system

In this chapter, we want to identify and formulate several optimisation problems. They can be extracted from the overall optimisation task and can be regarded as autonomous problems. It is important for our framework later that these problems can also be solved autonomously. Additionally we will also transformations that are necessary to complete the overall optimisation. A short overview of topics handled in this chapter along with their classification is given in figure 4.1.

As the basic topology we have a directed multi-graph $G(V, E)$ with vertices V and edges E. This represents the topology of the switches and the connections between them. For further reference and more detail we will introduce the switches denoted by $R = \{r_1, \ldots, r_n\}$ which is equal to the vertices in the graph but expresses more precisely our intentions in our models. Additionally we have a set of publishers $P = \{p_1, \ldots, p_n\}$ and a set of subscribers $S = \{s_1, \ldots, s_n\}$. For $p \in P$ and $s \in S$, we write $s^{Sub} \subset p^{Pub}$ if events of p match subscriptions s. The event rate which is streamed from p to s is determined by the rate at which p produces events, say λ_p and the overlap between s^{Sub} and p^{Pub}, say $s^{Sub} \cap p^{Pub}$.

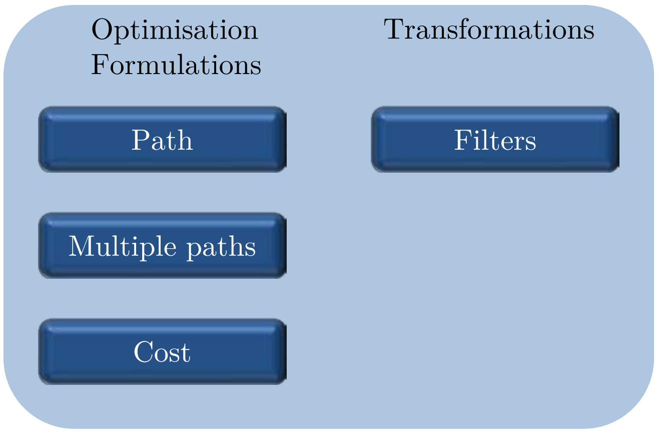


Figure 4.1: Overview and classification of the topics of the chapter

Path	Multiple Paths	Filters	Cost
Connectivity between two vertices	Connectivity between multiple vertices	Distinction of messages	Cost of the system

Table 4.1: Comparison between the formulations

4.1 Path

In a publish/subscribe system there exists at least one publisher and one subscriber. If the subscriber is interested in the advertisement of the publisher, then there must exist a connection between those two for ensuring the delivery. A path can also constrain the delay between two participants as it can constrain the number of hops between two participants. Therefore, the first formulation is that of a path between two participants of the system. We will define a path in three granularities, one by vertices, one by edges, and a hybrid of both. These three notations are necessary because a path can be used in different scenarios, which require different formulations. If there are multiple paths, then there also exists a shortest path, which we aim for later, but first we will model the path because a shortest path means also a low delay.

4.1.1 Path by Vertices

A path that is described by its vertices is a chain of vertices where each pair is connected and the first and the last vertex are fixed. Formally, we are looking to find a path for each possible pair of publishers and subscribers $(p, s) \in P \times S$ where $s^{Sub} \subset p^{Pub}$ and n defines the maximal length of a path

$$X_{p,s} = (x^1_{p,s}, ..., x^n_{p,s}). \tag{4.1}$$

Over each path events are streamed at rate

$$\lambda_{p,s} := \lambda_p \ \cdot \ s^{Sub} \cap p^{Pub}. \tag{4.2}$$

Each variable $x^i_{p,s}$ is assigned by the solver of the program to be a value in $R \cup \{0\}$. Paths need only to be deployed if they are needed:

$$\forall i \ x^i_{p,s} = 0 \text{ if } \ s^{Sub} \cap p^{Pub} = \emptyset. \tag{4.3}$$

Otherwise a path should always start in p and end in s.

$$x^1_{p,s} = r_p \in R \text{ s.t. } p \text{ is connected to } r_p \text{ if } s^{Sub} \cap p^{Pub} \neq \emptyset \tag{4.4}$$

$$x^n_{p,s} = r_s \in R \text{ s.t. } s \text{ is connected to } r_s \text{ if } s^{Sub} \cap p^{Pub} \neq \emptyset \tag{4.5}$$

$$x^i_{p,s} = r_i \in R \text{ if } s \cap p \neq \emptyset \tag{4.6}$$

To establish a valid path X, we need to enforce meaningful transitions in consecutive variables of X by defining constraints for the cost function. The cost function will be explained in more detail later on, we only need it here to show the connectivity of the path. Since paths can be of different length, we allow a path to consecutively utilise the same switch, i.e.,

$$c(x^i_{p,s}, x^{i+1}_{p,s}) = 0 \text{ if } x^i_{p,s} = x^{i+1}_{p,s}. \tag{4.7}$$

Furthermore, we have to ensure that the path for $x^1_{p,s}, ..., x^n_{p,s}$ is connected. Therefore we assign cost of ∞ to all pairs of switches which do not share a physical link in the network topology. That means given a physical topology $L : R \times R \Rightarrow 0, 1$ with

$$L(r_i, r_j) = 1 \text{ if } r_i \text{ and } r_j \text{ are connected} \tag{4.8}$$

$$L(r_i, r_j) = 0 \text{ otherwise} \tag{4.9}$$

then

$$c(x^i_{p,s}, x^{i+1}_{p,s}) = \infty \text{ if } L(x^i_{p,s}, x^{i+1}_{p,s}) = 0 \tag{4.10}$$

$$c(x^i_{p,s}, x^{i+1}_{p,s}) = w \text{ if } L(x^i_{p,s}, x^{i+1}_{p,s}) = 1 \tag{4.11}$$

where w is the weight of the edge.

With this, we have defined a single path over vertices, which are guaranteed connected.

4.1.2 Path by Edges

There also exists an alternative formulation to the problem. In the alternative, the path is defined by the edges between the switches instead of the switches which lie on the path. The path is not specifically stated like in the previous formulation, in particular, the order of the edges is not defined, but rather through the enabling of the use of the edge in a matrix of all the edges. Therefore, the assumption in this case is the existence of a matrix

$$y_{i,j} \text{ where } i, j \in R \cup S \cup P \tag{4.12}$$

$$y_{i,j} = \{0, 1\} \tag{4.13}$$

that represents all the edges with the properties

$$y_{i,j} = 1 \text{ if the edge is used and} \tag{4.14}$$

$$y_{i,j} = 0 \text{ if the edge is not used.} \tag{4.15}$$

We must include the publishers and subscribers into the set of endpoints for edges because we need to define the starting edge and the ending edge. These edges can only be the one from the publisher to the switch and from the switch to the subscriber. The other edges need to be flexible in their deployment.

Path by vertices	Path by edges	Hybrid
Set of vertices	Set of edges (including edges to publishers/subscribers)	Both sets (not including edges to publishers/subscribers)
Connectivity by cost	Connectivity by sum of edges	Connectivity by cost
List of vertices	Matrix of edges	Both

Table 4.2: Comparison between the formulations

A path for a publisher $p \in P$ and a subscriber $s \in S$ has therefore the following constraints:

$$y_{p,r_p} = 1 \text{ for } r_p \in R \text{ s.t } p \text{ is connected to } r_p \text{ if } s^{Sub} \cap p^{Pub} \neq \emptyset \tag{4.16}$$

$$y_{r_s,s} = 1 \text{ for } r_s \in R \text{ s.t } r_s \text{ is connected to } s \text{ if } s^{Sub} \cap p^{Pub} \neq \emptyset \tag{4.17}$$

$$\sum y_{i,j} = \sum y_{j,k}. \tag{4.18}$$

The last constraint is necessary to ensure connectivity. Contrary to the previous formulation, we do not enforce connectivity through the costs but through the number of outgoing and incoming edges. They need to have the same amount respectively their sum is zero.

4.1.3 Hybrid

We will introduce a third formulation that merges the two preceding formulations. It may seem unnecessary at first but we will see in the next section why this formulation is indeed necessary and convenient. We will not take all the parts of each formulation but enough to have a formulation that is satisfying enough for our intention.

We will take the path formulation from the path by vertices from equation (4.1) on page 16 and ensure connectivity through the cost between each pair of vertices. Additionally, we will take the matrix of the formulation by edges from equation (4.13) on the preceding page, but we can reduce the size of the matrix because we will not need the edges from publishers to switches and switches to subscribers. We will not adopt the constraints of the path by vertex formulation because they are not necessary if we have the other formulation. In table 4.2 a comparison between the formulations is given.

Additionally we need a mapping of the list of vertices to the matrix, which represents the edges. This mapping is done for every pair of consecutive vertices in the path. That is to say, every edge $y_{v_i,v_{i+1}}$ is set to one. Of course, this is overhead and it does not seem obvious why we should need it at this point, but the next chapter will give us more information about this.

4.2 Sub Graphs

In a publish/subscribe system we have more than one path. To model this we take the hybrid formulation because it is easy to extend it to multiple paths. That is also the reason for the hybrid formulation, to have a possibility to extend the path formulation to multiple paths and merge them into one formulation. This representation of multiple paths is also very convenient in respect to our goal, optimising.

Basically we have two kinds of sub graphs. The first we can call shortest path trees where there is one publisher with all its subscribers The second is the general sub graph where multiple publishers and subscribers can occur. In both cases, we can do pretty much the same as in the hybrid formulation, but with more than one path. For every start and end point pair a path $X_{p,s}$ will be formulated and the constraints for the connectivity are enforced. Additionally one matrix with all edges is needed. Therefore, what we have is a set of paths and a matrix

$$\{X_{p,s}^1, \dots, X_{p,s}^s\} \tag{4.19}$$

$$y_{i,j} = \{0,1\} : i,j \in R \tag{4.20}$$

The mapping is the same as in the hybrid formulation but will be done for every path. The only difference is the fact that if the edge is already set then no change will occur. The edge will be set to one, only if it is not set.

We will see the advantage of this formulation when we address the cost function later on. However, one thing in advance, it is easier to use a matrix for this. Also positive is the fact that we still have each individual path, which we can use later on in the section that addresses filters. Nevertheless, it also introduces a disadvantage that we will cover later on.

4.3 Filters

We mentioned filters before and used them as a tool to tell different subscriptions and advertisements apart. Now we look at them in more detail for providing a detailed explanation.

The task of filters in our scenario is to encode information about a subscription or advertisement. This is done by using binary strings, called a dz expression, as presented in [34], as filters. This binary string can encode the content of an event, subscriptions, or advertisement. A dz expression is defined as follows:

$$dz = \{dz^1, \dots, dz^n\} : dz^i \in \{0,1\} \tag{4.21}$$

Every digit represents a dimension in the space of possible attributes. The dz expressions can have a different length or number of dimensions, but they all have a maximum number of dimensions n. The dimensions are computed according to the content that is available, and can represent this content. The structure of a dz expression perfectly matches an IP address and so we only need to reserve an IP address space and can use these IP addresses as our filters. With this, we can use OpenFlow flows and set the matching to the IP address, which we previously computed. Without much effort we have a way to deploy the expressions directly and have a basic publish/subscribe system, although there can still be some complications.

With the definition of a dz expression we can formulate what the expression must be capable of. Let $Dz(r_i, r_j)$ denote one of the dz established between r_i and r_j. Obviously, we require

$$Dz(r_i, r_j) = \emptyset \text{ if } L(r_i, r_j) = 0. \tag{4.22}$$

Otherwise

$$Dz(r_i, r_j) \supset dz(s \cap p) \text{ if } \exists (x^i_{p,s}, x^j_{p,s}) \in X_{p,s}. \tag{4.23}$$

To simplify the expression above we define

$$Dz(r_i, r_j, p, s) := \perp \text{ if } L(r_i, r_j) = 0 \tag{4.24}$$

and otherwise

$$Dz(r_i, r_j, p, s) := dz(s \cap p) \text{ if } \exists (x^i_{p,s}, x^j_{p,s}) \in X_{p,s}. \tag{4.25}$$

From this, we can estimate the number of flows and overlaps. For instance,

$$flows(r_i) := \sum_{p,s,r_j} (Dz(r_i, r_j, p, s) \neq \perp) \tag{4.26}$$

gives the number of flows on a switch. If we constrain this number, we can prevent overloading of a switch if we include the data rate of the flows.

Furthermore we have some additional properties of the dz expression, that are needed. We will describe these properties in accordance with the formulation by vertices. In general, the filter needs to forward only those messages that are really needed later on and needs to cut off the messages that are not needed anymore. However, it is possible that filters on a path will get longer or shorter from time to time. This is due to our goal to optimise, which means, share paths. Because of this we cannot set exact constraints for the dz expressions. The only condition, that counts at any time, is that the filter must be the super set of the dz of the publisher and subscriber. Therefore

$$Dz(x^i_{p,s}) \supseteq dz(p^{Pub} \cap s^{Sub}) \text{ if } s^{Sub} \subset p^{Pub}. \tag{4.27}$$

The dz expressions can vary from switch to switch if one expression is shared by different paths. The expression can get longer or shorter which depends on the paths that are merged. Generally, there are two basic possibilities. The first is that previously shared filters split and the individual filters get longer or stay the same, which means they are more specialised and are therefore a sub set of the previous filter.

$$\forall(x_i, x_j) : i \leq j \Rightarrow Dz(x^j_{p,s}) \subseteq Dz(x^i_{p,s}) \tag{4.28}$$

The second possibility is that multiple filters come together on one edge and are merged into one less specialised filter, which is shorter and a super set of all the previous filters.

$$\forall(x_{i_l}, x_k) : i_l \leq k \Rightarrow Dz(x^k_{p,s_{\sum_l}}) \supseteq \sum_l Dz(x^{i_l}_{p,s_l}) \tag{4.29}$$

The expression on the last switch, on which the subscriber is connected, should be exactly the dz expression of the subscriber.

$$Dz(x^n_{p,s}) \subseteq dz(s) \text{ if } s^{Sub} \subset p^{Pub} \tag{4.30}$$

The dz expression on the first switch is dependent from the following filters like all other filters except the one that delivers messages to a subscriber.

Of course we also need a minimum and maximum length of the dz expression. Whereupon the maximum length depends on the IP space, we get and is therefore already set. Therefore we define $l(Dz) = i : i \in \mathbb{N}\ i = \{1, \ldots, n\}$ where n is the maximum length, set by the IP space. With this, we can define that all the dz expressions must be equal or greater than one:

$$\forall i : l(Dz(x^i)) \geq 1 \tag{4.31}$$

Also

$$\forall i : l(Dz(x^i)) \leq n \tag{4.32}$$

must apply.

At last, we have the possibility to compare filters. Not only to compare if they are the same but also to check if they share a common prefix and are therefore similar. We have two dz expressions, dz_1 and dz_2. What we want to know is how similar they are, that is to say, how long the prefix is that they share. The use for this will be later explained in the section about clustering. The comparison can be done by simple pattern matching. We generate a third string, which is the shared dz expression, dz_s with not previously known length. This represents the prefix of the two other expressions.

$$dz^i_s = 1 \text{ if } dz^i_1 = dz^i_2 = 1 \tag{4.33}$$

$$dz^i_s = 0 \text{ if } dz^i_1 = dz^i_2 = 0 \tag{4.34}$$

If $dz^i_1 \neq dz^i_2$ then this means that the prefix has ended.

We also introduce the function

$$DzS(dz_i, dz_j) = dz_s \text{ if } dz_i, dz_j \text{ have a shared prefix} \tag{4.35}$$

$$DzS(dz_i, dz_j) = \emptyset \text{ if } dz_i, dz_j \text{ have no shared prefix} \tag{4.36}$$

which results in a dz_s if the two dz expressions have a common prefix and an empty string if they do not have a common prefix.

4.4 Cost

Before we can define the cost function and its purpose we need to define what a weight and what its purpose is. The weights in our graph have the assignment to control the load in the system. We can manipulate these weights, they are not fixed. A lot of the optimisation will be done through the manipulation of the weights of the edges. First, we need to define what exactly the weights are and what characteristics apply.

The value of a weight $w_{i,j}$, $i, j \in R$, itself is element of $\mathbb{N}$ and has an upper bound B_u and a lower bound B_l.

$$w_{i,j} \in \{B_l, \dots, B_u\} \in \mathbb{N} \tag{4.37}$$

At the beginning, each edge in the graph has the same upper and lower boundaries but these boundaries can change from graph to graph. With the size of the interval, the granularity can be chosen. For some optimisation scenarios it can be necessary to have a very fine granularity, for others it is sufficient to have a coarse granularity. The bounds of the weight can be changed during an optimisation to converge the bound to an absolute value at the end.

With the weights and its properties defined, we can attend to the cost function. On the one hand, the cost function ensures the connectivity for our path formulation by vertices and on the other hand, the cost function obviously calculates and represents the costs of our choice of edges. Luckily, we can define a very easy cost function due to our definition of the weights. The weights of the edges as well as the boundaries of the weight can be changed. This allows us to outsource the weight computation and to concentrate on the cost function by assuming a weight matrix for every weight. The cost for taking an edge is therefore the weight of the edge itself.

$$c(i, j) = w_{i,j} \in \{B_l, \dots, B_u\} \in \mathbb{N} \tag{4.38}$$

Now we only need to multiply the weight matrix with the matrix of used edges, $y_{i,j}$, and we have the actual weight for the optimisation process. Afterwards the weights need to be updated with the new load on the edges.

The load on the edges can be expressed in form of the streaming rate over this edge. Every path that uses this edge and has not exactly the same messages as another path needs to be added. However, we will not only add the value of the streaming rate because we want that our two bounds convert to one absolute value and the graph has the optimal weight for the optimal routes. This is also convenient later in the clustering and fragmentation part. We can express the influence of other paths and even the influence of other applications this way. The overall streaming rate of a switch is

$$\lambda_{r_i} := \sum_{X_{p,s}} (\lambda_{p,s}) : \forall X_{p,s} \text{ where } r_i \in X_{p,s} \tag{4.39}$$

However, we can subtract the messages that are duplicates. Those will be forwarded only once. Of course, there needs to be a constraint that limits the stream rate for a switch like

$$\lambda_{r_i} \leq a. \tag{4.40}$$

where a is a certain value that cannot be exceeded.

The cost function has also the task to assure sharing of paths and optimising the bandwidth usage. The bandwidth usage is conveniently represented by the streaming rate, which influences the weight and automatically has an influence on the cost function. The sharing of the paths is done through the optimisation process where the result has the lowest weight. To achieve this, the paths need to be shared.

Through the definition of the cost function for unconnected switches, the cost function also ensures the connectivity of a path without any more effort. Additional constraints for the number of flows on a switch and the streaming rate of a switch balance out the load of a switch. For the length of a path, we do not need additional constraints, the path formulation takes care of that.

Now we can fully concentrate on finding the best balance between overall bandwidth usage, individual bandwidth usage on each switch, average delay and individual delay and of course load balancing.

Chapter 5

Sub Solution Formulation

In the previous chapter we presented several models that we need in our system. Now we will formulate the problem and the solution thereof. Some problems have multiple solutions each with its advantages and disadvantages. The idea is to select one solution for each problem and put them together into a framework which we will present in the next chapter. The problems and therefore the solutions are dependent on each other in this framework. So we will also present the input and output of each solution to give an overview how they can interact. We will also present solutions to the transformation problems and mechanisms which we presented in the previous chapter. Solutions in the framework can be exchanged for other solutions that solve the same part of the overall problem. For slightly different problems it can be sensible to choose another solution.

Some of the problems can be merged and solved in one solution. This can have its advantages but also its disadvantages. An advantage is certainly the knowledge that is combined in one program. We do not need mechanisms to communicate between the components or to transform the output. But combining multiple problems will most likely increase the complexity and also the time to execute the program. Although the communication can be an overhead the structure of the framework with multiple components increases its flexibility. The simple exchangeability of each part with a different one is very convenient and would be destroyed if we merge too many parts together. With the concept of exchangeability of parts we can easily optimise in different directions if we change a few components. We can choose the optimisation goal or the impact of parameters which can be the weight, the delay, load balancing or something completely different.

5.1 Optimisation solutions

At first we will present the optimisation solutions. These will be solving the optimisation models we presented in chapter 4 on page 15.

5.1.1 Shortest Path Tree Solution

The first solution that is presented addresses the problem of finding a path between two participants. Later in this subsection a solution for the problem of one publisher and multiple subscribers is presented. These solutions are straightforward and easy to implement. They also fit well in a multi-component framework.

Compute Shortest Path

The formulation for a program to search for the shortest path between two vertices is easy and this program solves the problem optimally.

