Providing QoS Guarantees in Large-Scale Operator Networks

Stamatia Rizou, Frank Dürr, Kurt Rothermel
Universität Stuttgart, Institute of Parallel and Distributed Systems, Universitätsstraße 38, 70569 Stuttgart, Germany
<firstname.lastname>@ipvs.uni-stuttgart.de

***Abstract*—Application areas like global sensor networks and data stream processing involve the on-line processing of large amounts of data in an overlay network of operators on top of the Internet infrastructure. Trying to fulfill QoS guarantees in such networks is a challenging task that should be realized under the requirement for optimal usage of common resources in the network. Therefore in this paper, we formalize a constrained optimization problem for the placement of operators in an overlay network which strives for satisfying user QoS constraints subject to latency, while minimizing the network load induced by the deployment of the operators in the network. Since the initial problem is NP-hard, we solve at a first step the problem in an intermediate continuous latency space and then we map the continuous solution to its discrete variant. Our evaluations provide an analysis about the inherent interdepedence between the two metrics, network usage and latency, subject to this paper and furthermore show that our algorithm achieves a good balance between the user requirements and the usage of the network resources.**

***Keywords*-stream processing, operator networks, quality of service (QoS), optimization, constraint, placement algorithm, overlay networks**

I. Introduction

Operator networks are a powerful abstraction to model the distributed processing of data streams. In such networks, streams of data originating from distributed sources are processed in an overlay network of operators on top of a communication infrastructure like the Internet, and the result is delivered to a set of applications (sinks). Operators enclose the functionality that is specific to the application. For instance, the operators of a large-scale sensor network [3, 5] might implement functions such as data selection and aggregation in order to jointly filter and process streams of sensor data. Other application areas include complex event processing [13] and distributed reasoning systems [11].

In large-scale operator networks, the placement of operators onto physical hosts has to be chosen carefully since it strongly impacts the performance of the system. On the one hand, the operator placement determines the scalability of the system with respect to the load put onto physical resources, namely processing load of operator hosts and communication load of network links. On the other hand, the placement influences the quality of service (QoS) delivered to the application, for instance, the end-to-end delay of messages transmitted through the overlay network from the sources to the applications.

In general, the operator placement problem can be seen as a *constrained optimization problem* that optimizes resource usage to maximize the scalability under certain QoS constraints specified by the application. In this paper, we consider a specific constrained optimization problem. As optimization criteria, we consider the *network usage* denoted by the bandwidth-delay product of inter-operator data streams, that is a measure for the network load induced by the deployed operator networks. Obviously, the minimization of this metric minimizes the stress of communication resources. Therefore, it is particular well-suited for applications producing large amounts of small data items like global sensor networks that require the in-network processing of data produced by many sensors distributed in a wide-area network. As QoS constraint we consider the end-to-end delay of messages between sources and 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 timely manner to sensor information to control physical processes.

The main contribution of this paper is an operator placement algorithm that guarantees a given end-to-end delay while minimizing the network usage. Our algorithm is based on a two-phase process. First, we find an operator placement that minimizes network usage (*unconstrained optimization phase*). Second, 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 [6, 7, 9]. In constrast 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 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 favor of a less costly solution.

Since our constrained optimization problem is an NP-hard problem, we provide a heuristic solution. We first define

Published in International Conference on High Performance Computing and Communications @IEEE 2010
DOI 10.1109/HPCC.2010.53

the problem in a continuous search space called the *latency space* that models the communication latencies between nodes. First, we solve the uncostrained optimization problem in the continuous space as described in [10]. Then, during the constraint satisfaction phase we calculate 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 distrete set of physical nodes.

We will both show, how operators can be placed at the initial deployment, and how operator positions can be adapted to dynamic network conditions during runtime. Moreover, we will show in extensive simulations that our heuristic achieves a good approximation of the optimal solution.

