Subscription Tree Pruning: A Structure-Independent Routing Optimization for General-Purpose Publish/Subscribe Systems

Sven Bittner and Annika Hinze

University of Waikato, New Zealand
{s.bittner, a.hinze}@cs.waikato.ac.nz

Abstract. A main challenge in distributed publish/subscribe systems is the efficient and scalable routing of incoming information (event messages). For large-scale publish/subscribe services, subscription forwarding has been established as a prevalent routing scheme. It reduces the network traffic for event routing due to selectively forwarding event messages to relevant parts of the network only. To further improve event routing, publish/subscribe systems apply routing optimizations. So far, optimizations for general-purpose publish/subscribe systems are still missing.
In this paper, we present the architecture, realization, and evaluation of our prototype of a large-scale publish/subscribe service applying a novel routing optimization, subscription tree pruning. We also show a comparison of five existing routing optimizations in respect to six important characteristic parameters affecting the suitability of these approaches in practice (including space usage, time efficiency (throughput), and network load). This comparative analysis clearly demonstrates the advantages of subscription pruning over other routing optimizations. In our practical experiments, we then investigate the behavior of our prototype regarding all quantitatively measurable parameters from our previously theoretically analyzed ones. Our evaluation of subscription pruning in this paper is more extensive than previous analyses of any routing optimizations for publish/subscribe systems, which focus on selected parameters only.

1 Introduction

The publish/subscribe (pub/sub) communication paradigm has gained increasing attention in both academia and industry within the last years. This interest results from the wide applicability of the pub/sub or, as it is also called, push-based approach. We can utilize systems supporting pub/sub for various practical purposes ranging from low-level monitoring of distributed systems [15] to high-level applications for electronic commerce [10].

Large-scale pub/sub services require implementations as distributed systems [16]. This allows to handle a large amount of clients, as well as subscriptions and event messages. Various realizations of these systems apply a routing scheme known as subscription forwarding [2]; they distribute subscriptions among their routing components to allow for a selective event routing and thus a reduction in network traffic. Additionally, distributed pub/sub systems apply routing optimizations to decrease the sizes of the resulting routing tables.

Current routing optimizations only work on restricted (i.e., purely conjunctive) subscription languages. Within each filtering component of the system, arbitrary subscriptions can be converted in disjunctive normal forms (DNFs). Then, the disjunctively combined elements of these DNFs, which are conjunctions per definition, are filtered separately. However, this approach strongly increases the memory requirements and thus decreases the scalability of a pub/sub system, as shown in [3]. For the distributed filtering and routing, this conversion to DNFs results in a tremendously increased size of routing tables since the effects observable in each single component of the network are multiplied within the whole network. Thus, handling arbitrary subscriptions in current pub/sub systems leads to low scalability due the lack in expressiveness in their subscription languages.

Next to this disadvantage regarding expressiveness, we have identified drawbacks in the assumptions made by recent optimization proposals: There have to exist strong similarities or relationships among subscriptions. However, such presumptions do not always hold in practice. We can especially observe their consequences in current experimental evaluations of routing optimizations: The chosen settings only contain very simple and restricted subscriptions on relatively tiny event spaces. Under all circumstances, for various high-level application scenarios, e.g., online auctions, today's simplistic assumptions do not occur in practice, as argued in [5]. Thus, current evaluations are too basic and simplistic to allow for their generalization to more advanced settings and applications.

In this paper, we provide the full details and the first evaluation of a prototype of a distributed pub/sub system utilizing a novel routing optimization, subscription tree pruning. Subscription pruning works on arbitrary Boolean subscriptions (i.e., it supports an expressive subscription language) and optimizes the routing by considering subscriptions independently of each other. Hence, we can apply subscription pruning regardless of the relationships and similarities among subscriptions. This allows for an optimization potential for all Boolean subscriptions regardless of their individual and collective structures. Thus, subscription pruning is the favorable routing optimization for general-purpose pub/sub systems.

We have structured the rest of this paper as follows: In Sect. 2, we introduce the general concepts and the background regarding filtering and routing in distributed pub/sub systems. Then, we analyze and compare five routing optimizations in respect to six important characteristic parameters affecting (among others) efficiency and scalability of event routing in Sect. 3. To our knowledge, this work represents the first comparative evaluation of routing optimizations focussing on more than only a small selection of parameters. The clear result of Sect. 3 is the recognition of subscription pruning as the most promising of the existing routing optimizations for general-purpose pub/sub systems.

In Sect. 4, we provide the complete details of a realization of a subscription pruning-based pub/sub system. We present our model for subscription pruning, analyze the effects of subscription pruning on the routing load, introduce two possible pruning realizations, give an overview of our prototype, and argue the combination potential of subscription pruning with other optimization proposals. Section 5 presents an extensive practical evaluation of the influence of subscription pruning on event routing using our prototype. We show the effects of subscription pruning on the three quantitatively measurable of the six previously identified characteristic parameters (Sect. 2), i.e., network load, routing efficiency (system throughput), and memory usage. Our experiments are the first of their kind focusing on the correlation of all of these parameters in an evaluation of routing optimizations. Finally, we conclude this paper in Sect. 6 and present our plans for future work.

2 Background Regarding Distributed Publish/Subscribe Systems

In this section, we give some background information about pub/sub systems in general, and the terminology and concepts used throughout this paper. We provide this information to keep the paper self-contained and to avoid confusion due to differently used terms in existing works. We first present the background of pub/sub (Sect. 2.1) followed by established concepts for distributed pub/sub services (Sect. 2.2). Finally, we focus on the routing in these systems (Sect. 2.3).

2.1 Publish/Subscribe Systems

Information delivery in pub/sub systems is typically performed on the content of messages, which, in this context, are referred to as event messages e. These messages are sent to the system by publishers P aiming at distributing their information to all interested clients, i.e., subscribers S. Interests of subscribers are specified by the help of subscriptions s that are registered with a pub/sub system.

A subscription is usually a Boolean filter expression; variables of this expression are called predicates p. Predicates specify filter operations on event messages, and they are represented by attribute-operator-value triples. That is, predicates specify conditions on the values of attributes. The Boolean filter expression of a subscription can be expressed by a tree structure [3].

Each event message consists of a set of attribute-value pairs presenting the content of the message. Optionally, messages may contain any other kind of information, e.g., a serialized object. In this case, we can understand the set of attribute-value pairs of an event as its description.

The evaluation of the Boolean filter expression of a subscription s on an event message e either leads to the result *true* or *false*. If s evaluates to *true* on e, we say that the subscription is fulfilled by the event message. We can additionally refer to this situation as event e matches subscription s. In case of matching, the pub/sub system sends a notification to the subscriber of subscription s stating the event fulfilling s. The set of event messages fulfilling subscription s is referred to as $E(s)$.

2.2 Distributed Publish/Subscribe Systems and Routing

A distributed pub/sub system consists of a network of so-called broker components B providing an interface to clients. Publishers and subscribers connect to one of these brokers (then referred to as local clients and broker) to publish and register their event messages and subscriptions, respectively.

In the simplest case, the network topology connecting brokers forms an acyclic graph. We can also support cyclic topologies by introducing mechanisms to avoid circulating messages or by forming minimum spanning trees on the given network [8]. There also exist extensions allowing for a dynamic network reconfiguration [18].

Internally, brokers store information about their local clients and their neighbor brokers in the network. For this purpose, brokers inform their neighbors about newly registered as well as deregistered subscriptions. This information is used to build up routing tables for the successful routing of incoming event messages to all interested parties. Entries in these routing tables state which event messages should be forwarded to local subscribers and neighbor brokers. Routing entries are realized via subscriptions being registered by subscribers or forwarded by brokers. This scheme is called subscription forwarding.

Generally, subscriptions change rarely compared to the frequency of incoming event messages. Exactly this property is exploited by subscription forwarding, which focuses on efficient event routing. The downside is a more complex registration process for subscriptions. Due to infrequent changes of subscriptions, subscription forwarding is a feasible solution.

2.3 Evaluation and Creation of Routing Tables

The evaluation of routing tables is obtained by filtering algorithms efficiently computing fulfilled subscriptions for incoming event messages e. If these subscriptions have been found, e is routed to the local subscribers of a broker and its neighbor brokers. Neighbor broker components then evaluate their routing tables to notify local subscribers as well as their neighbor brokers. Brokers do not return an event message e to that neighbor that sent e.

An example of the subscription forwarding scheme and the created routing tables is given in Fig. 1. There our network consists of four brokers B_1 to B_4. Furthermore, we present three subscribers S_1 to S_3 and a publisher P_1. Subscriptions are registered at their local brokers (e.g., subscription s_1 from subscriber S_1 at broker B_1). Brokers forward local subscriptions to their neighbor brokers (e.g., s_1 is forwarded from B_1 to B_2). Neighbors, in turn, distribute these subscriptions even further (e.g., s_1 is distributed from B_2 to B_3 and B_4). An extension of this scheme is the utilization of advertisements [6] stating potential event messages of publishers. Advertisements allow for a reduction in the number of brokers a subscription is forwarded to.

In Fig. 1, we also exemplarily present the routing of event e_{24}. This event fulfils subscriptions s_2 and s_4. Initially, e_{24} is published at the local broker B_4 of its publisher. Broker B_4 evaluates its routing table and forwards e_{24} to its neighbor B_2. B_2 also uses its routing table, and notifies local subscriber S_2 and additionally forwards e_{24} to B_3, which had previously forwarded subscription s_4. Finally, broker B_3 notifies its local subscriber S_3.

