Filter Similarities in Content-Based Publish/Subscribe Systems

Gero Mühl* Ludger Fiege* Alejandro Buchmann

Department of Computer Science
Darmstadt University of Technology, D-64283 Darmstadt
{fiege,gmuehl}@gkec.tu-darmstadt.de
buchmann@informatik.tu-darmstadt.de

Abstract. Matching notifications to subscriptions and routing notifications from producers to interested consumers are the main problems in large-scale publish/subscribe systems.
Most previously proposed distributed notification services either use flooding or, if filtering is performed, they assume that each event broker has global knowledge about all active subscriptions. Both approaches degrade the scalability of notification services as the former wastes network resources and the latter generates overly large routing tables.
In this paper we describe content-based routing algorithms that exploit filter similarities in order to reduce the size of routing tables and the number of control messages that are exchanged among the brokers in order to keep the routing tables up-to-date. In particular, the proposed algorithms do not assume global knowledge about all active subscriptions. Furthermore, we describe how these optimizations can be supported if the underlying data and filter model is based on structured records.

1 Introduction

Publish/subscribe provides means for the loosely coupled exchange of asynchronous messages. A *publish/subscribe system* consists of a set of nodes that communicate by exchanging notifications with the help of a notification service that is interposed between the producers and consumers. A broad range of applications can benefit from a solution that is based on publish/subscribe. For example, electronic trading platforms including stock exchanges, auction sites, and reverse auction platforms are inherently event-based [3, 19]. Also applications from the area of ubiquitous computing where clients are interested in up-to-date data and bandwidth is scarce or expensive are good candidates.

A *notification* is simply a message that contains some information called its *content*. *Clients* are producers, consumers, or both. *Producers* publish notifications and *consumers* subscribe to notifications by issuing *subscriptions* that are

* Supported by the German National Science Foundation (DFG) as part of the PhD program "Enabling Technologies for Electronic Commerce" at Darmstadt University of Technology.

essentially stateless message filters. Consumers can have multiple active subscriptions and after a client has issued a subscription the notification service is responsible for delivering all future matching notifications that are published until the client cancels the respective subscription.

The expressiveness of the subscription model is crucial for both the flexibility and the scalability of a notification service. Insufficient expressiveness can lead to unnecessary broad subscriptions stressing the network and raising the need for additional consumer-side filtering. On the other hand, scalable implementations of more expressive description models require complex delivery strategies [5].

Content-based filtering allows subscriptions to evaluate the whole content of a notification, leveraging finer notification selection both inside the notification service and at the clients. Therefore, it provides a more powerful and flexible notification selection than it is possible for channel- or subject-based notification services where the content of a notification is opaque.

Centralized solutions of content-based notification services do not scale and the use of a distributed implementation that is built upon a set of cooperating event brokers is the key for scalability. Besides flooding, content-based routing is known from the literature. Here, a broker forwards a notification that it processes to some other brokers that is determined by a filter-based routing table. Unfortunately, almost all approaches dealing with content-based routing assume that each broker has global knowledge about all active subscriptions [2, 21]. In our view, flooding and content-based routing that assumes global knowledge degrade the scalability of notification services because the former may waste network resources while the latter generates overly large routing tables.

As an alternative approach Carzaniga [4] has shown that global knowledge about all active subscriptions is not necessary in order to implement a routing algorithm that solely forwards notifications that match active subscriptions. He proposed to use covering tests among filters to reduce the amount of information that is needed by a broker to determine the set of brokers to which an incoming notification must be forwarded. However, his work has only considered some predefined constraints on primitive data types (e.g., comparisons among integers). In a previous paper we have presented how more complex data types and constraints can be supported and also how filters can be merged [17]. We also presented preliminary ideas how to apply covering tests and merging to semistructured data and objects [18].

In this paper we combine and build on previous results and describe a set of routing algorithms that exploit filter similarities by applying identity and covering tests as well as carrying out filter merging. We show how to support these routing optimizations if the underlying data and filter model is based on structured records.

The remainder of this paper is organized as follows: Section 2 presents our system model and describes some routing algorithms that exploit filter similarities. In section 3 we show how to support identity and covering tests as well as filter merging for structured records. In the final sections we give an overview of some related work and briefly depict our notification service called Rebeca.

2 Content-Based Routing

2.1 System Architecture

Clearly, a notification service that relies on a centralized broker cannot be scalable. It may match a notification against a large set of subscriptions very fast [1, 7], but it will not be able to communicate with millions of clients. Moreover, a centralized broker is a single point of failure. In consequence, an implementation is needed that distributes the functionality of the service.