The input to this computation is the topology with its weights $w_{i,j}$, $i, j \in R$, on the one hand and the publisher/subscriber pair on the other. That is to say both ends of the path are defined. The topology is a graph $G = (V, E)$ but only the edges are of interest to us. The input of the program is the set of edges $e_i \in E$ in G

$$I_{SP} = \{e_1, \dots, e_n\} \tag{5.1}$$

where two edges are already set. The two edges, that are set, are the one from the publisher to its switch and from the subscriber to its switch.

The output is therefore a set of shortest path

$$O_{SP} \subseteq I_{SP} = SP. \tag{5.2}$$

The program itself is the following:

$$\begin{aligned} \text{Min} \quad & \sum c_{i,j} y_{i,j} \quad i, j \in R && (5.3) \\ \text{Subject to} \quad & \sum y_{1,j} = 1 && (5.4) \\ & \sum y_{k,N} = 1 && (5.5) \\ & \sum y_{i,j} = \sum y_{j,k} && (5.6) \\ & y_{i,j} = \{0, 1\} && (5.7) \end{aligned}$$

$y_{i,j}$ are the edges and $c_{i,j}$ the cost of the edges which was explained in section 4.1.2 on page 17.

We minimise the amount of edges in the solution and have the connectivity as a constraint. This is formulated over the incoming and outgoing edges; their sum has to be zero except at the start and end point. It is possible to compute multiple shortest paths for one publisher/subscriber pair by excluding the previous solutions through constraints. This can be used in the next step, presented in the next sub subsection. It is also possible to get the optimal and some sub-optimal solutions with the same computation. Sub-optimal solutions can be used in the same way as optimal solutions in the next steps to give more variety for further processing.

If we run the computation several times we will have more than one shortest path in the set of shortest paths. Sub-optimal values for one path can also be put into the set if we have a further step that can make use of these solutions. It is also possible to incorporate a more sophisticated mechanism to prepare the results for further computation but that could also be too much insofar that we would solve the next step in this step already.

Either way we will take the formulation of a path by edges for a single shortest path where the path is a matrix where the used edges are marked.

$$y_{i,j} = \begin{cases} 0, & \text{if edge is not used} \\ 1, & \text{if edge is used} \end{cases} \tag{5.8}$$

This format will later be transformed into the hybrid formulation when we have more than one path to describe. This transformation can be easily done because we already have the matrix formulations of all paths. We only need to set the edges in the new matrix.

Advantages

1. Fast computation through implementation in linear programming.
2. Ensured connectivity for a pair of vertices on a path through the formulation by the edges.

Disadvantages

1. Very heavy tree with lots of filters and lots of messages because only individual paths are considered.

Build Shortest Path Trees

With the given set of paths, a tree for each publisher and its subscribers can be computed. We have two options in this case. The first is just adding all the paths to one matrix when we have only computed one path for each pair of publisher and subscriber. The second option is choosing the best path for every pair of publisher and subscriber out of a set of paths for those two. The choice of the option depends on the choice in the previous step. When we have chosen to compute only one path for every pair we can only do the first choice. The second option would not differ from the first one because we cannot choose between several paths. If we have several paths for every pair we need to take the second option and choose the best path.

When we have only one path for each pair and even if we have several paths but not all for each pair, we cannot guarantee that we will find the overall optimal solution for the tree this way. Even if only one path that would be needed for the optimal solution is missing, then we would have a loss of quality in the solution, which means we cannot reach the optimal solution but only a sub-optimal one. This is the case for every path that we choose but is not the optimal one for the tree.

The input I_{SPT} of the problem is a set of a set of shortest paths and a shortest path is a set of edges that is the subset of the topology.

$$I_{SPT} = \{SP_1, \ldots, SP_n\} \tag{5.9}$$

For each set of shortest paths a shortest path tree for a publisher will be computed. The output is therefore a set of shortest path trees

$$O_{SPT} = \{SPT_{p_1}, \ldots, SPT_{p_n}\} \tag{5.10}$$

The approach to the optimisation is straightforward. We need to select one path out of each set for a pair of endpoints for all pairs and the overall solution should be optimal in respect to the cost. The minimisation of the costs can be done through sharing of paths. Sharing effectively reduces the costs by helping to send a lesser amount of messages. But we cannot randomly share some paths because the connectivity of publisher and subscriber must remain.

The formulation of the program is as follows

$$\begin{aligned}
\text{Min} \quad & \sum c_{i,j} y_{i,j} && (5.11)\\
\text{Subject to} \quad & \sum_m B_{n,m} = 1 && (5.12)\\
& \text{if} \quad (Y_{n,m,i,j} = 1 \wedge B_{n,m} = 1) \quad \text{then} \quad y_{i,j} = 1 && (5.13)\\
& Y_{n,m,i,j} = \{0,1\} && (5.14)\\
& B_{n,m} = \{0,1\} && (5.15)
\end{aligned}$$

where $y_{i,j}$ are the edges and $c_{i,j}$ the cost of the edges.

$Y_{n,m,i,j} \in \{0,1\}$ represents the set of edges where the possible paths are represented as a two dimensional matrix $n \times m$, n the number of pairs and m the number of paths for each pair and i and j to identify affiliation. We need to choose one path for each pair that has the maximum amount of shared edges with the other paths and thus the minimal costs.

There can only be one path from each set for a pair because we do not need more than one connection between two end points. More than one connection could even be bad because the subscriber would get duplicated messages. To express this behaviour we introduce exclusivity which means that only one path for a set of paths for a pair can be chosen. Through equation (5.12) we can reach exclusivity for each n.

An edge $y_{i,j}$ is enabled, if a path uses this edge. If another path uses this edge as well, the costs will stay the same. For each chosen path the corresponding matrix $y_{i,j_{p,s}}$ will be merged with the existing matrix $y_{i,j}$ which represents the tree. As we want to reflect the sharing with the cost of an edge, we set the value of $y_{i,j}$ instead of adding. Equation (5.13) on the previous page ensured this. It is not important what the value of $y_{i,j}$ was before this execution, the only thing of importance is the value after execution. If the edge was enabled already, nothing changes. But if the edge was not enabled, it is now enabled.

Advantages

1. Groundwork can be done easily.
2. Dynamical with respect to number of paths for each pair.

Disadvantages

1. May not be optimal if all paths were not computed.

Combination Solution

Both of the computations above can be merged into one computation. But we must keep in mind that if we do more of the problem in one step, the step itself will get more complicated and will take more time. On the other hand, the quality of the solution could be better if we combine the steps. If we separate the steps, we would need to compute all possible solution sets to compute the optimal solution in the next step. We would need to do something similar in the combination solution, but it is nonetheless more efficient as we do not need to compute all solutions for the shortest paths. The basic approach is to break away from the idea of shortest path and consider all participants at once. Then we can find an overall solution more efficient. But of course we ensure the connectivity of publish subscriber pairs.

The input of this combination solution would be the same as in the shortest path solution with addition of the publishers and subscribers

$$I_{CS} = I_{SP} = \{e_1, \ldots, e_n, p_1, \ldots, p_n, s_1, \ldots, s_n\} \tag{5.16}$$

but the output will differ a bit from the previous solution. We will again have a set of edges that differs from the input

$$O_{CS} = \{e_1, \ldots, e_m\} \tag{5.17}$$

The formulation of such a program can be realised as follows. We define a path for each publisher/subscriber pair. This path is defined through a series of vertices. The first vertex is the switch to which the publisher is connected and the last one is the switch to which the subscriber is connected.

$$
\begin{aligned}
\text{Min} \quad & \sum_{i,j} c_{i,j} y_{i,j} && (5.18)\\
\text{Subject to} \quad & X_{l,p,s} \quad \forall x^i_{l,p,s} \in X_{l,p,s} \in R && (5.19)\\
& x^1_{l,p,s} = r_p \in R && (5.20)\\
& x^n_{l,p,s} = r_s \in R && (5.21)\\
& \sum_{r \in R} B(l,r,n) = 1 && (5.22)\\
& \sum_{r \in R} r * B(l,r,n) = X_{l,p,s} && (5.23)\\
& y_{x^i_{l,p,s}, x^{i+1}_{l,p,s}} = 1 \quad \forall x^i_{l,p,s} \in X_{l,p,s} && (5.24)
\end{aligned}
$$

So we have a set of paths in equation (5.19) following the formulation in section 4.1.1 on page 16 for a path. The index l gives us the number of paths and n in equation (5.21) the length of the path. A further variable $B_{l,r,n}$ is needed so that only one $r \in R$ is chosen for every position in $X_{l,p,s}$. This is done through equation (5.22) and the switch for this position is constrained to be in R through equation (5.23).

The connectivity is already ensured over the costs of each edge. If there is no connection between two switches, then this edge will have an infinite weight. Now we just define all the paths from the publisher to its subscribers and minimise the costs over it. The sharing of the paths is done automatically through the cost function. The mapping is done in equation (5.24) which maps the enabled edges of a path to the overall edges represented in $y_{i,j}$.

This solution is very variable and can be used for several publishers and their subscribers at once. We only need to take care that the included participants share a common prefix for their filters. As a drawback this solution can only be implemented as a mixed constraint oriented program because ensuring connectivity through the costs has its price.

Advantages

1. Optimisation in one step.
2. Multiple publishers are supported.
3. Weights can be used to express load.

Disadvantages

1. Consumes more time if the network is big.
2. Only implementation as mixed constraint oriented program.

5.1.2 Minimum Spanning Tree Solution

There is another solution to compute paths or rather a solution to compute a tree for a publisher, a minimum spanning tree. A minimum spanning tree is a tree with minimum weight that connects all the vertices. One could first compute a minimum spanning tree for each publisher and its subscribers. The paths will then be embedded in this spanning tree. After that multiple spanning trees can be merged by filters.

The input of this problem is the same as in the shortest path solution, the set of edges

$$I_{MST} = \{e_1, \ldots, e_n\} \tag{5.25}$$

and the output is a minimum spanning tree for the participating vertices, which is again a set of edges

$$O_{MST} = \{e_1, \ldots, e_m\} = MST \tag{5.26}$$

We can use a sub graph instead of the whole graph if we do not need all the vertices. But it is not a good idea to compute a spanning tree for all publishers and subscribers at once, except when the tree is balanced. But even then it is not a good idea and it would be better to group them together according to their filters. On the one hand, we want to have the least possible amount of messages, which means the least amount of edges, on the other hand, we do not want to overload some of the switches. We want to distribute the load of the switches evenly. We could do this by computing spanning trees for publishers and merging them when necessary. But we would need a fast way to compute spanning trees. Luckily, this is possible, at least with normal programming. There are polynomial algorithms for the minimal spanning tree problem. A linear programming solution is presented in [21].

$$\begin{align}
\text{Min} \quad & \sum c^T s \tag{5.27}\\
\text{Subject to} \quad & s(\gamma(U)) \leq |U| - 1 \quad \forall U, \emptyset \neq U \subset V \tag{5.28}\\
& s(E) = |V| - 1 \tag{5.29}\\
& s_e \leq 0 \quad \forall e \in E \tag{5.30}
\end{align}$$

We have the cost function in equation (5.27) where T is the edge set of a spanning tree and we minimise the cost function. Thereby c is the cost vector with the costs of each edge. Furthermore we have a set of vertices $U \subset V$ and $\gamma(U)$ which marks the edges which have both endpoints in U. The shortcut $q(B)$ is used to indicate $\sum(q_j : j \in B)$ for any set A and $B \subseteq A$ and vector $q \in \mathbb{R}^A$.

We see that it is indeed possible but it brings its problems if we formulate it in a linear program. One such problem is the huge amount of constraints that we would need. But minimum spanning trees also have the drawback that some paths can get really long. The worst case scenario is a chain of vertices. That is an undesired effect that we want to avoid. Although spanning trees have their advantages we need to look at a different approach to ensure a maximal path length.

Advantages

1. Known and fast algorithms.
2. Minimal weight.

Disadvantage

1. Linear programming formulation is inefficient.
2. Individual maximum path length cannot be guaranteed.

5.1.3 Balanced Trees

An approach that eliminates the problem of the path length and that can guarantee that the weight of the tree is not higher than a certain factor of the minimum spanning tree is presented in [32]. The ideas of a shortest path tree and a minimal spanning are combined. The goal is to get a good balance between minimising the total number of edges used and also minimising the delay between the participants through the length of the individual paths. Therefore the minimum spanning tree and the shortest path tree will be computed. An iteration, started from the root of the minimum spanning, begins and checks for each vertex if the length condition is fulfilled. If it is violated the shortest path for this edge is inserted and the iteration continues.

The input of this problem would be the minimum spanning tree and the shortest path tree

$$I_{BT} = (MST, SPT) \tag{5.31}$$

and the output would be a balanced graph with a maximum path length

$$O_{BT} = BT \tag{5.32}$$

If we try to formulate this as a linear or constraint oriented program we will soon see that it is very similar to the combination solution of the shortest path tree. We will start with a graph that consists of the minimum spanning tree and the shortest path tree. Edges that were not used in one of them are not available in the graph. To ensure the length of every individual path we need to keep track of the length of each path. The easiest way to do this is to have a path variable with a certain length. The vertices that are used for this path are saved in this variable. Then we try to minimise the edges. In other words it would be the same as in the combination solution. The reasons for this are the adjustments that need to be made to transform the solution in a constraint oriented programming one. We cannot simulate iterations very well in constraint oriented programs.

The original formulation will get complicated with multiple roots, also known as publishers. The formulation only considers one root and the length for a path is from this one root towards the vertex. Although this solution is good for one publisher it is not suited for multiple ones. The adjustments for the constraint oriented programming solution have the advantage that we can formulate it with multiple roots, although we need cycle detection and we could just take the combination solution.

Advantages

1. Not the minimal length and minimal weight but balanced results.

Disadvantage

1. Problematic with multiple publishers.
2. Constraint oriented programming formulation is the same as in the combination solution.

5.1.4 Steiner Tree

An often used approach in route optimisation is the Steiner tree algorithm. The Steiner tree algorithm is a combinatorial optimisation algorithm which tries to reduce the number of edges in a graph while it maintains the connectivity for a set of vertices. [30] The computation is done by adding additional vertices to the graph. There exist formulations of the algorithm like in [51] and also linear programming formulations [4] and its optimisations [3]. There also exist approximation algorithms that solve the problem almost optimally and in polynomial time. [50]

The input would be our vertices that we want to connect

$$I_{ST} = V \setminus W \tag{5.33}$$
$$\forall w \in W \subset V : w \neq P_{ST} \land w \neq S_{ST} \tag{5.34}$$

where P_{ST} are the publishers for the Steiner tree and respectively S_{ST} the subscribers.

We borrow the linear formulation from [4]. The vertex set V is divided into U, the normal vertices, and O, the Steiner points, $V = U \cup O$. Steiner points are vertices that are used to decrease the length between the vertices that need to be connected. In our case the Steiner points are the points we do not use. Additionally a tree $T(V', E')$ is defined. This tree is a spanning tree where $U \subset V' \subset V$ and $E' \subset E$. The goal is to find a tree $T^*(V^*, E^*)$, which is optimal with respect to the weight of the tree. This tree would then be the Steiner tree. Since this is a set covering formulation we need additional auxiliary means. First we need to partition V into two disjoint sets, $V = D \cup \overline{D}$, $D \cap \overline{D} = \emptyset$. $U \cap D \neq \emptyset$ and $U \cap \overline{D} \neq \emptyset$. $O = (D, \overline{D})$ is the cut set of edges between D and $\overline{D}$, there are multiple ones. The program is the following:

$$\text{Min} \quad \sum_{j=1}^{m} c_j y_j \tag{5.35}$$

$$\text{Subject to} \quad \sum_{j=1}^{m} a_{i,j} y_j \geq 1 \tag{5.36}$$

$$a_i = \begin{cases} 1, & \text{if edge } e_j \in \text{cut set } O_i \\ 0, & \text{otherwise} \end{cases} \tag{5.37}$$

$$y_j = \{0, 1\} \tag{5.38}$$

The variable $i = 1, \ldots, o$ defines the Steiner points where the maximum amount of points is o and the variable $j = 1, \ldots, m$ the edges in E where the maximum amount is m. The weight for an edge $e \in E'$ is c_j. We try to minimise this weight.

Now we define the set

$$y^* = (y_1^*, \ldots, y_m^*) \tag{5.39}$$

which represents the optimal solution for the program and the set

$$T^* = \{e_j | y_j^* = 1\} \tag{5.40}$$

which represents the optimal solution for the Steiner problem.

Although this is an established algorithm and is used quite often we are confronted with the same problem as in the subsection with the minimum spanning tree. Individual paths can get too long and the tree is not balanced. We cannot ensure the length of a path and that can be a problem if some delays become too high. We want to achieve an overall balanced route where no delay is significantly higher than any other. Besides, we can get a lot of cycles if we compute a tree with multiple publishers and want to use generalised rules for the filters. Also the initialisation of the first graph and the adding of new subscribers and publishers may not be as fast as needed.

Advantages

1. Well researched algorithms.
2. Approximation algorithms in linear time.

Disadvantages

1. Not balanced.
2. Individual paths can be very long.

5.1.5 Centre switches

This approach is similar to the one in section 5.3.1 on page 43. It is the attempt of constructing of a spanning tree with selected switches as centres of the paths. At first, there is only one switch as the centre. Every path from publisher to subscriber is via this switch. The input is therefore the multiple shortest paths and the first and only centre switch CS_1

$$I_{CS} = (SP_1, \ldots, SP_n, CS_1) \tag{5.41}$$

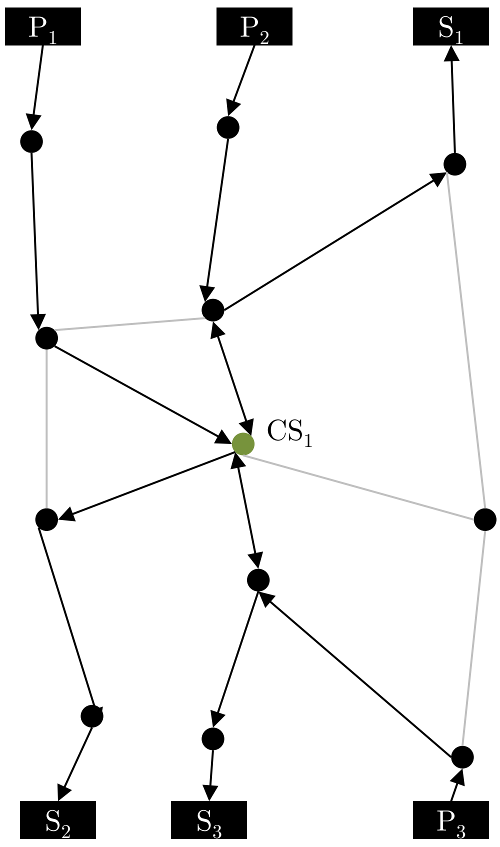
Through this we ensure that there are no cycles, because we control all the paths. But of course if the network has a certain size there would be too many paths via this switch. Therefore we search for more switches that can serve as centre switches. The paths are distributed between them according to the path lengths and the filters. We can reduce the load of the switches and manipulate the length of the paths accordingly. This results in the output of new paths and new centre switches

$$O_{CS} = (SP_1, \ldots, SP_n, CS_1, \ldots, CS_n) \tag{5.42}$$

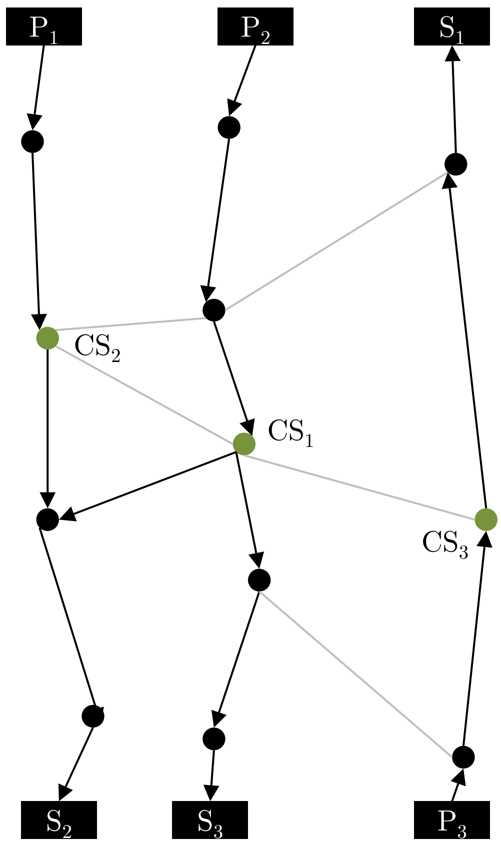
In figure 5.1 on the following page we start with one centre switch and end up with three of them.