The rest of the paper is structured as follows: In Section II we discuss the related work. In Section III we introduce our system model. In Section IV we formalize our problem and then in Section V we propose an algorithm that solves this constrained optimization problem. In Section VI, we discuss the adaptation of our solution to dynamic changes and finally in Section V-D we show the results of our evaluations before we conclude our work in Section VIII.

II. Related Work

Different operator placement algorithms have been proposed in the literature to address different metrics, such as load balancing or minimal network usage. In our previous work [10] we have presented an algorithm for minimizing network usage based on the idea of the latency space that was initially introduced by Pietzuch et al. [8], who proposed a placement algorithm called SBON. Both of these algorithms solve an unconstrained optimization problem that strives for minimizing network usage using the continuous latency space. However, as we show in [10] our approach outperforms SBON both in the quality of the result as well as in the communication overhead and migrations induced by the placement algorithm. Therefore, we are going to use this method for calculating the unconstrained optimum.

Other placement algorithms that consider QoS restrictions use algorithms, which first prune the search space by finding a set of candidate nodes and then find alternate placements by calculating all possible placements starting from the leaves and combining the candidate solutions in a bottom-up fashion. Gu et al. [6] proposed an algorithm that optimizes for load balancing while considering latency constraints. This algorithm first collects global information about all physical nodes in order to define a set of candidate nodes that can satisfy QoS requirements and then selects the best plan according to a congestion aggregation metric that models load balancing. Apart from having a different optimization metric, in our approach we do not need to check exhaustively all the nodes in order to find good candidates.

Other placement algorithms such as [9], [7] consider re-use of operators in order to reduce the end-to-end latency. Although the problem of operator re-use is a challenging problem, in this work we focus on the indepedent placement of the stream processing tasks and we leave as future work the extension of our algorithm to consider operator re-use.

III. System Model

In this section, we introduce our resource model and present the execution model for stream processing.

In the underlying system, we assume a set of physical nodes V distributed in the network that are able to host operators. Here we follow the idea of the latency space proposed initially by Pietzuch et al. [8], where each physical host is assigned a coordinate in an n-dimensional Cartesian space, such that the Euclidean distance between two nodes in the latency space models the corresponding latency between the physical hosts in the real world. In our context, we model latency as the propagation delay $L(\overline{\nu_i\nu_j})$ between two physical hosts ν_i, ν_j together with the packet processing and queuing delays, without considering the transmission delay to put the data "on the wire". This is a reasonable assumption when messages with small sizes such as simple sensor values are sent over long distances in the network. The coordinates of the physical nodes $P(V)$ can be determined efficiently in a distributed manner by algorithms that calculate network coordinates using delay measurements between physical hosts [4]. We assume that every physical node uses such an algorithm to determine its position in the latency space.

In our execution model a stream processing task is modeled as a directed operator graph $G = \{\Omega, S, A, P\}$ that consists of a set $\Omega = \{\omega_1, \ldots, \omega_n\}$ of *operators* that are connected by a set $L = \{\overline{\omega_1\omega_i}, \ldots, \overline{\omega_j\omega_n}\}$ of *links*. Set $S \subset \Omega$ is a set of special operators called *sources*, which generate data streams; therefore they have only outgoing links. Set $A \subset \Omega$ denotes the applications (sinks) that consume the final output data streams. Typically sources and sinks have fixed positions in the network (pinned operators), whereas the remaining operators of the graph can be placed on any host in V (unpinned operators). 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. If there is no direct connection between ω_i and ω_j the notation $\overline{\omega_i\omega_j}$ denotes the coressponding path, i.e. the union of links that connects the two operators. Each graph contains a set of end-to-end *paths* $P = \{\overline{\omega_i\omega_j}, \ldots, \overline{\omega_k\omega_l}\}$, where each end-to-end path $\overline{\omega_i\omega_j}$ connect a source operator $\omega_i \in S$ with one of the sinks $\omega_j \in A$ of the operator graph.

