Offloading content routing cost from routers

Janne Salo

Helsinki June 4, 2010

MSc thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Science	Department of Computer Science

Tekijä — Författare — Author
Janne Salo

Työn nimi — Arbetets titel — Title
Offloading content routing cost from routers

Oppiaine — Läroämne — Subject
Computer science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	June 4, 2010	64 pages

Tiivistelmä — Referat — Abstract

The publish/subscribe paradigm has lately received much attention. In publish/subscribe systems, a specialized event-based middleware delivers notifications of events created by producers (publishers) to consumers (subscribers) interested in that particular event. It is considered a good approach for implementing Internet-wide distributed systems as it provides full decoupling of the communicating parties in time, space and synchronization. One flavor of the paradigm is content-based publish/subscribe which allows the subscribers to express their interests very accurately. In order to implement a content-based publish/subscribe middleware in way suitable for Internet scale, its underlying architecture must be organized as a peer-to-peer network of content-based routers that take care of forwarding the event notifications to all interested subscribers. A communication infrastructure that provides such service is called a content-based network. A content-based network is an application-level overlay network.

Unfortunately, the expressiveness of the content-based interaction scheme comes with a price – compiling and maintaining the content-based forwarding and routing tables is very expensive when the amount of nodes in the network is large. The routing tables are usually partially-ordered set (poset) -based data structures. In this work, we present an algorithm that aims to improve scalability in content-based networks by reducing the workload of content-based routers by offloading some of their content routing cost to clients. We also provide experimental results of the performance of the algorithm. Additionally, we give an introduction to the publish/subscribe paradigm and content-based networking and discuss alternative ways of improving scalability in content-based networks.

ACM Computing Classification System (CCS):
C.2.4 [Computer-Communication Networks]: Distributed Systems — Distributed applications

Avainsanat — Nyckelord — Keywords
Publish/subscribe, content-based routing, poset

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

There is an increasing number of Internet-wide distributed systems. The traditional interaction scheme of point-to-point synchronous communication is too rigid and cumbersome for large-scale distributed systems. Many models have been proposed to solve the problem. One of them is the *publish/subscribe* (pub/sub) paradigm that allows for full decoupling of the communicating parties in time, space and synchronization [EFGK03]. A pub/sub system is an *event-based* middleware: the communicating parties either generate (publish) events or express their interest (subscribe) to certain kinds of events. It is the task of the middleware to deliver the event notifications to those interested; the event producers need not be aware of the event consumers or vice versa. The pub/sub paradigm is presented in more detail in Section 2, where we discuss the event-based nature of pub/sub systems, different variants of pub/sub interaction and ways to organize the architecture of such systems.

There are several ways to implement pub/sub middleware. In this work, we direct most of our attention to *content-based publish/subscribe* [CW02]. In content-based pub/sub, the subscribers express their interest in events by providing *filters* that are based on the whole content of the event. This scheme allows for very expressive and precise subscriptions; the subscribers receive only the events they are interested in. Other variants, like *topic-based* pub/sub, lack in expressiveness, because subscribers cannot accurately express their interests and may have to filter out unwanted content at their end. In Sections 3 and 4, we discuss the concepts of content-based networking and different strategies of forwarding and routing in content-based pub/sub middleware. A content-based network must also provide a sufficient guarantee of quality of service, security and reliability, but these issues are outside the scope of this work for the most part.

Unfortunately, implementing the content-based scheme in a scalable and efficient way is non-trivial. In this work, we present a method for improving scalability: offloading parts of the content-based routing algorithm from routers to clients (the term *client* in this context may refer to either a subscriber connected to the router or a neighboring router). Section 5 discusses *partially-ordered sets* (posets) and related data structures that are often used in storing and matching the subscription filters (i.e. forming the routing tables). The routing table offloading (RTO) and fixed filterset (FF) schemes described in Section 6 attempt to delegate parts of the computation involved in creating or updating the routing tables to the clients. In

Section 7, we give some experimental results for the performance of the RTO scheme in different cases. Section 8 briefly discusses alternative or additional methods that can be deployed to improve scalability, such as using distributed hash tables (DHTs) or Bloom filters. Section 9 gives some concluding remarks and discusses some questions left for future work.

2 The publish/subscribe paradigm

In the following, we discuss the publish/subscribe model in detail. We define the two main operations provided by any pub/sub interface, namely `publish()` and `subscribe()`. There is also an auxiliary operation called `advertise()` provided by many implementations. We also discuss the two dominant variants of pub/sub interaction: topic-based and content-based pub/sub. Later on, we focus solely on content-based pub/sub.

2.1 The event-based model

Pub/sub systems are *event-based*: subscribers (consumers) express their interest in an event or a pattern of events. The events are generated by publishers (producers) and any subscribers whose registered interests match the event are notified of it [EFGK03]. The pub/sub middleware is required to take care of storing and delivering the events and managing the subscriptions. We later discuss different ways to implement these requirements efficiently.

Subscribers register their interest in events by calling an operation named `subscribe()` on the event service. The event service then stores the subscription information. Subscriptions can be cancelled by using an `unsubscribe()` operation. Publishers generate *events* by calling a `publish()` operation on the service. The middleware then propagates the event to all interested subscribers, generating *event notifications* in relevant subscribers. Every subscriber typically provides a callback operation `notify()`, through which the event service delivers the notifications to the client.

In many event services, an operation called `advertise()` and its opposite `unadvertise()` are also available to the publishers. With `advertise()`, the publishers can express what kind of events they are going to produce in the future. In most systems, the input the `advertise()` operation accepts is identical to that of the

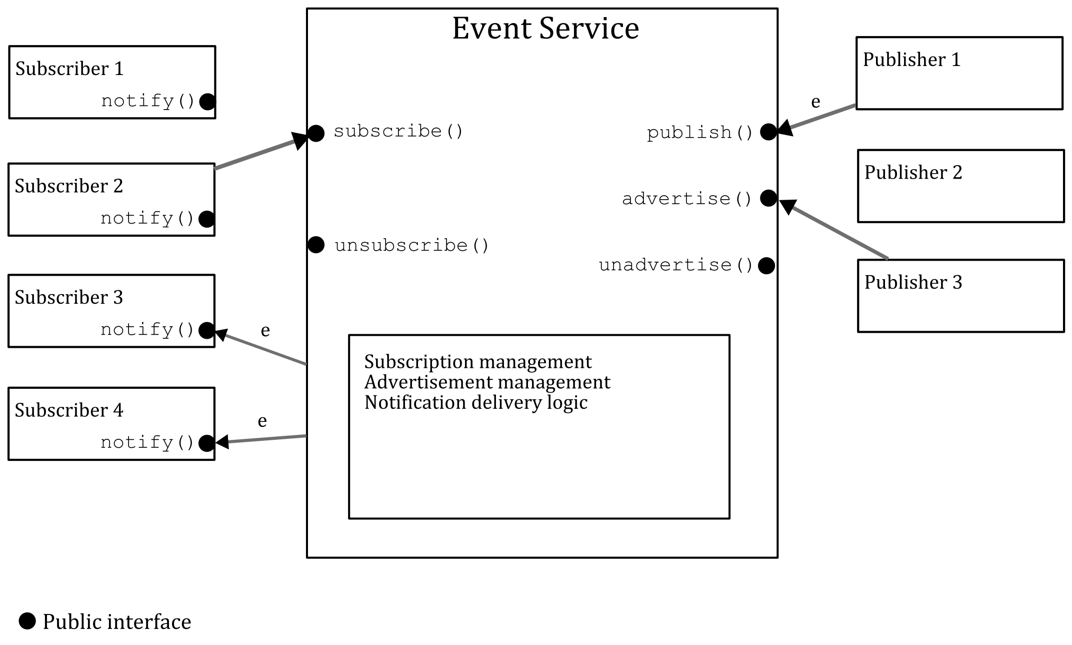


Figure 1: The high-level organization and public interfaces of a publish/subscribe system. In this example, Subscriber 2 has just issued a new subscription and Publisher 3 has issued a new advertisement. The event service notifies Subscribers 3 and 4 of event e that matches their previously issued subscriptions. The event e was published by Publisher 1, but the subscribers do not know its origin. Neither does Publisher 1 know who eventually receives a notification of the event it has published.

`subscribe()` operation, making `advertise()` the dual of `subscribe()` [Müh02, 4.4]. This allows for the event service to adjust its internal state in a way that makes delivering subsequent event notifications to interested subscribers more efficient. Advertisements can also be used to inform subscribers whenever a new type of information becomes available. We later argue that the `advertise()` operation plays very important role in content-based routing schemes. The organization and interfaces of a typical publish/subscribe system are shown in Fig. 1.

