Concepts and Algorithms for Efficient Distributed Processing of Data Streams

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Stamatia Rizou
aus Athen

Hauptberichter: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Mitberichter: Prof. Dr. Timos Sellis

Tag der mündlichen Prüfung: 26.11.2013

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2013

Acknowledgments

First, I would like to thank my Professor Kurt Rothermel, for giving me this unique opportunity to work with the group of Distributed Systems under his supervision. His feedback and comments during our regular meetings refined the concepts and fundamentals of my work and improved its scientific value. Next, I would like to thank my colleague Frank Dürr. The output of my research is a result of long discussions between us on several aspects of my work. This work would not have been possible without his invaluable contribution and his continuous support and guidance.

I would like also to thank all my colleagues in the Distributed System group. I would like to give special thanks to my colleagues Ralph Lange, Lars Geiger, Harald Weinschrott and Nazario Cipriani for our collaboration in the Nexus project and my colleague, Andreas Grau for his help on the implementation of one of my placement algorithms on NET Cluster.

I would like also to thank Prof. Timos Sellis for supporting my work from the very beginning, when he showed me the way to apply for a PhD in Germany until the end, by acting as a reviewer for my thesis.

During my stay in Stuttgart, I had the chance to meet special people that made this city feeling like home. Therefore, I would like to thank my friends, who shared the good and bad moments during the four years I spent in Stuttgart. My flatmate Theodora, but also Gianna and Maria who joined later and my friends Vangelis, Alexia, Angelos, Alexandros, Loukianos.

Finally, I want to thank my family for their unconditional love and care that gives me always strength to go on. My parents Vangelis and Ritsa, and my brother Vasilis with his family, his wife Mina and my two beloved nephews Vangelis and Manos.

Contents

List of Figures

List of Tables

Abstract

During the last years, the proliferation of modern devices capable of capturing context information through various sensors has triggered the blossom of context-aware systems, which automatically adapt their behaviour based on the detected context. For many emerging context-aware applications, context may include a huge amount of entities possibly dispersed geographically over a wide area. In such large-scale scenarios, the efficient processing of context information becomes a challenging task. In this dissertation, we are going to focus on the problem of the efficient processing of context information. In particular, we will consider the problem of deriving high-level context information, also referred to as situation in the literature, from sensor data streams captured by a large set of geographically distributed sensors.

First, we present the architecture of a distributed system that uses reasoning algorithms to detect situations in an overlay network of data stream processing operators. Then we are going to introduce our strategies for the optimal distribution of data processing between processing nodes in order to save network resources, by optimizing for bandwidth-delay product, and fulfill given QoS requirements, such as end-to-end latency constraints. To this end, we formulate three (constrained) optimization problems, which search for an optimal placement of operators onto physical hosts with respect to different application constraints. The proposed algorithms are executed in a distributed way, by using local knowledge of the system. Our evaluation shows that our algorithms achieve good approximations of the optimal solutions, while inducing limited communication overhead.

Deutsche Zusammenfassung

Während der letzten Jahre hat die Anzahl an vernetzten Sensoren und mit Sensoren ausgestatteten Geräten wie Smartphones stark zugenommen. Diese weitreichende Verfügbarkeit von Sensorinformationen hat zu einer Vielzahl so genannter kontextbezogener Anwendungen z.B. in der Logistik, der intelligenten Verkehrssteuerung, der Produktion („Smart Factory“) oder der Energiewirtschaft („Smart Grid“) geführt, welche in der Lage sind, ihr Verhalten automatisch an ihren Kontext anzupassen.

Viele Anwendungsszenarien basieren dabei auf einer großen Anzahl von Sensoren (Datenquellen), Kontextdatenprozessoren, welche aus Sensordaten höherwertige Kontextinformationen (Situationen) ableiten und Anwendungen (Datensenken), welche geographisch weit verteilt und über Weitverkehrsnetze bzw. das Internet miteinander vernetzt sind. Die Quellen produzieren dabei u.U. großvolumige Datenströme (z.B. kontinuierliche Videoaufzeichnungen) bzw. eine Vielzahl von Datenströmen (z.B. aus großen Netzen von Temperatursensoren, Kontaktschleifen entlang von Straßen, Verbrauchsdaten von „Smart-Meters“, usw.). Insbesondere in solch großen Szenarien stellt die effiziente Kommunikation und Verarbeitung von Sensordatenströmen eine große Herausforderung dar, der sich diese Dissertation widmet. Das übergeordnete Ziel dieser Arbeit ist dabei der Entwurf von Konzepten und Mechanismen zur effizienten verteilten Verarbeitung von Sensordatenströmen in einem Netz aus Kontextdatenprozessoren zur Ableitung von höherwertigen Situationen zur Unterstützung kontextbezogener Anwendungen.

Hierzu leistet diese Arbeit die folgenden Beiträge. Zunächst wird eine Architektur zur verteilten Verarbeitung von Sensordaten in einem dem physischen Netz überlagerten Overlay-Netz aus Datenprozessoren – so genannten Opera-

toren – entworfen sowie das Konzept der Operatorgraphen zur Modellierung der verteilten Verarbeitung formal eingeführt. Dieses Konzept ermöglicht insbesondere die verteilte Ausführung von Situationserkennungsoperatoren, z.B. basierend auf Bayes'schen Netzen.

Des Weiteren werden verschiedene Algorithmen zur optimalen Verteilung der Operatoren eines Operatorgraphen auf physischen Rechnern (Hosts) im Overlay-Netz vorgeschlagen (Operatorplatzierung). Ziel der Optimierung ist dabei die Steigerung der Skalierbarkeit durch die Entlastung des physischen Kommunikationsnetzes. Hierbei wird im Detail die Minimierung des Bandbreiten-Verzögerungsprodukts der Datenströme eines Operatorgraphen betrachtet. Ferner wird dieses zunächst reine Optimierungsproblem durch Randbedingungen in Form anwendungsspezifischer Dienstgüteeigenschaften (Quality of Service) erweitert. Betrachtet wird hierbei vor allem die Ende-zu-Ende-Verzögerung von den Datenquellen zur -senke als wichtige Randbedingung zeitkritischer Anwendungen und Prozesse. Neben der Betrachtung der Kommunikationsverzögerung werden dabei auch verarbeitungsintensive Anwendungen durch die Einbeziehung der Verarbeitungszeit auf den Rechenknoten berücksichtigt. Ein wesentlicher Beitrag dieser Arbeit ist ein verteilter Algorithmus zur näherungsweisen Lösung des Optimierungsproblems durch dezentrale Platzierungsentscheidungen der Operatoren basierend auf lokalem Wissen. Dieser Algorithmus wird in weiteren Schritten so erweitert, dass eine gegebene Ende-zu-Ende-Verzögerung eingehalten wird. Die im Rahmen dieser Dissertation durchgeführten Evaluierungen zeigen, dass diese Verfahren zu sehr guten Annäherungen der optimalen Lösung mit nur geringem Kommunikationsaufwand zur Ausführung des verteilten Algorithmus führen.

1 Introduction

1.1 Motivation

Context-aware systems adapt seamlessly their behaviour according to context changes, i.e., without the explicit intervention of the end-user. Context could be any relevant information regarding the interaction of the application and the user. More formally, context has been defined according to Dey [4] as "any information that can be used to characterize the situation of entities (i.e. whether a person, place, or object) that are considered relevant to the interaction between a user and an application, including the user and the application themselves". The automatic adaptation of the system to the current context leads to the "pervasive computing" vision, where applications are adapted to satisfy user expectations.

Context-aware applications include navigation and assistance, environmental monitoring, smart power grids, traffic and transportation. Imagine, for instance, a context-aware application that suggests minimal delay routes using public transportation in a smart city. The system monitors the current traffic congestion and detects situations that can lead to deviation from the normal traffic patterns, e.g., car accidents, traffic lights out of use, cable-fire at the tram. The detection of situations that can affect the normal function of the public means of transportation plays a critical role in order for the system to adapt to current conditions, for instance, to select alternative routes that can reduce the trip delay. Therefore context-aware systems should be able to interpret context that can be directly acquired from the environment to meaningful situations that are relevant to the application.

Sensors constitute the technological enabler to capture continuously sensor

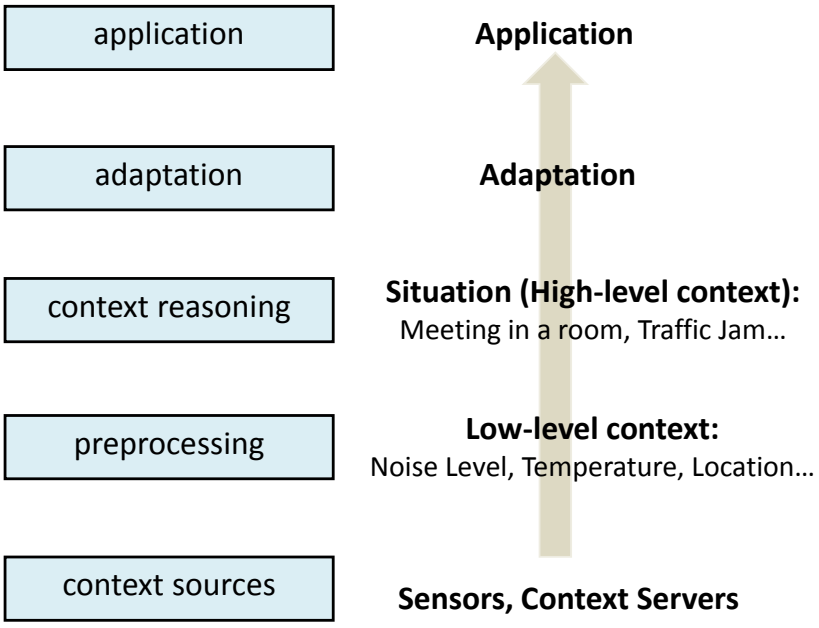


Figure 1.1: Layered Architecture of Context Aware Systems

data to monitor environmental variables such as temperature, humidity, or wind. Sensors could be stationary sensors, such as temperature sensors, induction loops, road-side units, cameras that are deployed at fixed locations and continuously track the current context, or mobile embedded in mobile devices such as smart phones that are carried by users. Managing, interpreting and processing sensor data is critical for the success of context-aware systems, since their behaviour relies on context information.

In order to achieve this goal, context-aware systems typically implement a layered architecture as shown in Figure 1.1. In the bottom layer, context sources provide either static data such as city maps or dynamic data such as sensor data by monitoring the environment. Sensor data can be translated to observable context, which is the primitive form of context since it can be directly acquired by sensors. Then, several pieces of observable context are combined to detect high level context changes and situations. Technically the correlation of low level context data to deduce situations can be real-

ized through context reasoning algorithms [97]. Finally the system adapts according to the detected situations.

To illustrate this process through an example, consider the scenario of the automated calculation of shortest routes in a smart city. First a sensor network, which includes induction loops, and road-side units is deployed along the roads in the city. To calculate the route between two points, the correlation of static data (city maps, bus routes) and dynamic data currently acquired from sensors, e.g., which streets are currently crowded, have to be collected and processed. Thus, the detection of the situation "traffic congestion" in a road segment would increase the delay estimation of the trip and adapt the estimation to current conditions. Finally, a response is returned back to the application. Now imagine that multiple users query the system to get notified about different situations. Context data that are generated from sensors deployed on different locations have to be transferred and processed in a timely and efficient manner. In such a setting, the amount of data that are transferred in the system affects the performance of the system, since an excessive amount of data could lead to bottlenecks and network congestion.

Already for these simple scenarios, we need several pieces of low level context (e.g., distances between cars, average speed), which could come from different sources (e.g., cars, road-side units, cameras on bridges). The question that naturally arises is at which server to correlate this distributed context data. One simple solution is to collect all the necessary information at a central server and perform the reasoning there. In line with this centralised approach, many of the existing context aware systems are designed to support specific use case scenarios (e.g., MS Easy Living [22] or Semantic Space [80]) and cover a limited geographical area (e.g,. one building or conference room). However this naive solution cannot provide a scalable solution in scenarios with a large number of geographically distributed context sources that is subject to our work due to several drawbacks: First, it does not utilize communication resources efficiently since unfiltered data has to be sent to a possibly distant central server. This increases the network load and might

lead to communication bottlenecks. Secondly, the timeliness of situation detection may increase since the communication with a distant server induces a longer delay, and communication bottlenecks further slow down this communication. To avoid these problems, it seems reasonable to distribute the reasoning process to several servers across the network.

An alternative to the centralised approach that increases scalability is the partitioning of the network. In [50] multiple servers, each one responsible for a certain geographic region, are used to perform context reasoning. Although this approach is a first step towards distributed context reasoning, it still executes reasoning tasks centrally on a dedicated server. Therefore, it may lead to poor utilization of network resources and limited system performance. Other existing approaches that enable the distribution of the reasoning task [51, 100] are method-specific, since they refer to a specific reasoning algorithm, and they do not address the problem of distributed context reasoning as an optimization problem to achieve efficient utilization of network resources and high system performance.

Given the limitations of existing approaches to provide a solution that allows the efficient distribution of reasoning tasks, our work addresses some of the challenges imposed by distributed context reasoning. First, we present an abstraction that allows for the distribution of reasoning tasks. Our proposed model is based on the operator concept which represents a basic reasoning task. Typically, the detection of a situation involves several sub-tasks to process sensor data from several distributed sensors, detect sub-situations, and combine these partial results to the final situation. By encapsulating processing tasks into processing operators, we allow for the distribution of the processing to several servers. Thus, the proposed system is based on a generic formalization of distributed reasoning that allows for the use of different reasoning algorithms and the distribution of the reasoning process according to different optimization and QoS criteria.

In detail, our system uses a situation-centric model, which contains predefined situation patterns, called *situation templates* that are stored as pre

knowledge in the system. Situation templates are built from observable context and processing units called *operators*. Different reasoning methods such as distributed Bayesian Networks or Petri Nets can be supported through different operators implementing the specific context correlators. Generally each situation template forms a graph of operators, which cooperatively performs a reasoning task. At runtime, situation detection is initialized by the creation of a *logical plan*, which is derived from a situation template. This plan describes the detection of a concrete situation at a certain location or for a given object by an operator graph. Subsequently the system finds a mapping of the operators of the logical plan to physical hosts according to the optimization goal of the operator placement. The result is a *physical plan* that is finally deployed to execute the reasoning process in an overlay network of operators.

Given this model, we argue that the problem of optimally placing operators onto a network of physical nodes, is an optimization problem that applies to distributed context management systems as well as to Complex Event Processing (CEP) and Distributed Stream Management Systems (DSMS). To this end, we focus on operator placement strategies that search for optimal mappings of operators to physical nodes such that the network load is minimized and application-defined latency restrictions are satisfied. Operator placement algorithms have been investigated mainly in the context of data stream processing [2, 3, 28, 84], but also in CEP systems [59, 99]. Overall the existing placement algorithms focus on different optimization objectives [63], e.g., latency, bandwidth or load depending on the system model and the application constraints assuming central [28, 52] or distributed network control [59, 84, 99]. In this dissertation, we target large-scale scenarios, where a centralized global view on the system is not possible. To this end, we propose operator placement algorithms that use only local knowledge to optimize for network load and satisfy application-defined latency constraints.

In more detail, the operator placement problems presented in this dissertation, target *communication intensive* applications, which require the online

processing of large amount of data. These applications may include environmental monitoring, IP network traffic analysis, global sensor networks. To this end, we first look at an *optimization problem* where the goal is to minimize the *network load* put on the system by the operator network. By minimizing the network load, we put less burden at the network and thus we contribute to the avoidance of network congestion and increase the scalability of the system. Furthermore, we formulate two *constrained optimization problems*, which consider application-defined latency constraints, under different assumptions on the application characteristics. In particular, we distinguish the following two categories of communication intensive applications, depending on the size of the data units that they communicate:

- Applications with *negligible processing delay*, where network latency is the main part of the end-to-end delay.

- Applications with *substantial processing delay*, where transmission and processing delays are substantial parts of the overall end-to-end latency.

For each of the two categories, we present a constrained optimization operator placement problem that considers a maximum end-to-end delay of detecting situations. In that respect, the application can specify a threshold of the maximum latency that it can tolerate. First we target the applications, where the processing delay is negligible, we consider the network latency as the dominant factor of the end-to-end latency. To this end, we propose an operator placement algorithm that solves the constrained optimization problem and we analyse the interdependence of the bandwidth-delay product and delay optimization. Then we provide a solution for the applications with substantial processing delay. In that case, the end-to-end latency is affected by network latency as well as by the processing delay. Therefore, we extend our system model to consider processing and transmission delays, and we present another operator placement algorithm solving this constrained optimization problem.

Before we give a detailed overview on the individual contributions of this

dissertation, we introduce the research project "Nexus", which provided the framework of this work.

1.2 Background

Our research in the area of distributed context reasoning is embedded into the joint research project Nexus (Collaborative research Centre 627) of the University of Stuttgart. The Nexus project is centred around the concept of a context model (also called world model) that provides context-aware applications with context information. This model includes static context information such as map information as well as dynamic information stemming from sensors. Moreover, this information can be classified as directly observable context information and high-level context information (situations). Since the Nexus platform federates the context models of the different providers and offers context-aware applications a global, consistent view on their context data, centralized context management systems are obviously insufficient. Therefore in Nexus we have adopted a scalable, distributed architecture that integrates different services such as distributed query processing and context reasoning. One of the core functionalities of the Nexus platform is the context reasoning service, which is relevant to the work presented in this dissertation. In particular, a basic contribution of this dissertation is to provide the concepts for efficiently deriving situations from observable context information as part of the Nexus model. Next, we will briefly describe the architecture of the Nexus platform, which gives the background framework for our work.

1.2.1 Architecture

Nexus uses a three layer architecture, where applications are located on the top layer. The middle layer forms a federation, which integrates the data stored on context providers at the bottom layer [77]. Nexus provides different services based on the application needs. One Nexus core service is the query processing service. For this service, Nexus follows a request-response model

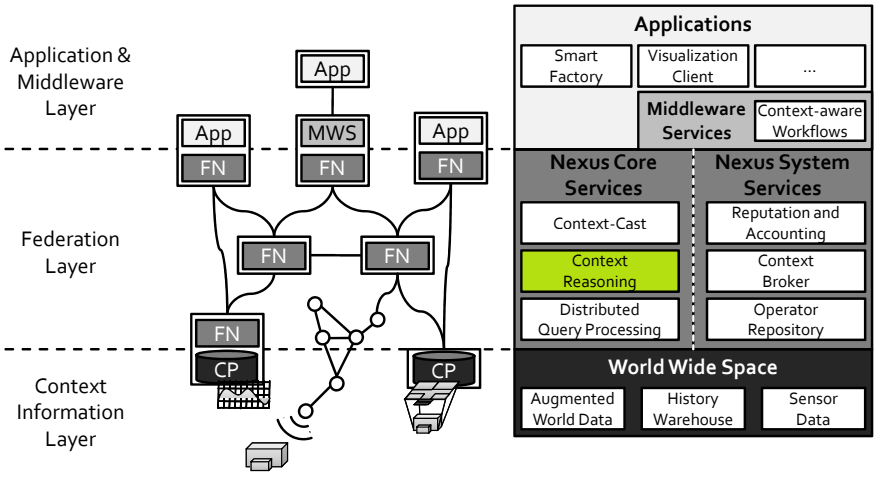


Figure 1.2: Extended Nexus Architecture

receiving queries from applications. Based on spatial restrictions in the query, the federation layer determines the relevant context providers and forwards the query to them. In a second step, it integrates the results and sends them back to the application [64]. Another important service in Nexus platform is the event management service. In Nexus, physical world events can be observed, by calculating the occurrence probability and comparing this to the specified threshold probability [16].

In this dissertation we tackle the problem of *distributed context reasoning* that comes as an additional service of the Nexus platform. Our work is part of the extended Nexus platform that was designed during the second funding period of the Nexus project. The extended Nexus architecture [65] retains the idea of separating applications, federation, and data providers. However, the extensions add more flexibility to the federation layer and integrate historical data and situations within the context data layer. Figure 1.2 depicts the extended Nexus architecture with its three layers: (1) *Context Information Layer*, (2) *Federation Layer* and (3) *Applications & Middleware Layer*. In the next, we present the extended Nexus platform as shown in Figure 1.2.

1.2.2 Context Information Layer

The *Context Information Layer* consists of context servers from arbitrary providers. It provides context data at different level of details ranging from sensed context data, over static context data to historical context data. Historical data, such as the trajectory of a moving object or the value pattern of a thermometer, is stored by specialized history context providers. Such data can be integrated into the context model by means of meta data for attributes, which represents the period when an attribute value is valid. History context providers typically use lossy data compression algorithms, e.g., line simplification, to reduce the amount of data to be stored [56, 66]. The data from history context providers can be exported to history warehouses for more sophisticated analysis, e.g., to develop algorithms for traffic jam prognosis.

1.2.3 Federation Layer

The *Federation Layer* is a distributed platform for context services. It works on hybrid systems [42] and integrates infrastructure-based networks and ad-hoc networks of mobile devices, as depicted in Figure 1.2. There are two types of predefined Nexus services: *Platform Services* are context services typically used by applications, such as *Context Reasoning*, *Context Cast*, or *Stream Query Processing*. In contrast, *Core Services* provide the functionality on which the Nexus Platform Services rely, including *Context Broker*, *Reasoning Templates*, or *Operator Repository*.

Context Broker. The Context Broker discovers relevant context providers for query processing or situation recognition. To this end, it indexes all context providers by means of their models and allows for querying for relevant providers whose models intersect a certain clipping of the federated context model. Moreover, it provides distributed index structures [67] for accessing trajectory data on moving objects. These objects are not bound to a specific context provider and their trajectory data may be distributed over many

providers.

Distributed Query Processing. Streamed data is highly volatile, potentially infinite, and allows only sequential access. This calls for dedicated stream processing functionality to enable on-the-fly processing of streamed data. The **Operator Repository** enables stream processing, providing suitable data stream operators. To avoid load congestion on a particular site, partitioning and distributing queries across processing node is an essential step to make stream processing affordable.

Contextcast. The Contextcast service enables applications and services to send messages to entities with a certain context. Message distribution does not rely on explicit multicast groups, but uses an overlay network of context-based routers to forward messages instead. This approach is similar to content-based publish/subscribe systems, however, the forwarding structures are adapted to exploit properties of context information such as more gradual changes.

Context Reasoning. Context reasoning derives new knowledge from low level context. Since distributed context reasoning is the focus of this disseration, we will determine the details of the approach during this thesis. In brief, Nexus uses a situation-centric approach describing each situation by a set of rules, which constitutes a **Situation Template**. Each Situation Template generates a logical execution plan, a directed graph describing the data flow and the steps of the algorithm. As already mentioned, for scalability reasons, the situation recognition process must be distributed to several physical nodes. The distribution of the logical execution plans to physical machines is governed by factors such as latency, bandwidth, and load.

1.2.4 Applications and Middleware Layer

Finally, the *Applications & Middleware Layer* enables application specific additions to the platform. It is possible to outsource parts of the application logic to the execution environment, with dedicated machines performing

application specific tasks. The application logic can be moved into the middleware layer using **Context-aware Workflows** [109] together with *Context Integration Processes* [110].

1.3 Contributions

The focus of this dissertation is on the development of concepts and mechanisms for a distributed context reasoning system. In detail, the contributions of this work are:

- **Generic System Model for Context Processing.** We present a generic system model using an abstraction that allows us to handle the problem of distributed context reasoning as an operator placement problem, known from data stream processing. In particular, we adopt the operator graph model from stream processing to express the correlation of several pieces of context to detect a situation.

- **Architecture of a Distributed Context Reasoning System.** We introduce an architecture that enables distributed context reasoning by distributing the reasoning process to several physical nodes such that the system performance is improved. The distribution of the reasoning process is transparent to the application, which has access only to the final outcome of the reasoning process. Therefore, the design of the architecture decouples the two problems of context reasoning and operator placement.

- **Operator Placement Algorithm for Minimizing Network Load.** We present a placement algorithm that finds a mapping of operators to physical hosts such that the induced network load is minimized. By minimizing the network load, we contribute to the scalability of the system, since the system gets slower loaded and thus, can handle a large number of data stream tasks. The proposed algorithm works in a

distributed way, i.e. the operators place themselves on physical nodes based on their local view.

- **Operator Placement Algorithm with Latency Constraints.** We also present two operator placement algorithms that consider application-defined latency requirements. The ultimate goal is to fulfill application-defined latency constraints while minimizing the network load. Thus, apart from the optimization goal, here we try to fulfill also end-to-end latency constraints. In a first step, we target applications with negligible processing delay. Then, we consider applications with significant processing and transmission delay.

- **Evaluation of Operator Placement Algorithms.** As part of this dissertation, we provide an evaluation of the proposed operator placement algorithms by using a network simulator as well as an emulator test bed that allows a more accurate testing of the performance of the placement algorithm that considers processing delays.

1.4 Structure

The structure of the dissertation is as follows: In Chapter 2, we present an architecture for a distributed context reasoning system that has been designed in the frame of this dissertation. In that chapter we will introduce the operator graph model, which is a core model abstraction for our approach to distributed context reasoning. Moreover, we present the proposed architecture and we explain in detail its components and functionalities, before we explain the novelty of our proposed architecture with respect to the state of the art context-management systems. In Chapter 3, we present the operator placement problems and algorithms considered in this dissertation. For each of the three operator placement problems, we first present the system model that help us to formulate the problem, before we present the corresponding operator placement algorithm. Furthermore, at the end of this chapter, we

present related work in the area of operator placement algorithms with respect to the proposed operator placement algorithms. In Chapter 4, we present the evaluation results that were collected during the testing of the operator placement algorithms presented in Chapter 3 before we conclude our work and we discuss directions for future research in this area in Chapter 5.

2 Architecture

In this chapter, we are going to present our solution for the design of a distributed context reasoning system [94]. In our approach, context reasoning is used to detect high-level contextual changes of the environment, called situations, from various pieces of low-level context that can be directly acquired by sensors. Our work focuses especially on large-scale scenarios where the context sources are distributed and cover a large geographic area. Imagine for instance a navigation service in a smart city, where real-time information coming from cameras and sensors, located in different places in the network, is correlated with static data, such as city maps to detect traffic congestion points and derive optimal routes for users. Context data coming from different places across the route should be transmitted through the network in order to get processed and finally the result should be delivered to the application.

Although extensive work has been done on the representation and reasoning of context information, most existing context reasoning systems do not address or only address partially the efficient in-network processing of context data. In order to tackle this problem, we propose a novel architecture that uses a graph-based representation for reasoning tasks, which allows for their distributed execution in the network.

More precisely, our system model is based on the abstraction of the operator graph, which formulates the context reasoning task through a directed graph of processing units, called operators. We show later how this model can depict several problems from different application domains. The operator graph is used as an interface between the context reasoning algorithms and the network control layer, which is responsible for the distribution of context

reasoning. On the one hand, the operators enclose the functionality of the context reasoning algorithm. On the other hand, they constitute the smallest processing unit that can be deployed on a physical host.

In the following, we present first our system model and we introduce the core notion of our architecture, the operator graph, before we present the architecture of the system and we discuss the related work in context management systems.

2.1 System Model

Our system model consists of a *physical network model* that represents the physical interconnected network of physical nodes hosting the reasoning tasks and an *execution model* representing the service functionality to be executed on the physical hosts.

In particular, we assume a network of physical nodes that are spread over a wide geographical area and are capable of hosting reasoning tasks. Each physical node has different specifications in terms of computing capacity and is placed in certain location in the network, thus inducing different network latency depending on the node to communicate. Therefore the execution of a reasoning task may differ in terms of communication and processing delay depending on the physical node that hosts the task. To this end, the selection of the physical hosts that will execute the reasoning tasks has a strong impact on the performance of the system in terms of the network load and end-to-end delay.

In our execution model, we assume that each reasoning task can be represented by an *operator graph*, which is a core abstraction of our proposed system architecture. Initially, the operator graph was introduced for distributed data stream processing to model a stream processing task as an interconnected graph of traditional relational operators such as merge, join, and select. However, this model can be adopted by other application domains, since the operator can represent an arbitrary processing task on its input

streams to generate an output stream as we explain in Section 2.4. Thus, the operator graph model provides a unified representation of the service specific models.

In more detail, the operator graph is a graph that constitutes an abstract representation of the various functionalities to be deployed, together with the description of their interdependency. In particular, the different functionalities are encapsulated into primitive processing units, which are called *operators*. The operators then act as black boxes which hide the functionality of the specific services. In addition, the edges of the operator graph denote information exchange between operators. Furthermore, additional information that is useful for placing the operators onto the physical network can be expressed by restrictions either on the operators (e.g., computational load, memory requirements) or on the edges of the graph (e.g., latency requirements, bandwidth consumption).

2.2 Situation Model

In our proposed system architecture, we use the operator graph model, presented in previous subsection, to represent a reasoning task that process observable context to detect situations. In that respect, the context reasoning task splits into basic processing units, each one representing a partial result of the complete reasoning task. The idea is to exploit this characteristic of combining partial results to generate higher level context, by assigning the partial reasoning tasks to different physical hosts in order to increase the performance of the system.

Therefore, we introduce here the context reasoning operators, which process observable context data to infer situations. Given the adopted operator graph model, several algorithms could be used to detect situations, as long as they follow the principle of combining partial results. As explained in the previous section, a situation is composed of multiple forms of elementary context and describes the combination of circumstances at a given moment, a state of

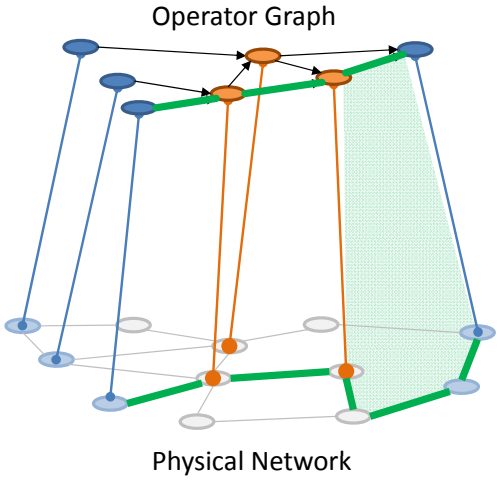


Figure 2.1: Mapping of operator graph to physical hosts.

affairs. Here we use a situation-centric approach, where each situation that can be detected by the system, is predefined by experts and stored as pre-knowledge of the system. For each situation one or more predefined situation recognition patterns, called *situation templates* that describe the relations between the various pieces of context, might exist. Situation templates are graphs consisting of nodes providing observable context and *operator* nodes. Operator nodes are method specific and describe the processing of the input data to derive high level context. In Fig. 2.2 we see an example of a situation template describing the situation "Traffic Jam". For the detection of this situation, we assume three kinds of observable context: sensor data about the number of cars in this part of the road, the average speed of the cars, and an internet text sensor which scans the WWW space to find context information related to the location of the situation. The unary operators connected with the external sources act as filters, which allow only the data within a range to pass to the next operator. Then Bayesian operators are applied to compute the probability of the (sub-)situations as described in [83].

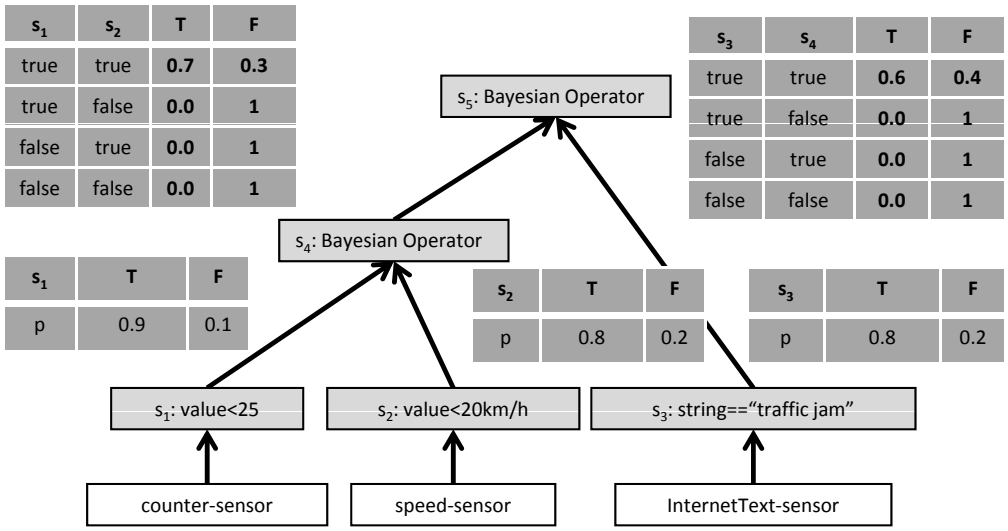


Figure 2.2: Situation Template: "Traffic Jam"

In order to calculate the probability of a (sub-)situation, we need to know the values of the so called *Contribution Probability Tables* (CPT). CPTs are not predefined, but situation template might include some initial values, as shown in Fig.1, that later will be changed by a learning process.

Fig.2.1 shows an example of mapping an operator graph -which corresponds to a situation template in our model- onto physical hosts according to our system model. In that respect, the operator graph acts as a *logical plan* by describing the operators and their interdependencies, while the overlay network of operators that is built after the mapping of the operators onto physical hosts as shown in Fig.2.1 represent the *physical plan*, since it assigns the reasoning operators onto physical hosts. Hence, it becomes challenging, given a logical plan and representation of the physical network, to find an optimal physical plan with respect to different optimization criteria e.g., network load or latency. Note that typically in an operator graph, the data sources and sinks are pinned, i.e., they are bound in specific physical hosts in the network. Therefore, the problem of converting a logical plan to a physical one, is mainly associated with the placement of the unpinned (reasoning) opera-

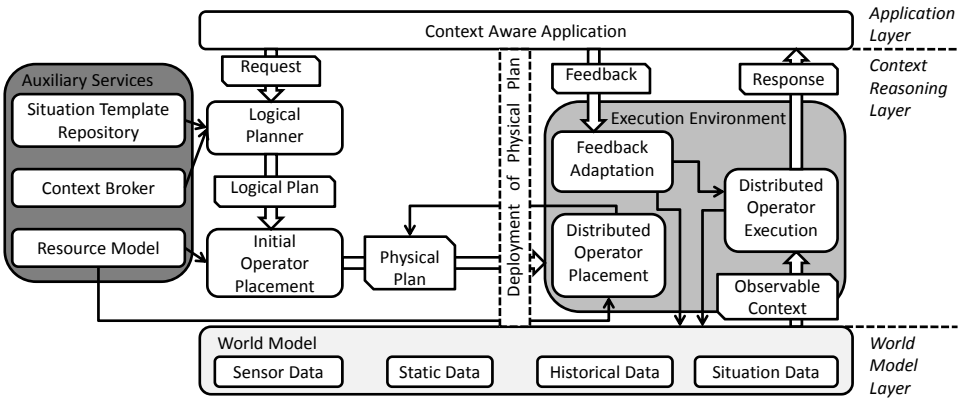


Figure 2.3: System Architecture

tors onto physical hosts. Later on, in Chapter 3, we define in a more formal way the operator placement problem and we describe different methods to solve this problem.