The key for a scalable notification service is to use content-based routing. In a publish/subscribe system that is based on *content-based routing* a set of cooperating *brokers* is arranged in a distributed topology. The *topology* of a distributed broker network is a connected undirected graph $G = (V, E)$ with a set of nodes $V = \{B_1, \ldots, B_n\}$ corresponding to the brokers and a set of edges $E \subseteq \{(B_i, B_j) \mid 1 \leq i < j \leq n\}$ representing connections among them. For convenience, we define a function $e(B_i, B_j)$ that returns (B_i, B_j) if $i < j$ and (B_j, B_i) otherwise.

In general, a topology is not static but it changes when new connections are established or existing connections are closed. One can distinguish between acyclic and generic topologies. In an *acyclic* topology between any two brokers exactly one path exists. In this case each broker is a single point of failure because if a single broker fails the topology is partitioned into two disconnected sub-topologies. Contrary to that, cycles can exist in a *generic* topology allowing for multiple paths between two brokers. Hence, a broker may not be a single point of failure but special care must be taken to avoid duplicated notifications and passing notifications and control messages in cycles. A simple and well-known method [4, 5] to support generic topologies is to define for each broker B a (minimal) spanning tree of G that is used to route notifications originating from B. With this approach every broker is still a single point of failure but if a broker fails the spanning trees can be adapted accordingly. In order to resiliently tolerate broker failures multiple independent paths must exist between any two brokers [13, 14, 24]. In this paper we concentrate on acyclic topologies. In first approximation, routing algorithms are independent of the underlying transport mechanisms (e.g., multicast and unicast). They only specify the flow of notifications. Please refer to [2, 21] for a detailed overview showing how to efficiently use multicast to disseminate notifications.

Each broker B has a set of *neighbor brokers* $N_B = \{H \mid e(B, H) \in E\}$ and manages an exclusive set of *local clients* L_B. Each client X has a set of active subscriptions S_X that changes if X subscribes or unsubscribes to a filter. Each broker B delivers a notification that it processes to all of its local clients that have a matching subscription, i.e., $\{X \mid X \in L_B \wedge \exists F \in S_X.\, n \in N(F)\}$. Additionally, every broker forwards a notification to a subset of its neighbors by evaluating a filter-based routing table.

Formally, a *filter* F is a stateless boolean function that maps a notification n to the boolean values *true* and *false*. A notification n *matches* a filter F iff $F(n)$ evaluates to *true*. We denote by $N(F)$ the set of all notifications that match F.

In our model the *routing table* T_B of a broker B consists of a set of *routing entries* (F, U) where F is a filter and U is a neighbor of B. Let $F_B^N(n)$ be the set of neighbors of B for whom there exists a routing entry that matches a given notification n, i.e., $\{U \mid U \in N_B \wedge \exists (F, U) \in T_B . n \in N(F)\}$. Then B forwards a notification that it processes to all neighbors in F_B^N if n has been published by a local client of B and to all neighbors in $F_B^N \setminus \{H\}$ if B has received n from a neighbor H. Finally, the brokers exchange *control messages* in order to keep their routing tables up to date.

2.2 Flooding vs. Filtering

Flooding The technique of *flooding* can be seen as the simplest approach to implement content-based routing. In this case the routing tables of all brokers are initialized with constant routing entries such that $F_B^N(n)$ evaluates to N_B for all notifications n. Brokers do not exchange any control messages and therefore, they have no knowledge about active subscriptions of clients of other brokers and solely perform filtering on behalf of their local consumers. Hence, every broker simply forwards a notification that is published by one of its local clients to all of its neighbors and if a broker receives a notification from a neighbor it simply forwards it to all other neighbors. In consequence, each published notification is eventually processed by every broker and a lot of notifications may be forwarded unnecessarily. On the other hand, flooding may be a rather good choice if subscription profiles are equally distributed among the clients [21].

Filtering at intermediate Brokers Alternatively to flooding, filtering can be performed at intermediate brokers to reduce the number of notifications that are unnecessarily forwarded. In order to determine the minimal set of neighbors to which a broker must forward a notification assume for a moment that the set of active subscriptions is static and that the edge between a broker B and one of its neighbors U is removed from E. In this case the graph G is partitioned into two not connected subgraphs. Let $V_{U,B}$ be the set of all brokers that are nodes of the subgraph that contains the broker U. We denote by $\eta_{U,B}$ the set of all notifications that are of interest to any local consumer of a broker in $V_{U,B}$, i.e., $\cup_{H \in V_{U,B}} I_H$ with $I_H = \cup_{X \in L_H} \cup_{F \in S_X} N(F)$. Let $\phi_B(n)$ be the set of all neighbors of B for which n is in $\eta_{U,B}$, i.e., $\{U \mid U \in N_B \wedge n \in \eta_{U,B}\}$. Since G is acyclic a routing algorithm must ensure that $F_B^N(n)$ is a superset of $\phi_B(n)$ for all notifications. We call a routing algorithm *perfect* if $F_B^N(n) = \phi_B(n)$ for all notifications. Otherwise it is called *imperfect*. In general, a perfect routing algorithm requires that each broker has a more detailed knowledge about the active subscriptions but it also minimizes notification forwarding.