An event-based service provides a full decoupling of publishers and subscribers in space, time and synchronization [EFGK03]. In decoupling lies the strength of the event-based model: it greatly increases the scalability of distributed systems and reduces the need for coordination by removing any explicit relations between the data producers and consumers.

Space decoupling implies that the publishers and subscribers do not need to know each other – all events are propagated through the event service. Publishers do not hold references to the subscribers; they do not know how many subscribers will receive the event they generated. Neither do subscribers hold references to the publishers; it is not known to them which or how many of the publishers generated the event they received.

Due to *time decoupling*, the parties do not need to be active or even connected to the service at the same time: subscribers get notified of events generated while they were unavailable, or the event service may deliver notifications from publishers that are no longer connected.

Synchronization decoupling guarantees that no subscribers are blocked when a publisher produces an event. Conversely, no publishers are blocked while a subscriber receives a notification of an event. Instead, subscribers get notified of an event asynchronously through the `notify()` callback.

It should be noted that despite the network-layer terminology we use in this work (e.g. ”routing” and ”forwarding”), all current pub/sub implementations (some of which we will discuss later on) work on the application layer. While there is no theoretical obstacle for replacing IP with a pub/sub-oriented network (provided of course that the infrastructure is sufficiently robust, efficient and scalable), in practice it is most likely infeasible considering the vast size of the current IP infrastructure.

2.2 Topic-based publish/subscribe

In *topic-based* (or *subject-based*) pub/sub scheme, each event belongs under some predefined topic. The publishers are required to classify the events under the topics. The earliest pub/sub systems were topic-based [EFGK03]. Many industrial-grade solutions, such as TIBCO Rendezvous and iBus MessageServer also utilize the topic-based approach.

In topic-based systems, the set of topics is usually predefined and static. Each topic is identified by an unique string, the topic name. Subscribers subscribe to topics by simply passing the topic name to the `subscribe()` operation. Every generated event is categorized under a certain topic by the publishers and delivered to all subscribers by the middleware. Alternatively, the system can consist of several interfaces (channels), one for each topic. Each interface has its own `publish()` and `subscribe()` operations. Subscribing to a topic is then done by calling the `subscribe()` operation on the corresponding interface.

The topics can be organized either in *flat* or *hierachical* manner [EFGK03]. Flat organization (or *flat addressing*) is seldom used. It simply divides the event space into disjoint topics, while hierarchical organization (or *hierarchical addressing*) allows for containment relationship between topics. In hierarchical addressing scheme, subscribing to a topic implies also subscribing to all its subtopics. Many systems also permit the use of *wildcards* in topic names.

2.3 Content-based publish/subscribe

The static nature of the topic-based approach is its biggest shortcoming. To achieve granularity finer than just the topics (for example, ”everything under topic T except events whose property P has a value less than 100”), the subscriber must either ”over-subscribe”, i.e. subscribe to more events than necessary and filter out events that are of no interest, or every topic must contain many subtopics that can be used to narrow down the subscription. Over-subscribing leads to inefficient bandwidth usage (as there are unnecessary messages sent through the event service). Extensive usage of subtopics in turn leads to too deep hierarchies and too many topics [EFGK03].

The above problem is solved by the *content-based* approach. In content-based pub/sub, the subscriptions are based on the properties (attributes of the event data structure or separate metadata) of the events themselves and not on any predefined criteria such as topics. There are several (mostly academic) implementations of content-based pub/sub systems. These include for example Siena [CRW01], Hermes [Pie04] and Rebeca [Müh02, Ch. 5].

Subscribers specify the events they wish to receive by passing a *filter* to the `subscribe()` operation. There are many ways to express filters, but in this work, we will mostly focus on string-based filter languages (e.g. the one used by Siena), where filters are sets of name-operator-value triplets that act as constraints on the event content. The event service parses these strings into e.g. boolean functions that are used in matching individual events against subscriptions. In Siena, filters can also be combined into more complex *subscription patterns* that are matched against several events instead of a single event [CRW01]. Later in this work, we will discuss the different methods of internal organization of the subscription patterns in event services. There are also alternative ways to represent subscription patterns: some systems use XML- or SQL-based filter languages or use so-called *template objects* or even executable code instead of strings [EFGK03]. In Section 3, we will give a formal definition of a filter.

The content-based model allows for very accurate subscriptions – the subscribers can precisely express which events they are interested in. Moreover, the model does not restrict the consumers or the producers in any way, contrary to the topic-based approach where producers must choose a category for the event from a predefined set of topics, thus allowing for arbitrary events to be produced and subscribed to. In fact, it is easy to see that a topic-based system can be implemented using a content-based one. It is even possible (although impractical) to implement the traditional IP unicast on top of a content-based system; in this case, the events would be IP datagrams and the subscription patterns simple filters of format $recv\_addr = my\ ip$ [CW02].

There is, however, a trade-off between the expressiveness and scalability of the interaction model. As the subscribers are permitted to express their interests in more detail, the complexity of the algorithms needed to route and forward the events to correct subscribers grows [CRW00]. Content-based routing in particular has received much attention and is considered the biggest bottleneck in the scalability of content-based services. In the remainder of this work, we will primarily concentrate on content-based networking strategies and their scalability.

2.4 Data model

We now consider the format in which the notifications, subscriptions and advertisements in content-based pub/sub systems is presented. The data format varies per system; for example XML-based notifications or objects can be used [Müh02, Ch. 4]. However, in this work, we consider only Siena-style *structured records*. In this model, an event notification is a nonempty set of *attributes*. Each attribute is a name-value pair, or, like in Siena, a typed name-value pair, where attribute types belong to a predefined set of primitive types (such as `int`, `float` or `string`) [CRW01]. The type and attribute names in the records are simple strings. In addition to the record of attributes, the notification may contain fixed administrative data, such as the identifier of the notification issuer and the time of issuing. An example of a notification is shown in Fig. 2(a). In subsequent examples we often omit the type definitions, but assume that type information is available to the routers if needed.

Filters passed to subscriptions and advertisements are formatted much in the same way as notifications. A filter is a nonempty set of (possibly typed) *name-operator-value* triplets (*attribute constraints*), where allowed operators vary per system. Usually at least normal comparison operators ($=, \neq, <, >, \leq, \geq$) are available. In some

`string`	type	=	flight
`string`	airline	=	IBERIA
`time`	departure	=	2010-05-23 12:50Z
`time`	arrival	=	2010-05-23 20:40Z
`string`	origin	=	BCN
`string`	destination	=	JFK
`string`	gate	=	58B
`float`	price	=	600.00

(a) An example notification

`string`	type	=	flight
`string`	airline	=	*any*
`time`	departure	>	2010-05-23 00:00Z
`string`	origin	=	BCN
`string`	destination	=	JFK
`float`	price	≤	800.00

(b) An example filter

Figure 2: An example of a notification and a filter that are formatted as structured records. The attributes are typed.

systems, the *value* field can also consist of a set of values, in which case the constraint is interpreted as a disjunction, i.e. any of the given values will satisfy the constraint [Müh02]. We also allow the special value *any* to denote that any value will satisfy the constraint in question. In addition to the filter record, a subscription or an advertisement message contains at least the identifier of the issuer. An example of a filter is shown in Figure 2(b). For more concise representation of a filter F, we sometimes use the following:

$$\begin{aligned} F = (&\text{type} = \text{flight} \wedge \text{airline} = \textit{any} \wedge \text{departure} > \text{2010-05-23 00:00Z} \\ &\wedge \text{origin} = \text{BCN} \wedge \text{destination} = \text{JFK} \wedge \text{price} \leq 800.0), \end{aligned}$$

which is equivalent to the filter in Fig. 2(b).

2.5 Publish/subscribe architectures

There are several alternatives for the internal architecture of a pub/sub system [EFGK03]. The most straightforward one is to implement a *centralized* system, where the system consists of a specific entity that takes care of managing subscriptions and delivering event notifications. This approach is adequate, if there are no stringent scalability or performance requirements for the system but instead a high degree of reliability, data consistency or transactional support is needed.

A pub/sub system can also be fully *distributed*, where there is no centralized entity. Instead, the event service's store and forward mechanisms are implemented by the producer and consumer processes themselves. Fully distributed architecture is suited for fast delivery of transient data such as multimedia broadcasting.