But the placement of the switches presents a problem. We only have knowledge about the shortest paths between each publisher and subscriber pair beforehand. We need to compute the quality of the placements of the switches and set them in the right spots. This is the same problem as in the previous solution but maybe we can use mechanisms similar to clustering mechanisms to do that.

It is important to distribute the paths according to the filters or else we would not have any benefit out of it. We then need to compute the shortest paths between the publishers and the centre switches and the centre switches and the publishers. This computation can be done in a very effective way so there is no problem in doing this multiple times. There also exist some paths between centre switches to distribute messages that arrive at one centre switch but the subscriber is connected to another one.



(a) Initial situation



(b) Situation after computation

Figure 5.1: Computation with centre switches

For a linear or even constraint oriented program we need to outsource the filter matching. Although it can be done in a linear program it is not efficient, which is shown later. Also, we need to divide the program further to formulate it. The process if iterating through several problems in one program is not working well, especially if some of the problems have sub problems themselves. The sub steps we need to do are the following:

1. Clustering of the filters.
2. Calculation of suited switches.
3. Computation of the shortest paths.

We have already shown the solution for the last point and the solution for the first point will follow later. The second point was already presented in the previous chapter. But if we do it with sub steps it is possible to get some cycles, only the complete step would prevent them, because only then we have complete control. On the other side, the complete step would be cumbersome to implement as a constraint oriented program.

Advantages

1. No cycles through the means of construction, if done in one step.
2. Centre switches can be chosen, but...
3. Partitioning according to the flows is possible.
4. Minimal amount of edges between centre switches.

Disadvantage

1. Computation can be very time consuming.
2. ... computation of the location is time consuming.
3. Not balanced in matters of path length.

5.1.6 Clustering

To get better results in computation time we can partition the graph into several sub graphs. The overall time will reduce, but it also has an effect on the quality of the solution. When we do not consider all vertices and edges in one computation we need to have some fixed vertices and edges to ensure connectivity between the partitions. These cannot be changed in the optimisation and can therefore reduce the quality. Clustering and fragmentation can both reduce the size of the graph and partition the graph into sub graphs. We will see clustering as a means to group similar participants together according to the content they produce or consume. That means that clustering will be done for the filters and reduces the edges in a sub graph. Fragmentation will be seen as a means to partition those sub graphs even further and reduce the number of vertices that are considered.

Prefix Matching

Prefix matching is one way to compare two filters. It is easy to implement it and it is sufficient for a prototype implementation. There are several algorithms for pattern and prefix matching.

If we want to use the topology in the best possible way and produce the minimum amount of messages and also false positives, we need to merge some paths that do not have the exact same dz expression but a matching prefix. Prefix matching can be done in an easy way with normal programming methods. The naive way would be to compare both dz expressions step by step, one bit at a time. Then one can save the length of the common prefix and the prefix itself. Incidentally the dz expressions are bit strings and can be easily compared. We could also compare intervals of dz expressions which reduces the time but also the accuracy of the prefix.

Before we try to reduce the time in this way, it is a better idea to use more advanced methods and accept more complexity in the implementation.

The input are the two dz expressions that we want to compare, dz_1 and dz_2

$$I_{PM} = (dz_1, dz_2) \tag{5.43}$$

and the output is a prefix, that is also a dz expression, dz_3, that can be empty

$$O_{PM} = dz_3. \tag{5.44}$$

There are optimised ways to do this like in [33] and [62]. But right now the important question is if this problem can be solved as a linear program. Currently there are no attempts to solve pattern matching as a linear program. One of the reasons is that it is not really an optimisation problem. We only compare two strings and give the longest prefix back. The other reason is the amount of constraints we get if we want to compare the expressions bit by bit. We would need to formulate a constraint for each comparison and the longer the expressions are the more constraints we would need. For a linear program it is therefore advisable to divide the expression in some parts and only compare the parts. This gives us lower granularity but lesser constraints and therefore lesser execution time. Additionally we face the problem that we need to formulate a linear program every time we compare two dz expressions and this is will happen very often.

Nonetheless we can formulate the linear program as follows. Again, we have two dz expressions dz_1 and dz_2. These are the expressions that we want to compare, both have a certain length. Additionally we need a variable dz_3 where we save the result of the comparisons. Then we need as many constraints as the length of the expressions are.

$$\begin{aligned} \text{Min/Max} \quad & l(dz_3) && (5.45) \\ \text{Subject to} \quad & dz_3^i = 1 \text{ if } dz_1^i = dz_2^i = 1 && (5.46) \\ & dz_3^i = 0 \text{ if } dz_1^i = dz_2^i = 0 && (5.47) \end{aligned}$$

What we set as our objective function is not important, we just use the length of dz_3 which we can minimise or maximise. The constraints ensure that there is only one value and this value is the length of the prefix.

Although we can formulate a linear program, the better way to do this is with normal programming because even when the linear program is faster in solving the problem we still need to initialise it every time.

Advantages

1. Easy to implement in normal programming languages.

Disadvantages

1. No reasonable linear programming implementation.

Distributed Spectral Cluster Management And Alternatives

Of course there exist several cluster mechanisms like the one in [52], this one is even a linear program, and the one in [58]. For the creation of the filters, [58] is used so the algorithm presented there is a suitable candidate for an optimised version of the framework. For our current goal, it is sufficient to take an easier one so that we do not spend too much effort on this part. Nevertheless, for future implementations of the framework this algorithm can be used.

The input of the problem is a set of dz expressions

$$I_{SC} = (dz_1, \ldots, dz_n) \tag{5.48}$$

and the output is a clustering of the dz expressions, which means k clusters cl of dz expressions

$$O_{SC} = (cl_1, \ldots, cl_k). \tag{5.49}$$

Right now we will formulate the program according to [52], but first we will look at a problem that arises with clustering. In clustering algorithms there exists a distance, which can be Euclidean, rectilinear or other distances. This distance is used as a way to decide which entities are clustered together. In many cases it is simply the spatial distance between two of the entities. But we need a different kind of distance in our case, one that represents the difference between content and that means between two filters. On easy way to do this is the difference that is represented through the overlapping prefix of the filters which means we need a mechanism to compare them, which is prefix matching. If we can compute something like a checksum and compare them it would be excellent, but such an approach does not exist currently. This makes it problematic to formulate the entire problem as a constrained oriented problem, because pattern matching is not the most efficient solution in constrained oriented programming. Nevertheless we can formulate the program as follows. What we want to minimise is the maximum diameter among the clusters we generate. The program looks like the following:

$$\begin{aligned}
\text{Min} \quad & D_{max} && (5.50)\\
\text{Subject to} \quad & D_k \geq d_{i,j} x_{i,k} x_{j,k} \forall i,j,k : i = 1,\ldots,n, j = 1,\ldots,n, k = 1,\ldots,K && (5.51)\\
& \sum_{k=1}^{k} x_{ik} = 1 \forall i : i = 1,\ldots,n && (5.52)\\
& D_{max} \geq D_k \forall k : k = 1,\ldots,K && (5.53)\\
& x_{i,k} \in \{0,1\} && (5.54)\\
& D_k \geq 0 \forall k : k = 1,\ldots,K && (5.55)
\end{aligned}$$

K is the total number of clusters, D_k is the diameter of cluster k and D_{max} is the maximum diameter of all clusters.

For the time being we will be using a simple clustering algorithm which utilises pattern matching and we will do this as a normal, not constrained oriented program. When there is an efficient way to compare the filters, we can easily implement a constrained oriented or even linear program and insert it into the framework.

Advantages

1. Clustering allows us smaller sub graphs where we need to be concerned about similar filters only.
2. Sub graphs allow us faster execution time.

Disadvantages

1. Can have great influence on the quality of the solution.

5.1.7 Summary

At the end we want to present the summary of the advantages and disadvantages of the presented optimisation problems in table 5.1 on page 41.

		Advantages	**Disadvantages**
Shortest Path Tree	Shortest Path	• Fast computation, LP implementation. • Ensured connectivity.	• Heavy tree.

		Advantages	Disadvantages
	Build Shortest Tree	• Groundwork can be done easily. • Dynamical.	• May not be optimal.
	Combination Solution	• Optimisation in one step. • Multiple publishers are supported. • Weights can be used to express load.	• Can consume much time. • Mixed constraint oriented program.
Minimum Spanning Tree		• Known and fast algorithms. • Minimal weight.	• LP formulation is inefficient. • Maximum path length not guaranteed.
Balanced Tree		• Not minimal, but balanced.	• Problematic with multiple publisher. • COP formulation the same as in combination solution.
Steiner Tree		• Well researched algorithms. • Approximation algorithms in linear time.	• Not balanced. • Paths can be very long.

		Advantages	**Disadvantages**
Centre Switches		• No cycles, if done in one step. • Centre switches can be chosen, but... • Partitioning possible. • Minimal amount of edges between centre switches.	• Can be very time consuming. • ... computation of the location is time consuming. • Not balanced path length.
Clustering	Prefix Matching	• Easy to implement.	• No reasonable LP implementation.
	Clustering	• Smaller sub graphs with similar filters. • Sub graphs allow faster execution time.	• Can have great influence on the quality of the solution.

Table 5.1: Summary of advantages and disadvantages of the optimisation solutions

5.2 Transformations

Here we will present a transformation that is necessary to complete the forwarding of the messages in the network.

5.2.1 Filter Embedding

Independent from the solution we choose to compute the paths we still need a solution to embed filters in these paths to reduce the number of overall messages and false positives. Although we did something similar by minimising the edges, we did not prevent false positives in every occasion. Through reasonable applying of filters we can reduce the number of messages that need to be sent. But it is important to maintain the correct delivery of positive messages. It never should be the case that a message does not arrive at its destination although it should have. Thus the filters should be as specific as possible to decrease false positives but sufficient unspecified to forward all messages that need to be delivered. It will not be possible to have no false positives at all but it would be desirable to have no false positives at the subscribers and only a minimum amount of false positives in the network.

The input will be a sub graph of G, G_s, the advertisement of the start point A_s and the subscription of the end point B_s

$$I_{FE} = (G_s, A_s, B_s). \tag{5.56}$$

The output will be G_s with added filters which we will call G_{sf}

$$O_{SF} = G_{sf} \tag{5.57}$$

G_{sf} itself is a sub graph of G_f which is G embedded with filters.

We need a mechanism to check if there are similarities between two filters. This can be done either through pattern matching or some clustering techniques. However, the formulation as a constrained oriented program brings the problem that we need to compare filters, which is a problem, because something like pattern matching is cumbersome. It is reasonable if we do not use constrained oriented programming in this case and instead use normal programming for this path. Nevertheless we can formulate a constrained oriented program but we would need one for each comparison because the length cannot be changed dynamically. This would not be very efficient.

Our basis is the graph

$$G_s(V_s, E_s) \tag{5.58}$$

with the paths

The program would look like the following:

$$\text{Min} \quad \sum_{v_s} Nu(v_s) \tag{5.59}$$

$$\text{Subject to} \quad Nu(v_s) \leq A \quad v_s \in V_s \tag{5.60}$$

$$Dz(x^n_{l,p,s}, s) = dz(s) \tag{5.61}$$

$$Dz(x^i_{l,p,s}, x^{i+1}_{l,p,s}) \supseteq dz(s) \tag{5.62}$$

$$X_{l,p,s} \quad \forall x^i_{l,p,s} \in X_{l,p,s} \in R \tag{5.63}$$

Nu defines the number of flows on a switch in equation (5.60). In equation (5.61) and in equation (5.62) we set the constraints for the dz expressions. The first defines the last expression to the subscriber and the second the other ones between publisher and subscriber.

Although this is a transformation, we can formulate it as a constrained oriented program. But that does not mean, that we can implement it easily. That is not the case. It would get inefficient quite fast as it would need a lot of constraints for the dz expressions and the limit of flows on the switches. This will be computed faster if we use a normal programming language.

Advantages

1. Helps in minimising the number of false positives.

Disadvantage

1. Too many constraints in constrained oriented programming formulation.

5.3 Mechanisms

In this section we present mechanisms that help us solving the optimisation problems. Some of the mechanisms are essential like the cycle detection, some are obligatory.

5.3.1 Virtual Publishers/Subscribers

It could also be a good idea to introduce overlapping constraints for the computation of the shortest paths. This can be advantageous for the choice of shortest path in a scenario where we compute more than one shortest path for each pair of publisher and subscriber. To realise this constraint we can introduce a set of virtual publishers VP and a set of virtual subscribers VS to serve as merge points for paths. These publishers and subscribers will be placed in such a way that two or more previously not joined paths will necessarily have some edges they share. The input is the old shortest paths and the virtual publishers and virtual subscribers

$$I_{Vp,s} = (SP_1, \ldots, SP_n, VP, VS) \tag{5.64}$$

and the output will be a set of new paths P_{new}

$$O_{Vp,s} = P_{new}. \tag{5.65}$$

But they can also be used in a different way for the fragmentation. How exactly they can be used will be explained later.

The new paths, that are the result of the constraint, may not be the shortest for individual publish/subscribe pairs but may be minimal in the overall view. For example in case of figure 5.2 on the next page there are less edges than before.

The idea behind this is to place virtual publishers and subscribers on vertices so that we can force paths to use this vertex. We would compute two shortest paths and can use the linear program that was presented earlier. The virtual subscriber consumes the messages that are sent from the real publisher and the virtual publisher republishes these messages to the original subscriber. The drawback of the solution is the additional time that is needed to compute the location of the virtual publishers and subscribers. This search itself is no easy task to solve and

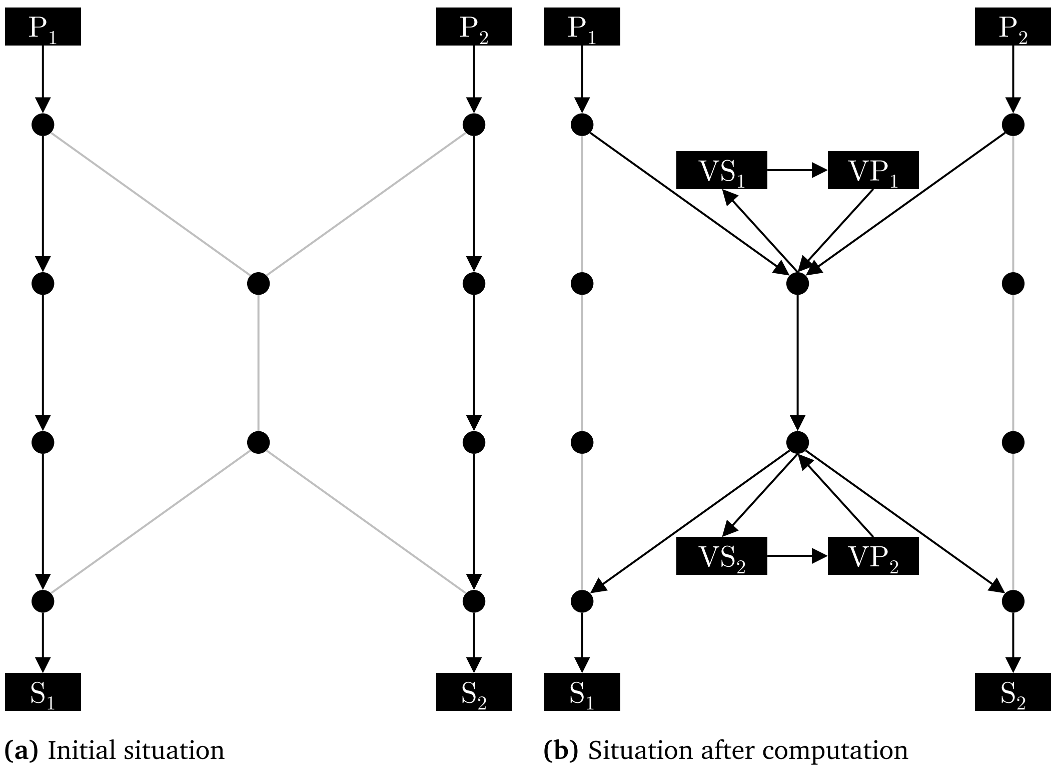


Figure 5.2: Computation with virtual publishers and subscribers

it is uncertain if it is worth the gain. We would need to solve something similar to the facility location problem. There exist several linear programming formulations [6, 35, 49] and we will present one of them. We will take the multi facility location problem out of [49] which is the following.

We have a number of new facilities f and old facilities g and a weight for the edges $w_{i,j}$ which describes the weight between a new facility i and an old one j. We also have a weight $v_{i,k}$ between two new facilities. Additionally we have two distance matrices $d(F_j, G_i)$ for the distances between a new and an old facility and $d(F_j, F_k)$ between two new ones. At last there are the location coordinates for the already existing facilities $G_i : (a_j, b_j)$. The output is $F_i : (a_i, b_i)$ for the coordinates of the new facilities. The standard cost function is as follows

$$\text{Min} \sum_{1 \leq j < k \leq f} v_{j,k} * d(F_j, F_k) + \sum_{j=1}^{f} \sum_{i=1}^{g} w_{j,i} * d(F_j, G_i). \tag{5.66}$$

This solution only ensures that a new facility is connected with any of the old ones. But we need specific connections between the publishers and virtual subscribers and between the virtual publishers and the subscribers. We can achieve this the same way as in the combination solution with a path where we define the start and end point. Of course, we would need the mapping again which adds to the complexity and the time that will be consumed.

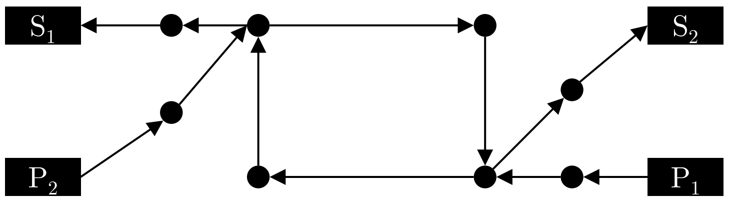


Figure 5.3: Cycle after the computation

Advantages

1. The sharing of paths can be actively influenced.

Disadvantages

1. The locations of the virtual publishers and subscribers need to be calculated.

5.3.2 Cycle Detection

Although, we will try to prevent cycles we cannot prevent them in any case. Therefore we need a mechanism that does cycle detection which analyses the graph and warns us if there are cycles. A depth-first-search plus an additional storage for already visited vertices like in Tarjan's algorithm for strongly connected components is quite sufficient. [60] It is possible to not only detect cycles but also save which vertices are included in the cycle. This is insofar important that we can add some constraints to our constrained oriented program which ensure that this cycle cannot occur in the next computation.

The input of this problem is a sub graph SG which can also be the complete graph

$$I_{CD} = SG \subseteq G \tag{5.67}$$

The output is a set of strongly connected components SCC and these sets should have only one element so that no cycle exists

$$O_{CD} = SCC \tag{5.68}$$

A cycle, like the one in figure 5.3, is quite a problem. The only option in this case would be to take a completely different route if we have more switches connected. If this is not the case we need to solve this cycle with filters only.

Advantages

1. Cycle can be detected.

Disadvantages

1. Cycles need to be resolved with additional computation.

5.3.3 Fragmentation

Sometimes it can be necessary to further divide the clustered sub graphs because their size is too big. The fragmentation of the previously clustered sub graphs can be done via virtual publishers and subscribers. Intermediate points will be chosen and they are fixed for the computation. Those points ensure that the fragments are connected and that the direction of the connection is correct. We take the whole sub graph and divide according to some criteria like the size or we can even try to approximate the best intermediate points in terms of the quality of the solution. The input is therefore the sub graph SG

$$I_F = SG \tag{5.69}$$

and the output is a set disjoint sub graphs SSG of the sub graph

$$O_F = (SSG) \tag{5.70}$$

$$\forall SSG_i \in SSG : SSG_i \cap SSG \setminus SSG_i \tag{5.71}$$

Approximating the intermediate points could be quite hard and we should not waste too much effort for it. The influence on the solution can be balanced by choosing different fragments in the following computation. We can also use more than one intermediate point if it is reasonable, for example if we have more than one connection between two fragments. The intermediate point will then be seen as a publisher/subscriber pair and will be treated as such in the computation. This means we can ensure the connectivity over multiple fragments while we optimise these fragments. An example is given in figure 5.4 on the next page.

It may also be possible to do this without fixed intermediate points but in this case we would need synchronisation through multiple computations. There needs to be some agreement which points will be taken while the computation is in progress. Right now it is not intended to have communication between different optimisations directly. It will also be challenging to implement this communication into a constrained oriented program. It may be impossible at all.

