Design and Analysis of an Efficient Distributed Event Notification Service

Sven Bittner, Annika Hinze
Department of Computer Science, University of Waikato
{s.bittner, hinze}@cs.waikato.ac.nz

Abstract

Event Notification Services (ENS) use the publish/subscribe paradigm to continuously inform subscribers about events they are interested in. Subscribers define their interest in so-called profiles. The event information is provided by event publishers, filtered by the service against the profiles, and then send to the subscribers. In real-time systems such as facility management, an efficiency filter component is one of the most important design goals.

In this paper, we present our analysis and evaluation of efficient distributed filtering algorithms. Firstly, we propose a classification and first-cut analysis of distributed filtering algorithms. Secondly, based on the classification we describe our analysis of selected algorithms. Thirdly, we describe our ENS prototype DAS that includes three filtering algorithms. This prototype is tested with respect to efficiency, network traffic and memory consumption. In this paper, we discuss the results of our practical analysis in detail.

1 Introduction

With the increasing popularity of the Internet one can access more and more information. A main problem for a user is to find the relevant information out of all available information. The Internet is based on the request/response paradigm: Users ask sources of information according to their interests. An answer concerning their request is replied. This principle is the basic concept of the WWW and the used HTTP protocol [30]: After requesting information from a web server, the specified web page is send as reply. Search engines (e.g., Google [5]) support user searches and improve the information retrieval process. They crawl through a variety of web sites, build indexes and the users have the opportunity to search the web by keywords. Search engines use the request/response paradigm. The flow of data is initiated by the user (pull mode): Each request is answered by exactly one response. New information about the information sources are only delivered to the users when a new request is made.

Another approach is the publish/subscribe paradigm, which is more focussed on the user needs: Users ask for information only once and are then continuously notified about new information and changes. The data flow is initiated by the sources of information (push mode). Newsgroups use this principle in a simple way with the

NNTP [28] protocol. There, users may subscribe for certain topics. Subsequently, they are notified about the topics chosen. An additional content-based filtering of the information beyond the general topics is not supported.

Event Notification Services solve this problem by using a more detailed description of user interests, frequent notification about occurred changes and publication of information. They act as a mediator between sources of information and interested users. Information is filtered by content by the Event Notification Service. According to the user's specifications, the filtered information is then forwarded to users. Predecessors of event notification services have been developed for selective dissemination of information [24] and information filtering [3].

Today, Event Notification Services are used in various applications, such as digital libraries, traffic control, medical science and information services in the Internet. Their functions vary from the publication of electronic papers and articles (e.g. Hermes [7]), the designing of medical therapies (e.g. PLAN [31]) and air traffic control [15] to the monitoring of web documents (e.g. Conquer [16]). In this paper we choose an application area with growing importance: facility management [10].

The tasks for which an Event Notification Service can be used are diverse, examples are the control of blinds, air conditioners and heating, communication with service providers in case of a facility breakdown, and the adaption of a room's brightness for optimal arranged workstations. A more detailed introduction into the principles of Event Notification Services are given in the next section.

1.1 Definitions

In this section, we describe and explain important terms in the context of Event Notification Services. We firstly describe Even Notification Services in general. Secondly, we present the principle of distributed Event Notification Services.

1.1.1 General Definitions

Event Notification Services (ENS) are systems that decouple sources of information and sinks of information. Sources of information are *publishers* of events. Sinks of information are called *subscribers* and the information itself is denoted as *events*. ENSs are used for the filtering of the publisher's events. They also notify subscribers about incoming events by so-called *notifications*. For the filtering the subscribers specify their interests with the help of *profiles*.

An illustration of an ENS in the context of facility management is shown in Fig. 1. On the right side of the figure, it shows the subscribers such as radiators and mobiles phones of the safety personnel. The left side shows the publishers, e.g. temperature sensors or other measuring instruments. The subscribers send profiles p_x (1) to the ENS, in which they define their interests. Publishers send events e_x (2) which describe changes of a state, an actual state at a certain time, or other external events. The ENS filters all incoming events and notifies interested subscribers using notifications b_x (3).

An event e_x is defined by a set of *tuples* $\mathcal{I}(e_x)$ of *attribute-value pairs* and an *event type* $\mathcal{T}(e_x)$: $e_x = (\mathcal{I}(e_x), \mathcal{T}(e_x))$. The event type T is defined by a set of *attributes*:

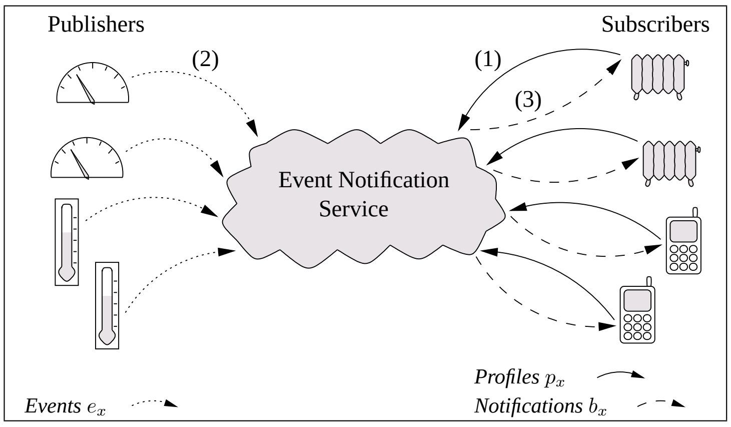


Figure 1: Overview of an ENS in the context of facility management

$T = \{a_1, \ldots, a_n\}$. The event type describes an event's structure. Each attribute a has a *domain* W. $\mathcal{W}(a)$ describes the domain of attribute a.

Each attribute of the event type of an event e_x has to be specified in exactly 1 attribute-value pair of e_x. The set of all events is denoted by $\mathcal{E}$, the publisher of an event e_x is denoted by $\mathcal{A}(e_x)$. For simplification an event e_x of type $T_1 = \{a_1, a_2\}$ with the attribute values $a_1 = w_1$ and $a_2 = w_2$ is written as follows

$$e_x : (a_1 = w_1, a_2 = w_2, T_1).$$

Subscribers are interested in profiles and specify their interests with the help of a *profile definition language*. Formally profiles specify *filter operations* on events, so subscribers are able the filter incoming events with the help of the ENS. The set of all profiles is denoted by $\mathcal{P}$. The subscriber of a profile p_x is stated as $\mathcal{A}(p_x)$.

An ENS filters all incoming events e_x of the publishers and notifies all subscribers of matching profiles p_x with a notification $b_x = (p_x, e_x)$. If event e_x matches profile p_x, we denote this by $e_x \succ p_x$.

More formally, a profile p_x is specified by a set of *predicates* $\mathcal{PR}(p_x)$ and an event type $\mathcal{T}(p_x)$: $p_x = (\mathcal{PR}(p_x), \mathcal{T}(p_x))$. A predicate $pr \in \mathcal{PR}(p_x)$ is a triple $pr = (a, op, o)$. Here a is an attribute, op a binary comparison operator (e.g. $=, <, >$) and o the right operand of the operator op. In a profile you can specify at most 1 predicate per attribute of its type. An attribute-value pair $t_y = (a_y, w_y)$ matches a predicate $pr_x = (a_x, op_x, o_x)$ (denoted by $t_y \succ pr_x$) iff

$$a_x = a_y \wedge w_y \; op_x \; o_x.$$

An event e_x matches a profile p_x ($e_x \succ p_x$) iff

$$\mathcal{T}(p_x) = \mathcal{T}(e_x) \wedge \forall pr \in \mathcal{PR}(p_x) \exists t \in \mathcal{I}(e_x)(t \succ pr).$$

In other words, an event matches a profile if both specify the same type and all predicates of the profile evaluate to *True* on the event's attribute-value pairs. The set of events matching a profile p_x is denoted by $\mathcal{E}(p_x)$ and defined by

$$\mathcal{E}(p_x) = \{e_y \in \mathcal{E} | e_y \succ p_x\}.$$

For simplification, we write a profile p_x of type $T_1 = \{a_1, a_2\}$ with the predicates $(a_1, =, w_1)$ and $(a_2, >, w_2)$ as follows

$$p_x : (a_1 = w_1 \wedge a_2 > w_2, T_1).$$

Efficient filtering algorithm are usually implemented in main memory. The use of secondary memory and mechanisms for persistency in traditional databases would lead to a significant loss of efficiency (see [12, 25]).

1.1.2 Distributed Event Notification Services

Distributed ENSs are more efficient that centralised solutions - efficiency is measured in number of events that can be filtered per time unit when using a profile set of size $|\mathcal{P}|$. The use of a distributed system is additionally triggered by the need for scalability regarding the number of clients and events, since the necessary communications cannot be handled by a central system. A distributed ENS (Fig. 2) is described by a network

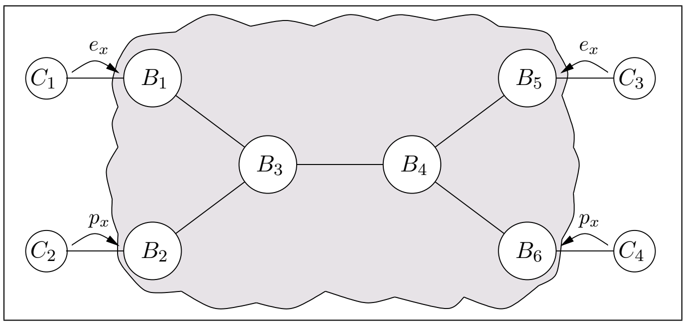


Figure 2: Overview of a distributed ENS

of *brokers* B. These brokers form an arbitrary network topology. A broker does not have to know the whole network $\mathcal{V}$. Instead, it is sufficient for broker B to know the set of *neighbour brokers* (denoted by $\mathcal{N}(V)$).

Subscribers and publishers (also called clients) are connected to an arbitrary broker B, which acts as interface to the distributed ENS. This broker is then called *local broker*. Its clients are referred to as *local publishers* or *local subscribers*. In Fig. 2, we illustrate a distributed ENS with 6 brokers B_1 to B_6. The clients C_1 and C_3 are publishers of events e_x. They are connected to their local brokers B_1 and B_5. C_2 and C_4 are subscribers of profiles p_x and they are communicating with the brokers B_2

and B_6. In Fig. 2 the neighbour brokers of broker B_4 are B_3, B_5 and B_6 ($\mathcal{N}(B_4) = \{B_3, B_5, B_6\}$).

Differences in distributed filtering strategies are, for example, the spreading of filter complexity and the used network topology. Design questions for distributed ENS are: Which brokers perform the filtering? How do we communicate with the subscribers? Are events filtered several times or exactly once? The next sections address these design questions.

1.2 Focus and Outline

In this paper, we describe the design and the results of an extensive evaluation of a distributed ENS. The paper is divided into two parts. In the theoretical part, we analyse and classify distributed filtering algorithms. In the practical part, we present the architecture and the evaluation of our system DAS - a prototype of a distributed ENS.

The paper is structured as follows. In Section 2 we give a first overview about related work regarding the design and evaluation of ENSs. In Section 3, we present a classification of distributed filtering algorithms (which is a generalisation and extension of the discussion of related work in Section 2). We select three algorithms for practical implementation, analysis and evaluation. The implementation of the centralised and distributed filtering algorithms and their architecture is presented in Section 4. Section 5 presents the results and discussion of extensive experiments undertaken with DAS. We evaluated the influence of different systems parameters on efficiency, network load and memory consumption. Finally, we conclude and discuss future work in Section 6.

2 Related Work

This section describes related work in the area of distributed event filtering in Section 2.1. We present previous strategies for the evaluation of ENSs in Section 2.2. We finish the section with a summary in Section 2.3.

2.1 Related Systems

Two main strategies for distributed event filtering can be distinguished: Rendezvous nodes, which are described in Section 2.1.1 and distributed filtering, which is presented in Section 2.1.2 and Section 2.1.3. In Section 2.1.4, strategies to minimise redundancies among profile definitions are shown.

2.1.1 Rendezvous Nodes

Rendezvous nodes have been introdued by Rowstron et al. [23] and extended by Pietzuch and Bacon [21]. The used network topology is an acyclic graph. A rendezvous node B is a broker that specialises in the filtering of a given set of event types. Rendezvous nodes are responsible for distinct event types; for the filtering of each profile $p \in \mathcal{P}$ and each event $e \in \mathcal{E}$ there exists exactly one rendezvous node. Rendezvous nodes are meeting points for events and profiles of their specialised types. Therefore, all events e_x and profiles p_x are forwarded to their respective rendezvous node B according to their event type ($\mathcal{T}(e_x) \in \mathcal{T}(B)$ or $\mathcal{T}(p_x) \in \mathcal{T}(B)$). The notification about an event is delivered on the reverse path over which the profile has been forwarded by the network of brokers.