To allow for a high throughput of event messages, the evaluation of routing tables in brokers should be performed efficiently. Thus, the filtering algorithm utilized for this evaluation is a critical part of a pub/sub system. For Boolean subscriptions, the algorithm in [3] has been shown to support time and space efficient filtering [3]. For restricted conjunctive subscriptions, we should prefer the approaches in [1, 12].

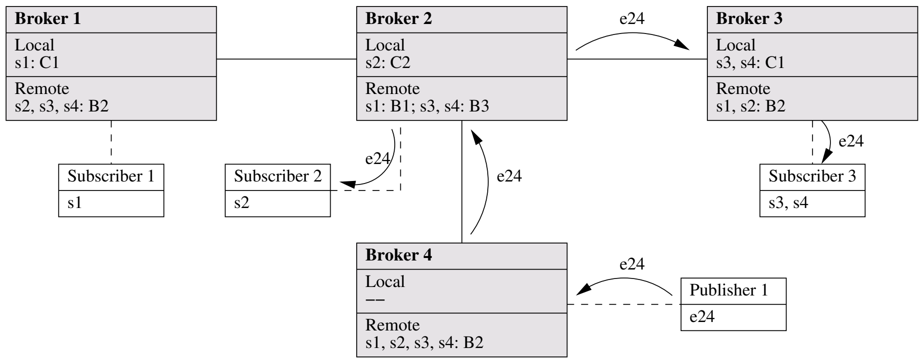


Fig. 1. Example of routing tables in a distributed pub/sub system

Next to this requirement for time efficient filtering algorithms, we preferably want to minimize the sizes of our routing tables. This reduction in number and complexity of routing entries is obtained by applying routing optimizations. We present and analyze existing optimizations in the next section; their comparison is finally shown in Sect. 3.5.

3 Analyzing Routing Optimizations in Distributed Publish/Subscribe Systems

In the literature, we can find five approaches targeting at the optimization of routing tables in pub/sub systems. Subscription covering [17] and subscription merging (with the two variants perfect and imperfect merging) [16] try to decrease the number of entries in routing tables. Hence, they aim at reducing the size of the filtering problem. This might in turn lead to a more efficient event routing and thus a higher system throughput.

Predicate replacement [5] and subscription tree pruning [5] target at reducing the complexity of routing table entries and thus the size of routing tables. Hence, these optimizations aim at decreasing the complexity of the filtering problem. This reduces the memory requirements for routing tables and increases routing efficiency.

The application of routing optimizations affects some characteristic parameters of routing algorithms. In combination, these parameters influence (among others) efficiency and scalability of event routing. We have identified six characteristics that need to be taken into account when evaluating the effects of routing optimizations:

1. Applicability of the optimization approach
2. Memory requirements for routing tables
3. Network load created by event routing
4. Computational complexity to route events, i.e., to evaluate routing tables
5. Expressiveness of supported subscription classes, i.e., routing table entries
6. Overhead for handling deregistrations

Parameter 1 refers to the applicability of an optimization approach. This includes both general requirements to be able to apply an optimization (e.g., size of event spaces) and assumptions on subscriptions and events (e.g., relationships among them as well as their distribution and structure).

The potential of an optimization to reduce the sizes of routing tables is described by Parameter 2. This characteristic parameter directly influences the scalability of broker components in respect to growing subscription numbers. Parameter 3 designates the influence of an optimization on the network load for event routing, i.e., does an optimization strongly increase the network load? This parameter influences the scalability of the overall system.

The relief in computational effort that is required to evaluate routing tables is referred to by Parameter 4. That is, it affects the efficiency of event routing and thus the system throughput. Generally, a routing optimization should increase the system throughput. Parameter 5 describes which classes of subscriptions one can utilize if applying an optimization, i.e., which subscriptions languages are supported. Parameter 6 determines the overhead that is required if deregistrations occur. These situations appear rarely; although, optimizations should not increase the already demanded overhead for deregistrations when using subscription forwarding.

In the following subsections, we introduce existing routing optimizations and evaluate them according to the six previously introduced parameters. This allows for the discovery of advantages and disadvantages of optimizations. Their comparison is then presented in Sect. 3.5.

3.1 Subscription Covering

Subscription covering aims at removing redundant entries in routing tables. The definition of coverings is based on the set of event messages fulfilling subscriptions. We state that subscription s_x covers s_y if and only if $E(s_x) \supseteq E(s_y)$ [17]. This definition realizes the intuitive idea that subscriptions s_y leading to a subset of notifications of subscriptions s_x do not need to be considered in filtering, i.e., they might be excluded from routing decisions without affecting the correctness of the system.

Covering is exploited when forwarding subscriptions: We do not need to forward subscriptions to a neighbor broker if we have already forwarded a covering subscription. Additionally, we can remove all covered subscriptions that have been forwarded by the same neighbor as a newly registered subscription. Subscriptions registered by local brokers are never removed from routing tables to allow for the determination of all subscriptions that are fulfilled by an incoming event message. We now briefly investigate subscription covering in respect to the six parameters affecting routing algorithms:

Parameter 1 - Applicability. Covering strongly depends on registered subscriptions. If only a few subscriptions cover each other, the possible amount of optimization is very small. Additionally, the definition of covering questions the applicability of this optimization approach: Even if only one potential event message might fulfil s_x and not s_y and vice versa, there is neither a covering nor a covered by relationship between s_x and s_y. Thus, we cannot apply this routing optimization in most cases.

Parameter 2 - Memory requirements. If coverings exist among subscriptions, subscription covering reduces the number of routing table entries. If there are no coverings, this optimization does not lead to any reduction in memory requirements.

There are no evaluations analyzing covering properties in practical settings. However, mostly we cannot find assumptions as in [16], e.g., the probability of coverings among any two registered subscriptions is 50%. Thus, the reduction in memory usage in practice heavily depends on application semantics and is still questionable.

Parameter 3 - Network load. The network load created by event routing is not affected by the application of coverings. This is due to the fact that, on the one hand, coverings do not introduce false positives. On the other hand, when utilizing coverings, we do not influence the correctness of the filtering and routing algorithm [16]. The network load remains the same since event messages fulfilling several subscriptions are sent to neighbor brokers only once.

Parameter 4 - Computational load. If coverings exist among subscriptions, the computational load in broker components is decreased. Thus, the system throughput increases. This is because we do not need to perform evaluations of covered subscriptions (less routing table entries). Again, the amount of optimization depends on the relationships among subscriptions.

Parameter 5 - Expressiveness of subscriptions. Coverings have been applied and investigated in the pub/sub systems PADRES [14], REBECA [17], SIENA [7], and XROUTE [9]. All of these systems only support conjunctive subscriptions. REBECA [17] additionally restricts subscriptions to contain at most one predicate per attribute. Algorithms to compute coverings for arbitrary Boolean subscriptions do not exist in literature.

Parameter 6 - Overhead for deregistrations. Deregistering subscriptions requires a specialized handling when applying covering. This is because covered subscriptions are not stored in non-local brokers. Hence, if a covering subscription s_x is deregistered, we need to forward all covered subscriptions s_y to neighbor brokers to register these subscriptions. Potentially, s_x might cover a large number of subscriptions leading to numerous registrations in various brokers and thus a high network and computational load.

3.2 Subscription Merging

Subscription merging tries to reduce the number of registered subscriptions and thus the number of routing table entries. There exist two variants of merging: perfect and imperfect merging. It is stated that a subscription s_x is a merger of a set of subscriptions $\mathcal{S}$ if and only if $E(s_x) \supseteq \bigcup_{s_y \in \mathcal{S}} E(s_y)$ [16]. In case of set equality, s_x is denoted as perfect merger. For a proper subset relation, one refers to s_x as imperfect merger. Finding a merging satisfying given conditions has been proven as an NP-hard problem [11].

In practice, we can apply merging to summarize any subscriptions that have been forwarded by the same neighbor broker. This reduces the number of registered subscriptions and thus the number of routing table entries. Imperfect merging introduces false positives. The general problems one has to face when applying merging are when, what, and how to merge [14] subscriptions.

Independently of these problems, we now characterize perfect and imperfect merging in respect to the six parameters of event routing algorithms identified above.

Parameter 1 - Applicability. One can find a perfect merger for all possible subscriptions assuming the existence of an expressive subscription language. However, most pub/sub systems, e.g., PADRES [14], REBECA [17], SIENA [7], and XROUTE [9], only support conjunctions. Mostly, such a restrictive subscription language cannot represent a perfect merger. Imperfect merging is applicable under all circumstances. The degree of inaccuracy (amount of false positives) depends on the subscription language used and the subscriptions to be merged. Generally, less expressive languages imply more inaccuracies.

Parameter 2 - Memory requirements. The space complexity of a perfect merger cannot be reduced (assuming one cannot minimize the merger any further). However, if we might find several perfect mergers, the one requiring the least memory in routing tables should be utilized for routing. In particular, if the subscription set $\mathcal{S}$ to be merged does not show strong similarities, the representation of a perfect merger can easily require the same space as the sum of the memory requirements of all single subscriptions $s_y \in \mathcal{S}$.

Using imperfect merging, one can always find a merger. There exists a tradeoff between space complexity and amount of false positives. Generally, we can decrease the memory requirements of an imperfect merger by increasing its inaccuracy and thus the number of false positives, which in turn increases the network load.

Parameter 3 - Network load. Perfect merging does not increase the network traffic for event routing compared to the situation without utilizing merging [19].