In a real system however, the set of active subscriptions changes as clients issue new or cancel existing subscriptions. The problem with this is that the set of notifications a client is interested in changes instantly while the routing tables cannot reflect this immediately. This means that the delivery of notifications which are exclusively matched by a new subscription is not guaranteed until

all necessary updates to the routing tables have been carried out. Besides the delivery of matching notifications a routing algorithm should try to minimize unnecessary forwarding of notifications. For example, the routing tables should be updated in reaction to cancellations of subscriptions, too.

In the next subsections we describe some routing algorithms that are based on *control message forwarding*: For each new and canceled subscription the broker that manages the corresponding client sends an individual control message to some of its neighbors. A broker that receives a control message from a neighbor updates its routing table (if necessary) and sends an individual control message to some of its other neighbor. Brokers process incoming control messages serially and in FIFO-order.

2.3 Routing based on Global Knowledge

A simple approach is to incorporate a routing entry for every active subscription into the routing tables of all brokers. To achieve this, the broker that manages the subscribing/unsubscribing client sends a control message that contains the new/canceled subscription S to all of its neighbors. A broker that receives such a control message from one of its neighbors H adds/extracts a corresponding routing entry (S, H) to its routing table and forwards the control messages unchanged to all of its other neighbors.

2.4 Routing based on Filter Identity

The simple routing algorithm described in the previous subsection enforces that all brokers have knowledge about all active subscriptions. Clearly, this is not feasible in a large scale system because this would result in huge routing tables and costly coordination. Therefore, it is crucial to minimize the number of routing entries that is needed in the routing table of a broker to determine F_B^N while still retaining the same quality of filtering. This can be achieved by taking into account *similarities* among the filters, i.e., the subscriptions. Of course, this requires that it is possible to detect a relation between the sets of matched notifications among filters.

For example, assume that we are able to detect that two filters match the same notifications. Formally, two filters F_1 and F_2 are *identical*, denoted by $F_1 \equiv F_2$, iff $N(F_1) = N(F_2)$. A broker does not need to forward a new/canceled subscription S_2 to a neighbor if another identical subscription S_1 has already been forwarded to that neighbor for that no corresponding unsubscription has been received yet. This reduces the size of the routing tables because routing entries with identical filters regarding the same neighbors are avoided. Moreover, the number of exchanged control messages is reduced, too.

2.5 Routing based on Filter Covering

The direct extension of identity-based routing is to apply more complex similarity tests among subscriptions. The next step is to exploit covering among filters.

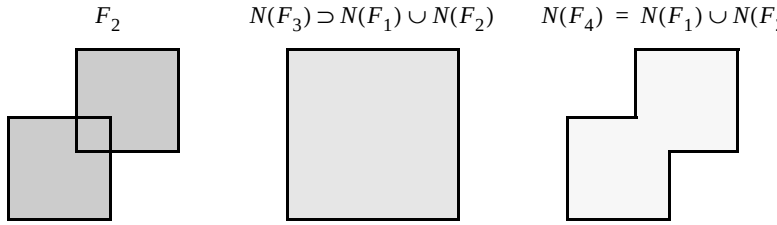

Fig. 1. Illustrating perfect and imperfect merging.

Let F_1 and F_2 be two arbitrary filters. We say that F_1 *covers* F_2, denoted by $F_1 \sqsupseteq F_2$, iff $N(F_1) \supseteq N(F_2)$. It is easy to see that if $F_1 \sqsupseteq F_2$ then $n \in N(F_2)$ implies $n \in N(F_1)$ and $n \notin N(F_1)$ implies $n \notin N(F_2)$. The pair $(\mathcal{F}, \sqsupseteq)$ defines a partial order over the set of all filters $\mathcal{F}$ and can be illustrated by a Hasse diagram.

Covering tests can be used to reduce the number of routing entries in routing tables as well as the number of control messages that must be forwarded. A broker B does not need to forward a subscription/unsubscription S_2 to a neighbor H if B has already forwarded a subscription S_1 to H that covers S_2 for which B has not received a corresponding unsubscription yet. Moreover, if B receives a new subscription S from a neighbor U it can drop those routing entries regarding U whose filters are covered by S. But this implies that in the case that B forwards an unsubscription S to a neighbor H then also all subscriptions that are covered by S of those routing entries regarding all brokers except H must be forwarded to H again.

In the context of notification services the use of covering was first described by Carzaniga [4, 5]. Computing covering tests is in general very expensive or even intractable. For example, computing covering tests for relational expressions or linear context-free grammars is *NP*-complete [10]. Fortunately, in practice special cases exist for which covering can be determined quite efficiently [17, 18].