If we do not allow the re-use of operators, we get a special case of the operator graph, called operator tree T. Operator trees are hierarchical structures that have only one sink. Since operator re-use would raise a number of problems that are not within the scope of this paper, we are going to consider only operator trees for the rest of the paper.

IV. Problem Statement

Our goal is to find an operator placement to 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.

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 $r(\overline{\omega_i\omega_j})$ specifies the data rate of the stream communicated over that link and $L(\overline{\omega_i\omega_j})$ is the delay of that link. The link delay is defined by $L(\overline{\omega_i\omega_j}) = |\vec{x}_{\omega_i} - \vec{x}_{\omega_j}|$, where $\vec{x}_\omega$ denotes the position of the operator ω in the latency space.

The so-called *Single-operator Placement (SOP) Problem* and *Multi-operator Placement (MOP) Problem* [10] consider the problem of optimal placement of a single unpinned operator or multiple operators of an operator tree, respectively, subject to network usage. In particular, a SOP problem seeks for the optimal position of an upinned operator with fixed neighbors, while the MOP problem considers the optimal placement of all unpinned operators in an operator tree T w.r.t. network usage. More formally, for a given placement $(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ T's network usage is defined by $U_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}) = \sum_{\overline{\omega_i\omega_j} \in L} r(\overline{\omega_i\omega_j})L(\overline{\omega_i\omega_j})$. This expresses the function to be minimized, i.e. our optimization goal. Network usage intuitively expresses the network load put on the network links of the system. Therefore its minimization leads to the efficient usage of network resources and subsequently contributes to the scalability of the system.

In addition to this optimization goal, we introduce the objective function to express the constraints in terms of latency. As a first step 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 $L(\overline{\omega_i\omega_j}) = \sum_{\overline{\omega_k\omega_l} \in \overline{\omega_i\omega_j}} L(\overline{\omega_k\omega_l})$. Note that normally each operator introduces a certain processing delay. As we have mentioned earlier we consider applications that send small messages over long distances. Therefore we consider the delay to process these small messages to be negligible in comparison with the propagation delay.

Then, the latency of an operator tree will be the maximum latency of all the end-to-end paths of T: $L(T) = L(\overline{\omega_i\omega_j}, \ldots, \overline{\omega_k\omega_l}) = \max_{\overline{\omega_i\omega_j} \in P} L(\overline{\omega_i\omega_j})$. Finally we define our *constrained optimization* problem as follows:

$$U_{\text{global}}(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n}) = min, \text{s.t. } L(T) \leq C \qquad (1)$$

,where C is a user defined constraint for the maximum delay.

The problem can be solved for *virtual* and *physical* nodes, resulting in *continuous* or *discrete* variations of the problem respectively. In the continuous variation, we assume that there exists a virtual node at every possible position in the latency space, i.e., $\vec{x}_\omega \in \Re^3$. Whereas in the discrete case, the operators can only be mapped to those positions in the latency space that are occupied by physical nodes, i.e. $\vec{x}_\omega \in P(V)$. As we see later, the continuous problem will help us to efficiently calculate an approximation for the initial NP-Hard discrete problem.

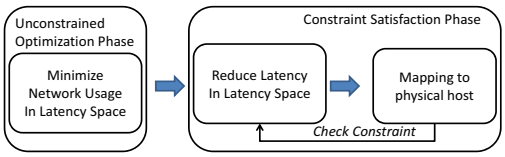


Figure 1. Process flow of the intial placement.

V. Constrained Optimization Algorithm

Next we present our approach for solving the constrained optimization problem of Equ. 1. First we give an overview of the whole process and then we describe in detail how operator positions are calculated.

A. Constrained Optimization Process Overview

The whole process of our constrained optimization method consists of two phases as shown in Fig. 1. 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 tree becomes minimal, i.e. we solve the continuous version of the optimization problem of Equ. 1 under no latency constraint. 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. 1); 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 VI 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.