When using imperfect merging, the network load created by event messages increases due to false positives. The amount of false positives depends on the accuracy of the imperfect merger. Per definition, each imperfect merger increases the network load to a certain amount.

Parameter 4 - Computational load. A perfect merger might not optimize the overall system throughput [19]. It might not even increase routing efficiency. This happens in cases of merging subscriptions involving little similarities.

For imperfect merging, we experience a tradeoff between the computational load in individual brokers and the amount of false positives. For example, a merger that is less complex to filter on could create more false positives. Using this merger results in less filter complexity in the broker performing the merging. However, other brokers have to filter more event messages (the false positives are forwarded), which in turn degrades the overall throughput of the system.

Parameter 5 - Expressiveness of subscriptions. The works in [11, 14, 17, 19] support subscription merging. They are all restricted to purely conjunctive subscription languages. The underlying reason is the complexity of the general merging problem, which is NP-hard [11].

Parameter 6 - Overhead for deregistrations. When deregistering a subscription that has been merged into a merger, this merger becomes (more) inaccurate. At some point of inaccuracy, the merger needs to be replaced by its constituting subscriptions. Thus, all its constituents need to be forwarded to other brokers. If considering a whole network and various merging operations, this easily leads to cascading deregistrations because previously merged subscriptions have been forwarded and merged again. The result is a large amount of network traffic in case of deregistering only one subscription.

3.3 Predicate Replacement

We can look at predicate replacement [5] as a variant of subscription covering. This optimization is also based on relationships among sets of fulfilling event messages. The difference is that we only analyze predicates instead of complete subscriptions. We state that predicate p_x covers p_y if and only if $E(p_x) \supseteq E(p_y)$.

The optimization idea of predicate replacement is to substitute predicates in subscriptions from neighbor brokers: All occurrences of a predicate p_x in subscriptions are replaced by another predicate p_y that is covering p_x. Applying this scheme, on the one hand, increases inaccuracies of subscriptions. On the other hand, the routing table does not contain p_x anymore, which leads to a more time efficient event filtering and less memory requirements for routing tables. Subscriptions registered by local subscribers are not modified to ensure correct event routing.

According to our six parameters, we assess predicate replacement as follows:

Parameter 1 - Applicability. Similar to subscription covering, predicate replacement depends on the subscriptions registered with the pub/sub system. The more predicates cover each other, the more replacement operations we can perform. This dependency on registered subscriptions is not that problematic as in the subscription covering approach. There, we are looking at covering relationships among whole subscriptions, which are rare compared to covering relationships among predicates. Thus, the optimization potential and applicability of predicate replacement is much higher than the potential of subscription covering.

Parameter 2 - Memory requirements. Whenever it is possible to replace a predicate p_x by a covering one, we do not need to filter on p_x any longer. Thus, we can remove p_x from internal predicate index structures. In turn, the memory required for these structures decreases. The amount of decrease in memory is rather small since we can only eliminate predicates from filtering structures. The size of subscriptions themselves, however, remains unchanged.

Parameter 3 - Network load. Replacing predicates by covering ones means to introduce false positives. Thus, the network load for event routing in a pub/sub system is affected by predicate replacement. The more inaccuracies we introduce by replacements, the more the network load increases. The amount of inaccuracies does not depend on the number of replacement operations. It is rather influenced by the predicates used in subscriptions and the predicates chosen for replacement.

Parameter 4 - Computational load. After replacing any predicate, the computational load in individual broker components is decreased. The influence of replacements on the overall system throughput depends on the inaccuracies introduced by replacements (similar to imperfect merging).

Parameter 5 - Expressiveness of subscriptions. In [5], the approach of predicate replacement has been introduced. We can utilize this optimization for all kinds of Boolean subscriptions. This includes conjunctive subscriptions.

Parameter 6 - Overhead for deregistrations. Deregistrations do not require a specialized handling when using the predicate replacement approach.

3.4 Subscription Tree Pruning

Subscription tree pruning [5] aims at the generalization of subscriptions in order to reduce memory requirements and computational complexity. In turn, it introduces false positives and thus increases the network load for event routing.

Practically, we can prune subscriptions that have been registered by neighbor brokers. Similar to predicate replacement, we do not prune subscriptions registered by local subscribers to ensure correct event routing. Pruning eliminates some of the predicates of a subscription. Not all possibilities of removing predicates realize a correct pruning. When performing a valid pruning of s_x leading to s_y, it has to hold $E(s_x) \subseteq E(s_y)$.

In respect to our six parameters, we rate subscription tree pruning as follows:

Parameter 1 - Applicability. We can utilize subscription tree pruning under all circumstances regardless of actually registered subscriptions. In contrast to all other optimization approaches, subscription pruning does not depend on relationships or similarities among subscriptions. It rather optimizes subscriptions independently of each other. Various application scenarios tend to allow for a strong optimization potential when utilizing subscription pruning [5].

Parameter 2 - Memory requirements. With each performed pruning operation, the memory requirements for our routing tables decrease. Firstly, pruning reduces the memory needed to store subscriptions. Secondly, in case of removing all occurrences of certain predicates, predicate index structures are reduced in size.

Parameter 3 - Network load. Pruning subscription trees leads to the introduction of false positives and thus to an increased network load. The amount of additional network load depends on the application domain. In [5], it is reasoned that subscription pruning is suitable for various practical applications, i.e., it does not strongly increase the network load.

Parameter 4 - Computational load. Subscription pruning does not only relieve memory resources. Additionally, the complexity of subscriptions is decreased. This allows for a faster evaluation of subscriptions and thus a more efficient event routing in individual broker components. False positives lead to an increased number of filtering operations in the overall system. These two factors affect the overall system throughput.

Parameter 5 - Expressiveness of subscriptions. We can utilize subscription tree pruning for all classes of subscriptions. In [5], this optimization is introduced for arbitrary Boolean subscriptions. This also includes conjunctive subscriptions, which are currently used in various pub/sub systems [7, 9, 14, 17] out of efficiency aspects.

Parameter 6 - Overhead for deregistrations. Subscription tree pruning does not summarize any subscriptions, nor does it prevent subscriptions from being forwarded to other broker components. Thus, deregistrations can be handled as uncomplicated as in un-optimized routing approaches.

3.5 Comparison of Routing Optimizations

In the previous subsections, we have described, analyzed, and evaluated five current routing optimizations in respect to our six characteristic parameters. We now summarize these findings and compare the different optimization approaches.

Table 1 gives an overview of our comparison. In the rows of the table, we present our six characteristic parameters affecting the event routing process. In Column 2 to Column 6, we illustrate our ratings of the optimization approaches. We have split subscription merging into its two variants, perfect (Column 3) and imperfect merging (Column 4).

Table 1. Overview of characteristic parameters of routing optimizations ranging from excellent (++) over suitable (+−) to poor (−−)

Characteristic parameter	Covering	Perfect m.	Imperfect m.	Predicate r.	Subscript. p.
1. Applicability	−−	++	++	+−	++
2. Memory requirements	++	−−	++	+−	++
3. Network load	++	++	+−	+−	+−
4. Computational load	++	++	+−	+−	+−
5. Expressiveness	−−	−−	−−	++	++
6. Overhead deregistrations	−−	−−	−−	++	++

In Table 1, we give our ratings for the different parameters independently of each other, i.e., for the rating of each parameter we assume the best case scenario for all other parameters. This allows for the incorporation of certain drawbacks of optimizations into a small subset of parameters only (or even one parameter only).

The applicability (Parameter 1) of covering is rated poor because coverings rarely occur in practice. We evaluate predicate replacement as suitable since coverings among predicates are relatively likely. The other three approaches are applicable for all subscriptions regardless of their collective and individual structure.

Regarding memory requirements (Parameter 2), perfect merging is evaluated poor because a perfect merger is likely to require the same amount of memory as its constituents. Predicate replacement only reduces the size of predicate index structures. Imperfect merging and covering decrease the number of registered subscriptions (if applicable); subscription pruning reduces the complexity of subscriptions under all circumstances.

Covering and perfect merging do not introduce false positives. Thus, their network load for event routing (Parameter 3) is rated excellent. The remaining three approaches (all of them lead to false positives) show a suitable increase in network usage.

The computational load (Parameter 4) of covering and perfect merging has the best rating since no additional events need to be filtered and the number of subscriptions to filter on is reduced (if applicable). Imperfect merging, predicate replacement, and subscription pruning might introduce false positives leading to more load in worst case. However, the number (imperfect merging) or the complexity (predicate replacement and subscription pruning) of registered subscriptions decreases, which counteracts the influence of more events to filter on. We will discuss and practically show this effect and its influence on the system throughput in Sect. 4.2 and Sect. 5.4, respectively.

We can apply predicate replacement and subscription tree pruning for all classes of Boolean subscriptions. Thus, they receive the best rating regarding expressiveness (Parameter 5). Covering and merging can only be utilized for conjunctive subscriptions. Hence, our evaluation results in poor.

Only predicate replacement and subscription pruning do not affect the process of deregistrations (Parameter 6) and receive an excellent rating. The other three optimization approaches require the forwarding of various subscriptions if only one subscription is deregistered; this is evaluated as poor.

Choosing the Favorable Routing Optimization. Our previous comparison clearly shows that subscription tree pruning is the most promising routing optimization in pub/sub systems for general applications: It is applicable regardless of registered subscriptions and application domain, strongly reduces the memory requirements for routing tables, only increases the network load to a suitable extend, not negatively affects the computational load and thus the throughput, supports expressive subscription languages, and does not introduce any additional overhead in case of deregistrations. Predicate replacement clearly rates worse for two of our parameters. The two merging approaches as well as covering do not support the required subscription languages and introduce an enormous overhead for deregistrations, which makes them inapplicable in practice. Additionally, perfect merging does not relieve memory resources and covering strongly depends on currently registered subscriptions.