2.6 Routing based on Filter Merging

In this subsection we describe how to extend the routing algorithm outlined in the previous section in order to exploit filter merging in addition to covering. In contrast to covering, merging does not merely rely on the filters that have been issued by the clients. Instead, new filters are derived from existing ones such that each new filter covers the set of filters it was generated from. We say that F is a *merger* of (or covers) a set of filters $\{F_1, \ldots, F_n\}$, denoted by $F \sqsupseteq \{F_1, \ldots, F_n\}$ iff $N(F) \supseteq \big(\cup_i N(F_i)\big)$. The merger F is *perfect* if the equality holds and *imperfect*, otherwise (see Fig. 1).

The basic idea of merging-based routing is that each broker can merge filters of existing routing entries and forward the generated merger to a subset of its neighbors. As a merger covers the filters it was generated from, a broker that

receives a merger from a neighbor will drop those routing entries that belong to this neighbor and represent the merged filters. Hence, the number of routing entries is reduced. Periodically or triggered by the receipt of a control message every broker investigates its routing table and checks whether it can generate new mergers and if the existing mergers can be kept. After that the broker forwards subscriptions/unsubscriptions corresponding to each new/canceled merger to a specific subset of its neighbors. If it forwards an unsubscription corresponding to a canceled merger, the broker also embeds the subscriptions that were covered by the merger into the control message.

The following example illustrates that generating a perfect merger is not sufficient to guarantee that no notifications will be forwarded unnecessarily. In fact, the subset of neighbors to which a merger is forwarded plays an important role. Consider a broker B_3 with two neighbors B_1 and B_2. The routing table of B_3 consists of two routing entries (F_1, B_1) and (F_2, B_2) with $N(F_1) \neq N(F_2)$. Now, B_3 decides to forward a perfect merger of F_1 and F_2 to all of its neighbors, i.e., B_1 and B_2. In this example, broker B_1 will also forward notifications that match F_2 but not F_1 and broker B_2 will forward notifications that match F_1 but not F_2 to B_3 although B_3 will not forward any of these notifications to B_1 or B_2. In fact, the set of neighbors to which a merger can be forwarded without raising this problem depends on the routing entries from which it was generated: a merger can be forwarded to all neighbors if for each routing entry (F, H) from which it was generated there exists another routing entry (G, I) such that $N(F) = N(G)$ and $H \neq I$.

At a first glance, imperfect merging seems to be less promising, but in situations in which perfect merging cannot be applied it might be a good compromise. On one hand, imperfect merging results in notifications being forwarded that do not match any of the original subscriptions and one must be careful to avoid that the effects of imperfect merging are chained along delivery paths such that the routing degenerates to flooding. But on the other hand, imperfect merging can greatly reduce the amount of subscriptions that must be dealt with.

In order to apply merging it must be possible to efficiently compute mergers and if imperfect merging is performed the fraction of the irrelevantly matched notifications must be sufficiently small. Merging is powerful but also complex and its usability needs further investigation.

2.7 Use of Advertisements

Advertisements are filters that are issued by producers to indicate their intention to publish notifications. In our model each notification that is published by a producer must match one of its active advertisements. Advertisements can be used as additional mechanism to further optimize content-based routing [4]. For this purpose they are propagated through the broker network in the same way as described for subscriptions in order to route subscriptions more efficiently: a subscription is only forwarded to a neighbor if it overlaps with an advertisement that has been received from this neighbor. The only underlying assumption is

that it is possible to detect whether a given advertisement A and a given subscription S are *overlapping*, i.e., whether or not $N(S) \cap N(A) \neq \emptyset$. All routing algorithms presented in the former sections can be easily extended in such a way that advertisements are used.

2.8 Discussion

The routing algorithms are getting more complex by applying the proposed routing optimizations such as covering and merging. But on the other hand, these improvements reduce the size of the routing tables and may also reduce the number of exchanged control messages. If they are used, the efficient evaluation of these optimizations will be crucial for the load induced on the brokers. Normally, a more expressive data and filter model tends to make these optimization more complex, too. Moreover, there exists a trade-off between network and computing resource usage/wastage and in our view there will be no static solution that is optimal for all application scenarios. We propose to use statistical online evaluation of connection and filter selectivity as a basis to adapt routing algorithms: the forwarding broker disables filtering if the matching rate exceeds a certain threshold while the receiving broker can request to turn filtering on again if the relative amount of forwarded notifications that do not match is too large.

3 Supporting Filter Similarities for Structured Records

Many systems model notifications similar to structured records consisting of a set of name/value pairs called attributes. Examples are SIENA [4], Gryphon [1, 2], REBECA [8], and the CORBA Notification Service [20]. In this model attributes are addressed by their unique name and constraints are imposed on the values of the respective attributes. Besides *flat* records in which values are atomic types, *structured* records in which attributes may be nested can also be supported by using a dotted naming scheme (e.g., *Position.x*).