In this work, we are interested in an intermediate approach, used by Siena (among many others): the pub/sub service is provided by a set of dedicated servers (sometimes also called *event brokers*) that work in a distributed fashion. This removes the burden of implementing the delivery logic from the subscribers and publishers. A subscriber S usually interacts with only one server at a time. This server is called the *local server (of S).*

There are different ways to organize the topology of the dedicated servers [CRW01]. The most straightforward of them is the *hierarchical architecture*, where server-to-server connections are asymmetric; a server acts as a client to the one ("master server") that is higher in the hierarchy. A server may have several "client" servers but at most one master server. The server that has no master server is the *root server.* Subscriptions, advertisements and notifications are ultimately forwarded to the root server, making the hierarchical architecture essentially an extension of the centralized one. This approach has two major shortcomings. The load on the servers that are high in the hierarchy can easily become too high. Furthermore, every server is a single point of failure as a failure in one server disconnects a whole subtree from the network.

The first, but not the second, problem of the hierarchical architecture can be overcome by using *acyclic peer-to-peer architecture.* In such architecture, servers interact as peers, allowing the flow of advertisements and subscriptions in both directions. The acyclic property of the topology allows for relatively simple routing algorithms [CRW01]. In this approach, each server is still a single point of failure as there is only one path in the network from a server to another. Also, maintaining the acyclic property in a wide-area service may be costly.

In the *generalized peer-to-peer architecture*, the acyclic property is not needed; the servers form a general undirected graph. General peer-to-peer architecture is more fault tolerant and flexible and requires less coordination than the other approaches.

The flexibility comes with the price of more complex routing: the algorithms must avoid cycles and possibly choose the optimal paths, causing the need for example for finding minimal spanning trees in the network. In subsequent sections, we discuss routing only in a generalized peer-to-peer architecture, unless stated otherwise.

Also hybrid architectures are possible [CRW01]. For example, the backbone of a pub/sub system may be organized in general peer-to-peer setting, but each event broker in the backbone network can actually be the root server of its own subnetwork. This organization is analogous to approaches used in traditional networking.

3 Content-based networking

Implementing content-based publish/subscribe middleware benefits from a specialized communication infrastructure that supports the interface of the middleware. This infrastructure is referred to as a *content-based network* [CW02]. In this and the following section, we discuss the properties and strategies of content-based networking: addressing, forwarding and routing. We also give a formal definition of a content-based filter and covering relations between filters as these relations are important in content-based routing (for instance, the concepts of content-based subnetting and supernetting follow immediately from the covering relation between filters).

Current content-based networks are application-layer *overlay networks* built on top of an existing TCP/IP infrastructure (however, as mentioned before, there is no theoretical obstacle for implementing a content-based network on the network layer). A content-based network consists of router nodes that are connected by point-to-point communication links. Formally, we treat a content-based network as a undirected general graph (which implies generalized peer-to-peer architecture), where router nodes form the set of vertices and there is an edge between vertices if and only if the corresponding router nodes are connected by a point-to-point link.

It should be noted that the fact that two router nodes are connected in the overlay network does not imply that they are connected (or even topologically close to each other) in the physical network. Client nodes are connected to the router nodes. Thus, a router acts as a local server for a set of clients and is responsible of delivering notifications to those clients and handling subscriptions made by them. A client is usually connected to one router at a time. A client acts as a subscriber or a publisher (or as both).

A router in a content-based network performs two functions, *routing* and *forwarding* [CW03]. The purpose of routing is to form forwarding tables based on the topological features of the overlay network and the subscriptions and advertisements issued by the clients. The forwarding function uses the information provided by the routing function to determine the set of next-hop destinations (either neighboring routers or client nodes) for an event notification. We discuss these functions in detail in Sections 3.3 and 4.

3.1 Filters

Formally, a filter is a stateless boolean function

$$F : N \rightarrow \{true, false\},$$

where N is the set of all notifications [Müh02]. A notification $n \in N$ is said to *match* F if $F(n) = true$. Further, we let $N(F)$ denote the set $\{n \in N | F(n) = true\}$, i.e. the notifications that match F. Filters F_1 and F_2 are *identical*, written $F_1 \equiv F_2$, if and only if $N(F_1) = N(F_2)$. In terms of our data model, a filter is considered a conjunction of attribute constraints and a notification matches a filters if its attributes satisfy every attribute constraint of the filter. It should be noted that Siena follows this model only in subscription filters; advertisement filters in Siena are disjunctions of attribute constraints [CRW01, Müh02]. In this work, we consider the subscription and advertisement filter models identical – Siena-like functionality in advertisements can be achieved by allowing multiple values or value ranges in an attribute constraint (see Section 2.4). If an attribute A present in notification n is not present in a filter F that is being evaluated for n, it considered that F implicitly contains the constraint $A = any$. If a filter contains an attribute constraint that refers to an attribute not present in the notification being evaluated, then that constraint is not satisfied (and thus the notification does not match the filter). The notification in Figure 2(a) is an example of a notification that matches the filter in Fig. 2(b).

Filters F_1 and F_2 *overlap*, written $F_1 \sqcap F_2$, if and only if $N(F_1) \cap N(F_2) \neq \emptyset$. Further, a filter F_1 *covers* filter F_2, denoted $F_1 \sqsupseteq F_2$, iff $N(F_1) \supseteq N(F_2)$. In other words, F_1 is *more general* than F_2. If $F_1 \not\sqsupseteq F_2$ and $F_2 \not\sqsupseteq F_1$, the filters F_1 and F_2 are said to be *unrelated*. Note that using this terminology, two filters may overlap and still be unrelated. For example, consider the following filters:

$$\begin{aligned}
F_1 &= (\text{price} < 300.00 \wedge \text{price} > 400.00),\\
F_2 &= (\text{price} < 200.00 \wedge \text{price} > 500.00),\\
F_3 &= (\text{price} < 450.00 \wedge \text{price} > 600.00),\\
F_4 &= (\text{price} \notin [300.00, 400.00]).
\end{aligned}$$

For these filters, the following statements are true:

$$\begin{aligned}
&F_1 \equiv F_4,\\
&F_1 \sqsupseteq F_2,\\
&F_2 \sqcap F_3,\\
&F_2 \not\sqsupseteq F_3 \wedge F_3 \not\sqsupseteq F_2.
\end{aligned}$$

Furthermore, let $\mathcal{F}$ denote some set of filters. Now, the *predecessors* of a filter $G \in \mathcal{F}$ are defined as the set $Pred(G) = \{F \in \mathcal{F} | F \neq G \wedge F \sqsupseteq G\}$. The *successors* of G are defined similarly: $Succ(G) = \{F \in \mathcal{F} | F \neq G \wedge G \sqsupseteq F\}$. If $F \sqsupseteq G$ and there is no $F' \in \mathcal{F}$ such that $F \sqsupseteq F'$ and $F' \sqsupseteq G$, we say that F *immediately covers* G and denote it with $F \succ G$. Now we get the set of *immediate predecessors* and *immediate successors* of a filter G as $ImPred(G) = \{F \in \mathcal{F} | F \neq G \wedge F \succ G\}$ and $ImSucc(G) = \{F \in \mathcal{F} | F \neq G \wedge G \succ F\}$.

The covering relation $\sqsupseteq$ is reflexive, transitive and antisymmetric and thus defines a *partial order* over the set of all filters [TK06]. A set with a partial order is called a *partially ordered set* or *poset*. Poset-based data structures and the covering relation are important when implementing covering- or merging-based routing. These routing strategies and poset data structures are discussed in detail later.

3.2 Addressing

We now define the implicit content-based addressing scheme. In this, we follow the concepts introduced by Carzaniga and Wolf [CW02]. In a content-based network, each node (client or router) advertises a *receiver predicate* (*r-predicate*) that defines the set of datagrams the node is interested in receiving. Optionally, a node may advertise a *sender predicate* (*s-predicate*), which defines the datagrams the node intends to send. The *content-based address* of a node n is its r-predicate p_n.