Two systems use rendezvous nodes for filtering: Scribe [23] and Hermes [21]. In Scribe, no content-based filtering is supported. Subscribers can subscribe for different topics to be notified about all related events. The filtering of events is exclusively done in rendezvous nodes as described above. Hermes uses a more sophisticated approach. There filtering takes place in the rendezvous nodes but also in all brokers which have forwarded profiles to rendezvous nodes. So events are forwarded to their rendezvous node and on that path these events are already filtered.

Rendezvous nodes can be seen as a combination of a centralised and a distributed filtering strategy because selected brokers are responsible for the filtering of a predefined set of profiles. So the advantage of distributing the filtering (partitioning filter task and memory consumption) is lost. Hermes tries to solve this problem by filtering at intermediate nodes at the expense of memory consumption. However, there is only an improvement in special cases because each rendezvous node still has the complete filter load. All nodes in subtrees not rooted on the path from an event's local broker to the rendezvous node do not profit by this strategy. In addition, there is high network traffic at the rendezvous node.

2.1.2 Distributed Filtering: Hierarchical Networks

Two distributed filtering strategies use hierarchical networks: Link State Matching [2] and hierarchical brokers [6, 32].

Link State Matching assumes that each broker knows each subscribed profile. Prior to the filtering, a tree-based filter structure [1] is built. In this structure, each level evaluates exactly one attribute. For each level of this tree, a vector is maintained with a size equal to the number of children (that are brokers) in the network structure. The vectors store three values per slot: *Yes*, *No*, *Maybe*. The former two values indicate if profiles are contained in the corresponding network path. In the latter case a decision cannot be made in this tree level. When filtering events the tree is processed until all values in the vector contain *Yes* or *No*. Then each event is forwarded to the respective brokers.

Yu et al. [32] arrange their brokers in groups that are connected in a tree structure. The roots of each group can also be members of other groups. Therefore, the overall structure is a tree with trees as nodes. For the filtering the following protocol is used: Brokers store the information that is necessary to forward events to all clients with matching profiles or neighbour brokers in their group. Exchanging information is done by forwarding profiles to all children or as far on the path to the root as necessary. Events are recursively forwarded to all interested neighbours. Carzaniga et al. [6] use a related strategy that assumes a simple tree structure. Events are filtered in each broker and forwarded to the parent node. As a consequence, both strategies assume that parent nodes know all profiles of their children.

All three strategies have the problem of high load at the root node. In Link State

Matching, the root has to filter each incoming event. An advantageous fact is that not all attributes need to be evaluated. Yu et al. minimise the number of nodes involved in filtering by their structure with trees as nodes of trees.

2.1.3 Distributed Filtering: Point-to-Point Networks

Carzaniga et al. describe in [6] two algorithms for the filtering in point-to-point networks. The network topology is an acyclic graph.

Profile forwarding [6] forwards profiles in the whole network of brokers. Events are filtered at the publisher's local broker. Each broker performs filtering steps to forward profiles to neighbours with related subscribers or such neighbours itself. Subscribers are notified on the reverse path their profile was forwarded to the filtering broker. Carzaniga states that profiles should be filtered as close as possible to the publishers.

Event Forwarding [6] uses an opposite strategy. Events are forwarded in the whole network instead of profiles. Subscribers are notified by their local brokers. Profiles are not forwarded. The filtering is done as close as possible to the subscribers.

As summary we can state that point-to-point networks are a generalisation of hierarchical networks since acyclic graphes (free trees) are a superset of trees with a fixed root. Profile forwarding seems to be a good strategy if less events match profiles. If nearly all events are matched event forwarding seems appropriate because events have to be forwarded in the whole network of brokers in any case. So choosing an appropriate strategy depends on the distribution of events and profiles.

2.1.4 Minimising Redundancies

Hermes [21] and other distributed filtering strategies [2, 6, 32] use additional optimisations. Their aim is to minimise redundancies among profiles and thereby maximise the filter efficiency. There are two main strategies: covering and merging. For these strategies brokers act as subscribers. In the case of a notification, a post-filtering is applied to notify the real subscribers. Here we only consider covering.

Definition of Covering. According to Mühl et al. [19, 20] profile p_x covers profile p_y (written $p_x \sqsupseteq p_y$) if $\mathcal{E}(p_x) \supseteq \mathcal{E}(p_y)$ holds. If $\mathcal{E}(p_x) \supset \mathcal{E}(p_y)$ holds, we say real covering ($p_x \sqsupset p_y$).

An optimisation is that profile p_x only needs to be forwarded from broker B_x to a neighbour broker if no profile p_y with $p_y \sqsupset p_x$ has already been forwarded. Otherwise $\mathcal{A}(p_x)$ is also notified correctly because the set of notifications for p_y is a superset of the set of notifications for profile p_x. Because of filtering in B_x subscribers of p_x and p_y are both notified correctly. Another optimisation is: If neighbour broker B_x forwards a profile p_x the profiles $\{p_y \in \mathcal{P} | \mathcal{A}(p_y) = B_x, p_y \sqsubseteq p_x\}$ (profiles from B_x covered by p_x) do not need to be filtered anymore.

Computation of Covering. The computation of coverings is sometimes quite complex, possibly NP-hard [8]. The computation of the covering can be simplified by restricting the operators. The computation can be done either when subscribing and unsubscribing the profiles or on request. Both concepts are described in the following paragraph.

Immediate Computation Carzaniga et al. [6] suggest a poset structure to immediately compute coverings. A poset is a partially ordered set. All profiles are inserted in and removed from a poset whenever subscriptions or unsubscriptions take place. Using the poset in that way results in inefficiency, since insertions and removals take $O(k)$ time if the poset consists of k elements. Next to this running time storing all edges needs lots of main memory.

Computation on Request Mühl suggests in [17] to compute coverings only on request so that the covering relations do not need to be stored. Unfortunately, the computation can become expensive. Instead of computing the set of directly covered profiles, we calculate the set of covered profiles. In most cases this creates a higher network traffic load.

2.2 Related Evaluations

Several evaluation of ENSs have been performed. Most of them use simulations to derive statements about certain filtering algorithms. Direct comparisons of different filtering strategies are hardly to find. It is also questionable whether the results of the developed simulators can be generalised to practical situations.

In [6], Carzaniga uses 100 brokers to compare event forwarding and profile forwarding in a simulator implemented for this task. There is a maximum of 1,000 distinct profiles or events in the experiments. Altogether only 10,000 profiles can subscribe to the ENS. The network traffic is used for evaluation. The costs for any computations are not examined.

Mühl uses 107 brokers in [18] in a real setting. There is a maximum of 120,000 profiles, but there is only 1 publisher of events. There are 2 kinds of experiments with either 1,000 or 100,000 distinct events. For an evaluation mainly the sizes of the routing tables and the network traffic is used.

Pietzuch and Bacon have also developed an own simulator in [22]. They compare rendezvous nodes [21] with a kind of profile forwarding [6]. The virtual network consists of a maximum of 1,000 brokers and at most 25,000 profiles are subscribed. They evaluate the sizes of the routing tables, the number of messages and the latency for notifications. But this latency measures per notification. So more profiles mean less latency, which contradicts the intuitive idea of latency.

However, a detailed evaluation of different filtering algorithms is still missing. Especially, previous work is lacking variations in numerous system parameters. Moreover, the time for computation is often ignored.

2.3 Summary

In this section, we presented selected filtering strategies and initial steps of a theoretical evaluation of the strategies. In Section 2.1.1 we described rendezvous nodes as specialised types of brokers. In Section 2.1.2 distributed filtering in hierarchical networks was presented. Using point-to-point networks with the same strategy was the subject of Section 2.1.3. For minimising redundancies among profile definitions we introduced covering in Section 2.1.4. Finally, we compared previous evaluations of filtering algorithms in Section 2.2. We pointed out the lack of comparisons of different strategies in real settings with varying system parameters.

The filtering strategies we described in this section are specific solutions. Other approaches are possible. In order to undertake a detailed evaluation of distributed filtering algorithms, a detailed classification of algorithms is needed. This would allow for a structured combination of different approaches, using concepts of different strategies and achieve further improvements (i.e. more efficiency or less network traffic). Furthermore, we can evaluate and compare different algorithms in a more structured way. Our first-cut classification and a more detailed comparison and theoretical evaluation of the algorithms is presented in the next section.

3 Classification of Filter Algorithms

In the last section we presented different strategies for distributed filtering in ENSs. We now generalise these strategies and classify the previously presented algorithms. In particular, we combine optimisation strategies with various filtering strategies. The most important assessment criteria for distributed filtering strategies are efficiency and scalability (efficiency is needed for fast filtering and scalability for efficiency in case of large amounts of profiles and high event frequencies). The criteria we use for our analysis are in detail:

Network traffic: Network connections between brokers and clients only have a limited transfer rate. Less traffic is therefore advantageous for a distributed filter algorithm. Especially because of the time needed for input/output in relation to computation in brokers. Traffic in case of subscribe and unsubscribe (which only emerges at startup and reconfiguration) is not as important as traffic caused by events.

Memory usage: Memory usage in case of increasing numbers of profiles is a very important criteria. Naturally we have limited resources. So we should prefer algorithms with less memory consumption.

Efficiency: Efficiency measures how many events can be filtered in a second (or an arbitrary time unit) dealing with a fixed number of profiles. Efficiency is influenced by network traffic and memory consumption. But also other factors are important, such as the number of filtering steps until the final notification is delivered.

Scalability: Scalability is a measure for the behaviour of a distributed filter algorithm in case of increasing numbers of profiles. Generally an algorithm is able to filter less events if the number of profiles increases. Also using a fixed number of profiles and increasing the number of brokers the algorithm should be able to filter more events per time unit. Scalability is partially influenced by network traffic and memory usage.

In the following subsections we present our classification of filter algorithms. In Sect. 3.1 we analyse the communication with subscribers. Section 3.2 studies the spreading of filtering complexity. The filtering location is investigated in Sect. 3.3 and the memory strategy is analysed Sect. 3.4. Section 3.5 combines the presented distinctive features to complete filter algorithms and evaluates them. Finally in Sect. 3.6 a conclusion is drawn.

3.1 Communication with Subscribers

In this section we present different option for the communication with subscribers. There are three strategies for brokers: direct communication, forwarding via the network of brokers and transparent subscriptions using brokers as proxies.

3.1.1 Direct Communication

Using this strategy, brokers and subscribers communicate directly. Notifications are delivered via a direct network connection from the filtering broker to the subscriber of the matching profile. Other brokers are not involved into this notification.

Advantages: Notifications are delivered directly after the filtering. The delay between filtering and notification is less, since other brokers are not involved.

Disadvantages: Since brokers cannot handle permanently open connections with all subscribers, the connections have to be set up the moment they are needed. This initiation of connections requires time and resources, which reduces the advantageous performance of this approach. Furthermore, subscribers need to accept connections, act as servers and are therefore higher loaded. We also need an advanced error handling. Using a connectionless protocol results in unreliable notifications.

Direct communication is used in event forwarding [6] and Link State Matching [2].

3.1.2 Forwarding via the Network of Brokers

This strategy uses indirect communication via the network of brokers. Notifications are delivered to neighbour brokers, which forward them either to local subscribers or to their neighbour brokers.

Advantages: No connections between brokers and non-local subscribers are needed. So subscribers and brokers are unburdened: Subscribers need not to accept connections and brokers not to set up them.

Disadvantages: Using forwarding increases the latency between filtering and notification. Several options exist for determining the neighbour node that delivers a notification. Either the notification is forwarded to several neighbours at once (which implies higher network traffic) or it is forwarded each neighbour after the other (which increases latency). Additionally, we can index all subscribers in all brokers (increases memory usage). Finally, the path of each subscription can be stored. For a notification, the path would be followed backwards.

Forwarding via the network of brokers can be used in profile forwarding [6], in the systems Hermes [21] and Scribe [23] and in the hierarchical approach in [32].Note that these are possible combinations of approaches, not actual implementations.

3.1.3 Transparent Subscriptions Using Brokers as Proxies

This approach uses brokers as proxies of subscribers. When forwarding a profile to a neighbour broker the forwarding broker acts as subscriber. In case of notifications this proxy decides on further delivery (they should be delivered to all neighbours with matching profiles).

Advantages: We do not need direct communication with subscribers (except local subscribers and neighbour brokers). Furthermore we can apply the optimisations (presented in Sect. 2.1.4) to profiles of different subscribers.

Disadvantages: The expected memory usage is higher because the ENS needs to store information about real subscribers (under circumstances that are again proxies). However, that information is spread among all brokers. Since events are processed in several broker also latency is increased.

Profile forwarding [6] and Hermes [21] suggest the usage of proxies.

3.2 Spreading of Filtering Complexity

A difference among distributed filtering strategies is the spreading of filtering complexity among brokers. There are two approaches: exclusive filtering and distributed qualified filtering.

3.2.1 Exclusive Filtering

Using exclusive filtering brokers are responsible for filtering a predefined set of profiles, e.g. classified by event types. After filtering subscribers are notified directly or via the network of brokers (Sect. 3.1).

Advantages: This approach is easy to realise. Also there are no redundant profile information among brokers, which reduces memory usage.