Advantages

1. Allows us to shorten the computation time.
2. Allows parallelisation.

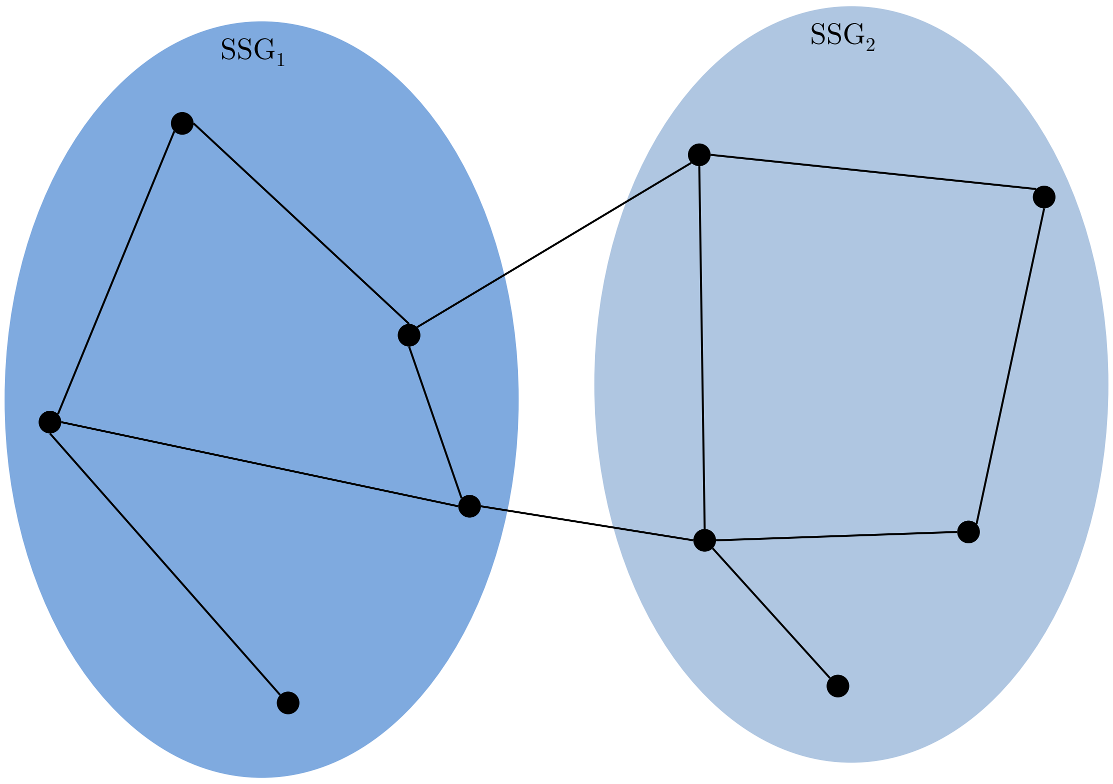


Figure 5.4: Fragmentation of a graph into two sub graphs

Disadvantages

1. Intermediate points need to be found.
2. Can reduce the quality of the solution.

5.3.4 Re-balancing

Weights describe the traffic, delay or other properties of an edge or a physical connection, depending on what we want to optimise. We then try to minimise the overall value of the weights. It is therefore important to represent the weights reasonably. We want them to represent the load on that edge. We already have some way to control the delay to a certain degree, the length of the path. But we do not have any way to control the load. We will use the weights to represent the current load on the edges. This can be done in two ways.

1. Represent the total load on the edge, for example in percent.
2. Represent a suggestion to choose or not choose the path.

The first possibility is more static and is not guaranteed to converge to one value. It is possible that in a future iteration a filter set will not use the edge anymore and we need to subtract the whole weight that this filter set produced. This can lead to multiple extra computations, but it can be prevented if we use a more dynamic approach.

The second one is the dynamic approach. It will converge to one value if there is no change in publishers, subscribers and subscriptions. The weight will only increase slightly to indicate that the load has increased and that other edges may be more attractive. Furthermore we will change the boundaries of the edge each time so that the boundaries converge to one absolute value which represents the final weight. Of course, additionally there is a hard constraint to ensure that the switch and the edge will not get overloaded. We will call this approach the re-balancing of weights.

Advantages

1. Allows us to converge the computation.

Disadvantages

1. Additional computation time.

5.3.5 Summary

At the end we want to present the summary of the advantages and disadvantages of the presented mechanisms in table 5.2 on the following page.

	Advantages	**Disadvantages**
Virtual Pulishers / Subscribers	• Sharing of paths can be actively influenced.	• Locations of VP and VS need to be calculated.
Cycle Detection	• Cycle can be detected.	• Cycles need to be resolved with additional computation.
Fragmentation	• Allows us to shorten the computation time. • Allows parallelisation.	• Intermediate points need to be found. • Can reduce quality of the solution.

	Advantages	**Disadvantages**
Re-balancing	• Allows to converge the computation.	• Additional computation time.

Table 5.2: Summary of advantages and disadvantages of the mechanisms

Chapter 6

Framework

While we identified isolated problems in building a publish/subscribe system in chapter 5 on page 24, we now aim to build a framework that couples selected problems into a holistic solution to minimise the bandwidth usage in a publish/subscribe system. In this framework the isolated solutions for the problems are components and are coupled together. The idea hereby is to use those isolated problems in a sequential order and let the component only solve a single problem. Between the components transformation of the data needs to be done, so that the next isolated problem can be solved. The components do not depend on each other and can be exchanged for another component.

The idea for the framework was inspired by the algorithm presented in [32]. Two trees are generated, a minimum spanning tree and a shortest path tree. While the shortest path tree is very heavy but ensures short individual path length, the minimum spanning tree is light but cannot ensure the individual path length. By using the shortest path for a vertex, when the individual path in the minimum spanning tree is too long, one can ensure the individual path length and also realise a weight of the tree that is optimal under these circumstances. The usage of a basis of two trees to compute another one is appealing for a publish/subscribe system with the constraints we have. In our case we start from the reverse direction, the shortest paths, because the solution in [32] cannot be easily transformed for multiple publishers. Although we will concentrate on a static system with one controller and no parallelisation, our approach can easily be adapted to a distributed environment of controllers with parallel computing abilities.

The framework gives us the flexibility to choose the effort we use for each problem. The effort is mainly the time that is consumed for the problem. With more time, in the worst case indefinitely long, we can achieve an optimal result. The most time consuming part will be the optimisation of the sub graphs, which is also the part which has the most impact on the solution. The time for this part can be influenced in many ways, which is shown later.

The basic idea of the framework is presented in figure 6.1 on the following page, where we can see two phases. The first phase is the *initialisation phase* where we consider the individual paths for each pair and build a shortest path tree out of them for each publisher. This correlates with the problem presented in the previous chapter in section 5.1.1 on page 25, especially the problems in section 5.1.1 on page 25 and in section 5.1.1 on page 26. Into shortest path tree

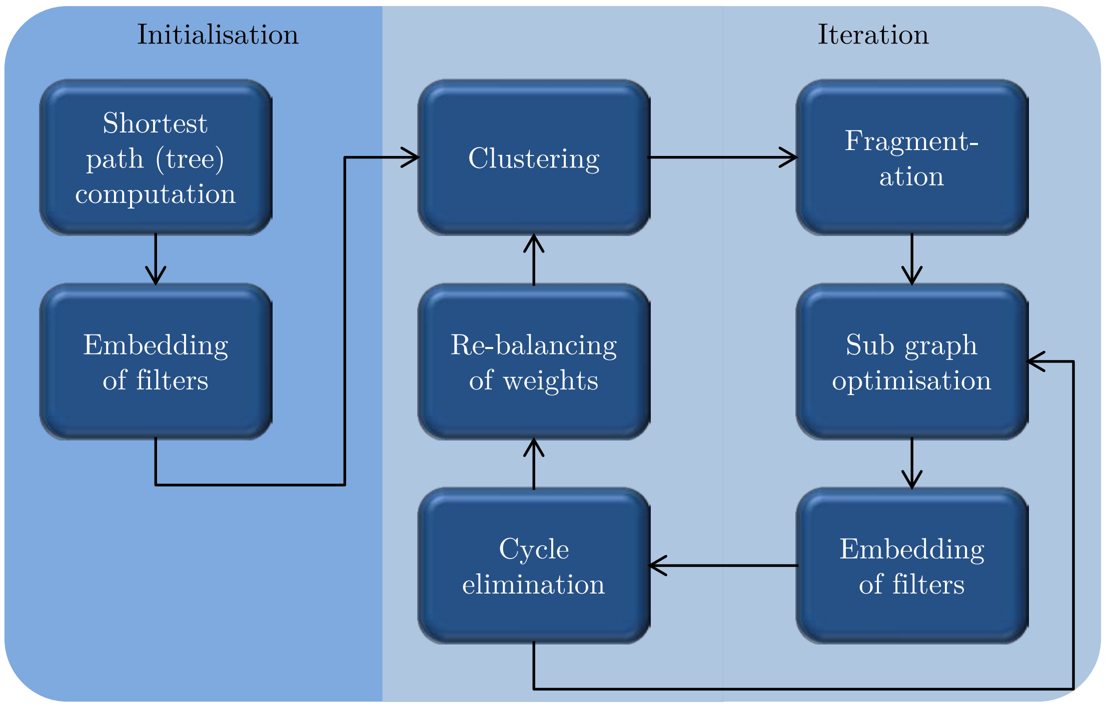


Figure 6.1: The flow diagram of the framework

filters are embedded, where the associated problem was presented in section 5.2.1 on page 41. In the second phase, the *iteration phase*, we consider the whole graph with all its participants and filters. The first phase serves as preparation for the second phase to get an overview over the connections of the participants. In the *iteration phase* we have multiple components which are all important for the optimisation. First, the clustering decides which filters we can group together, that means, which filters are similar enough to be grouped together. This corresponds with the problem presented in section 5.1.6 on page 36. Following clustering is one of the components which influence the solution. The fragmentation can be used to partition the graph further so that we do not need too much time. It can also be ignored when we want to have the optimal solution for the cluster, but then the computation time will be higher. The problem of fragmentation was presented in section 5.3.3 on page 46. Next is the sub graph optimisation where the problem was presented in section 5.1.1 on page 28. Here we compute the optimal solution for the graph this component gets. If the graph is not fragmented then it will be the optimal solution for this cluster. But if the cluster is fragmented, then it will be only the optimal solution for this fragment. The size of the graph, as well as the number of paths that need to be considered, influence the computation greatly. Clustering and fragmentation are a way to influence both parameters and determine the time the computation needs. After the computation we have again the embedding of filters and then the cycle detection. The sub graph optimisation can produce cycles and need therefore be checked as show in section 5.3.2 on page 45. The sub graph computation will be repeated to ensure an optimal solution for the graph. Constraints to eliminate the cycle are set in this computation. The only thing left is to rebalance the weights so that we have a representation of the actual situation. The problem of the weights and connected with it, the cost function, was presented in section 5.3.4 on page 47. The overall cycle in the *iteration phase* is not important right now. This one is used when we have a dynamic and not a static system. But for the sake of completeness we added it to the figure.

These components will be embedded in an encapsulation program. This program will take care of the administrative tasks and organise the components. Some of the components will be implemented as a linear program, some as constrained oriented programs and some will be implemented as normal algorithms. But each component will be exchangeable with another solution for the problem it solves. In the following we will present each component that we have chosen in detail.

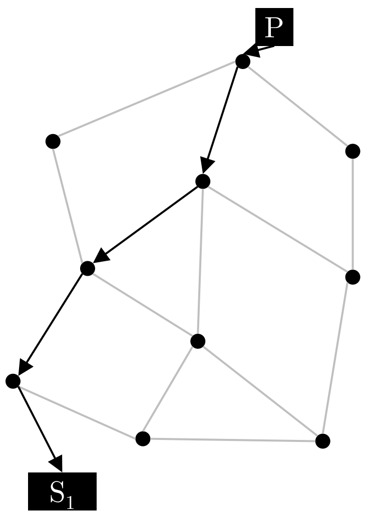
6.1 Shortest path tree (SPT) computation

The SPTs serve as a starting point for further computations thus the shortest paths will be computed via a linear program as shown in section 5.1.1 on page 25. For this program we need the graph with its weights and the end points of each path, in this case the edges between publisher and switch and subscriber and switch. For each publisher/subscriber pair, a shortest path will be computed through this program. These paths are optimal with respect to weight for each individual pair but not optimal in a global view. These paths are saved and for each publisher a tree will be constructed from all the multiple shortest paths, where the publisher is one of the endpoints. The tree will be constructed by adding all the paths into one graph.

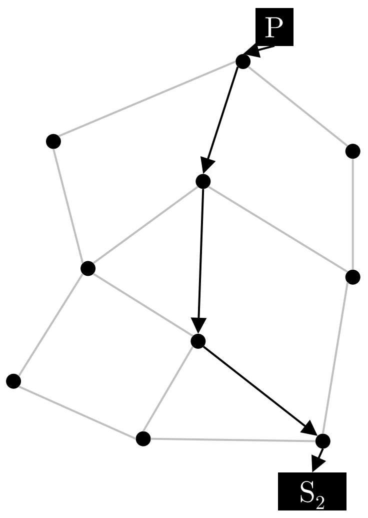
These trees represent the flow of the messages from the publisher to its subscribers, thus these trees are directed. If we embed all these trees in our given network topology of physical links we get a directed graph where one edge can be used multiple times. With the embedding of the trees into the network topology, we need to add the filters to distinguish the multiple paths on the edges. The complete process of computing the shortest paths, shortest path trees and filter embedding is presented in figure 6.2 on the next page.

There are two functions for the shortest path computations. First is the approximate overview which edges and vertices are used for a publisher and its subscribers. The second, and most important, function is to ensure that the publisher is connected to its subscribers and the subscribers can receive the messages.

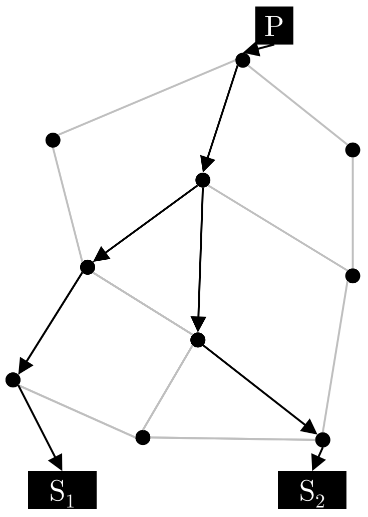
The performance is dependent on the number of vertices and edges we have included but these parameters do not matter much. When we have a lot of publishers and subscribers and hence a lot of paths, the overall computation time can cause a small delay. But in later computations the time for this algorithm is negligible.



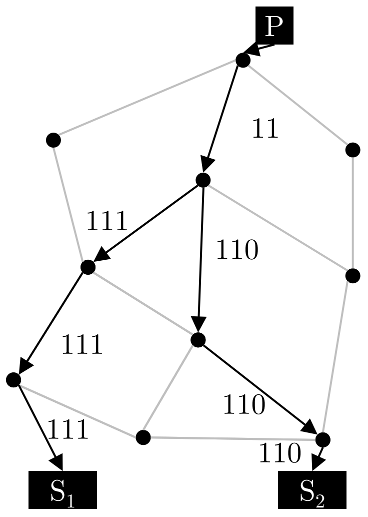
(a) Path from P to S_1



(b) Path from P to S_2

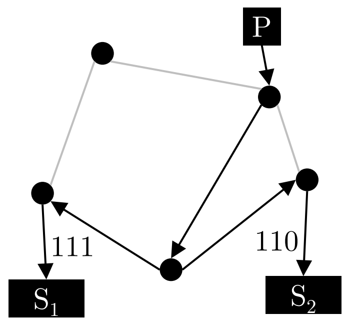


(c) The shortest path tree

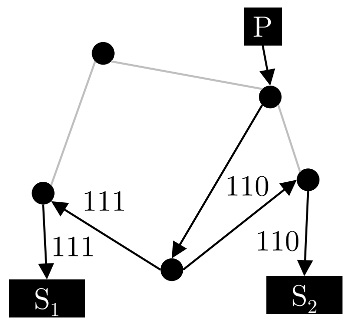


(d) Adding filters to the tree

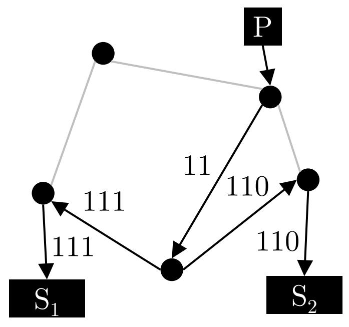
Figure 6.2: Computation shortest path trees



(a) Filters for Subscribers



(b) First iteration



(c) Second iteration

Figure 6.3: Embedding of filters through iteration

6.2 Filter embedding

The embedding of the filters is done in the surrounding program. If there exists a more efficient way to do pattern matching in linear or constrained oriented programs than the one presented in section 5.1.6 on page 36, one can think about transforming the problem into one of those. As it is now, the easiest way to do the embedding is in the surrounding program where we can do pattern matching in an efficient way. It would be desirable to formulate this as a linear or constrained oriented program because it is normally more efficient, the solvers are highly optimised. It can also be seen as an optimisation problem and it is therefore preferable to solve it as a linear or constrained oriented program. But we need pattern matching to determine the difference between two filters and get the common prefix, so we cannot avoid it.

The input for the program is either a shortest path tree or an optimised sub graph. Both can be seen as graphs, more accurate, as sub graphs of our original graph. The computation is in both cases the same. Furthermore we have the position of the publisher and subscribers and the subscriptions of the subscribers and the publications of the publishers.

The setting of the filters should be in such a way that the number of messages will be further reduced. It is a valid scenario to assume that not all subscribers will have the same subscriptions. This means that not all the subscribers want all the messages from a publisher. We need to ensure that no unnecessary messages will be delivered and this can be done if we specialise the filters accordingly. This specialisation should be done as soon as possible to reduce the messages in the network. If a message is not needed at a vertex, that means no filter exists for this message, then it is not necessary to forward this message to this particular vertex. The filter on the predecessor can be more specialised to cut this message of or only forward it to vertices that are interested in this message and have an appropriate filter.

A first idea to realise this, if we assume we have some kind of pattern matching in linear or constrained oriented programming, is to formulate the constraints in such a way, that the filters at the subscribers are fixed and all the other ones can be changed. The constraint that needs to be fulfilled is that the predecessor of the filter needs to deliver all messages that the successor will deliver. The decision will be made according to pattern matching or similar mechanisms because the filters are coded as a bit string. This bit string defines which dimensions a subscription or filter has. If we take a filter that is shorter than another one, but has exactly the same prefix as the other filter, than the shorter filter delivers more messages than the other one because it has less dimensions than the other filter which means it is less specialised and filters out lesser messages. A filter needs to be short enough to let enough messages through but specialised enough to filter the unnecessary messages out. The optimisation goal would then be to reduce the number of filters on each switch, which means to reduce the length of the filters. We have reached the optimal solution when we reach the minimal amount of filters on the switches under consideration of the constraints.

We do not have a linear or constrained oriented program so the previous solution does not work. We will compute the solution with a normal program but similar to the presented solution. We will start the same way as before, we will set the filters to the subscribers. But then we will use the directions of the directed graph and follow them in the inverse direction. With this we make sure that we filter in the right direction, the one that is more specialised, the direction from publisher to subscriber. Now we set the filters in such a way, that the filters forward the same amount of messages as its successor or more, in some cases less when multiple flows join together on the successor. This will be done in an iterative way until all the publisher and subscriber pairs are connected with filters so that the subscribers get their messages. We need to wait at forks until every path after the fork has a filter, until then we cannot continue.

With this approach, we only filter the absolute necessary messages and not more than that. Every filter is optimal in regard to its successor and predecessor.

6.3 Clustering of filters

What we now have is a graph where publishers and subscribers are connected and the filters are embedded so that every subscriber gets its messages. We mentioned before that this solution is only optimal regarding the weight for each path individually. Clustering is serving as a preparation for the sub graph optimisation to cluster similar filters together so that the sub graph optimisation does not need to consider filters in its optimisation.