Note that contrary to a conventional network address, the content-based address of a node is implicit and that the rate at which a content-based of a node address changes

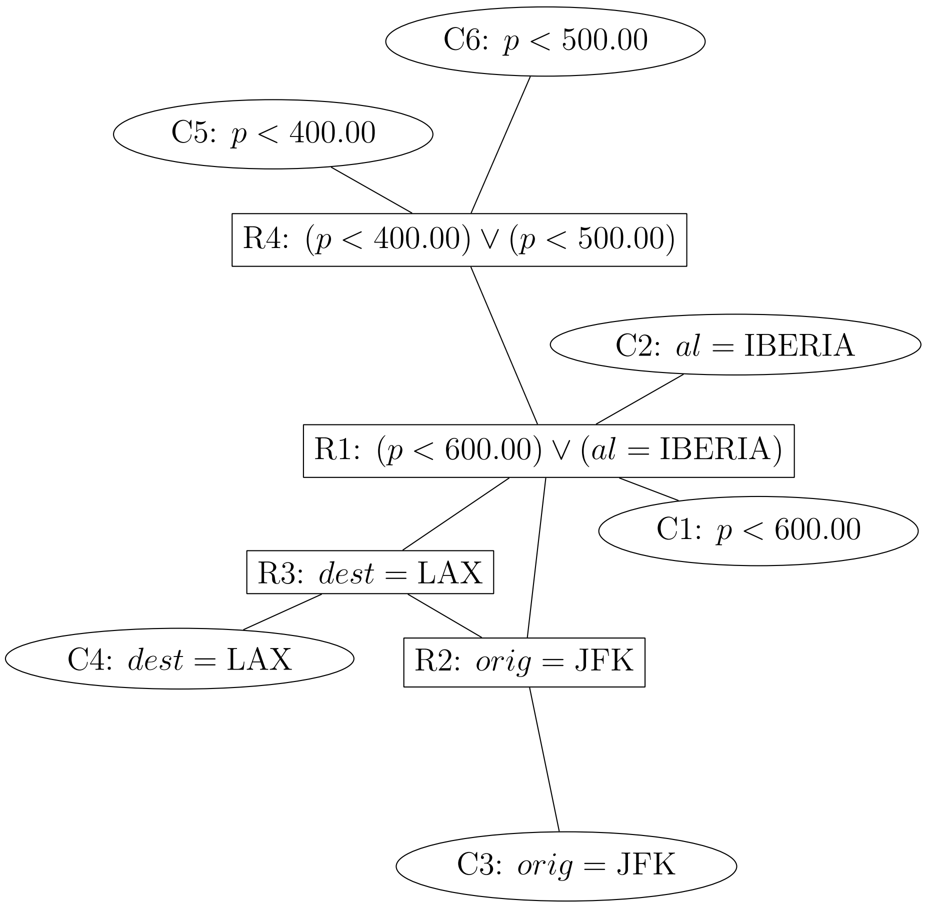


Figure 3: A simple content-based network with four routers (R1–R4) and six clients (C1–C6). The content-based addresses of the nodes are represented by filters.

can be several orders of magnitude faster than with traditional addresses. Also, several nodes may have the same content-based address. Nodes are not assigned any unique addresses in a content-based network (however, their network-level addresses can be used as unique identifiers in routing and forwarding protocols [CW03]).

The model presented by Carzaniga and Wolf is generic; it does not fix the format of datagrams and predicates. For the purposes of this work, we consider the structured records presented in Section 2.4 our *datagram model*. As our *predicate model*, we use the string-based filter language and semantics defined in Sections 2.4 and 3.1. We use the same predicate model for both r- and s-predicates. Using these models, the content-based address (r-predicate) of a node is defined as

- Set of notifications defined by disjunction of all its subscription filters, if the node is a client node.
- Set of notifications defined by disjunction of all subscription filters of clients connected to it, if the node is a router node.

Similarly, the s-predicate of a node is either disjunction of its advertisements (for client nodes) or disjunction of its clients' advertisements (for router nodes). Also note that identical filters $F_1 \equiv F_2$ make for the same abstract predicate p; they are merely different *representations* of the same predicate. In other words, when we refer to a filter F as a r- or s-predicate, we actually mean the set $N(F)$ as it is unambiguous. As long as this is kept in mind, these terms can be used interchangeably without loss of precision. An example of a content-based network and addressing is shown in Figure 3.

Defining the content-based address of a router node as above allows us to abstract away the client nodes and only examine networks of routers. We can assume it is sufficient that a datagram meant for a client be delivered to local router of that client; the router can then relatively easily forward the datagram to its final destination. Therefore, in subsequent examples we often leave out the client nodes from the network and consider only routers.

In traditional networking, a *subnet* is a set of nodes with similar addresses, usually topologically close to each other. Subnets are identified by a single address, allowing routers to treat a subnet as a single entity, greatly improving scalability. Routers also attempt to do *supernetting*, i.e. combining subnets into larger subnets while executing their routing protocols. These principles can be applied also in content-based networking. Let filters F_1 and F_2 denote representations of r-predicates p and q, respectively. If $F_2 \sqsupseteq F_1$, then p is a *content-based subnet* if q and q is a *content-based supernet* of p. For example, in Fig. 3, router R4 could advertise $p < 500$ as the subnet of clients that can be reached through it, because $N(p < 500)$ is the content-based supernet of $N(p < 400)$.

3.3 Forwarding

The purpose of the forwarding function is to determine the set of next-hop destinations for a datagram that has arrived to a router [CW02]. The set of next-hop destinations may contain both adjacent routers and client nodes connected to the router. The router computes the next-hop destinations based on the datagram content and its *forwarding table*. The forwarding table is a data structure internal to the router. It is compiled and updated by the routing function. Conceptually, content-based forwarding and routing tables are different entities; the former may be optimized for fast matching while the latter is optimized for efficient update and remove operations [CRW04]. The forwarding table is then periodically updated or rebuilt by the routing function, based on the topological data it has gathered. On the implementation level, however, routing and forwarding tables may be implemented by the same data structure. Figure 4 shows an example of a content-based forwarding table.

Interface	*Node ID*	*Address*
I_0	*local*	$(p < 600.00) \vee (al = \text{IBERIA})$
I_1	R2	$orig = \text{JFK}$
I_2	R3	$dest = \text{LAX}$
I_3	R4	$(p < 400.00) \vee (p < 500.00)$

Figure 4: A content-based forwarding table maintained by router R1 (see Fig. 3).

Formally, the forwarding table can be interpreted as a map from the set interfaces of the router, to the set of r-predicates (content-based addresses) [CW02]. The interfaces of the router represent the neighboring nodes. Usually, the client nodes directly connected to the router are treated as a single *local interface* I_0 for convenience [CW03]. Thus, performing the forwarding function is equal to finding the set of next-hop interfaces for incoming datagram d, that is, to deliver it to any interface whose r-predicate contains the datagram d.

Usually, the data in a content-based forwarding table is used together with some *broadcast forwarding function* that is applied to the set of interfaces yielded by the content-based forwarding function to further limit the set of next-hop destinations in order to avoid forwarding loops [CW03]. That is, an incoming datagram is forwarded only to nodes that are on a broadcasting path from the source. The broadcasting path is defined by the used broadcast function; for example, it may be determined by a spanning tree rooted at the source of the datagram.

The router usually forwards incoming datagrams sequentially and in FIFO order. As the throughput of the forwarding function determines the throughput of the router (as is with conventional network-level routers), it is important that the routing function be computed efficiently. Optimizing the forwarding function a joint effort between forwarding algorithm and routing function improvements [CW02]. The forwarding throughput can be improved by developing faster matching algorithms or more efficient forwarding table data structures. Having the routing function to produce smaller routing tables and to reduce the amount of unnecessary traffic by implementing better routing protocols also affects the throughput of the router. In Section 4, we discuss the methods of producing small forwarding tables and designing efficient routing protocols, but first we briefly mention some examples of research done on forwarding algorithms.

Early research related to the forwarding problem was conducted in the context of event matching in centralized pub/sub systems [CW03]. In this kind of setting, the straightforward approaches would be either to flood all events to all subscribers (effectively making subscribers responsible of filtering) or to match all subscriptions against the event one by one [BCM+99]. As neither of these approaches scales well, it is evident that the subscription filters (or in more general setting, the content-based addresses associated with the interfaces of the routers) need to be stored in a more advanced data structure.

The basic approach for more efficient event matching and forwarding table storage is the *counting algorithm* [YGM94] that keeps track of count of satisfied constraints in partially matched filters. The counting algorithm has later been extended by for example Carzaniga and Wolf [CW03]. The algorithm due to Carzaniga and Wolf uses the forwarding table as a dictionary-type data structure that is optimized for searches but not for updates. The data structure is indexed by attributes that are present in the filters in the forwarding table. For each of the attributes in an incoming event notification, the index is searched for satisfied constraints that involve the attribute in question and the count of satisfied constraints in a incremented for each conjunction the constraint is present. Because filters associated with interfaces are disjunctions of conjunctions, a filter can be added to the forwarding set if any of the conjunctions in the filter is satisfied. In their work, Carzaniga and Wolf also presented further improvements in their forwarding algorithm.