Disadvantages: There is a problem in case of notifications of non-local subscribers. Under circumstances if we use forwarding via the network of brokers we send multiple notifications to the same neighbour broker. Using direct communication results in setting up connections to subscribers.

The system Scribe [23] and Link State Matching [2] use exclusive filtering. Event forwarding [6] and profile forwarding [6] could use exclusive filtering.

3.2.2 Distributed Qualified Filtering

Distributed qualified filtering uses a division of work. Each broker accomplishes the filtering steps to find all neighbour brokers with matching profiles. The event is then forwarded to them. Additionally, all local profiles are filtered and their subscribers are notified. An attenuation is the determination of all brokers with matching local profiles and the forwarding of events to them.

Advantages: Distributed qualified filtering saves memory. Events are only forwarded to brokers with matching profiles. So we minimise network traffic and divide the filtering among brokers.

Disadvantages: Events are filtered more than once when using this approach. So we increase latency.

Profile forwarding [6], the system Hermes [21] and the hierarchical approach in [32] use distributed qualified filtering.

3.3 Filtering Location

There are different approaches for the filtering location: filtering as close as possible to the subscribers, filtering as close as possible to the publishers and filtering at fixed brokers. Finally the filtering can be spread among brokers (distributed qualified filtering, Sect. 3.2.2).

3.3.1 Filtering as Close as Possible to the Subscribers

In this approach events are forwarded to all brokers. They accomplish filtering for their local subscribers.

Advantages: The memory usage is determined by the number of local subscribers. If we bound the number of subscribers a broker can handle we achieve well scalability.

Disadvantages: Events have to be forwarded in the whole network of brokers. If we expect high event frequencies the network is heavily loaded.

Examples using this strategy are event forwarding [6], Link State Matching [2] and the hierarchical approach in [32].

3.3.2 Filtering as Close as Possible to the Publishers

Here we forward all profiles to all brokers with related local publishers. The filtering is accomplished at the local broker of an event's publisher.

Advantages: Since events are not forwarded, high event frequencies are not a problem using this approach. The filtering is directly accomplished if events are delivered to the ENS.

Disadvantages: Problematic is the spreading of profiles. Fortunately in facility management we do not expect heavy load because profile reconfigurations appear very seldom. However, in brokers lots of profiles have to be stored, which increases memory usage.

An example for filtering as close as possible to the publishers is profile forwarding [6].

3.3.3 Filtering at Fixed Brokers

That approach uses specialised brokers that are known in the whole network and are used for filtering by all local brokers. A common idea is to use brokers specialised in event types.

Advantages: With the chance to choose filtering brokers we can use the most powerful brokers for filtering. If a broker is heavily loaded it can select supporting brokers, which is transparent to the residual of the network.

Disadvantages: Both, profiles and events have to be forwarded to the filtering broker. More network load is produced and according to the network structure the paths are longer or shorter. Under circumstances the memory usage can become a problem: If we choose the filtering broker according to event types in case of strong usage of special types the load of the filtering broker grows.

The systems Scribe [23] and Hermes [21] use filtering according to event types.

3.4 Memory Strategy

Using distributed qualified filtering and the optimisations for minimising redundancies (covering, Sect. 2.1.4) results in two memory strategies for storing profiles. Using preventive storing the brokers try to maximise the profiles stored even if a filtering of some profiles is not necessary to find the neighbours to notify. In optimistic storing it is tried to minimise profile information in brokers.

3.4.1 Preventive Storing

This strategy aims at storing as many profiles as possible. If a covered profile from a neighbour is subscribed it is stored but it is not filtered. Then in case of unsubscriptions the filtering of the covered profile can occur automatically.

Advantages: Unsubscriptions do not produce high network load. The processing of unsubscriptions is sufficient since all covered profiles are known.

Disadvantages: For storing plenty of profiles we need plenty of memory. Temporarily redundant profiles are stored in several brokers.

3.4.2 Optimistic Storing

Optimistic storing aims at filtering with as minimal profile information as possible. So we store only profiles that are really filtered. That means that in non-local brokers we do not store covered profiles.

Advantages: Using this approach results in less memory consumption.

Disadvantages: Unsubscriptions of profiles increase the network load, since besides the unsubscriptions all formerly covered profiles have to be distributed in parts of the network.

Profile forwarding [6] and Hermes [21] can use both of the described strategies. Here, the available literature does not give sufficient detailed information about the actual implementation details.

3.5 Combination of Categorisation

Based on our categorisation scheme we now combine the different categories to filter approaches. Subsequently, these approaches are evaluated according to the criteria network traffic, memory usage, efficiency and scalability.

Table 1 shows the different combinations of the 4 categories. Column 1 contains the name, Column 2 the filter location, Column 3 the 2 possible spreadings of filter complexity and Column 4 the communication with the subscribers. The evaluation can be found in Column 5.

In Table 1 we can find 3 types of algorithms: event forwarding (EF), profile forwarding (PF) and rendezvous nodes (RN). These types result out of the filter location in Column 2. Except event forwarding we can find several subtypes of the algorithms. In the next sections we describe and evaluate these algorithms.

3.5.1 Event Forwarding

Using event forwarding the filter location is as close to the subscribers as possible. Profiles are subscribed at the local brokers and not forwarded, instead the events are forwarded to all brokers. Since each broker only filters for local subscribers we use exclusive filtering and direct communication. We also cannot use strategies for minimising the redundancies among profiles, since covering cannot be realised for local subscribers (the local broker has to notify about all profiles). Event forwarding can be evaluated as follows:

Network traffic: In case of high event frequencies we expect a high network traffic. Generally because of the flooding we derive the maximal network load.

Memory usage: Each broker only filters for local subscribers, so the memory usage is low. By limiting the number of subscribers per broker (and their profiles) we can directly control the memory usage according to the existing resources.

Algorithm	Filter Location			Spreading of Compl. (Memory Strategy)			Communication			Evaluation			
	Subscribers	Publishers	Arbitrary	Exclusive	Distributed Qualified		Direct	Forwarding	Transparent	Network Traffic	Memory Usage	Efficiency	Scalability
					Preventive Storing	Optimistic Storing							
EF	$\times$			$\times$			$\times$			– –	+ +	+ –	–
PF_1		$\times$		$\times$			$\times$			+	– –	+ –	– –
PF_2		$\times$		$\times$				$\times$		+ –	– –	+ –	– –
PF_3		$\times$			$\times$		$\times$			+	–	+	–
PF_4		$\times$			$\times$			$\times$		+ –	–	+	–
PF_5		$\times$			$\times$				$\times$	+	+ –	++	+ –
PF_6		$\times$				$\times$	$\times$			+	+ –	+	+ –
PF_7		$\times$				$\times$		$\times$		+ –	+ –	+	+ –
PF_8		$\times$				$\times$			$\times$	+	+	++	+
RN_1			$\times$	$\times$			$\times$			–	– –	–	– –
RN_2			$\times$	$\times$				$\times$		–	– –	–	– –
RN_3			$\times$		$\times$		$\times$			+ –	– –	+ –	–
RN_4			$\times$		$\times$			$\times$		–	– –	+ –	–
RN_5			$\times$		$\times$				$\times$	+ –	–	+	+ –
RN_6			$\times$			$\times$	$\times$			+ –	–	+ –	–
RN_7			$\times$			$\times$		$\times$		–	–	–	–
RN_8			$\times$			$\times$			$\times$	+ –	+	+	+ –

Table 1: Classification and theoretical evaluation of Distributed Filter Algorithms (EF - event forwarding, PF_x - profile forwarding, RN_x - rendezvous nodes, $\times$ = feature is supported, Evaluation: – – to ++ = poor to excellent results)

Efficiency: The communication overhead for forwarding all events to all brokers results in an initial efficiency decrease. Additional decrease is caused by the filtering of each event in each broker. The filter complexity is not spread among brokers. However, the simplicity of the algorithm is an advantage.

Scalability: Simply increasing the number of brokers when using the same total number of profiles does not lead to more processed events per time unit. Instead the network is more loaded, which is limited to a maximal capacity.

3.5.2 Profile Forwarding

In all subtypes of profile forwarding (PF_x) we filter as close as possible to the publishers. So profiles are forwarded to several brokers, which are able to filter them.

In subtypes PF_1 and PF_2 all brokers can filter all profiles. PF_1 directly communicates in case of notifications, PF_2 forwards via the network of brokers. Transparent subscribers contradict exclusive filtering.

PF_3 to PF_8 use a division of work. So profiles are forwarded to neighbour brokers and optimisations can be used. PF_3 und PF_6 forward profiles with its real subscribers as subscribers. In case of notifications the filtering broker directly contacts the subscribers, which have to perform a post-filtering (because of the optimisations). The optimisations are used in a per subscriber manner. The difference in PF_4 and PF_7 is that the network of brokers is used to forward notifications.

PF_5 and PF_8 use brokers as proxies to forward profiles. So each broker only filters for neighbours (clients or brokers). Optimisations can be used for different real subscribers.

PF_3 to PF_5 store profiles that are not filtered, which is advantageous in case of unsubscriptions. PF_6 to PF_8 only store currently filtered profiles, which then causes more network traffic.

Network traffic: In PF_2 under circumstances several notifications about one event are forwarded along the path of the same brokers, which increases the network load. In PF_1, PF_3 and PF_6 this characteristic is omitted. PF_4 and PF_7 have the same problem as PF_2. PF_5 and PF_8 notify a neighbour broker exactly once if one of its neighbours is a local broker with matching profiles. Therefore, the subtypes PF_5 and PF_8 of profile flooding create less network traffic.

Memory usage: In PF_1 and PF_2 we have very high memory usage since all brokers filter all profiles of all subscribers. PF_3 to PF_5 require a bit more memory than PF_6 to PF_8 because more profiles are stored. PF_5 and PF_8 decrease memory utilisation by using the covering among different subscribers. PF_3, PF_4, PF_6 and PF_7 only use this in a per subscriber manner.

Efficiency: In PF_1 and PF_2 efficiency is only moderate since a broker has to filter all profiles. PF_3, PF_4, PF_6 and PF_7 use optimisations per subscribers to improve efficiency (less filter operations). PF_5 and PF_8 filter exactly that profiles that are necessary to notify the neighbour brokers. This leads to good efficiency.

Scalability: Scalability in PF_1 and PF_2 is very bad, since adding brokers does not decrease the filter complexity. PF_6 to PF_8 achieve better scalability than PF_3 to PF_5 using the same communication strategy because less brokers are involved to manage the same numbers of profiles. PF_5 and PF_8 use optimisations among different profiles, which leads to less redundancies in the brokers and for this reason to better scalability.

3.5.3 Rendezvous Nodes

The subtypes of rendezvous nodes (RN_x) filter in specialised nodes, e.g. according the event types. When using RN_1 or RN_2 only the specialised rendezvous nodes filter a subset of profiles. Notifications are delivered directly (RN_1) or via the network of brokers (RN_2).

In distributed qualified filtering (RN_3 to RN_8) both memory strategies and optimisations are possible. All brokers on the path from the subscriber to the rendezvous node store profiles and filter them. In RN_5 and RN_8 brokers act as proxies for subscribers. In RN_3 and RN_6 brokers do not act as subscribers. So the notification is done by the filtering broker. Using RN_4 and RN_7 the network of subscribers forwards notifications. The subtypes of rendezvous nodes are evaluated as follows:

Network traffic: RN_1 and RN_2 cause high traffic: All events are forwarded to the rendezvous node and notifications are delivered (in RN_2 even multiple ones across the same path) via the network. This also happens using RN_4 and RN_7. The other subtypes do not send multiple notifications about the same event. The complexity of forwarding events is proportional to the length of the path between a subscriber and the rendezvous node.

Memory usage: Lots of memory is needed by RN_1 and RN_2 since rendezvous nodes store all profiles. RN_3 to RN_5 need a bit more memory than RN_6 to RN_8 when using the same spreading of filter complexity. RN_5 and RN_8 show less memory usage because of the computation of coverings among subscribers. Instead RN_3, RN_4, RN_6 and RN_7 only use covering per subscriber.

Efficiency: RN_1 and RN_2 only have moderate efficiency since all profiles are filtered by rendezvous nodes. RN_3, RN_4, RN_6 and RN_7 use optimisations in a per subscriber manner, which improves efficiency. The best efficiency is obtained in RN_5 and RN_8. Altogether the efficiency is not as good as in profile forwarding because rendezvous nodes are always heavily loaded.

Scalability: In RN_1 and RN_2 we can expect very bad scalability since only the rendezvous nodes have more filtering overhead in case of increasing profile numbers. RN_3, RN_4, RN_6 and RN_7 show bad scalability because more profiles have to be filtered by the same brokers and the covering is only used per subscriber. RN_5 and RN_8 lead to a bit better scalability since redundancies are minimised among subscribers. Even in case of adding brokers the rendezvous nodes remain as bottleneck since they have to filter each event according to their specialisation.