Thus, subscription tree pruning is the favorable routing solution for general-purpose pub/sub systems. In the remainder of this paper, we present our prototype realizing this subscription tree pruning approach. We additionally show a detailed analysis of its properties and an evaluation of practical experiments (Sect. 5) confirming our theoretical results presented in this section.

4 Practical Routing Using Subscription Pruning

We start this section by answering the question of how to decide which subscriptions should be pruned (Sect. 4.1). When then theoretically investigate the effects of subscription pruning on the routing load in Sect. 4.2. Section 4.3 presents two variants of pruning. One of them, pre-pruning, aims at optimizing the routing load in the network as a whole. The other variant focuses on optimizing the load in individual broker components (post-pruning).

We have practically realized the subscription pruning approach in a prototype of a distributed pub/sub system. In Sect. 4.4, we give an overview of this prototype and present our practical realization of filtering and pruning structures as well as the system architecture. Finally, in Sect. 4.5 we argue how to utilize subscription pruning in combination with other routing optimizations.

4.1 Estimating Selectivity and Selectivity Degradation

Before performing a pruning operation, we have to determine (i) which subscription, and (ii) which part of its subscription tree should be pruned. This decision is based on the degradation of selectivities of subscriptions. We define selectivity degradation as the difference in selectivity between subscriptions before and after pruning. Generally, we should prune that subscription that supports a pruning leading to the least selectivity degradation. Then, this pruning operation should be executed.

We can determine the selectivity of predicates p by counting the number of event messages fulfilling p. Similarly, by using this method we are able to compute the selectivity of registered subscriptions. However, this approach requires historic information, which is not available for newly registered subscriptions as well as subscriptions created by prunings. But these selectivities after performing a pruning operation are required to determine selectivity degradation.

Regarding new registrations, we are able to determine the selectivity of unused predicates based on the selectivities of similar ones, e.g., as presented in [13]. Thus, it remains to calculate the selectivities of subscriptions without comprising historic information. Since the determination of selectivities of general Boolean expressions requires a huge computational effort, we need a simple method to estimate these selectivities.

Key factor in pub/sub systems are efficient and scalable event filtering [12]. In broker components, scalability is directly influenced by the memory requirements for filtering structures [3, 4]. So, a selectivity estimation approach needs to demand both little memory and computational resources.

Selectivity Estimation. A selectivity estimation allowing for these two properties uses three values [5]: the minimal possible, the average, and the maximal possible selectivity. The minimal possible selectivity describes the worst case, i.e., the smallest value of selectivity holding for all distributions of event messages; the average case assumes a uniform distribution of all possible event messages and independent predicates in subscriptions; the best case is described by the maximal possible selectivity, i.e., the largest selectivity value for any distribution of event messages.

For predicates, all three estimations are the same. We can calculate them by the counting approach mentioned before. For Boolean subscriptions, we can successively compute the selectivity estimate based on the utilized operators[1]:

For a conjunctive node, we present the pseudo code in Algorithm 1. This algorithm walks through all possible children of the input node of a subscription tree and estimates their selectivity (Line 4). The minimal value describes the case that the sets of event messages fulfilling different children do overlap minimally (Line 6). In the end, it needs to be adjusted to be not less than zero (Line 9). The average value assumes that the selectivity of event messages fulfilling one child equally holds for event messages fulfilling the other children (Line 7). Finally, for the maximal value, the fulfilling event messages of all children are included in those event messages fulfilling the least selective child (Line 8).

[1] The filtering approach in [3] shifts negations down into leaf nodes, i.e., predicates. We can derive the selectivity of other Boolean operators from the conjunctive and disjunctive case.

Algorithm 1: Estimating the selectivity for a conjunctive node
Input: A conjunctive node C and the total number of filtered events n
Output: The three components of the estimated selectivity (min, avg, max)

```
ESTIMATESELECTIVITY(C, n)
(1)   min ← 1.0
(2)   avg ← 1.0
(3)   max ← 1.0
(4)   foreach c in C.children
(5)       e ← ESTIMATESELECTIVITY(c,n)
(6)       min ← min + e.min - 1.0
(7)       avg ← avg * e.avg
(8)       max ← MIN(max, e.max)
(9)   if min < 0.0
(10)      min ← 0.0
(11)  return (min, avg, max)
```

Algorithm 2 illustrates the calculation for disjunctive nodes. Once more, we walk through all children of the input node (Line 4). The minimal value assumes that all sets of fulfilling event messages of children are included in the largest one (Line 6). Again, the average case assumes uniformly distributed event messages, i.e., event messages fulfilling any child lead to the same selectivity (Line 7). For the maximal value, the sets of fulfilling events of all children are assumed to be maximally disjoint (Line 8). Finally, this value is corrected not to increase over 1.0 (Line 9).

With this method, we are able to estimate the selectivities of Boolean subscriptions purely based on the known selectivity values of predicates.

Algorithm 2: Estimating the selectivity for a disjunctive node
Input: A disjunctive node D and the total number of filtered events n
Output: The three components of the estimated selectivity (min, avg, max)

```
ESTIMATESELECTIVITY(D, n)
(1)   min ← 0.0
(2)   avg ← 0.0
(3)   max ← 0.0
(4)   foreach c in D.children
(5)       e ← ESTIMATESELECTIVITY(c,n)
(6)       min ← MAX(min, e.min)
(7)       avg ← avg + e.avg - (avg * e.avg)
(8)       max ← max + e.max
(9)   if max > 1.0
(10)      max ← 1.0
(11)  return (min, avg, max)
```

Selectivity Degradation. To determine the best among all possible prunings, we need to quantify the effects of these pruning operations on selectivity. We denote this effect as selectivity degradation; it describes the difference in selectivity of a subscription before and after pruning.

To determine selectivity degradation, we calculate the difference in selectivity in all three components of our selectivity estimation. Then, the degradation equals the maximal difference among those components. This degradation measure is an absolute one comprising the expected additional number of fulfilling event messages for a pruned subscription. That is, a smaller degradation value describes a smaller decrease in selectivity.

4.2 Effects of Subscription Pruning

Subscription pruning targets at increasing the overall throughput in distributed pub/sub systems. This is obtained by reducing the complexity of the filtering and thus the routing problem.

There are several effects we can experience if applying pruning: Firstly, the routing load per event in each individual broker component decreases. This is due to the fact that subscriptions become less complex after performing pruning. Secondly, subscriptions become more general, i.e., inaccurate, due to pruning. This increases the network load among broker components. A consequence is the routing of more subscriptions to neighbors in the network. This, thirdly, increases the number of event messages brokers have to filter on compared to the situation without performing pruning.

Regarding throughput (routing efficiency), the third effect, the forwarding of more event messages, counteracts the advantage of filtering on less complex subscriptions (first effect). If more general subscriptions lead to an introduction of various false positives, this negative effect of subscription pruning might outweigh the advantage of filtering on less complex subscriptions. However, if the amount of false positives keeps reasonable, the positive effect of a less complex filtering process per event outbalances the drawback of filtering on more subscriptions.

We expect the filtering on less complex subscriptions to be the predominating factor regarding the overall efficiency and thus the throughput in pub/sub systems. To optimize this effect, we should prune subscriptions in order of the degradations introduced by these operations. This allows for a decrease in complexity (increasing overall efficiency) without heavily affecting accuracy (negatively affecting efficiency).

In practice, we expect an increasing routing efficiency up to a certain amount of pruning operations. However, if performing a large extent of prunings, we introduce too many false positives leading to an overall decrease in system throughput. The number of prunings we can perform to lead to improved efficiency depends on the structure of subscriptions and the application domain.

Next to affecting efficiency, subscription pruning always leads to a relief in memory resources. Thus, we can always apply subscription pruning to minimize the sizes of routing tables. Pruning up to a certain degree additionally improves routing efficiency. We show these effects in our practical experiments in Sect. 5.

4.3 Variants of Subscription Pruning

There are two options to apply subscription tree pruning as routing optimization: When using pre-pruning, each broker aims at optimizing the routing load in the network as a whole. Post-pruning, on the other hand, focuses on optimizing the routing load for individual broker components.

Pre-pruning. When applying pre-pruning, broker components prune subscriptions before forwarding them to neighbors. That is, each individual broker tries to optimize the routing load in the network as a whole. Consequently, only local brokers integrate unpruned subscription trees into their routing tables (with the exception that brokers decide not to prune subscriptions before forwarding).

The decision about prunings should be based on heuristics and statistic information from neighbor brokers, e.g., their memory usage or the available bandwidth of the respective network connections. This also allows a broker to treat neighbors differently. Especially in heterogenous networks involving variably equipped machines, this option is preferable in respect to optimizing the overall throughput of the distributed pub/sub system.

We can see a drawback of pre-pruning in the fact that decisions to prune subscription trees used in a broker B are obtained in other brokers than B. This could lead to increasing inaccuracies in selectivity estimates, especially if the distribution of event messages sent by publishers is not evenly distributed among the network of brokers.

The instant effect on the network usage one can experience when applying pre-pruning is a decrease in the network load created by forwarding subscriptions.