Some systems restrict constraints to depend on a single attribute (e.g., $x = 1$) while other systems allow them to depend on multiple attributes which are combined by operators (e.g., $x+y = 5$). Multiple constraints can be combined by boolean operators (e.g., $y < 3 \wedge x = 4$). SIENA and REBECA limit constraints to depend on a single attribute and the combination of constraints to conjunctions in order to allow for efficient evaluation of routing optimizations. In the following we present some of the basics that underlie the proposed routing optimizations such as covering and merging.

3.1 Notifications

Formally, a notification n is a set of attributes $\{A_1, \ldots, A_n\}$ where each A_i is a *name/value pair* (n_i, v_i) with *name* n_i and *value* v_i. We assume that names are unique, i.e., $i \neq j$ implies that $n_i \neq n_j$, and that there exists a function that uniquely maps each n_i to a type T_k that is the type of the corresponding value v_i

(e.g. *Integer*). Moreover, a notification has a mandatory attribute with name *type* that indicates the type of the notification (e.g., *StockQuote*) in order to enable *type-based filtering* based on a type hierarchy. For each type a set of additional *mandatory attributes* is defined that may be empty. An example of a notification in this model is $\{(type, StockQuote), (name, \text{"Foo Inc."}), (price, 45.0)\}$.

3.2 Filters

A filter consisting of a single atomic predicate is a *simple filter* or *constraint*. Filters that are derived from simple filters by combining them with boolean operators are *compound filters*. A compound filter that is a conjunction of simple filters is called a *conjunctive filter*. Any compound filter can be converted into its *DNF (disjunctive normal form)* that consists of a disjunction of a set of conjunctive filters whose size may be exponential in the worst case. As multiple subscriptions of a single client are interpreted disjunctively this implies that it is sufficient to support conjunctive filters. Therefore, we restrict the discussion to conjunctive filters for the rest of this paper.

3.3 Attribute Filters

We model filters as conjunctions of attribute filters that are simple filters and impose a constraint on the value of a single attribute (e.g., $\{name = \text{"Foo Inc."}\}$). Hence, a notification n matches a filter F iff it satisfies all attribute filters of F. Moreover, a filter with an empty set of attribute filters matches any notification.

An *attribute filter* is defined as a tuple $AF_i = (n_i, Op_i, C_i)$ where n_i is an attribute name, Op_i is a test operator and C_i is a set of constants that may be empty. The name n_i determines to which attribute the constraint applies. If the notification does not contain an attribute with name n_i then AF_i evaluates to *false*. Therefore, each constraint implicitly defines an existential quantifier over the notification. Otherwise, the operator Op_i is evaluated using the value of the addressed attribute and the specified set of constants C_i. We assume that the types of operands are compatible with the used operator. The outcome of AF_i is defined as the result of Op_i that evaluates either to *true* or *false*. We also provide an attribute filter that simply checks whether a given attribute is contained in n. For the sake of simplicity we use the more readable notation $\{price > 10\}$ instead of $\{(price, >, \{10\})\}$. An example for a conjunctive filter consisting of attribute filters is $\{(type = StockQuote), (name = \text{"Foo Inc."}), (price \notin [30, 40])\}$. Note, that this filter potentially matches also all notifications whose type is a subtype of *StockQuote*.

By $L(AF_i) \subseteq dom(T_k)$ we denote the set of all values that cause an attribute filter to match an attribute, i.e., $\{v_i \mid Op_i(v_i, C_i) = true\}$. We assume that $L(AF_i) \neq \emptyset$. An attribute filter AF_1 covers an attribute filter AF_2, written $AF_1 \sqsupseteq AF_2$, iff $n_1 = n_2 \wedge L(AF_1) \supseteq L(AF_2)$. For example, $\{price > 10\}$ covers $\{price \in [20, 30]\}$.

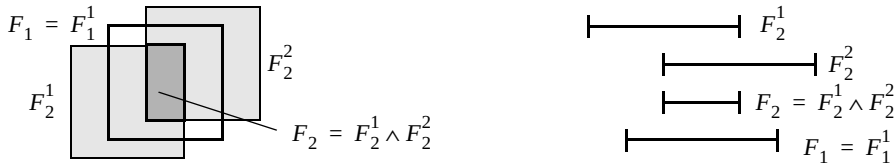

Fig. 2. $F_1 \sqsupseteq F_2$ although neither $F_1^1 \sqsupseteq F_2^1$ nor $F_1^1 \sqsupseteq F_2^2$ (two examples).