3.6 Summary

In this section, we presented a classification and evaluation of distributed filtering strategies. We introduced the following categories:

1. The communication with subscribers can be realised directly, by forwarding via the network of brokers or by using transparent subscriptions and proxies (Sect. 3.1).

2. The filtering complexity can be distributed in the whole network of brokers or it can be done exclusively by specialised brokers (Sect. 3.2).

3. The filtering location can be chosen as close as possible to the subscribers, as close as possible to the publishers or at arbitrary fixed brokers (Sect. 3.3).

4. The memory strategy when distributing the filter complexity is selectable as preventive or optimistic (Sect. 3.4).

In Sect. 3.5 we combined the previous categories to 17 distributed filtering strategies of 3 kinds. Then we evaluated all strategies according to network traffic, memory usage, efficiency and scalability.

From the 3 kinds of algorithms we can now easily choose the best algorithm according to our 4 evaluation criteria for an practical implementation and evaluation: EF, PF_8 and RN_8. In the remainder of the paper, we will refer to the three subtypes by the names of the approaches they represent: EF will be referred to as *event forwarding*, PF_8 as *profile forwarding* and RN_8 as *rendesvouz nodes*.

With a practical evaluation we can assess these 3 algorithms in more detail. In the following sections we firstly describe the realisation of the algorithms and afterwards their extensive practical evaluation.

4 System Design and Architecture

In the previous section, we selected three distributed filter algorithms for an implementation. In this section, we describe their implementation in our prototype called DAS (distributed alerting service). We explain the design and the architecture of DAS. We give details about the initial centralised filter algorithm, considerations about the network, and alternatives to implement the covering optimisation and the distributed architecture.

4.1 Centralised System

In [4], we described a centralised filter algorithm for ENSs (implemented in the prototype PrimAS). It is implemented in Java and builds the foundation for our distributed system DAS. In the following, we first describe the filtering in PrimAS and then explain the extensions, which result in less memory consumption and a dynamic filter structure. Then, we give details about profile definitions and the computation of coverings.

4.1.1 Basis System

The starting point for the filtering algorithm PrimAS is the work of Gough and Smith [9], which proposes a tree-based filter structure. There out of the profiles' predicates a tree is built. Each level of the tree filters 1 attribute. Figure 3(a) shows such a filter tree for 5 profiles with 3 attributes a_3, a_4 and a_5 of type T_4. The last level of the tree are the leafs, where the profiles are recorded. For the filtering of events we follow the outgoing edge with the respective attribute value of an event. After that we evaluate the next node. If we reach a leaf we find the matching profiles. Otherwise there are no matching ones. Since profiles do not have to specify all attributes there are *-edges in the tree. We follow them if no value of the edges matches the event's attribute value.

Our implementation marks the edges with intervals instead of values. So we can easily support more operators (see Sect. 4.1.3). The intervals are realised by an array and each entry describes the half-open interval of the previous (exclusive) and the current entry (inclusive). The pointers to the profiles are stored in another array of the same size. A more detailed explanation of this filter structure can be found in [4].

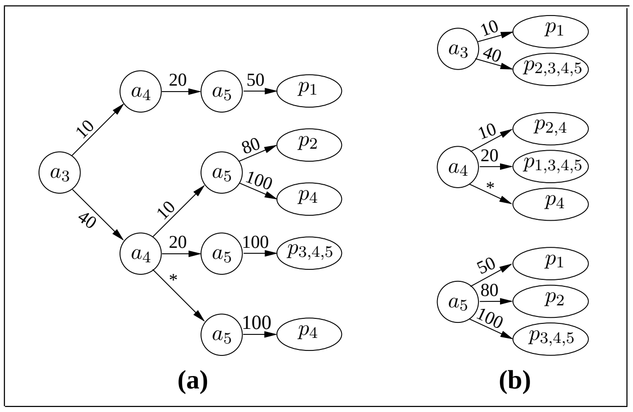


Figure 3: Filtering tree (a) and trees (b) with the 5 profiles $p_1 : (a_3 = 10 \wedge a_4 = 20 \wedge a_5 = 50, T_4)$, $p_2 : (a_3 = 40 \wedge a_4 = 10 \wedge a_5 = 80, T_4)$, $p_3 : (a_3 = 40 \wedge a_4 = 20 \wedge a_5 = 100, T_4)$, $p_4 : (a_3 = 40 \wedge a_5 = 100, T_4)$, $p_5 : (a_3 = 40 \wedge a_4 = 20 \wedge a_5 = 100, T_4)$.

A problem of the tree-based filter structure is its enormous memory consumption. So we developed an extended filter algorithm in [4] with less memory usage. In contrast this algorithm is more time consuming. In this algorithm we do not construct a tree structure. Instead we generate 1 node per attribute. Figure 3(b) shows this filter structure with the 3 attributes of the event type T_4. For simplification we labelled the edges with values instead of intervals. For filtering events we process all nodes of all attributes of the events' types by following their corresponding edges. The resulting profile sets are intersected successively. So the result is the set of profiles matching all attributes. In Fig. 3(b) filtering the event $e_1 : (a_3 = 40, a_4 = 10, a_5 = 80, T_4)$ results in the following profile set: $\{p_2, p_3, p_4, p_5\} \cap \{p_2, p_4\} \cap \{p_2\} = \{p_2\}$.

4.1.2 System Extensions

In DAS we further extend the centralised filter structure from PrimAS. We firstly describe a strategy to minimise the memory consumption. After that we propose an extension to support dynamic changes in our filter structure in DAS.

Bit List. Our minimisation of memory usage is based on the management of pointers to profiles in the edges of the filter structure. We do not store pointers in these edges, instead we use their indices as a bit list (realised as a array of integers).

Generally, if there is a small number of profiles in an outgoing edge it is reasonable to use direct pointers in the leaves. If 3.125% or more of the profiles are stored in a leaf we should use the bit lists, since a pointer consumes 4 bytes of memory and a marked bit in the list only 1 bit. In addition we need an array with pointers to all profiles.

There are also differences when intersecting the profiles in the leaves. In dense leaves (many profiles stored) an intersection using the bit lists is more efficient, since we can use logical operations on the bit level. Otherwise we should prefer direct pointers.

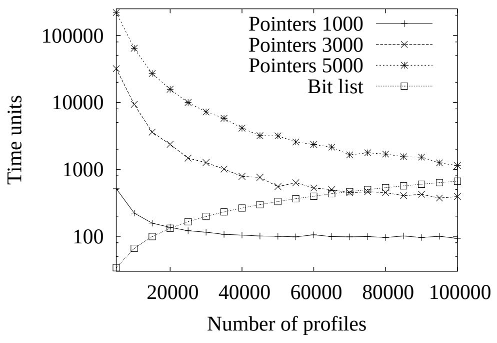


Figure 4: Time for intersections

Figure 4 shows the time for intersecting dependent on the number of profiles. There we are intersecting profile sets of the sizes 1,000, 3,000 and 5,000 out of profiles on the abscissa. If more elements exist in the set the intersection is more costly when using pointers. When using bit lists the intersection time does not depend on the number of profiles in the set (it depends on the total number of profiles). In the figure we see that up to a certain number of profiles in the sets a usage of pointers is less efficient. But when using large numbers of profiles pointers are more efficient than bit lists.

Because of these dependencies of memory usage and efficiency we can configure the management of the profiles in leaves in DAS. A user can decide from what utilisation ratio a bit list should be used. So we can exploit the advantages of both options and decrease the memory consumption and increase the efficiency compared to PrimAS.

Dynamic Filter Structure. When designing a distributed ENS it is essential to have a dynamic filter structure. in this way, one can avoid regular rebuilds, since in a distributed system not all profiles are subscribed at once. In facility management it is beneficial to support subscriptions and unsubscriptions at any time to reconfigure the buildings. So DAS supports this feature. Next to this we can optimise the order of the evaluation of the attributes at any time as describe in [11].

The dynamic filter structure is implemented with the help of dynamic arrays. They are used for the edges of the filtering nodes (with the intervals) and for the edges, both in the profile pointers and the bit lists. At runtime these arrays can increase and decrease. The amount of shrinking or growing and the number of empty entries permitted can be specified by the system's users. So we can optimise DAS according to its current application.

4.1.3 Profile Definition Language

After describing the centralised filter algorithms we now illustrate the profile definition language. It is developed in [13] and in DAS we only need a few of the supported features. For example the profile $p : (a_1 = 5 \wedge a_2 > 10, T_1)$ is defined as follows:

```
    PROFILE(PRI(a1=5,a2>10,TYPE=T1))
```

In the following we describe the attribute's domains and the supported operators.

Domains. In the application of facility management the domains used are exclusively of numerical types [14]. Besides we can define enumerations in DAS. The description of domains in DAS happens as follows.

Integers: An interval $[a, b]$ is sufficient to define integers in DAS.

Fixed Point Numbers: An interval $[a, b]$ and the number of digits after the decimal n has to be defined.

Enumerations: For definition you have to specify some elements (as strings) and their order.

By using these definition we only obtain limited and ordered domains. They are an assumption of the filter algorithm previously described in Sect. 4.1.1.

In the profile's predicates we support 5 logical operators to evaluate the attribute values:

Equality $(=)$**:** An event's attribute value has to exactly match the value of the profile's predicate. Technical: $(a, w_x) \succ (a, =, w_y)$ iff $w_x = w_y$.

Greater $(>)$**:** An event's attribute value has to be greater than the value of the profile's predicate. Since the usage of limited domains we can also express greater or equal. Technical: $(a, w_x) \succ (a, >, w_y)$ iff $w_x > w_y$.

Less $(<)$**:** An event's attribute value has to be less than the value of the profile's predicate. Since the usage of limited domains we can also express less or equal. Technical: $(a, w_x) \succ (a, <, w_y)$ iff $w_x < w_y$.

Included$(\in)$**:** An event's attribute value has to be included in the set specified by the profile's predicate. Technical: $(a, w_x) \succ (a, \in, W_y)$ iff $w_x \in W_y$.

Between$(\dashv)$**:** An event's attribute value has to be included in the interval specified by the profile's predicate. Technical: $(a, w_x) \succ (a, \dashv, (w_y, w_z))$ iff $w_z \geq w_x \geq w_y$.

4.1.4 Computation of Covering

We propose two strategies for the computation of coverings in DAS: Profile-based, which directly compares profile definitions and interval-based, which uses the attributes' domains and our filter structure previously described.

Profile-based computation: Since a fixed number of operators can be used in profile definitions, we can apply case differentiations according to the operators used in the profiles we compare. So when both operators are equality tests we simply need to compare the values specified in profiles. Since our limited space we neglect the presentation of the different cases.

Interval-based computation: By analysing the profiles in the leaves of our filter structure we can also compute coverings. For example if a predicate contains the greater than operator we know that all profiles that only occur in subsequent edges are covered. Analogously, we can compute profiles covered by other operators. By intersecting the results from all attributes we can derive coverings of profiles.

4.2 Network

We now describe our assumptions and decisions concerning the network topology and network/transportation protocol. We generally assume stable and reliable network connections, so we do not need to integrate extensive error handling mechanisms.

4.2.1 Network Topology

The distributed filter algorithms chosen in Sect. 3 are working with two topologies: hierarchical networks and point-to-point networks. Mostly such topologies are not based on physical connections, so we have to use an overlay network instead. This overlay network abstracts from the physical connections and medium access control protocols. This means our system is located in the application layer and we can develop more complex and sophisticated filter algorithms.

For the overlay network we choose a connected, acyclic graph structure (free tree). Since it is a logical network the assumption of acyclicity is easy to realise. Furthermore we can implement simpler and more efficient algorithms, since many problems are displaced to lower layers in the protocol stack (e.g. prevention of circulating and duplicated messages). The distinct connections between each pair of nodes are no problems in our overlay network. Since it is only the logical topology lower layers will find a path between 2 nodes as long as there is no partition in the physical layer. So cyclic logical topologies behave exactly the same in case of link errors.

4.2.2 Network Protocol

The implementation of DAS uses the popular TCP/IP protocol family [26, 27, 29]. So all communication is based on TCP/IP sockets. But this reference implementation in DAS is arbitrarily exchangeable, as long as the following conditions hold:

- Error free data transmission
- Retaining the order of messages (FIFO)

4.3 Architecture

After describing the central filter component and the network properties we now illustrate the architecture of our publish/subscribe prototype DAS. We make use of 3 different parts that are presented in the following: brokers, subscribers and publishers. DAS is implemented in Java. So it offers object representations of all constituents, such as events, profiles, domains, operators, attributes, subscribers, publishers and brokers.