The clustering of sub graphs can be done in a great variety of possibilities. There exist a lot of clustering techniques, some more suitable, some less, but none that could be used out of the box. The time and effort spend for the clustering affects its quality which in turn affects the quality of the overall solution. This is due to the coupling of the clustering and the sub graph optimisation. The sub graph optimisation does not consider filters because the comparison of filters would again create problems for the constrained oriented program. Therefore we need to prepare the sub graphs in a suitable way with clustering. For now it is sufficient if

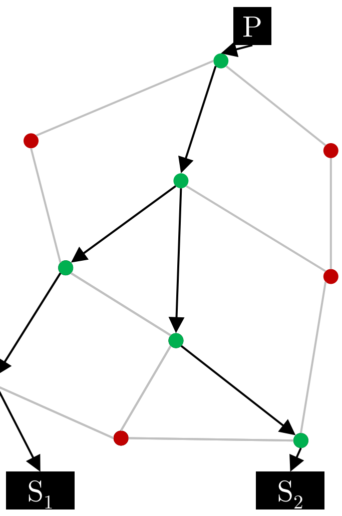


Figure 6.4: Selection of vertices for the sub graph

we use a simple algorithm that gives us appropriate results. This will not lead to an optimal solution in the results but a clustering algorithm for this scenario would be a thesis in itself. There exist algorithms like the one we mentioned in the previous chapter and is presented in [58] that are related to our scenario. But adapting this algorithm for our scenario would still take some time. Adapting this algorithm could be one possibility for future work. We will focus on the framework and thus take a simple algorithm, like going through the graph with a depth-first-search and pattern matching, to get some clusters.

When we have a cluster composed of vertices and edges we need to add other vertices and edges to raise the possibility of finding the best path between two participants. We will do this in an easy and highly scalable way. We will take the neighbours of the vertices already included and its edges into the cluster. In case of the example presented in figure 6.4 the green ones are the vertices that are included because one path uses them and the red ones are the neighbours of these. This can be done again with the neighbours of the neighbours and so on. Each step adds some vertices and edges to the sub graph and makes it bigger. If we take all the edges and vertices we can be sure to get the overall optimal solution because no possible edge is left out. To manage the time consumption of the sub graph optimisation the size is an important factor which can be influenced through clustering with the presented method. Another option, if we want to save time, is the partitioning of sub graphs inside a cluster if these sub graphs are not connected.

6.4 Optimisation of sub graphs

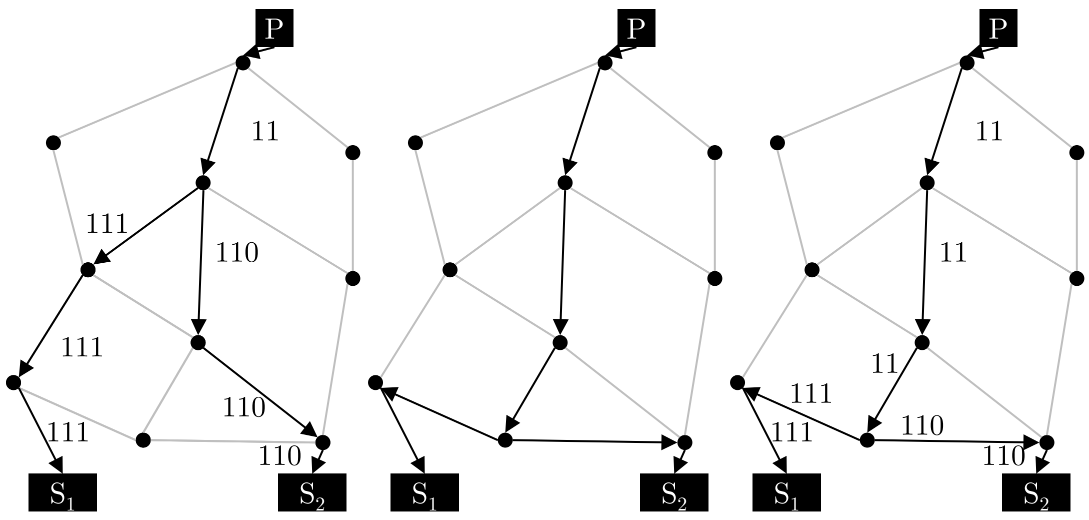
After we clustered the graph into sub graphs according to the similarity of the filters we want to optimise each of the sub graphs without regard to the respective filters. The filters are added again after the optimisation.

The optimisation of the sub graphs is a major key point in our optimisation strategy. It influences the quality of the solution in a great deal. The quality of the optimisation is itself influenced mainly by the size of the sub graphs and the quality of the clustering. Although we want to have a solution that can handle clusters as big as possible we will have to rely on clustering and fragmentation to reduce the size for reasonable usage where the solution can be sub optimal. The constraints for the program are rising very fast for additional paths and vertices. This is due to the guarantee of connectivity between the participants that we need to ensure.

The goal is to minimise the amount of edges we use and to minimise the overall weight of the sub graph. The idea is that if two or more subscribers have overlapping subscriptions, it would be cheaper to have a long common path and a fork should only appear at the end of the shared paths to that the weight of edges is minimal. We therefore share some paths, even if it means additional messages on this path. But these additional messages have a very similar content and it is likely that the subscribers which are interested in the content of one of the publishers will also be interested in the content of the other publishers for which this path is shared.

Another point is the length of the paths. No path should be too long so that the delay on each path is acceptable. Here we will use a hard constraint that limits the length of the path. The value of this constraint can be determined in the surrounding program according to the sub graph. This can vary from sub graph to sub graph dependent on the size of the sub graph. In the following, the course of action will be presented where the problem was presented in section 5.1.1 on page 28 and a result of such an approach is given in figure 6.5 on the next page.

First, we need to make sure that the publishers and subscribers are connected. We will define a path over vertices with the publisher in the first position and the subscriber in the last position that makes sure that this happens. The cost between disconnected vertices is theoretically infinitely high, actually, for the implementation, it is simply a very high value that is far out of the range of the normal weights, so that by minimising the cost we only have connected vertices in the path. We now have multiple paths, each representing one publisher and one subscriber. The edges these paths use will be mapped onto a matrix so that we know which edges are used altogether. The cost of these edges is represented in another matrix. Now we minimise the cost accordingly and get an optimal minimal graph for the given problem. We only need to add the filters so that we can use this graph. Here we use the filter embedding again.



(a) Shortest spanning tree with filters **(b)** Optimised sub graph without filters **(c)** Optimised sub graph with filters

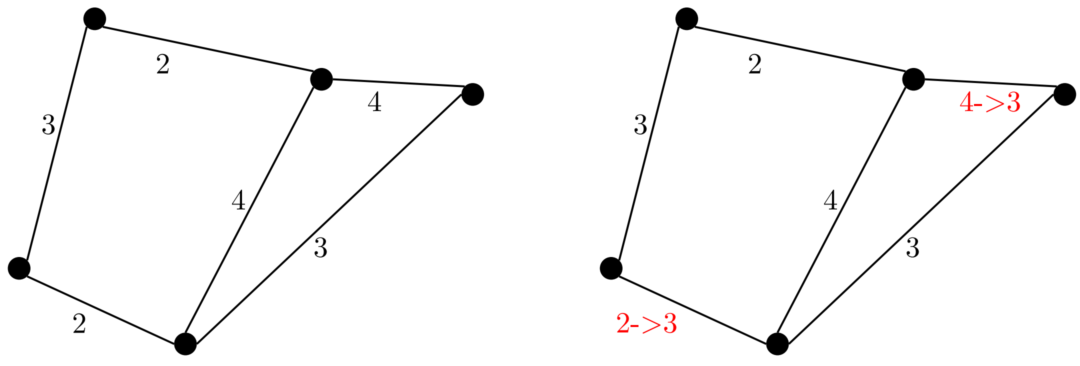
Figure 6.5: Optimisation of the previous SPT

The optimisation itself can be very time consuming if the sub graph has a certain size and a certain number of paths. We cannot avoid that at some point the time for computation would be too high for reasonable usage. This is especially the case for dynamical systems. In a dynamical system we only have a certain amount of time and until then we need to have a solution. Although the focus lies with a static system we nonetheless will introduce the fragmentation later on so that we can compensate the time consumption through smaller sub graphs.

6.5 Re-balancing of the weights

After we have calculated the sub graphs and embedded filters into it we still do not have a representation of the actual load in the overall graph. To get the representation we need to reconfigure the weights on the edges. But we will not be contend with only a representation, we will use this step for our optimisation as well by changing and converting the boundaries of the weight on the edges as well.

Not only the conjunctive filters influence the weight of the edges but also the disjoint filters influence it. As we stated before we do not want to overload any vertex and any edge. To achieve this, we must consider the weights from all filters and modify the weight so that it represents the actual load on the edge. But we will not simply add the weight to the edge because we use the weight in all our optimising processes. We want to converge the weight of the edge to an optimal weight that represents not only the usage but also a hint if another



(a) Weights before the re-balancing **(b)** Weights after the re-balancing

Figure 6.6: Change of weights through re-balancing

flow should use this edge or not. Every single weight is in dependence of all the other weights, if one weight changes, some others can change too. For the process of converging on the weights we have an upper and lower bound on each weight. The upper bound signals if there is a transgression of the load on the edge and the lower one gives the minimum amount of load that is actually used on this edge. With all the steps of sub graph optimisation we try to converge on those two bounds. In the ideal case we have one absolute value for the weight of each edge that represents the ideal weight so that all paths are optimal according to these weights, the length and their individual weight. Of course, this is only the case if we can assume a static graph, which we can for this assignment.

6.6 Fragmentation

Network topologies can get quite big and the optimisation would need too much time to finish if we compute a cluster one step. Therefore, we need the possibility to fragment the cluster into multiple parts so that we can use the sub graph optimisation on smaller fragments. We will use a similar approach to the one mentioned in [8]. We will use the previously defined virtual publishers and subscribers in the sub graphs to simulate the flow in this sub graph. The routes will then be computed as if these virtual publishers and subscribers are real ones. This is not a problem at all if we look at it in an abstract way. Every vertex that forwards messages can be seen as a publisher from the point of view of its successors and every vertex in return can be seen as a subscriber from the point of view of its predecessor. Vertices only know from where they got a message and to where they need to send this message. If the edge that connects the sub graphs is fixed then the flow of the message does not change and every real subscriber gets the messages it has subscribed to. Although it is some additional overhead to get the edge that is fixed, we need to search for edges that start in a fragment and end in another one, we will save time for execution of the sub graph optimisation at the expense of quality of the solution.

6.7 Summary

With the proposed framework and its components we are able to solve the entire problem of route optimisation that we presented. Every component can solve a part of the problem and with the transformations and mechanisms, which we presented; the components can interact with each other. The components are not that complex anymore and can be exchanged for other components that solve the same problem with a different approach. Through the defined input and output this exchange does not pose any problem. Although, some additional problems, like the need for cycle detection, arise, the trade-off between reducing complexity and additional overhead is acceptable. The only thing that remains is the evaluation of the framework itself.

Chapter 7

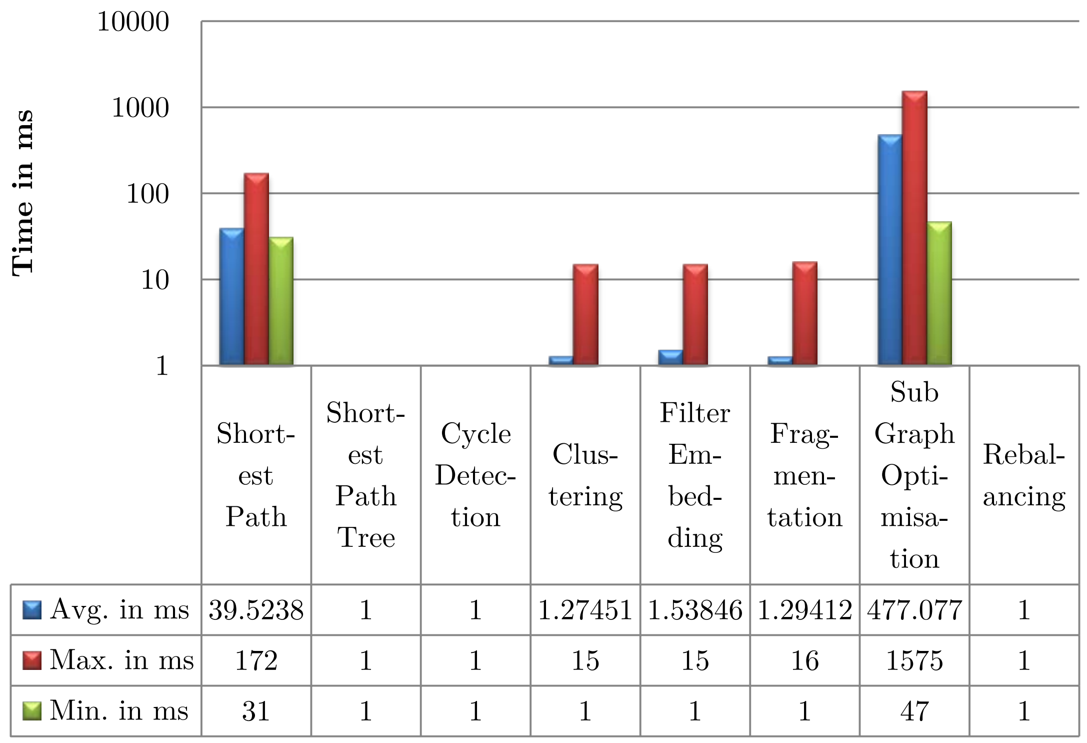
Analysis And Results

After we presented the framework we now want to evaluate the performance and the quality of the solutions it can deliver. We will take a look at the development of the number of flows and bandwidth usage during the optimisation as well as the time we need to compute the optimisation. We will have different settings for different sizes to evaluate the performance and the effect of the fragmentation on the solution. The main focus lies on the two optimisation processes, the shortest path optimisation and the sub graph optimisation. But we will also evaluate the other components even if they do not have an optimisation implementation. It is still interesting to see which of these components needs the most time for their computation.

7.1 Setting

The evaluation was done on a virtual machine with four cores, but the solver for the linear and constraint oriented programs did not use more than one core so there was no parallelisation in the optimisation components. This calls for manual parallelisation of the whole optimisation process. We did run the tests with different settings. The implementation is Java based and for the optimisation processes AIMMS with its SDK was used. AIMMS has the advantage to formulate the optimisation programs in an abstract way so that we are not limited to one solver only.

For the tests mainly we changed the number of publishers and the number of switches. The number of subscribers is coupled with the publishers. A static number multiplied with the number of publishers equals the number of subscribers. This was done so that we have comparable settings without too much external influence. The graphs were created randomised with the number of switches it should have and a maximum number of edges between the switches. The filters were also created randomly but with a minimum length. The randomisation was done to get different settings and to show that the optimisation can deal with all graphs. The settings were with 10 switches with 5, 10 and 25 publishers and with 25 switches with 10, 25 and 50 publishers.



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	39.5238	1	1	1.27451	1.53846	1.29412	477.077	1
🟥 Max. in ms	172	1	1	15	15	16	1575	1
🟩 Min. in ms	31	1	1	1	1	1	47	1

Figure 7.1: Time consumption for 10 switches and 5 publishers

We chose different sizes for the topology to demonstrate the importance of the size of the network for the optimisation process, in time as well as in quality. The number of publishers was adapted to the size of the topology accordingly.

The results presented for each setting will be divided into a graph for the time consumption, a graph for the time distribution and two graphs for flow reduction and data rate reduction.

7.2 Results

Now we will take a look at the results of the tests to get a better understanding of the performance of the framework. We divided the tests according to the number of switches.

7.2.1 10 Switches

First of all we consider the situation where we have 10 switches and 5 publishers. We did not use fragmentation in this case because the sub graphs were not that big. But we used clustering to distinguish sub graphs with unsimilar filters. Figure 7.1 shows us the average, maximum and minimum time consumption of each component. Unsurprisingly, the sub graph

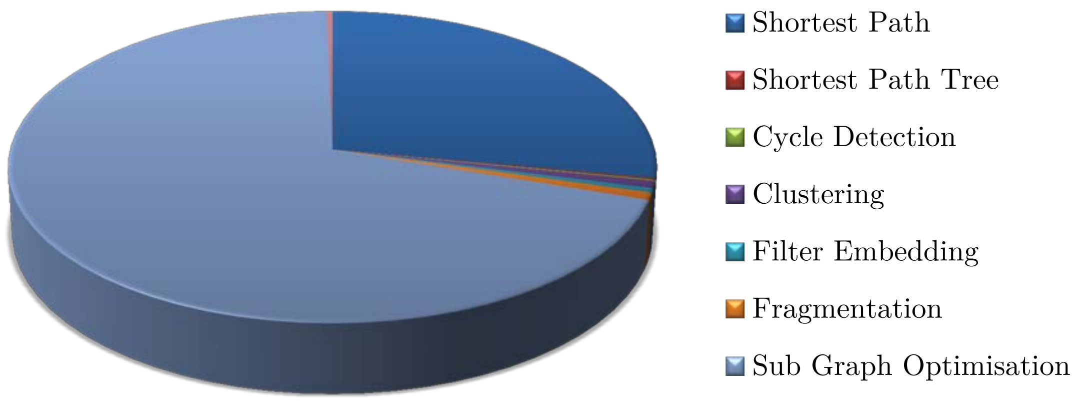


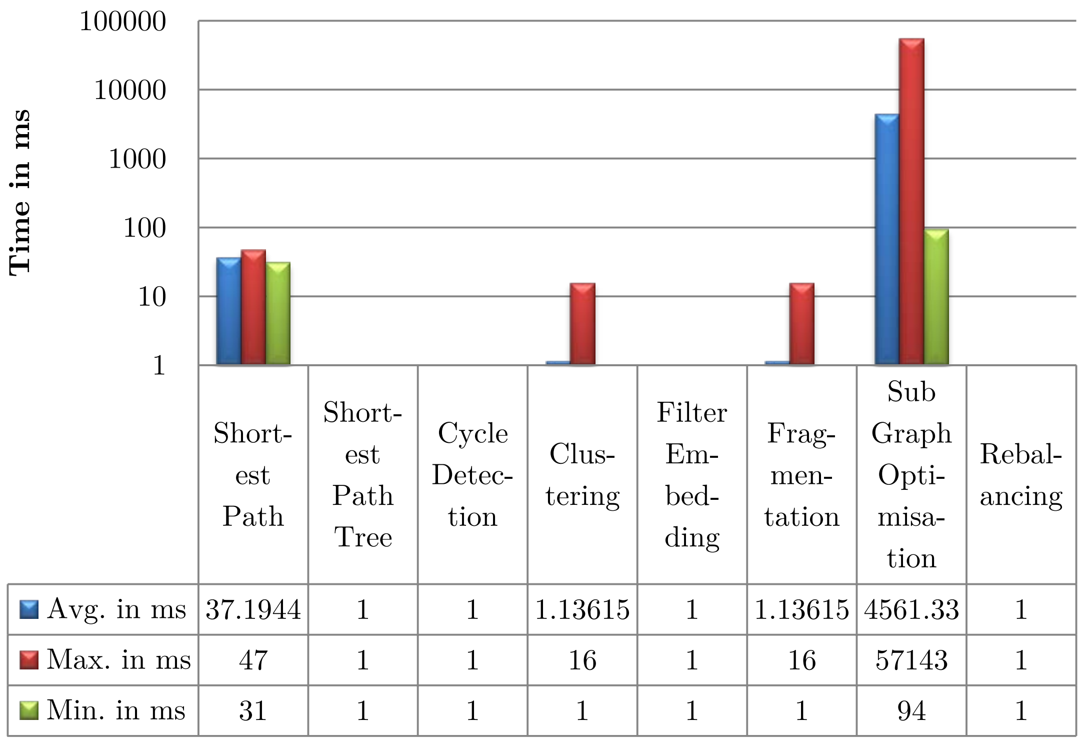
Figure 7.2: Time distribution for 10 switches and 5 publishers

optimisation has the highest average and maximum value. Most of the optimisation relies on this component. While the shortest path optimisation needs the second most time, time for rest of the components is negligible.

In figure 7.2 we see the time distribution for the components. Nearly two third of the computation time is used by the sub graph optimisation whereas the shortest path optimisation needs nearly one third. The time usage of the shortest path computation results from the number of paths we need to initialise. The quantity of sub graph optimisation is in comparison much lower.

If we add more publishers and therefore more subscribers to the topology we logically have an increase of paths. This increase means that we have to compute more shortest paths in the beginning but the impact of these computations on the overall time is more and more negligible. More paths also means more sub graphs or bigger sub graphs which increases the time spent for sub graph optimisation. This can be seen in figure 7.3 on the following page as well as in figure 7.5 on page 65 where the average as well as the maximum time for the sub graph optimisations rises.