2.3 System Architecture

We now present our system architecture for distributed context reasoning. Fig. 2.3 shows the components and the interfaces of the distributed reasoning system, which belong to three different layers: *World Model Layer*, *Context Reasoning Layer*, and *Application Layer*. In the basic layer, the World Model provides the observable context to the situation detection components, which constitute the second layer that processes the observable context to derive high level context in an efficient way. The context-aware application lies on the top layer, representing the user that interacts with the system either to query for situations of certain objects or locations, or subscribing for events on detected situations. In addition to these basic parts, the auxiliary services support the core components by providing additional information to situation detection components.

The system operates in two distinct phases: the *Initialization Phase* and the *Execution Phase*. During the initialization, the system creates a query plan,

which describes a reasoning task by an operator graph with pinned context sources and sinks. Then an optimization step takes place, which maps the operator graph to an overlay network, where the free operators are placed to physical nodes such that an optimization goal is achieved. The operator graph is then deployed on the physical network and the system enters the execution phase. During the execution phase the reasoning task is executed in a distributed way on the physical network while the system continuously optimizes the mapping of the operator graph by adapting the overlay operator network to the current network condition. Next we describe in detail the core components of the architecture:

Query Planner. The Query Planner receives the user specifications and it retrieves the corresponding situation template from the *Situation Template Repository*, which stores all the available situation templates. The user specifications include the definition of the detectable situation as well QoC (Quality of Context) and QoS (Quality of Service) requirements of the user.

After the retrieval of the situation template, the Query Planner contacts the *Context Broker* [64] to discover the context sources needed to perform the reasoning task. For instance, in the traffic jam scenario, it might ask for all camera sensors at a certain road or the context servers providing information about the average speed of cars on this road. In general, the Context Broker can be realized as a distributed lookup service for context sources, where each source is described by the kind of data it provides, the quality of the provided data and the spatial area covered by the data. Finally, the Query Planner encapsulates each partial reasoning task in an operator, as specified in the situation template, and pins the sources and the application to their corresponding physical hosts in the network. The result of this procedure is an operator graph, which contains pinned (sources, application) and unpinned operators. This operator graph acts as an interface between the Query Planner and the initial placement component.

Initial Operator Placement. The Initial Operator Placement assigns the unpinned operators of the operator graph to physical hosts according to

defined optimization criteria. To achieve this goal, it executes an operator placement algorithm in a centralized way. In particular, the initial placement component first contacts the *Resource Model* to get the information about the physical nodes and links that represent the available resources in the physical network such as latency, available bandwidth, or load. The Resource Model is dependent on the placement algorithm and can be realized as a distributed lookup service. After retrieving information about the current network condition, the initial placement should find a mapping of the unpinned operators to physical hosts which optimizes for a certain criterion. Usual criteria for placement optimizations are network usage, latency, and load [8,63,84,92]. In Chapter 3, we present different placement algorithms that target different optimization goals and we discuss other existing approaches for the placement of operators onto physical hosts. The output of the placement algorithm is an overlay operator network, which extends the operator graph with the additional information of the physical mapping of the operators. Then the operator graph is finally deployed on the physical network and the system enters the execution phase.

Distributed Operator Execution. After the deployment of the physical plan, it starts the distributed execution of the operators, which realizes the reasoning task in a distributed way. If the user has subscribed for certain situations, this task is executed permanently and the user is notified of new situations when they are detected. In particular the distributed operator execution receives the context data from the selected sources of the World Model, performs the reasoning task and then notifies the application. Furthermore it also writes the result of the context reasoning back to the World Model. As we have already mentioned, the situation is a part of the World Model and therefore its current status is to be updated. This approach also allows for the storage of historic situations.

Distributed Operator Placement. Since the network conditions might change during the Execution Phase, the initial placement might not fulfill at some point in time its optimization goal anymore. The distributed operator

placement service is responsible for the adaptation of the operator placement to the current network conditions. Here, the operator placement is done in a distributed way and it modifies, if necessary, a part of the physical plan. In other words, when the distributed placement algorithm finds a better placement for an operator, it initiates the migration of this operator to another physical host by modifying this part of the physical plan. Then the execution environment is responsible for the deployment of the new physical plan. This process is an event-driven process, which is triggered by changes of the network conditions. Most of the existing placement algorithms provide distributed placement strategies that adapt the operator placement during the execution of the operator graph based on local information. For instance in [92] we proposed a distributed version of our placement algorithm optimizing for network usage.

Feedback Adaptation. The user can send feedback to the system about the occurrence of the detected situation in the real world (e.g. false positives/negatives). The user feedback is used by the *Feedback Adaptation* to improve the quality of the situation detection. In particular the feedback adaptation component is responsible for the re-configuration of the operators during the distributed operator execution. The operator configuration is dependent on the reasoning algorithm. For instance, in case of the Bayesian Networks, the algorithm proposed in [120] can be used to calculate the new values of the CPTs.

2.4 Overview of existing approaches and systems

In this section, we discuss related work in the field of context management systems but also in the related fields of Complex Event Processing (CEP) and Data Stream Management Systems (DSMS). As we analyse later, these different research communities have developed systems that share some common goals and aspects. In particular, from a network viewpoint, context management systems share common characteristics with CEP and DSMS systems,

since they all require the timely processing of data flows from a set of sources dispersed over the network to several sinks. In that respect, Gucola et al. [74] have tried to analyse the commonalities and differences between complex event processing and data stream processing. In this work, they introduce the concept of information flow processing (IFP), which aims to provide an abstraction model that applies for both CEP and DSMS systems. This model could serve also as a baseline to discuss the common characteristics between IFP and context management systems.

The following section gives an overview of state-of-the art approaches in context management systems. Then, we discuss CEP and DSMS systems as IFP systems and we analyse their differences and commonalities with the presented context management systems.

2.4.1 Context management systems

In the last years, researchers have developed several context management systems proposing different architectures depending on the target applications. Although existing systems support distributed application scenarios, they address the problem of the scalability and efficiency of context data processing in a distributed environment partially. Context reasoning is usually considered independently on the strategies used for distributed context processing, which might lead to inflexible models that cannot exploit the distributed nature of context data. In our architecture, we address the problem of distributed context reasoning as a whole. Therefore, we propose a model that provides the interface between the context reasoning methods and the distribution algorithms and enables the efficient distributed context reasoning. We see now in more detail how our system differs compared to existing context aware systems.

Context management systems may vary according to the adopted context abstraction and the respective context model. Other differentiation criteria refer to the architectural design of the context management systems and

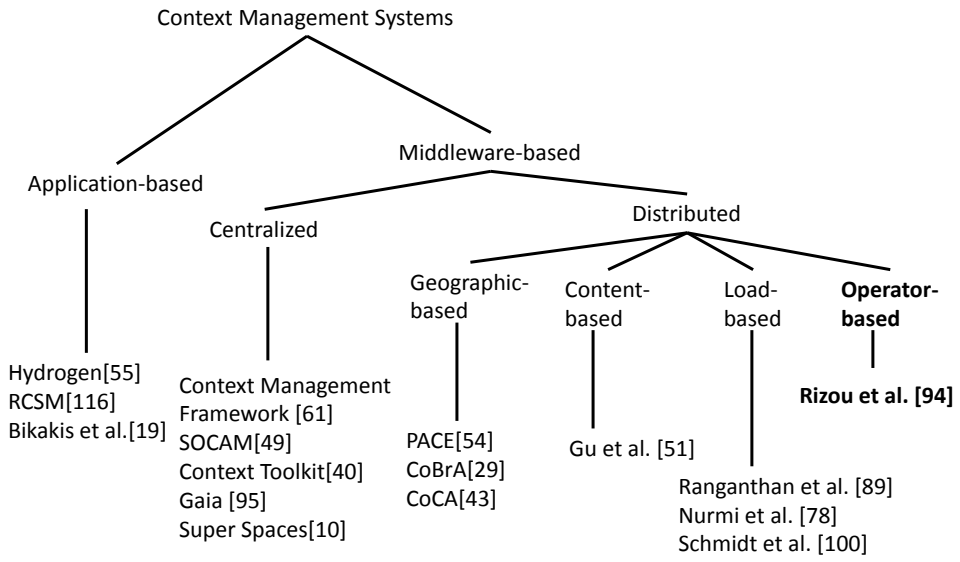


Figure 2.4: Context aware systems classification

the respective system types. Since the main contribution of our proposed system, is the flexible execution of context reasoning tasks to support large-scale scenarios and address the geographical dispersion of the context sources and sinks, our analysis will cover mainly the architectural design principles of the context management systems.

Context management systems typically consist of the context acquisition layer being comprised by the context sources and the context consumption layer, which is realized through the context consumers, i.e., sinks. Context processing could imply an additional optional layer representing the middleware, which is responsible for processing the context and deliver it to the context consumers. We classify the context management systems that do not use any middleware infrastructure as application-based systems, since they rely solely on the context processing on the application side (Figure 2.4). Furthermore, we distinguish middleware infrastructure systems in two main categories according to the processing model they adopt. The simpler approach is the centralized architecture where a single central context server

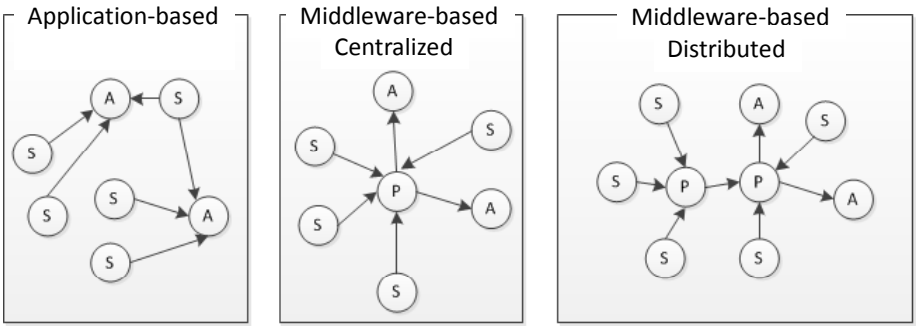


Figure 2.5: Processing Models

is used to collect, process and deliver the related context to the consumers. This approach has the obvious drawbacks of the centralized solutions, having a single point of failure.

An alternative solution is a distributed architecture, where multiple servers distributed in the network are available. Here we distinguish among the different distributed architectures proposed in the literature according to the rationale of the distribution they follow. The *geographic-based* distribution refers to the dispersion of multiple servers, where each one is responsible for a specific domain and cooperates to process and deliver the context to the consumers. Although this approach increases scalability compared to the centralized solution, it may still lead to poor system performance, since it does not allow the distribution of the reasoning tasks to multiple server according to specific optimization goals e.g., minimization of network load. The *semantic-based* distribution clusters peers according to the type of the queries they can answer, while the *load-based* distribution performs load balancing among multiple context servers. Although these approaches follow a distributed processing scheme, they target specific goals, e.g., reuse of partial results (semantic-based distribution) or load balancing among the server (load-based distribution). Our approach follows an *operator-based* distribution to allow the optimization of the distribution (operator placement) ac-

cording to different optimization goals. Thus, we propose a flexible distribution scheme which considers different optimization objectives according to the application characteristics and the system conditions.

Figure 2.4 provides an overview of the major context management systems and their position with respect to the adopted system architecture and its respective processing model. Next, we explain in detail the design of these systems and their relevance to our work.

2.4.1.1 Application-based Architectures

The context-aware systems classified in this category, do not use middleware infrastructure and rely solely on the processing performed on the application nodes.

Hydrogen [55] is one representative system of this category. Hydrogen follows a fully decentralized approach for mobile context sharing assuming a network of mobile nodes, willing to share context information. In this respect, Hydrogen model differentiates between the remote and the local context and it enables context sharing between nodes that lie in close proximity. In particular, Hydrogen architecture is a three-tier architecture consisting of an application, a management and an adaptor layer. All layers are realized in each context-aware device and enables the communication with other devices. Hydrogen framework covers small scale scenarios where context sources and sinks are located close to each other.

In [116], Reconfigurable Context-Sensitive Middleware (RCSM)is presented to facilitate the development and operation of real-time context-aware software in ubiquitous environments. RCSM is a context-sensitive middleware, which uses an object-oriented embedded middleware. By context-sensitive here it is meant the capability of the device to initiate and manage the ad-hoc communication with other devices based on the contextual of the local devices and its surroundings. RCSM categorizes the context according to its source, i.e., network context, device context, and user interaction context and

it provides specifications about the relationships of various pieces of context. Moreover, it enables the context-aware adaptation through the invocation of appropriate methods upon an event of a context match.

In [19] the authors have proposed a distributed reasoning method that is based on the Multi-Context Systems paradigm. According to their approach, ambient agents encode local context knowledge in rules (contexts) and exchange this information with other agents. In that respect, each peer in the network can evaluate the remote and local context to detect high level context changes. The reasoning method allows the resolution of conflicts according to the confidence of the context source.

All these systems assume a different system model with respect to ours, since they use limited computing capacity due to the energy constrained application hosts, which are typically mobile devices. Furthermore, in their system model, the communication among the applications and the sources is done in ad-hoc way. Therefore these systems cover usually small-scale scenarios, in contrast to our proposed architecture that targets large-scale scenarios.

2.4.1.2 Middleware Infrastructure Context Management Systems

In this category, we classify systems that do not rely only on the computing capabilities of the context sources and sinks, but also use middleware infrastructure such as context servers, that are responsible for collecting and processing context data. These architectures typically enable the collection of pieces of context from multiple dispersed sources. As shown in Figure 2.4, we distinguish between centralized and distributed middleware-based systems. In the next paragraphs, we present existing systems that belong to these two main subcategories of middleware-based systems.

2.4.1.2.1 Centralized In [61], a context aware framework is presented that facilitates the development of context-aware applications. The framework

provides an Application Programmer Interface (API) using an extensible ontology which defines the contexts that can be used from clients. The whole architecture of the framework is based on a blackboard-based approach. According to this communication paradigm, all context data update a blackboard that acts as a central knowledge base and is kept by the context server. In the proposed framework the context server is a mobile terminal having direct communication with other clients. Before the context data are communicated to the context server, a pre-processing step that convert raw measures captured from sensors into a representation defined in the context ontology. Therefore this approach is appropriate for small-scale scenarios, when the context sources and sinks are mobile devices that remain in close proximity.

The Context Toolkit [40] is one from the first attempts to provide a framework for the support of the design and development of context-aware applications. The Context Toolkit provides a conceptual framework that separates the acquisition and representation of context from the delivery and reaction to context changes by the context-aware application. A fundamental concept of the framework is the context widget which provides an abstraction that hides the complexity and variety of context acquisition mechanism, e.g., sensors, RFID, etc. On top of context widgets, the context interpreters are responsible for performing logical inference on the primitive (low-level) context to derive high level context. Furthermore, context aggregators are used to collect multiple pieces of context within the same software component and make it available to the context-aware applications. Context services are the responsible components for performing the reaction to the contextual changes. In that respect, they provide an abstraction, similar to the abstraction for context acquisition by the context widgets, for the adaptation of context-aware applications to contextual changes. Finally, another important component of the Context Toolkit is the discovery component which enables the discovery of the various context widgets, interpreters, aggregators and services in the framework. The implementation and actual architecture of the Context Toolkit relies on a centralized model, where a single central server processes

the multiple pieces of context. Although the conceptual framework could be extended to provide a federation of interpreters, aggregators and discoverers, Context Toolkit does not tackle the problem of optimally distributing the workload among several context interpreters.

The Service-Oriented Context-Aware Middleware (SOCAM) [49] is a middleware that facilitates the development of context-aware applications. The proposed middleware aims to convert physical spaces to semantic spaces, where context can be exchanged and used to adapt the behaviour of the systems to changes of the environment. In more detail, the system architecture is comprised by the Context Providers, Context Interpreter, Context Database, Service Location Service and Context-aware Mobile Services. The overall approach is based on a set of distributed context providers which communicate with a central server (context interpreter) that performs the context reasoning and delivers its output to the mobile clients (context-aware mobile services). Context representation and sharing is achieved through the use of ontologies. SOCAM architecture follows a centralized approach and therefore it cannot support large-scale scenarios.

Gaia [95] is a middleware solution that enables the management of context aware applications. Gaia introduces the concept of Active Space that represent a small-scale physical space, e.g., room that is controlled by a context management entity. In particular, the Gaia architecture consists of three major components: the Gaia Kernel, the Gaia Application Framework, and the Applications. The Gaia Kernel is responsible for the management and deployment of distributed objects and basic services that are used by all applications. Gaia Appplication Framework provides a set of component building blocks that support the development of context-aware applications and address mobility and dynamism. The applications provide the actual functionality of the context-aware applications converts a physical space into an Active Space. Gaia supports the development and deployment of context-aware applications in small scale and therefore it relies on a centralized context server. To address the problem of scalability, authors propose the construction of

the SuperSpaces [10] that are supersets of Active Spaces. For an instance, a building could be a Super Space of multiple Active Spaces rooms. To address scalability the authors propose the use of an additional interaction layer that interconnects the Active Spaces. The communication among the basic context management entities (Active Spaces)could be realized through a recursive or a peer-to-peer interaction. Nevertheless, this work does not provide an insight on the actual distribution of reasoning tasks as it is based on a partitioned control of the global environment.

2.4.1.2.2 Distributed Closer to our work are approaches that adopt a distributed architecture, assuming multiple context servers. Existing approaches, such as [29, 43, 54] propose a geographical-based distribution of servers, where each server is responsible for a specific region e.g. a building.

For instance, Chen et al. [29] proposed the Context Broker Architecture (CoBrA) as a framework to build smart environments. According to this approach, a central server called Context Broker is used to collect context data and derive high level context. The context consumers (clients) are subscribed to context brokers so that they get notified about the detection of high level contextual changes. Context Broker has three main components: the CoBrA Ontology which defines the context vocabulary for sharing context knowledge, the CoBrA resoning engine which performs the actual context reasoning to derive high level context and the Module for Privacy Protection (MoPP) which uses a policy language that enable users to define privacy protection rules according to which the permission to share a user's contextual information is decided. From an architectural viewpoint, CoBrA addresses large-scale scenarios, through the collaboration of multiple context brokers, distributed over the network, forming a broker federation layer.

In [43] a Collaborative Context-Aware (CoCA) service platform is presented to enable the development and operation of context-aware applications. The platform consists of four major building blocks: the interface, the data source, the core service and the supplementary service. The in-

terface manager manages the user interface and the interface of the CoCA platform to application-specific modules. Data source represents the group of components that provide context data to the core service. The context data are represented according to the Generic Context Management (GCoM) model which uses generic as well as domain-specific ontologies for knowledge representation. The core service collects the low-level data from the data source and performs the reasoning tasks to derive high level context. Therefore it uses a RAID-Action engine (Reasoning, Aggregation, Interpretation, Decision and Action engine) that processes the low level context by aggregating partial context information or reasoning over it to detect high level contextual changes. Finally, the supplementary service includes components for knowledge discovery and collaboration services. The collaboration manager supports peer-to-peer negotiation and communication protocols among devices to assist RAID process. To this end, CoCA platform supports the participation of multiple servers in the reasoning process due to the collaborative peer-to-peer communication among devices, such as PDAs or PCs in the neighbourhood.

Henricksen et al. [54] have also motivated the need for middleware in context-aware systems. In their work, they provide an overview of state-of-the-art middleware in context-aware systems and describe their proposed solution, the so-called PACE (Pervasive, Autonomic, Context-aware Environments) platform. The authors present a set of requirements for middleware including mobility, security, scalability, and ease of deployment. The main components of the PACE platform is the context management module, which handles the context and the preference management component, which tailors the decision-support to the different context-aware applications. The authors propose the use of a distributed context management layer consisting of multiple context servers, collaborating to efficiently perform reasoning tasks. However, similar to [29, 43], they consider the partitioning of the network to different servers and they do not tackle the problem of efficiently distributing the reasoning tasks in multiple servers.

Gu et al. [51] proposed a content-based distribution, where the reasoning tasks are distributed to the servers according to the context they refer to. In particular, the authors proposed a protocol for exchanging messages about context information which enables the performance of reasoning in a distributed fashion. Their system model is based on an overlay network where the peers are grouped in semantic clusters according to the type of the queries that they can answer, expressed in first-order logic. In our architecture, we cope with uncertain data that need more sophisticated reasoning methods and furthermore we assume large-scale overlay networks, where the distribution of the reasoning task should be done automatically according to different optimization criteria.

Finally other existing approaches, such as [78,89,100] proposing distributed solutions for context management, they aim to distribute the workload in multiple servers targeting a load-based distribution. In more detail, Ranganathan et al. [89] developed a middleware infrastructure which is based on distributed context servers called context synthesizers. The context synthesizers are spread in the network and support different reasoning methods. This approach distributes the computational load among multiple context servers. However it does not allow for the distributed execution of a reasoning task on multiple servers, and as a consequence it may lead to poor network and system performance compared to the optimized distributed execution proposed by our system architecture. Nurmi et al. [78] present a distributed agent-based architecture for distributing the reasoning process. In this model devices perform simple context reasoning and send their results to a remote server for more advanced context reasoning such as classification that requires more powerful computational capabilities. Although this work enables distributed context reasoning, it lacks a strategy about how the reasoning task is distributed in the network. Another approach [100] uses distributed Bayesian Networks and proposes a placement algorithm that clusters the nodes of a Bayesian network to reduce the communication overhead. Although this approach is close to our work, it still only provides a solution

to a method specific problem. Our goal is to create a generic formalization, where different reasoning algorithms can be distributed by different placement algorithms.

From the overview of existing distributed context management systems, it is evident that existing approaches focus on specific distribution aspects using multiple servers to perform load balancing or increase the reuse of partial results. In that respect, existing systems lack the flexibility to adjust the distribution of reasoning tasks such that resources are used efficiently and application-defined constraints are respected. Our approach fills this gap by proposing an operator-based distribution, where the reasoning task is decomposed in primitive subtasks that can be placed onto physical hosts according to different optimization goals, such as network usage, latency, throughput etc. The proposed operator graph model allows the consideration of different optimization goals based on the application characteristics and the system condition. For instance, communication-intensive applications that put heavy load on the network could be optimized for reducing the amount of data communicated in the network. To this end, our proposed architecture, which decomposes the reasoning tasks into a graph of reasoning operators that can be flexibly mapped onto the physical network enables the efficient processing of context data in large-scale scenarios.

2.4.2 Information Flow Processing

Relevant to context-aware systems are Complex-Event Processing (CEP) systems and Data Stream Management Systems (DSMS). On the one hand, data stream management systems have been an evolution of the traditional database management systems aiming at handling continuous data streams without first storing data into a database. On the other hand, complex event processing systems have their roots in the traditional Pub/Sub (Publish/Subscribe) systems, which aim to efficiently disseminate information from a group of publishers (data sources) to a group of subscribers (data sinks).

Cugola et al. [74] have introduced the term Information Flow Processing (IFP) to collectively refer to CEP and DSMS systems, since they share a common goal, namely, the processing of continuous flows of information units. In their work, they present an overview of existing IFP systems and they discuss their commonalities and their differences. As it is analysed in more detail in their work, although these systems share some common aspects, they differ also in several ways, e.g., in the data model, the query language or the processing model. In the next paragraphs, we provide a short overview of the goals and the execution models used in DSMS and CEP systems and then we discuss their architectural models based on the analysis presented in [74] and their relation to context management systems.

2.4.2.1 Data Stream Management Systems

Traditional Database Management systems (DBMS) are passive, in the sense that they retrieve data only when they are triggered by the application. Realizing the limitation of this model, to react autonomously upon events, the database community has introduced the Active Databases, which are capable to react upon the detection of predefined situations. More specifically, in active databases the rules are composed by three different parts, namely, Event, Condition and Action (ECA). Events could be either internal, e.g., the insertion of a tuple, or external events, e.g., clock triggers or external sensors. Examples of systems classified in active databases are the following HiPAC [38], Ode [47], Snoop [26].

Although active databases when linked to external sources of events (e.g., sensors) are closer to the IFP model, they have the fundamental difference that they rely on persistent storage, similar to the traditional DBMS. Therefore, distributed stream management systems were introduced to enable the real-time processing of unbounded data streams. In DSMS, no assumption can be made on the data arrival order, and data streams are processed on the time of arrival due to size and time constraints. In DSMS, queries are

typically continuous, i.e., they are continuously updated as the data streams arrive. The continuous queries can either be executed periodically or continuously whenever a new data stream item arrives.

The hierarchical network of *operators*, forming a so-called *aggregation tree* [72], which corresponds to an acyclic tree-based operator graph, has been widely accepted by the database community due to its conceptual simplicity and its applicability in practical scenarios (e.g., aggregation trees in wireless sensor networks) [45]. Typical data stream operators can implement either *algebraic* queries, meaning that they can keep the distribution properties of the aggregation tree, i.e., by communicating partial results to their neighbours in the tree, they are able to compute an *exact* query answer; or they may be *holistic* queries, i.e., they require the centralized processing of all data in a root node, which keeps a global view on the data observed so far [17]. Examples of algebraic queries are Sum, Mean, Max queries, whereas examples of holistic queries are Median, Distinct Count, and Histogram queries. In that respect, the tree-based operator graph, adopted in our model, is used for evaluating algebraic queries but also for calculating approximations of holistic queries in DSMS.

It is worth mentioning that there is extensive work from the database community in the optimization of the logical plans to reduce the communication overhead during the in network processing of aggregation queries. Typical methods for *logical optimization* are the use of filters close to the data sources [45], that reduce the communication overhead on the aggregation tree; the calculation of so-called *summaries* (e.g. [46, 98]), which reduce the communication overhead by communicating a subset of data to the root node, while respecting quality guarantees for the query answer or as recently introduced in [82], the change of the data granularity based on the application quality requirements. In our approach, we focus on the *physical plan optimization*, i.e., we use the operator graph, as an abstraction in order to separate the problem of optimization of the physical plans by searching for optimal mappings of operators onto physical hosts.

Although the wide-spread acceptance of the operator graph model, there are multiple data stream processing research prototypes relying on centralized processing of data streams e.g., NiagaraCQ [30],OpenCQ [70], Tribeca [106], CQL [13], Stream [12], Aurora [3], Gigascope [33], Stream Mill [14]. Closer to our proposed architecture, examples of existing *distributed data stream processing engines* using in-network processing of data streams are Telegraph CQ [28], Borealis [2], Tag [72] and, NexusDS [32].

2.4.2.2 Complex Event Processing Systems

Complex event processing systems unlike DSMS, associate semantics on the detected events captured by the data sources. In that respect, the goal of CEP systems is mainly to detect complex event-patterns using sequencing and ordering relationships that they are not common in DBMS. Traditionally CEP has been based on the Pub/Sub (Publish/Subscribe) paradigm. In Pub/Sub systems users subscribe to get notifications from publishers (data sources) upon the detection of specific events. Typically Pub/Sub systems can either be topic-based, meaning that a user could subscribe on a topic or content-based if the subscribers could use complex event filters to define the content of the desired notifications. CEP could be seen as an extension of Pub/Sub systems that allow the subscription on complex, composite events i.e. correlated events following certain sequence patterns.

In terms of execution model, CEP applications form typically *multicast trees*, where sources (publishers) communicate events to a set of sinks (subscribers). For detecting composite events, in-network processing of events by *event correlators* is required to provide efficient event correlation.

Examples of CEP systems are Traditional PubSub [44], [75], Rapide [71], GEM [73], Padres [69], DistCED [86], CEDR [15], Cayuga [21], NextCEP [101], PB-CED [9], Raced [34], Amit [5], Sase+ [6, 53], Sase [112], Peex [58], Tesla/T-Rex [35,36]. Furthermore, one of the most popular commercial CEP systems is the Commercial System S [11,57,113]. Closer to our work, examples

of CEP systems allowing for the detection of composite events are Rapide [71], Padres [69], DistCED [86], GEM [73].

2.4.2.3 Comparison between DSMS, CEP and Context Management Systems

DSMS, CEP and context management systems share some common aspects but they also have fundamental differences. One fundamental distinction coming from the analysis and comparison between DSMS and CEP is that DSMS focus mainly on the efficient data processing and handle homogeneous flows of data, whereas CEP systems focus on event detection and handle typically heterogeneous flows of data (events) that are combined to detect complex events. One step further, context management systems are closer to CEP systems in the sense that they use context reasoning methods to enable the complex correlation of events into meaningful situations.

Given the focus of our work on the distributed architecture of context management systems, we discuss here the architectural models used by DSMS and CEP systems based on [74] that provides a comprehensive overview of IFP systems with respect to their architecture. In their work, they categorize the IFP engines in *centralized* and *distributed* and they further distinguish distributed IFP engines to *clustered* and *networked* engines. Clustered IFP engines use a cluster of strongly connected machines that belong usually to the same administrative domain, while networked IFP engines assume physical hosts distributed in a Wide Area Network (WAN) that are connected typically by Internet links. Note that for networked architectures, the minimization of the network usage becomes critical, since physical hosts run typically in different administrative domains.

According to [74], the most common architecture, especially in Active databases and DSMS is the centralized solution that uses a single server which collects and processes all the data centrally. The clustered solution is followed by some commercial systems e.g. Aleri, Coral8 [74], IBM System

S [11, 57, 113] and a few DSMS systems (Telegraph CQ [28], Aurora [3]). Finally the networked architecture is applied for some CEP systems (GEM [73], Padres [69], DistCED [86]). Our architecture could be also classified in the networked architecture, since we assume a network of physical hosts dispersed in a wide area network.

As a conclusion, although CEP, DSMS and context management systems, implement different operator semantics to define application-specific streaming tasks, they all share a common representation of an overlay network of operators, processing cooperatively a distributed stream processing task. Thus, the operator graph model could be seen as a unified model that introduces the problem of physical plan optimization, which seeks for optimal mapping of operators on physical hosts in the network. In that respect, our proposed architecture uses the operator graph model to map the problem of efficient processing of context data into an operator placement problem, which is known from the database community. By using the operator abstraction, we allow the use of existing operator placement algorithms initially designed for other application domains, such as DSMS and CEP to tackle the problem of distributed context reasoning in context management systems.

Although the operator placement problem is a fundamental common problem among CEP, DSMS and context management systems, different assumptions regarding the underlying physical network (e.g., LAN or WAN), may lead to different optimization objectives depending on the system model and the target application. To this end, in the next chapter, we formally consider three variations of the operator placement problem that may apply to CEP, DSMS as well as context management systems as long as they assume a networked architecture, where physical hosts are distributed in a WAN. In that respect, the main objective of the placement algorithms presented in the next Chapter, is the minimization of the network load, which applies in large-scale scenarios. Moreover, we consider application-defined end-to-end latency requirements targeting different applications depending on the size of the communicated data units in the overlay network of operators.

3 Operator Placement Algorithms

In this chapter, we discuss the operator placement problem in three different variations that consider different constraints, and we present operator placement algorithms that solve the resulting problems. Our main concern is to provide scalable operator placement algorithms that can be used in a distributed setting. As briefly introduced in Chapter 2, the operator placement problem seeks for an optimal mapping of operators onto physical nodes to fulfill application constraints and minimize resource costs. The operator placement affects QoS and efficiency since different placements could lead to different response times for the application or a different consumption of network resources.

In particular, here we consider as optimization goal the minimization of network usage that is formally defined as the bandwidth-delay product of inter operator data streams of an operator graph. The network usage metric quantifies the network load put onto the system since it is an indicator of the network traffic. Imagine, for instance, a large-scale camera network that processes images from distributed data sources to detect activities inside buildings or across road segments. In this use case, large chunks of data are to be transmitted from the sources in order to get processed by operators and finally delivered to the application. For such applications, the amount of data that is in transit in the network can be a hindrance for the scalability of the system since it could lead to traffic congestion and bottlenecks.

To illustrate how placement decisions could affect the induced network load, we present a simple example in Figure 3.1 that shows two different placements of the same operator graph. For each placement we calculate the bandwidth-delay product. Since latency depends on the communication

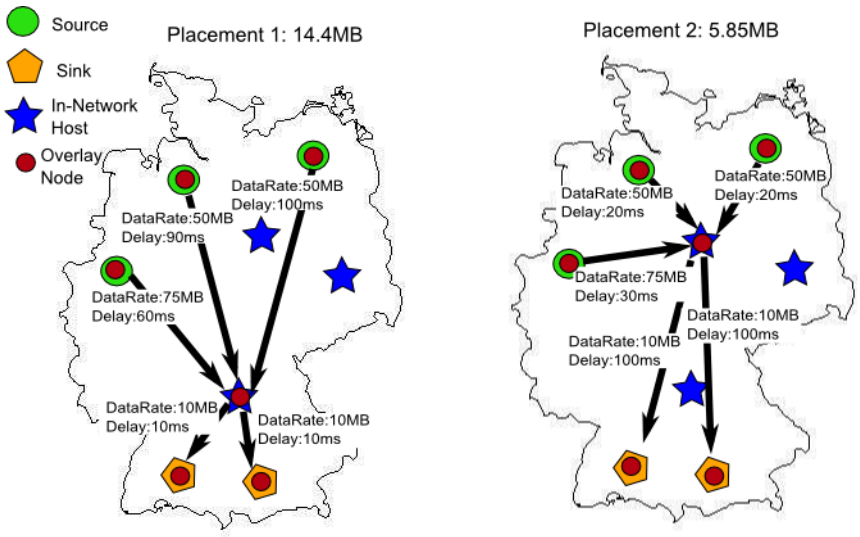


Figure 3.1: Two different placements with respective resulting network usage.

link between the physical hosts, although the data rates remain the same, the overall bandwidth-delay product changes significantly from 14.4 MB to 5.85 MB between the two placements, i.e. 8.55 MB could be saved. Thus, reducing network usage could relieve the system from network load. Thus, the minimization of the network load leads to the avoidance of network congestion and, therefore, contributes to the scalability of the system.