Post-pruning. Post-pruning is self-dependently performed in broker components to decrease their routing load. Subscriptions are always forwarded in their original form to neighbors in the network. Brokers might also collect statistical or status information from neighbors; however, the effects of pruning operations influence all neighbors in the same way. That is, brokers can mostly only optimize their routing load except in the case that all neighbors require the same actions.

Due to the fact that brokers always prune based on the selectivities estimated on their own, these estimates are more accurate than those obtained in pre-pruning. This makes post-pruning advantageous over pre-pruning in respect to this property.

The decisions to perform post-pruning may also be based on the agreement of all or various broker components. This can avoid problems of only one broker B performing a large amount of pruning operations. This results in an increased network load created by this broker B and, due to no optimizations in neighbors, a higher load in the neighbors of B.

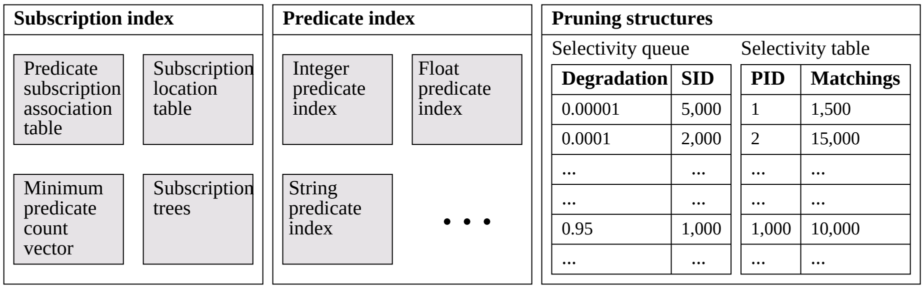
Next to performing each of the two pruning options individually, we can combine both of them. That is, forwarding brokers perform pre-pruning according to heuristics or status information from neighbors. Afterwards, brokers can further apply post-pruning to optimize their load or relieve memory resources.

4.4 System Overview

We have implemented a prototype of a distributed pub/sub system supporting subscription pruning. Our system uses the filtering algorithm presented in [3, 4] to determine all subscriptions matching an incoming event message. In the following, we describe the filtering and pruning structures as well as our system architecture.

Filtering and Pruning Structures. Event filtering works in two steps involving two kinds of indexes: predicate and subscription indexes. Predicate indexes are applied in the first filtering step and return all predicates fulfilled by an incoming event message. In our prototype, we use one-dimensional index structures as predicate indexes, following the proposal in [3]. These indexes are specialized for data types and operators. Currently, we only support integers with equality, greater than, and less than operators. An extension is uncomplicated and straightforwardly to integrate.

The second filtering step returns all matching subscriptions and uses subscription index structures: A predicate subscription association table allows for the determination of subscriptions using certain predicates. The minimally required number of fulfilled predicates per subscription is stored in the minimum predicate count vector. A subscription location table maps subscription identifiers to memory addresses. These addresses contain subscription trees [3], a space efficient encoding of the filter expressions of subscriptions. An overview of all of these indexing structures is illustrated in the left part of Fig. 2.



Degradation	**SID**
0.00001	5,000
0.0001	2,000
...	...
...	...
0.95	1,000
...	...

PID	**Matchings**
1	1,500
2	15,000
...	...
...	...
1,000	10,000
...	...

Fig. 2. Overview of filtering structures in our prototype

To support subscription pruning, we can utilize the previously described filtering structures without internal modifications. Though, we need to add two data structures: A selectivity table (Fig. 2, right) allows for the determination of the selectivities of predicates using the counting method described in Sect. 4.1. The selectivity table maps predicate identifiers to a counter stating the number of matchings of a predicate. For example, the predicate with identifier 2 had 15,000 matching event messages in the example in Fig. 2. This information allows for the estimation of the selectivities of subscriptions (cf. Sect. 4.1).

The second structure, the selectivity queue (Fig. 2, right), implements a priority queue ordered by degradation; it is required to perform post-pruning. Whenever a subscription is registered, we calculate the best pruning option for this subscription. Then, we insert this degradation (associated with the subscription identifier) into the selectivity queue, e.g., degradation 0.0001 for subscription identifier 2,000 in Fig. 2. Our selectivity queue orders its entries according to degradation and allow us to efficiently determine the subscription leading to the least selectivity degradation. To perform pruning, we remove the top element from the queue and perform the best pruning operation for the stored subscription.

To incorporate changes in selectivities, we can compare the current selectivity degradation of a subscription to the value stored in the degradation queue. There are various options to handle changes: (a) we always perform the pruning even if the degradation has worsened to a large extend, or (b) we only perform the pruning if the new degradation value is less than the one stored for the new top element in the queue. Another option, (c) is to allow changes in the degradation but only to a certain extend. In our experiments, presented in the next section, we do not consider these changes in degradation, i.e., we always prune the subscription stored on top of the degradation queue.

System Architecture. We have illustrated the architecture of our system in Fig. 3: The central part of our distributed pub/sub system is the routing component, which performs subscription forwarding and routing decisions. It contains the index and pruning structures, as described above, to determine all matching subscriptions for incoming event messages. This component is used for routing decisions: Incoming event messages are filtered and, if matching subscriptions have been registered by neighbor brokers, are routed towards these neighbors. Otherwise, subscribers are notified about these events. Both, notifications for subscribers and routings to neighbor brokers, is obtained via notification structures stating the actions to perform for subscriptions in case of matching. They allow for a transparent handling of notifications for brokers and clients: Notifications for subscribers initiate the notification of the client. For neighbor brokers, they initiate the routing of event messages. For each neighbor, we perform the forwarding of an event message only once.

Each broker component also contains a message handler, which runs in a separate thread. It consists of an incoming message queue as well as processing components, a subscription and an event handler. The message handler extracts incoming messages and assigns them to the message processing elements. In case of an event message, the event handler performs routing and notifications by the help of the index structures in the routing component. For subscriptions, the subscription handler initiates the forwarding of subscriptions and their registration, again by the help of the index structures.

For each neighbor in the network, a neighbor handler, running in an own thread, manages the sending and receiving of messages. Other components store outgoing messages in a queue (i.e., the message handler stores subscriptions to be forwarded and the routing component events to route). The sending handler sends these messages to the neighbor broker via a TCP/IP socket connection. The receiving handler waits for incoming messages (subscriptions and event messages); these are put into the incoming message queue to be processed by the message handler.

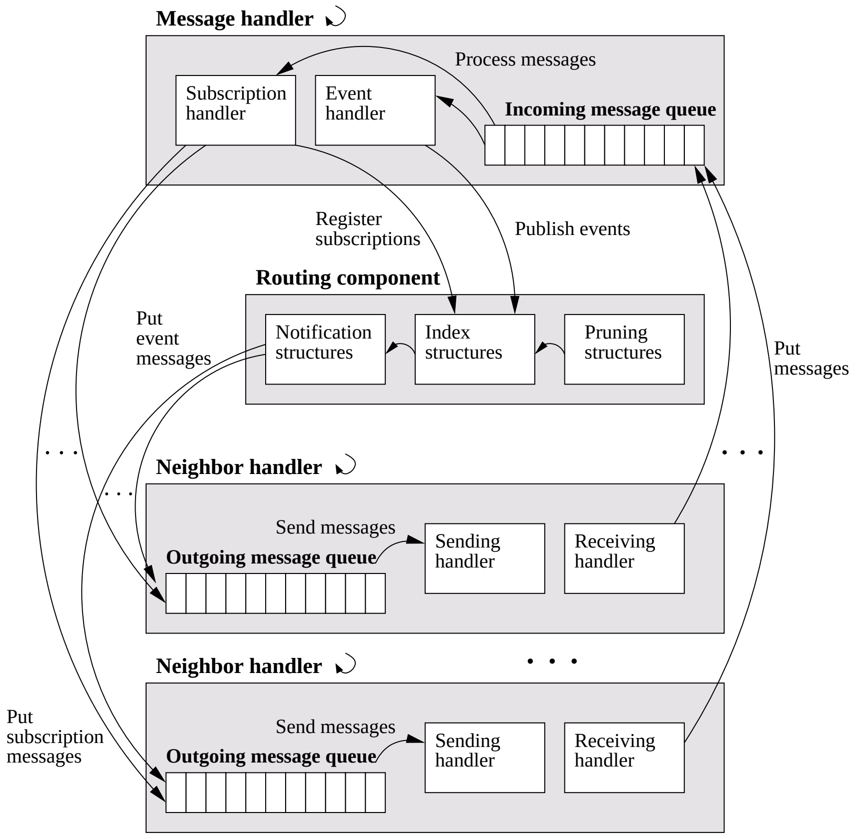


Fig. 3. Overview of the components of a broker in our prototype

To analyze the routing in our prototype when performing prunings, we have run a series of experiments. Their results are presented in Sect. 5.

4.5 Subscription Pruning and Other Routing Optimizations

We can utilize subscription pruning in combination with existing routing optimizations. In the following subsection, we discuss the combination potential of subscription pruning with other optimizations. Afterwards, we examine how to use subscription pruning to build an imperfect merger, i.e., how to utilize subscription pruning to solve the merging problem.

Combining Routing Optimizations. In contrast to the covering optimization approach (Sect. 3.1), which aims at decreasing the number of routing table entries, subscription pruning targets at reducing the complexity of routing entries. This allows for the combination of pruning and covering due to these opposing dimensions of optimization:

Brokers should utilize coverings among subscriptions to remove redundant routing entries. Additionally, we can apply subscription pruning to reduce the complexity of these entries.