The algorithm of Carzaniga and Wolf represents the approach where the attributes of a notification are used as the starting point and find the matching constraints based on them. This approach is also used by Fabret et al. [FJL+01]. An opposite approach, used in the tree matching algorithm due to Aguilera et al. [ASS+99] and binary decision diagrams due to Campailla et al. [CCC+01], is to start from the attribute constraints present in the forwarding table [CW03].

4 Content-based routing

The purpose of content-based routing is to compile and maintain the content-based forward tables in routers. In order to do this, a router must exchange routing information with adjacent nodes and use that information to maintain a *content-based routing table*. The routing tables are used to create a forwarding state in which the paths for notifications are set by subscriptions. A good routing strategy results in compact forwarding tables (and thus to more efficient forwarding) and aims to minimize unnecessary network traffic in terms of propagating subscriptions.

A good routing strategy should also follow the principles of *downstream replication* and *upstream evaluation* [CRW01]. Downstream replication means that forwarding state set by the routing function should be such that notifications are forwarded in one copy as far as possible and replicated downstream, as close as possible to the subscribers. The upstream evaluation principle states that filters are applied upstream, as close as possible to the publishers in order to stop the propagation of a notification towards uninterested parties as early as possible. Thus, the subscriptions should be pushed close to the publishers.

A routing scheme may be further improved by the use of advertisements that effectively set the paths for subscriptions. The internals of a content-based routing table are discussed in Section 5. In this section, we discuss different routing algorithms and their impact on the performance and scalability of a content-based pub/sub system. We do not, however, address fault-tolerance in these schemes in detail. We start by presenting some simple routing strategies and move on to more advanced ones, including those that make use of advertisements. As a concrete example, we look at the *CBCB routing scheme*, due to Carzaniga et al. [CRW04].

We use the following notation. Let F be a filter and X be an unique identifier (e.g. IP address) of a node (either a router or a client). Now, the message $\mathit{subscribe}(F, X)$ is defined as a subscription to notifications matching F, issued by node X. Messages $\mathit{unsubscribe}(F, X)$, $\mathit{advertise}(F, X)$ and $\mathit{unadvertise}(F, X)$ are defined similarly. Further, we let T_R denote the routing table of router R. A routing table entry is a pair $(F, X) \in T_R$, where F is a filter and X is a node identifier, i.e. the destination where notifications matching F should be forwarded to.

4.1 Simple routing

In *simple routing*, each router in the network keeps track of every subscription in the system [Müh02]. The scheme requires flooding subscriptions to the whole network and is realized by the following strategy:

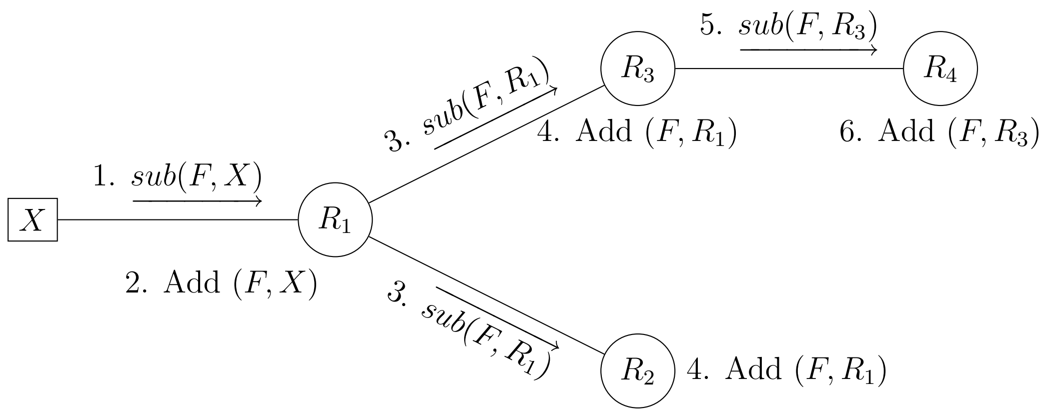
- When router R receives a message *subscribe*(F, X), where X can be either a neighboring router or a local client:
  - The pair (F, X) is added to the routing table if it is not yet present there.
  - R sends a new message, *subscribe*(F, R), to all neighboring routers except X.

Unsubscriptions are handled in similar fashion. An example scenario of simple routing is shown in Fig. 5.

If the network topology is acyclic, simple routing minimizes the notification traffic in the network. In generalized topologies, the same effect can be achieved by using some broadcasting function instead of flooding. This scheme is suitable for small networks or for networks where subscriptions are relatively static. In large-scale networks, however, the administrative traffic (subscriptions and unsubscriptions) amounts to too large part of the network traffic. Also, the forwarding tables in this routing scheme tend to grow quite large. Obviously, a router needs to do more complex processing of a (un)subscription message in order to limit the set of neighbors it is forwarded to. Thus, there is trade-off between bandwidth usage and processing overhead.

4.2 Identity-based routing

Identity-based routing is a simple improvement over simple routing. In simple routing, the routing tables may contain redundant entries. Let F and G be identical filters, i.e. $F \equiv G$. A simple routing table might thus unnecessarily contain entries (F, X) and (G, X) when either one of them alone would suffice to forward a matching notification to X. In identity-based routing, a subscription is not forwarded to a neighboring router if an identical subscription has been forwarded to that router before [Müh02]. This involves some processing overhead due to filter identity testing. On a high level, the protocol works as follows:



X
R_1
R_2
R_3
R_4
1. $\xrightarrow{sub(F,X)}$
2. Add (F,X)
3. $\xrightarrow{sub(F,R_1)}$
3. $\xrightarrow{sub(F,R_1)}$
4. Add (F,R_1)
4. Add (F,R_1)
5. $\xrightarrow{sub(F,R_3)}$
6. Add (F,R_3)

Figure 5: An example of simple routing. The subscription issued by client X for some filter F is propagated to all routers in the network and corresponding routing table entries are added.

- When router R receives a message $subscribe(F, X)$, where X can be either a neighboring router or a local client:
  - Form a set $D = \{Y|(G, Y) \in T_R \wedge F \equiv G\}$, that is, the set of nodes who have issued a subscription using an identical filter and the subscription has not been cancelled. After this, add (F, X) to T_R if X is a client. This step is needed for unsubscriptions, as explained later.
  - If X is in D, do nothing, because an identical subscription has already been forwarded to all neighbors.
  - Else if D is an empty set, add (F, X) to the routing table (if X is a router) and send message $subscribe(F, R)$ to all neighboring routers except X.
  - Else, add (F, X) to the routing table (if X is a router) and do for each node Y in D: If N is a router, send $subscribe(F, R)$, unless there exists an entry (Z, G) in T_R for which $Y \neq Z$ and $F \equiv G$, because existence of such entry implies that an identical subscription has been forwarded to Y before (when it arrived from Z).

An example scenario is shown in Figure 6.

With identity-based routing, handling unsubscriptions requires some processing. If a local client X of router R issues an unsubscription for filter F, the entry (F, X) is removed from T_R. An unsubscription message is propagated to neighboring routers only if no local client has an outstanding subscription with identical filter. In prop-

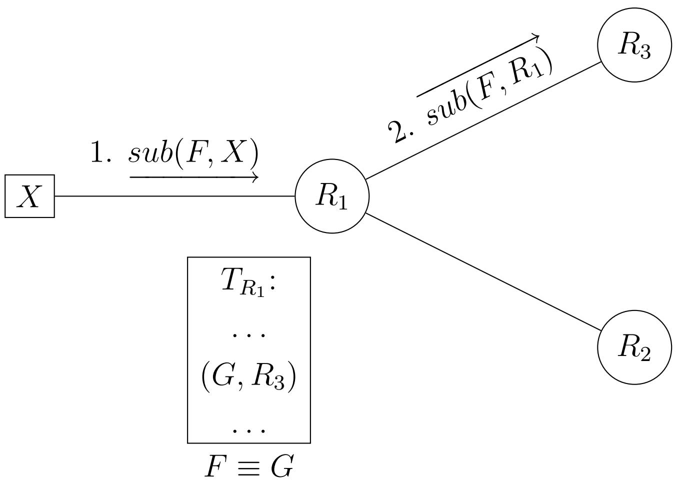


Figure 6: An example of identity-based routing. Router R_1 does not forward the subscription to R_2, because it has received a subscription with identical filter G from R_3 at some point in the past. Thus, it has already forwarded that subscription to R_2.

agating the unsubscriptions, an operation inverse to the one used in subscription propagation is used.