Although the minimization of bandwidth-delay product minimizes indirectly also the network latency, which is an important factor of the response time of the system, it does not directly consider any constraint on the end-to-end latency between the data sources and sinks of an operator graph. End-to-end latency is an important application-level quality of service metric for delay-sensitive applications since it significantly influences the latency of detecting situations. Therefore, the definition of end-to-end latency constraints is an important requirement for such delay-sensitive applications. A guaranteed maximum end-to-end delay is critical for instance for control systems based on a global network of widely dispersed sensors that have to react in a

Problem	**Algorithms**
Network Usage Optimization	MOPA
Network Delay Constrained Optimization	MOPA-LMAX
Processing & Network Delay Constrained Optimization	MOPA-LPMAX

Table 3.1: Overview of placement problems and algorithms

timely manner to sensor information to control physical processes. In order to fulfill delay constraints for the application, we have formulated a *constrained optimization problem*, which optimizes network usage, while also considering constraints on the network latency imposed by the application. In particular, in this constrained optimization problem, we consider applications, where the network latency is the dominant factors of the end-to-end latency. In that respect, in our approach the goal of this constrained optimization problem is to keep the maximum network latency between a data source and a sink of an operator graph under a certain threshold.

Then, we formulate another constrained optimization problem, targeting applications where the *processing delay* contributes significantly to the end-to-end latency. For instance, in the case of the large-scale camera network, data units are images which induce significant transmission and processing delay. Fulfilling latency constraints for this type of applications requires a more complex system model which includes processing and transmission delays. To this end, we formulate another constrained optimization problem and present our approach, which first optimizes for network usage and then applies a constraint satisfaction algorithm that fulfils the end-to-end latency constraints.

The remainder of this chapter is structured into three different sections, where we describe the three different placement problems and corresponding solutions. Table 3.1 shows an overview of the placement problems and the corresponding algorithms presented in this chapter. In particular, in

Section 3.1, we formulate the *network usage optimization problem* and we present a distributed placement algorithm called Multi-operator Placement Algorithm (MOPA), which solves the optimization problem in a distributed way by letting the operators to get placed autonomously according to their local view. Although our initial goal is to provide a scalable distributed algorithm, we also present in this section an *integer linear program* (ILP) that solves the network usage optimization problem in a centralized way. This centralized solution will be used as reference for measuring the performance of the proposed distributed algorithm in Chapter 4. Section 3.2 introduces a *constrained optimization problem*, which applies a maximum threshold in the *network delay* experienced by the application. An algorithm called MOPA-LMAX is being introduced that solves the presented constrained optimization problem, by processing the solution provided by the MOPA algorithm such that the latency constraint is fulfilled. Then, in Section 3.3 we present the MOPA-LPMAX algorithm, which also tries to fulfill an application-defined latency constraint, by considering both *network and processing delays*. MOPA-LPMAX uses a heuristic approach for the selection of candidate nodes to host the operators. In Section 3.3, we present different heuristic approaches for the candidate selection, which we are going to evaluate in Chapter 4.

Each of the three sections, presenting the different placement problems has the following structure: First, we present the system model based on which the optimization problems are formulated. Since the constrained optimization problems are extensions of the initial unconstrained optimization problem, we initially present in Section 3.1.1 a basic system model, and then we extend this basic model in the next sections and in particular in 3.2.1 and in 3.3.1 in order to introduce the two constrained optimization problems. Given the specific system model, we then formally introduce the corresponding problem, before we describe the details of the proposed algorithms.

3.1 Network Usage Optimization

In this section, we present an algorithm that minimizes the bandwidth-delay product of the inter-operator streams of an operator graph. By minimizing the network usage, we put less load onto the network links and thus we slow down network congestion. Therefore, the optimization of this metric contributes to the scalability of the system. In particular, this optimization is important for communication intensive applications producing big data volumes that need to be transferred across the network and traverse possibly multiple network links.

We propose a distributed algorithm to solve the so-called Multi-Operator Placement (MOP) problem, which formally describes the optimal placement of all operators of an operator graph [92]. The basic idea of this approach is that each operator finds its optimal placement by solving a *local optimization problem*. The sum of these Single-Operator Placement (SOP) problems is then the solution of the MOP problem. To facilitate the distributed solution of these placement problems, we use a heuristic solution based on a continuous search space called latency space, which is used to model delays between nodes in the underlay network. Assuming there exists a virtual node at every position in the latency space, we propose a distributed algorithm, where each operator autonomously finds an optimal virtual node in the latency space. In a second step, the selected virtual nodes are mapped to the available physical nodes in the latency space.

According to our general goal to use communication resources efficiently, we optimized the distributed operator placement algorithm for low communication overhead by reducing the number of management messages and operator migrations. We will provide a proof on the optimality of the global solution with respect to the local solutions, and show by experiments in Chapter 4, that this continuous solution approximates the discrete solution very well. Beyond the proposed distributed algorithm, we present also at the end of this section an integer linear program that solves the multi-operator placement

problem in a centralized way assuming global knowledge of the network and system conditions.

Next, we present the basic system model upon which we formulate the unconstrained optimization problem before we present our proposed distributed placement algorithm.

3.1.1 System Model

As a prerequisite of the formal problem formulation, we first introduce the system model together with assumptions and a formal notation. Table 3.2 summarizes the basic definitions of the system model, introduced in this subsection. Our system model consists of three main parts. A part of the definitions provided in this paragraph describes the *physical network*, i.e. the network of physical nodes that are capable of hosting the operators. Another set of definitions relates to the *latency space* an abstraction of the physical network proposed in the literature [84]. Finally, the third group of definitions describes the stream processing task, which is formally represented by an *operator graph*.

In more detail, we consider a physical network graph $\mathcal{H} = (\mathcal{V}, \mathcal{E}, l)$ consisting of a set of *physical* nodes $\mathcal{V}$ which are capable of hosting operators needed for stream processing. These nodes are connected through a set of communication links $\mathcal{E}$, such as the Internet links, allowing nodes to communicate with each other directly. Similar to [84], we assume a so-called *latency space*, which is an n-dimensional Cartesian space, where every node ν has a position $\vec{x}_\nu \in \mathbb{R}^{\text{dim}}$ such that the Cartesian distance $d(\overline{\nu_i \nu_j}) = |\vec{x}_{\nu_i} - \vec{x}_{\nu_j}|$ between any pair of nodes ν_i, ν_j corresponds to the propagation delay $l(\overline{\nu_i \nu_j})$ between these nodes[1]. The latency space can be constructed efficiently in a distributed manner using delay measurements between physical nodes and an algorithm for calculating network coordinates such as the Vivaldi algorithm

[1]The concrete dimensionality was investigated in the original paper, where the latency space was introduced [37] and is out of the scope of this dissertation.

Physical Network Model	
$\mathcal{H}$	Underlay network of physical hosts
$\mathcal{V}$	Set of physical nodes hosts
$\nu \in \mathcal{V}$	Physical host in the underlay
$\mathcal{E}$	Set of (Internet) links between hosts
$\overline{\nu_i \nu_j} \in \mathcal{E}$	(Internet) link between hosts
$l(\overline{\nu_i \nu_j})$	Latency between hosts
Latency Space Model	
$\vec{x}_\nu$	Position (coordinates) in the latency space
$d(\overline{\nu_i \nu_j})$	Cartesian distance between nodes ν_i, ν_j in the latency space
$d(\overline{\omega_i \omega_j})$	Cartesian distance between nodes ω_i, ω_j in the latency space
$C(\mathcal{V})$	Set of coordinates of the physical nodes $\mathcal{V}$
Operator Graph Model	
$\mathcal{G}$	Overlay network of operators
Ω	Set of operators
$\omega \in \Omega$	Operator of the overlay network
Ω_{pinned}	Set of operators placed on specific hosts
Ω_{free}	Set of operators that can be placed freely on a physical host
S	Set of pinned operators that generate data (sources)
A	Set of pinned operators that consume data (sinks)
$\mathcal{F}$	Set of links in the overlay network
$\mathcal{F}_\omega$	Set of in- and out-going links attached to operator ω
$\overline{\omega_i \omega_j} \in \mathcal{F}$	Link between ω_i and ω_j in the overlay network
$r\overline{\omega_i \omega_j}$	Data rate of the stream communicated over the link $\overline{\omega_i \omega_j}$

Table 3.2: System Model Notation

proposed by Dabek et al. [37]. To determine its position in the latency space, every node performs these calculations and provides this information to other nodes as described later. The set of coordinates of the physical nodes in $\mathcal{V}$ is denoted as $C(\mathcal{V})$. Note that the latency space is dynamic in the sense that the positions of physical nodes are continuously adapted depending on current delays. An overloaded path in the communication network leads to an increase of the delays between nodes using this path to communicate, and thus the distance between these nodes increases also. Since our placement algorithm dynamically adapts to changing node positions, the placement is adapted by choosing nodes that are close to each other, i.e., nodes whose communication paths are not overloaded.

In our execution model, a stream processing task is modelled as an *acyclic* directed graph of connected operators, called *operator graph*. As already introduced informally in Chapter 2, the set of the deployed operator graphs constitutes the *operator network*. Formally, a stream processing task is modelled as a tree-based operator graph $\mathcal{G} = \{\Omega, \mathcal{F}, r\}$ consisting of a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ of *operators* that are connected by a set $\mathcal{F} = \{\overline{\omega_1\omega_i}, \ldots, \overline{\omega_j\omega_n}\}$ of *links*. Operators, which perform any kind of stream processing operations, may have a number of incoming and outgoing streams. A link is an unidirectional communication relationship between a pair of operators. Link $\overline{\omega_i\omega_j} \in \Omega \times \Omega$ connects operators ω_i and ω_j, where the former produces a stream that is communicated to and consumed by the latter. $\mathcal{F}_\omega$ denotes the set of in- and out-going links attached to operator ω. In our system model, we assume unreliable communication protocol, such as UDP, between the physical nodes. An operator ω could be either pinned $\omega \in \Omega_{pinned}$, i.e., its mapping to physical node is given and *fixed*, or free $\omega \in \Omega_{free}$, in the sense that they could be *freely* assigned to any available node in $\mathcal{V}$. Pinned operators could be either *sources* or *sinks*. More formally, a subset of pinned operators $S \subset \Omega_{pinned}$ only have outgoing links (producers of data streams) and hence are called *sources*-for instance, sensors, while another subset $A \subset \Omega_{pinned}$ denotes the set of *sink* operators, which only have incoming links and typically

represent application entities. Finally, r denotes the inter-operator data rates in the overlay network, with $r(\overline{\omega_i\omega_j})$ specifying the data rate of the stream communicated over the link $\overline{\omega_i\omega_j}$.

Based on this system model, we formally define the optimization problem to be solved next.

3.1.2 Problem Statement

Our placement algorithm tries to minimize the network usage for each individual operator graph. As already discussed earlier, by optimizing the placement according to network usage metric, we increase scalability as the communication load generated by operator graphs is minimized.

Network usage is measured by the number of bytes that are in transit on the links of the operator graphs at a certain point in time. Formally, the network usage of link $\overline{\omega_i\omega_j}$ is defined by the bandwidth-delay product $r(\overline{\omega_i\omega_j})l(\overline{\omega_i\omega_j})$, where $r(\overline{\omega_i\omega_j})$ (according to Table 3.2) specifies the data rate of the stream communicated over that link, and $l(\overline{\omega_i\omega_j})$ is the delay of that link. In our system model, we use the euclidean distance in the latency space $d(\overline{\omega_i\omega_j}) = \sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2}$ to approximate the link delay $l(\overline{\omega_i\omega_j})$. Thus, the link delay in our model is defined by $d(\overline{\omega_i\omega_j}) = |\vec{x}_{\omega_i} - \vec{x}_{\omega_j}|$, where $\vec{x}_\omega$ denotes the position of the operator ω in the latency space that it is mapped (as we explain later) to the closest coordinate $\vec{x}_\nu$ of the physical node ν which hosts the operator.

To formalize our optimization problem, we first introduce the *Single-operator Placement (SOP) Problem*, which considers the optimal placement of a single unpinned operator, say ω, relative to the placement of its neighbours in the operator graph. For the SOP problem, we assume that the neighbours are pinned, and only ω is a free operator. This can be interpreted as a snapshot of the neighbour positions that ω is using to find its optimal placement relative to these current neighbour positions. The SOP optimization goal is to minimize the network usage of all the links connected to ω, i.e., the aggregated

bandwidth-delay product of the links in $\mathcal{F}_\omega$ (links connected to ω) is to be minimized. Equation 3.1 expresses the network usage $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$ associated with placement $\vec{x}_\omega$:

$$\begin{aligned}\mathcal{U}_{\text{local}}(\vec{x}_\omega) &= \sum_{\overline{\omega\omega_i}\in\mathcal{F}_\omega} r(\overline{\omega\omega_i})d(\overline{\omega\omega_i}) = \sum_{\overline{\omega\omega_i}\in\mathcal{F}_\omega} r(\overline{\omega\omega_i})|\vec{x}_\omega - \vec{x}_{\omega_i}| = \\ &= \sum_{\overline{\omega\omega_i}\in\mathcal{F}_\omega} r(\overline{\omega\omega_i})\sqrt{(x-x_i)^2+(y-y_i)^2+(z-z_i)^2}\end{aligned} \tag{3.1}$$

Given the above definition the SOP problem is formally defined by:

$$\begin{aligned}\min \mathcal{U}_{\text{local}}(\vec{x}_\omega) &= \min \sum_{\overline{\omega\omega_i}\in\mathcal{F}_\omega} r(\overline{\omega\omega_i})|\vec{x}_\omega - \vec{x}_{\omega_i}| \\ \text{variables } \vec{x}_\omega &\in \mathbb{R}^3 \text{ (continuous solution)} \\ \text{variables } \vec{x}_\omega &\in C(\mathcal{V}) \text{ (discrete solution)}\end{aligned} \tag{3.2}$$

The local SOP problem of placing a single operator optimally can be extended to the global *Multi-operator Placement (MOP)* problem, which seeks for the optimal placement of all unpinned operators of an operator graph $\mathcal{G} = \{\Omega, \mathcal{F}, r\}$. The goal is to minimize the overall network usage of $\mathcal{G}$. For a given placement $(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$, $\mathcal{G}$'s network usage $\mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is defined as follows:

$$\mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}) = \sum_{\overline{\omega_i\omega_j}\in\mathcal{F}} r(\overline{\omega_i\omega_j})d(\overline{\omega_i\omega_j}) \tag{3.3}$$

And the respective optimization problem is formulated as:

$$\begin{aligned}\min \mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}) &= \min \sum_{\overline{\omega_i\omega_j}\in\mathcal{F}} r(\overline{\omega_i\omega_j})d(\overline{\omega_i\omega_j}) \\ \text{variables } \vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n} &\in \mathbb{R}^3 \text{ (continuous solution)} \\ \text{variables } \vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n} &\in C(\mathcal{V}) \text{ (discrete solution)}\end{aligned} \tag{3.4}$$

This also minimizes the network usage for the entire operator network if all operator graphs in the operator network (set of all deployed operator graphs) have only one sink, i.e., sinks do not share common operators.

As shown in the equations above, both SOP and MOP problems can be solved for *physical* and *virtual* nodes. In the latter case, we assume that there exists a virtual node at every possible position in the continuous latency space, i.e., $\vec{x}_\omega \in \Re^3$, whereas in the former case, the operators can only be mapped to those positions in the latency space that are occupied by physical nodes, i.e., $\vec{x}_\omega \in C(\mathcal{V})$. Since the solution space can be either continuous $\Re^3$ or discrete $C(\mathcal{V})$, we distinguish between the continuous and discrete variant of the MOP and SOP problem.

3.1.3 Multi-operator Placement Algorithm (MOPA)

In this section, we propose a novel *distributed* placement algorithm approximating the optimal solution of the discrete MOP problem. Since our goal is to provide a distributed scalable placement algorithm, we propose here a heuristic approach that is based on the idea to solve first the corresponding *continuous* MOP problem and then map the selected virtual nodes to the available physical nodes to yield an approximation of the discrete MOP solution. In Chapter 4, we discuss the performance of our algorithm with respect to the optimal solution.

The continuous MOP problem can be solved in a distributed fashion by letting each unpinned operator autonomously solve the continuous SOP problem. In other words, each unpinned operator determines its optimal virtual node (i.e., its optimal position in the latency space) depending on the current virtual positions of its neighbouring operators. We prove in Subsection 3.1.3.3 that the collection of the continuous SOP solutions yields an optimal continuous MOP solution. The proposed distributed algorithm can be used for both the initial placement of an operator graph as well as for adapting the placement when delay or bandwidth conditions change significantly during

the execution of an operator graph. For the initial placement, we simply execute the algorithm in a centralized way; for the continuous adaptation, the algorithm is executed in a distributed fashion by the operators.

Algorithm 1 shows an overview of the steps/algorithm performed by each unpinned operator ω. First, a solution for the continuous SOP Problem is calculated, i.e., the position of the virtual node for ω is determined such that $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$ is minimal (line 1). For the algorithmic details of these calculations we refer to Section 3.1.3.1. In the next step, the selected virtual node is mapped to the closest available physical node ν(line 2) in the latency space. If ν_{new} differs from the current hosting node ν_{current}, the operator is migrated to the new physical node (lines 3–5), if the selected physical host is not overloaded after the deployment of the additional operator. Otherwise, the algorithm excludes this physical node from the search space to prevent bottlenecks and assigns the operator to the next nearest physical host. Typically, a number of iterations of the algorithm are required to approximate the optimal solution. To reduce the number of migrations, a *lazy migration* strategy can be applied. In such a case, migrations can be delayed for some iterations without affecting the final outcome of the algorithm.

For mapping virtual nodes to the available physical nodes, we use a nearest neighbour search mechanism. As stated in Subsection 3.1.1, we assume that the position of each physical node is known. Therefore, the nearest neighbour search can be realized using a distributed index as describe in [107]. For example, this index can be realized by the physical nodes forming a peer-to-peer network. The implementation of such a distributed index is beyond the scope of this dissertation, and we refer to [107] for more details.

After the initial placement, the algorithm is executed in an event-driven manner. It is triggered for operator ω in two cases: A neighbour operator informs ω that the neighbour's virtual node position or the data rate of a link connected to ω has changed. The first case occurs whenever a neighbouring operator calculates a new virtual node position when performing the algorithm. For detecting the second case, each unpinned operator measures the

Algorithm 1 Multiple Operator Placement Algorithm (MOPA)

Require: ω is placed at physical node ν_{current}
Require: Virtual coordinates of ω's neighboring operators
Require: Estimations of data rates of links in $\mathcal{F}_\omega$
Ensure: ω is placed on optimal physical node
1: find $\vec{x}_\omega$ such that $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$ is minimal
2: find closest non-overloaded physical node ν_{new} with $\vec{x}_{\nu_{\text{new}}}$ to $\vec{x}_\omega$
3: **if** $\nu_{\text{new}} \neq \nu_{\text{current}}$ **then**
4: migrate ω to ν_{new}
5: **end if**

data rate of each incoming and outgoing link using an exponential moving average. We explain in more detail, the adaptation mechanism in Subsection 3.1.3.4. For the initial placement we can estimate the data rate of each link according to the type of application or based on statistics gathered from previous deployments.

3.1.3.1 Single-operator Placement Algorithm

Here we describe in detail the subalgorithm of Algorithm 1 (line 1) which approximates the optimal continuous SOP solution in the continuous search space. The SOP problem corresponds to the well known Fermat-Weber Problem [27], which asks for the position $\vec{x}_\omega$ that minimizes $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$ (Equ. 3.2). It is known that there exists no closed formula for the calculation of the optimal solution of the Fermat-Weber problem unlike the mass centroid calculation that can be computed directly. Furthermore, the optimal solution cannot be defined exactly but only be approximated since it contains square roots that may be irrational numbers. There exist several approximation algorithms proposed in the literature that solves this problem. The most common one is the Weiszfeld method that implements a gradient method for this problem. However the Weiszfeld method may experience slow conver-

Algorithm 2 Single Operator Placement Algorithm

Require: Virtual node coordinates of ω's neighboring operators: $\{\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}\}$

Require: Data rates of links in $\mathcal{F}_\omega$: $\{r(\overline{\omega\omega_1}), \ldots, r(\overline{\omega\omega_n})\}$

Ensure: $\mathcal{U}(\vec{x}_\omega)$ is minimal

1: $\vec{x}_\omega \leftarrow \text{ManhattanApproximation}\{r_1\vec{x}_{\omega_1}, \ldots, r_n\vec{x}_{\omega_n}\}$
2: $\vec{x}_\omega \leftarrow \text{CheckDeadPoints}\{\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}\}$
3: $step \leftarrow \max\{|\vec{x}_{\omega_1} - \vec{x}_\omega)|, \ldots, |\vec{x}_{\omega_n} - \vec{x}_\omega)|\}$
4: **repeat**
5: $\quad\vec{f} \leftarrow \nabla\mathcal{U}(\vec{x}_\omega)$
6: $\quad$**if** $\mathcal{U}(\vec{x}_\omega + step \times u(\vec{f})) < \mathcal{U}(\vec{x}_\omega)$ **then**
7: $\quad\quad\delta \leftarrow \mathcal{U}(\vec{x}_\omega + step \times u(\vec{f})) - \mathcal{U}(\vec{x}_\omega)$
8: $\quad\quad\vec{x}_\omega \leftarrow \vec{x}_\omega + step \times u(\vec{f})$
9: $\quad$**else**
10: $\quad\quad step \leftarrow step/2$
11: $\quad$**end if**
12: **until** $\delta < \delta_t$

gence in the case that the solution is a *dead point*, i.e., a point where the derivative is not defined.

For our implementation, we have selected to use a simple approximation algorithm that speeds up the convergence of the algorithm and handles the dead points, followed by a gradient method with varying step. To calculate the gradient, $\nabla\mathcal{U}(\vec{x}_\omega) = \{\frac{\partial\mathcal{U}_{\text{local}}(x)}{\partial x}, \frac{\partial\mathcal{U}_{\text{local}}(y)}{\partial y}, \frac{\partial\mathcal{U}_{\text{local}}(z)}{\partial z}\}$ at the current position of operator ω we use the following equations[2]:

[2]For dead points, where $d(\overline{\omega\omega_i}) = 0$, we use a hyperbolic approximation $d^H(\overline{\omega\omega_i}) = \sqrt{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2 + \epsilon}$, where ϵ a small constant [119].

$$\frac{\partial \mathcal{U}_{\text{local}}}{\partial x} = \sum_{\overline{\omega\omega_i} \in \mathcal{F}_{\omega_i}} r(\overline{\omega\omega_i}) \frac{(x - x_i)}{d(\overline{\omega\omega_i})} \tag{3.5}$$

$$\frac{\partial \mathcal{U}_{\text{local}}}{\partial y} = \sum_{\overline{\omega\omega_i} \in \mathcal{F}_{\omega_i}} r(\overline{\omega\omega_i}) \frac{(y - y_i)}{d(\overline{\omega\omega_i})} \tag{3.6}$$

$$\frac{\partial \mathcal{U}_{\text{local}}}{\partial z} = \sum_{\overline{\omega\omega_i} \in \mathcal{F}_{\omega_i}} r(\overline{\omega\omega_i}) \frac{(z - z_i)}{d(\overline{\omega\omega_i})} \tag{3.7}$$

Algorithm 2 shows the gradient method used for searching the minimum of $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$. In each iteration, we first calculate the direction of the major flow $\vec{f}$, which corresponds to the gradient $\nabla\mathcal{U}(\vec{x}_\omega)$ at the current position of operator ω (line 5). Then, we move towards this direction, which is given by the unit vector $u(\vec{f})$, with a certain step length *step* until we reach the minimum (cf. Figure 3.2). Initially we set the step to the maximum distance from ω to all of its neighbours (line 3) as the solution is restricted to the interior of the polygon that the neighbours form. If the current step length would overshoot the minimum, then it is halved (line 10). In each iteration the algorithm calculates the new network usage $\mathcal{U}(\vec{x}_\omega)$ at the next estimated virtual position, and if this is smaller than the current network usage (line 6), it moves to the new virtual position (line 8) and sets as δ the difference between the old and the new network usage (line 7). After a number of iterations the minimum is trapped and the algorithm terminates when the difference to the current network usage becomes smaller than a predefined threshold δ_t (line 12).

As already mentioned earlier, although the gradient method is simple and easy to implement, it faces problems of slow convergence when the solution is at a point where the derivative is not defined, also referred in the literature as dead point [41]. This is a general problem of the iterative methods that solve the Weber Problem [41]. To improve the speed of our algorithm and avoid slow convergence to the dead points, we make a preprocessing in order to find a good initial point that approximates well the optimal solution.

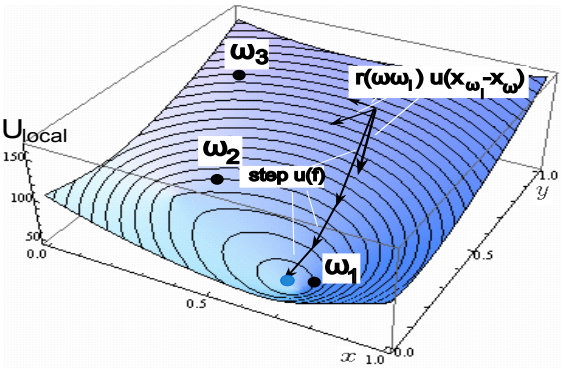


Figure 3.2: Example of the gradient method for a 2-dimensional SOP problem.

To estimate the solution we make an approximation of the corresponding solution for the Manhattan metric (L_1 norm), which is also proposed as a fast approximation method for the Weber Problem in [20]. The idea is to consider the problem for the Manhattan metric instead of the Euclidean metric. In more detail, if we assume the Manhattan metric for computing the distances in the latency space[3], we get the following equation:

$$\mathcal{U}_{\text{local}}^{\text{Mhtn}}(\vec{x}_\omega) = \sum_{l \in \mathcal{F}_{\omega_i}} r_l |\ \vec{x}_\omega - \vec{x}_{\omega i}\ |_1 = \tag{3.8}$$

$$\sum_{l \in \mathcal{F}_{\omega_i}} r_l(|\ x - x_i\ | + |\ y - y_i\ | + |\ z - z_i\ |)$$

The derivative of this function is given by the following equation:

$$\frac{\partial \mathcal{U}_{\text{local}}^{\text{Mhtn}}}{\partial \vec{x}_{\omega_i}} = \sum_{l \in \mathcal{F}_{\omega_i}} r_l \text{sgn}(\vec{x}_{\omega i} - \vec{x}_\omega) \tag{3.9}$$

[3] Without loss of generality we assume a 3-dimensional latency space

In particular, the above equation can be split for each dimension as follows:

$$\frac{\partial \mathcal{U}_{\text{local}}^{\text{Mhtn}}(x)}{\partial x} = \sum_{l \in \mathcal{F}_{\omega_i}} r_l \text{sgn}(x - x_i) \tag{3.10}$$

$$\frac{\partial \mathcal{U}_{\text{local}}^{\text{Mhtn}}(y)}{\partial x} = \sum_{l \in \mathcal{F}_{\omega_i}} r_l \text{sgn}(y - y_i) \tag{3.11}$$

$$\frac{\partial \mathcal{U}_{\text{local}}^{\text{Mhtn}}(z)}{\partial x} = \sum_{l \in \mathcal{F}_{\omega_i}} r_l \text{sgn}(z - z_i) \tag{3.12}$$

In Fig. 3.3 we see an example of how the gradient $\frac{\partial \mathcal{U}_{\text{local}}^{\text{Mhtn}}(x)}{\partial x}$ for a local function $\mathcal{U}_{\text{local}}(x) = 25(x - 0.2) + 25(x - 0.4) + 50(x - 0.6) + 50(x - 0.8)$ approximates the gradient of the $\mathcal{U}_{\text{local}}$ function for Euclidean distance. In particular, the curve in the figure shows the derivative of the $\mathcal{U}_{\text{local}}$ in Euclidean distance, while the discontinuous straight line shows the derivative of the corresponding Manhattan metric function $\mathcal{U}_{\text{local}}^{\text{Mhtn}}(x)$ in x-dimension. As we could also deduce from the Equ. 3.10 the derivative for the Manhattan metric is an increasing function that changes its value only at the positions of the neighbours. To this end, the point where the derivative turns from negative to positive values approximates the root of the Equ. 3.1.

More formally in Alg. 3 we give the algorithm for the Manhattan approximation. The algorithm gets as input the coordinates of the neighbours in increasing order (in each dimension x_i) and it calculates the sum of the positive and negative factors of Equ. 3.9 as the value of the Manhattan approximation increases. At first, the Manhattan approximation is equal to the lowest coordinate x_{ω_1} (line 1). Then, in each iteration the sum of the data rates is calculated (line 5-8) given the position of x_{Mhtn}. Thus, in each iteration the next factor (data rate) in increasing order of the Equ. 3.9 turns from negative to positive and the new sum is computed until it becomes greater or equal to zero (line 10). The stopping condition indicates the change of the sign of the derivative, which means that at this point x_{Mhtn} we have reached the desired approximation solution. This process must be repeated

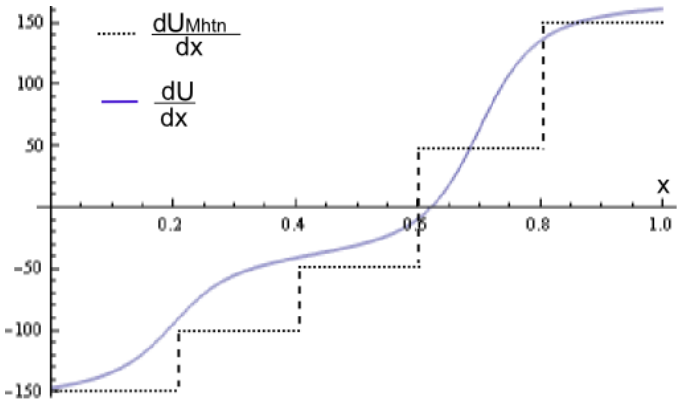


Figure 3.3: Example of approximation for function $\mathcal{U}_{\text{local}}(x) = 25(x - 0.2) + 25(x - 0.4) + 50(x - 0.6) + 50(x - 0.8)$.

for each dimension and finally we get a vector $\vec{x}_\omega$, which is the solution of the Manhattan approximation.

After having specified an approximation of the initial point, we check for dead points by comparing the network usage at the approximated position and the network usage at the neighbours (Alg. 4). If the network usage in one of the points is lower than the one given by the Manhattan approximation, we use this point as initial position and finally we give the output of this procedure to the iterative method that computes the local minimum.

3.1.3.2 Clustering

In the previous paragraph we have described an approximation algorithm for the continuous SOP problem. Although this algorithm approximates the optimal solution for the SOP problem, it might not yield an optimal solution for the continuous MOP problem in cases where two unpinned operators tend to collapse at one position. Such cases can happen when the SOP solution lies at the position of a neighbour that is an unpinned operator [96].

To overcome this problem we use the technique of *operator clustering* [23, 88], whenever the SOP solutions of two neighboring operators coincide.

Algorithm 3 Manhattan Approximation

Require: Coordinates of ω's neighboring operators in increasing order in dimension x_i: $\{x_{\omega_1}, \ldots, x_{\omega_n}\}$

Require: Estimations of data rates of links in L_ω: $\{r(\overline{\omega\omega_1}), \ldots, r(\overline{\omega\omega_n})\}$

Ensure: x_ω is the median of the neighbours in dimension x_i

1: $i \leftarrow 1$
2: **repeat**
3: DataRatesSum $\leftarrow 0$
4: $x_{\text{Mhtn}} \leftarrow x_{\omega_i}$
5: **for all** $x_{\omega_j} \in \{x_{\omega_1}, \ldots, x_{\omega_n}\}$ **do**
6: **if** $x_{\text{Mhtn}} > x_{\omega_j}$ **then**
7: DataRatesSum $\leftarrow$ DataRatesSum $+ r(\overline{\omega\omega_j})$
8: **else**
9: DataRatesSum $\leftarrow$ DataRatesSum $- r(\overline{\omega\omega_j})$
10: **end if**
11: **end for**
12: $i \leftarrow i + 1$
13: **until** DataRatesSum ≥ 0
14: $x_\omega \leftarrow x_{\text{Mhtn}}$

In detail, if the distance between an operator and its nearest neighbour is dropping below a threshold, both operators form a cluster where one operator acts as cluster head. The head of the cluster performs the optimization of the SOP problem for both operators that to the outside now act as one operator. Generally, this procedure can be repeated until a non trivial solution is found, i.e., the solution is a differentiable point or it is a pinned operator. It has to be mentioned that the clustering is evaluated in each iteration of the algorithm. Therefore, an operator might detach later from a cluster if the virtual coordinates of the clustered operators have diverged. However since the operators of a cluster are placed on the same physical node, these local computations do not affect the network load induced by the algorithm.

Algorithm 4 Check Dead Points

Require: Coordinates of ω's neighboring operators: $\{\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}\}$

Ensure: Finds dead point if exists

```
1: for all x_{ω_i} do
2:     if U(x_{ω_i}) < U(x_ω) then
3:         x_{ω_i} ← x_ω
4:     end if
5: end for
```

3.1.3.3 Distribution Properties

If every operator independently optimizes its local position by solving the continuous SOP problem (Algorithm 2), then eventually *every* operator will be placed in a local optimal position. We call this solution an *all-local optimal solution*. Here we prove the following proposition: An all-local optimal solution is a global optimal solution of the MOP problem.

First, we prove the following necessary condition:

Theorem 1 *In the global optimal state, where* $\mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$*is minimal, each operator* ω *is at its local optimal position such that* $\mathcal{U}_{\text{local}}(\vec{x}_\omega)$ *is minimal.*

Proof. We will prove this claim using a proof by contradiction: Assume there exists a minimal solution, $\mathcal{U}_{\text{min}} = \mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}) = \text{min}$, such that there exists at least one operator ω_* that is not at its local optimal position, i.e., $\mathcal{U}_{\text{local}}(\vec{x}_{\omega_*}) \neq \text{min}$. (Otherwise there is nothing to prove since every operator is already at its local optimal position.) Then, the resulting global network usage is $\mathcal{U}_{\text{min}} = \mathcal{U}_{\text{local}}(\vec{x}_{\omega_*}) + \sum_{\omega_j \in \Omega \setminus \omega_*} \mathcal{U}_{\text{local}}(\vec{x}_{\omega_j})$.

Assume all operators besides ω_* are fixed to the places of the global minimal solution. Then, we can do a local optimization for ω_* by moving ω_* to a new position $\vec{x}'_{\omega_*}$ with $\mathcal{U}_{\text{local}}(\vec{x}'_{\omega_*}) = \text{min}$. The resulting global network usage is then defined as $\mathcal{U}'_{\text{min}} = \mathcal{U}_{\text{local}}(\vec{x}'_{\omega_*}) + \sum_{\omega_j \in \Omega \setminus \omega_*} \mathcal{U}_{\text{local}}(\vec{x}_{\omega_j})$.