B. Unconstrained Optimization Phase

During the unconstrained optimization phase, we search the minimum of the unconstrained optimization problem of Equ. 1. A detailed description of this algorithm solving the unconstrained optimization problem was given in our previous paper[10]. Here we only give a short decription to keep the paper self-contained.

To find the minimum network usage for the whole operator tree, we let each operator autonomously find its optimal position in the latency space according to the current position of the neighboring operators. Each operator indepedently calculates a solution of the continuous SOP problem by running an optimization algorithm based on a gradient method and then it maps the continuous solution to the discrete set of the physical hosts by performing a nearest neighbor query. Subsequently the operator communicates its new position in the latency space to its neighboring operators and the neighbors optimize their own positions based on the updated neighbor coordinates. This process is done repeatedly until operator positions do not change anymore, i.e. the system has converged to a global solution for the whole operator tree.

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 VI.

C. Constraint Satisfaction Phase

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 tree might violate the given delay constraint. Next we present the constraint satisfaction algorithm 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 immediatelly 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 constraint satisfaction algorithm shown in Algorithm 1 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 doing a mapping to physical hosts, we only consider non-overloaded physical hosts to prevent bottlenecks. Then, we check whether latency of the operator tree T fulfills the given latency constraint (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 tree 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 sub-section V-D. 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.

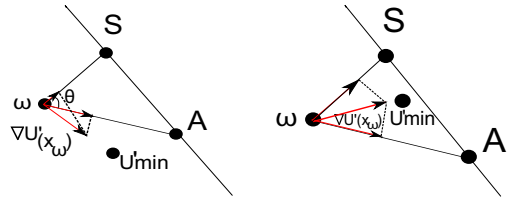
Algorithm 1 Constraint Satisfaction Algorithm

Require: $U(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal
Ensure: Finds a mapping $(\nu_1, \ldots, \nu_n)$ such that $L(T) \leq C$ and $U(\vec{x}_{\omega_1}, \ldots, \vec{x}_{\omega_n})$ is minimal
1: map each operator ω_i to closest non-overloaded ν_i
2: **while** $(L(T) > C)$ **do**
3: determine maximum latency path $\overline{\omega_i \omega_j}$
4: select operator $\omega \in \overline{\omega_i \omega_j}$ and direction $\overrightarrow{dir}$ to move
5: **if** $\overrightarrow{dir} = 0$ **then** {already at latency minimum}
6: **return** current mapping $(\nu_1, \ldots, \nu_n)$
7: **end if**
8: move operator ω by a step length $step$ into $\overrightarrow{dir}$
9: map operator ω to closest non-overloaded ν_{new}
10: **end while**
11: **return** current mapping $(\nu_1, \ldots, \nu_n)$

The step size 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 neighbor search. For the implementation of this search, we can utilize for instance a peer-to-peer infrastructure such as [12], where the physical nodes manage their positions in the latency space in a distributed peer-to-peer system. 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.