In figure 7.4 on the following page and figure 7.6 on page 65 we can see that the sub graph optimisation takes more and more of the overall time for itself. All other parts become negligible. This also means that we can easily spend more time for other components to have better preparation for the sub graph optimisation. We can easily get the time back that we lose



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	37.1944	1	1	1.13615	1	1.13615	4561.33	1
🟥 Max. in ms	47	1	1	16	1	16	57143	1
🟩 Min. in ms	31	1	1	1	1	1	94	1

Figure 7.3: Time consumption for 10 switches and 10 publishers

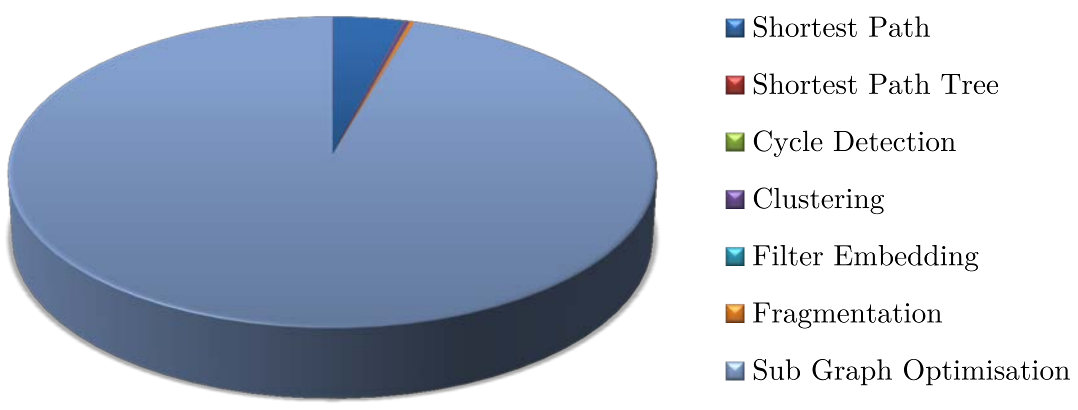
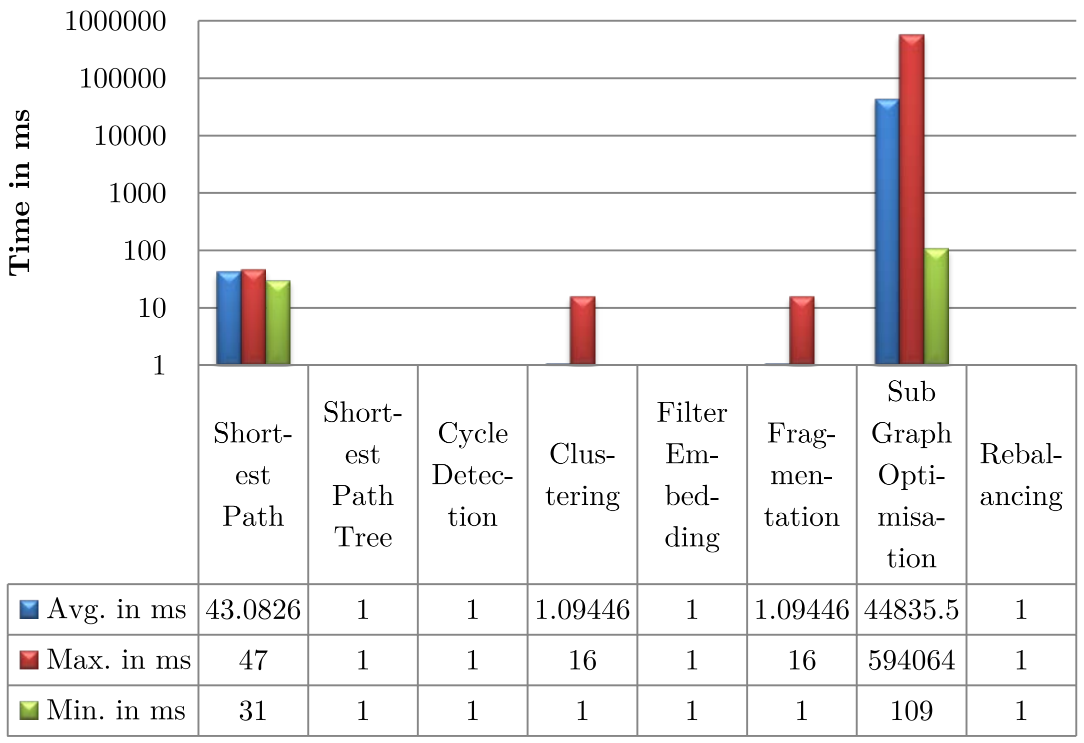


Figure 7.4: Time distribution for 10 switches and 10 publishers



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	43.0826	1	1	1.09446	1	1.09446	44835.5	1
🟥 Max. in ms	47	1	1	16	1	16	594064	1
🟩 Min. in ms	31	1	1	1	1	1	109	1

Figure 7.5: Time consumption for 10 switches and 20 publishers

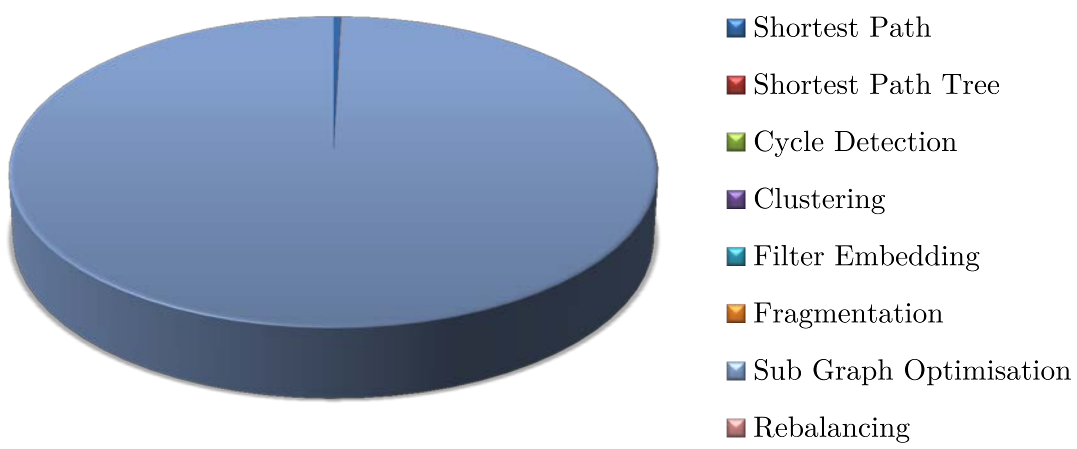


Figure 7.6: Time distribution for 10 switches and 20 publishers

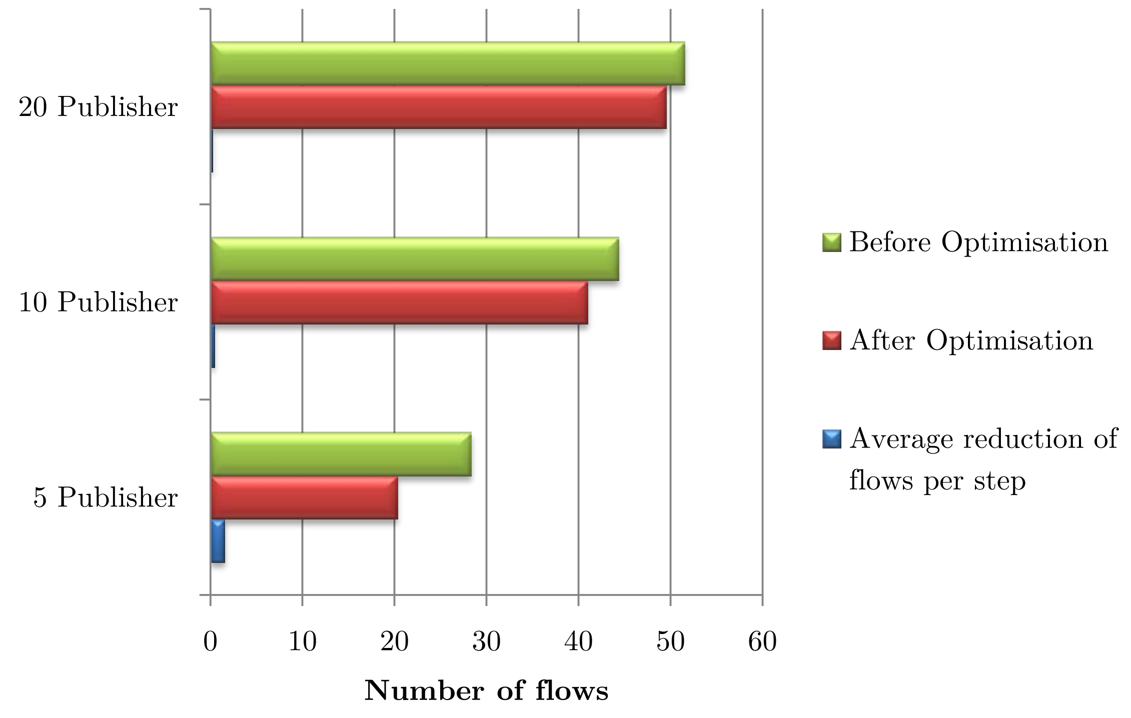


Figure 7.7: Flow reduction for 10 switches

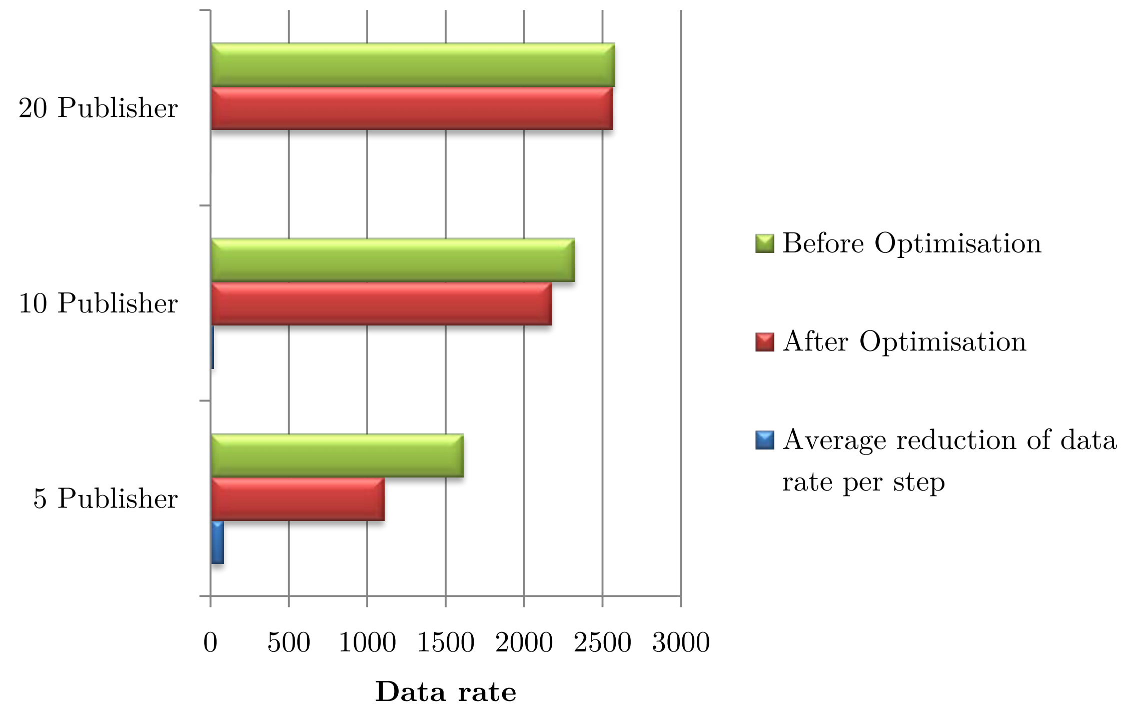
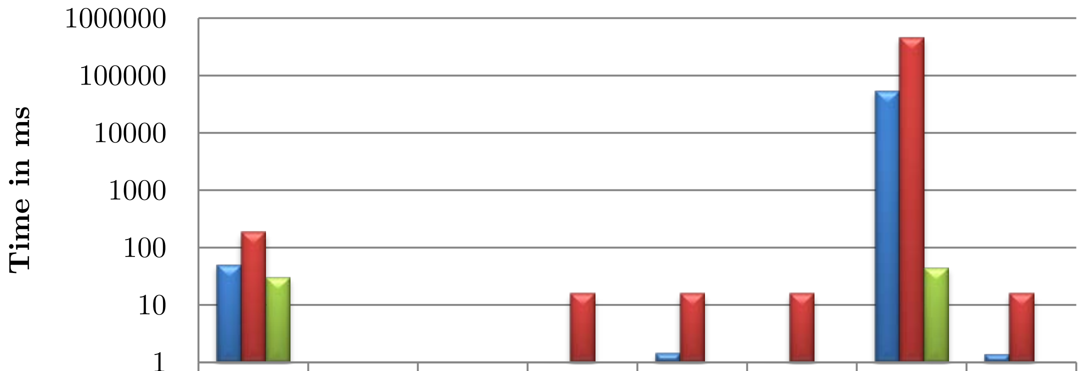


Figure 7.8: Data rate reduction for 10 switches



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	51.7944	1	1	1.03995	1.45455	1.03893	55807.1	1.42424
🟥 Max. in ms	188	1	1	16	16	16	476439	16
🟩 Min. in ms	31	1	1	1	1	1	46	1

Figure 7.9: Time consumption for 25 switches and 10 publishers

in the preparation. But we also see that the sub graph optimisation cannot handle sub graphs that are too big. Although, it seems that the average time consumption is pushed high through one or two sub graph optimisations that need very much time. To approach the problem of big graphs, we have introduced fragmentation. Results for this are presented later.

Now let us take a look at the development of the amount of filters and the data rate when we use the sub graph optimisation. In figure 7.7 on the preceding page we can see the reduction of flows for three different numbers of publishers. The number of flows reduces with a reduction in number of publishers. If we have more publishers, we have lower variety for the deployment of flows, which results in lesser reduction of the flows.

In figure 7.8 on the previous page we can make the same observation for the overall data rate in the network topology. The more publishers we have, the less we can achieve through optimisation because we do not have enough variation in possibilities, which in turn results in lower data rate reduction. Although flow reduction does not necessarily mean reduction of the overall data rate, we can still achieve data rate reduction with flow reduction.

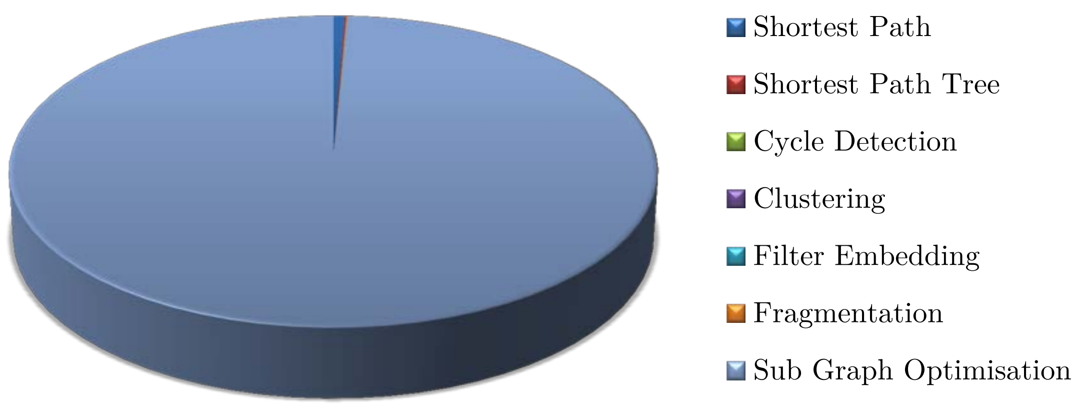


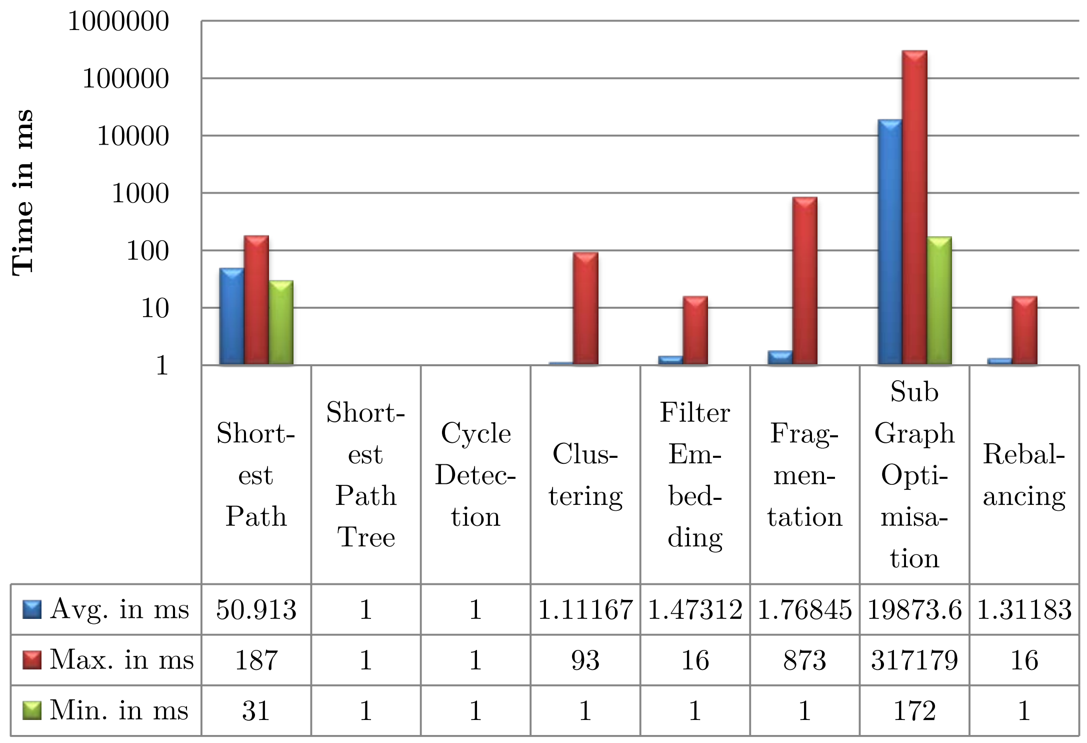
Figure 7.10: Time distribution for 25 switches and 10 publishers

7.2.2 25 Switches

When we look at the time consumption and time distribution between different components in the setting with 25 switches, we can see a similar picture. The overall time consumption rises as the time consumption for the sub graph optimisation rises as we see from figure 7.9 on the preceding page to figure 7.11 on the next page and figure 7.13 on page 70. The time distribution stays the same as in the previous setting with the sub graph optimisation as the main consumer as we see in figure 7.10, figure 7.12 on the following page and in figure 7.14 on page 70. In general, this means that we should focus on the sub graph optimisation for performance optimisation or on preparation for the sub graph optimisation.

In contrast to the setting with 10 switches, in the setting with 25 switches, we can better optimise the number of flows and data rate. When we have more switches we have more possibilities to choose from and can find better overall solutions. This is reflected in figure 7.15 on page 71 and figure 7.16 on page 72 where we can see the difference in the situation at the start and at the end of the optimisation. But we need more time because we have more computation steps which is a slight drawback.