Since $\mathcal{U}_{\text{local}}(\vec{x}'_{\omega_*}) < \mathcal{U}_{\text{local}}(\vec{x}_{\omega_*})$, $\mathcal{U}'_{\text{min}} < \mathcal{U}_{\text{min}}$, which is a contradiction to the assumption that $\mathcal{U}_{\text{min}}$ is minimal. ■

The sufficient condition that an all-local-optimal solution is always the optimal MOP solution, remains to be proven. To prove the sufficient condition we first show that *each* all-local optimal solution is a (possibly local) minimum of MOP $\mathcal{U}_{\text{global}}$:

Lemma 1 *For any operator graph $\mathcal{G}$, an all-local optimal solution $(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is also a local minimum of the global function $\mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$.*

Proof. If the solution is at a differentiable point, the partial derivatives of $\mathcal{U}_{\text{global}}$ are equal to zero since it holds that:

$$\frac{\partial \mathcal{U}_{\text{global}}}{\partial \vec{x}_{\omega_j}} = \frac{\partial \mathcal{U}_{\text{local}}}{\partial \vec{x}_{\omega_j}} = \sum_{\overline{\omega_j\omega_i} \in \mathcal{F}_{\omega_j}} r(\overline{\omega_j\omega_i}) \times u(\vec{x}_{\omega_j} - \vec{x}_{\omega_i})$$

Therefore the all-local optimal solution is a local minimum.

If one of the partial derivatives is not defined, then the solution of the corresponding SOP problem lies on a non differentiable point (dead point). According to our algorithm in this special case the operator will be clustered with its neighbour and the MOP solution is given by the solution of the clustered operator graph. Clustering is repeated until either it finds a differentiable solution for the clustered operator graph, which we have proved to be a local minimum, or it finds a SOP solution at a non-differentiable point, in case the MOP problem is degraded to a trivial SOP, with one unpinned operator and a set of pinned neighbours (sources and sinks).

Thus we have proven that an all-local optimal solution is also a minimum of the global function.■

Up to now we have proven that the global function is minimal only in all-local optimal states, i.e., an all-local optimal solution is a *local* minimum of $\mathcal{U}_{\text{global}}$. It remains to be proven that an all-local optimal solution is also a sufficient condition for a *global* minimum of $\mathcal{U}_{\text{global}}$. We show this by using the convexity properties of the global function $\mathcal{U}_{\text{global}}$, which has only one

minimum. In that respect, we prove the sufficient condition that finalizes our proof.

Theorem 2 *For any operator graph $\mathcal{G}$, an all-local optimal solution $(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is also the unique minimum of the global function $\mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$.*

Proof. From Lemma 1 we know that an all-local optimal solution is a local minimum of $\mathcal{U}_{\text{global}}$. Furthermore, we know that $\mathcal{U}_{\text{global}}$ has only one minimum since it is a convex function, i.e., if we have found a local minimum of $\mathcal{U}_{\text{global}}$ we also have found its global minimum. ■

So finally we know that an all-local optimal solution is a local minimum of $\mathcal{U}_{\text{global}}$. Furthermore, we have shown that $\mathcal{U}_{\text{global}}$ has only one minimum, i.e., if we have found a local minimum of $\mathcal{U}_{\text{global}}$ we also have found its global minimum. Therefore, an all-local optimal solution must be the global optimal solution of continuous MOP problem, which is approximated by our distributed placement algorithm.

3.1.3.4 Dynamic adaptation of Operator Placement

After the initial placement, the positions of operators is optimized continuously according to Algorithm 1. Note that although the initial placement yields already the final positions of operators, the quality of this initial solution might degenerate due to dynamically changing network conditions. Therefore, we let each physical node dynamically re-evaluate the positions of its hosted operators, in order to migrate operators if necessary.

In detail, an operator re-placement for operator ω is triggered in two situations:

1. The coordinates of a neighbouring operator of ω in the latency space change.

2. The input or output data rate of ω changes.

The first situation might occur, if a neighbouring operator re-evaluates its position in the latency space for the same reasons (a neighbour changed its position or data rates changed). An operator reacts to this change, using an event-driven mechanism. If an operator calculates a new SOP solution, it sends its new coordinates to its neighbouring operators. This event will also trigger a re-evaluation of the receiver's coordinates.

Here we have to mention that different all-local-optimal solutions might exist that correspond to the same minimum. We have only proven that there is a unique minimum but not that this minimum is reached by only one solution. In fact if we visualize the problem in the coordinate space, we see that there might be multiple symmetric points where the distance to the neighbours remain the same. Assume, for instance, the simple example in Figure 3.4, where we have only one source and one sink and the data rates to both directions of the free operator ω are equal (r). In this example, all the positions of ω that lie on the straight line between the source and the sink cause the same global network usage $\mathcal{U}_{\text{global}}(\vec{x}_\omega) = r \times d$. The existence of the symmetric solutions can lead to oscillations for more complex topologies as the neighbors will move back and forth between the symmetric positions and trigger re-placement events although the minimum has been reached already. To avoid such oscillations, every operator checks if the difference between the new minimum after the change of the coordinates of the neighbouring operators and the one before the dynamic change is below a threshold, and in that case it does not send the new position to its neighbours.

The second situation is due to rate changes of the incoming data streams. For instance, a sensor might produce streams of considerably different rates during the day and night. Consider for instance a traffic flow sensor, measuring the number of cars passing-by a certain location. During rush hours the data rate of this sensor is obviously higher than during the night. Thus, subsequent operators attached to this sensor might receive streams of different data rates during different periods. Moreover, an operator might produce a different output data rate if the values of the input data change. For instance,

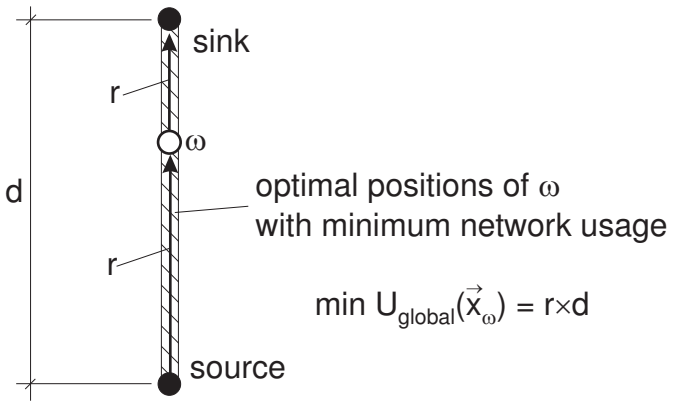


Figure 3.4: Symmetric Operator Placement Solutions.

a temperature filter might be configured such that it only reports temperatures higher than 20° Celsius. So depending on the input temperature, this operator produces different output data rates.

Obviously, such dynamic changes cannot be foreseen in advance. Rather, we let each operator measure the current input and output data rates continuously using an exponential moving average:

$$r_{\text{new}} = \alpha \times r_{\text{current}} + (1 - \alpha) \times r_{\text{old}}$$

With this formula, the new data rate of a stream is calculated based on the currently measured data rate r_{current} and the previous data r_{old}. Parameter $\alpha \in [0, 1]$ defines, how much the data rate is smoothed by weighting historic values.

If a significant change of data rates is detected by an operator, the operator will re-evaluate its position in the latency space, send this position to its neighbours, and possibly initiate a migration if a different physical node is closer to the new position than the current one.

3.1.4 Integer Linear Programming Formulation

In this section, we formulate the optimization problem in (Equ. 3.2) as an integer linear programming (ILP). This problem formulation is directly applicable to a centralized ILP solver. Although we explicitly strive for a distributed algorithm rather than a centralized solution requiring the gathering of global knowledge at a central node, this ILP serves as a reference for the evaluation of our distributed approach.

The problem formulation is based on the ILP for subgraph isomorphism presented in [68]. In particular, we adapt the ILP for finding sub-graph isomorphisms by introducing constraints for pinned operators. Moreover, we changed the objective of the optimization to reflect our goal, namely, minimizing the sum of delay-data rate products.

In detail, for the integer linear problem formulation, we keep the same system of the physical network $\mathcal{H} = (\mathcal{V}, \mathcal{E}, l)$ and the operator graph $\mathcal{G} = \{\Omega, \mathcal{F}, r\}$ as introduced in Subsection 3.1.1, and we use integer linear programming, instead of the latency space abstraction to formulate our problem. To this end, we introduce the following definitions. We define the placement of each free operator $\omega_i \in \Omega_{free}$ by a binary vector $x_i \in \{0,1\}^{|\mathcal{V}|}$. Since an operator has to be placed on exactly one host, the vector x_i has *exactly* one 1 at the position of host v_j, where the operator is placed. Furthermore, we define a vector $y_i \in \{0,1\}^{|\mathcal{E}|}$ for each operator graph link, such that the vector has exactly one 1 at the position of edge e_j , where the operator graph edge is placed.

Given the above definitions, the problem of minimizing the network usage can be expressed as:

$$\min \sum_{f \in \mathcal{F}} \sum_{e \in \mathcal{E}} r(f) l(e) y_{f,e}, \tag{3.13}$$

subject to :

$$x_{\omega,\nu_\omega} = 1 \quad \forall\, \omega \in \Omega_{pinned} \tag{3.14}$$

$$\sum_{\nu \in \mathcal{V}} x_{\omega,\nu} = 1 \quad \forall\, \omega \in W \tag{3.15}$$

$$\sum_{e \in \mathcal{E}} y_{f,e} = 1 \quad \forall f \in \mathcal{F} \tag{3.16}$$

$$\sum_{(\nu_1,\nu_2) \in \mathcal{E}} y_{(\omega_1,\omega_2),(\nu_1,\nu_2)} = x_{\omega_1,\nu_1} \quad \forall\, \nu_1 \in \mathcal{V},\, \forall\, (\omega_1,\omega_2) \in \mathcal{F} \tag{3.17}$$

$$\sum_{(\nu_1,\nu_2) \in \mathcal{E}} y_{(\omega_1,\omega_2),(\nu_1,\nu_2)} = x_{\omega_2,\nu_2} \quad \forall\, \nu_2 \in \mathcal{V},\, \forall\, (\omega_1,\omega_2) \in \mathcal{F} \tag{3.18}$$

In the objective (3.13), we minimize the sum of bandwidth-delay products for each underlay edge e (host-to-host connection) on which an operator graph edge f is placed. Constraint (3.14) ensures that each pinned operator $\omega_i \in \Omega_{pinned}$ is placed on the given host ν_ω. Moreover, constraint (3.15) and (3.16) guarantee that each operator ω is placed on exactly one host and each operator graph edge f is placed on exactly one underlay edge e. Finally, constraint (3.17) and (3.18) ensure that the operator placements and operator graph edge placements defined by the vectors x and y, respectively, are consistent. That is, if two operator graph edges start at the same source node (operator), then they must be mapped such that the target edges in the underlay network also start at the source node (host). Moreover, this host must also be the target of the operator mapping of the source operator defined by x. The same constraints applies to the destination nodes of operator graph edges.

Given the above ILP formulation of our problem, we can use an ILP solver to solve the discrete multi-operator placement problem in a centralized manner. In Chapter 4, we present evaluation results of the execution of the above ILP compared to the distributed algorithm (MOPA).

3.2 Network Delay Constrained Optimization

Although the optimization of network usage contributes to the scalability of the system, it does not take into consideration any constraint from the application's point of view. Here, we consider also latency constraints from the application. In this section, our target are applications that do not require intensive processing of operators. In other words, we assume that the transmission delay as part of the network delay[4] as well as the processing delay are negligible. In the next section, we will consider processing-intensive applications. This assumption applies to scenarios where data units are small and only simple processing operations are required. A typical example is the communication of temperature values and checking whether the values exceed a certain threshold. Since a sensor value only contains few bytes, its transmission delay (time to put the sensor data on the wire) is small. Moreover, comparing a simple value like an integer or floating point against a threshold induces a very small delay.

In particular, the proposed operator placement algorithm considers a given end-to-end delay while trying to minimize the network usage. Our algorithm is based on a two-phase process [91]. First, we find an operator placement that minimizes network usage (*unconstrained optimization phase*). Secondly, we distort the optimal solution such that the QoS constraint is fulfilled while minimizing the impact onto the network usage (*constraint satisfaction phase*). This basic approach is different from related constrained optimization approaches that usually first enumerate a set of feasible solutions with respect to the QoS constraint and from this set select the best solution with respect to the optimization criteria [52, 81, 90]. In contrast to these approaches, our approach enables us to calculate the costs in terms of the optimization metric that we have to pay to fulfill the given QoS constraint. The knowledge about the individual costs for achieving the specific constraint can be a useful

[4]Network delay=transmission delay+signal propagation delay+queuing delay of routers+processing delay of routers.

Operator Graph Model	
$\overline{\omega_i\omega_j} \in \mathcal{Q}$	Set of link(s) that connect ω_i and ω_j in the overlay network
$\mathcal{Q}$	Set of paths in an operator graph
$\mathcal{Q}_{S\rightarrow A}$	Set of end-to-end paths in an operator graph

Table 3.3: Extended Network Delay Constrained System Model Notation

information for the system in order to negotiate the level of QoS provision. For instance, if achieving the QoS guarantees involves negligible cost, it can be acknowledged without further negotiation. However, if it would require large costs, a re-negotiation could be initiated to relax the QoS constraint in favour of a less costly solution.

To solve this constrained optimization problem, we use a two-phase optimization process. In the first optimization phase, we use the algorithm presented in the previous chapter to minimize the network usage. Then we apply a constraint satisfaction method, which calculates a solution for the constrained optimization problem by moving operators in the latency space along a path of minimal increase of network usage to a new position fulfilling the delay constraint after the mapping of the continuous solution to the discrete set of physical nodes. We will both show at the rest of this section, how operators can be placed at the initial deployment, and how operator positions can be adapted to dynamic network conditions during runtime.

In the next subsection we are going to extend the system model presented in Subsection 3.1.1 to express the targeted constrained optimization problem. Then we are going to formulate the constrained optimization problem, before we present our approach.

3.2.1 System Model

Here we use as a baseline the system model introduced in Subsection 3.1.1. In that respect in the underlying system, we use the physical network model and

the latency space abstraction as presented in Subsection 3.1.1 , i.e. we assume a set of physical nodes $\mathcal{V}$ distributed in the network, where each physical host uses a network coordinate algorithm to determine its position in the latency space.

Furthermore, in our execution model, we assume, similar to Subsection 3.1.1, a stream processing task modelled as a directed operator graph $\mathcal{G} = \{\Omega, \mathcal{F}, r\}$ that consists of a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ of *operators* connected by a set $\mathcal{F} = \{\overline{\omega_1\omega_i}, \ldots, \overline{\omega_j\omega_n}\}$ of *links*. However, we extend this execution model in order to express the end-to-end latency. Table 3.3 summarizes the new concepts of the extended execution model. As explained earlier a link $\overline{\omega_i\omega_j} \in \Omega \times \Omega$ is a directed connection that links one operator ω_i to another ω_j. Up to now, we have only used this notation, for one-hop neighbours in the operator graph. Here, we extend this notation to *paths*between operators consisting of several hops in the overlay network. That is, $\overline{\omega_i\omega_j}$ denotes the path between ω_i and ω_j defined as the union of the links on this path. An *end-to-end path* $\overline{\omega_i\omega_j}$ denotes a path connecting a source ω_i and a sink (application) ω_j. Each graph typically contains a set of *end-to-end paths* $\mathcal{Q}_{S\to A} = \{\overline{\omega_i\omega_j}, \ldots, \overline{\omega_k\omega_l}\}$.

3.2.2 Problem Statement

Our goal is to find an operator placement on physical hosts such that the network usage of inter-operator data streams is minimized under a given latency constraint. Next, we give a formal definition of the resulting placement problem.

We consider the optimization problem as presented in the previous chapter (Equ. 3.4). In addition to this optimization goal, we introduce the objective function to express the constraints in terms of latency. In general, to calculate the latency, we have to sum up the network and processing delays, since normally each operator introduces a certain processing delay. However, as we have mentioned earlier we consider applications that send small messages

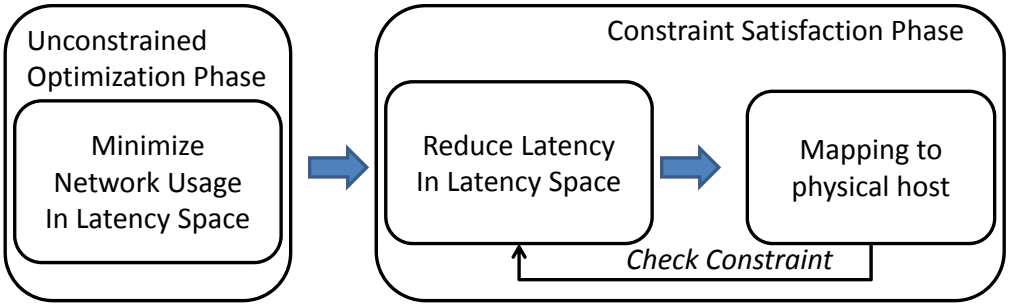


Figure 3.5: Process flow of the initial placement.

over long distances. Therefore, we consider the delay to process these small messages on the physical host to be negligible in comparison with the network delay. Given this assumption, we introduce the latency of a path $\overline{\omega_i\omega_j}$ as the sum of the delays of all the links participating in the path $\mathcal{L}(\overline{\omega_i\omega_j}) = \sum_{\overline{\omega_k\omega_l}\in\overline{\omega_i\omega_j}} \mathcal{L}(\overline{\omega_k\omega_l})$.

Then, the latency of an operator graph will be the maximum latency of all the end-to-end paths of $\mathcal{G}$:

$$\mathcal{L}(\mathcal{G}) = \mathcal{L}(\overline{\omega_i\omega_j}, \ldots, \overline{\omega_k\omega_l}) = \max_{\overline{\omega_i\omega_j}\in\mathcal{Q}_{S\to A}} \mathcal{L}(\overline{\omega_i\omega_j})$$

Finally we define our *constrained optimization* problem as follows:

$$\min \mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}), \text{subject to } \mathcal{L}(\mathcal{G}) \leq l_{max} \tag{3.19}$$

,where l_{max} is a user defined constraint for the maximum delay.

3.2.3 Constrained Optimization Algorithm

Next we present our approach for solving the constrained optimization problem of Equ. 3.19. First we give an overview of the whole process and then we describe in detail how operator positions are calculated.

3.2.3.1 Constrained Optimization Process: Overview

The whole process of our constrained optimization method consists of two phases as shown in Fig. 3.5. In the first phase, called *Unconstrained Optimization Phase* we find the optimal position of the operators in the latency space such that the network usage of the operator graph becomes minimal, i.e., we solve the continuous version of the unconstrained optimization problem of Equ. 3.4. In the second phase, called *Constraint Satisfaction Phase*, we try to find a solution that fulfills the given latency constraint on the one hand. On the other hand, the calculated solution should deviate from the (unconstrained) optimal placement w.r.t. network usage only minimally. Therefore, we start at the optimal positions in terms of network usage that were calculated in the first phase, and move operators towards the latency minimum on paths that increase the network usage the least. This movement is executed in the continuous latency space. After we have moved an operator for a certain distance towards the latency minimum, we map the continuous positions to the discrete positions of physical hosts and check whether the latency constraint has been fulfilled. If it is fulfilled, we have found a solution of the constrained optimization problem (Equ. 3.19); if it is not fulfilled, we initiate another iteration by moving operators further into the direction of the latency minimum.

For the initial placement, we execute the algorithm centrally on one dedicated physical node, called *coordinator node*. After the deployment of operators, the adaptation of the solution is done in a distributed manner. This means that an event-driven model initiates a new round of optimizations each time it detects a change of the conditions of the problem. In Section 3.2.3.5 we are going to describe in detail, how the algorithm is executed to adapt the placement of operator to dynamic changes, after we have described the centralized execution of the optimization and constraint satisfaction phase in the next sub-sections.

3.2.3.2 Unconstrained Optimization Phase

During the unconstrained optimization phase, we search the minimum of the unconstrained optimization problem of Equ. 3.4. We use for that purpose the unconstrained optimization placement algorithm presented in the previous section. As presented earlier, the placement algorithm can be executed centralized as well as distributed. During the initial placement, the above mentioned coordinator node executes this algorithm centrally. We assume for the initial placement that the data rates are derived from the type of application or estimated based on statistics gathered from previous deployments. During the execution of the operators, the adaptation algorithm can adapt these values by measuring the data rates during runtime as we see in Section 3.2.3.5.

3.2.3.3 Constraint Satisfaction Algorithm (MOPA–LMAX)

After the unconstrained optimization, all operators are in a position such that the induced network usage is minimal. However, since the unconstrained optimization only solves the unconstrained optimization problem, the maximum latency path in the operator graph might violate the given delay constraint. Next we present the constraint satisfaction algorithm, which we call MOPA-LMAX, that moves operators from their optimal position such that: (1) the latency is reduced, (2) the deviation of the network usage after re-placement is minimal compared to the optimal network usage immediately after the unconstrained optimization. First, we give an overview of this constraint satisfaction algorithm before we explain it in detail.

The general course of actions of the MOPA-LMAX algorithm, shown in Alg.5, is as follows. First, we map the current continuous positions of the operators to the closest physical hosts in the latency space, in order to be able to check the latency constraint after the mapping to physical hosts (line 1) rather than onto virtual hosts. Whenever we map operators to physical hosts, we only consider non-overloaded physical hosts to prevent bottlenecks. Then,

we check whether the latency of the operator graph $\mathcal{L}(\mathcal{G})$ fulfills the given latency constraint l_{max} (line 2). If it fulfills the constraint, we have found a suitable operator placement and return this mapping (line 2, 11). If the latency constraint is violated, we find new coordinates for the nodes. First, we determine the maximum latency path (line 3) of the operator graph in the continuous space. Then, we select one operator on this path and determine a direction of movement that reduces the latency of this path (line 4) and at the same time increases network usage the least. Details about this step are presented in Subsection 3.2.3.4. If we cannot find a direction that reduces the latency, we cannot find a solution that satisfies the given latency constraint and return the current mapping of operators (line 5-6). Otherwise, we move the selected operator by a certain step length into the calculated direction in the latency space (line 8). Then we repeat the steps of calculating a mapping to physical hosts (line 9), and checking for the satisfaction of the latency constraint.

The step size of the iterative algorithm should be selected carefully since it affects the accuracy of the solution. For our evaluation, empirically we see that a step of 1 gives a good approximation of the solution. In order to map the continuous solution to physical nodes whose positions in the latency space are closest to the calculated virtual node positions, we realize a nearest neighbour search [107], similar to the unconstrained algorithm presented in Section 3.1. The coordinator node queries this infrastructure to perform the mapping step. Finally, it deploys the operators on the physical nodes and the execution of the operator tree starts.

3.2.3.4 Operator Selection and Direction of Movement

Next, we explain in detail how we select the operator to move and its respective direction (line 4, Algorithm 1). For this purpose, we first calculate the optimal direction for each operator on the maximum latency path and then select the one node with the minimal impact on the network usage.

Algorithm 5 MOPA-LMAX

Require: $\mathcal{U}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal

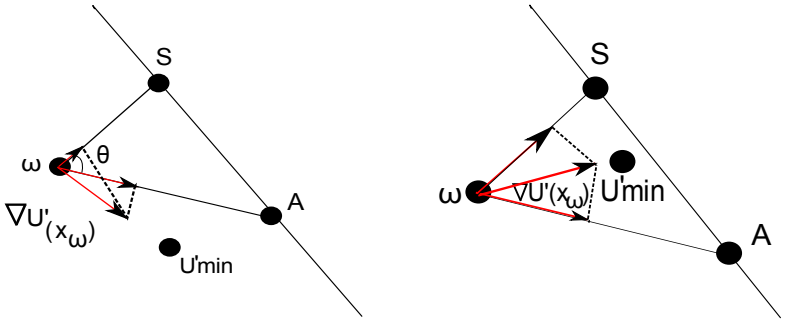
Ensure: Finds a mapping $(\nu_1, \ldots, \nu_n)$ such that $\mathcal{L}(\mathcal{G}) \leq l_{max}$ and $\mathcal{U}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal

```
map each operator ω_i to closest non-overloaded ν_i
while (L(G) > l_max) do
    determine maximum latency path ω_i ω_j
    select operator ω ∈ ω_i ω_j and direction dir to move
    if dir = 0 then {already at latency minimum}
        return current mapping (ν_1, ..., ν_n)
    end if
    move operator ω by a step length step into dir
    map operator ω to closest non-overloaded ν_new
end while
return current mapping (ν_1, ..., ν_n)
```

More formally, we first search for a direction $\overrightarrow{dir} = (dir_x, dir_y, dir_z)$ to move each unpinned operator ω on the maximum latency path, such that: (1) $\mathcal{L}(\overline{\omega_i\omega_j})$ is reduced, (2) the increase of $\mathcal{U}(\vec{x}_\omega)$ is minimal if the operator is moved into the direction $\overrightarrow{dir}$.

In general, the impact on the network usage when moving into a certain direction $\overrightarrow{dir}$ is inverse proportional to $\phi_\omega(\overrightarrow{dir}) = \nabla\mathcal{U}(\vec{x}_\omega) \cdot \overrightarrow{dir}$, where $\cdot$ denotes the dot product of the network usage gradient $\nabla\mathcal{U}(\vec{x_\omega})$ and the direction of the movement $\overrightarrow{dir}$, i.e., $\frac{\partial\mathcal{U}_{\text{local}}(x)}{\partial x} * dir_x + \frac{\partial\mathcal{U}_{\text{local}}(y)}{\partial y} * dir_y + \frac{\partial\mathcal{U}_{\text{local}}(z)}{\partial z} * dir_z$. Note that since $\overrightarrow{dir}$ is a unit vector, ϕ_ω models the projection of the gradient onto the direction of the movement. For instance, if $\phi_\omega < 0$, then the operator is moving inversely to the gradient and therefore the network usage will increase proportional to the value of the gradient.

More formally, if D is the set of possible directions that reduce the latency, our goal is to maximize the function $\phi_\omega(\overrightarrow{dir}) = \max_{\overrightarrow{dir} \in D}\{\nabla\mathcal{U}(\vec{x}_\omega) \cdot \overrightarrow{dir}\}$. Since $\mathcal{L}(\overline{\omega_i\omega_j})$ is a convex function, moving into the direction of L_{min} will



(a) $\mathcal{U}'_{\min}$ outside $\Delta S\omega A$. (b) $\mathcal{U}'_{\min}$ inside $\Delta S\omega A$.

Figure 3.6: Direction of the movement for MOPA-LPMAX

certainly reduce latency and in the ultimate case will lead to the minimum latency path. Actually, $L_{\min}$ might not be a single point but a line segment connecting a source S and a sink A since obviously all positions on a direct connection between S and A will lead to minimum latency.

Based on the observation that $L_{\min}$ is a line segment rather than a unique point, we show next how to calculate the direction $\overrightarrow{dir}$ that points towards $L_{\min}$ and maximizes $\phi_\omega(\overrightarrow{dir})$. In Fig. 3.6 we see an example where an unpinned operator ω should be moved towards the latency minimum $L_{\min}$ defined by the line segment SA. As we see in this figure, the possible directions that point to the latency minimum are inside the angle θ between the vectors $\overrightarrow{\omega A}$ and $\overrightarrow{\omega S}$. Since the possible directions belong only to the plane defined by the points $S\omega A$, the direction $\overrightarrow{dir}$ will be affected only by the projection of the network usage gradient $\nabla\mathcal{U}'(\vec{x_\omega})$ on the plane $S\omega A$. Thus, in Fig. 3.6 we see that the direction that maximizes the dot product ϕ_ω is the direction that has the smallest angle θ to the projection of the gradient of the network usage on the plane $S\omega A$.

In general, we can distinguish two different cases according to the position of the projection of the network usage minimum $\mathcal{U}'_{\min}$ on the plane $S\omega A$: (1) $\mathcal{U}'_{\min}$ is outside the triangle $\Delta S\omega A$ (Fig. 3.6). In this case, the

Algorithm 6 Operator and Direction Selection

Require: Positions $\vec{x_\omega}, S_1, A$

Ensure: Finds ω_{opt} and $\overrightarrow{dir}_{\text{opt}}$ such that ϕ_ω is maximum

1: **for all** $\omega \in \overline{\omega_i\omega_j}$ **do**
2: **if** $\nabla\mathcal{U}'(\vec{x_\omega}) \times u(\overrightarrow{\omega A}) \cdot \nabla\mathcal{U}'(\vec{x_\omega} \times u(\overrightarrow{\omega S})) < 0$ **then**
3: $\phi_\omega(\overrightarrow{dir}) \leftarrow \|\nabla\mathcal{U}'(\vec{x_\omega})\|$
4: $\overrightarrow{dir} \leftarrow \nabla\mathcal{U}'(\vec{x_\omega})$
5: **else**
6: $\phi_\omega(\overrightarrow{dir}) \leftarrow \max_{\overrightarrow{dir} \in \{u(\overrightarrow{\omega A}), u(\overrightarrow{\omega S})\}} \nabla\mathcal{U}'(\vec{x_\omega}) \cdot \overrightarrow{dir}$
7: $\overrightarrow{dir} \leftarrow \arg\phi_\omega(\overrightarrow{dir})$
8: **end if**
9: **if** $\phi_\omega > \phi_{\text{opt}}$ **then**
10: $\phi_{\text{opt}} \leftarrow \phi_\omega$, $\omega_{\text{opt}} \leftarrow \omega$, $\overrightarrow{dir}_{\text{opt}} \leftarrow u(\overrightarrow{dir})$
11: **end if**
12: **end for**

direction $\overrightarrow{dir}$ should be the direction of the vector $\overrightarrow{\omega A}$ or $\overrightarrow{\omega S}$, whichever has the smallest angle θ to the projection of the gradient $\nabla\mathcal{U}'(\vec{x_\omega})$ on the plane $S\omega A$. Therefore, this vector will maximize the dot product, i.e., $\overrightarrow{dir} = \arg\{\max_{\overrightarrow{dir} \in \{u(\overrightarrow{\omega A}), u(\overrightarrow{\omega S})\}} \nabla\mathcal{U}'(\vec{x_\omega}) \cdot \overrightarrow{dir}\}$, where u denotes the unit vector. (2) $\mathcal{U}'_{\text{min}}$ is inside the triangle $\Delta S\omega A$ (Fig. 3.6). In this case, the direction $\overrightarrow{dir}$ should be the direction of the projection of the gradient $\nabla\mathcal{U}'(\vec{x_\omega})$, since this will induce a maximal decrease of network usage.

In order to distinguish between the two cases, we have to identify when $\mathcal{U}'_{\text{min}}$ is inside the triangle. As we see in the example of Fig. 3.6, $\mathcal{U}'_{\text{min}}$ is inside the triangle when vector $\overrightarrow{\omega A}$ and $\overrightarrow{\omega S}$ are on different sides of $\nabla\mathcal{U}'(\vec{x_\omega})$. This condition (the relative position of $\nabla\mathcal{U}'(\vec{x_\omega})$) cannot be identified only through the dot product. Therefore, we need to calculate the cross products $\nabla\mathcal{U}'(\vec{x_\omega}) \times \overrightarrow{\omega A}$ and $\nabla\mathcal{U}'(\vec{x_\omega}) \times \overrightarrow{\omega S}$. Note that the cross product of two vectors $\vec{A} = \{A_1, A_2, A_3\}$ and $\vec{B} = \{B_1, B_2, B_3\}$ in three dimensional Euclidean space, is given by: $\vec{A} \times \vec{B} = (A_2B_3 - A_3B_2)i + (A_3B_1 - A_1B_3)j + (A_1B_2 -$

$A_2B_1)k$. Therefore, the result of the cross product is another vector which is perpendicular to the plane containing the two input vectors. If the two vectors $\overrightarrow{\omega A}, \overrightarrow{\omega S}$ lie on different sides of vector $\nabla\mathcal{U}'(\vec{x_\omega})$, then their cross products $\nabla\mathcal{U}'(\vec{x_\omega}) \times u(\overrightarrow{\omega A})$, $\nabla\mathcal{U}'(\vec{x_\omega}) \times u(\overrightarrow{\omega S})$ have different directions, i.e., the dot product of their cross products are negative.

Algorithm 6 shows the final algorithm that we use to determine the operator to move and the direction of the movement. For all operators on the path, we find the optimal direction that maximizes the dot product ϕ_ω (line 2-12). To this end, we first check if the projection of the network usage minimum is inside the triangle, i.e., the dot product of the cross products is negative (line 2). Then the optimal direction is the direction of the projection of the gradient (line 3-4). In any other case, ϕ_ω is set to the maximum of the dot products $\nabla\mathcal{U}'(\vec{x_\omega}) \cdot u(\overrightarrow{\omega A}), \nabla\mathcal{U}'(\vec{x_\omega}) \cdot u(\overrightarrow{\omega S})$ (line 6-7). Finally, we compare ϕ_ω to the current maximum dot product of the path. If ϕ_ω exceeds the current maximum, we keep the identifier for the operator to move as well as the direction of the movement (line 9-11). The iterative process continues until we have checked all the operators on the path. The output of the algorithm is the identifier of the best operator to move ω_{opt}, together with its optimal direction.

3.2.3.5 Dynamic adaptation of placement

After the initial placement of operators, the operator graph is deployed in the network. During the execution of the operators a change in network conditions or the data rates of inter-operator data streams might degrade the initial placement by rendering the initial solution suboptimal or violating the given delay constraints. Therefore, the placement of operators has to be adapted to dynamic network conditions. Next, we describe the dynamic adaptation during runtime.

The adaptation process is based on an event-driven model that triggers the re-placement of operators whenever the position of neighbouring operators

change or if the data rates of incoming or out-coming data streams change. In case of such changes, the operator graph enters the unconstrained optimization phase where operator positions are optimized for minimal network usage. The unconstrained optimization is realized by running the algorithm presented in Section 3.2.3 in a distributed manner [92]. In the distributed case, each operator optimizes its local network usage and exchanges messages with its neighbours to communicate its new position, until the positions of its neighbours do not change any more. After a number of iterations, the distributed algorithm yields the final solution for the operator graph.

Subsequently, the operator graph enters the constraint satisfaction phase. However, since the operators are distributed on different hosts, the operators should coordinate to decide when and how to enter the constraint satisfaction phase. For this purpose, as for the initial placement, we again use a coordinator node. For the initial placement, the position of the coordinator node is not crucial, whereas for the adaptation it is beneficial to choose the root node as coordinator, since it can help as the root node of an aggregation tree. To detect the transition between the two phases, we create an aggregation tree where state information (the current position of operators) is propagated bottom-up towards the root. To avoid additional message overhead, we piggy-back this state information of a subtree onto the messages that are communicated during the unconstrained optimization phase. Thus, the coordinator node has a global view onto the operator graph at each point in time with a delay that depends on the time to transmit the messages along the tree.