D. 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 with the minimal impact on the network usage.



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

Figure 2. Direction of the movement

More formally, we first search for a direction $\overrightarrow{dir}$ to move each unpinned operator ω on the maximum latency path, such that: (1) $L(\overline{\omega_i\omega_j})$ is reduced, (2) the increase of $U(\vec{x}_\omega)$ 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 $\phi_\omega(\overrightarrow{dir}) = \nabla U(\vec{x}_\omega) \cdot \overrightarrow{dir}$, where $\cdot$ denotes the dot product[1] of the network usage gradient $\nabla U(\vec{x_\omega})$ and the direction of the movement $\overrightarrow{dir}$. 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 inverse 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 $\phi_\omega(\overrightarrow{dir}) = \max_{\overrightarrow{dir}\in D}\{\nabla U(\vec{x}_\omega) \cdot \overrightarrow{dir}\}$. Since $L(\overline{\omega_i\omega_j})$ is a convex function, moving into the direction of $L_{\min}$ will 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. 2(a) 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 only affected by the projection of the network usage gradient $\nabla U'(\vec{x_\omega})$ on the plane $S\omega A$. Thus, in Fig. 2(a) 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. In general, we can distinguish two different cases according to the position of the projection of the network usage minimum $U'_{\min}$ on the plane $S\omega A$: (1) $U'_{\min}$ is outside the triangle $\Delta S\omega A$ (Fig. 2(a)). In this case, the 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 $\nabla U'(\vec{x_\omega})$ on the plane $S\omega A$ and therefore maximizes the dot product, i.e. $\overrightarrow{dir} = \arg\{\max_{\overrightarrow{dir}\in\{u(\overrightarrow{\omega A}),u(\overrightarrow{\omega S})\}} \nabla U'(\vec{x_\omega}) \cdot \overrightarrow{dir}\}$, where u denotes the unit vector. (2) $U'_{\min}$ is inside the triangle $\Delta S\omega A$ (Fig. 2(b)). In this case, the direction $\overrightarrow{dir}$ should be the direction of the projection of the gradient $\nabla U'(\vec{x_\omega})$, since this will induce a maximal decrease of network usage.

In order to distinguish between the two cases, we have to identify when $U'_{\min}$ is inside the triangle. As we see in the example of Fig. 2(b), $U'_{\min}$ is inside the triangle when vector $\overrightarrow{\omega A}$ and $\overrightarrow{\omega S}$ are on different sides of $\nabla U'(\vec{x_\omega})$. This condition (the relative position of $\nabla U'(\vec{x_\omega})$) cannot be identified only through the dot product. Therefore, we need to calculate the cross products[2] $\nabla U'(\vec{x_\omega}) \times \overrightarrow{\omega A}$ and $\nabla U'(\vec{x_\omega}) \times \overrightarrow{\omega S}$. 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 $\nabla U'(\vec{x_\omega})$, then their cross products $\nabla U'(\vec{x_\omega}) \times u(\overrightarrow{\omega A})$, $\nabla U'(\vec{x_\omega}) \times u(\overrightarrow{\omega S})$ have different directions, i.e. the dot product of their cross products are negative.

Algorithm 2 shows the final algorithm that we use to determine the operator 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). Thus, we first check if the projection of the network usage minimum is inside the triangle, i.e. the dot 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 $\nabla U'(\vec{x_\omega}) \cdot u(\overrightarrow{\omega A}), \nabla U'(\vec{x_\omega}) \cdot u(\overrightarrow{\omega S})$ (line 6-7). Finally, we compare ϕ_ω to the current maximum dot product of the path and if this exceeds the 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.

VI. Dynamic adaptation of placement

After the initial placement of operators, the operator tree 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.

[1]The dot product of two vectors $\vec{A} = \{A_x, A_y, \ldots, A_d\}$ and $\vec{B} = \{B_x, B_y, \ldots, B_d\}$ in d dimensions is given by: $\vec{A} \cdot \vec{B} = (A_x * B_x) + (A_y * B_y) + \ldots + (A_d * B_d)$.

[2]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: $\vec{A} \times \vec{B} = (A_2B_3 - A_3B_2)i + (A_3B_1 - A_1B_3)j + (A_1B_2 - A_2B_1)k$.

Algorithm 2 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

for all $\omega \in \overline{\omega_i \omega_j}$ **do**
  if $\nabla U'(\vec{x_\omega}) \times u(\overrightarrow{\omega A}) \cdot \nabla U'(\vec{x_\omega} \times u(\overrightarrow{\omega S})) < 0$ **then**
    $\phi_\omega(\overrightarrow{dir}) \leftarrow \|\nabla U'(\vec{x_\omega})\|$
    $\overrightarrow{dir} \leftarrow \nabla U'(\vec{x_\omega})$
  else
    $\phi_\omega(\overrightarrow{dir}) \leftarrow \max_{\overrightarrow{dir} \in \{u(\overrightarrow{\omega A}), u(\overrightarrow{\omega S})\}} \nabla U'(\vec{x_\omega}) \cdot \overrightarrow{dir}$
    $\overrightarrow{dir} \leftarrow \arg \phi_\omega(\overrightarrow{dir})$
  end if
  if $\phi_\omega > \phi_{\text{opt}}$ **then**
    $\phi_{\text{opt}} \leftarrow \phi_\omega$, $\omega_{\text{opt}} \leftarrow \omega$, $\overrightarrow{dir}_{\text{opt}} \leftarrow u(\overrightarrow{dir})$
  end if
end for