4.3.1 Brokers

Brokers are server components and much more complex than publishers and subscribers as clients. Figure 5 illustrates their architecture, which consists of the following parts:

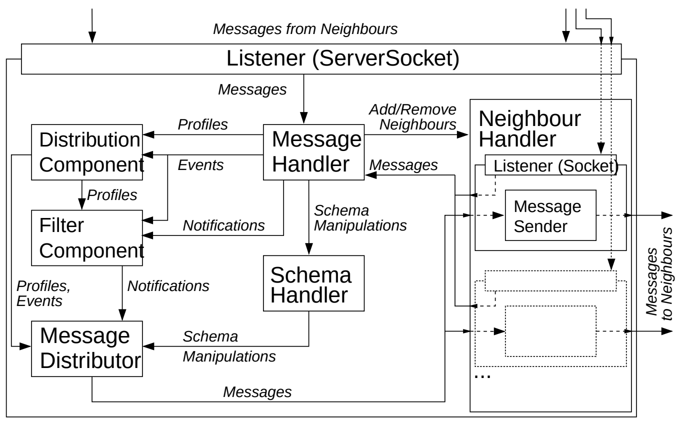


Figure 5: Architecture of the brokers

Listener: Connection requests from brokers and clients are handled by the Listener in its own Thread. Messages are forwarded to the Message Handler.

Neighbour Handler: Neighbours (brokers and clients) are managed by the Neighbour Handler, which allows to access them.

Neighbour: Each Neighbour represents a neighbour in network. A listener waits for messages, which are forwarded to the Message Handler. Via the Message Sender we can forward messages to the represented neighbour in the network.

Message Handler: Incoming messages are handled by this component. Such messages are connection and disconnection requests, subscriptions and unsubscriptions of profiles, events, notifications and schema manipulations.

Schema Handler: Event types and domains are managed by this handler. Manipulations from neighbours are forwarded to the Schema Handler by the Message Handler.

Distribution Component: This component realises the distribution of profiles and events in network. According to the algorithm chosen (EF, PF or RN) profiles and events are processed by adding to the Filter Component, filtering or forwarding to neighbours.

Filter Component: Here the real filtering takes place. Events are filtered, profiles are added to or removed from the filter structure and also notifications are processed by post-filtering. Basis is the centralised algorithm presented in Sect. 4.1.

Message Distributor: Messages are forwarded to neighbours via this component.

The main class of the brokers is an abstract class Broker, which realises Listener, Neighbour Handler and Message Distributor. For our 3 algorithms the specialisations EventForwardingBroker, ProfileForwardingBroker and RendezvousBroker exist. The component Neighbour is represented by an abstract class Neighbour with its specialisations NeighbourSubscriber, NeighbourPublisher and NeighbourBroker. The Schema Handler consists among others of the classes EventTypePool and DomainPool.

4.3.2 Subscribers and Publishers

Subscribers and publishers as clients are not as complicated as the server components. Figure 6 illustrates the subscriber's architecture. We can find the following parts:

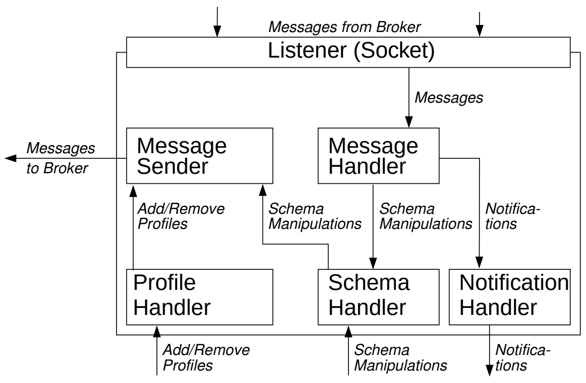


Figure 6: Architecture of the subscribers

Listener: Messages from the local broker are handled by the Listener, which runs in an own Thread. These messages are forwarded to the Message Handler.

Message Handler: This component manages all incoming messages. Schema information are forwarded to the Schema Handler, notifications are forwarded to the Notification Handler.

Notification Handler: Here the real notification takes place. That means notifications from the local broker are assigned to notifications specified by the subscriber and executed.

Message Sender: This component realises the delegation of messages to the local broker. Therefore messages are firstly coded and secondly sent.

Profile Handler: This element accepts subscriptions and unsubscriptions from the user (as objects), codes them as messages and forwards them to the Message Sender.

Schema Handler: Analogous to the broker event types and domains are handled here. Manipulations via brokers have been forwarded via the Message Handler. Manipulations from the user are announced via the Message Sender to DAS.

Subscribers can be implemented via deriving the class DistributedSubscriber. This class offers methods for subscribing and unsubscribing profiles, which can be specified by a profile definition language developed by us (Sect. 4.1.3).

The publishers are the simplest components and not shown here since limited space. The only new part is the Event Handler, which accepts events from the user as objects and forwards them via the Message Sender to the publisher's local broker as messages. Implementation of publishers is done via deriving the class DistributedPublisher. Here a method for the publication of events is offered, which can be specified via a simple text description.

4.3.3 Configuration

The system's configuration is easy and flexible. So we can choose a distributed filter algorithm, specify the network addresses of brokers, declare the overlay network and configure the rendezvous nodes (if using them). Brokers are started with a parameter specifying their identification. Then they try to connect to each other to build the overlay network. After that DAS is ready to handle subscriptions and events according to the filter algorithm chosen.

4.4 Protocols

After presenting the general system architecture we now briefly describe the protocols of the 3 filter algorithms (EF, PF, RN) supported by DAS.

4.4.1 Event Forwarding

This is the simplest algorithm, since events are flooded and brokers only filter for local subscribers. Subscriptions are added to the broker's filter structure and in case of unsubscriptions the profiles are removed. Events are flooded to all neighbour brokers except the sender and filtered. When events match profiles their subscribers are notified.

4.4.2 Profile Forwarding

Profile forwarding uses coverings among profiles, so subscribing a profile p_x and unsubscribing a profile p_y are a bit more complicated. If p_x is registered by a broker we can remove all profiles covered by p_x registered by this broker. If altogether no covered profiles exist, we register p_x at all neighbour brokers except the sender as an own profile. If all covered profiles are registered by the same neighbour broker we register p_x at this broker as own profile. Finally we add p_x to the filter structure. When unsubscribing p_y we register all profiles covered by p_y at all neighbour brokers except the sender, which have not been registered by the respective broker. We also send the unsubscription to all neighbours except its sender. Finally we remove p_y from the filter structure.

Published events are filtered and all subscribers of matching profiles are notified. If the subscriber is a broker we notify it exactly once about each event even if several profiles match. When notifications arrive at brokers we filter the contained event and further notify all subscribers except the sender. Brokers are again notified exactly once.

4.4.3 Rendezvous Nodes

Also rendezvous nodes use coverings among profiles, which results in more complicated processes when subscribing p_x or unsubscribing p_y. If p_x is registered by a broker we remove all covered profiles registered by the sender. If we are not the rendezvous node for the profile's type we send p_x into the direction of its rendezvous node. Finally we add p_x to the filter structure. Unsubscribing p_y works as follows: If we are not the rendezvous node and covered profiles exist, we send the covered profiles into the direction of the rendezvous node. In all cases we then send the unsubscription into the rendezvous node's direction. After that we remove p_y from the filter structure.

Events are filtered and neighbour brokers (except the sender) are notified exactly once in case of matching. Then we forward an event into the direction of the rendezvous node. Notifications are processed by filtering the contained event. We notify all subscribers excluding the sender. Brokers are again notified not more than once per event.

Rendezvous nodes are specified when configuring the network. If brokers connect to each other to build up the overlay network they also exchange information about known rendezvous nodes. Therefore it is sufficient to know via which neighbour a rendezvous node is reachable and for which event types the broker is the rendezvous node itself. With this solution we have a simple and scalable managing of rendezvous node related information.

4.5 Summary

In this section, we presented the design and the architecture of DAS. In Sect. 4.1 we described the centralised filter algorithm PrimAS, its extensions and adaptations (less memory usage, dynamic filter structure) to support a distributed filtering. Then we described the profile definition language, the domains for attributes and the supported operators. The network topology and the network protocols have been discussed in

Sect. 4.2. We use an acyclic overlay network in combination with TCP/IP. Finally we described the architecture of the brokers, subscribers and publishers in Sect. 4.3. There we also presented the protocols of the 3 filter algorithms (event forwarding, profile forwarding, rendezvous nodes) supported by DAS.

In the next section, we describe the results of our analysis of these three filter algorithms. We discuss details about the performed experiments and their results.

5 Experiments

After describing the theoretical foundations of publish/subscribe and its implementation in DAS we now evaluate our system. Therefore we use our prototype in a real setting in a LAN with 100 mbps bandwidth, machines with 1GHz and 256 MB main memory running under Linux. We especially want to evaluate the influence of different system parameters, namely portion of matching profiles, portion of matching events, number of brokers, portion of coverings, number of event types, locality of profiles and events and number of profiles. Our analysis relates to the following units of measurement:

Filter efficiency: The number of processable events per second is described by the (filter) efficiency. So efficiency states which event frequencies can be processed without congestion. Our computation of this value is done as follows. Firstly we send all events from the publishers to their local brokers without processing them. Lets say this done at time t_1. Then we do the filtering by all brokers. If all events are processed by all brokers we have a time stamp t_2. Then we compute the frequency by the number of published events divided by $t_2 - t_1$. So we derive the time spent by the brokers and not by the LAN to send events and notifications.

Network load per event: Out of the network load we can derive the amount of forwarding of events. It is computed by summing up the number of received data by all brokers divided by the number of published events while summing up.

Duplication of profiles: The average number of brokers a profile is registered to is described by the duplication of profiles. A value of 1.0 is caused by event forwarding. The value of 2.0 states that each profile is registered by 2 brokers in average. The system is influenced by the duplication, since more memory is needed to store the same number of profiles. This memory consumption results in page swaps and less efficiency. Duplication is computed by dividing the number of totally registered profiles by the number of profiles registered by clients.

We evaluated the standard deviation of our results by subscribing a large number of profiles and then publishing $450,000$ events. We repeated this experiment 40 times, which resulted in a standard deviation of 0.73% for efficiency. So we decided to repeat the following experiments only 3 times, since each measure alone is quite stable. This is because what we call 1 measure is in reality the average of $450,000$ runs in the previously described experiment. Duplication of profiles and network load per event do not need a statistical analysis, since they are independent from external influences.

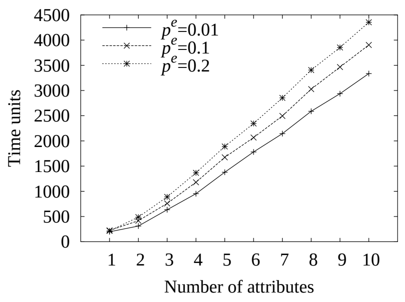


Figure 7: Filter time depended on the number of attributes

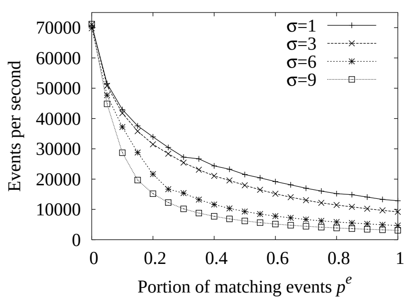
We distinguish between the portion of matching events and profiles. The portion of matching events p^e describes how many events have one or several matching profiles. It is computed by the number of events with matching profiles divided by the number of events totally published. The portion of matching profiles p^p describes how many profiles are notified and is computed by the number of profiles notified divided by the number of events published. We further define the utilisation of events σ by $\sigma = \frac{p^p}{p^e}$. The utilisation σ states how many profiles are notified by a matching event in average.

In the following experiments we only use event types with one attribute. But we can easily derive the behaviour of our algorithms in cases of more attributes. This can be seen in Fig. 7, which shows the filter time for the filtering of $100,000$ events against $10,000$ profiles with different numbers of attributes and values of p^e ($p^e = p^p$, since only unique profiles are used). In this figure, we assume that non-matching events are recognised after the evaluation of half of the type's attributes in average (mean value of recognition after each attribute). Since our algorithm can minimise the number of attributes evaluated to recognise non-matching events [11], DAS behaves even better in real scenarios.

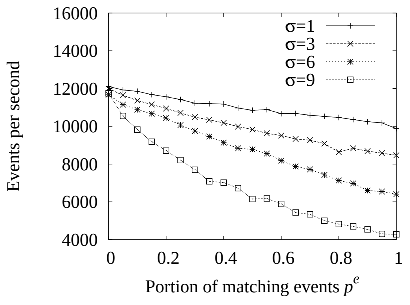
Another assumption we make is that only one publisher and one subscriber is connected to each broker. In real scenarios we rather expect more clients with individually less profiles and events, but summed up with the same quantity. But we can generalise our results, since the number of profiles and events are responsible for efficiency and scalability. For example more clients increase the costs for synchronisation, but this can be transferred to proxies. These proxies handle connections to clients and are the only instances communicating with brokers. So our assumption holds. We also state a unique distribution of profiles and events if not described otherwise. In the following we describe our performance measurements in detail.