Fig. 3.7 exemplifies our approach for an operator graph of 15 nodes. In that case, during the optimization phase, the 6 free operators exchange messages to cooperatively converge to a network usage minimum. Note that the operators connected to the sink, which is the coordinator node, send messages to the coordinator node to update the global view onto the operator graph. Based on this global view, the coordinator node assumes that the unconstrained optimization has reached a stable state, when it does not receive any message

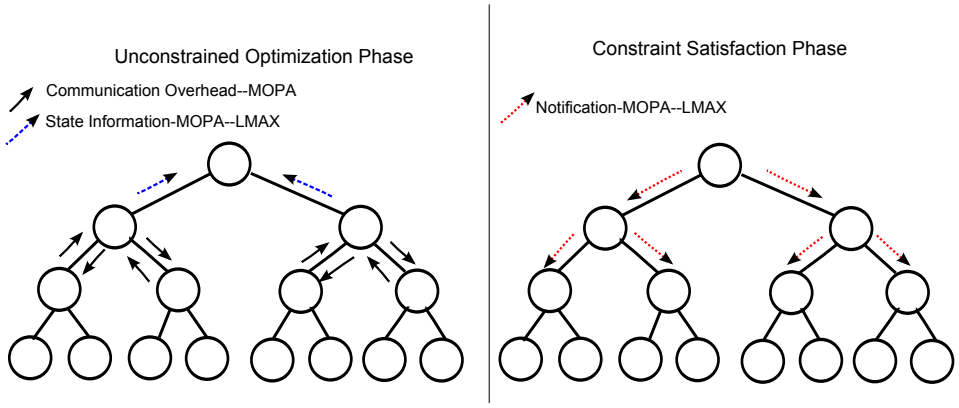


Figure 3.7: Communication Overhead Example for MOPA & MOPA-LPMAX

from its neighbouring operators for a certain time interval Δt. If this time expires and no state update messages have been received, the coordinator node performs the constraint satisfaction phase centrally as described in the previous section. When it finds a new solution of the constrained optimization problem, the root propagates a message to all nodes/operators in the tree, containing the mapping of the unpinned operators as shown in Fig. 3.7. After the propagation of the message along the tree, all the operators are informed about their final position and initiate a migration if necessary.

The time of the transition from the optimization phase to the constraint satisfaction phase depends on the time of the dynamic changes that may happen unexpectedly in the system. Therefore,it is possible that a new dynamic change triggers the optimization phase, while the coordinator node performs the constraint satisfaction step. In that case, the coordinator node will not propagate the solution of the constrained optimization problem to the tree and the whole optimization process will start from the beginning. In that respect, the point in time to pass from the optimization to constraint satisfaction phase does not affect the final outcome of the algorithm, but it determines the responsiveness of the system to dynamic changes.

In particular, the parameter Δt defines the time to respond to dynamic

changes. If it is set very low, then the system will react faster to dynamic changes by re-calculating the physical mappings of operators more frequently and thus resulting possibly in more migrations in the physical network. Since the migrations are costly both in terms of communication overhead and latency, we try to avoid entering the constraint satisfaction phase before we reach a stable state by approximating an upper bound for the time to get the messages transmitted along the operator tree. Thus, we propose to set this parameter equal to the time to send a message from the most distant source in the tree to the root plus a small constant.

In very dynamic environments, the unconstrained optimization might take a long time to reach a stable state during which the delay constraint is possibly not fulfilled. In order to avoid being stuck in the unconstrained optimization phase for a long period, we introduce an additional parameter ΔT that defines the maximum time interval that the root should wait until it executes the constraint satisfaction algorithm.

Finally, we analyse the communication overhead induced by the adaptation algorithm. In general, the induced message overhead mainly consists of the following messages: (1) The messages required to distributively solve the unconstrained optimization problem during the unconstrained optimization phase. An analysis of this overhead is presented in Chapter 4, (2) The state information propagated upwards in the aggregation tree to determine the end of the unconstrained optimization. The additional overhead introduced for transmitting state information is expected to be small since we can re-use the information propagated during the unconstrained optimization in step 1—in this phase, nodes already exchange their coordinates. (3) The notification messages about new operator positions propagated downwards along the operator tree. This requires only $\#unpinnedOperators$ messages.

Fig. 3.7 shows an example of the communication overhead for MOPA and MOPA-LMAX for an operator graph of 15 nodes. We see that during the unconstrained optimization phase, the unpinned operators exchange messages to cooperatively find a global network usage minimum. To this end, the

additional overhead introduced for transmitting state information to the root of the aggregation tree is proportional to the communication overhead of MOPA. At the constraint satisfaction phase, we need only 6 messages equal to the number of the unpinned operator in the operator graph. Overall, we see that only Steps (2) and (3) introduce a small amount of additional messages compared to the unconstrained optimization.

3.2.4 Integer Linear Programming Formulation

In this section, we extend the formulation of the optimization problem in (Equ. 3.2) as an integer linear programming (ILP) presented in Subsection 3.1.4, to provide an ILP formulation for the constrained optimization problem in (Equ. 3.19). Similar to the ILP formulation presented in Section 3.1.4, this problem formulation is directly applicable to a centralized ILP solver and it will be used as a reference for the evaluation of the constrained satisfaction algorithm Alg. 5 in Chapter 4.

For the integer linear programming formulation of the constrained optimization problem, we keep the same definitions as presented in Section 3.1.4 and we extend our model similar to Subsection 3.2.1 with the definition of the end-to-end paths $\mathcal{Q}_{S\to A} = \{q_1, \ldots, q_n\}$. For the ILP formulation each path q_i is defined as binary vector: $q_i \in \{0,1\}^{|\mathcal{F}|}$, with

$$q_{i,f} = \begin{cases} 1, & \text{if operator graph edge } f \in \mathcal{F} \text{ is part of the end-to-end path.} \\ 0, & \text{otherwise.} \end{cases}$$

A valid solution must satisfy the latency constraint for each end-to-end path q_i. Therefore, to solve the constrained optimization problem in (Equ. 3.19), we add in the ILP formulation presented in Subsection 3.1.4 the following constraint:

$$\sum_{f\in\mathcal{F}} \sum_{e\in\mathcal{E}} q_{i,f} y_{f,e} l(e) \geq l_{max}, \forall\, q_i \in \mathcal{Q}_{S\to A} \tag{3.20}$$

3.3 Processing and Network Delay Constrained Optimization

In previous section, we introduced an algorithm fulfilling latency constraints while optimizing the network usage. However, the previous algorithm targets applications communicating small data units where the transmission and processing delays are negligible. To this end, in this section, we extend the constraint optimization problem to also consider processing and transmission delays. Thus, we target processing intensive applications. Consider, for instance, a multimedia streaming application [76], transferring larger chunks of data to be processed. In this category of applications, processing delay and data transmission delay could affect significantly the end-to-end delay of the system.

Our approach first optimizes for network usage and then applies a constraint satisfaction algorithm that fulfils the end-to-end latency constraints [93]. The computing resources are used in an efficient way in the sense that nodes with more residual resources are preferred over others, and only if they reduce the processing delay of the corresponding operators to be placed.

3.3.1 System Model

In order to consider transmission and processing delays during the placement, we need an extended system model also modelling the size of data to de transmitted and the processing at hosts. Next, we introduce this extended model.

In our execution model, we use the basic definitions presented in Subsection 3.1.1 and Subsection 3.2.1, i.e., a stream processing task is modelled as a directed operator graph $\mathcal{G} = \{\Omega, \mathcal{F}, r\}$ that consists of a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ of *operators* that are connected by a set $\mathcal{F} = \{\overline{\omega_1\omega_i}, \ldots, \overline{\omega_j\omega_n}\}$ of *links*. Additional to this execution model, we introduce here the notion of a *data unit* which represents the minimal discrete data unit to be transmitted between the operators and processed at hosts. A sequence of data units forms a data

Physical Network Model	
c_{ν_i}	Capacity of host ν_i
$mips_{\nu_i}$	Processing speed of host ν_i
q_{ν_i}	run queue length
Operator Graph Model	
τ	Minimal discrete data unit to be transmitted between the operators
$\mathcal{P}(\omega_k, \nu)$	Processing delay that a data unit experiences at host ν hosting operator ω_k.
$\mathcal{T}(\tau, \nu)$	Transmission delay for putting a data unit on the wire at host ν
$\mathcal{LP}(\overline{\omega_i \omega_j})$	Total delay of an end-to-end path $\overline{\omega_i \omega_j}$, including processing and network delays
s_τ	Size of data unit

Table 3.4: Extended Processing and Network Delay Constrained System Model

stream. We define s_τ as the size of a data unit τ. A data unit forms the basic unit of processing for each operator. Typically, sources generate sequences of data units in intervals that are then processed by operators and finally consumed by the sinks.

Since, in this section, we consider that the end-to-end delay contains processing as well as network delays, we extend the system model presented in Section 3.2.1 accordingly. In particular, $\mathcal{T}(\tau, \nu)$ defines the *transmission delay* for putting a data unit on the wire at host ν. In order to estimate $\mathcal{T}(\tau, \nu)$, we continuously measure the transmission delay $\mathcal{T}(\tau', \nu)$ of a real data unit τ' (probe unit) of size s'_τ on a physical host ν and we calculate the transmission delay as $\mathcal{T}(\tau, \nu) = (s_\tau / s'_\tau)\mathcal{T}(\tau', \nu)$.

Moreover, to be able to estimate the processing delay on a physical host, we introduce here a simple processing model that assumes that the processing

power of a host is equally distributed to all operators running on this host (which is typically the case if all processes have the same priority). More formally, we define as $\mathcal{P}(\omega_k, \nu)$ the *processing delay* that a data unit experiences at host ν hosting operator ω_k. For the local host where the operator is currently located, $\mathcal{P}(\omega_k, \nu)$ can be measured directly. However, determining $\mathcal{P}(\omega_k, \nu)$ is not a trivial task for other hosts where the operator is currently not located. Note that the placement algorithm needs information about $\mathcal{P}(\omega_k, \nu)$ *before* it actually places the operator on host ν to make a decision which host is suitable with respect to processing delay before actually migrating an operator. Therefore, the basic problem is to *estimate* the processing delay of an operator ω when executed on host ν taking into consideration the fact that hosts have dynamic processing load and different processing power. Here, we use a simple model for estimating the processing delay, which provides a sufficiently good estimation according to the results presented later in this section. However, our approach is open to other more sophisticated performance models, based, for instance on black box or white box tests and more on elaborate machine models [62, 104].

Our estimation is based on two metrics to define the processing power and load of each host, respectively. On the one hand, we use the *bogomips* metric to define the speed of a machine [108]. Bogomips express the number of iterations per second of a loop with empty body. It is used, for instance, by the Linux system at the beginning of the boot process. Obviously, this metric cannot capture every aspect of the speed of a host such as different relative speeds for integers and floating point operations. However, it gives a coarse estimate to compare two machines and proved to be sufficiently accurate for our purpose in our measurements. On the other hand, we use the *run queue length* of the processor to express the load of a host. The run queue length defines the number of processes waiting for the CPU. Intuitively, the share of processing time an operator receives will shrink proportional to the number of processes running on the host (here, a process can be another operator as well as any other process running on the host).

Assume that the operator is currently running on host ν_i and we want to estimate the processing delay of that operator if it migrates to host ν_j. The current capacity c_{ν_i} of host ν_i with processing speed $mips_{\nu_i}$ and run queue length q_{ν_i} is given by the following formula:

$$c_{\nu_i} = \min\{mips_{\nu_i}, \frac{mips_{\nu_i}}{q_{\nu_i}}\}$$

The capacity of the other host is given by:

$$c_{\nu_j} = \min\{mips_{\nu_j}, \frac{mips_{\nu_j}}{q_{\nu_j} + 1}\}$$

Here, $mips/q$ defines the bogomips that one process receives if q processes are competing for the CPU. On host ν_i where the operator is currently placed, q_{ν_i} already includes the operator. On the (candidate) host ν_j we have to add 1 to q_{ν_j} to reflect the queue size after the migration to ν_j. The minimum function ensures that on an unloaded host and short processing times with longer idle periods between data units the operator cannot receive more than 100% of the CPU. As an indicator of the current relative performance of the two hosts we define the *speedup factor*: $speedup_{ij} = \frac{c_{\nu_i}}{c_{\nu_j}}$. Finally, we approximate the remote time to run the operator on host ν_j as the product of the speedup factor and the local processing time at host ν_i:

$$\mathcal{P}(\omega, \nu_i) = speedup_{ij} * \mathcal{P}(\omega, \nu_j)$$

In order to evaluate our processing model, we ran several experiments where we tried to estimate the processing time of an operator, given that different numbers of operators were already deployed at the candidate host. For these experiments, we considered operators that realize a matrix multiplication with different matrix sizes. Moreover, we approximated the run queue length by using an exponential moving average with a smooth factor equal to 0.05, using the system activity report (sar) command to query the current value of the run queue length of a Linux system. Figure 3.9 and Figure 3.8 show the approximated and the real processing time for operators with different matrix

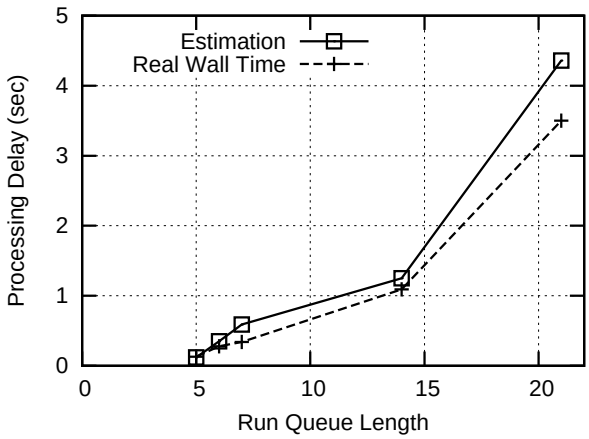


Figure 3.8: Estimated processing delay (matrix multiplication operator; matrix size:100)

size. As expected, the processing time increases by the increase of the run queue length and the model captures correctly the tendency of the processing delay. For operators with matrice size 100, the average relative error equals to 20%, while for operators with matrix size 1000, the corresponding average relative error is lower than 11.2%.

3.3.2 Problem Statement

Before we formulate our constrained optimization problem, we define here the end-to-end latency $\mathcal{LP}(\overline{\omega_i\omega_j})$ of an end-to-end path $\overline{\omega_i\omega_j}$. More formally, extending the definition of end-to-end latency from Section 3.2.2, end-to-end latency is defined as the time that a data unit τ needs to get transmitted and processed along a path between source ω_i and sink ω_j:

$$\mathcal{LP}(\tau, \overline{\omega_i\omega_j}) =$$

$$\sum_{\omega_k\omega_l \in \overline{\omega_i\omega_j}} \{\mathcal{L}(\overline{\nu_k\nu_l}) + \mathcal{T}(\tau, \nu_k) + \mathcal{T}(\tau, \nu_l)\} + \sum_{\omega_k \in \overline{\omega_i\omega_j}} \mathcal{P}(\omega_k, \nu_k)$$

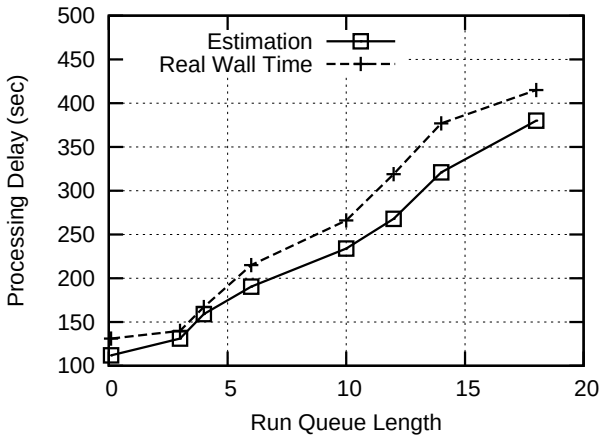


Figure 3.9: Estimated processing delay (matrix multiplication operator; matrix size:1000)

In this equation,as already introduced in the previous sections, $\mathcal{L}(\overline{\nu_k\nu_l})$ defines the *communication link delay* of a link $(\overline{\nu_k\nu_l})$ on the path, i.e., the time it takes to transmit a single bit between the two physical hosts, hosting operator ω_k and ω_l. Similar to the previous sections, for modelling the propagation delay, we use the latency space model as already introduced in Subsection 3.1.1.

As already mentioned, in this section we consider the processing delays $\mathcal{T}(\tau,\nu)$ and $\mathcal{P}(\omega_k,\nu)$ to contribute significantly in the end-to-end delay. Based on the previous definitions of $\mathcal{T}(\tau,\nu)$ and $\mathcal{P}(\omega_k,\nu)$, we can define the latency of an operator graph as the maximum end-to-end latency contained in operator graph $\mathcal{G}$, i.e. the maximum latency that a tuple experiences traversing the longest path in the operator graph. Formally speaking, the latency of an operator graph $\mathcal{G}$ is defined by:

$$\mathcal{LP}(\mathcal{G}) = \max_{\overline{\omega_i\omega_j}\in\mathcal{Q}_{S\to A}} \mathcal{LP}(\tau\overline{\omega_i\omega_j}) =$$

$$= \max_{\overline{\omega_i\omega_j}\in\mathcal{Q}_{S\to A}} \sum_{\overline{\omega_k\omega_l}\in\overline{\omega_i\omega_j}} \{\mathcal{L}(\overline{\nu_k\nu_l}) + \mathcal{T}(\tau,\nu_k) + \mathcal{T}(\tau,\nu_l)\} + \sum_{\omega_k\in\overline{\omega_i\omega_j}} \mathcal{P}(\omega_k,\nu_k)$$

Based on the end-to-end latency and network usage definitions, we can now formally define our placement problem. This problem is defined as constrained optimization problem where a user defined maximum end-to-end latency restriction lp_{max} has to be fulfilled while minimizing the induced network usage:

$$\min \mathcal{U}_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}), \text{ subject to } \mathcal{LP}(\mathcal{G}) \leq lp_{max} \tag{3.21}$$

3.3.3 Placement Algorithm

In this section, we present the operator placement algorithm to solve the above constrained optimization problem. We start with an overview of the algorithm, and then present further details in the following subsection.

3.3.3.1 Overview of Algorithm

The basic idea of the algorithm is similar to the constraint placement algorithm presented in Subsection 3.2.3. Again, we use a two-step placement process. In the optimization step, we search for an optimal placement w.r.t. network usage. In the second step, we modify this unconstrained solution such that the end-to-end latency constraint is satisfied and the network usage is only increased as few as possible compared to the unconstrained solution. Unlike the problem introduced in Section 3.2.2, the problem of Equ. 3.21, includes processing and transmission delays. By moving the operators in the latency space as proposed in Section 3.2.3, we can reduce the communication latency only. Thus, for solving the extended problem of Equ. 3.21, we need a strategy that considers also processing and transmission delay.

Intuitively, to reach a better solution, the operators should be placed on hosts that reduce the end-to-end latency, either by moving to faster nodes (reducing processing delay) or by reducing the network latency. Theoretically, we could find the optimal solution of the constraint placement problem by an exhaustive search that considers every host in the system. However,

obviously this would lead to high overhead for larger sets of hosts and operators. Therefore our solution is based on the idea to find some *candidate* hosts that reduce the end-to-end latency. We find promising nodes by searching in certain areas of the latency space—later we will show in detail how to find a good set of candidates. Then, we communicate with the candidates to get their processing and transmission delay. Finally as we see later, in order to keep the network usage as low as possible, we iterate over the candidate nodes and we select those that reduce the end-to-end latency while increasing the network usage minimally.

Depending on the phase, the output of the constraint satisfaction algorithm will be either an initial placement or a new placement of the operators. In the later case, the operators are migrated to the new hosts.

3.3.3.2 Constraint Satisfaction Algorithm (MOPA–LPMAX)

Next, we describe the details of the constraint satisfaction step. As mentioned, the constraint satisfaction algorithm, to which we refer as MOPA-LPMAX, depicted in Algorithm 7 is invoked after the optimization step. Therefore, before the execution of this algorithm all operators are placed on hosts such that Equation 3.4 is minimal. For the explanations below, it is important to realize that $\mathcal{U}(\mathcal{G})$ is a function that depends on the coordinates of the hosts hosting operators in the latency space since the Euclidean distance between hosts in the latency space defines the propagation delay (Function $\mathcal{L}$) between hosts and therefore their operators. In the beginning, $\mathcal{U}(\mathcal{G}) = \mathcal{U}_{\text{min}}$ where $\mathcal{U}_{\text{min}}$ denotes the minimal network usage, which is found by the optimization step.

However, although $\mathcal{U}(\mathcal{G})$ is minimal after the optimization step, the latency of the longest path of the graph might be higher than the requested maximum latency, i.e., Equation 3.21 is not fulfilled in general. Algorithm 5 now tries to distort this optimal solution to stay as close as possible to $\mathcal{U}_{\text{min}}$ and fulfill the latency constraint.

Algorithm 7 MOPA-LPMAX Algorithm

Require: $\mathcal{U}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal

Ensure: Finds a mapping $(\nu_1, \ldots, \nu_n)$ such that $\mathcal{L}(\mathcal{G}) \leq l_{max}$ and $\mathcal{U}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal

1: **while** $(\mathcal{LP}(\mathcal{G}) > lp_{max})$ **do**
2: find maximum latency path $\overline{\omega_i\omega_j}$
3: **if** candidate set $candidates(\overline{\omega_i\omega_j})$ does not exist **then**
4: **for all** operator $\omega \in \overline{\omega_i\omega_j}$ **do**
5: find candidate set $candidates(\omega)$
6: sort $candidates(\omega)$ by distance to $\mathcal{U}_{\text{min}}$
7: $candidates(\overline{\omega_i\omega_j}) \leftarrow candidates(\overline{\omega_i\omega_j}) \cup candidates(\omega)$
8: **end for**
9: **end if**
10: **if** $candidates(\overline{\omega_i\omega_j}) = \emptyset$ **then** {already at latency minimum}
11: notify application
12: **else**
13: **for all** operator $\omega \in \overline{\omega_i\omega_j}$ **do**
14: get next candidate ν' in $candidates(\omega)$
15: $\vec{x}'_(\omega) \leftarrow \vec{x}'_\nu$
16: $\Delta\mathcal{U} \leftarrow \mathcal{U}(\vec{x}'_\omega) - \mathcal{U}(\vec{x}_\omega)$
17: **end for**
18: **end if**
19: assign operator ω with minimal $\Delta\mathcal{U}$ to ν'
20: delete candidate ν' from $candidates(\omega)$
21: delete candidate ν' from $candidates(\overline{\omega_i\omega_j})$
22: **end while**
23: **return** current mapping $(\nu_1, \ldots, \nu_n)$

Algorithm 5 gets as input an initial mapping of the operators to hosts such that the network usage of the operator graph is minimal. First, the algorithm finds the longest path in the operator graph, and checks if the latency restriction is already fulfilled(line 1). In that case, it simply returns the current mapping. Otherwise, it enters the main body of the algorithm, where it checks for alternative mappings.

For each operator on the longest path, the algorithm finds a set of candidate hosts where the operator could be migrated to (line 5). The candidate set is calculated once in the beginning for each operator on a path (line 3-9). The candidates are selected such that moving an operator to a candidate host decreases the latency of the longest path. The calculation of the candidate set includes networks delays as well as estimated processing delays (in the next subsection, we are going to discuss in detail how this candidate set is determined). If the candidate set of all operators on the maximum delay path is empty, the latency cannot be decreased any further and the algorithm stops without finding a valid solution (line 10). In this case, the application is notified that the latency constraint cannot be fulfilled, and the application might choose to decrease its requirements or simply stop (line 11). If the candidate set is not empty, the latency can be further decreased by migrating to any candidate host. The idea is, not to choose an arbitrary candidate but a candidate that increases the network usage the least in order to distort the optimal solution w.r.t. to network usage the least. To this end, the hosts of the candidate set are sorted according to the distance to $\mathcal{U}_{\text{min}}$ (line 6), and the host with the minimal distance leading to the minimal network usage increase $\Delta\mathcal{U}(\omega)$(line 13-19) is chosen as new host for operator ω.

This process is continued until either the candidate set is empty, i.e., the latency cannot be further decreased (see above), or the latency constraint is fulfilled. In the later case, the mapping of operators $(\omega_1, \ldots, \omega_n)$ to hosts $(\nu_1, \ldots, \nu_n)$ respectively is returned, and the operators are migrated to these hosts.

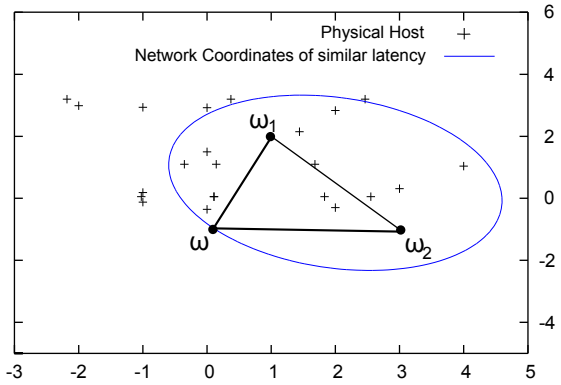


Figure 3.10: Candidate set for one unpinned operator with one sink and one source.

3.3.3.3 Selection of Candidates

Calculating the candidate set is a crucial operation during the constraint satisfaction step. If the candidate set is too big, the overhead increases since every candidate has to be contacted and checked with respect to its processing and network delay. If the candidate set is small and misses some valid hosts that would decrease latency, no valid solution might be found although it exists in the network. In order to find a good trade-off between overhead and success rate, we considered different candidate selection strategies, which are described next.

First, we introduce a selection strategy that uses an optimal restriction of the search space. Next, we illustrate this idea through a simple example and we prove a pruning criterion that reduces the search space further. It is important to observe that all suitable candidates are restricted inside ellipsoidal shapes in the Euclidean latency space. Figure 1 visualizes a simple example of an operator ω with one source ω_i and one sink ω_j. The end-to-end delay $\mathcal{LP}$ for this simple example is the sum of the propagation delays of the operator to its neighbours[5] $\mathcal{L} = \mathcal{L}(\omega\omega_1) + \mathcal{L}(\omega\omega_2)$, plus the processing

[5]Without loss of generality, we assume here that the sink and the source have no

Algorithm 8 Candidate Selection Algorithm

Require: Bounding box for ellipse E

Ensure: Candidate set *candidates* of size k

```
1: find all hosts hosts inside ellipse E [range query]
2: sort hosts hosts by distance to U_min
3: while #candidates < k do
4:    contact next host ν' in hosts
5:    if P(ω, ν') < P(ω, ν) + T + N − N_min then
6:       candidates ← ν'
7:    end if
8: end while
9: return candidates
```

delay $\mathcal{P} = \mathcal{P}(\omega, \nu)$ at the host ν of operator ω, and the transmission delays $\mathcal{T} = \mathcal{T}(\tau, \nu) + \mathcal{T}(\tau', \nu)$ of the input tuple τ and output tuple τ'. Each value of the end-to-end delay $\mathcal{LP}$ represents an ellipse in the latency space with foci points defined by the positions of the two neighbours ω_i and ω_j. Note that according to the definition of the ellipse, all points on an ellipse have the same distance to the foci points, i.e., they lead to the same end-to-end delay.

It is straightforward to see that only the nodes that reside inside the ellipse $\mathcal{LP} = \mathcal{L} + \mathcal{P} + \mathcal{T}$, can lead to better solutions since for nodes outside the ellipse even if the processing delay is zero, the network delay would still exceed the current latency $\mathcal{LP}$. Thus, the candidate nodes are restricted inside the ellipse $\mathcal{LP}$. In order to find the candidate hosts within E, we could perform a range query in the latency space using the latency space service and query range $\mathcal{LP}$.

Although checking all nodes inside the ellipse includes all valid candidates, it might lead to very high communication overhead if the set of enclosed nodes within the ellipse is large. As we discuss in Chapter 4, the set of enclosed nodes, is related to the difference between the network and processing delay.

processing delay

If processing delay is in order of seconds, then checking of all nodes inside the ellipse might lead to an exhaustive search that makes the application of such method impossible in practice. Thus, in order to strictly limit the candidate nodes to a reasonable size, we should select only k hosts among all the nodes in the ellipse to contact. To this end, in this section we discuss possible heuristics to be used to prune the search space that will be evaluated in Chapter 4.

Straightforward solutions to that problem are to choose the k closest nodes with respect to network usage minimum that reside in the ellipse, or to select k random hosts inside the ellipse. Random selection of hosts could be beneficial in case the suitable hosts do not lie in the direct vicinity of the current host. However, these heuristics do not consider the value of the total delay during the selection of the candidates, which is important to identify promising candidate nodes. Therefore, we propose a method that uses a pruning criterion, which filters out some of the nodes inside the search space. To this end, we introduce the following pruning criterion for the processing delay of the candidate hosts:

Pruning Criterion Let ω be an operator placed on a host ν with communication latency $\mathcal{L}$ to two neighbouring operators and with processing delay $\mathcal{P}$. Assume also a data unit τ with transmission delay $\mathcal{T}$. A host ν' can only lead to a better solution than that of ν w.r.t. latency, if and only if the following condition is fulfilled: $\mathcal{P}(\omega,\nu') < \mathcal{P}(\omega,\nu)+\mathcal{T}+\mathcal{L}-\mathcal{L}_{\min}$, where $\mathcal{L}_{\min}$ represents the minimum network delay of operator ω to its two neighbours.

Proof. Assume that the total delay on host ν' is equal to $\mathcal{P}(\omega,\nu')+\mathcal{L}'+\mathcal{T}'$. If ν' is a better candidate host, it should hold that $\mathcal{LP}' < \mathcal{LP}$, thus the following inequality should hold:

$$\begin{aligned} \mathcal{P}(\omega,\nu') + \mathcal{L}' + \mathcal{T}' < \mathcal{P}(\omega,\nu) + \mathcal{L} + \mathcal{T} \Rightarrow \\ \mathcal{P}(\omega,\nu') < \mathcal{P}(\omega,\nu) + \mathcal{L} + \mathcal{T} - (\mathcal{L}' + \mathcal{T}') \end{aligned} \tag{3.22}$$

We can find a maximum bound for the equation by minimizing $\mathcal{L}'+\mathcal{T}'$. Network latency is minimized when the host ν' lies on the line segment between

the two neighbouring operators ω_1 and ω_2, leading to a minimum possible network delay $\mathcal{L}_{\min}$. If we also assume that ν' has negligible transmission delay, i.e., $\mathcal{T} = 0$, we get a minimum bound for $\mathcal{L}' + \mathcal{T}'$:

$$\mathcal{L}' + \mathcal{T}' < \mathcal{L}_{\min} \tag{3.23}$$

From Eq. 3.22 and Eq. 3.23, we finally get:

$$\mathcal{P}(\omega, \nu') < \mathcal{P}(\omega, \nu) + \mathcal{L} + \mathcal{T} - (\mathcal{L}' + \mathcal{T}') < \mathcal{P}(\omega, \nu) + \mathcal{L} + \mathcal{T} - \mathcal{L}_{\min} \Rightarrow$$

$$\mathcal{P}(\omega, \nu') < \mathcal{P}(\omega, \nu) + \mathcal{T} + \mathcal{L} - \mathcal{L}_{\min}$$

, which proves the pruning criterion. ■

Algorithm 8 shows the pseudocode for the candidate selection strategy using the pruning criterion. According to this method, we first get all the hosts that reside in the ellipse by performing a range query on the latency space. Then we contact one by one the next nearest host with respect to network usage minimum inside the ellipse and we check if it satisfies the pruning criterion. In that case, the host is included in the candidate set. The process is repeated until k hosts that satisfy the pruning criterion are found. Obviously, this method induces higher overhead, than the naive solutions proposed earlier, but it is expected to give better quality results, since it takes also into consideration the pruning criterion. Although, the criterion is likely to return less nodes, it still does not strictly limit the number of returned candidates, since the selectivity of the filter depends on the speed of the current host. Thus, if the current host is quite fast, the criterion tends to filter out more hosts, while in case of a slow current host, less candidate hosts will be filtered out.

In Chapter 4, we are going to provide an evaluation of the proposed heuristic using the pruning criterion, compared to simple heuristics, i.e., Random, and, k-Nearest Neighbour selection as discussed earlier.

System	Optimization Objectives/ Constraints
Padres [69]	Routing delay, Network traffic
FAIDECS [111]	Throughput, Latency
Hermes [86]	Bandwidth, Latency, Reliability, Load
Cordies [59]	Stability, Application constraints
DHCEP [99]	Network usage, System/Application Constraints

Table 3.5: Existing CEP systems supporting distributed event recognition

3.4 Related Work

In previous chapter, we have provided the related work in the field of context-aware, distributed stream processing and complex event processing systems and we have identified similarities in the architectural approach for processing streams of data. In-network processing that is based on a fully distributed model, i.e., an overlay network of processing operators, is mainly used in the IFP systems but as analysed in Chapter 2, there is a potential in using this model for increasing the scalability of context-management systems. Therefore, in this section we discuss the strategies for the operator placement problem that has been investigated in different contexts, i.e., as part of data stream management or CEP systems but also a relevant approach from control systems.

To this end, this section is structured in three different subsections depending on the targeted system, namely complex event processing, distributed stream processing, control systems. As we analyse later, each approach focuses on different aspects of the placement problem, trying to fulfil different constraints.

3.4.1 Complex Event Processing

Before we present in detail the existing work in this field, we briefly discuss the relevance of the operator placement problem to the CEP systems. As already mentioned in Chapter 2, CEP systems process flows of events in an effort to detect and forward (composite) events to interesting peers (subscribers). CEP systems rely on Pub/Sub systems that connect the Publishers, producing primitive events to the Subscribers that consume events. In that respect, similar to the operator graph model, CEP tasks form an overlay network with a set of sources (publishers), a set of sinks (subscribers) and possibly a set of in-network event correlators that generate composite events by aggregating primitive events. In that respect, the position of the physical node(s) that the event composition is performed is similar to the general operator placement problem.

In the field of CEP systems, several systems that allow for distributed event detection have been proposed (FAIDECS [111], Siena [25], Hermes [86], Gryphon [7], Padres [69]). Early works in CEP systems allow for the subscription to basic primitive events and do not consider in-network aggregation of events. These systems rely mostly on a network of broker nodes, which perform matching between advertisements and subscriptions and forward the events accordingly. In that respect, systems, such as SIENA [25] and Gryphon [7] have focused on the efficient routing of primitive events by reducing the communication costs between clients and brokers and thus avoiding the flooding of events to all subscribers. However, these works do not consider placement of complex event correlators since they focus mainly on the filtering and routing of primitive events and leave the aggregation and composition of events to the application programmer.

Closer to our work, are CEP systems that allow for the aggregation and composition of events and consider the placement of event correlators. Table 3.5 provides an overview of existing CEP systems, that we discuss in the next paragraphs, which consider placement problems with respect to their

main optimization goals.