4.3 Covering-based routing

Exploiting the covering relation $\sqsupseteq$ instead of the identity relation $\equiv$ is an obvious improvement, utilized by *covering-based* routing approach [CRW01, Müh02]. This approach further reduces redundancy in routing table entries. The basic idea and strategy in subscription propagation is to some extent the same as in identity-based routing, the major difference being covering testing instead of identity testing: a subscription is not forwarded to a neighbor if a more general subscription (that is still outstanding) has been forwarded to that neighbor before. If a set of subscriptions becomes covered by a new subscription and the new subscription is not covered by some other subscription, the router adds the subscription filter as a *root filter* in its routing table [CRW01]. The covered subscriptions are retained in the routing table or a separate data structure, such as the Siena *filters poset* [CRW01] or *poset-derived forest* [TK06].

Using covering-based routing comes with slightly added complexity in unsubscription handling when compared to identity-based routing. When an unsubscription of filter F is issued by neighbor X, the router forwards the unsubscription message to its neighboring routers if the filter F associated with the unsubcription message is a root filter. If F is not a root filter, then there still are some outstanding subscriptions that cover F and thus no action except removing (F, X) from the poset structure is required. Now, it is not sufficient that the router forward only the unsubscription. With the forwarded message, the router must also pass the (possibly empty) set $ImSucc(F)$, i.e. the immediate successors of F, because otherwise these subscriptions would be left uncovered once F is removed from the routing table. This is why the router needs to keep track of covered subscriptions also. This is not the only way of handling unsubscriptions in covering-based routing, as we will see when discussing the CBCB routing scheme.

4.4 Merging-based routing

Merging-based routing is not a separate routing scheme. Rather, it can be implemented on top of any of the routing algorithms described above, although it is usually coupled with covering-based routing. An example of this is the Rebeca system [Müh02].

In merging-based routing, instead of propagating subscription filters, routers may propagate *merger filters* (mergers) that are composed of several filters associated with the same destination node. The purpose of this is to further reduce subscription traffic in the network. A merger is said to be *perfect* if it does not cover any subscriptions that the filters used to create it did not cover. Otherwise, the merger is called *imperfect*.

Using perfect mergers is preferable, because imperfect mergers cause *false positives*, notifications that are forwarded to a node even if the node is not interested in that particular notification. For example, merging filters ($price \in [50, 100]$) and ($price \in [110, 150]$) into ($price \in [50, 150]$) would produce an imperfect merger, because it would accept also notifications, where $price \in (100, 110)$, in which no subscriber was originally interested. On the other hand, filters ($price \in [50, 100]$) and ($price \in [90, 120]$) would produce a perfect merger ($price \in [50, 120]$).

Any implementation of merging-based routing must answer at least the following questions [Müh02]:

- When and how should filters be merged or mergers be cancelled?

- To which neighbors should the mergers and their cancellations be forwarded?
- How should the mergers be administered?

Any practical filter merging algorithm is an approximation at best; it has been shown that optimal filter merging is an NP-complete problem [CBGM03]. Some research has been done on filter merging frameworks that allow integrating merging-based routing into content-based routers in a transparent way [TK05] and on merging algorithms and mergeability detection [Tar08].

4.5 Advertisement-based routing

Similar to merging-based routing, *advertisement-based routing* can be deployed to enhance the performance of any routing scheme presented here [Müh02]. As noted before, advertisements are filters identical to those used in subscriptions. Using advertisements, a client X may define the set of events they intend to generate by sending an $advertise(F, X)$ message with some filter F. (Un)advertisements are propagated through the network using any of the routing schemes presented above [Müh02]. For advertisements, the routers maintain an *advertisement routing table* separate from the subscription routing table described above. The advertisement routing table entries and structure are the same as in subscription routing tables.

While the subscription routing table is used to create forwarding tables for notification traffic, the advertisement routing table has a similar function for subscriptions. Subscriptions are routed only along the reverse path of advertisements (while also possibly being subject to other forwarding restrictions imposed by routing schemes discussed above). An (un)subscription message for filter F and source X is forwarded (at most) to those neighbors for which there exists an entry (G, Y) in the advertisement-based routing table, where $X \neq Y$ and F overlaps with G. Note that overlap check is indeed needed instead of just identity or containment checks, because an overlap implies that the neighbor Y may be a potential source of some (if not all) events the subscriber is interested in. Also, the fact that subscriptions are only forwarded towards advertisement issuers implies that advertising is not optional; the publishers must issue an advertisement before publishing any content (and the published content must comply to the issued advertisement).

4.6 CBCB routing scheme

As an example of a content-based routing scheme, we use the *combined broadcast and content-based* (CBCB) routing scheme, due to Carzaniga et al. [CRW04]. It is essentially a covering-based routing scheme in which also merging and advertisements can be supported easily. CBCB is a *two-layer* routing scheme; in addition to a content-based routing protocol, the router runs a *broadcast routing* protocol. A broadcasting protocol is needed to maintain a forwarding state that allows for sending messages from a node to all other nodes in the network. Because the underlying network topology forms a general graph and thus may contain cycles, simple flooding cannot be used. The purpose of the content-based layer in the scheme is to prune the broadcast trees by utilizing the content-based data. As there are several well-known ways to implement a broadcasting function (such as minimal spanning trees, shortest-path trees or reverse path forwarding), we assume one is available and do not address the broadcasting part of the routing scheme in further detail.

In the following, we consider a network of router nodes (or simply ”nodes” from now on), each of which has a content-based address (r-predicate) p_n, where n is an identifier unique to each node. As explained in Section 3.2, the clients can be abstracted away from this model by defining the content-based address of a router as a disjunction of its local clients' content-based addresses. As usual, it can be considered that the predicates are represented by filters and the actual content-based address is the set of matching notifications defined by that filter.

The content-based layer of the CBCB scheme employs a push-pull mechanism. The nodes push routing data to their neighbors by using *receiver advertisements* (RAs). Further, the nodes can request (pull) routing information from the network by sending out *sender requests* (SRs) and waiting for *update replies* (URs). This process is referred to as the *SR/UR protocol*.

4.6.1 Receiver Advertisements

Receiver advertisements are the primary vessel of propagating content-based routing data in the CBCB scheme. An RA is issued by a node whenever its content-based address changes (basically when one of its clients issues an (un)subscription). RAs can also be sent out periodically. An RA can be represented as the pair (n, p_n), where n is a node identifier and p_n the content-base address of node n. In practice, an RA may also contain additional fields, such as timestamps. RAs should not be confused with advertisements discussed earlier; an RA is analogous to a subscription whereas an advertisement indicates what type of content a publisher intends to produce. Advertisements could, however, be integrated into the CBCB scheme quite easily.

When a router receives an RA (n, p_n) through interface i, it processes the message as follows. If there is a predicate p_i associated with interface i in the routing table, meaning that notifications matching p_i should be forwarded to interface i (recall that the set of local clients is treated as a single interface also), then the router performs a containment check: if p_n is already covered by p_i, the router simply drops the RA. This is called the *RA ingress filtering rule*. Otherwise, the router uses the broadcasting function to compute the set of next-hop destinations on the broadcast tree rooted in n. The router also updates its routing table by setting the predicate associated with interface i as $p_i := p_i \vee p_n$. Some filter merging algorithm may also be utilized in this step.

4.6.2 The SR/UR protocol

Relying solely on receiver advertisements leads to *inflation* of content-based addresses. Consider a case where a router r has associated a predicate p_i with interface i. When the router receives an RA through interface i, it applies the ingress filtering rule to it and drops the RA if its predicate is already covered by p_i. However, the reason the router received a predicate already covered may be due to an unsubscription. Now, because the RA was dropped, the neighboring routers still forward all notifications matching p_i to r, even if no one in the subnet behind interface i is interested in part of them anymore. In order to avoid excessive address inflation and false positives, the routers periodically send out sender requests. An SR contains the identifier of its issuer, an *SR number* and a timeout field. The SR number is used to differentiate between several SRs from the same issuer. The timeout indicates how long the issuer is going to wait for a reply.