The adaptation process is based on an event-driven model that triggers the re-placement of operators whenever the position of neighboring operators change or if the data rates of incoming or outcoming data streams change. In case of such changes, the operator tree 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 V in a distributed manner [10]. In the distributed case, each operator optimizes its local network usage and exchanges messages with its neighbors to communicate its new position, until the positions of its neighbors do not change anymore. After a number of iterations, the distributed algorithm yields the optimal solution for the operator tree.

Subsequently the operator tree 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. 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 on the operator tree at each point in time with a delay that depends on the time to trasmit the messages along the tree.

Based on this global view, the coordinator node assumes that the unconstrained optimization has reached a stable state, when the positions of operators in the tree have not changed 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 the tree, containing the mapping of the unpinned operators. 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 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 realizing the physical mapping 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 analyze 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 distributedly solve the unconstrained optimization problem during the unconstrained optimization phase. A detailed analysis of this overhead has been presented in our previous paper [10], (2) The state information propagated upwards in the aggregation tree to determine the end of the unconstrained optimization. Actually, the additional overhead introduced for transmitting state information is very 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 $\#unpinnedOperators - 1$ messages. Overall, we see that only Step (3) introduces a very small amount of additional messages compared to the unconstrained optimization.

VII. Evaluation

Next, we present the performance evaluation of our constraint satisfaction algorithm by comparing it to the theoretic optimum and our unconstrained optimization algorithm.

A. Experimental Settings

To evaluate the performance of our algorithm we implemented our algorithm for the network simulator PeerSim.

For the underlying (physical) network topology, we used data gathered from a real network, namely the PlanetLab [2]. The *PlanetLab topology* consists of 226 physical nodes including real measurements of the delays between the nodes. The coordinates of the physical nodes in the latency space were found using a prototype implementation of the Vivaldi algorithm [1] that maps the physical nodes in the latency space with an average error of 15 ms w.r.t. to the measured delays. The PlanetLab topology gives us the chance to asses the practical performance of our algorithm in a real system.

For the structure of the operator trees, we use *operator trees* of 6 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. 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, depending on the concrete experiment.

Since there is no related approach that solves the same constrained optimization problem, we compare our constrained optimization algorithm with the theoretic optimum and our unconstrained optimization algorithm [10]. To find the real optimum, we execute an exhaustive search on all possible placements. Therefore we use small operator trees with 6 nodes, since the complexity of the exhaustive search grows exponential with the number of nodes of the tree.

B. Relation Between Network Usage and Latency

First we analyze the basic relation between the two metrics subject to this paper, 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 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 $LS_{\text{unconstr\_opt,min}} = \frac{L_{\text{unconstr\_opt}}}{L_{\min}}$. Similarly the network usage stretch is defined as $US_{\text{constr\_opt,min}} = \frac{U_{\text{constr\_opt}}}{U_{\min}}$, 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 tree, 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. inverse proportional to h. Thus for large h, the input data rates of an operator are unbalanced, while the output data rate of the operator are low.

Fig. 3 shows the results for varying values of the heterogeneity factor h. We see that as the heterogeneity increases, the stretch factors both in terms 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 tree's network usage.

C. Performance of the Constrained Optimization Algorithm

Next, we continue with the analysis of the performance of the algorithm for the constrained optimization problem proposed in this paper. 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 $[L_{\min}, L_{\text{unconstr\_opt}}]$, 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 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 $[L_{\min}, L_{\text{unconstr\_opt}}]$ also increases, formulating thus a larger spectrum of latency constraints.