For instance, PADRES [69] use rule-based brokers that are capable of composing atomic events to complex composite events. The event composition is performed on rule-based brokers that are preferably close to publishers. Although the heuristic strategy of placing rule-based brokers (that represent in-network operators in our model) close to the publishers reduces the communication overhead since the events are filtered close to the sources, it does not lead to optimal placement decisions with respect to network usage optimization goal.

In another work, Hermes [87] provides a set of heuristic solutions called distribution policies for the placement of mobile complex event (CE) detectors in the network. Initially, Hermes [86] used a DHT (Distributed Hash Table) to determine the rendezvous nodes that perform the in-network composition of events between publishers and subscribers. Then, in [87] an extension of Hermes framework was presented that incorporates new distribution policies. In particular, in [87] authors propose five different distribution policies that optimize different metrics such as bandwidth consumption, latency, load, reliability, and stability. Each distribution policy depending on the objective takes advantage of the decomposition, re-use and locality of CE detectors. In their evaluation, they show that by applying these simple heuristics, they could reduce the communication overhead especially in the part of the wireless network since the CE detectors could be reused. The proposed heuristics are based on the decomposition, reuse and locality of the CE detectors and they do not use any network- and system-specific information, e.g., network or computing capacity information, that is necessary in order to take good placement decisions. Our placement algorithms use network- and system-specific information by incorporating information by the latency space and the processing model presented in Subsection 3.3.1.

FAIDECS [111] considers also composite events and applies a broadcast algorithm for effectively sending all related events to the interested subscribers. The broadcast strategy is based on Hermes approach [86], which uses DHT

to determine rendezvous nodes (mergers) for publishers and subscribers. In addition to this approach, FAIDECS proposes the replication of mergers to increase the availability. However, the selection of merger nodes is based on DHT nodes that are not always optimal for reducing application-related properties such as latency, since they are designed to reduce the number of hops.

Relevant to our work, is also the approach of Koch et al. [59] that adopts also an operator-driven distribution for CEP systems. In particular, the authors present Cordies [59], a novel CEP system that enables efficient distributed event correlation. Cordies uses an expressive language for implementing CE operators and enables distributed event correlation through Correlation Description (CD) placement. Cordies is able to integrate user-defined placement algorithms. In their work, they formulate the placement problem as a constraint satisfaction problem (CSP) and they propose a heuristic solution that solves this problem. Their approach uses the application constraints to prune the search space of candidate physical hosts and optimizes the placement for stability. In [99], Schilling et al. have also proposed a placement algorithm for CEP systems. In their work, they assume a heterogeneous network of physical hosts that limit significantly the search space for the placement decisions. Therefore, they first find a valid initial placement that they optimize after deployment for network usage. During the optimization phase, the algorithm uses a simulation annealing technique to find alternative better solutions based on its local knowledge. Although this approach optimizes also for network usage, this work is based on different assumptions since the proposed placement algorithm tries to find initially a set of feasible solutions and subsequently optimizes the placement with respect to network usage based on local knowledge.

To summarize, distribution policies have been in the focus of several works in CEP systems. However, these works were based on different system models, i.e. assuming an overlay network of specific physical nodes, called brokers, that are capable of performing the event composition. To this end, the pro-

posed placement algorithms focus mainly on the satisfaction of the application and system constraints and the identification of a set of feasible solutions. In our model, we address the optimization problem given a large search space of physical hosts that are capable of hosting operators.

3.4.2 Data Stream Processing

In-network processing data has been applied in several distributed data stream processing systems. Placement strategies vary both in terms of their system model and optimization goal. In particular, system model may consider mobile nodes that are linked via wireless links and/or Internet topology-like networks that communicate via Internet links. Each of the system model, has different properties (e.g. in terms of energy consumption, reliability of links and nodes) that lead to different placement strategies. Therefore in this section, we distinguish between operator placement algorithms for Infrastructure-based systems considering infrastructure nodes as physical hosts and placement strategies for wireless and ad-hoc networks. Since our work focuses on the Wide Area Networks (WAN), where physical nodes communicate via Internet links, we provide a detailed overview of existing algorithms for these systems and then we briefly discuss placement strategies supporting mobility of nodes.

3.4.2.1 Operator Placement in Infrastructure-based Systems

Lakshmanan et al. [63] provide a comprehensive overview of existing operator placement algorithms for large-scale scenarios. Their study show that the diversity of optimization goals leads to different placement algorithms. According to this work, popular optimization goals for data stream processing include load, latency, bandwidth, system constraints, and operator importance optimizations. Triggered by this work, we have clustered the related work in four main categories, i.e., *network usage*, *latency and other constraint optimization*, *load balancing*, and *availability*, which correspond to main op-

timization goal considered by the placement strategies in the cluster.

3.4.2.1.1 Network Usage Optimization Ahmad et al. [8] at first proposed an approach for operator placement optimizing the bandwidth-delay product. With this approach, nodes are chosen that lie on the paths between two endpoints of a DHT-based overlay network. However, in [85] Pietzuch et al. showed that looking for candidate nodes on DHT paths leads to a poor approximation of the optimal solution since the actual goal of the DHT routing tables is to minimize the number of hops rather than network usage.

Closest to our optimization algorithm is the work of Pietzuch et al. [84], who were the first to propose the usage of the latency space as an intermediate continuous search space for operator placement problem. In their approach, called SBON, the operator placement in the latency space is based on a physical model of springs. The goal of the proposed algorithm is to minimize the overall energy of the corresponding physical system. However, in this model energy is proportional to the square of the latency while the network usage is only linear dependent on the latency. In other words, SBON optimizes the metric bandwidth $\times$ delay^2, which does not intuitively model network usage. For instance by doubling the length of a physical path between two operators, the number of bits in transit on this path is only doubled rather than quadrupled. In contrast, MOPA, presented as a solution to the network usage optimization problem, actually optimizes bandwidth $\times$ delay, while MOPA-LMAX provides a trade-off solution between network usage and network latency. Moreover, our algorithm fully exploits the locality of the problem by finding at each iteration the current local optimal solution, while SBON uses another model, which gradually moves at each iteration towards the local optimum. Our evaluations presented in 4show that our algorithm outperforms SBON not only in the quality of optimization results, but also in terms of the communication and operator migration overhead induced by the placement algorithm.

3.4.2.1.2 Latency and other Constraint Optimization Next, we describe methods that consider other performance metrics which directly or implicitly optimize for latency or other application specific QoS metrics. In more detail, some approaches combine latency guarantees with load balancing. Gu et al. presented an algorithm that uses global knowledge to checks exhaustively all hosts in order to identify some candidate hosts [52]. Then, it selects the hosts that minimize a congestion aggregation metric modelling the processing and network residual resources. Such an optimization metric can be useful for cases where the network is heavily loaded, but it is less efficient for other situations. Moreover this work assumes global knowledge of the network conditions which is not always a realistic assumption.

A decentralized approach for operator placement has been presented in [118]. The authors propose a decentralized solution that enables the local cooperation of the nodes to optimize the so-called performance ratio, which models the relative performance of a query, i.e. the end-to-end latency of a data unit divided by the inherit complexity of the query. The proposed solution uses local knowledge to find a solution for the optimization problem. One important assumption considered in this problem, is that the nodes are interconnected by a local network. For our placement problems we consider that physical nodes are interconnected in a WAN (wide area network) via Internet. Therefore, given our system model the solution proposed in [118] may be trapped in local optima, since nodes seek for better placements in close vicinity in terms of geographical proximity that might not be proportional to network delay.

In [11], Amini et al. introduced a placement problem which optimizes for the weighted throughput, which is an indicator of the total productive work done by the system. The proposed approach, called ACES (Adaptive Control for Extremescale Stream processing systems), is a two-tiered approach for adaptive, distributed resource control. In more detail, the first tier optimizes the placement of operators onto physical hosts to maximize weighted throughput, while the second tier configures the input and output rates such

that they adapt to varying incoming load. This solution does not consider any latency constraint and it has a different optimization goal with respect to network usage optimization.

NexusDS [32] focuses on the fulfillment of the QoS application constraints, e.g., bandwidth, latency, reliability requirements. The proposed approach is based on six subsequent steps: Conflation, Early prune, Graph Assembly, Ranking, Mapping and, Execution. During conflation phase, adjacent nodes in the operator graph are merged to create virtual nodes. Early prune finds promising candidate nodes and links that fulfill the QoS constraints. Then, during graph assembly, the set of feasible solutions is being identified by combining nodes and edges that fulfill the application criteria. Finally a score value for each QoS property is considered to rank the feasible solutions and find the most appropriate one that better fits QoS constraints. The placement problem considered in this paper differs compared to ours in the sense that it does not optimize for network usage but the solution is determined mainly by the application constraints. Moreover, this work focuses on the discovery of an initial placement of operators and it does not address the problem of continuous adaptation of operator placement.

3.4.2.1.3 Load Balancing Optimization Some initial works in the operator placement problem focuses on load balancing techniques that optimize for fair load distribution across the different physical nodes. Flux [102] has proposed a load balancing scheme for continuous queries. In their proposed solution, they use a central controller to monitor the load of the computing nodes and make load balancing decisions. Our work uses a distributed resource lookup and does not consider global system knowledge. Borealis [114] has solved another operator placement problem. In their work, they optimize the time correlations among different operators in an effort to distribute load fairly among servers. However in both of these works, network resources are considered abundant and the network costs are not part of the optimization strategy.

Unlike the previous works [102] [114], in our approach we do not try to optimize directly for load balancing. Load balancing strives to distribute as fairly as possible the operators on physical nodes such that the computational load is balanced. As motivated earlier, our primary optimization goal is to minimize the network load and thus make the network more scalable. In MOPA-LPMAX algorithm, we consider the computational load implicitly by integrating the processing delay to the end-to-end delay. If the processing delay gets significantly large -possibly because of an overload situation on a node-, then the placement will prefer less loaded nodes. Thus, it makes an indirect load balancing, only when this is necessary, but it can allow load unfairness as long as the latency constraints are not violated.

3.4.2.1.4 Availability Optimization Other placement algorithms, try to optimize for availability and exploit operator re-use [90] [18]. Repantis et al. [90] have proposed a placement algorithm that maximizes availability while fulfilling bandwidth limitations. Their approach uses a distributed placement algorithm that discovers a set of candidate nodes and rank them in decreasing latency. The goal is to provide high available distributed data stream processing. Benzing et al. [18] have proposed a system that allows operator reuse and provides flexible data stream retrieval in different resolutions. Although our model allows the re-use of operators, it does not explicitly consider the optimization problem for larger operator networks that are generated by the merging of different operator graphs. In that respect, the investigation of methods and algorithms that provide high availability is out of the scope of the placement problems and algorithms presented in this dissertation.

3.4.2.2 Operator Placement Considering Wireless Communication

In this paragraph, we discuss placement algorithms that consider also mobile nodes connected via a wireless network. Mobile nodes are typically energy-constraint, which imposes additional placement restrictions in the system level.

Closer to our work, are strategies that consider both wired and wireless communication. For instance, in [105], the authors solve an operator placement problem, which considers network transmission delays and energy consumption, based on a hierarchical system model of physical nodes with increasing computing capacity. This system model implies a heterogeneous underlying network that may consist of mobile as well as infrastructure nodes. Ying et al. [117] have formulated the operator placement and intermediate data caching problem to minimize an aggregated cost based on computation, communication and storage costs. The authors present distributed algorithms that solve the problem assuming a sensor network of diameter L_{max}.

Our algorithms could be used to support heterogeneous, including infrastructure and mobile nodes. However, since our initial goal is to design placement algorithm for Infrastructure networks, a study on the performance of this algorithm and possible extensions for supporting these mobile scenarios is out of the scope of this dissertation.

Finally, other approaches in wireless sensor networks consider energy efficiency [103, 115] and bandwidth constraints [39] under quality constraints on the accuracy of the query results. The consideration of data accuracy is another aspect, usually considered in the wireless sensor networks, that is not part of our system model. As already presented, in our approach, we consider operator placement to be the only degree of freedom for our optimization, without taking into consideration other possible changes in the structure and semantics of the operators, e.g. semantics of operators, control of input/output data rates that could change the accuracy of the results.

3.4.3 Control Systems

Operator placement problem has been recently investigated in the context of control systems, e.g., plant control networks, where monitored data have to be communicated to control processes running on different physical nodes. Finding optimal placement for controllers, resembles the problem of finding

optimal placement of operators over a network of physical nodes.

In that respect, Carabelli et al. [24] have been motivated by our problem formulation, to investigate centralized techniques based on integer linear programming that solve a variation of discrete multi-operator placement problem exactly. Their ILP formulation solves a routing problem (shortest path problem) instead of adopting a subgraph isomorphism formulation as we did for the ILP formulation in Subsection3.1.4. In that respect, [24] makes strong assumptions on the execution environment. First, it assumes availability of underlay network topology to define shortest paths in underlay and finds an optimal placement with respect to the underlay making a deployment in today's network infrastructures more complex. Our assumptions are much weaker. Approach could be deployed already in today's Internet infrastructure as an overlay network. In contrast, we strive for an overlay network approach applicable to today's internet infrastructure. Secondly, [24] does not consider adaptation and assumes unrestricted bandwidth. Our approach indirectly considers bandwidth restrictions through latency space. When links become overloaded, the latency increases due to longer queues in routers. This in turn increases the delay-bandwidth product.

Moreover, in their proposed solution, all data has to be collected at a central node, called a placement controller. This centralized approach potentially puts high stress on the central node and its links, in particular, for larger operator graphs and/or dynamic state. As in any centralized solution, fault tolerance becomes an issue, since the central node is a single point of failure. Our approach for solving the unconstrained optimization problem (MOPA) provides a lightweight heuristic solution that is executed in a distributed way, i.e., information only has to be exchanged between neighbours in operator graph (local communication only).

4 Evaluation

In this chapter, we discuss the experimental results of the placement algorithms solving the three placement problems presented in the previous chapter. For the evaluation of the algorithms, we have used two different methods, namely simulation and emulation. In particular, we have used PeerSim as a network simulator in order to evaluate the MOPA and MOPA-LMAX algorithms, which solve the unconstrained optimization and the network latency constraint optimization problems respectively. Then, we have used the NET emulator [48] developed at the University of Stuttgart to test the MOPA-LPMAX algorithm, which solves the general constrained optimization problem which considers also processing delays. The reason we have selected a different evaluation tool for the MOPA-LPMAX algorithm is that since MOPA-LPMAX considers processing delays, a *real* system under test is necessary to provide real measurements on the processing delays. This environment can be provided by an emulation test bed such as the NET emulator.

Table 4.1 shows an overview of the algorithm under test, the reference algorithms and the evaluation environment. In more detail, as part of our simulation experiments, in Section 4.1 we compare the solution of the MOPA algorithm, which solves the unconstrained optimization problem, presented in Section 3.1, with the optimal solution as well as with the SBON approach that as already discussed uses a spring relaxation method to minimize network usage. For the comparison with the optimal solution, we have implemented the integer linear program presented in Subsection 3.1.4. Although ILP is not applicable in our system model, since it assumes central global knowledge of the system, we use ILP as reference to evaluate the optimality of our solution.

Problem	**Algorithms under Test**	**Evaluated against**	**Testing Environment**
Network Usage Optimization	MOPA	Optimal, SBON	PeerSim (Simulation), Cplex (MIP solver)
Network Latency Constraints	MOPA-LMAX	Optimal, MOPA, ILP-LMAX	PeerSim (Simulation), Cplex (MIP solver)
Network and Processing Latency	ckNN	EL, Random, kNN	NET (Emulation)

Table 4.1: Overview of placement algorithms under test

Then, in section 4.2, we compare the solution found by our proposed MOPA-LMAX algorithm for the network latency constrained problem, presented in Section 3.2, with the optimal solution and the solution of the unconstrained optimization. For the comparison with optimal solution, we use an exhaustive search for small operator graphs. For the analysis of the scalability of the algorithm with respect to the size of the operator graphs we have also implemented the integer linear program presented in Subsection 3.2.4, which calculate the optimal solution. Note that the reason we compare the unconstrained solution found by MOPA with the constrained solution found by MOPA-LMAX, is to provide an analysis on the trade-off between network usage and network latency optimization. Finally, in Section 4.3, we present the evaluation for MOPA-LPMAX by comparing the performance of different candidate selection methods presented in Subsection 3.3.4

For the setup of the experiments, we have specified the parameters of the

physical network (number of nodes, network latency among them etc.) and the operator graphs (structure, data rates etc.). Regarding the setup of the physical network, for *all* experiments, we have used the same underlying (physical) network topology. In particular, we have used data gathered from a real network, namely the PlanetLab [31]. The *PlanetLab topology* consists of 226 physical nodes including real measurements of the delays between the nodes globally distributed. Thus, providing a wide-area scenario with majority of network latencies in the range of $[45, 205]$ ms. The coordinates of the physical nodes in the latency space were found using a prototype implementation of the Vivaldi algorithm [1] that achieves to map the physical nodes in the latency space with an average error of 15 ms w.r.t. to the measured delays. The real PlanetLab topology gives us the chance to assess the practical performance of our algorithm in a realistic system.

The rest of this chapter is structured similar to the previous one, i.e., in each Section, we discuss for each of three placement problems introduced in the previous chapter, the evaluation of the proposed placement algorithms. For each of the three problems, we first present the evaluation setup, by providing details on the configuration of the various experiments. Since the setup for the physical network is the same for all experiments, in next sections, we describe the setup of the parameters of the operator graphs, depending on the goal of each experiment and present the evaluation objectives for each set of experiments. Then we go into detail in the main evaluation results, before we provide a brief summary for each of the evaluation sections.

4.1 Network Usage Optimization

First, we evaluate the performance of the unconstrained optimization algorithm. As explained in the introduction, we have used the network simulator PeerSim, to test our algorithm.

4.1.1 Setup

Since the structures of the operator graphs to be deployed possibly influence the performance of the placement algorithm we use, depending on the concrete experiment, operator graphs with different sizes, varying from 6 up to 15 nodes. Moreover, we assume that every operator has two or three children since we assume that this represents the usual case of an operator graph well.

The data rates on the links are generated randomly by varying the initial output data rates of the sources and the selectivity of the operators in a certain interval. The output data rates of the sources are distributed uniformly in the interval between 100 and 200 kbps or 50 and 500 kbps, depending on the scenario. The selectivity of an operator is defined as the percentage of the output data rate with respect to the input data rate of the operator. Thus, operators with a selectivity close to 0 act as highly selective filters in the network and generate very low output data rates, whereas operators with selectivity close to 1 generate output data rates equal to the incoming rate. In our evaluation, we vary the selectivity of the operators between 0 and 1.

4.1.2 Evaluation objectives

As already mentioned earlier, we compare our unconstrained optimization placement algorithm, to one state of the art algorithm called *SBON* [84]. Moreover, we compare our algorithm to the theoretical optimal placement algorithm (called *MOPopt*) solving the discrete MOP problem.

For the comparison with the optimal solution, we use two approaches. For large operator graphs, we use a mixed integer programming (MIP) solver that implements the ILP formulation presented in Section 3.1.3. to test also the speed of our algorithm compared to a MIP solver. For small operator graphs, we use a simple exhaustive search to verify the optimality of our solution.

Table 4.2 shows an overview of the performance metrics used for the evaluation of MOPA. The main objectives of our evaluation are to measure the *quality* of the solution and the *convergence properties* of the algorithm. To

measure the quality of the solution, we compare our algorithm to SBON [84] and to the optimal solution (called *MOPopt*) that minimizes the network usage. We compare the quality of our solution both for the continuous as well as for the discrete variation of the multi-operator placement problem. Furthermore, we test the convergence properties of our algorithm.

First, we measure the performance of MOPA compared to SBON in the continuous latency space and then we provide a comparison of both MOPA and SBON with respect to the optimal solution both for operator graphs of varying size. Next, we investigate the convergence properties of our algorithm. In particular, we measure the overhead induced by our algorithm in a distributed setting, by measuring the number of messages to be exchanged among the operators in order to find a new optimized placement as well as by measuring the number of migrations which lead to a better placement. Finally, we investigate the scalability of our algorithm with respect to the operator graph size. In that respect, we measure the execution time and the precision of MOPA, our proposed distributed algorithm, compared to a centralized integer linear programming solver, which solves the integer linear program introduced in Section 3.1 for varying operator graph size.

For each evaluation objective, we have assigned specific *performance metrics*, that we explain in detail in the following subsections.

4.1.3 Quality: Continuous MOPA Solution

First we evaluate the quality of a placement determined by MOPA to an SBON placement w.r.t. the continuous solutions, i.e. the coordinates of the free operators in the continuous latency space. In this experiment, we use operator graphs with 12 operators and 2 to 3 children per operator.

For the comparison, we use the *virtual stretch factor* $S_{\text{SBON,MOPA}}$ as performance metric. The virtual stretch expresses the network usage of SBON relative to the network usage of MOPA based on their continuous solutions:

Objective	**Performance Metric**	**Definition**
Quality	Virtual (Network Usage) Stretch Factor	$\frac{\hat{U}_{\text{global,SBON}}}{\hat{U}_{\text{global,MOPA}}}$
	Physical (Network Usage) Stretch Factor	$\frac{\hat{U}_{\text{global,SBON\|MOPA}}}{\hat{U}_{global,optimal}}$
Convergence Properties	Messages	# messages
	Migrations	# migrations
Scalability (w.r.t. Operator Graph Size)	Execution Time	Value in sec
	Physical (Network Usage) Stretch Factor	$\frac{\hat{U}_{\text{global,SBON\|MOPA}}}{\hat{U}_{global,optimal}}$

Table 4.2: Overview of performance metrics

$$S_{\text{SBON,MOPA}} = \frac{U_{\text{global,SBON}}}{U_{\text{global,MOPA}}} \tag{4.1}$$

That is, the virtual stretch compares solutions in the continuous virtual latency space rather than the results after mapping to the physical nodes. A comparison of the latter can be found in the next Subsection 4.1.4. For example a virtual stretch of 1.2 shows that the output of SBON algorithm is 20% worse than the optimal MOPA solution.

Figure 4.1 shows the cumulative distribution of the virtual stretch factors of SBON. This figure is the result of the placement of 1000 graphs. First of all, we can see that SBON always has a virtual stretch greater than 1.0. That means, MOPA always achieved higher quality continuous solutions than SBON. This is due to the spring relaxation algorithm of SBON that finds

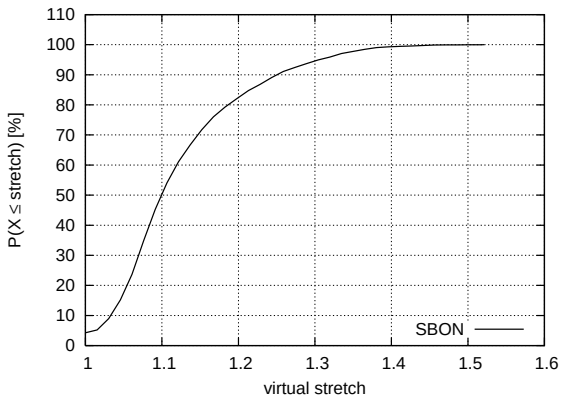


Figure 4.1: Relative network usage of SBON w.r.t. MOPA (Continuous solutions).

the mass centroid rather than the solution to the continuous multi-operator placement problem.

In approximately half of the cases the virtual stretch of SBON is lower than 1.1. Thus, SBON achieves a good estimation with a maximum difference of 10% of the optimal in 50% of the cases. This good result of SBON is due to the fact that similar to MOPA the spring relaxation algorithm moves operators in the correct direction of the major flow. However, in contrast to MOPA, SBON stops too early before it reaches the minimum, whereas MOPA moves on until the optimal placement is reached. The remaining 50% of the measurements have a stretch factor between 1.1 and 1.5, i.e., in 50% of the cases, MOPA reduces the network usage significantly leading to 10% to 50% less network usage than SBON.

So we see that in the continuous space MOPA is always better achieving an average improvement compared to the SBON of 12% and a maximum of 52%.

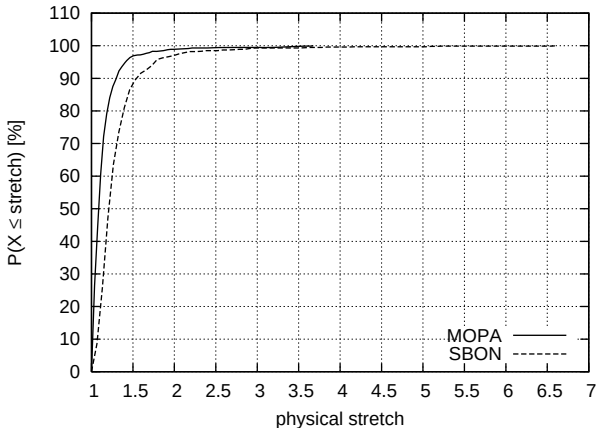


Figure 4.2: Physical stretch factor of SBON and MOPA w.r.t. optimal discrete MOP solution (Operator Graph Size:6).

4.1.4 Quality: Discrete MOPA Solutions

As a first step we investigated the quality of approximated discrete MOP solutions.

As performance metric, we use the *physical stretch factor*:

$$S_{\text{SBON|MOPA,global}} = \frac{\hat{U}_{\text{global,SBON|MOPA}}}{\hat{U}_{global,optimal}}$$

$\hat{U}_{\text{global,SBON|MOPA}}$ denotes the network usage of a discrete network usage optimization solution given by SBON and MOPA, respectively. $\hat{U}_{\text{global,optimal}}$ defines the optimal discrete MOP solution determined by MOPopt. This optimum serves as a reference of the approximated solutions achieved by SBON and MOPA. We perform this evaluation both operator graphs 6 nodes as well as for larger operator graphs of 15 nodes. For the small operator graphs, we used an exhaustive search to calculate the optimal solution of the network usage optimization problem, while for the large graphs, we used a mixed integer programming solver to run the ILP program presented in Subsection 3.1.4. The details of this implementation will be explained in the next subsection.

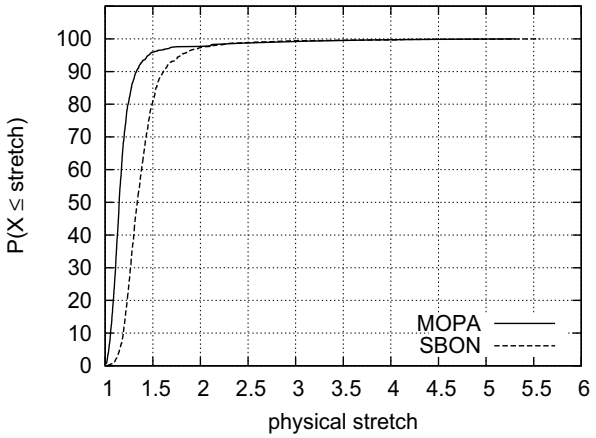


Figure 4.3: Physical stretch factor of SBON and MOPA w.r.t. optimal discrete MOP solution (Operator Graph Size:15).

Figure 4.2 shows the cumulative distribution (for the small operator graph) of this experiment resulting from 1000 simulation runs. We see that in 70% of the measurements MOPA has a stretch factor lower than 1.1. The average stretch factor of MOPA is 1.14. Thus we see that although the latency space sparsely populated with the 256 physical nodes of Planetlab topology, the optimal continuous MOP solution does not degenerate significantly after the physical mapping.

We also see that for 70% of the measurements, SBON has a stretch factor of 1.3 which is 16% higher than the physical stretch of MOPA for the same percentage. The average stretch factor of SBON is 1.29 compared to 1.14 for MOPA. Thus, MOPA keeps its theoretical advantage of having optimal continuous MOP solutions also after the mapping to physical nodes.

The highest stretch factor for SBON is 6.61, whereas the maximum of MOPA is only 3.67. In these cases the approximation is not close to the physical optimum. On the one hand, such a case can be caused by a bad mapping of a physical node in the latency space where the delays between physical nodes modelled in the latency space do not accurately reflect the real

delays. On the other hand, the sparse character of the network topology can lead to bad discrete MOP approximations, where no well-matching physical node for the calculated virtual node position can be found.

Figure 4.3 shows the corresponding cumulative distribution for the large graphs of 15 nodes resulting from 1000 runs. We see that the performance of MOPA and SBON algorithms compared to the optimal solution slightly degrades with respect to their performance for the small operator graphs. In more detail, the average stretch factor for MOPA is 1.21 compared to an average stretch factor of 1.39 for SBON. Moreover, for 70% of the measurements, MOPA has a stretch factors of lower than 1.19, which is 23% lower than the corresponding stretch factor for SBON (1.42%). Thus we see that for larger operator graphs, which contain more free operators, the discovery of the optimal solutions becomes more challenging. However, as we see MOPA still keeps a signifant improvement over SBON by achieving 18% lower stretch factor on average.

4.1.5 Convergence: Message Overhead and Migrations

Finally, we evaluate the convergence properties of MOPA compared to SBON. We consider two performance metrics. First, we measure the induced *network overhead* denoted by the number of messages that have to be exchanged in order to communicate virtual node coordinates to neighbouring operators whenever a new operator position has been calculated. Secondly, we measure the number of *operator migrations* that are performed until the equilibrium is reached. Since migrations largely outweigh local computations, the number of migrations is also an indicator for the convergence time.

For this experiment, we use operator graphs with size of 12 nodes. We first let both algorithms converge to a stable solution. Then, we generate a dynamic change by resetting the output data rates of all sources to new random values (in the range of 100 to 200 Kbps). The sudden change of the data rates provokes the re-placement of the operators. Note that a sudden

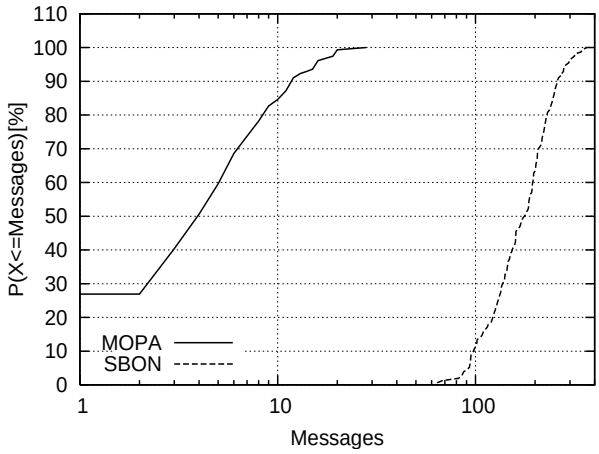


Figure 4.4: Cumulative distribution of number of messages exchanged (data rates 100-200Kbps).

change of *all* the data rates is a worst case scenario since the whole operator graph is affected. We placed 2000 operator graphs and measured the number of migrations and messages exchanged until the equilibrium is reached.

Figure 4.4 shows the cumulative distribution of the number of required messages for SBON and MOPA. We see that MOPA needs significantly fewer messages to converge to a new equilibrium in all the cases. In detail, MOPA needs between 0 and 28 messages, while SBON needs 64 to 365 messages. On average, MOPA only needs 3.25% of the messages that are needed by SBON. Furthermore, MOPA in 26.9% of the instances needed less than 2 messages to converge to the new solution. This result implies that the dynamic change was not significant enough to change the location of the network usage optimum.

Thus, in order to investigate this result further, we have run another experiment, where we created a dynamic change that is more challenging for MOPA, by changing the relative values of the data rates more drastically. In more detail, we have set the output data rates of the sources in the range of 50 to 500 Kbps. Thus, we have increased the variation in the value of data rates on the links, which alters possibly the network usage minimum. Figure 4.5

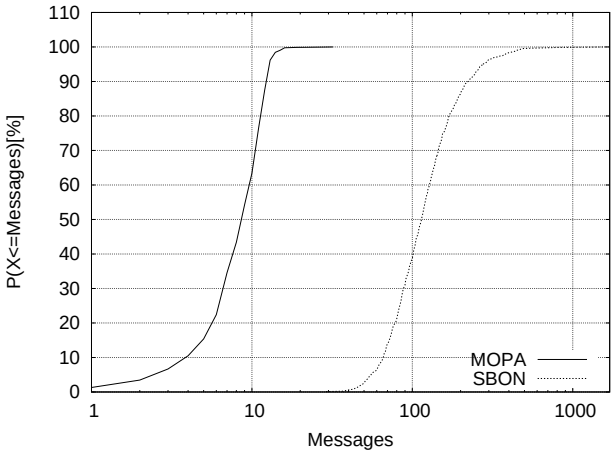


Figure 4.5: Cumulative distribution of number of messages exchanged (data rates 50-500Kbps).

shows the cumulative distribution for this experiment. We see that also in this case MOPA has a superior performance needing 4 messages on average, compared to 135 messages for SBON. In more detail, MOPA needs less than 11 messages in 63% of the instances, while SBON needs 134 messages for the same percentage of simulation runs. Thus, we see that even for significant dynamic changes MOPA adapts its solution by sending only few messages (up to 32 messages).

To get a better insight into this result, we have measured for each algorithm, MOPA and SBON the number of local iterations, i.e., how many times the operators should contact their neighbours until the operator graph converges to a new solution. Figure 4.6 shows the cumulative distribution for the local iterations for MOPA and SBON. As expected MOPA needs significantly less iterations from 0 up to 11 iterations, while SBON needs from 10 up to 551. This result shows that MOPA is able to move in larger steps, while SBON moves slowly, making only small progress in each iteration. Furthermore, we also measured the number of the suppressed messages, i.e., messages that are not sent over the network, since the neighboring operator resides on the

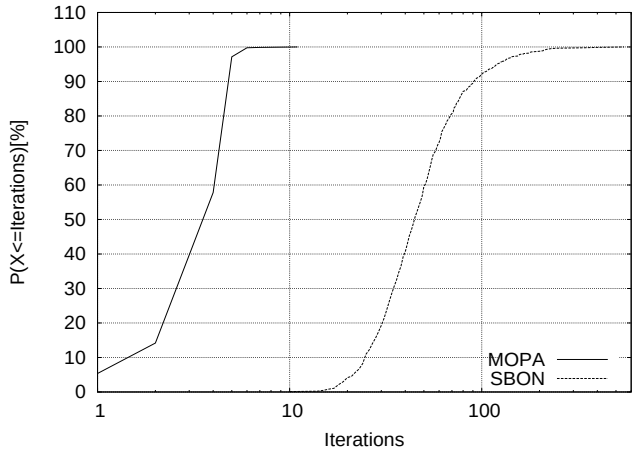


Figure 4.6: Cumulative distribution of local iterations.

same physical node. Figure 4.7 shows the percentage of sent and suppressed messages with respect to the total number of messages. We see that for SBON the suppressed messages are 6.4%, while for MOPA the corresponding percentage is 21.8%. Thus, we see that in addition to the convergence speed, MOPA is more probable to create clusters, which reduce the communication overhead of the algorithm.