An SR issued by a router r is broadcast to all routers. Any router that receives an SR estimates a new timeout for it (the estimation details are omitted here) and forwards it downstream on the broadcasting tree. If r is a leaf router, it immediately responds with an update reply. A UR consists of the SR issuer identifier, the SR number the UR is reply for, and a predicate. In the case of a leaf node, the predicate it puts in the UR is its own content-based address, i.e. the predicate p_0 associated with its local interface 0. A non-leaf node must wait for URs from all interfaces it forwarded the SR. When the node has received all URs (or a timeout occurred), it sends an UR with the predicate set to disjunction of predicates in the received URs and sends it along the reverse path of the SR.

The broadcast nature of the SR/UR protocol implies that the amount of control traffic in the network may grow too large if all routers issue SRs on regular basis. Carzaniga et al. proposed some improvements for the protocol. The basic version of the SR/UR protocol only allows for the original SR issuer to use the resulting URs in updating the routing table. In general, this behavior is needed, because the URs triggered by an SR are specific to the broadcast tree rooted at the SR issuer. However, in certain cases an intermediate router may be allowed to use an UR to update its own routing table and/or cache an UR so that it can be used the next time a similar SR is issued, greatly reducing traffic caused by the protocol. Additionally, the use of SRs could be limited – instead of broadcasting SRs periodically, the routers could send them only to a subset of neighbors and only when needed.

4.7 Conclusions

We have discussed some possible schemes for content-based routing: simple, identity-based and covering-based. Each of these schemes can be augmented with filter merging and/or advertisements. We also noted that there exists a trade-off between the amount of control traffic (subscriptions and advertisements) in the network and processing time needed in a router: by performing e.g. identity or covering checks on the filter associated with an incoming subscription, a router may be able to stop the propagation of that subscription. Furthermore, if the routers work in a general topology, any routing scheme must be combined with a broadcasting function in order to avoid redirect loops. As an example of this, we discussed the CBCB routing scheme.

In the remainder of this section, we briefly mention some work on done on performance analysis and fault-tolerance of content-based routing. In a series of tests conducted by Mühl et al. [MFGB02], it was concluded that advanced routing algorithms (i.e. covering and merging) should be considered mandatory in large-scale systems; otherwise the routing tables and amount of control traffic grows too large. Further, using advertisements in any of the schemes greatly improves scalability compared to the same scheme without advertisements. Also, in practical applications, the routing schemes benefit from the effects of *locality*. The range of subscription is usually not uniformly distributed over the network. Rather, subscribers close to each other commonly have similar interests, allowing the routers to do more efficient merging and covering-based pruning of subscriptions.

It should be noted that Mühl et al. did not test the CPU overhead caused by more complex filter handling in the more advanced routing schemes. The amount of processing required at routers is significant in covering-based routing [TK06, Tar07], which is a limiting factor in terms of scalability. The main goal of this work is to reduce that overhead. In Section 5, we discuss some data structures needed for implementation of covering-based routing and in Section 6, we present our method for reducing the processing overhead.

None of the routing schemes presented above are fault-tolerant, with possibly the exception of CBCB routing scheme, which can recover from link and node failures (if the overlay network forms a general graph in which there are several paths from one node to another) provided that RAs and SRs are issued periodically rather than just per need. Fault tolerance (as well as such topics as congestion control) is mostly outside the scope of this work, but we mention some work here. One proposed approach that is considered to be fault-tolerant, is using *soft states.* A soft state can be defined as a state that can be lost due to a failure without causing any permanent disruption of service. Work on implementing a soft state in pub/sub systems has been done e.g. in context of the Hermes system [Pie04] and by Jerzak and Fetzer [JF09]. Furthermore, Mühl [Müh02] has done work on formalizing fault-tolerance requirements, such as *self-stabilization* in pub/sub systems. Mühl also proposed a practical fault-tolerant routing scheme based on *subscription leasing*, i.e. an arrangement where subscriptions are considered leased, implying that they need to be renewed periodically.

5 Posets and related data structures

In this section, we discuss methods of representing partially ordered sets (posets) used in content-based routing tables. In particular, we focus on the Siena *filters poset* and *poset-derived forests.* The Siena filters poset [CRW01] is a directed acyclic graph (DAG) based data structure that stores filters (and corresponding subscriptions or advertisements) by their immediate covering relations. The poset-derived forest and its variants [TK06] do not keep track of all covering relations, but instead aim at providing fast operations for insertions, deletions and computing the *root set* (the set of filters that cover all other filters) of the filters in the data structure.

5.1 Siena filters poset

The Siena filters poset (FP) is a data structure used as the content-based routing table of Siena routers [CRW01]. In the following, we consider posets for subscriptions; advertisement posets are almost identical to the subscription ones. An FP consists of a set of filter entries and links between them.

5.1.1 Definition

Let $\mathcal{F}$ denote the set of subscription filters stored in the FP of some router. In an entry for filter F, two lists of pointers to other entries are maintained: one for $ImPred(F)$ and another for $ImSucc(F)$. As defined in Section 3.1, the set $ImPred(F)$ is the set of filters in $\mathcal{F}$ that are immediate predecessors of F, i.e. filters that immediately cover F. Similarly, $ImSucc(F)$ is the set of filters in $\mathcal{F}$ immediately covered by F. The set of filters for which $ImPred$ is an empty set, is called the *root set* (also *non-covered set* or *minimal cover set*) of the poset. The FP can be visualized as a directed acyclic graph, where the filters in $\mathcal{F}$ act as nodes and there is a directed edge from F to G if and only if $G \in ImPred(F)$. Fig. 7 shows an example of an FP.

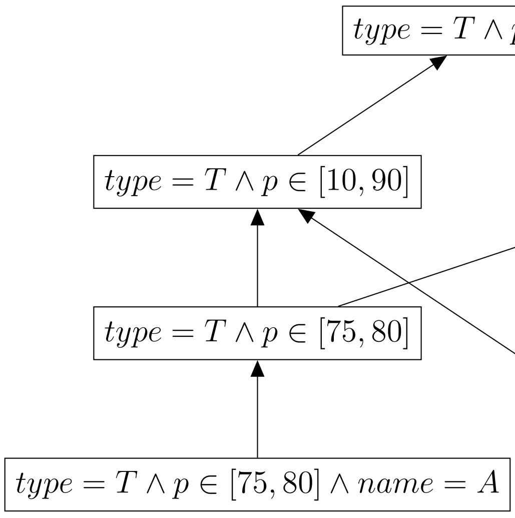

$type = T \wedge p \leq 100$
$type = T \wedge p \in [10, 90]$
$type = T \wedge p \in [70, 95]$
$type = T \wedge p \in [75, 80]$
$type = T \wedge p \in [90, 95]$
$type = T \wedge p \in [75, 80] \wedge name = A$
$type = T \wedge p = 90$

Figure 7: An example filters poset. The directed edges reflect the immediate covering relations between the filters.

In addition to the links to other entries, there are two sets associated with a filter entry. The set $subscribers(F)$ is the set of interfaces through which the router has received a subscription with filter F (or an identical filter). The set $forwards(F)$

defines the set of interfaces towards which F needs to be forwarded. It is usually not stored with the entry rather than computed per need (that is, when an (un)subscription for F is received over some interface). The $forwards(F)$ set is defined as [CRW01]

$$forwards(F) = neighbors - NST(F) - \bigcup_{F' \in \mathcal{F} \wedge F' \neq F \wedge F' \sqsupseteq F} forwards(F'), \qquad (1)$$

where *neighbors* is the set of all interfaces of the router in question. The term $NST(F)$ stands for ”Not on any Spanning Tree” and makes sure that in general topologies, only interfaces downstream in spanning trees rooted at original subscribers of F are taken into consideration in order to avoid forwarding loops, similarly to the CBCB routing scheme discussed above. In acyclic topologies, the set $NST(F)$ consists only of the interface through which the subscription was received, since the network itself already is a spanning tree. The last term of (1) simply formalizes the idea of covering-based routing: F is not forwarded to interfaces to which a more general subscription F' has been forwarded before.

5.1.2 Subscriptions and unsubscriptions

Whenever a Siena router receives a subscription for filter F through interface X, it searches its filters poset $\mathcal{F}$ for one of the following:

1. A filter F' that covers F and for which $X \in subscribers(F')$. If such filter is found, no further actions are needed, as the subscription has been handled before.

2. A filter F' for which $F' \equiv F$ and $X \notin subscribers(F')$. If such filter is found, X is added to $subscribers(F')$ and removed from all filters covered by F'.