5.1 Influence of Matching Events and Profiles

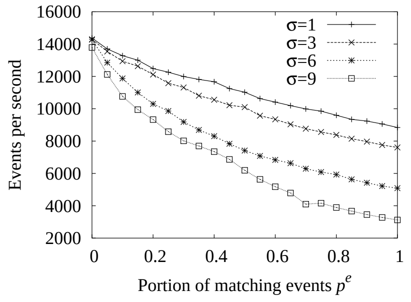
This section analyses the influence of p^e and p^p on efficiency and network load. We do not consider duplication of profiles, since the profile definitions are not changed during the experiments. We use 4 brokers connected as a linear bus. The rendezvous node is located at a central broker. Each broker manages $50,000$ local profiles. We also analyse different values of σ.



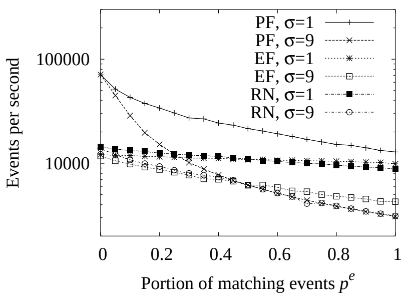
(a) Profile forwarding



(b) Event forwarding



(c) Rendezvous nodes



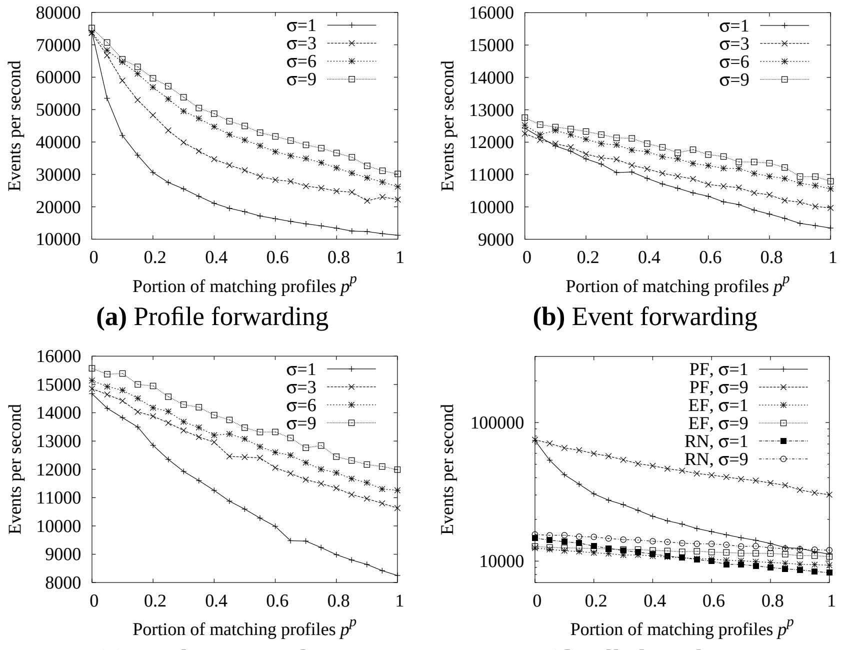
(d) All algorithms

Figure 8: Efficiency of distributed filter algorithms dependent on the portion of matching events p^e

With increasing p^e and p^p we expect less efficiency using each of the algorithms. Especially with small p^p and p^e PF should show better efficiency results than the other algorithms. The network load should be smallest in PF, followed by RN and EF. For EF we expect the maximum network load regardless of p^p and p^e.

Figure 8 shows the efficiency dependent on the portion of matching events p^e. Consider the different scalings and the logarithmic ordinate in Fig. 8(d) when analysing the results. PF (Fig. 8(a)) is very efficient in case of small p^e. With increasing p^e we can see a strong decline in efficiency because of the costly notifications and the post-filtering. EF (Fig. 8(b)) shows less changing with increasing p^e, since the absence of post-filtering. Here only the number of notifications increases, which is resulting in a linear efficiency decrease. RN (Fig. 8(c)) are more influenced than EF, but less influenced than PF in case of increasing p^e. Reasons are both post-filtering (which always occurs since forwarding to the rendezvous node) and more notifications. Figure 8(d) shows the 3 algorithms in comparison.

Figure 9 shows the influence of increasing p^p (also consider different scalings and the logarithmic ordinate in Fig. 9(d)). Here PF (Fig. 9(a)) shows the best efficiency



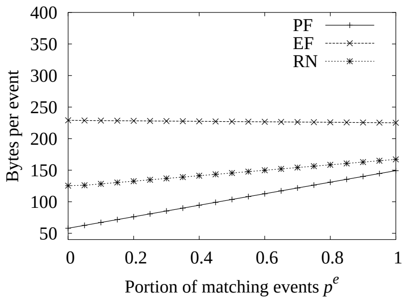
(a) Profile forwarding **(b)** Event forwarding

(c) Rendezvous nodes **(d)** All algorithms

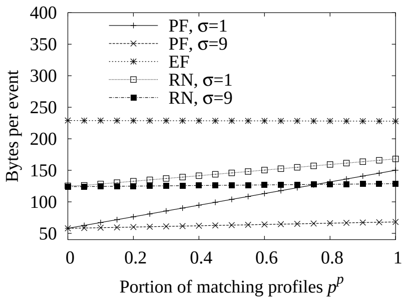
Figure 9: Efficiency of distributed filter algorithms dependent on the portion of matching events p^p

again. But with increasing σ the efficiency becomes better, since less post-filtering is needed and the number of notifications stays constant. Again EF (Fig. 9(b)) shows less changing. This is the result of flooding events in all cases. The improvement can be explained with the earlier reject of non-matching events in case of increasing σ. RN (Fig. 9(c)) lie between EF and PF because of the same reason described above (forwarding to rendezvous nodes in all cases). A comparison of the 3 algorithms is presented in Fig. 9(d).

The network load is shown in Fig. 10. With constant p^e the utilisation of events σ does not influence the network load (Fig. 10(a)). In all cases the same number of events is distributed in the network. EF shows the highest load since the flooding of all events. RN forward all events to the rendezvous node implying less network load. The least load shows PF, since only matching events are forwarded. When using constant p^p the network load is influenced by σ (except using EF with the flooding of events in all cases). Increasing σ results in decreasing network load, since less events notify the same number of profiles.



(a) Matching events p^e



(b) Matching profiles p^p

Figure 10: Network load dependent on the portion of matching events p^e and matching profiles p^p

5.2 Influence of Number of Brokers

This section analyses the influence of the number of brokers on efficiency, duplication of profiles, network load and parallel efficiency e. In the experiments we use a special network of brokers that is illustrated in Fig. 11. We use 1 event type and our network size varies from 1 to 9. Broker B_2 acts as rendezvous node (except when using only 1 broker). Altogether $200,000$ unique profiles are subscribed at DAS. We expect im-

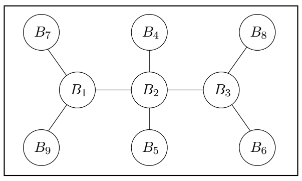
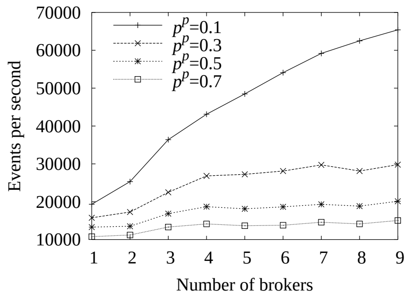


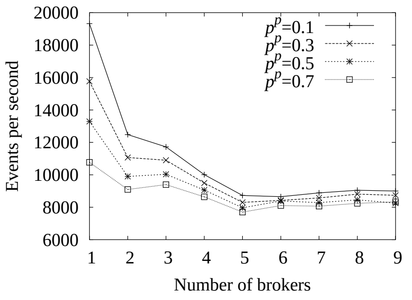
Figure 11: Used network topology for brokers

proving efficiency using more brokers in case of PF and an understated improvement when using RN. EF should not result in better efficiency. The network load is expected to increase in all 3 algorithms, most in PF, followed by RN and EF. For the profile duplication we expect the opposite: EF duplicates no profiles, PF all profiles and RN is a compromise between both of them. PF should show the best parallel efficiency and EF the worst one.

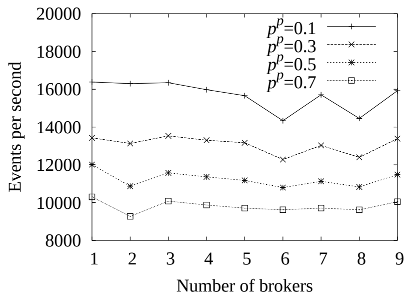
Figure 12 shows the filter and the parallel efficiency of the algorithms with different scalings. The behaviour of PF is shown in Fig. 12(a). With $p^p = 0.1$ there is a high increase in efficiency when adding more brokers. Increased values of p^p lower the filter efficiency. Adding brokers in these cases only results in less efficiency improvements (the main load is caused by notifications). EF shows decreasing filter efficiency when adding more brokers (Fig. 12(b)), which can be explained with the increased commu-



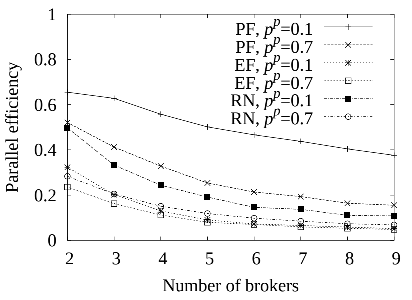
(a) Efficiency using profile forwarding



(b) Efficiency using event forwarding



(c) Efficiency using rendezvous nodes

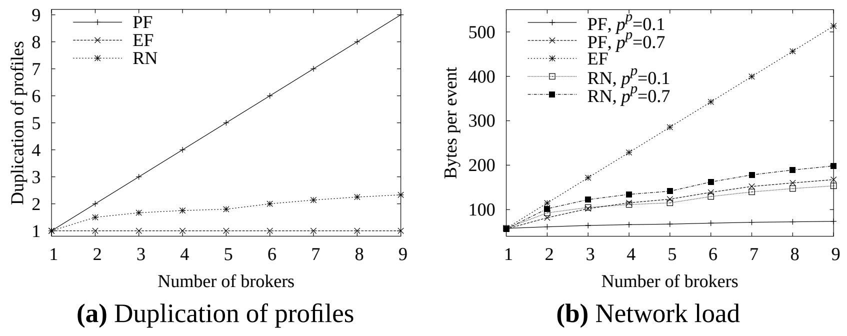


(d) Parallel efficiency

Figure 12: Filter efficiency and parallel efficiency dependent on number of brokers

nication overhead. Using more brokers the influence of p^p is small, since the additional overhead produced by notifications is less compared to the other communication complexity. Using 5 brokers the efficiency is the smallest, since broker B_2, which is the system's bottleneck, is mostly loaded. RN's efficiency (Fig. 12(c)) is nearly unchanged when adding brokers. Here the systems bottleneck is the rendezvous node, which performs the same amount of filtering steps regardless of the network size. Partially the filter efficiency decreases when adding brokers. The explanation is the asymmetrical network of brokers appearing in these cases, since some brokers are more loaded than others because of the uniquely distributed events and profiles.

The parallel efficiency is shown in Fig. 12(d). The best results can be achieved using PF because of its efficient load distribution. Altogether these results are disappointing, but implied by the high communication overhead between the brokers themselves. The duplication of profiles (Fig. 13(a)) increases linearly when using PF. Since the profiles are unique, each broker stores all profiles. EF shows constant values of 1.0. RN lie between PF and EF with values less than 2.5. As expected the network load (Fig. 13(b)) behaves contrary to the profile duplication. EF shows a constant increase. PF only distributes matching events with the result of low network load. With high values of p^p the difference in network load between RN and PF is low, since the forwarding to the rendezvous node is little additional expense (events have to be



(a) Duplication of profiles **(b)** Network load

Figure 13: Duplication of profiles and network load dependent on the number of brokers

forwarded to the rendezvous node anyway).

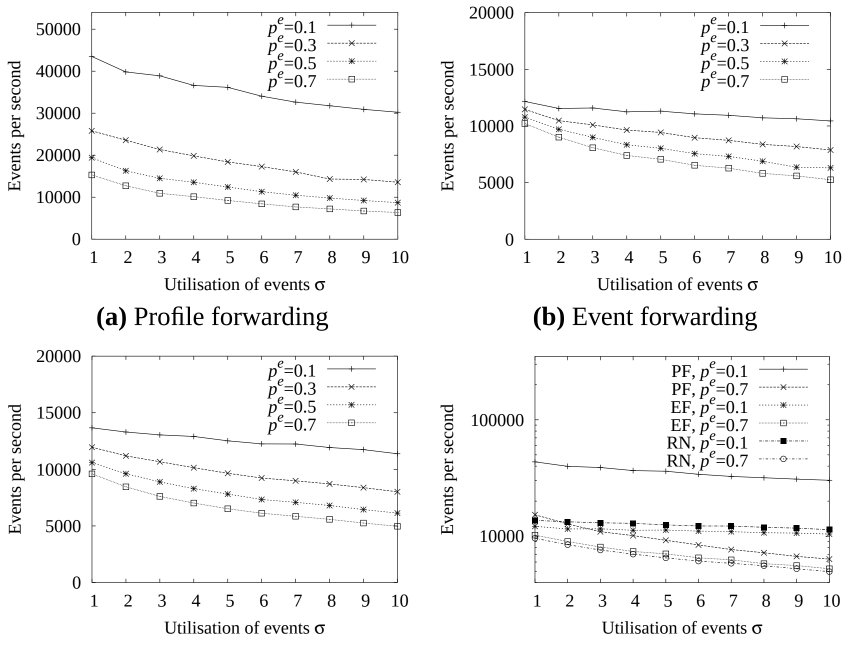
5.3 Influence of Covering

In this section we show the influence of coverings. We are using only equality operators, so the utilisation of events σ is equivalent to coverings. This means that $\sigma = 5$ stands for 5 covered profiles per profile. Coverings only appear between profiles of 1 local broker. We use the same network of brokers as described in Sect. 5.1. We use 1 event type and $200,000$ profiles. We analyse efficiency, duplication of profiles and network load.