Figure 4.8 and Figure 4.9 depict the cumulative distributions of the number of migrations in the physical network for MOPA and SBON for dynamic changes depending on the output data rates. The performance of the algorithms is similar for both dynamic changes. Both MOPA and SBON needs from 0 up to 8 migrations to converge to the new solution. Again, MOPA outperforms SBON by an average of 26.8% (for changes in the range from 100 to 200 Kbps) and 26.4% (for changes in the range from 50 to 500 Kbps) less migrations. Moreover, we see that in more than 90% of the simulations, MOPA needs less migrations than SBON. Similar to the number of exchanged messages, the reason for the smaller number of migrations of MOPA is the faster convergence due to larger step size.

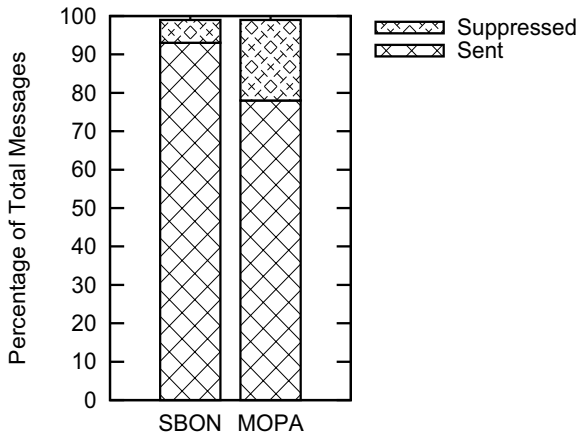


Figure 4.7: Stacked histogram of sent and suppressed messages.

4.1.6 Scalability: Execution time and Performance

In Subsection 3.1.4 we have presented an integer linear program (ILP) that can be used to calculate a solution of the operator placement problem centrally on one host with global knowledge. In contrast to our distributed algorithm MOPA, this ILP cannot provide a distributed solution and rely on local knowledge. However, it can serve as a reference with respect to the computational efficiency (executing any communication overhead for the distributed execution of MOPA for a fair comparison). For our comparison, we have used the commercial mixed integer programming solver CPLEX 12.5.0 from IBM, which is considered to be one of the fastest mixed integer programming solvers currently [60] Both CPLEX and MOPA were executed locally on one machine (Intel Core i5, 2.67 GHz, 4 cores, 12 GB RAM).

For our experiments, we used the PlanetLab topology as the underlying network. For the settings of the operator graph, we use a tree-based graph with varying size starting from operator graphs with 10 nodes up to 50 nodes. At each run, we alter the data rates and the location of the pinned operators. We used $1,000$ operator graphs to compare ILP with MOPA. Figure 4.10

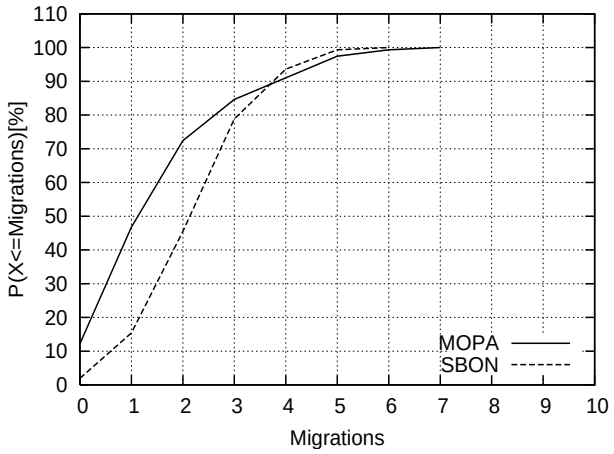


Figure 4.8: Cumulative distribution of migrations (data rates 100–200Kbps).

summarizes the measurements in terms of execution time for both MOPA and CPLEX. We observe that MOPA keeps a low execution time varying from 0.66 seconds up to 3 seconds for operator graphs of 50 nodes, while CPLEX needs from 1.91 seconds up to 18.45 seconds to calculate the optimal solution. This results show that the execution time of CPLEX increases significantly with the increase of he operator nodes, while MOPA is more scalable keeping its execution time in the order of a few seconds even for large operator graphs of 50 nodes.

Finally, Figure 4.11 shows the average network usage stretch factor for both MOPA and SBON colpared to the optimal solution calculated by CPLEX for varying operator graph size. We observe that the average network usage stratch factor does not vary significantly with size of the operator gpaph for both SBON and MOPA. For SBON the average stretch factor varies from 1.40 up to 1.46, while for MOPA the corresponding value varies from 1.18 to 1.21. Thus, we see that MOPA can provide high quality solutions deviating only 21% on average from the optimum even for large operator graphs of 50 nodes.

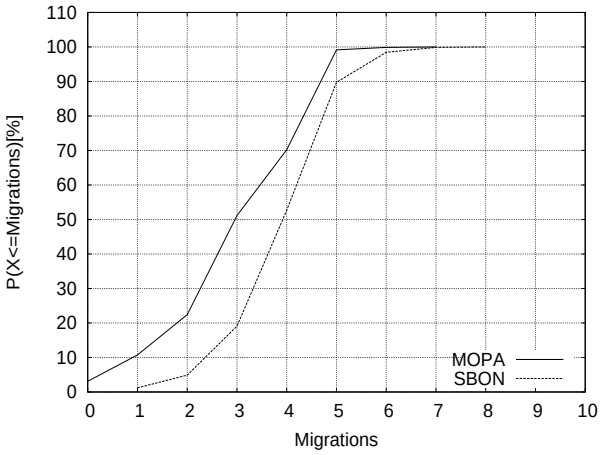


Figure 4.9: Cumulative distribution of migrations (data rates 50–500Kbps).

4.1.7 Summary

The evaluation results of the MOPA algorithm show that MOPA can achieve a good estimation (14%– 21%) of the optimal solution, by using a fully distributed approach that considers only local knowledge of the system. Furthermore, we showed that MOPA finds better solutions than SBON, since the latter approach approximates the optimal solution by calculating the mass centroid, which does not necessarily coincide with the geometric median which is the actual network usage minimum. Finally, MOPA produces also significantly less overhead than SBON, by using only 3.25% of the messages that are needed by SBON, since it makes large steps towards the network usage minimum and it is more likely to merge free operators into clusters. Moreover, MOPA is able to calculate the solution even of large operator graphs of 50 nodes in 3 seconds on average, while one of the fastest state-of-the-art (centralized) linear programming solver need 18.45 seconds to calculate the solution for the same operator graph size.

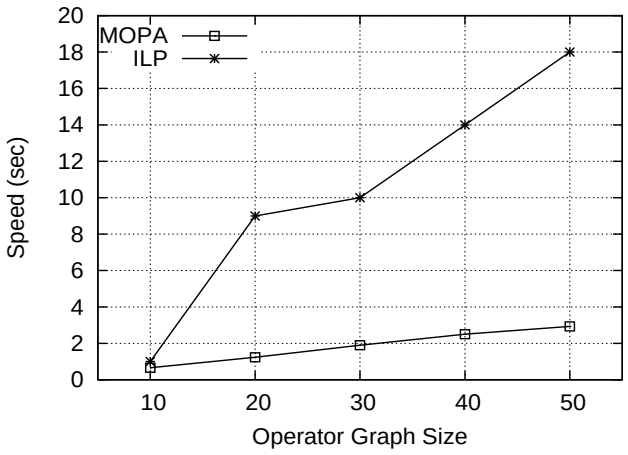


Figure 4.10: Execution time of MOPA and CPLEX w.r.t. graph size.

4.2 Network Delay Constrained Optimization

Next, we present the performance evaluation of the MOPA-LMAX algorithm which solves the network delay constrained optimization algorithm by comparing it to the theoretic optimum and our unconstrained optimization algorithm.

4.2.1 Setup

Similar to the evaluation of the MOPA algorithm, we evaluate the performance of MOPA-LMAX in the network simulator PeerSim. For our experiments, we use the PlanetLab physical network as presented in the introduction of this Chapter. For the settings of the operator graphs, we use operator graphs with 6 nodes. Similar to the evaluation setup of MOPA presented in Subsection 4.1.1, we alter the selectivity of the operators and we generate output data rates from the sources in the interval between 100 and 200 kbps. Since there is no related approach that solves the same constrained optimization problem, we compare our constrained optimization algorithm with the theoretic optimum and the MOPA algorithm presented in 3.1 which solves

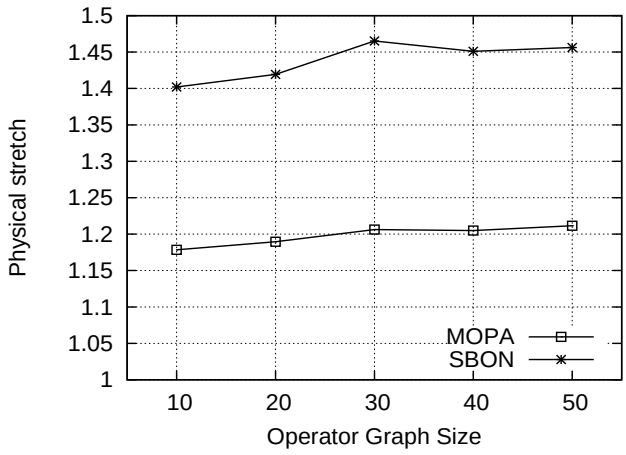


Figure 4.11: Physical Stretch Factor of MOPA and SBON w.r.t. graph size.

the unconstrained network usage optimization problem. To find the real optimum, we execute an exhaustive search on all possible placements.

4.2.2 Evaluation Objectives

The evaluation objectives for MOPA-LMAX are related to the quality of the solution and the scalability of the algorithm. We do not provide a further evaluation on the communication overhead induced by MOPA-LMAX since this is similar to MOPA overhead as discussed in Subsection 3.2.3. As discussed in the introduction, we compare MOPA-LMAX with the unconstrained optimization algorithm MOPA and with optimal solution, found by an exhaustive search for small operator graphs. First, we investigate the relationship between the minimization of bandwidth-delay product and the minimization of network delay. Our comparison is based on the quality of the solution in terms of resulting network usage and latency. Next, we calculate the success rate, which is the percentage of the experiments that fulfill the latency constraint and we provide a further insight on the distribution of instances with respect to network usage and latency constraint. Finally, we provide an insight on

the scalability of the algorithm with respect to the operator graph size by using the extended ILP formulation of the constrained optimization problem presented in Subsection 3.2.4. Table 4.3 shows the performance metrics used for our evaluation, that will be further explained in the next subsections.

Objective	**Performance Metric**	**Definition**
Quality	Latency Stretch Factor	$\frac{L_{\text{unconstr\_opt}}}{L_{\text{min}}}$
	Network Usage Stretch Factor	$\frac{U_{\text{constr\_opt}}}{U_{\text{min}}}$
	Success Rate	$\frac{\#\text{successful\_experiments}}{\#\text{experiments}}$
Scalability (w.r.t. Operator Graph Size)	Execution Time	Value in sec
	Latency Stretch Factor	$\frac{L_{\text{constr\_opt}}}{L_{\text{min}}}$
	Network Usage Stretch Factor	$\frac{U_{\text{constr\_opt}}}{U_{\text{min}}}$

Table 4.3: Overview of performance metrics

4.2.3 Quality: Relation Between Network Usage and Latency

First, we analyze the basic relation between the two metrics subject to this algorithm, namely network usage and latency. Since the network usage contains as one factor the delay between operators, in this experiment we see how close an unconstrained optimization of the network usage can get to the latency minimum.

We have conducted 1000 experiments and measured the latency and the

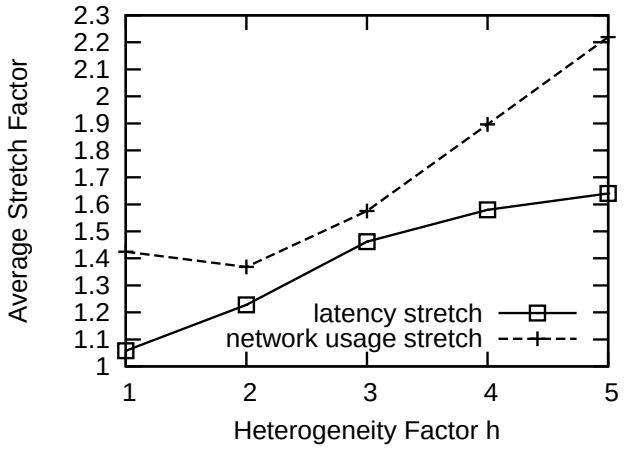


Figure 4.12: Latency and Network Usage stretch for varying heterogeneity.

network usage minimum. In detail, we have calculated by exhaustive search the theoretic latency minimum L_{min} and the latency $L_{\text{unconstr\_opt}}$ achieved by the optimal unconstrained optimization of the network usage. In order to quantify the difference w.r.t. latency between the network usage optimum and the latency minimum, we calculated the *latency stretch* factor defined by $S_{\text{unconstr\_opt,min}} = \frac{L_{\text{unconstr\_opt}}}{L_{\text{min}}}$. Similarly, the network usage stretch is defined as $S_{\text{constr\_opt,min}} = \frac{U_{\text{constr\_opt}}}{U_{\text{min}}}$, where $U_{\text{constr\_opt}}$ is the network usage of the constrained optimization with minimum latency constraints and U_{min} is the theoretic optimum of the unconstrained optimization.

To parametrize the heterogeneity of the operator graph, we introduce the heterogeneity factor h. In detail, for an operator connected to n sources, we set the output data rates of $n-1$ sources at the same random value r and the remaining output data rate at $h \cdot r$, i.e. proportional to h. Moreover, the selectivity of the unpinned operators is set to $1/h$, i.e., inversely proportional to h. Thus, for large h, the input data rates of an operator are unbalanced, while the output data rates of the operator are low.

Figure 4.12 shows the results for varying values of the heterogeneity factor h. We see that as the heterogeneity increases, the stretch factors both in terms

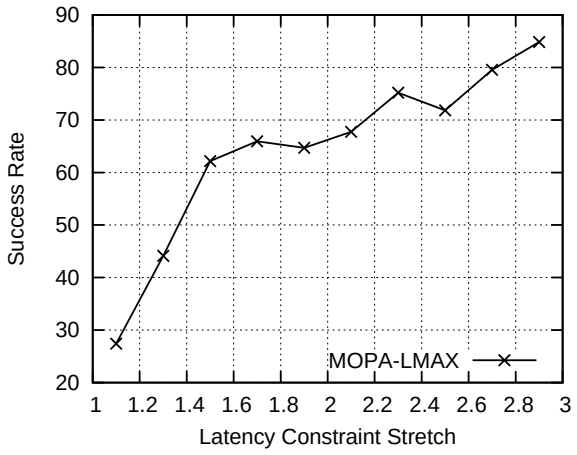


Figure 4.13: Success rate according to the constraint latency stretch.

of latency and network usage are also increasing, since there are more high data rate sources making the unconstrained network usage and the latency minimum considerably different. Moreover, we see that the network usage stretch is generally larger than the latency stretch. This is due to the fact that the latency is bounded by the distance between the sources and the sinks, whereas the network usage is affected by the values of the data rates that can eliminate or amplify some of the factors of the total sum of an operator graph's network usage.

4.2.4 Quality: Fulfillment of Network Latency Constraints

Next, we continue with the analysis of the performance of the MOPA-LMAX algorithm for the network delay constrained optimization problem. First, we evaluate the ability of our algorithm to achieve a given latency constraint. In the following experiment, we vary the given latency constraint in the interval $[L_{\text{min}}, L_{\text{unconstr\_opt}}]$, i.e., between the theoretic latency minimum and the latency achieved by the unconstrained optimization algorithm. Choosing a lower bound of L_{min} ensures that in every case a solution exists. By

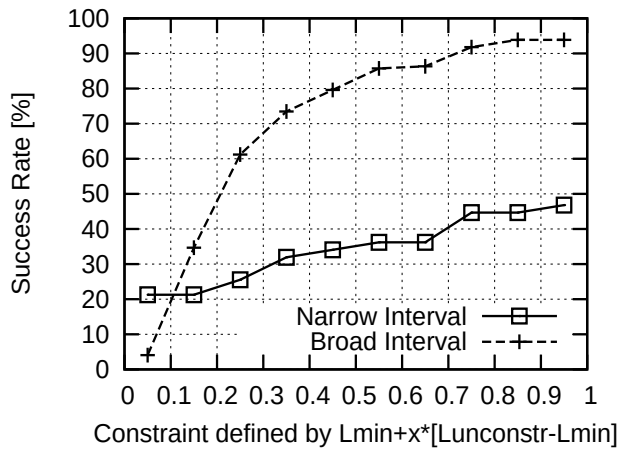


Figure 4.14: Success rate for narrow/broad latency stretch interval

choosing an upper bound of $L_{\text{unconstr\_opt}}$ we evaluate cases with non-trivial solutions that would not be achieved by an unconstrained optimization algorithm. Moreover, in order to distinguish between challenging cases, where the solution of the unconstrained optimization algorithm is far from the latency minimum, we classify our experiments according to the achieved latency stretch factor of the unconstrained optimization algorithm. For instance the class $[1.0, 1.2]$ contains all experiments, where the unconstrained solution has a latency stretch of 1.0 to 1.2 compared to the theoretic latency minimum. In general, as the latency stretch of the unconstrained solution increases, the constraint interval $[L_{\text{min}}, L_{\text{unconstr\_opt}}]$ also broadens.

For our experiment, we have generated 1000 operator graphs with varying heterogeneity factor $h \in [1, 3]$ and measured the performance of our algorithm by calculating the percentage of the experiments that achieved a latency below the constraint by $\text{success}_{\text{rate}} = \frac{\#\text{successful\_experiments}}{\#\text{experiments}}$, i.e. successful experiments, where the latency constraint was met, divided by the number of all experiments. Furthermore, to evaluate the cost for satisfying the constraint, we calculate the network usage stretch with respect to the network usage of the unconstrained problem that we get after the unconstrained optimization:

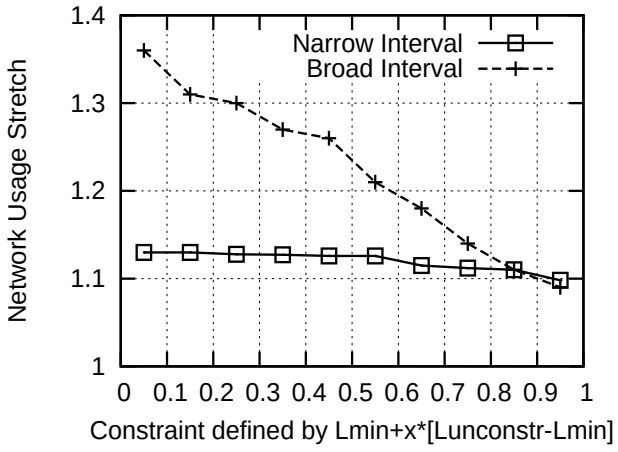


Figure 4.15: Network usage stretch for narrow/broad latency stretch interval

$S_{\text{constr\_opt,unconstr\_opt}} = \frac{U_{\text{constr\_opt}}}{U_{\text{unconstr\_opt}}}$.

Fig. 4.13 shows the success rate of our unconstrained optimization algorithm for different classes. Here, we see that for low latency stretch, e.g., below 1.2 of the unconstrained solution, our algorithm has a low average success rate of 27%, while for larger latency stretch, e.g., between $1.4 - 1.6$, the algorithm works better achieving an average success rate of 62%. For even higher latency stretch, e.g., between $2.6 - 2.8$, our algorithm can achieve an average success rate of 79%. We can explain the poor average success rate of our algorithm for low latency stretch of the unconstrained solution since low latency stretch means a narrow interval of the latency constraints. Thus, in such cases all requested latency constraints are very close to the real optimum. However, as we see in the next subsection, also in these cases our algorithm returns a good approximation of the optimum.

Figure 4.14 and Figure 4.15 show the success rate and the network usage stretch for operator graphs with latency stretch values between 1.0 and 1.2 (narrow interval), and 2.0 and 2.2 (broad interval), respectively. In Figure 4.14, we see on the x axis the latency constraints that are requested, varying gradually in a step of 10% of the total constraint interval $[L_{\text{min}}, L_{\text{unconstr\_opt}}]$

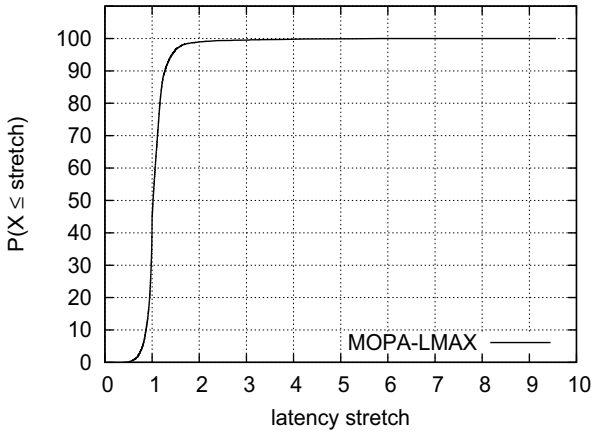


Figure 4.16: Cumulative distribution of latency stretch.

at a time, i.e., $l_max \in [L_{\min} + x * (L_{\text{unconstr\_opt}} - L_{\min})]$, where $x \in [0, 1]$. On y axis we have depicted the success rate of the constrained solution. Similarly, in Figure 4.15 we have depicted the latency constraint in the x axis and the network usage stretch in the y axis. On the one hand, we see that when the latency stretch is low (Figure 4.14), the average success rate is increasing slowly from 20% to 46% while the average network usage stretch is kept low and decreases slowly from 1.13 to 1.1 (Figure 4.15). On the other hand, for large latency stretch values (Figure 4.14), we see that the success rate increases gradually, going from 29% for strict constraints where $x < 0.1$ up to 98% for relaxed constraints where x is above 0.8, and the network usage costs decrease significantly from 1.37 to 1.09 (Figure 4.15) for more relaxed constraints.

Thus, we see that for small latency stretch of the unconstrained solution, the success rate remains in general low, while the cost is also low since even the unconstrained optimization algorithm can achieve a good approximation of the latency constraint, while for larger latency stretch our algorithm performs better as the constraints become more relaxed, resulting in higher success rates and lower costs with respect to network usage.

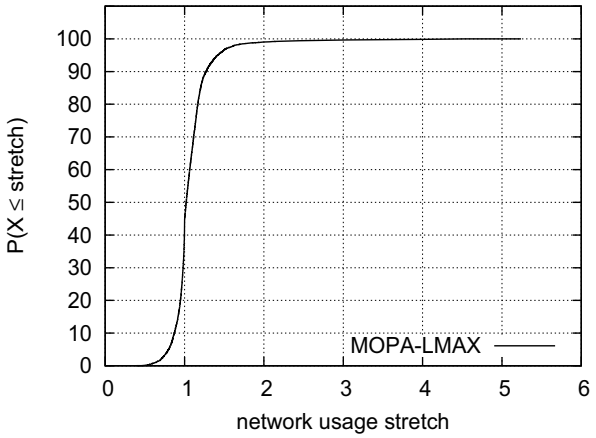


Figure 4.17: Cumulative distribution of network usage.

4.2.5 Quality: Deviation from Network Delay Constraints

In the previous experiment, we have presented the evaluation results with respect to the fulfillment of the constraints. We have seen that in some cases especially where the latency constraint was close to the latency minimum, we get low success rates, since it becomes hard to approximate the latency minimum and satisfy these strict constraints. To get a better understanding of the performance of the algorithm, in this experiment, we have a closer look on the quality of solutions by considering the distribution of the achieved latencies around requested latency constraints. On the one hand, this evaluation shows how far apart unsuccessful solutions are from the requested constraints. On the other hand, it also shows us the degree of overshooting of successful solutions.

For this experiment, we use a generic scenario with heterogeneity factor $h \in [1, 5]$ and latency constraints randomly set in the interval $[L_{\text{min}}, L_{\text{unconstr\_opt}}]$ to get a general picture of the precision of the algorithm.

The quality of a solution in terms of latency can be evaluated by the latency stretch of the solution, $S_{\text{constr\_opt},\mathrm{l}_{max}} = \frac{L_{\text{constr\_opt}}}{\mathrm{l}_{max}}$, i.e. the constrained op-

timum compared to the requested latency constraint l_{max}, which intuitively shows how close the solution is to the requested constraint.

Fig. 4.16 shows the cumulative distribution of the latency stretch for a set of 4,000 simulation runs. Overall, 56% of the solutions were successful, i.e. the latency constraint was met, whereas in 44% of the simulations, the latency constraint was violated. In detail, 70% of the unsuccessful experiments that were above l_{max} have a latency stretch between 0.9 and 1. Moreover 75% of the successful solutions, where the requested latency l_{max} was met, have a latency stretch below 1.15%. Thus, we see that the majority of the instances are distributed closely around the constraint. However, there are some instances with larger deviation from l_{max}, e.g. 5% that are above 1.4. As we have seen during the evaluation of MOPA algorithm in Section 4.1, such bad approximations of the optimal solution may exist due to the mapping of continuous to discrete solution.

Moreover, we calculate the network usage stretch compared to the network usage of the theoretic constrained optimum found by exhaustive search $S_{\text{constr\_opt,theoretic\_constr\_opt}} = \frac{U_{\text{constr\_opt}}}{U_{\text{theoretic\_constr\_opt}}}$. Figure 4.17 shows the corresponding cumulative distribution. We see that there is a percentage of 41% that have a smaller network usage than the constrained optimum. In these cases, the latency constraint was not met by the solution. Therefore, the network usage stretch can be even smaller than the theoretical constrained optimum. Moreover for the possibly successful solutions that have a stretch above 1, we see that our algorithm achieves a very good approximation of the constrained optimum with an average network usage stretch of 1.09%. In 80% of these cases the network usage stretch is below 1.17%, showing that our algorithm achieves its goal to keep the network usage low.

4.2.6 Scalability: Execution Time and Performance

To evaluate the scalability of MOPA-LMAX, we have implemented the integer linear program formulation presented in Subsection 3.2.4 in CPLEX, the

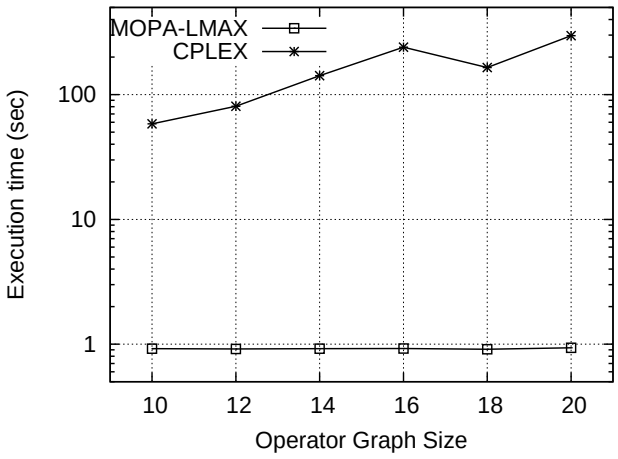


Figure 4.18: Execution time of MOPA-LMAX, CPLEX w.r.t. Graph Size.

mixed integer programming solver used also in the experiment presented in Subsection 4.1.6. Both CPLEX and MOPA-LMAX were executed locally on one machine (Intel Core i5, 2.67 GHz, 4 cores, 12 GB RAM).

For our experiments, we used the PlanetLab topology as the underlying network. At each run, we alter the location of the pinned operators and we used 100 operator graphs to compare the extended ILP presented in Subsection 3.2.4 with MOPA-LMAX. Note that if we provide a latency constraint lower than the latency minimum, CPLEX will not return any solution, since the constraint is infeasible. Therefore, to ensure that the latency constraints set for the execution of the ILP are feasible solutions, we have first executed MOPA-LMAX with latency constraint equal to zero. In that case, MOPA-LMAX has returned the best possible solution. The latency returned by MOPA-LMAX was then fed as latency constraint to the ILP.

For the settings of the operator graph, we use a tree-based graph with varying size starting from operator graphs with 10 nodes up to 20 nodes. We do not provide results for operator graphs with more than 20 nodes, since the execution time of the ILP presented in Subsection 3.2.4 in CPLEX is already for operator graphs of 20 nodes very high. In particular, as it is shown in

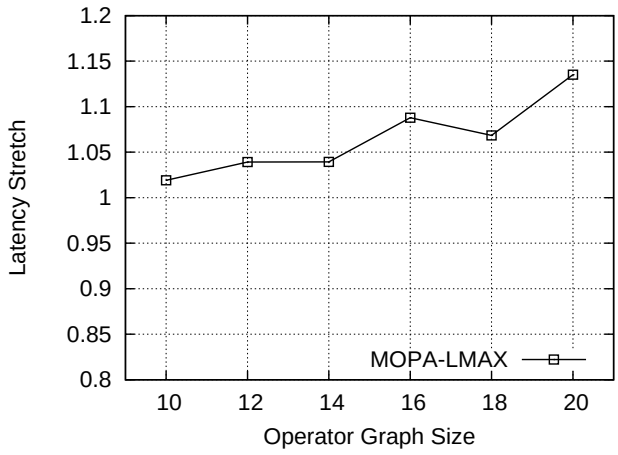


Figure 4.19: Latency Stretch of MOPA-LMAX w.r.t. Graph Size.

Figure 4.18 . the execution time for CPLEX varies from 58 seconds for operator graphs with 10 nodes up to 296 seconds on average for operator graphs of 50 nodes, while MOPA-LMAX needs almost $0.92 - 0.93$ second on average to calculate its solution. Thus, we see that the additional latency constraint introduced in Subsection 3.2.4 has significantly increased the execution time of the extended ILP compared to the ILP for the unconstrained optimization problem presented in Subsection 3.2.4, while MOPA-LMAX has kept a low execution time compared to MOPA, keeping its scalability properties.

Figure 4.19 and Figure 4.20 show the average latency and network usage stretch factor for MOPA-LMAX for varying operator graph size. Since we have used the as latency constraint, the latency value returned by MOPA-LMAX, CPLEX has returned equal or better solutions both in terms of latency and network usage. Therefore, we have used as reference for the calculation of the latency stretch factor and network usage stretch factor of MOPA-LMAX the latency achieved by CPLEX. In Figure 4.20 we observe that the average network usage stretch factor varies from 1.08 for operator graphs of 10 nodes up to 1.14 for graphs of 20 nodes, while the corresponding latency stretch factor Figure 4.19 varies from 1.02 up to 1.13. Thus, we see

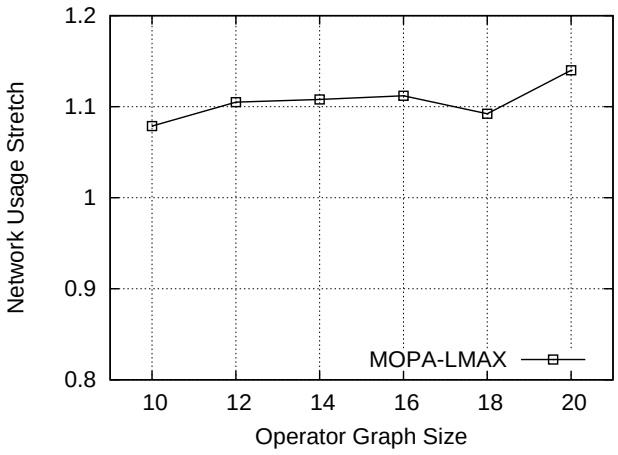


Figure 4.20: Network Usage Stretch of MOPA-LMAX w.r.t. Graph Size.

that MOPA-LMAX keeps its performance even for larger operator graphs of 20 nodes by achieving an average latency and network usage stretch factor of 1.13 and 1.14 respectively.

4.2.7 Summary

As a conclusion of this evaluation, we can say that our algorithm achieves a good balance between network usage optimization and satisfaction of latency constraints with an average success rate 62% for constraints with latency stretch of $1.4-1.6$. In cases that the constraint is not satisfied, our algorithm still finds a good approximation of the solution with a latency stretch below 1.15% for 75% of the instances, while minimizing the cost in terms of network usage by achieving an average network usage stretch of 1.09%.

Furthermore, our evaluation results provide an analysis of the relationship between network usage and network latency. Since network usage includes network latency, its minimization implies also reduction of the overall end-to-end latency. In our evaluation, we have seen that the heterogeneity of the operator graphs, meaning the variation of the data rates on its links

is a determinant on the ability of MOPA, which solves the unconstrained optimization problem, to minimize the network delay. Finally, we also showed that MOPA-LMAX is a scalable algorithm achieving good quality solutions with average network usage stretch factor of 1.14 and latency stretch factor of 1.13, even for larger operator graphs of 20 nodes.

4.3 Processing and Network Delay Constrained Optimization

In this section, we present the evaluation results for the MOPA-LPMAX algorithm, which solve the processing and network delay constrained optimization. We start with a description of the evaluation setup. Then, we evaluate in detail the performance of the placement algorithm in terms of optimality w.r.t. network load and its capability to satisfy latency constraints.

4.3.1 Setup

To evaluate MOPA-LPMAX, we have implemented them for the NET cluster [48], an emulation environment developed at the University of Stuttgart. NET provides an emulation environment for testing distributed systems and communication protocols. It combines the benefits of real-time experiments and network simulation. NET consists of a compute cluster, where every cluster node hosts several virtual nodes (in our case the operator hosts) that execute real implementation of the "software under test". Nodes are connected by an emulated communication network that can be parametrized such that it resembles a given network (including network topology and link characteristics such as latency and bandwidth). Using emulation instead of simulation gives us the chance to test a real implementation of our placement algorithm under realistic conditions.

For the physical network, we use the PlanetLab topology as described in the introduction of this chapter. On top of the physical network, we used

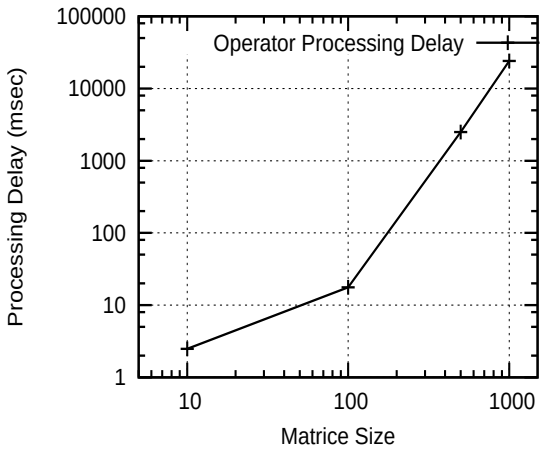


Figure 4.21: Processing delay w.r.t operator complexity (matrice size).

the Pyxida system running on each host [1] to calculate the latencies of the experiments *online*. Pyxida implements the Vivaldi algorithm [37] in order to calculate accurate coordinates where the distance closely matches the propagation delay.