3.4 Covering of Conjunctive Filters

Here, we investigate covering of filters that are conjunctions of attribute filters.

Proposition 1. *Given two filters* $F_1 = AF_1^1 \wedge \ldots \wedge AF_1^n$ *and* $F_2 = AF_2^1 \wedge \ldots \wedge AF_2^m$ *that are conjunctions of attribute filters, the following holds:* $\forall i \exists j.\ AF_1^i \sqsupseteq AF_2^j$ *implies* $F_1 \sqsupseteq F_2$.

ASSUME: $\forall i \exists j.\ AF_i^1 \sqsupseteq AF_j^2$
PROVE: $F_1 \sqsupseteq F_2$
PROOF: If an arbitrary notification n is matched by F_2 then n satisfies all AF_2^j. This fact together with the assumption implies that n also satisfies all AF_1^i. Therefore, n is matched by F_1, too. Hence, $F_1 \sqsupseteq F_2$. □

If several attribute filters can be imposed on the same attribute then $\forall i \exists j. AF_1^i \sqsupseteq AF_2^j$ is not a necessary condition for $F_1 \sqsupseteq F_2$ (see also Fig. 2). If we restrict conjunctive filters to have at most one attribute filter for each attribute then we can strengthen Proposition 1 to an equivalence:

Proposition 2. *Given two filters* $F_1 = AF_1^1 \wedge \ldots \wedge AF_1^n$ *and* $F_2 = AF_2^1 \wedge \ldots \wedge AF_2^m$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute, the following holds:* $F_1 \sqsupseteq F_2$ *implies* $\forall i \exists j.\ AF_1^i \sqsupseteq AF_2^j$.

ASSUME: $\neg(\forall i \exists j.\ AF_1^i \sqsupseteq AF_2^j)$
PROVE: $\neg(F_1 \sqsupseteq F_2)$
PROOF: We construct a notification n that matches F_2 but not F_1 to prove that F_1 does not cover F_2. The assumption implies that there is at least one AF_1^k that does not cover any AF_2^j. If there exists an AF_2^l that constrains the same attribute as such an AF_1^k then choose for this attribute a value that matches AF_2^l but not AF_1^k. Such a value exists because $L(AF_1^k) \neq \emptyset$ and $AF_1^k \not\sqsupseteq AF_2^l$. Add name/value pairs for all other attributes that are constrained in F_2 such that they are matched by the appropriate attribute filters of F_2. The constructed notification matches F_2 but not F_1. Therefore, F_1 does not cover F_2. □

Corollary 1. *Given two filters* $F_1 = AF_1^1 \wedge \ldots \wedge AF_1^n$ *and* $F_2 = AF_2^1 \wedge \ldots \wedge AF_2^m$ *that are conjunctions of attribute filters with at most one attribute filter per attribute,* $F_1 \sqsupseteq F_2$ *is equivalent to* $\forall i \exists j.\ AF_1^i \sqsupseteq AF_2^j$.

PROOF: by Proposition 1 and 2. □

The limitation to at most one attribute filter for each attribute is not severe because our system provides complex data types as attribute values and an extensible set of constraints that can be imposed. Moreover, it is often possible to merge several conjunctive constraints imposed on a single attribute into a single constraint on the same attribute. If the result of a conjunction of two constraints of some constraint type always yields another constraint of the same type then this set of constraints is either contradicting or can be replaced by a single constraint of the same type. We call such types of constraints and their corresponding attribute filters *conjunction-complete*. For example, constraints testing whether a point is in a given rectangle in a two-dimensional plane are conjunction-complete. If a constraint type is not conjunction-complete it is often possible to substitute a set of such constraints by a single constraint of a more general type. For example, a set of ordering constraints defined on a totally ordered set (e.g., integer numbers) are either contradictory or can be replaced by a single interval constraint. In a previous paper [17] we have presented an algorithm that determines the possibly empty set of filters that cover a given filter which is derived from the predicate counting matching algorithm [26].

3.5 Identity and Overlapping of Conjunctive Filters

The following two propositions show how identity and overlapping of conjunctive filters can be reduced to their respective attribute filters. The proofs are left out due to space reasons.

Proposition 3. *Two filters* $F_1 = AF_1^1 \wedge \ldots \wedge AF_1^n$ *and* $F_2 = AF_2^1 \wedge \ldots \wedge AF_2^m$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute are identical iff they contain the same number of attribute filters and* $\forall AF_1^i \exists AF_2^j . \left(n_1^i = n_2^j \wedge L(AF_1^i) = L(AF_2^j)\right)$.