For our experiment, we have generated 1000 operator trees with varying heterogeneity factor $h \in [2, 4]$ 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, divided by the number of all

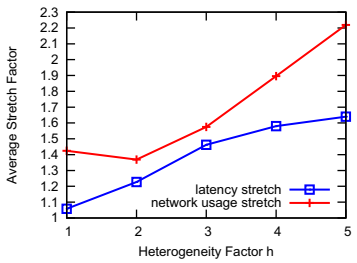


Figure 3. Latency and Network Usage stretch for varying heterogeneity.

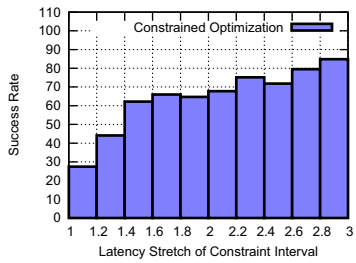


Figure 4. Success rate according to the constraint latency stretch.

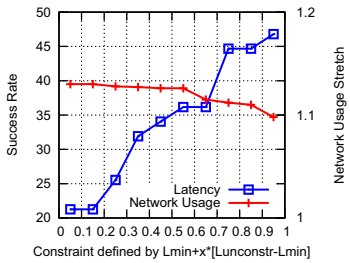


Figure 5. Success rate and network usage stretch for latency stretch $1.0 - 1.2$

experiments. Furthermore, to evaluate the cost for satisfying the constraint, we calculate the network usage stretch in comparison with the network usage of the unconstrained problem that we get after the unconstrained optimization: $US_{\text{constr\_opt,unconstr\_opt}} = \frac{U_{\text{constr\_opt}}}{U_{\text{unconstr\_opt}}}$.

Fig. 4 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 more 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 and thus in such cases all requested latency constraints are very close to the real optimum. However as we see later, also in these cases our algorithm returns a good approximation of the optimum.

Fig. 5 and Fig. 6 show the corresponding graphs for operator trees with latency stretch between 1.0 and 1.2, and 2.0 and 2.2 respectively. In each graph we see on the x axis the latency constraints that are requested, varying gradually in a step of 10% of the total constraint interval $[L_{\text{min}}, L_{\text{unconstr\_opt}}]$ at a time, i.e. $C \in [L_{\text{min}} + x * (L_{\text{unconstr\_opt}} - L_{\text{min}})]$, where $x \in [0, 1]$. On the left y axis we have depicted the success rate and on the right y axis, the network usage stretch factors of the constrained solution. On the one hand, we see that when the latency stretch is low (Fig. 6), the average succes rate is increasing slowly from 20% to 46% while the average cost is kept low and decreases slowly from 1.13 to 1.1. On the other hand, for large latency stretch (Fig. 5), 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 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.

D. Quality of Solutions

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 constraint 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, $LS_{\text{constr\_opt,C}} = \frac{L_{\text{constr\_opt}}}{C}$, i.e. the constrained optimum compared to the requested latency constraint C, which intuitively shows how close the solution is to the requested constraint.

Fig. 7 shows the cummulative distribution of the latency stretch for a set of $4,000$ simulation runs. In detail, 70% of the 44% of simulations that were below C have a latency stretch between 0.9 and 1. Moreover 75% of the remaining 56% instances that are above C 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 C, e.g. 5% that are above 1.4. Such bad approximations can be the result of either the coordinates of the latency space that in (rare) cases do not accurately reflect the real latencies between nodes. Or it can be due to a sparsely populated latency space where too few physical nodes are available close to the continuous positions of the solutions.