For the operator graphs, we have used operators with different complexity. In particular, in order to vary the processing load induced by operators, we used operators implementing a matrix multiplication with different sizes. Besides giving us the opportunity to easily manipulate the processing load of operators, matrix multiplication is a common operation used, for instance, for traffic matrices in network monitoring or image recognition. We varied the size of matrices in the range from 50 to 500 by defining four discrete sizes of $\{50, 100, 200, 500\}$ elements. Thus, we cover a large spectrum of heterogeneous operators in terms of processing load. Consequently, the size of the data unit is defined by the size of the matrices.

To demonstrate the use of the matrix multiplication operator as configuration parameter to vary the processing delay, we have measured the induced processing delay of operators executing matrix multiplication with different matrix size on unloaded physical nodes. Figure 4.21 shows the results. In

Method	**Basic Idea**
EL	Return all hosts in E
kNN	Return the k nearest neighbours
Rand	Return k random nodes that reside in the ellipse
ckNN	Return the k nearest neighbours fulfilling the pruning criterion

Table 4.4: Overview of candidate selection algorithms

Objective	**Performance Metric**	**Definition**
Quality	Latency	Value in ms
	Network Usage	Value in Kb
Overhead	Messages	# messages

Table 4.5: Overview of performance metrics

particular, for operators of size 10 and 100, the average processing delay is 2.48 msec and 17.71 msec respectively. Moreover, for a matrix size of 500, the average processing delay is 2.49 sec, while for 1000 the processing delay goes up to 24.09 sec. Thus, we see that by using variable matrix size, we can vary the induce processing delay as expected.

For our experiments, an operator graph, has typically two free operators to be placed. The data sources feed the operators with data every 20 up to 120 seconds following a uniform distribution leading to heterogeneous data rates. Moreover, the data sources and sinks are uniformly distributed on random hosts in the network. The parameter k that defines the size of the candidate set is set to 5 hosts, i.e., 2.5% of the total number of hosts in the network.

4.3.2 Evaluation Objectives

We evaluated our placement algorithm MOPA-LPMAX with different candidate selection strategies as discussed in Subsection 3.3.3. Table 4.4 summarizes the candidate selection strategies presented in Subsection 3.3.3. *Ellipse* (EL) represents the strategy that checks all nodes inside the search ellipse. *K-Nearest Neighbour* (kNN) implements the k-nearest neighbour search with respect to the network usage minimum. *K-Random* (kRand) implements the random selection strategy, which selects random hosts that reside inside the ellipse. Finally, *Conditional K-Nearest Neighbor* (CkNN) implements the pruned search according to the pruning criterion presented earlier. For each candidate selection strategy, we measure the resulting end-to-end latency, the network usage, and the communication overhead to discover the candidate hosts.

For the evaluation of MOPA-LPMAX, we have focused on the quality and communication overhead. In particular, we compare the four different selection strategies presented in Subsection 3.3.3 in terms of the resulting latency and network usage, and we measure the messages exchanged to candidate hosts. Table 4.5 summarizes the performance metrics used for our evaluation. For the quantification of latency and network usage, we have used absolute values as we discuss in the next subsections.

4.3.3 Quality: Processing and Network Latency

In the first experiment, we evaluate the QoS capabilities of our placement algorithm with the different candidate selection strategies. In order to explore the limitations of the different strategies, we consider an extreme case with very hard latency constraints: By setting the latency constraint to zero, we let the placement algorithm search for the operator placement with minimum possible latency. We deployed up to 240 operators gradually and measured the achieved latency for each candidate selection strategy.

Figure 4.22 shows the achieved latency over the number of deployed oper-

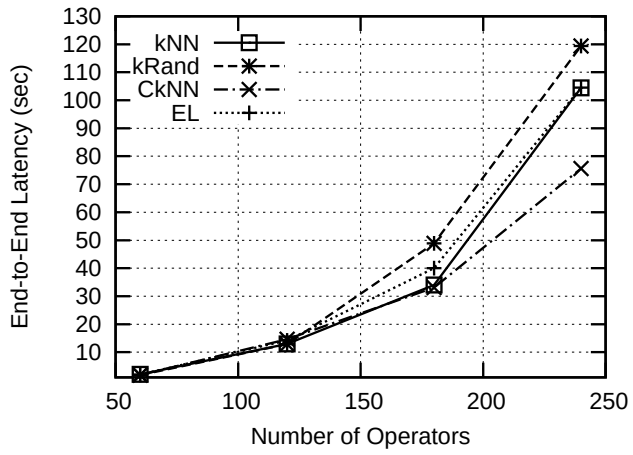


Figure 4.22: Network and Processing Latency for increasing number of operators.

ators. As expected the latency increases with the number of deployed operators since the system load increases. Initially all methods almost perform similarly since initially the system has no load and all hosts can execute the operators with the same expected (low) delay. In this case, the solution is mainly defined by the network latency and not by the processing delay.

As the number of operators increases, some hosts get more load and become slower in comparison to other hosts. In that case, the latency of the random strategy kRand increases faster compared to the other strategies since it selects randomly hosts inside the ellipse. The greedy strategy kNN is more resilient to the load but finally deviates also significantly from CkNN up to 38% since it only considers a limited set of hosts in the vicinity of the network usage minimum. Another interesting result is that the approach that searches all nodes in the ellipse (EL) performs similarly to the greedy kNN strategy, without achieving the best result.

This behaviour can be explained given the absolute values of the processing delays. As discussed earlier, for operators with matrix size 500, the processing delay is expressed in seconds. Since the communication latency in the latency

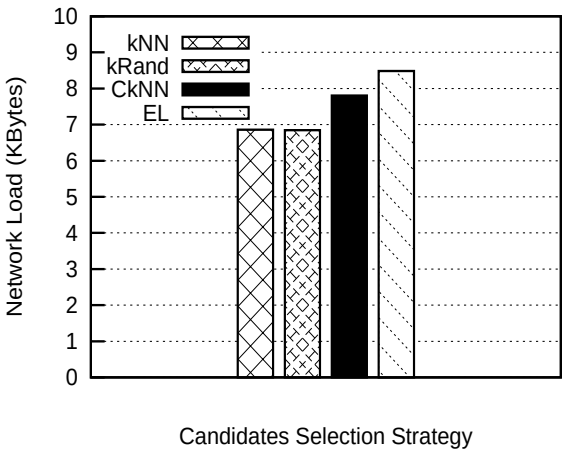


Figure 4.23: Resulting network usage for candidate selection.

space is typically expressed in milliseconds, the resulting ellipse in such case would lead to an exhaustive search over the network, where the physical nodes will be evaluated mainly based on their processing delay. Thus, given the expected error of the processing model between 11% and 20% depending on the operator complexity as presented in 3.3.1, this heuristic could be disoriented while trying to find the fastest physical node in the network.

4.3.4 Quality: Network Usage

Next, we analyse the performance of the different candidate selection strategies in terms of network usage. As already mentioned in Section 3.3.3, the different strategies try to leave out some possible solutions to limit overhead. Therefore, we expect the approaches with better QoS performance to have the higher costs in terms of network usage. For the same experiment as before, we calculate the average network load of the deployed operator graphs. In order to measure the network load, we have taken snapshots of the data that were in transit at certain points in time measured in *KBytes*. Finally, we have calculated the average data load over the time for the different strategies.

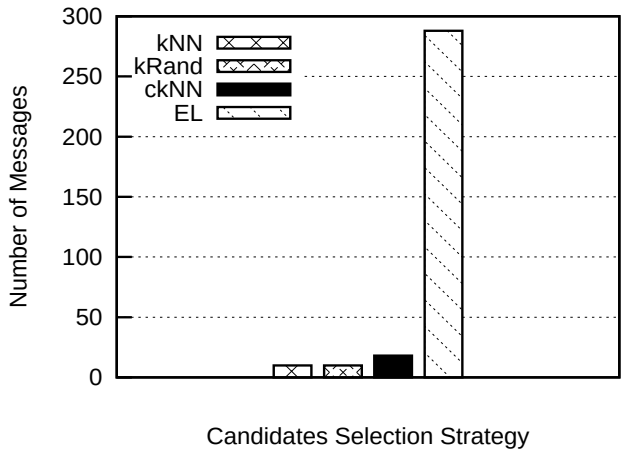


Figure 4.24: Communication Overhead.

Figure 4.23 shows the absolute values in $KBytes$ of the network load for the different candidate selection strategies. As we see in Figure 4.23 the average network usage is low $6.8KBytes$ for the greedy kNN strategy and the random strategy. That is expected, since these approaches do not fulfill optimally the latency constraints and, therefore, can achieve a lower network usage. Moreover, CkNN induces 14% greater network load ($7.8KBytes$) compared to kNN and random strategy- However, given that CkNN provides a 38% lower latency on average, it still achieves a good balance between the network usage minimization and the fulfilment of the latency constraints. Finally, EL does not manage to find good candidate hosts and also induces high network load. This could be interpreted due to the sensitivity of EL heuristic to base its decision on the approximation of the processing delay.

4.3.5 Overhead: Messages for candidate selection methods

Finally, we discuss the communication overhead induced by each candidate selection strategy. Figure 4.24 shows the average number of messages communicated between the coordinator and other hosts in order to define a candidate

set. For kNN and kRand the number of messages are 10, since by default these strategies communicate with only $k = 5$ hosts and they need two messages (request/response) for each contact to a candidate host. For the EL algorithm using the optimal restriction in the latency space, the average number of messages is 288, with a standard deviation of 44 messages. This means that a host contacts on average 144 out of 200 hosts to decide on a placement. As already discussed, this is a result of the absolute values between the communication latency and processing delay. Thus, the induced communication overhead severely impacts the practical application of this method, due to the high communication overhead that it induces, but also because it cannot guarantee high quality solutions, since it would react extremely slow at each network change using possibly outdated delay measurements.

Finally, for CkNN the number of messages is 18, with a standard deviation of 4 messages. Thus, we see that the strategy that uses the pruning criterion not only performs better in terms of the constraint satisfaction problem, but also keeps the number of messages very low querying on average about 5% of the total hosts. In other words, we see that it is sufficient to check only a small subset of all hosts that reside in the ellipse.

4.3.6 Summary

In this section, we have presented the results for the MOPA-LPMAX algorithm by using four different candidate selection strategies. Our results show, that the conditional K-Nearest Neighbour method, which uses the pruning criterion introduced in Section 3.3.3, outperforms the simple kNearest neighbour and the random selection of k physical hosts inside the ellipse, since it achieves a 38% lower latency with the cost of 14% greater network usage. Moreover, the proposed method induces limited overhead by using only 18 messages on average to find a solution for an operator graph with two free operators.

4.4 Conclusion

The evaluation results presented in this chapter have provided insights on the performance of the proposed placement algorithms, presented in Chapter 3. Our evaluations show that MOPA achieves nearly optimal solutions (with average stretch factor of $14\% - 21\%$), depending on the size of the graph, while using only local knowledge. Although the algorithm is executed in a distributed way, the induced overhead in terms of messages exchanged and migrations is smaller than state-of-the-art method SBON.

Furthermore, MOPA is used as a baseline algorithm for solving the two constrained optimization problems presented in Chapter 3. Therefore, the two algorithms inherit the properties of MOPA in terms of quality of solution and overhead. For the network latency constraints, we see that in heterogeneous operator graphs, minimizing the network usage leads to a network latency that is significantly different to the latency minimum. In that respect, especially for scenarios where the data rates of the operator graph vary significantly, we need a constraint satisfaction algorithm that tailors the solution to meet application-defined latency constraints. We see that the success rate of our proposed algorithm MOPA-LMAX depends on the strictness of the latency constraint. For relaxed latency constraints (with latency stretch greater than 2 with respect to the minimal feasible latency), our algorithm always find a good solution achieving up to 98% success rate on average. Only if the constraint is very close to the minimal feasible latency, the success rate decreases. Moreover, MOPA-LMAX can find a solution to the network latency constrained optimization problem in a few seconds (3 seconds) even for larger operator graphs of 20 nodes, while the centralized MIP solver needs hundreds of seconds to find the optimal solution. Furthermore, in terms of accuracy, MOPA-LMAX approximates the optimal solution found by the extended ILP by 13% greater latency and 14% network usage on average.

Finally, we evaluated MOPA-LPMAX with four different selection strategies and we have identified cKNN as the best candidate selection strategy.

In our experiments, we have seen that considering the processing delay during optimization becomes more important when the system has a significant load, and when the physical nodes become heterogeneous in terms of speed. Therefore, the MOPA-LPMAX algorithm is more relevant for scenarios where the physical machines vary in terms of their computing capabilities and load.

As a conclusion of our evaluation, we see that the proposed algorithms MOPA, MOPA-LMAX, MOPA-LPMAX significantly reduce the network usage and are able to meet given latency constraints in realistic scenarios. The selection of the most appropriate algorithm between the three depends on the properties of the scenario and in particular, on the heterogeneity of the operator graph, and the heterogeneity of the physical network in terms of load and computing power.

5 Summary and Future Work

Finally, we provide a brief summary of the contents of this dissertation, before we discuss possible future extensions of this work.

5.1 Summary

In this dissertation, we have presented concepts and algorithms for the efficient processing of distributed context streams. Our work has been motivated by the need of designing a system that enables distributed context reasoning. To this end, we have proposed a novel architecture for distributed context reasoning that uses the concept of the operator graph. Given this generic system model that allows for the distribution of reasoning tasks, we then formulated three operator placement problems that target different application scenarios and we presented algorithms that solve these problem considering local knowledge.

In detail, we have first formulated the Multi-operator placement problem, which seeks for an optimal placement of operators minimizing the bandwidth-delay products of the inter-operator data streams. For this problem, we have presented a distributed algorithm that allows for the autonomous placement of the operators based on their local view. As reference for our distributed algorithm, we have presented an integer linear program that solves the network usage optimization problem in a centralized way assuming global knowledge of the system. Our evaluation showed that our distributed algorithm finds near optimal solutions (on average 14% deviation from the optimum). Furthermore, the algorithm achieves higher quality solutions and induces less overhead in terms of messages and migrations with respect to another rele-

vant state-of-the-art algorithm.

Secondly, we extended the network usage optimization problem by adding an application-defined latency constraint, which considers the communication latency as the dominant factor of the end-to-end delay. For this problem, we provided a constraint satisfaction algorithm that starts from an optimal placement with respect to network usage and tries to degrade the solution minimally in terms of network usage to reach the delay constraint. Our evaluation results showed that our algorithm achieves high success rates up to 98%.

Finally, we have considered a latency-constrained optimization problem, that additionally takes processing and network transmission delays for large data items into account. Our algorithm finds promising candidate nodes that may decrease the processing or network delay. We have used different candidate selection methods depending on the proximity of the nodes to the network usage minimum and their processing delay. Our evaluation showed that the candidate selection method, which uses our proposed pruning criterion can achieve a better balance between network usage and latency with respect to other simple heuristic solutions.

5.2 Future Work

The work presented in this dissertation could be extended into different directions by considering different systems models and/or placement problems.

As an immediate extension of the presented work, one could consider the problem of optimizing a set of operator graphs rather than a single graph. Here, the concept of sharing operators between graphs becomes essential. Sharing operator can be beneficial, for instance, to reduce the computational and communication overhead (an operator only has to be executed once and its output can be re-used). However, in some cases a shared operator might prevent to find a solution meeting given latency constraints. Therefore, the placement algorithm has to be carefully designed to make the right decisions

when to share an operator.

A second extension of the presented work could be the integration of mobile nodes into the system model. Taking into account the trend of modern smart phones, such mobile nodes could serve as sensors, sinks (applications), and hosts for operators. However, there are also several challenges that have to be solved to benefit from the large crowd of available mobile nodes. For instance, energy becomes an important constraints that has to be considered during operator placement. Moreover, node mobility and availability change the model of fixed sources and sinks. Node mobility might also trigger the frequent migration of operators to constantly fulfill given end-to-end latency constraints. First steps into this direction have already been taken in another work at the University of Stuttgart [79]. This work also shows that it might be beneficial to consider modern execution environments such as powerful compute clouds or edge servers close to the mobile devices and therefore available with small latency.

Another possible research direction is the design of optimal placement strategies for application models other than stream processing. In particular, it would be interesting to investigate placement algorithms for multi-tier applications using a request/response model. In such a system, the application consists of client/server components where the components of tier n act as clients to servers of tier $n + 1$. A typical example are web applications following a three-tier architecture: Frontend (GUI), middle tier (application logic), backend (persistent data storage; database). The middle tier itself could be split up into further tiers if application servers can be ordered according to client/server relationships between application servers (application servers use other application servers). If we consider the frontend (clients) and backend (database) to be fixed (pinned), the question is where to place the middle tier services to minimize network usage (or cost in general) and guarantee a certain maximum response time (typically tens of milliseconds for many web services)? In particular for new infrastructure models such as cloud computing environments consisting of multiple data centers, this

placement problem becomes highly relevant. Since often cloud services are driven by pay-as-you-go pricing models, minimizing communication cost and computational cost, while achieving high computing elasticity, through optimal placement strategies are of great importance. To solve this problem, our model needs to be adapted to a request/response model considering the whole round trip between frontend and backend, a dynamic set of clients, the replication of services as another degree of freedom, and further optimization goals such as minimum monetary cost or elasticity.

Finally, the optimization problems presented in this dissertation could also be extended to consider optimizations of the logical plan, i.e. the degradation of query result by load shedding or the consideration of multiple data granularities, which imply different data rates. In that case, our model should be extended to include the quality of the logical plan, to find a trade-off solution between quality and computing (and network) costs.

References

[1] Network Coordinate Research at Harvard. `http://www.eecs.harvard.edu/~syrah/nc/`.

[2] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, 2005.

[3] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[4] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *HUC 1999: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 304–307, London, UK, 1999. Springer-Verlag.

[5] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[6] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD 2008, pages 147–160, 2008.

[7] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC 19Distributed Stream Management using Utility-Driven Self- Adaptive Middleware,99*, 1999.

[8] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *VLDB 2004*, pages 456–467, 2004.

[9] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. *Proc. VLDB Endow.*, 1(1):66–77, August 2008.

[10] J. Al-Muhtadi, Shiva Chetan, A. Ranganathan, and R. Campbell. Super spaces: a middleware for large-scale pervasive computing environments. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 198–202, 2004.

[11] L. Amini, N. Jain, Anshul Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS 2006. 26th IEEE International Conference on Distributed Computing Systems, 2006.*, page 71, 2006.

[12] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. Stream: The stanford stream data manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[13] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[14] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, CIKM 2006, pages 337–346, 2006.

[15] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *CIDR 2007*, pages 363–374, 2007.

[16] Martin Bauer. *Observing Physical World Events through a Distributed World Model.* Dissertation, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, May 2007.

[17] Luca Becchetti, Ioannis Chatzigiannakis, and Yiannis Giannakopoulos. Streaming techniques and data aggregation in networks of tiny artefacts. *Computer Science Review*, 5(1):27 – 46, 2011.

[18] Andreas Benzing, Boris Koldehofe, and Kurt Rothermel. Efficient support for multi-resolution queries in global sensor networks. In *Proceedings of the 5th International Conference on Communication System Software and Middleware*, COMSWARE 2011, pages 11:1–11:12, New York, NY, USA, 2011. ACM.

[19] A. Bikakis and G. Antoniou. Defeasible contextual reasoning with arguments in ambient intelligence. *Knowledge and Data Engineering, IEEE Transactions on*, 22(11):1492–1506, 2010.

[20] Prosenjit Bose, Anil Maheshwari, and Pat Morin. Fast approximations for sums of distances, clustering and the Fermat-Weber problem. *Computational Geometry: Theory and Aplications*, 24:135–146, 2002.

[21] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD 2007, pages 1100–1102, 2007.

[22] Barry Brumitt, Brian Meyers, John Krumm, A Kern, and Steven Shafer. Easyliving: Technologies for intelligent environments. In *Proceedings of the 2nd international symposium on Handheld and Ubiquitous Computing*, pages 12–29. Springer-Verlag, 2000.

[23] P. Calamai and Charalambous C. Solving multifacility location problems involving euclidean distances. *Naval Research Logistics Quarterly*, 27(4):609–620, 1980.

[24] B.W. Carabelli, A. Benzing, F. Durr, B. Koldehofe, K. Rothermel, G. Seyboth, R. Blind, M. Burger, and F. Allgower. Exact convex formulations of network-oriented optimal operator placement. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 3777–3782, 2012.

[25] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001.

[26] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14:1 – 26, 1994.

[27] R. Chandrasekaran and A. Tamir. Algebraic optimization: the Fermat-Weber location problem. *Math. Program.*, 46(2):219–224, 1990.

[28] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy,

Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD 2003: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 668–668, New York, NY, USA, 2003. ACM.

[29] Harry Chen, Tim Finin, and Anupam Joshi. Semantic web in the context broker architecture. In *PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04)*, page 277, Washington, DC, USA, 2004. IEEE Computer Society.

[30] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: a scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD 2000, pages 379–390, New York, NY, USA, 2000. ACM.

[31] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: An overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, pages 3–12, 2003.

[32] Nazario Cipriani, Mike Eissele, Andreas Brodt, Matthias Grossmann, and Bernhard Mitschang. NexusDS: a flexible and extensible middleware for distributed stream processing. In *IDEAS 2009: Proceedings of the 2009 International Database Engineering; Applications Symposium*, pages 152–161, New York, NY, USA, 2009. ACM.

[33] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD 2003, pages 647–651, 2003.

[34] Gianpaolo Cugola and Alessandro Margara. Raced: an adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, ARM 2009, pages 5:1–5:6, 2009.

[35] Gianpaolo Cugola and Alessandro Margara. Tesla: a formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS 2010, pages 50–61, 2010.

[36] Gianpaolo Cugola and Alessandro Margara. Complex event processing with t-rex. *J. Syst. Softw.*, 85(8):1709–1728, August 2012.

[37] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM 2004: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 15–26, New York, NY, USA, 2004. ACM.

[38] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, March 1988.

[39] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Bandwidth-constrained queries in sensor networks. *The VLDB Journal*, 17(3):443–467, May 2008.

[40] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Hum.-Comput. Interact.*, 16(2):97–166, December 2001.

[41] Z. Drezner and H.W. Hamacher. *Facility Location: Applications and Theory*. Springer, 2004.

[42] Dominique Dudkowski, Harald Weinschrott, and Pedro José Marrón. Design and implementation of a reference model for context management in mobile ad-hoc networks. In *Proc. of AINA Workshops*, 2008.

[43] D. Ejigu, M. Scuturici, and L. Brunie. Coca: A collaborative context-aware service platform for pervasive computing. In *Information Technology, 2007. ITNG 2007. Fourth International Conference on*, pages 297–302, April.

[44] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[45] Minos Garofalakis. Distributed data streams. In LING LIU and M.TAMER Ã–ZSU, editors, *Encyclopedia of Database Systems*, pages 883–890. Springer US, 2009.

[46] Minos Garofalakis and Phillip B. Gibbons. Wavelet synopses with error guarantees. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 476–487, New York, NY, USA, 2002. ACM.

[47] Narain H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, VLDB 1991, pages 327–336, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[48] A. Grau, K. Herrmann, and K. Rothermel. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. In *18th Internatonal Conference on Computer Communications and Networks*, Aug. 2009.

[49] Tao Gu, H.K. Pung, and Da Qing Zhang. A middleware for building context-aware mobile services. In *Vehicular Technology Conference, 2004. VTC 2004-Spring. 2004 IEEE 59th*, volume 5, pages 2656–2660 Vol.5, May.

[50] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A service-oriented middleware for building context-aware services. *J. Netw. Comput. Appl.*, 28(1):1–18, 2005.

[51] Tao Gu, Hung Keng Pung, and Daqing Zhang. Peer-to-peer context reasoning in pervasive computing environments. In *PERCOM '08: Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 406–411, Washington, DC, USA, 2008. IEEE Computer Society.

[52] Xiaohui Gu, Philip S. Yu, and Klara Nahrstedt. Optimal Component Composition for Scalable Stream Processing. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS 2005, 2005.

[53] D. Gyllstrom, J. Agrawal, Yanlei Diao, and N. Immerman. On supporting kleene closure over event streams. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 1391–1393, 2008.

[54] Karen Henricksen, Jadwiga Indulska, Ted McFadden, and Sasitharan Balasubramaniam. Middleware for distributed context-aware systems. In *International Symposium on Distributed Objects and Applications (DOA*, pages 846–863. Springer, 2005.

[55] T. Hofer, W. Schwinger, M. Pichler, G. Leonhartsberger, J. Altmann, and W. Retschitzegger. Context-awareness on mobile devices - the hydrogen approach. In *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, page 10 pp., jan. 2003.

[56] Nicola Hönle, Matthias Großmann, Daniela Nicklas, and Bernhard Mitschang. Preprocessing position data of mobile objects. In *Proc. of MDM*, 2008.

[57] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. Towards a streaming sql standard. *Proc. VLDB Endow.*, 1(2):1379–1390, August 2008.

[58] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. Probabilistic event extraction from rfid data. In *ICDE*, pages 1480–1482, 2008.

[59] Gerald Koch, Boris Koldehofe, and Kurt Rothermel. Cordies: Expressive Event Correlation in Distributed Systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, 2010.

[60] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, RobertE. Bixby, Emilie Danna, Gerald Gamrath, AmbrosM. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, DanielE. Steffy, and Kati Wolter. Miplib 2010. *Mathematical Programming Computation*, 3(2):103–163, 2011.

[61] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.-J. Malm. Managing context information in mobile devices. *Pervasive Computing, IEEE*, 2(3):42–51, July-Sept.

[62] Niels Rode Kristensen, Henrik Madsen, and Sten Bay JøRgensen. Parameter estimation in stochastic grey-box models. *Automatica*, 40(2):225–237, February 2004.

[63] Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement strate-

gies for internet-scale data stream systems. *Internet Computing, IEEE*, 12(6):50–60, Nov.-Dec. 2008.

[64] Ralph Lange. *Scalable Management of Trajectories and Context Model Descriptions*. Dissertation, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Germany, Dezember 2010.

[65] Ralph Lange, Nazario Cipriani, Lars Geiger, Matthias Grossmann, Harald Weinschrott, Andreas Brodt, Matthias Wieland, Stamatia Rizou, and Kurt Rothermel. Making the world wide space happen: New challenges for the nexus context platform. *Pervasive Computing and Communications, IEEE International Conference on*, 0:1–4, 2009.

[66] Ralph Lange, Frank Dürr, and Kurt Rothermel. Online trajectory data reduction using connection-preserving dead reckoning. In *Proc. of MobiQuitous*, 2008.

[67] Ralph Lange, Frank Dürr, and Kurt Rothermel. Scalable processing of trajectory-based queries in space-partitioned moving objects databases. In *Proc. of ACM GIS*, 2008.

[68] Pierre Le Bodic, Pierre HéRoux, SéBastien Adam, and Yves Lecourtier. An integer linear program for substitution-tolerant subgraph isomorphism and its use for symbol spotting in technical drawings. *Pattern Recogn.*, 45(12):4214–4224, December 2012.

[69] Guoli Li and Hans-Arno Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, Middleware 2005, pages 249–269, 2005.

[70] Ling Liu, C. Pu, and Wei Tang. Continual queries for internet scale

event-driven information delivery. *Knowledge and Data Engineering, IEEE Transactions on*, 11(4):610–628, 1999.

[71] David C. Luckham. Rapide: a language and toolset for simulation of distributed systems by partial orderings of events. In *Proceedings of the DIMACS workshop on Partial order methods in verification*, POMIV 1996, pages 329–357, 1997.

[72] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.

[73] Masoud Mansouri-Samani and Morris Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96, 1997.

[74] Alessandro Margara and Gianpaolo Cugola. Processing flows of information: from data stream to complex event processing. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS 2011, pages 359–360, New York, NY, USA, 2011. ACM.

[75] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[76] Sumedh Mungee, Nagarajan Surendran, and Douglas C. Schmidt. The Design and Performance of a CORBA Audio/Video Streaming Service. In *Hawaiian International Conference on System Sciences*, 1999.

[77] Daniela Nicklas, Matthias Großmann, Thomas Schwarz, Steffen Volz, and Bernhard Mitschang. A model-based, open architecture for mobile, spatially aware applications. In *Proc. of SSTD*, 2001.

[78] Petteri Nurmi, Michael Przybilski, Greger Lindén, and Patrik Floréen. An architecture for distributed agent-based data preprocessing. In *AIS-ADM*, pages 123–133, 2005.

[79] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 1–12. ACM Press, Juni 2013.

[80] Sebastian Padó and Mirella Lapata. Constructing semantic space models from parsed corpora. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 128–135, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[81] Olga Papaemmanouil, Yanif Ahmad, Ugur Çetintemel, and John Jannotti. Application-aware Overlay Networks for Data Dissemination. In *ICDE Workshops*, page 76, 2006.

[82] Kostas Patroumpas and Timos Sellis. Multi-granular time-based sliding windows over data streams. In *Proceedings of the 2010 17th International Symposium on Temporal Representation and Reasoning*, TIME '10, pages 146–153, Washington, DC, USA, 2010. IEEE Computer Society.

[83] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann, September 1988.

[84] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE 2006: Proceedings of the 22nd International Conference on Data Engineering*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

[85] Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. [evaluating dht-based service placement for stream-based overlays.

[86] Peter R. Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware 2003, pages 62–82, 2003.

[87] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, 2004.

[88] Francisc Rado. The euclidean multifacility location problem. *Operations Research*, 36(3):485–492, 1988.

[89] Anand Ranganathan and Roy H. Campbell. A middleware for context-aware agents in ubiquitous computing environments. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware 2003, pages 143–161, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[90] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing Aware Component Composition for Distributed Stream Processing Systems. In *Middleware*, pages 322–341, 2006.

[91] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Providing QoS Guarantees for Large-Scale Operator Networks. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, pages 337–345, Melbourne, VIC, Australia, September 2010. IEEE Computer Society Press.

[92] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Solving the Multi-operator Placement Problem in Large Scale Operator Networks. In *19th Internatonal Conference on Computer Communications and Networks*, 2010.

[93] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Fulfilling End-to-End Latency Constraints in Large-scale Streaming Environments. In

Proceedings of the 30th IEEE International Performance Computing and Communications Conference: IPCCC'11, pages 1–8. IEEE Xplore, November 2011.

[94] Stamatia Rizou, Kai Häussermann, Frank Dürr, Nazario Cipriani, and Kurt Rothermel. A System for Distributed Context Reasoning. In *ICAS 2010*, pages 84–89, 2010.

[95] Manuel Román, Christopher Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, October 2002.

[96] J. B. Rosen and G. L. Xue. On the Convergence of Miehleś Algorithm for the Euclidean Multifacility Location Problem. *Operations Research*, 40(1):188–191, 1992.

[97] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[98] Dimitris Sacharidis, Antonios Deligiannakis, and Timos Sellis. Hierarchically compressed wavelet synopses. *The VLDB Journal*, 18(1):203–231, January 2009.

[99] B. Schilling, B. Koldehofe, and K. Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In *High Performance Computing and Communications (HPCC), 2011 IEEE 13th International Conference on*, pages 355–364, 2011.

[100] Roman Schmidt and Karl Aberer. Efficient Peer-to-Peer Belief Propagation. In *Fourteenth International Conference on Cooperative Information Systems (CoopIS)*, 2006.

[101] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS 2009, pages 4:1–4:12, 2009.

[102] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 25–36, 2003.

[103] A. Sharaf, Jonathan Beaver, Alexandros Labrinidis, and K. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *The VLDB Journal*, 13(4):384–403, December 2004.

[104] Jonas Sjöberg, Qinghua Zhang, Lennart Ljung, Albert Benveniste, Bernard Deylon, Pierre yves Glorennec, Hakan Hjalmarsson, and Anatoli Juditsky. Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31:1691–1724, 1995.

[105] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *In PODS*, pages 250–258, 2005.

[106] Mark Sullivan and Andrew Heybey. Tribeca: a system for managing large databases of network traffic. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC 1998, pages 2–2, Berkeley, CA, USA, 1998. USENIX Association.

[107] Egemen Tanin, Deepa Nayar, and Hanan Samet. An efficient nearest neighbor algorithm for P2P settings. In *Proceedings of the 2005 National Conference on Digital Government Research*, pages 21–28. Digital Government Society of North America, 2005.

[108] William van Dorst. The quintessential linux benchmark: All about the "bogomips" number displayed when linux boots. *Linux J.*, 1996(21es), January 1996.

[109] Matthias Wieland, Oliver Kopp, Daniela Nicklas, and Frank Leymann. Towards Context-Aware Workflows. In *Proc. of CAiSE*, 2007.

[110] Matthias Wieland, Daniela Nicklas, and Frank Leymann. Managing technical processes using smart workflows. *ServiceWave*, December 2008. to appear.

[111] Gregory Aaron Wilkin, K. R. Jayaram, Patrick Eugster, and Ankur Khetrapal. Faidecs: fair decentralized event correlation. In *Proceedings of the 12th ACM/IFIP/USENIX international conference on Middleware*, Middleware 2011, pages 228–248, Berlin, Heidelberg, 2011. Springer-Verlag.

[112] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *SIGMOD 2006*, 2006.

[113] Kirsten W. Wu, Kun-Lung, Wei Fan, Philip S. Yu, Charu C. Aggarwal, David A. George, Buğra Gedik, Eric Bouillet, Xiaohui Gu, Gang Luo, and Haixun Wang. Challenges and experience in prototyping a multi-modal stream analytic and monitoring application on system s. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB 2007, pages 1185–1196, 2007.

[114] Ying Xing, Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE 2005, pages 791–802, Washington, DC, USA, 2005. IEEE Computer Society.

[115] Yong Yao and Johannes Gehrke. The cougar approach to in-network

query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, September 2002.

[116] Stephen S. Yau and Fariaz Karim. Context-sensitive middleware for real-time software in ubiquitous computing environments. In *Proc. 4 th IEEE International Symp. on Object-Oriented Real-time Distributed Computing (ISORC 2001*, pages 163–170, 2001.

[117] Lei Ying, Zhen Liu, D. Towsley, and C.H. Xia. Distributed operator placement and data caching in large-scale sensor networks. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 977–985, 2008.

[118] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System. In *Proceedings of the 2006 Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I*, ODBASE'06/OTM'06, pages 54–71, 2006.

[119] Horst W. Hamacher Zvi Drezner. The Fermat-Weber Problem. In *Facility Location: Applications and Theory*, pages 1–24. 2005.

[120] Oliver Zweigle, Kai Häussermann, Uwe-Philipp Käppeler, and Paul Levi. Extended TA Algorithm for adapting a Situation Ontology. In *Proceedings of the FIRA RoboWorld Congress 2009, Progress in Robotics*, volume 44 of *Communications in Computer and Information Science*, pages 364–371, Incheon, Korea, August 2009. Springer Verlag.