Similarly to covering, we can apply pruning in combination with subscription merging (Sect. 3.2), which also optimizes in respect to a dimension opposed to the one of subscription pruning. Thus, next to its own positive effects on the routing process, subscription pruning can exploit the benefits of other optimizations due to the possibility to combine it with other approaches.

Utilizing Pruning for Imperfect Merging. We can also utilize subscription pruning to build an imperfect merger for arbitrary Boolean subscriptions. This overcomes the restrictions of current merging approaches, which only work on conjunctions (Sect 3.2).

In order to find an imperfect merger, we firstly create a perfect merger by building the disjunction of all subscriptions to be merged. This results in a subscription summarizing the descriptions of all merged subscriptions. Secondly, we prune the resulting merger. With each pruning step, the merger becomes more inaccurate, i.e., imperfect. The number of performed merging operations determines the degree of imperfectness of our merger.

This approach to determine an imperfect merger automatically overcomes the problem of deciding how to merge subscriptions [14]. Since we are using the pruning approach based on selectivity estimations, the imperfect merger is expected to be relatively accurate for its size and less complex than the original perfect merger, i.e., we can faster evaluate the imperfect merger.

5 Experimental Evaluation

In this section, we present an extensive experimental evaluation of subscription pruning using our prototype. We focus on the three quantitatively measurable characteristic parameters of event routing we have identified in Sect. 3: memory requirements, network load, and computational complexity (i.e., system throughput). In the next subsection, we identify the drawbacks of current evaluations of routing optimizations. Then, in Sect. 5.2, we describe our experimental setup. We analyze the routing behavior in one broker component when applying subscription pruning in Sect. 5.3. The influence of subscription pruning on a network of brokers is evaluated in Sect. 5.4.

5.1 Previous Evaluations of Routing Optimizations

In the existing literature, we can find some evaluations of routing optimizations in pub/sub systems. However, recent analyses are too restricted in respect to two attributes: Firstly, they merely analyze specialized settings requiring conjunctive subscriptions only. Secondly, these evaluations focus on a limited set of parameters or even on one parameter only, e.g., network load.

For the REBECA system, the effects of subscription covering and merging on network load and memory usage have been investigated in [16]. REBECA only supports conjunctive subscriptions that are even further restricted to contain at most one predicate per attribute. The assumptions in [16] are rather limited: Subscriptions contain at most two conjunctive predicates; the probability of a covering relationship between two subscriptions is 50%.

Also the work in [14], part of the PADRES project, analyzes subscription covering and merging. The presented filtering algorithms and their evaluations only consider conjunctions. Experiments focus on routing table size, routing time and amount of false positives. This appears as a rather extensive analysis; however, the authors only analyze a centralized setting and assume high similarities among subscriptions, i.e., 200,000 subscriptions contain at most 5,000 distinct predicates.

SIENA [7] supports conjunctive subscriptions and utilizes subscription covering. Only the network load for event routing is considered in the evaluation. There exist at most 1,000 objects of interest, i.e., distinct event messages, and 10,000 interested parties, i.e., subscriptions. Hence, the similarity among subscriptions is particularly high, and the event space is rather tiny. This does not allow for a generalization of the results to general settings.

The XML-based pub/sub system presented in [9], XROUTE, assumes conjunctive subscriptions and supports the determination of coverings among subscriptions. There is a brief evaluation of memory usage for filtering; however, the influence of coverings on filtering is not analyzed. The work in [19] investigates a variant of imperfect merging. Evaluations only consider routing time; subscriptions are restricted to conjunctive forms. CBCB [8] supports subscriptions in DNF, i.e., several conjunctive subscriptions can be combined to one subscription. However, arbitrary Boolean subscriptions are not supported. The evaluation mainly considers network load; there is a tiny section on memory usage. The effects of coverings are not explicitly analyzed.

This selection of pub/sub systems and their analyses shows the limitations of current routing optimizations: Firstly, they only support conjunctive subscription languages, and thus they are not applicable to general application settings. Secondly, recent evaluations restrict their test settings to allow for the relationships or similarities among subscriptions required by recent optimizations.

Subscription pruning is briefly analyzed in [5]. There, the expected network load and the memory usage is evaluated for a centralized setting. Our analysis in this paper is more extensive, focuses on more parameters, and involves a distributed setting.

5.2 Experimental Setup

For our evaluation, we have chosen a setting characteristic for electronic commerce applications using the results of an analysis of online book auctions on eBay[2] [5]: Events specify ten attributes, e.g., author, format, and price, and they follow a typical distribution for online book auctions. For subscribers, we analyze three characteristic patterns of subscriptions for book auctions: Subscription class 1 involves six predicates and four operators (one disjunction and three conjunctions); Subscription class 2 contains 12 predicates using 10 Boolean operators (two of them disjunctions). Class 3 of subscriptions uses seven predicates and six operators (three disjunctions). Attributes and op-

[2] http://www.ebay.com/

erators of predicates in these subscriptions are predefined; their values are determined randomly in our experiments. For a detailed description of these classes, we refer to [5].

In all experiments, we register 200,000 randomly created subscriptions conforming to the three subscription classes described above. We always show our results for the three classes individually; additionally, we present a setting involving randomly created (uniformly distributed) subscriptions from all three classes. Our results of measurements regarding time efficiency represent the average value for publishing 100,000 event messages.

Filtering brokers are run on machines with a total of 512 MB of RAM using a 2 GHz processor. This setup, using only moderately-equipped machines, is allowed by the focus of our filtering and routing algorithms: efficient filtering without utilizing tremendous memory resources. In the distributed setting, brokers are connected by a 10 Mbps network. To avoid influences and characteristics of a chosen network topology, we decided to connect our brokers as a straight line. This is the best way to directly show the influence of pruning operations in a distributed setting without incorporating effects originating from a specific topology. In our experiments, we restrain our network to contain five broker components.

5.3 Results in One Broker

In this subsection, we present the influence of subscription pruning in single broker components without forwarding event messages to neighbor brokers. We analyze space usage, time efficiency, and the expected increase in network load; we also correlate these parameters to each other.

In Fig. 4(a), 4(b), 5(a), and 5(b), we illustrate the space usage of index structures correlated to the expected network load. Figures 4(c), 4(d), 5(c), and 5(d) show the time efficiency of event filtering[3] correlated to the expected increase in network traffic. We have chosen this arrangement of our figures to allow for the recognition of the relationships between memory usage (upper subfigures) and filter efficiency (lower subfigures) as well. The abscissae of the figures always present the proportional amount of performed pruning operations ranging from 0 (situation without prunings) to 1.0 (all prunings have been performed, i.e., the next pruning would remove a whole subscription).

Right ordinates show the proportion of matching events compared to the 100,000 published ones. This indicates the increase in network load since, in a distributed setting, these events are forwarded to the neighbors. Left ordinates show the proportion of removed predicate subscription associations due to prunings (Fig. 4(a), 4(b), 5(a), and 5(b)). The maximal value of 100% is reached if all possible pruning operations have been performed (then, the abscissa also equals 100%). In Fig. 4(c), 4(d), 5(c), and 5(d), left ordinates represent the time efficiency, i.e., the average filtering time per incoming event message. Less filtering times mean increasing time efficiency and system throughput.

[3] Time efficiency can be directly converted into system throughput by building the reciprocal value.

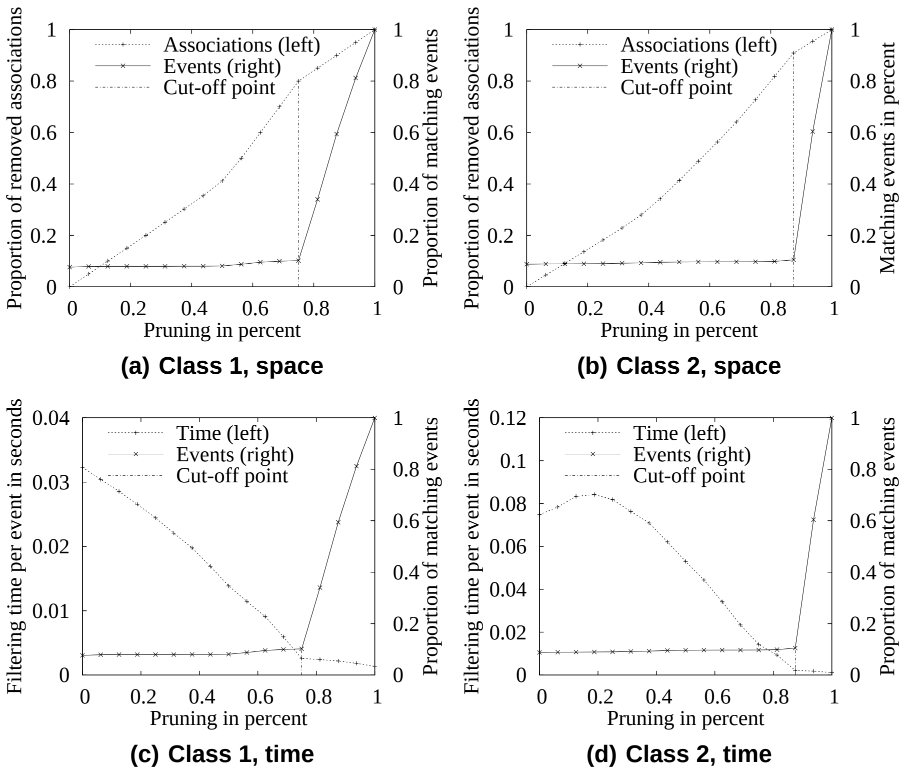
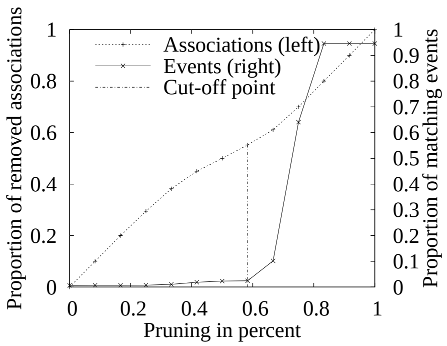