It is a bit surprising that the shortest path solution is not that bad with respect to the number of flows. Although, it is quite bad if we look at the overall data rate usage for the shortest path



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	50.913	1	1	1.11167	1.47312	1.76845	19873.6	1.31183
🟥 Max. in ms	187	1	1	93	16	873	317179	16
🟩 Min. in ms	31	1	1	1	1	1	172	1

Figure 7.11: Time consumption for 25 switches and 25 publishers

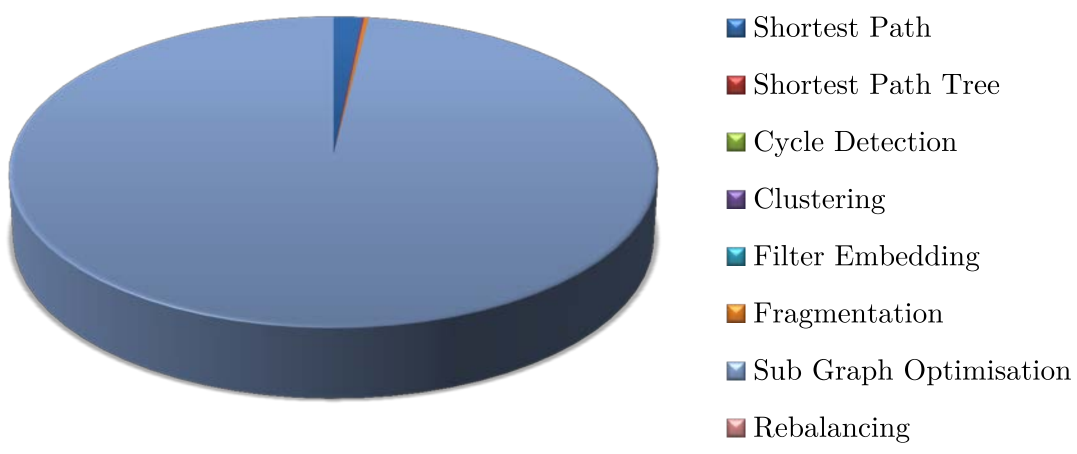
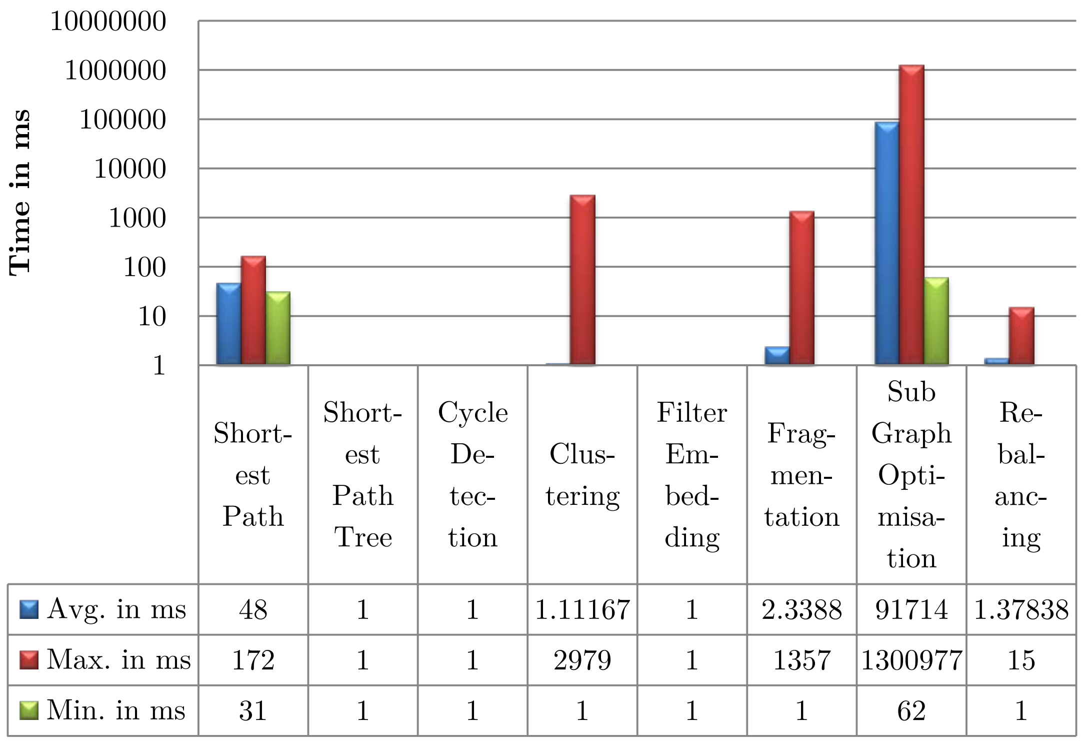


Figure 7.12: Time distribution for 25 switches and 25 publishers



	Short-est Path	Short-est Path Tree	Cycle De-tec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Re-bal-anc-ing
🟦 Avg. in ms	48	1	1	1.11167	1	2.3388	91714	1.37838
🟥 Max. in ms	172	1	1	2979	1	1357	1300977	15
🟩 Min. in ms	31	1	1	1	1	1	62	1

Figure 7.13: Time consumption for 25 switches and 50 publishers

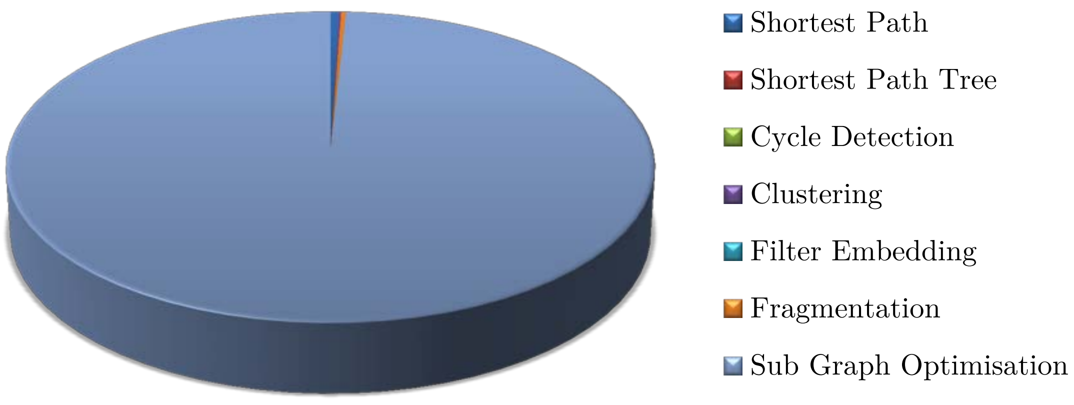


Figure 7.14: Time distribution for 25 switches and 50 publishers

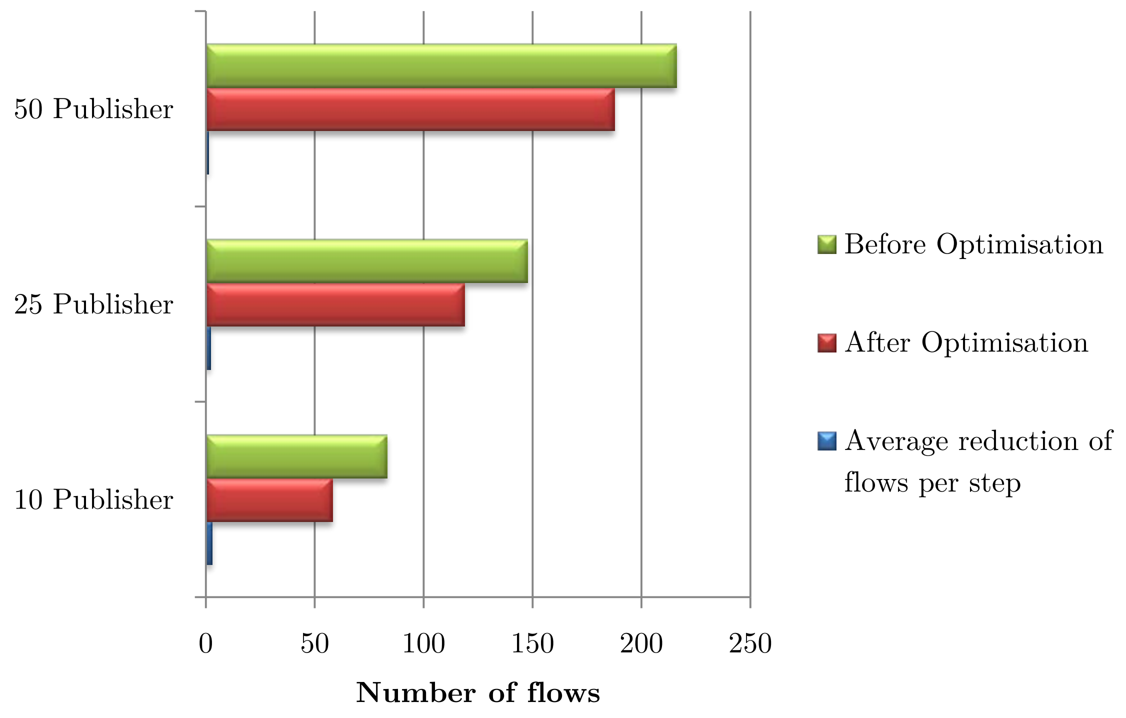


Figure 7.15: Flow reduction for 25 switches

solution. Nonetheless, this can be helpful for further consideration to change the focus of the optimisation process to a different component.

7.2.3 10 Switches with smaller fragments

To evaluate the fragmentation we performed the test with the same settings for 10 switches but with reduced fragment size. We would expect lower time consumption and a bit of loss in quality which means, lower flow and data rate reduction.

While the time consumption has indeed reduced, as can be seen in figure 7.17 on the following page, figure 7.19 on page 73 and in figure 7.21 on page 74, the time distribution does not change much, as can be seen in figure 7.18 on page 73, figure 7.20 on page 74 and in figure 7.22 on page 75. The sub graph optimisation still needs most of the time and the other components need the same amount of time as before. It is not surprising that the sub graph optimisation needs most of the time, but it is good to see, that we can reduce the average and maximum time consumption of the sub graph optimisation significantly. With fragmentation and the time reduction that is the consequence we can take on bigger graphs with this framework.

The quality of the solution does not suffer much, as can be seen in figure 7.23 on page 75 and in figure 7.24 on page 76. This may seem a bit surprising at first, but it is a very positive result,

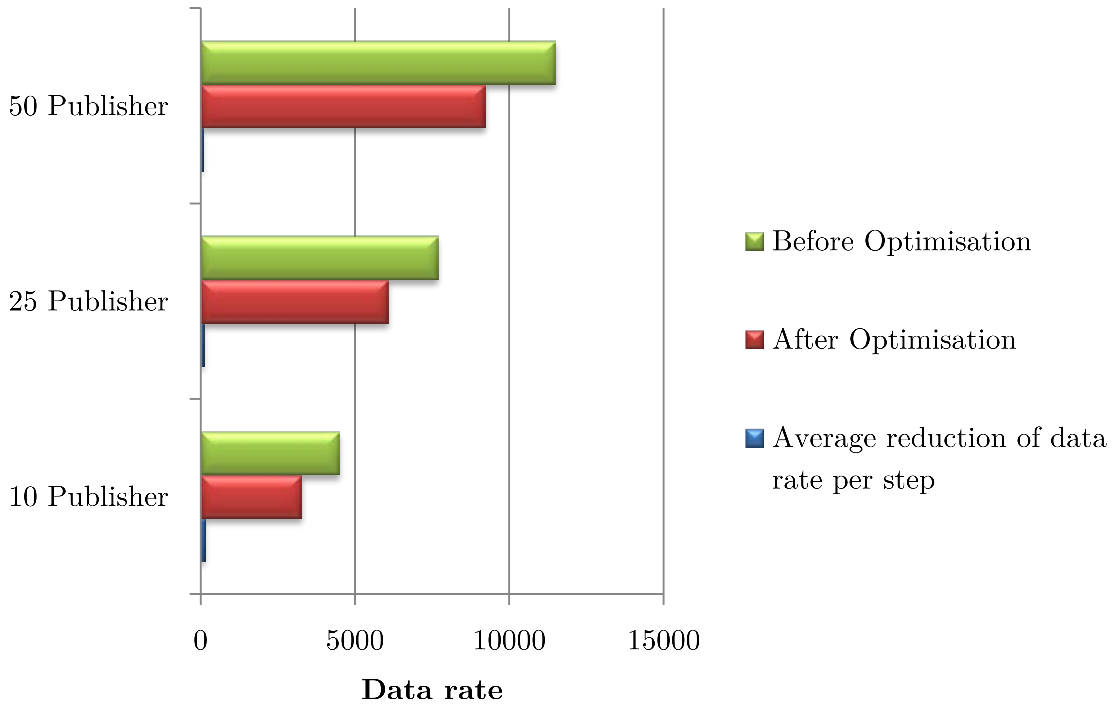
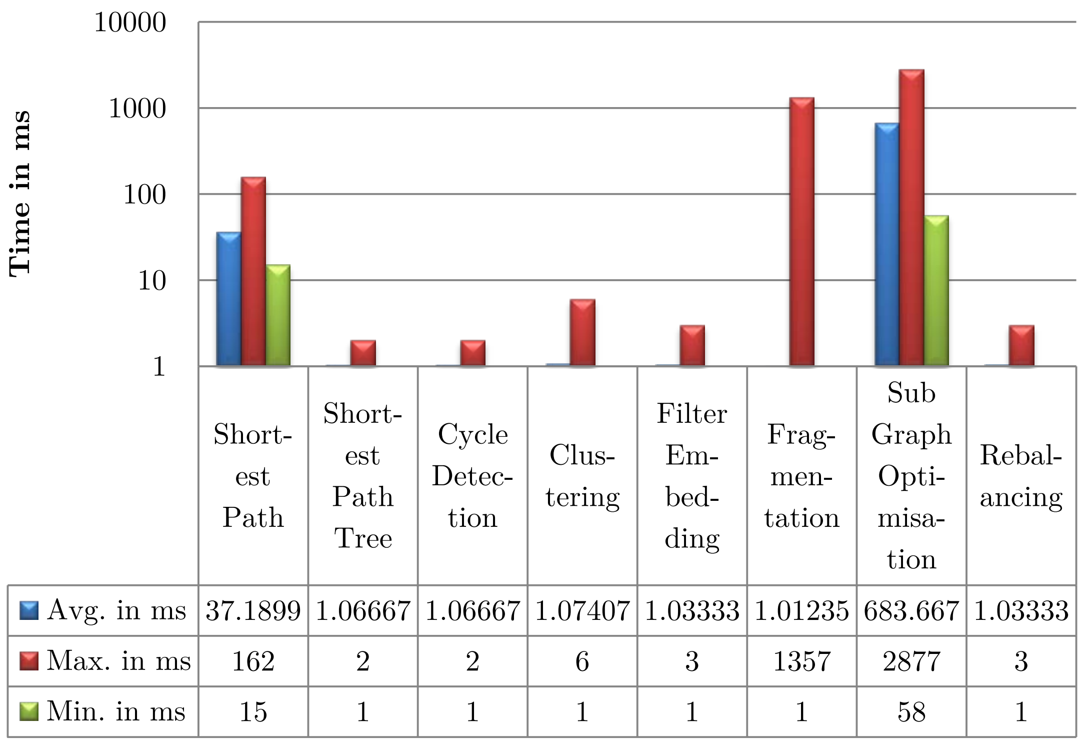


Figure 7.16: Data rate reduction for 25 switches



	Short-est Path	Short-est Path Tree	Cycle Detec-tion	Clus-tering	Filter Em-bed-ding	Frag-men-tation	Sub Graph Opti-misa-tion	Rebal-ancing
🟦 Avg. in ms	37.1899	1.06667	1.06667	1.07407	1.03333	1.01235	683.667	1.03333
🟥 Max. in ms	162	2	2	6	3	1357	2877	3
🟩 Min. in ms	15	1	1	1	1	1	58	1

Figure 7.17: Time consumption for 10 switches and 5 publishers with smaller fragments

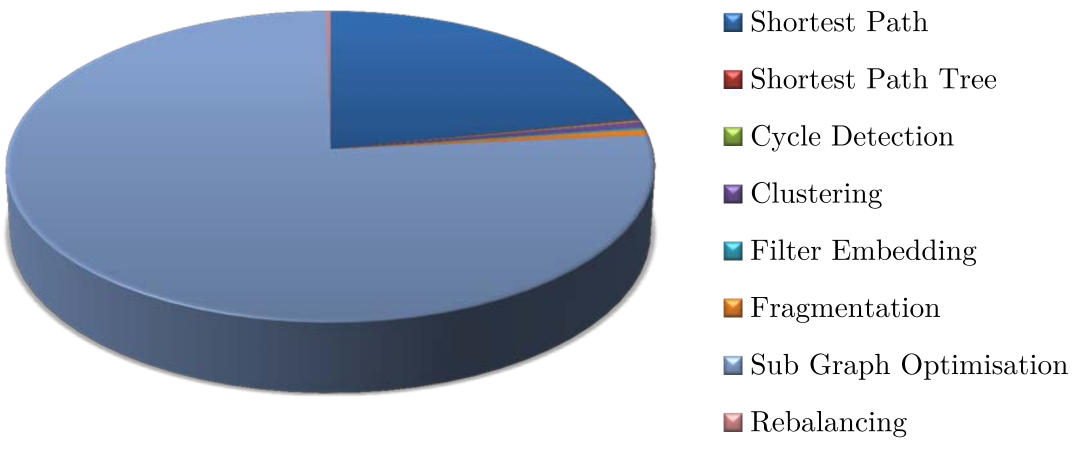
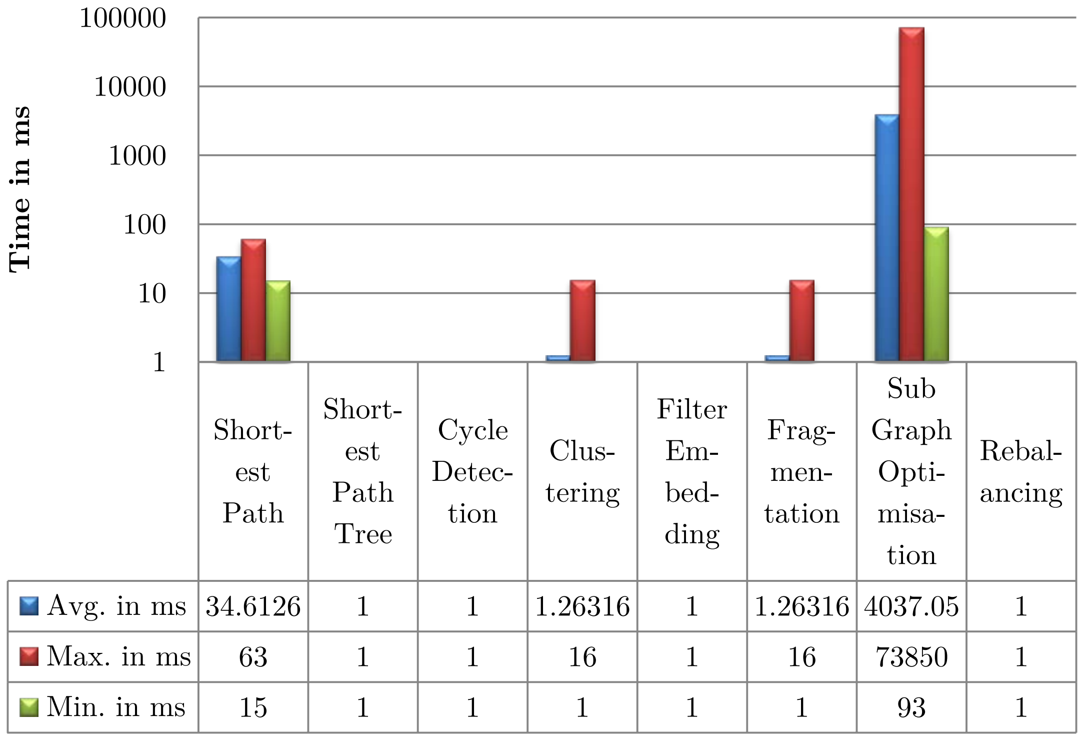


Figure 7.18: Time distribution for 10 switches and 5 publishers with smaller fragments



	Shortest Path	Shortest Path Tree	Cycle Detection	Clustering	Filter Embedding	Fragmentation	Sub Graph Optimisation	Rebalancing
🟦 Avg. in ms	34.6126	1	1	1.26316	1	1.26316	4037.05	1
🟥 Max. in ms	63	1	1	16	1	16	73850	1
🟩 Min. in ms	15	1	1	1	1	1	93	1

Figure 7.19: Time consumption for 10 switches and 10 publishers with smaller fragments