We expect decreasing efficiency with increasing coverings using constant p^e (more load because of more notifications). Using constant p^p the system should behave oppositely (less forwarding and filtering steps). The duplication of profiles is expected to decrease when using more coverings (since taking advantage of them). The network load should stay unchanged with constant p^e and changing σ. With constant p^p and increasing σ the network load is expected to decrease, since less events are forwarded (except using EF).

Figure 14 shows the efficiency dependent on the portion of matching events p^e. Consider the different scalings and the logarithmic ordinate in Fig. 14(d). All 3 algorithms show decreasing efficiency using more coverings, since more notifications are generated. With a high value of p^e the differences among the algorithms are marginal (Fig. 14(d)), since nearly all events have to be flooded. With small p^e PF (Fig. 14(a)) is by far the most efficient algorithm. As more as p^e grows as less the efficiency decreases (proportion of complexity of notifications to all-over complexity). Using EF (Fig. 14(b)) and RN (Fig. 14(c)) the decrease in efficiency is lower.

Figure 15 shows the efficiency with a changing portion of matching profiles p^p (again consider different scalings and a logarithmic ordinate in Fig. 15(d)). With increasing σ and constant p^p we observe better efficiency. The reason is that the same number of notifications are obtained by the distribution and filtering of less events. PF



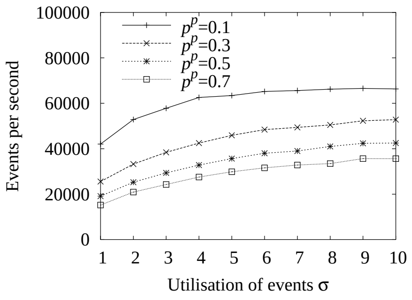
(a) Profile forwarding **(b)** Event forwarding

(c) Rendezvous nodes **(d)** All algorithms

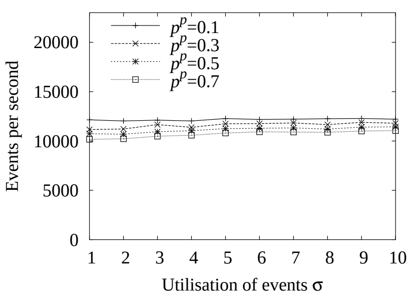
Figure 14: Efficiency dependent on the utilisation of events σ with various portions of matching events p^e

(Fig. 15(a)) shows the best improvements in efficiency, since no events without matching profiles are forwarded. The distance to the other algorithms grows with increasing σ, which is contrary to the case described above. Using RN (Fig. 15(c)) we can also observe an improvement in efficiency. It has a lower dimension, since events are always forwarded to the rendezvous node. EF (Fig. 15(b)) is independent from σ because events are forwarded to all brokers in all cases.

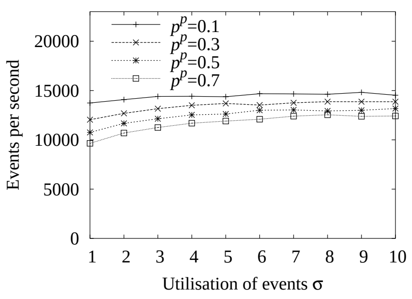
The network load stays constant with unmodified p^e (Fig. 16(a)), since always the same number of events is forwarded (in spite of that there are more notifications). As expected high values of p^e increase the network load and EF shows the highest load. Using constant p^p (Fig. 16(b)) we can observe decreasing network load because 1 event matches multiple profiles (which decreases communication among brokers). But with growing σ this effect becomes less important. High values of p^p increase network load (except using EF). The duplication of profiles (Fig. 16(c)) decreases with growing coverings. PF shows the biggest duplication followed by RN and EF, whereas EF never distributes profiles. With lots of covering PF and RN converge to EF.



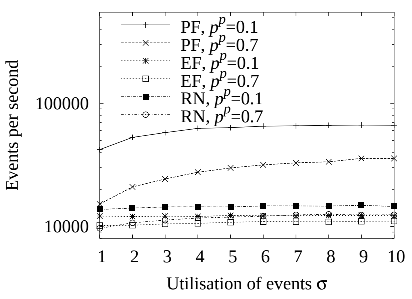
(a) Profile forwarding



(b) Event forwarding



(c) Rendezvous nodes



(d) All algorithms

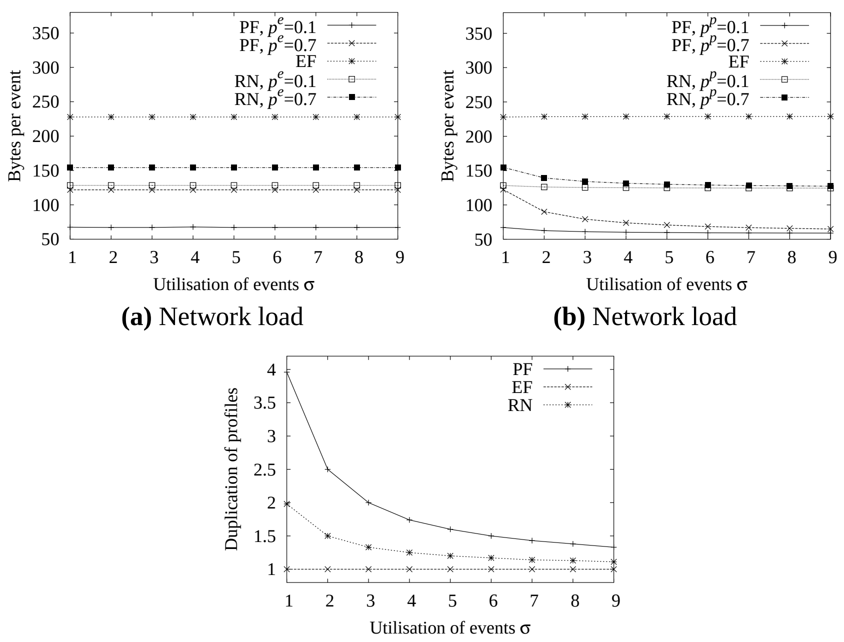
Figure 15: Efficiency dependent on the utilisation of events σ with various portions of matching profiles p^p

5.4 Influence of Event Types

In this section we analyse the influence of the number of event types. We use the network topology illustrated in Fig. 11. Thereby each broker is a rendezvous node for at most 1 type. In the experiments $180,000$ unique profiles are registered, so if using 3 types we have $60,000$ profiles per type and Brokers 1 to 3 are rendezvous nodes of these types. We analyse efficiency, duplication of profiles and network load.

We expect that the number of event types hardly influences the efficiency. PF and EF should be nearly independent on this number. Using RN the efficiency should increase when arranging rendezvous nodes well. The duplication of profiles should behave independent on the number of event types, except when using RN. There the paths to the rendezvous nodes affect the duplication of profiles. The same should hold for the network load.

The filter efficiency is illustrated in Fig. 17 (consider the different scalings). PF (Fig. 17(a)) shows nearly constant values. We can see a small increase in performance because of our central filter algorithm (Sect.4.1), which builds an own filter structure per event type. Increasing p^p decreases the performance because of more notifications. EF (Fig. 17(b)) behaves in a similar way. Except the fact that p^p has nearly no influence, since the overhead of flooding dominates the processing of notifications. RN



(a) Network load

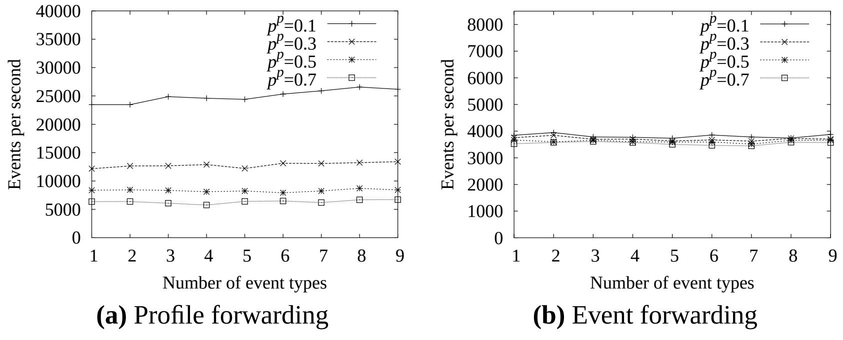
(b) Network load

(c) Duplication of profiles

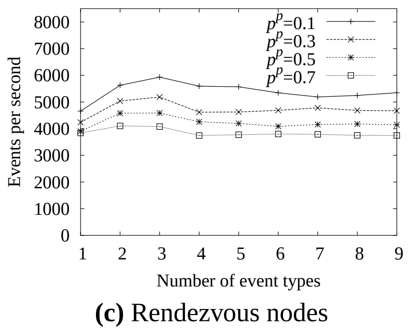
Figure 16: Network load and duplication of profiles dependent on the utilisation of events σ with various portions of matching events p^e and profiles p^p

(Fig. 17(c)) behave differently according to their location in the network. If they are located centrally (cases up to 3 types) we recognise an improvement. There rendezvous nodes are disburdened, since less events have to be filtered. Using more than 4 types the efficiency decreases. The reason is that rendezvous nodes are partially located in borders of the network. So inner nodes have to forward all events and become a bottleneck.

The duplication of profiles (Fig. 18(a)) is independent on the number of event types. Since we only use unique profiles, the duplication is 9.0 using PF, 1.0 using EF and between both of them using RN. When using more than 2 types, the duplication increases because of the position of the rendezvous nodes. An analogous behaviour is shown in Fig. 18(b) for the network load (consider the logarithmic ordinate). PF and EF show stable values with switched dimensions (PF less load, EF high load). Using RN we firstly realise a decrease and secondly an increase (longer paths to rendezvous nodes). Increasing p^p increases the network load in case of PF and RN. RN are less influenced, since there is a kind of ”needless” forwarding to the rendezvous node all the time.



(a) Profile forwarding **(b)** Event forwarding



(c) Rendezvous nodes

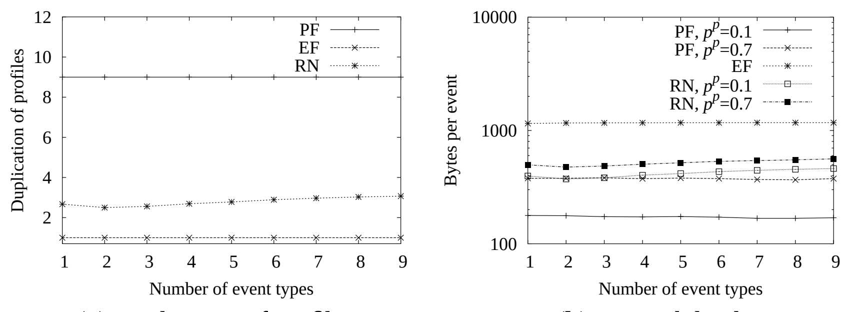
Figure 17: Efficiency dependent on number of event types using several portions of matching profiles p^p

5.5 Influence of Locality

The locality of profiles and events is analysed in this section. Hereby locality means that profiles from a broker's local publishers only match profiles from local subscribers. In the experiments we use 4 brokers as described in Section 5.1 as linear bus. $160,000$ profiles use exactly 1 event type. In the experiments we increase the number of matching profiles per broker (p^p (per broker), which means the locality). We analyse efficiency and network load. The duplication of profiles is not considered, since only events are changed in the experiments.

We expect an efficiency increase in case of more locality between profiles and events when using PF (since less notifications are forwarded to neighbours). Even using RN a little increase should occur: there is always some communication because of forwarding to the rendezvous node, so we only save a smaller part of communication complexity. Using EF we expect independence between locality and efficiency. Analogous the network load should behave. In PF and RN we expect less load and in case of using EF we expect independence.