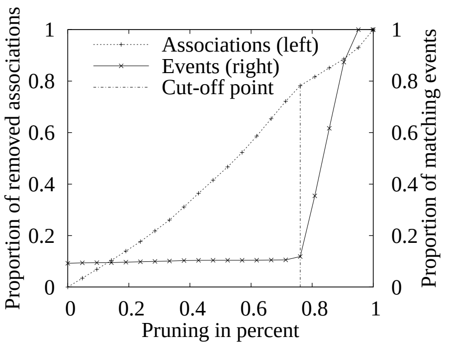
Fig. 4. Space efficiency and time efficiency compared to the number of matching events/expected network load using Subscription classes 1 and 2

In all figures, we also show the cut-off point. This point describes after which amount of pruning operations we can realize a strong increase in fulfilled event messages, i.e., a sharp bend in the respective curve. This point varies from 58% for Subscription class 3 to 88% for Subscription class 2. Up to the cut-off point, the number of matching events increases only slightly, i.e., subscription tree pruning is a valuable optimization. Relating the cut-off point to the memory requirements (Fig. 4(a), 4(b), 5(a), and 5(b)) shows the change in space usage when performing the respective amount of pruning operations. For example for Subscription class 1, the space usage has been relieved by approximately 80% of its maximal value. Concerning filter efficiency (Fig. 4(c), 4(d), 5(c), and 5(d)), for Subscription class 1 we could reduce the filtering time from 0.032 seconds per event to 0.0026 seconds per event. This is a decrease in filtering time by 92%. Similarly, we can correlate memory usage, filter efficiency, and expected network load for the other settings at all stages of pruning (shown at the abscissae of the different figures).

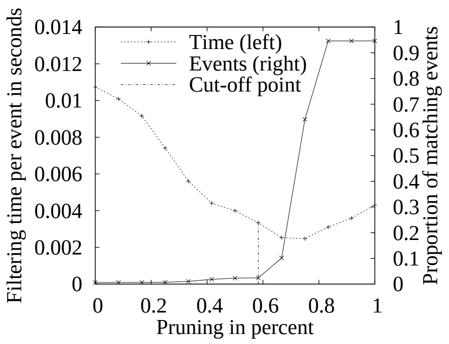
An overview of the results at the cut-off point is given in Table 2. The second column states the absolute increase in matching events at the cut-off point (using a total of 100,000 published events). Column 3 describes the proportional relief in memory



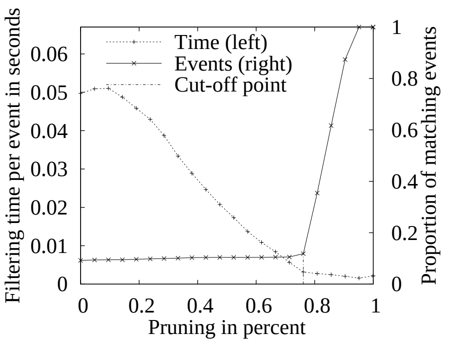
(a) Class 3, space



(b) Classes 1–3, space



(c) Class 3, time



(d) Classes 1–3, time

Fig. 5. Space efficiency and time efficiency compared to the number of matching events/expected network load using Subscription classes 3 and 1–3

resources compared to the maximal possible decrease in space usage when applying pruning. The proportional reduction of filtering time at the cut-off point compared to the non-pruning situation is shown in the last column.

Table 2 clearly demonstrates the benefits of applying subscription pruning as routing optimization: Without heavily decreasing the accuracy of subscriptions (i.e., number of matching events/expected increase in network load, Column 2), we realize a strong reduction in both memory requirements (Column 3) and filtering time (Column 4) if considering individual broker components. The relief in memory usage varies for different subscription classes. In our settings, we could decrease the memory to at least 55% and at most 91% of the maximal possible reduction. At the same time, the filtering load was 69% to 97% less than in the routing algorithm without utilizing subscription pruning.

For Subscription class 2, we realize an initial increase in filtering time when performing pruning (Fig. 4(d)). This is due to the fact that selectivity degradation describes the effect of pruning operations on selectivity. However, a pruning operation also influences the filter algorithm: In case of Class 2, more subscriptions are regarded as candidate subscriptions [3]. Thus, if the complexity of subscriptions is only slightly de-

Table 2. Summary of results in one broker at the cut-off point

Subscription class	Increase in events	Relief in memory	Decrease in filtering time
Class 1	2,531	0.8	0.92
Class 2	1,718	0.91	0.97
Class 3	1,804	0.55	0.69
Classes 1–3	2,642	0.78	0.94

creased, the filtering load rises faintly. However, after performing more pruning operations, this side effect is compensated by the strong reduction in subscription complexity. For the setting involving the combination of all classes of subscriptions (Fig. 5(d)), we can experience this effect to a much smaller extend.

Our results in this section show the influence of pruning on individual brokers without considering the actual routing of messages. When considering a distributed setting, subscriptions tree pruning is also advantageous. However, the amount of reduction in computational complexity will be reduced due to the need to forward messages in the network of brokers. This is a time-consuming activity affecting the achievable degree of optimization in system throughput. We analyze the distributed setting in the next subsection.

5.4 Results in a Network of Brokers

After our analysis of the behavior of individual broker components when performing subscription tree pruning, we now investigate the influence of our routing optimization on the distributed pub/sub system as a whole. Again, we correlate space usage, time efficiency, and network load (in this experiments it is the real network traffic) to each other.

Figures 6(a), 6(b), 7(a), and 7(b) show the correlation of memory usage and network traffic. The difference to the figures in the last section is that right ordinates state the real proportional increase in network load. The same holds for our results regarding time efficiency (Fig. 6(c), 6(d), 7(c), and 7(d)). Note that there are different scales of these axes in different figures.

Comparing the change in routing efficiency when applying pruning in the distributed setting to our results in the last subsection (the setting involving a single broker only), we realize a smaller increase in throughput in the current setting. This meets our expectations and is due the the increased overhead for routing event messages in the network, i.e., events need to be serialized before sending and deserialized after receiving. However, the overall throughput using the distributed system is always much higher than the throughput using a single filtering component (cp. Fig. 6(c), 6(d), 7(c), and 7(d) illustrating the distributed system to Fig. 4(c), 4(d), 5(c), and 5(d) describing the centralized case).

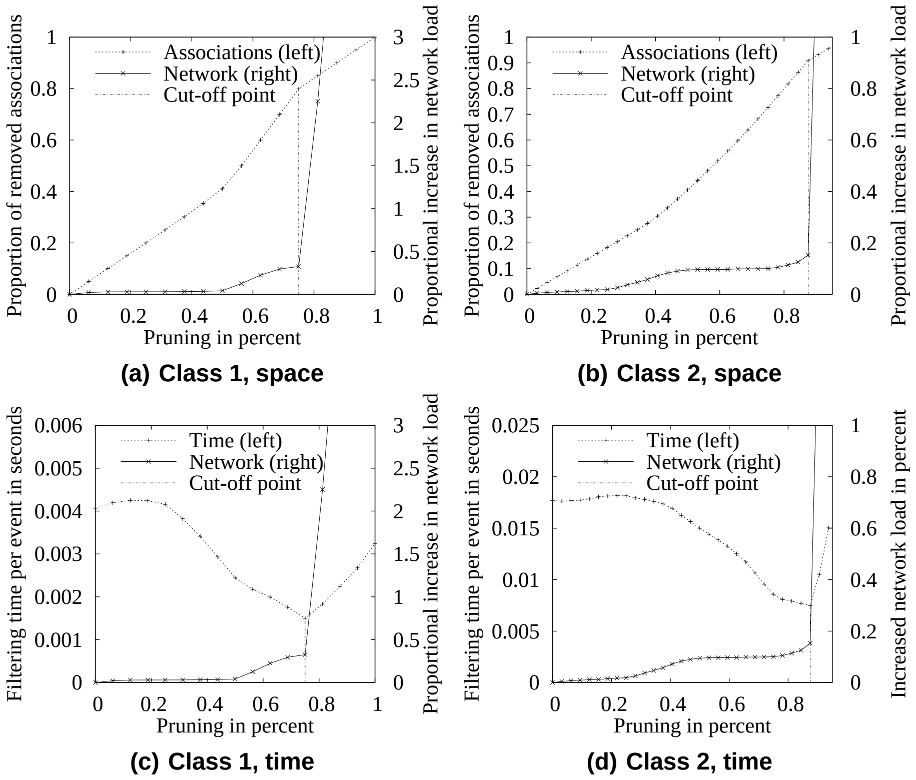
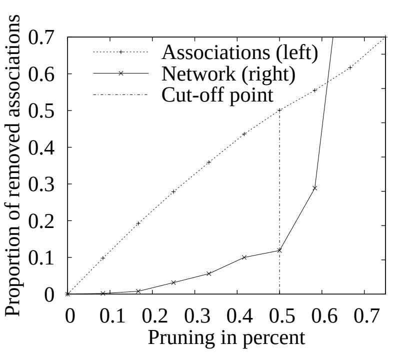


Fig. 6. Space efficiency and time efficiency compared to network load using Subscription classes 1 and 2

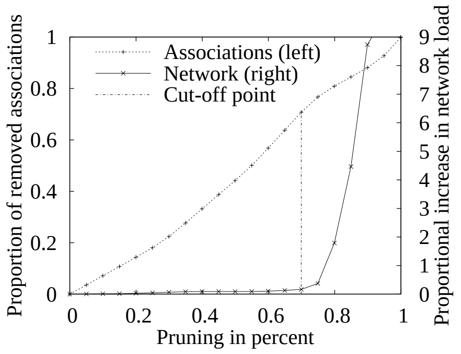
Furthermore, the distributed setting is more sensitive to increasing numbers of event messages to filter on. This results in a left-shifted cut-off point for Subscription class 3 (cp. Fig. 7(a) to Fig. 5(a)), which also affects the setting involving all classes of subscriptions (cp. Fig. 7(b) to Fig. 5(b)). This effect is caused by the additional overhead for sending messages, i.e., an only slightly increasing number of false positives introduces a noticeable computational overhead.