Proposition 4. *Two filters* $F_1 = AF_1^1 \wedge \ldots \wedge AF_1^n$ *and* $F_2 = AF_2^1 \wedge \ldots \wedge AF_2^m$ *that are conjunctions of attribute filters with at most one attribute filter for each attribute are overlapping iff* $\nexists AF_1^i, AF_2^j . \left(n_1^i = n_2^j \wedge L(AF_1^i) \cap L(AF_2^j) = \emptyset\right)$

3.6 Merging of Conjunctive Filters

In the general case purely algebraic merging techniques have exponential time complexity. Alternatively, a predicate proximity graph can be used to implement a greedy algorithm [15]. For many practical cases (e.g., the set operators) efficient algorithms exist.

An algorithm that determines the possibly empty set of filters which are candidates to be merged with a given filter was depicted in a previous paper [17]. From the set of merging candidates the set of attribute filters to be merged can be easily extracted. This set is used as input of a merging algorithm which has a specialized implementation for each type of constraint. Only in rare cases it is necessary to use an exhaustive combinatorial or a suboptimal greedy algorithm.

Perfect Merging A set of conjunctive filters with at most one attribute filter for each attribute can be perfectly merged into a single conjunctive filter if for all except a single attribute their corresponding attribute filters are identical and if the attribute filters of the distinguishing attribute can be merged into a single attribute filter. For example, the two filters $F_1 = \{x = 5 \wedge y \in \{2, 3\}\}$ and $F_2 = \{x = 5 \wedge y \in \{4, 5\}\}$ can be merged to $F = \{x = 5 \wedge y \in \{2, 3, 4, 5\}\}$.

The characteristics of the constraints that are used to define attribute filters are important for merging. Constraints which only exist in a normal and a negated form can be directly merged by using some basic laws of boolean algebra. For example, a filter $F_1 = P_1 \wedge P_2$ can be merged perfectly with a filter $F_2 = P_1 \wedge \bar{P}_2$ to a filter $F = P_1$. Although these cases also exist for more complex constraints (e.g., $x = 5$ and $x \neq 5$) constraints are not restricted to be the negated form of each other. Better merging can be achieved by taking the specific characteristics of the imposed constraints into account.

A class of constraints that is *complete under disjunction* allows to merge a set of constraints of this class into a single constraint of the same class. Examples for disjunction-complete constraints are *set inclusions* (e.g., $x \in \{2, 3, 7\}$) and *set exclusions* (e.g., $x \notin \{2, 3, 7\}$) while *comparison constraints* (e.g., $x < 4$) are not disjunction-complete. If a constraint class is not disjunction-complete it may still be possible to carry out merging if a specific *merging condition* is met. For example, a set of *interval tests* (e.g., $x \in [2, 4]$ and $x \in [3, 5]$) can be merged into a single interval test (here, $x \in [2, 5]$) if the intervals form a connected set. Otherwise, merging may be possible if a more general constraint is considered as merging result. For example, two comparison constraints (e.g., $x < 4$ and $x > 7$) can be merged to an interval test (here, $x \notin [4, 7]$).

Imperfect Merging In order to use imperfect merging a set of heuristics is necessary that define in what situations and to what degree imperfect merging should be carried out. For example, filters that differ in few attribute filters could be merged imperfectly by imposing on each attribute a constraint that covers all original constraints. This could also be accomplished by explicitly replacing an attribute filter with another that only tests for the existence of the given attribute or by simply dropping the attribute filter. Note, that an existence test is equivalent to no constraint if the attribute is mandatory for the corresponding type of notification.

4 Related Work

Answering Queries using Views: Covering relations are known from the database theory and in particular from the area of answering queries using views [12, 25]. There, the question is whether the result set of a given query Q can be solely obtained from a set of predefined views V whose elements can be combined by the usual relational operators, i.e., whether Q is covered by some combination of the views in V. Answering this question for relational expressions is NP-hard even without comparison operators. If only the union operator is allowed, this

is still a more general scenario than ours. Although special cases have been investigated, we were not able to find an approach that is closely related to ours.

Semantic Caching: Lee and Chu [16] describe a semantic caching algorithm for conjunctive point queries that exploits covering between conjunctive predicates to find cache entries which cover a given query. However, this work is restricted to point queries involving the equivalence and the like operator. Godfrey and Gryz [11] depict an architecture for predicate-based caching that is similar to answering queries using views. Therefore, it is not surprising that their algorithms are *NP*-complete, too. Keller and Basu [15] propose a predicate-based caching scheme for client/server database architectures. They perfectly merge predicates in the cache to obtain a more compact cache description and to speed up query processing. Their algorithm has exponential time complexity.

Query Merging: Crespo et al. [6] propose merging of queries that are evaluated periodically against a database. As example, they use geographical queries represented by a rectangle. Before the queries are processed a merging algorithm is run that combines similar queries and outputs a set of merged queries whose answers contain all tuples of the original query. Their aim is to find a set of mergers which is cost optimal. They show that in the general case query merging is *NP*-complete and discuss optimal and heuristic algorithms.