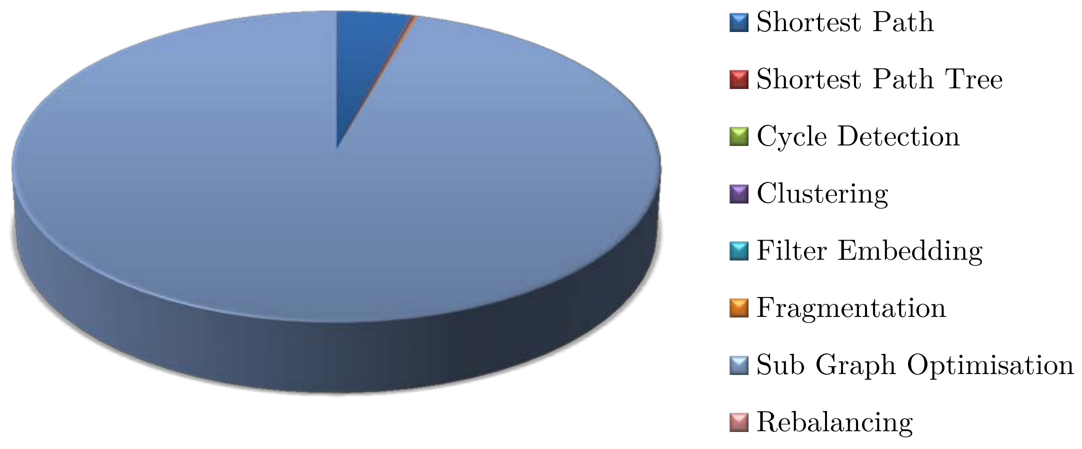
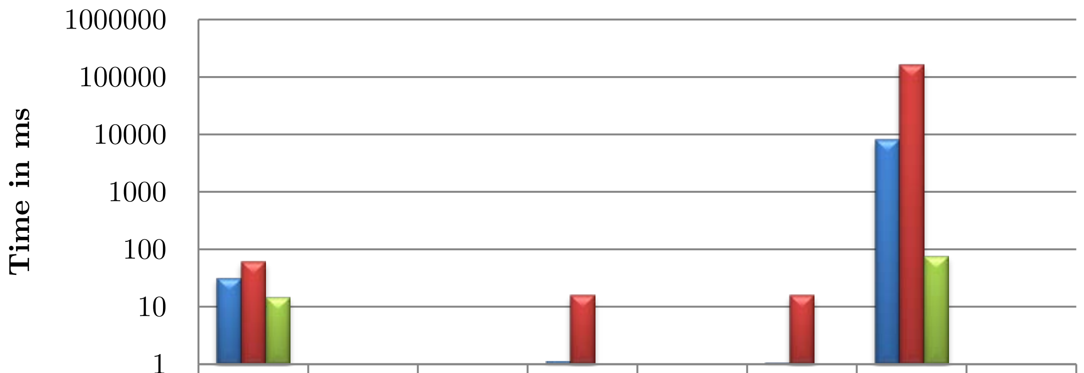


Figure 7.20: Time distribution for 10 switches and 10 publishers with smaller fragments



	Shortest Path	Shortest Path Tree	Cycle Detection	Clustering	Filter Embedding	Fragmentation	Sub Graph Optimisation	Rebalancing
🟦 Avg. in ms	32.4118	1	1	1.15464	1	1.07732	8187.48	1
🟥 Max. in ms	62	1	1	16	1	16	172146	1
🟩 Min. in ms	15	1	1	1	1	1	78	1

Figure 7.21: Time consumption for 10 switches and 20 publishers with smaller fragments

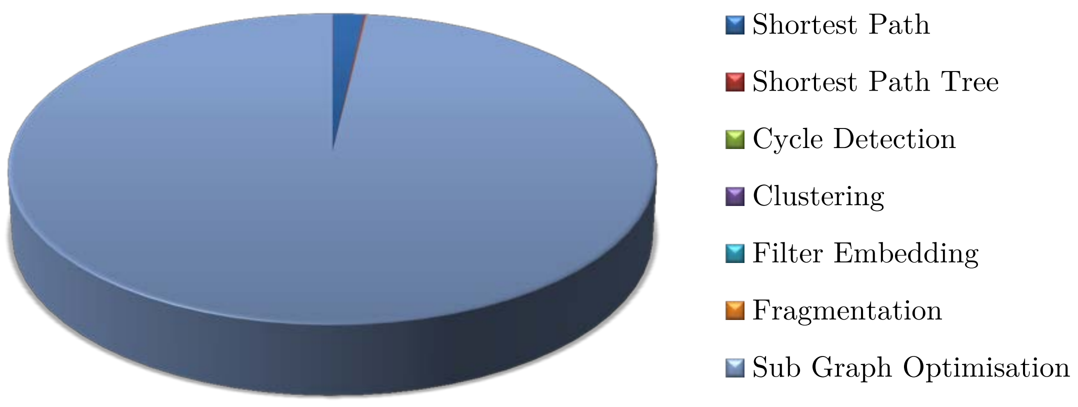


Figure 7.22: Time distribution for 10 switches and 20 publishers with smaller fragments

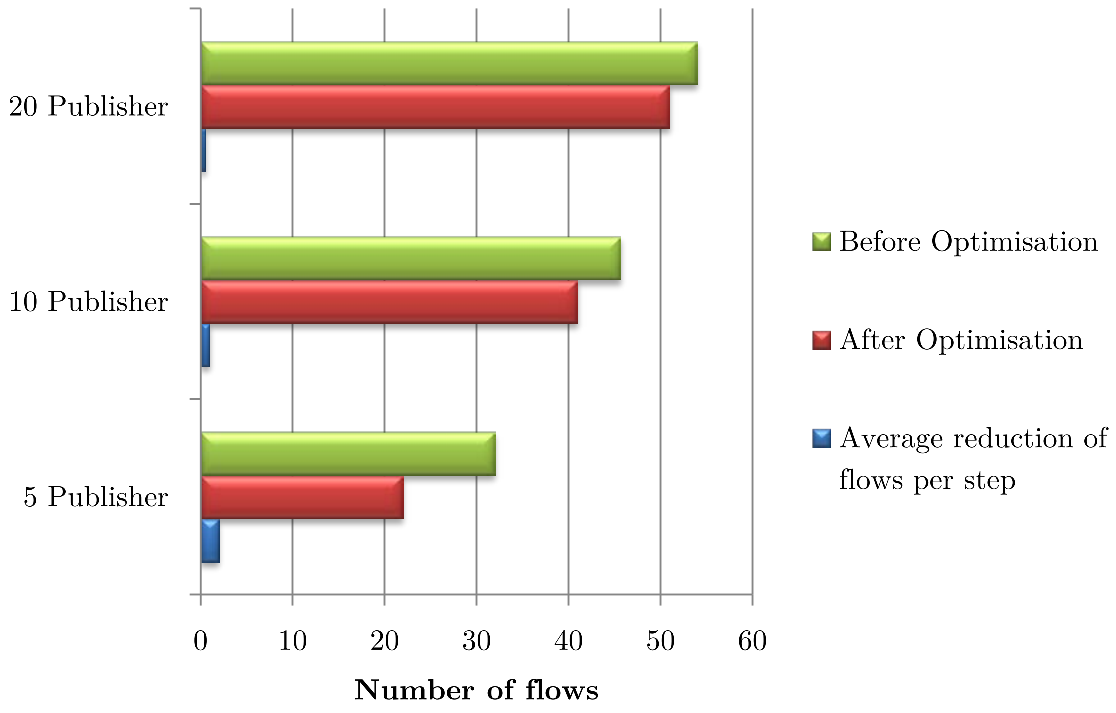


Figure 7.23: Flow reduction for 10 switches with smaller fragments

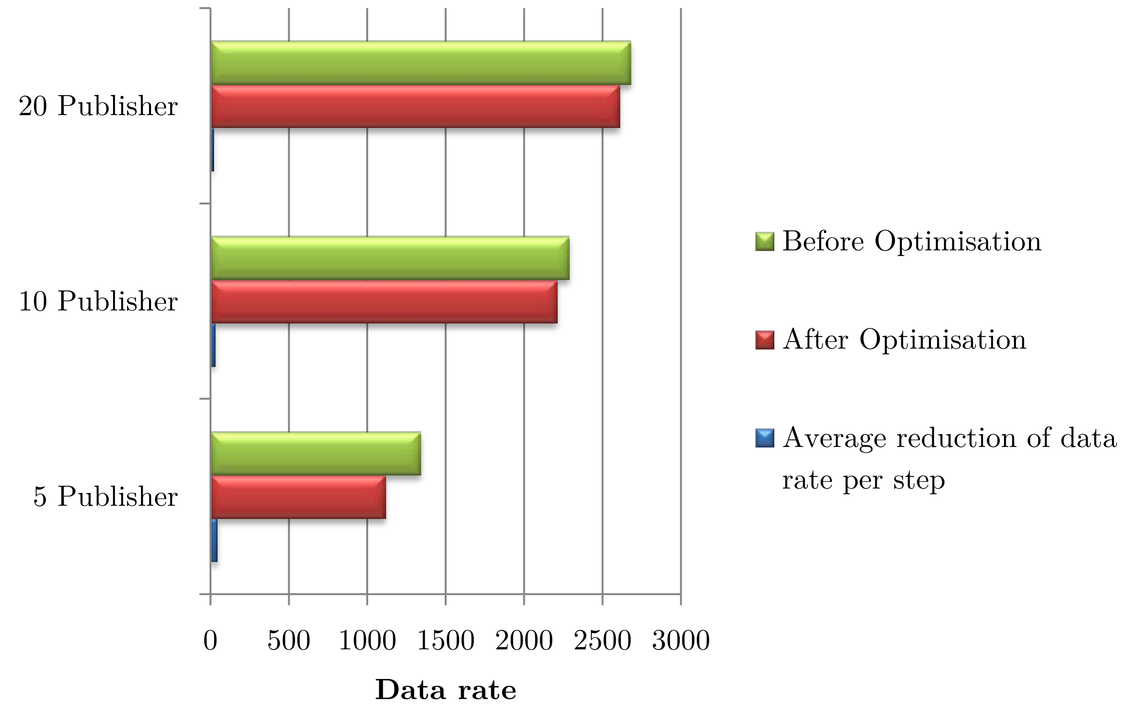


Figure 7.24: Data rate reduction for 10 switches with smaller fragments

because it means, we can use fragmentation to reduce the size of the sub graphs and still have good results while reducing the time consumption significantly. This is especially interesting for bigger graphs where it is necessary to fragment the graph.

Chapter 8

Summary and Outlook

We have shown that it is possible to divide the route optimisation problem in a publish/subscribe network into multiple smaller problems which represent the whole problem respectively. In chapter 4 on page 15 we have identified and modelled which parts are important and how we can represent them suitably for an optimisation formulation. From the beginning, we started to find problems as small as possible and with the potential to work together.

In chapter 5 on page 24, we presented several solutions for the identified problems and considered their advantages and disadvantages. Not all were suitable for our endeavour, so we choose the most suitable ones and combined them with several mechanisms and transformations in chapter 6 on page 50 into a framework that can solve the overall problem. The strength of the framework is the composition out of different components and the easy exchangeability of these components.

In chapter 7 on page 61, we have evaluated the framework with different settings and have shown its strengths and weaknesses. The evaluation has also shown where future work can start and which parts have the most potential.

Outlook

There are at least three points where future work can start with. One is the parallelisation of the execution of the components. Almost every component can be parallelised which allows for better performance and usage of multiple cores. This can help greatly in the sub graph optimisation where time is an issue. But with parallelisation comes a few problems. One is synchronisation, but normal multi-threading techniques should be enough to solve this. Parallelisation is also interesting because the solver itself does not support parallelisation. We could run multiple sub graph optimisations at the same time and utilise all cores of a machine.

Another point is what we can call *continuous optimisation*. This means adapting the framework for dynamic systems. This can be done by completing the cycle in the *iteration phase* and adding some mechanisms that detect where changes were made, which publishers and subscribers were added and so on. Then the framework can react and compute new sub graphs each time new participants join. The question will be how big the sub graphs should be and how exactly new participants should be handled. Should they first be added with a shortest path and then a sub graph optimisation? And how can we change the weight on the edges accordingly?

At last an obvious future work is the optimisation of the algorithms or replacing the algorithms with better performing ones. The clustering algorithm is a clear choice for this as there is high potential in good clustering of the filters and the resulting sub graphs.

It is also still open how good the solution compared to a theoretically optimal one is. Although, some components can compute an optimal solution for their input and according to the constraints, for example the shortest path computation and sub graph computation, some components will reduce the quality. At least with the current selection of the components. Every component and the framework itself can be analysed and examined for this.

Bibliography

[1] AIMMS. URL http://business.aimms.com.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*, pp. 53–61. ACM, 1999.

[3] E. Althaus, T. Polzin, S. Daneshmand. Improving linear programming approaches for the Steiner tree problem. In *Experimental and Efficient Algorithms*, volume 2647 of *Lecture Notes in Computer Science*, pp. 1–14. Springer Berlin Heidelberg, 2003.

[4] Y. P. Aneja. An integer linear programming approach to the Steiner problem in graphs. *Networks*, 10(2):167–178, 1980.

[5] R. Baldoni, A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. *DIS, Universita di Roma La Sapienza, Tech. Rep*, 2005.

[6] M. L. Balinski. Integer programming: methods, uses, computations. *Management Science*, 12(3):253–313, 1965.

[7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pp. 262–272. 1999.

[8] S. Bhowmik. *Distributed control algorithms for adapting publish/subscribe in software defined networks*. Master's thesis, University of Stuttgart, 2013. URL http://elib.uni-stuttgart.de/opus/volltexte/2013/8820.

[9] A. Bley. An integer programming algorithm for routing optimization in IP networks. *Algorithmica*, 60(1):21–45, 2011.

[10] J. A. Briones-Garcıa, B. Koldehofe, K. Rothermel. Spine: Adaptive publish/subscribe for wireless mesh networks. *Studia Informatika Universalis*, 7(3):320–353, 2009.

[11] S. Cabello, E. W. Chambers, J. Erickson. Multiple-source shortest paths in embedded graphs. *SIAM Journal on Computing*, 42(4):1542–1571, 2013.

[12] A. Campailla, S. Chaki, E. Clarke, S. Jha, H. Veith. Efficient Filtering in Publish-subscribe Systems Using Binary Decision Diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, pp. 443–452. IEEE Computer Society, 2001.

[13] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[14] A. Carzaniga, M. Rutherford, A. Wolf. A routing scheme for content-based networking. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 2, pp. 918–928. 2004.

[15] I. T. R. Center. The Gryphon Project. URL `https://www.research.ibm.com/distributedmessaging/gryphon.html`.

[16] A. Chanda. *Content delivery in software defined networks*. Rutgers the State University of New Jersey - New Brunswick, 2013.

[17] A. Chanda, C. Westphal, D. Raychaudhuri. Content based traffic engineering in software defined information centric networks. *Computing Research Repository*, 2013.

[18] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047, 2000.

[19] C. Chitra, P. Subbaraj. Multiobjective optimization solution for shortest path routing problem. *International Journal of Computer and Information Engineering*, 4(2):77–85, 2010.

[20] V. Chvátal. *Linear programming*. Freeman, New York, 15th edition, 1997.

[21] W. J. Cook, W. H. Cunningham, W. R. Pulleyblank, A. Schrijver. *Combinatorial optimization*. Wiley, New York, 1998.

[22] S. Dharmapurikar, P. Krishnamurthy, D. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, 2006.

[23] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[24] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Rec.*, 30(2):115–126, 2001.

[25] S. Gilpin, S. Nijssen, I. Davidson. Formalizing hierarchical clustering as integer linear programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. 2013.

[26] M. X. Goemans. *Analysis of linear programming relaxations for a class of connectivity problems*. Ph.D. thesis, Massachusetts Institute of Technology, 1990.

[27] A. Gupta, O. D. Sahin, D. Agrawal, A. E. Abbadi. Meghdoot: content-based publish/-subscribe over P2P networks. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pp. 254–273. Springer-Verlag New York, Inc., 2004.

[28] X. Han, P. Kelsen, V. Ramachandran, R. Tarjan. Computing minimal spanning subgraphs in linear time. *SIAM J. Comput.*, 24(6):1332–1358, 1995.

[29] J. N. Hooker. Logic, optimization, and constraint programming. *INFORMS Journal on Computing*, 14(4):295–321, 2002.

[30] F. K. Hwang, D. S. Richards. Steiner tree problems. *Networks*, 22(1):55–89, 1992.

[31] E. L. Johnson. Facets, subadditivity and duality for group and semi-group problems. *SIAM*, 1980.

[32] S. Khuller, B. Raghavachari, N. Young. Balancing minimum spanning trees and shortest-path trees. *Algorithmica*, 14(4):305–321, 1995.

[33] D. E. Knuth, J. H. Morris, Jr, V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[34] B. Koldehofe, F. Dürr, M. A. Tariq, K. Rothermel. The power of software-defined networking: line-rate content-based routing using OpenFlow. In *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, pp. 3:1–3:6. ACM, 2012.

[35] J. Kratica, D. Dugošija, A. Savić. A new mixed integer linear programming model for the multi level uncapacitated facility location problem. *Applied Mathematical Modelling*, 38(7–8):2118 – 2129, 2014.

[36] J. Kruskal, Joseph B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the ACM*, 7(1):48–50, 1956.

[37] G. Li, S. Hou, H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pp. 447–457. 2005.

[38] C.-M. Lin, Y. T. Tsai, C. Y. Tang. Balancing minimum spanning trees and multiple-source minimum routing cost spanning trees on metric graphs. *Information processing letters*, 99(2):64–67, 2006.

[39] Y. Liu, B. Plale, et al. Survey of publish subscribe event systems. *Computer Science Dept, Indian University*, 16, 2003.

[40] G. Mühl. Generic constraints for content-based publish/subscribe. In *Cooperative Information Systems*, volume 2172 of *Lecture Notes in Computer Science*, pp. 211–225. Springer Berlin Heidelberg, 2001.

[41] G. B. Mishra. *Providing in-network content-based routing using OpenFlow*. Master's thesis, University of Stuttgart, 2013. URL `http://elib.uni-stuttgart.de/opus/volltexte/2013/8560`.

[42] G. Mühl. *Large-scale content-based publish-subscribe systems*. Ph.D. thesis, TU Darmstadt, 2002.

[43] C. Park, Y. Seo, K. youl Park, Y. Lee. The concept and realization of context-based content delivery of NGSON. *Communications Magazine, IEEE*, 50(1):74–81, 2012.

[44] P. Pietzuch, J. Bacon. Hermes: a distributed event-based middleware architecture. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pp. 611–618. 2002.

[45] P. R. Pietzuch. *Hermes: A scalable event-based middleware*. Ph.D. thesis, University of Cambridge, 2004.

[46] R. C. Prim. Shortest connection networks and some generalizations. *The Bell Systems Technical Journal*, 36(6):1389–1401, 1957.

[47] D. A. G. Pritchard. *Linear programming tools and approximation algorithms for combinatorial optimization*. Ph.D. thesis, University of Waterloo, 2009.

[48] S. Ratnasamy, M. Handley, R. M. Karp, S. Shenker. Application-Level Multicast Using Content-Addressable Networks. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, pp. 14–29. Springer-Verlag, 2001.

[49] Z. Reza, H. Masoud. *Facility location: concepts, models, algorithms and case studies*. Springer-Verlag Berlin Heidelberg, 2009.

[50] G. Robins, A. Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 770–779. SIAM, 2000.

[51] G. Robins, A. Zelikovsky. Minimum steiner tree construction. *The Handbook Of Algorithms For VLSI Physical Design Automation*, pp. 487–508, 2009.

[52] B. Sağlam, F. S. Salman, S. Sayın, M. Türkay. A mixed-integer programming approach to the clustering problem with an application in customer segmentation. *European Journal of Operational Research*, 173(3):866–879, 2006.

[53] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, M. Ward. Gryphon: an information flow based approach to message brokering. *eprint arXiv:cs/9810019*, 1998.

[54] A. Sultan. *Linear programming: an introduction with applications*. Academic Press, Boston, 1993.

[55] H. A. Taha. *Integer programming: theory, applications, and computations*. Academic Press, New York, 1975.

[56] M. Tariq, B. Koldehofe, G. Koch, K. Rothermel. Providing probabilistic latency bounds for dynamic publish/subscribe systems. In *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pp. 155–166. Springer Berlin Heidelberg, 2009.

[57] M. A. Tariq, B. Koldehofe, G. G. Koch, I. Khan, K. Rothermel. Meeting subscriber-defined QoS constraints in publish/subscribe systems. *Concurrency and Computation: Practice and Experience*, 23(17):2140–2153, 2011.

[58] M. A. Tariq, B. Koldehofe, G. G. Koch, K. Rothermel. Distributed spectral cluster management: a method for building dynamic publish/subscribe systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pp. 213–224. ACM, 2012.

[59] M. A. Tariq, B. Koldehofe, K. Rothermel. Efficient Content-based Routing with Network Topology Inference. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, pp. 51–62. ACM, 2013.

[60] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[61] M. Waldvogel, G. Varghese, J. Turner, B. Plattner. Scalable high-speed prefix matching. *ACM Transactions on Computer Systems*, 19(4):440–482, 2001.

[62] P. Weiner. Linear pattern matching algorithms. *IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[63] G. Zhao, B. Luo, J. Ma. Matching scenarios patterns by using linear programming. In *Proceedings of the Fourth International Conference on Fuzzy Systems and Knowledge Discovery*, volume 3, pp. 346–350. IEEE Computer Society, 2007.

All links were last followed on May 06, 2014.

Declaration