Meeting our expectations, the influence of less complex subscriptions to filter on (effect of applying pruning) outweighs the additional overhead for routing and filtering on more event messages (false positives). At the cut-off point, which is determined by the sharp decrease in filter efficiency in the distributed setting, we realize a more efficient event filtering in all scenarios (depicted in Fig. 6(c), 6(d), 7(c), and 7(d) for the different subscription classes). The filter efficiency increases by 26% to 63% in our settings. At the same point, the relief in memory usage is comparable to our results for individual broker components.

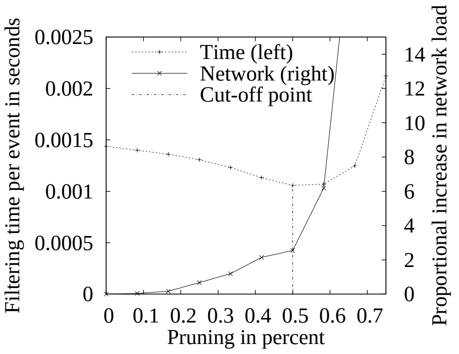
An overview of all results at the cut-off point is given in Table 3. Note that Columns 2 and 3 show the overall results for all brokers as well as Column 4 that is based on the filtering time for the distributed system.



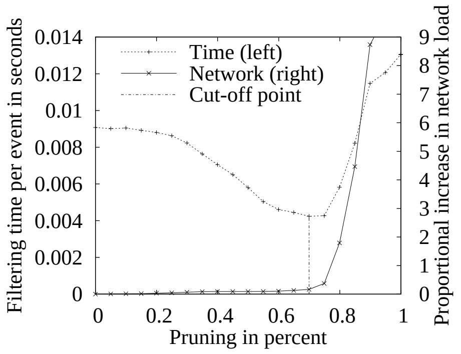
(a) Class 3, space



(b) Classes 1–3, space



(c) Class 3, time



(d) Classes 1–3, time

Fig. 7. Space efficiency and time efficiency compared to network load using Subscription classes 3 and 1–3

The results in Table 3 verify our theoretical analyses of subscription pruning in Sect. 3 and Sect. 4.2: Subscription pruning decreases the memory requirements for routing and filtering structures to a large extend (up to 91% of the optimal value in our experiments). This result is achieved without heavily increasing the network traffic created by event routing, i.e., the amount of false positives introduced by subscription pruning remains low. Also the overall routing efficiency of the pub/sub system is increased at the same time. In our settings, the efficiency (throughput) of the overall system improved by 26% to 63%.

6 Conclusions and Future Work

We have started this paper by analyzing five routing optimizations (covering, perfect merging, imperfect merging, predicate replacement, and subscription tree pruning) for distributed publish/subscribe systems. For this evaluation, we have identified six dimensions of parameters affecting the suitability of routing optimizations for general-purpose publish/subscribe systems. Our analysis has clearly identified subscription tree pruning as the favorable routing optimization for publish/subscribe systems for general settings.

Table 3. Summary of results in a network of brokers at the cut-off point

Subscription class	Increase in events	Relief in memory	Decrease in filtering time
Class 1	11,932	0.8	0.63
Class 2	6,322	0.91	0.58
Class 3	7,786	0.5	0.26
Classes 1–3	7,355	0.71	0.53

This is due to the optimization potential of subscription tree pruning for all Boolean subscriptions in contrast to only conjunctive subscriptions that are supported by covering and merging. Additionally, we can apply subscription tree pruning regardless of the individual and collective structures of subscriptions, such as their overlapping or similarity.

Then, we presented the details of our implementation of a subscription pruning-based publish/subscribe system. Our descriptions included the proposed system architecture as well as the required filtering, routing, and pruning structures. We also identified two variants to apply subscription pruning: Pre-pruning aims at optimizing the routing load in the publish/subscribe network as a whole whereas post-pruning focuses on optimizing the load in individual broker components. Furthermore, we discovered the potential to combine subscription pruning with other routing optimizations, which is further improving the overall optimization effects. We also argued how to utilize subscription pruning to solve the merging problem for arbitrary Boolean subscriptions.

The final part of this paper is a practical evaluation of subscription pruning using our system. We have shown the effects of subscription pruning on the quantitatively measurable of our previously identified characteristic parameters, i.e., network load, routing efficiency (system throughput), and memory usage. For our analysis, we have chosen an online auction scenario involving typical classes of subscriptions and distributions of events. We could show that subscription pruning is an effective routing optimization and have identified the existence of a cut-off point optimizing the benefits of subscription pruning: For our setting involving all subscription classes, at this point the routing efficiency and thus the throughput is increased by 53%. Additionally, the space usage is decreased by 71% of the maximal possible reduction. These strong improvements only cause a slight overhead in network traffic. Our experiments and their analyses are the first of their kind focusing on the correlation of several dimensions of parameters affecting the routing in publish/subscribe systems. They are also the pioneer work evaluating routing optimizations for general Boolean subscriptions.

In the future, we want to extend our prototype to support more data types for attributes and operators for predicates. We also plan to evaluate subscription pruning in the restricted settings that are supported by other optimization approaches. This allows for conclusions about the behavior of subscription pruning in comparison to these limited solutions.

References

1. G. Ashayer, H.-A. Jacobsen, and H. Leung. Predicate Matching and Subscription Matching in Publish/Subscribe Systems. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, pages 539–548, Vienna, Austria, July 2–5 2002.
2. S. Bittner and A. Hinze. Classification and Analysis of Distributed Event Filtering Algorithms. In *Proceedings of the 12th International Conference on Cooperative Information Systems (CoopIS 2004)*, pages 301–318, Agia Napa, Cyprus, October 25–29 2004.
3. S. Bittner and A. Hinze. A Detailed Investigation of Memory Requirements for Publish/Subscribe Filtering Algorithms. In *Proceedings of the 13th International Conference on Cooperative Information Systems (CoopIS 2005)*, pages 148–165, Agia Napa, Cyprus, October 31–November 4 2005.
4. S. Bittner and A. Hinze. On the Benefits of Non-Canonical Filtering in Publish/Subscribe Systems. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '05)*, pages 451–457, Columbus, USA, June 6–10 2005.
5. S. Bittner and A. Hinze. Pruning Subscriptions in Distributed Publish/Subscribe Systems. In *Proceedings of the Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, Hobart, Australia, January 16–19 2006.
6. A. P. Buchmann, C. Bornhövd, M. Cilia, L. Fiege, F. C. Gärtner, C. Liebig, M. Meixner, and G. Mühl. DREAM: Distributed Reliable Event-Based Application Management. In *Web Dynamics*, pages 319–352. Springer, 2004.
7. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems (TOCS)*, 19(3):332–383, 2001.
8. A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A Routing Scheme for Content-Based Networking. In *Proceedings of the 23rd IEEE Conference on Computer Communications (INFOCOM 2004)*, Hong Kong, China, March 7–11 2004.
9. R. Chand and P. A. Felber. A Scalable Protocol for Content-Based Routing in Overlay Networks. In *Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA 2003)*, pages 123–130, Cambridge, USA, April 16–18 2003.
10. M. Cilia and A. P. Buchmann. An Active Functionality Service For E-Business Applications. *ACM SIGMOD Record, Special Issue on Data Management Issues in Electronic Commerce*, 31(1):24–30, 2002.
11. A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query Merging: Improving Query Subscription Processing in a Multicast Environment. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):174–191, 2003.
12. F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD 2001)*, pages 115–126, Santa Barbara, USA, May 21–24 2001.
13. M. Guimarães and L. Rodrigues. A Genetic Algorithm for Multicast Mapping in Publish-Subscribe Systems. In *Proceedings of the Second IEEE International Symposium on Network Computing and Applications (NCA 2003)*, pages 67–74, Cambridge, USA, April 16–18 2003.
14. G. Li, S. Hou, and H.-A. Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems based on Modified Binary Decision Diagrams. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, pages 447–457, Columbus, USA, June 6–10 2005.

15. M. Mansouri-Samani and M. Sloman. GEM: A Generalised Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, 1997.
16. G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, September 2002.
17. G. Mühl and L. Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe Systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001.
18. G. P. Picco, G. Cugola, and A. L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfiguration. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS '03)*, pages 234–243, Rhode Island, USA, May 19–22 2003.
19. Y.-M. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, and P. Larson. Summary-based Routing for Content-based Event Distribution Networks. *ACM SIGCOMM Computer Communication Review*, 34(5):59–74, 2004.