Moreover, we calculate the network usage stretch compared to the network usage of the theoretic constrained optimum found by exhaustive search $US_{\text{constr\_opt,theoretic\_constr\_opt}} = \frac{U_{\text{constr\_opt}}}{U_{\text{theoretic\_constr\_opt}}}$. Fig. 8

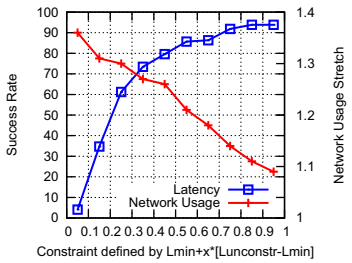


Figure 6. Success rate and network usage stretch for latency stretch $2.0 - 2.2$

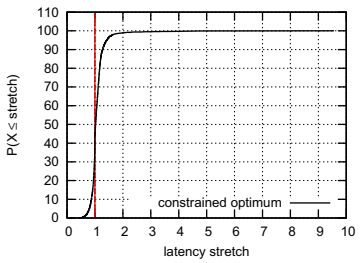


Figure 7. Cummulative distribution of latency stretch.

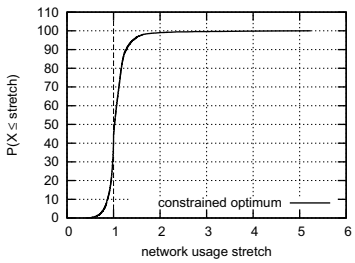


Figure 8. Cummulative distribution of network usage.

shows the corresponding cummulative 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 successfull 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.

As a conclusion of our evaluation, we can say that our algorithm achieves a very good performance when the constraints are not very strict and in other cases it still finds a very good approximation of the solution close to the requested latency constraint, while minimizing the cost in terms of network usage.

VIII. Conclusions and Future Work

In this paper, we have formulated a constrained optimization algorithm for fulfilling latency constraints under the optimization of network load formally expressed by the network usage of the inter-operator data streams of an operator tree. Unlike previous approaches, our constrained optimization strategy finds first an unconstrained optimum in a continuous latency space and then distorts this solution by causing minimal loss of our optimization gains until it achieves the requested latency constraint. Besides an algorithm for the initial placement of operators, we also presented an adaptive algorithm that adapts operator placements under dynamic network conditions, without inducing significant message overhead compared to the unconstrained optimization algorithm. Our evaluation results show that our algorithm performs well both for relaxed constraints where it fulfills the requested latency constraint with high success rate as well as for strict constraints, where it achieves a good approximation of the optimum.

In our future work, we intend to extend our model in order to include the processing delay of operators, thus targeting a larger variety of applications. Moreover we want to address the problems that arise by allowing the re-use of operators among different operator trees.

Acknowledgment

The work presented in this paper was supported by the German Research Foundation (DFG) within the Collaborative Research Center (SFB) 627.

References

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

[2] Planetlab. http://www.planet-lab.org.

[3] K. Aberer, M. Hauswirth, and A. Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM 2007*, pages 198–205, 2007.

[4] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM '04*, 2004.

[5] P. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a Worldwide Sensor Web. *IEEE Pervasive Computing*, 2(4):22–33, 2003.

[6] X. Gu, P. S. Yu, and K. Nahrstedt. Optimal Component Composition for Scalable Stream Processing. In *ICDCS '05*, pages 773–782, 2005.

[7] O. Papaemmanouil, Y. Ahmad, U. Çetintemel, and J. Jannotti. Application-aware Overlay Networks for Data Dissemination. In *ICDE Workshops*, page 76, 2006.

[8] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE '06*, 2006.

[9] T. Repantis, X. Gu, and V. Kalogeraki. Synergy: Sharing Aware Component Composition for Distributed Stream Processing Systems. In *Middleware*, pages 322–341, 2006.

[10] S. Rizou, F. Dürr, and K. Rothermel. Solving the Multi-operator Placement Problem in Large Scale Operator Networks. In *ICCCN'10*, 2010.

[11] S. Rizou, K. Häussermann, F. Dürr, N. Cipriani, and K. Rothermel. A System for Distributed Context Reasoning. In *ICAS'10*, pages 84–89, 2010.

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

[13] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD '06*, 2006.