Figure 19 shows the efficiency dependent on the locality. Consider the different scalings and the logarithmic ordinate in Fig. 19(d). Using PF (Fig. 19(a)) the efficiency increases up to a factor of 2 to 3.5 when increasing locality from 0 to 1. The reason



(a) Duplication of profiles **(b)** Network load

Figure 18: Duplication of profiles and network load dependent on number of event types

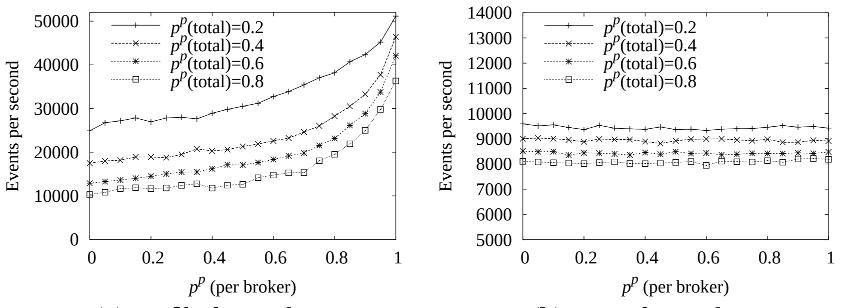
is the early reject of events at their local brokers. EF (Fig. 19(b)) is independent on locality. We have stable frequencies, since all events are always flooded in the network of brokers. RN (Fig. 19(c)) are less influenced than PF. So we only achieve an improvement of a factor of 1.25. The explanation is the forwarding of events to rendezvous nodes in all cases. Altogether PF shows a better adaptation to locality than the other 2 algorithms (Fig. 19(d)). RN do not show the adaptation as expected, since the communication overhead of the communication among brokers on the path to the rendezvous node exceeds the advantage of filtering in less brokers. The network load is shown in Fig. 20. EF is not influenced by the locality (flooding). PF and RN show decreasing network load (early rejecting of non-matching events).

5.6 Influence of the Number of Profiles

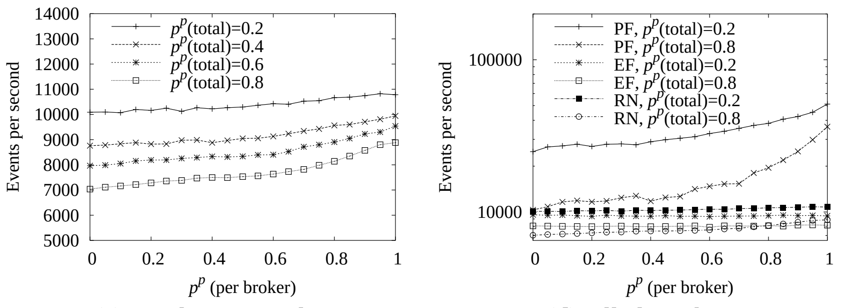
In this section we observe the influence of the total number of profiles. We again use 4 brokers connected as linear bus. We subscribe different numbers of unique profiles ($\sigma = 1$). The portion of matching events is chosen by $p^e = 0.8$. We only analyse efficiency.

With an increasing number of profiles the efficiency should become worse. Starting from a certain value the efficiency should heavily decline. Here the main memory is fully loaded and swapped out to secondary memory. Using PF and RN this happens very fast. EF should behave more stable when using large numbers of profiles, since they are not duplicated.

Figure 21 shows the system's efficiency with an increasing number of profiles. Up to $100,000$ (PF, RN) resp. $350,000$ (EF) unique profiles can be managed and filtered in the main memory. When Using coverings this value increases using PF or RN, since covered profiles are only filtered by local brokers. PF shows the best efficiency as long as the profiles are stored in main memory, followed by EF and RN. Since the large portion of matching events ($p^e = 0.8$), RN are less efficient than EF as described in Sect. 5.1. Using more than $100,000$ profiles in cases of PF and RN means



(a) Profile forwarding **(b)** Event forwarding



(c) Rendezvous nodes **(d)** All algorithms

Figure 19: Efficiency dependent on locality using different portions of matching profiles p^p

a heavy decline in efficiency. Either the rendezvous nodes are the system's bottleneck or all brokers when using PF. In EF this effect appears at $350,000$ profiles because we use 4 brokers that can managed approx. 4 times more profiles (since there are no duplications).

5.7 Conclusions

In this section, we described our extensive analysis of the prototype DAS. We evaluated the influences of several system parameters on the efficiency of the system. The results of our analysis can be summarised as follows:

Filter efficiency: PF is the most efficient algorithm in most cases. Especially if there is a low portion of matching profiles or events PF is much better than EF or RN. High portions of matching profiles mean a convergence of the 3 algorithms, since all events have to be flooded. In rare cases, EF is the most efficient algorithm. The reason is the simple filter protocol with less overhead in cases of high portions of matching profiles. RN mostly achieves an efficiency between the values of the other 2 algorithms. There is no advantage in using RN when using large numbers of event types, since inner brokers always have to forward all events.

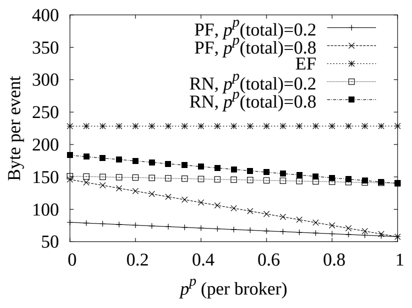


Figure 20: Network load dependent on locality

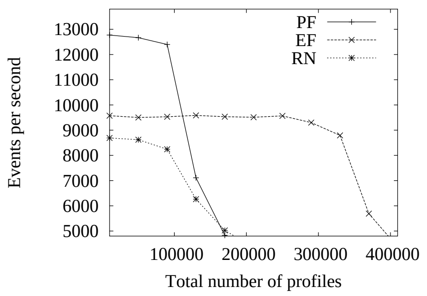


Figure 21: Efficiency dependent on the number of profiles

Network load per event: The network load using EF is mostly independent from other system parameters (except the number of brokers), since all events are always flooded. PF ever shows the least network load (only forwarding of matching events), RN show values between PF and EF (forwarding to the rendezvous node in all cases). When increasing p^e or p^p the network load also increases when using PF or RN, but the amount of forwarding using EF is never reached.

Duplication of profiles: Duplication of profiles shows the largest values using PF. Especially when using unique profiles the duplication (which implies memory usage) is high (each broker filters each profile). The same holds for RN, only less pronounced. If covering exist amongst the profiles, this duplicating effect vanishes. EF does not duplicate profiles and can filter the largest amounts of profiles.

Due to this dependency of the filter algorithms on the system's parameters, an ENS should support different filter algorithms. According to the system's load and its current application, the system can then choose an optimal algorithm. If many profiles match an event we should choose EF with its simple protocol. Also flooding is not a problem, since the events have to be forwarded anyway. We should also use EF in case of large amounts of profiles subscribed. PF should be used in most of the other cases (less profiles, small portions of matching events, coverings). Hereby less network load is produced and the filtering is much more efficient compared to EF and RN. Unfortunately, rendezvous nodes have not been advantageous in any of our tested system configurations.

6 Conclusion

In this paper, we proposed a classification scheme for distributed filter algorithms applicable in Event Notification Services. We then classified existing filter algorithms according to the proposed scheme and presented a first-cut theoretical evaluation. Based on the results of this evaluation we selected the three most promising filter algorithms for a practical analysis: profile forwarding, event forwarding and rendezvous nodes. The analysis has been performed using our prototype of a distributed ENS. The filter algorithm used in each broker of DAS is an extension of the most efficient tree-based filter algorithm.

In our practical evaluation we analysed the influences of several system parameters (portion of matching profiles, portion of matching events, number of brokers, portion of coverings, number of event types, locality of profiles and events and number of profiles) on filter efficiency, network load and memory consumption. Our experiments showed that profile forwarding mostly generates the lowest network traffic and the best efficiency. In contrast the memory consumption is very high. Event forwarding shows the best efficiency if there is a high portion of matching events. Because of its optimal memory consumption this approach results in good scalability. However, the network load is very high. The third algorithm, rendezvous nodes, have never been the best filter strategy in our experiments.

Based on the results of our practical analysis, we conclude the an ENS should support several filter algorithms. The distributed ENS should adapt its current filter algorithm according to the current system load and the application. This adaptation requires a reconfiguration of the ENS. The efficient realisation of this adaptation is part of our future work. A next step is the efficient support of composite profiles (combinations of several profiles) in DAS. Here we need to develop techniques to spread the filter complexity within the network. Finally, we plan to develop optimisation strategies for minimising redundancies among composite profiles, i.e., the application of coverings to composite profiles.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-Based Subscription System. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 53–61, Atlanta, USA, May 4–6 1999.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems. In *Pro-*

ceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99), pages 262–272, Austin, USA, May 31–June 4 1999.

[3] N. Belkin and W. B. Croft. Information retrieval and filtering: Two sides of the same coin. *Communications of the ACM (CACM)*, 35(12):29–38, 1992.

[4] S. Bittner. Implementierung eines effizienten Matchingverfahrens für Benachrichtigungssysteme. Student research project, Freie Universität Berlin, Institut of Computer Science, September 2002.

[5] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Interfaces and Algorithms for a Wide-Area Event Notification Service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999. revised May 2000.

[7] D. Faensen, L. Faulstich, H. Schweppe, A. Hinze, and A. Steidinger. Hermes – A Notification Service for Digital Libraries. In *Proceedings of the ACM/IEEE Joint Conference on Digital Libaries (JCDL '01)*, pages 373–380, Roanoke, USA, June 24–28 2001.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[9] J. Gough and G. Smith. Efficient Recognition of Events in a Distributed System. In *Proceedings of the 18th Australasian Computer Science Conference (ACSC-18)*, Adelaide, Australia, February 1 – 3 1995.

[10] A. Hinze. *A-MEDIAS: Concept and Design of an Adaptive Integrating Event Notification Service*. PhD thesis, Freie Universität Berlin, Institute of Computer Science, July 2003.

[11] A. Hinze and S. Bittner. Efficient Distribution-based Event Filtering. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, pages 525–532, Vienna, Austria, July 2–5 2002.

[12] H. A. Jacobsen, G. Ashayer, 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.

[13] S. König. Implementierung und Untersuchung eines parametergesteuerten Benachrichtigungssystems für kombinierte Events. Diploma thesis, Freie Universität Berlin, Institute of Computer Science, April 2004.

[14] Lichtvision. Personal communication with facility mangement experts from Lichtvision GmbH. Contact Lichtvision at `http://www.lichtvision.de/` as of September 2003.

[15] C. Liebig, B. Boesling, and A. Buchmann. A Notification Service for Next-Generation IT Systems in Air Traffic Control. In *Proceedings of the GI-Workshop: Multicast-Protokolle und Anwendungen*, pages 55–68, Braunschweig, Germany, May 19–21 1999.

[16] L. Liu, C. Pu, W. Tang, and W. Han. CONQUER: A Continual Query System for Update Monitoring in the WWW. *International Journal of Computer Systems, Science and Engineering, Special Issue on Web Semantics*, 14(2):99–112, 1999.

[17] G. Mühl. Generic Constraints for Content-Based Publish/Subscribe Systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS '01)*, pages 211–225, Trento, Italy, September 5–7 2001.

[18] G. Mühl. *Large-Scale Content-Based Publish/Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, September 2002.

[19] 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.

[20] G. Mühl, L. Fiege, and A. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS '02)*, pages 224–238, Karlsruhe, Germany, April 8–12 2002.

[21] P. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW '02)*, pages 611–618, Vienna, Austria, July 2–5 2002.

[22] P. Pietzuch and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS '03)*, San Diego, USA, June 8 2003.

[23] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Proceedings of the 3rd International Workshop on Networked Group Communications (NGC 2001)*, pages 30–43, London, UK, November 7–9 2001.

[24] G. Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, New York, 1968.

[25] H. Schweppe, A. Hinze, and D. Faensen. Database Systems as Middleware – Events, Notifications, Messages. In *Proceedings of 2000 ADBIS-DASFAA Symposium Symposium on Advances in Databases and Information Systems*, pages 21–22, Prague, Czech Republic, September 5–8 2000.

[26] The Internet Engineering Task Force (IETF). Internet Protocol, September 1981. Edited by J. Postel. RFC 791.

[27] The Internet Engineering Task Force (IETF). Transmission Control Protocol, September 1981. Edited by J. Postel. RFC 793.

[28] The Internet Engineering Task Force (IETF). Network News Transfer Protocol, February 1986. Edited by B. Kantor und P. Lapsley. RFC 977.

[29] The Internet Engineering Task Force (IETF). Internet Protocol, Version 6 (IPv6), December 1998. Edited by S. Deering and R.Hinden. RFC 2460.

[30] The Internet Engineering Task Force (IETF). Hypertext Transfer Protocol – HTTP, June 1999. Edited by R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach und T. Berners-Lee. RFC 2616.

[31] B. Wu and K. Dube. PLAN: A Framework and Specification Language with an Event-Condition-Action (ECA) Mechanism for Clinical Test Request Protocols. In *Proceedings of the 34th Hawaii International Conference on System Science (HICSS-34)*, Maui, USA, January 3–6 2001.

[32] H. Yu, D. Estrin, and R. Govindan. A Hierarchical Proxy Architecture for Internet-scale Event Services. In *Proceedings of IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE '99)*, pages 78–83, Stanford, USA, June 16–18 1999.