Geometrical Algorithms: In the context of geometrical algorithms [22], for example, polygon inclusion, intersection, and containment of convex polygons are investigated. These algorithms can be integrated with our work to support efficient matching, covering, and merging of notifications containing geometric objects. Such objects are, for example, prevalent in geographical information systems.

Notification Services: SIENA [4] exploits covering relations between filters and applies them to subscription and advertisement forwarding, but their support for data types and constraints is very limited. Moreover, they do not support merging. Elvin [23] supports quenching in which notifications are first evaluated against a broader subscription that covers the disjunction of all subscriptions but no algorithms are described.

5 Implementation

In the context of our research project REBECA [9, 17] we investigate event-based architectures for E-commerce applications. We have realized a prototype of a content-based publish/subscribe middleware that relies on content-based routing and exploits covering and merging. The routing algorithms are implemented on top of a flexible routing framework in order to enable the testing of various

routing algorithms. Optionally, our system can use subscription and advertisement leasing in order to be more fault-tolerant. We also implemented support of notifications about new/canceled subscriptions and advertisements.

We have implemented a stock trading application based on real-time quotes to test the system under an observable load. At the moment we investigate the effects of using different data models and routing algorithms on the performance of the system. Moreover, we are developing another application dealing with meta-auctions that are a generalization of normal Internet auctions.

6 Conclusion

In this paper we outlined a set of content-based routing algorithms that exploit similarities among filters. In particular, we have described how identity and covering tests as well as filter merging can be used in order to reduce the size of routing tables and the number of exchanged control messages. We have also presented the basic mechanisms and assumptions that underly these optimizations and how they can be supported if the underlying data and filter model uses structured records. We suggested to use statistical on-line adaption of the filtering strategy to cope with the trade-off between network resource waste and processing cost overhead. Future work will include detailed studies of the effects of different filtering strategies based on our prototypical publish/subscribe middleware and the implemented applications.

References

1. M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching events in a content-based subscription system. In *PODC: 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 53–61, 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 *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–272, 1999.
3. C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An infrastructure for meta-auctions. In *Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'00)*, San Jose, California, June 2000.
4. A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998.
5. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
6. A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Efficient query subscription processing in a multicast environment. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*, 2000.
7. F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD 2001*, pages 115–126, 2001.

8. L. Fiege and G. Mühl. Rebeca Event-Based Electronic Commerce Architecture, 2000. http://www.gkec.informatik.tu-darmstadt.de/rebeca.
9. L. Fiege, G. Mühl, and F. C. Gärtner. A modular approach to build structured event-based systems. In *ACM Symposium on Applied Computing (SAC)*, Madrid, Spain, 2002.
10. 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.
11. P. Godfrey and J. Gryz. Answering queries by semantic caches. In *Database and Expert Systems Applications (DEXA) LNCS Vol. 1677*, pages 485–498. Springer, 1999.
12. A. Y. Halevy. Theory of answering queries using views. *SIGMOD Record*, 29, Dec. 2000.
13. Y. Huang and G.-M. Hector. Exactly-once semantics in a replicated messaging system. In *Proc. of the 17th International Conference on Data Engeneering (ICDE)*, 2001.
14. Y. Huang and G.-M. Hector. Replicated condition monitoring. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.
15. A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.
16. D. Lee and W. W. Chu. Conjunctive point predicate-based semantic caching for wrappers in web databases. In *Workshop on Web Information and Data Management*, 1998.
17. G. Mühl. Generic constraints for content-based publish/subscribe systems. In *Proceedings of the 6th International Conference on Cooperative Information Systems (CoopIS)*, pages 211–225. Springer, 2001.
18. 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.
19. G. Mühl, L. Fiege, and A. Buchmann. Evaluation of cooperation models for electronic business. In *Information Systems for E-Commerce, Conference of German Society for Computer Science / EMISA*, pages 81–94, Nov. 2000. ISBN 3-85487-194-5.
20. Object Management Group. Corba notification service. OMG Document telecom/99-07-01, 1999.
21. L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, and D. Sturman. Exploiting ip multicast in content-based publish-subscribe systems. In J. Sventek and G. Coulson, editors, *Middleware 2000*, volume 1795 of *LNCS*, pages 185–207. Springer-Verlag, 2000.
22. F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer, 1985.
23. W. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, September 1997.*, 1997. http://elvin.dstc.edu.au/doc/papers/auug97/AUUG97.html.
24. A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based content routing using xml. In *18th ACM Symposium on Operating System Principles*, 2001.
25. J. D. Ullman. Information integration using logical views. In *6th Int. Conference on Database Theory; LNCS 1186*, pages 19–40. LNCS 1186, Springer, 1997.
26. T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM Transactions on Database Systems*, 19(2):332–334, 1994.