3. The possibly empty sets $ImPred(F)$ and $ImSucc(F)$. The router adds F to $\mathcal{F}$ as a new filter between these sets, adds X to $subscribers(F)$ and removes X from all filters covered by F. The router also forwards the subscription to interfaces in possibly empty set $forwards(F)$.

In cases 2 and 3, when an interface is removed from the *subscribers* set of an entry covered by F, the entry in question is also deleted from $\mathcal{F}$ if the removed interface was the only one in its *susbcribers* set.

As discussed in Section 4.3, handling of unsubscription has slight complexities in covering-based routing. An unsubscription may cancel more than one subscription at a time, if it is issued with a filter that covers many subscriptions from the same source. Further, an unsubscription may cancel a root subscription and may uncover some subscriptions that need to be forwarded to the neighbors. When an unsubscription of filter F is received through interface X, a router takes following actions. First, the set $f_{old} = forwards(F)$ is computed. Then, X is removed from all entries covered by F and a new, possibly empty, forwards set f_{new} is computed for F. The unsubscription is forwarded to interfaces in the set $f_{old} \setminus f_{new}$. Any uncovered subscriptions are forwarded along the unsubscriptions. Uncovered subscriptions can be recognized, because they have at least on new element in their forwards sets, added there due to removal of a covering filter.

5.1.3 Adding and deleting entries

Deleting or adding an entry F in the FP is fairly straightforward [TK06]. The deletion algorithm first disconnects F from its successors by walking through the entries in $ImSucc(F)$ and deleting F from their $ImPred$ sets. If F is a root filter, its immediate successors become new root filters as a result of this procedure and the operation terminates. If F is not a root filter, it is removed from the $ImSucc$ sets of entries in $ImPred(F)$. Then the immediate predecessors and successors of F are connected with each other; an entry $F_p \in ImPred(F)$ is connected with an entry $F_s \in ImSucc(F)$ if F_p has no immediate successor that would already cover F_s.

In adding operation, the first step is to build the $ImPred(F)$ and $ImSucc(F)$ sets for the new enty F. This is achieved by walking the FP in depth-first order, starting from the root filters. In each branch, the last filter that covers F is added to $ImPred(F)$. The $ImSucc(F)$ set can be constructed by walking the subposet (that may consist of the whole poset if F is to be added as a root filter) defined by $ImPred(F)$ in breadth-first order. Once the immediate predecessors and successors of F are known, the new entry can be added into the FP by manipulating the successor sets of the predecessors and predecessors set of the successors accordingly.

5.2 Poset-derived forests

The *poset-derived forest* (PF) [TK06] aims to providing faster insert and delete operations by storing only a subset of the covering relations. In particular, in poset-derived forest, each filter entry has at most one parent. Formally, the PF is defined as the pair $(\mathcal{F}, \sqsupseteq')$, where $\mathcal{F}$ is a set of filters and $\sqsupseteq'$ is a subset of the covering relation $\sqsupseteq$. The $\sqsupseteq'$ relation is what makes the data structure a forest – for each $F \in \mathcal{F}$ there is at most one $G \in \mathcal{F}$ for which $G \sqsupseteq' F$. Further, if for some $F, G \in \mathcal{F}$ it holds $G \sqsupseteq' F$, then $G \sqsupseteq F$. Figure 8 has an example of a poset-derived forest.

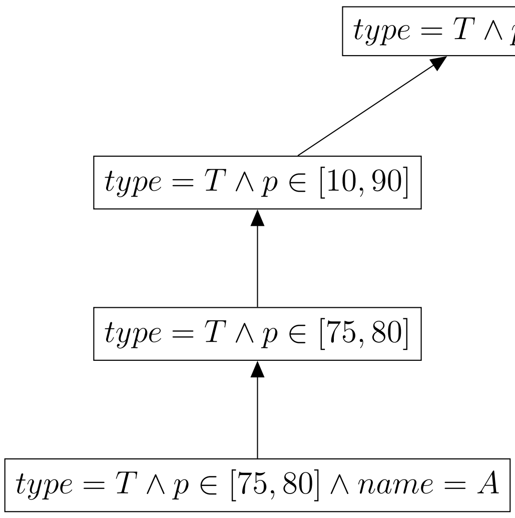

$type = T \wedge p \leq 100$
$type = T \wedge p \in [10, 90]$
$type = T \wedge p \in [70, 95]$
$type = T \wedge p \in [75, 80]$
$type = T \wedge p \in [90, 95]$
$type = T \wedge p \in [75, 80] \wedge name = A$
$type = T \wedge p = 90$

Figure 8: A poset-derived forest formed from the set of filters in Fig. 7.

A poset-derived forest $(\mathcal{F}, \sqsupseteq')$ is said to be *maximal* if no filter pairs can be added to the $\sqsupseteq'$ so that $(\mathcal{F}, \sqsupseteq')$ would remain a poset-derived forest. It is easy to make any poset-derived forest maximal by adding pairs to the $\sqsupseteq'$ relation, i.e. combining trees in the forest while respecting the definition of the $\sqsupseteq'$ relation until no more combinations can be done. In a maximal poset-derived forest, the set of roots of the trees in the forest is also the root set (minimal cover) of $\mathcal{F}$ [TK06]. This is a useful property and ensures that the minimal cover can be computed efficiently in poset-derived forests.

Another property usually required from a PF is *sibling-purity*. A PF is sibling-pure at node a if there exist no $b, c \in \mathcal{F}$ for which $a \sqsupseteq' b$ and $a \sqsupseteq' c$ and either $b \sqsupseteq c$ or $c \sqsupseteq b$. A poset-derived forest is sibling-pure if it is sibling-pure at every node. This definition simply states that children of every node in the forest are unrelated. Note that a may refer to so-called *imaginary root*, which is a convenience node that has all the root nodes of the trees in the forest as children. Maintaining sibling-purity at every node ensures that filters are stored as deep in the forest as possible. This is desirable because in a sibling-pure forest the number of covering tests needed when adding or matching filters is minimized. The forest in Figure 8 is both maximal and sibling-pure.

The add and delete operations of poset-derived forests maintain sibling-purity. The add operation for new filter F works as follows:

1. If F already exists in the forest, terminate.
2. Else set the imaginary root as the current node and repeat until F is inserted into the tree:
   - If F is unrelated with all children of the current node, add it as a child of the current node.
   - Else if F covers one or more children of the current node, move those covered children to be children of F and add F as a child of the current node.
   - Else pick one of the children that covers F and set it as the current node.

It is easy to see that the add operation maintains sibling-purity in the forest. Note that the data structure does not support fast testing of an existence of a filter in the forest. For this, a separate data structure, usually a hash table, can be used [TK06]. This provides a fast checking of *syntactic equivalence*, i.e. if a filter by the same representation is present in the forest. However, checking for *semantic equivalence* i.e. identity of filters is considerably more complex computationally and does not benefit from hash tables. Because of this, identity checking is usually omitted.

Maintaining sibling-purity in delete operation is slightly more complex. When a filter (node) F is being removed from a poset-derived forest, the node is removed and its children are added back into the forest by calling the add operation with the parent node of F (which may be the imaginary root) as the current node for each of the children. The child node that is being reinserted carries with it the subtree rooted at that node. When a new parent is found for the node, its children may need to be relocated further down the tree in order to maintain sibling-purity [TK06].

Unlike the Siena filters poset, the poset derived forest described above does not keep track of the interfaces through which the filters were received. The PF can be extended to handle multiple interfaces by storing the set of interfaces in the filter nodes, as is done in the filters poset. When a filter F is received through interface i, it is added in the forest if there is no filter that covers F and has i in its set of interfaces. If a node is added, all descendants of the new node that have the same interface i in their interface sets are removed from the tree.

The information about the interfaces can also be taken into use in implementing *interface-based balancing.* In an interface-balanced forest, filters associated with the same interface are kept close to each other. This can be achieved by storing the interfaces of the descendants of a node in each node. The insert operation can then use this index to choose a subtree that has similar interfaces in it. The interface index is updated by every insert and delete operation.

Additional steps can be taken to remove redundancy from the forest. A filter (F, i), where i is an interface, is redundant if there is a filter (G, i) for which $G \sqsupseteq F$. A poset-derived with multiple interfaces is *non-redundant* if there are no redundant filters. Note that the process described above removes redundant filters in a subtree rooted at a newly added filter, but in order to remove all redundant filters, extra measures are needed. Non-redundancy requirement has a negative impact on the performance of the data structure. Checking for redundant filters can however be made relatively efficient by utilizing the interface indices at each node. This allows for pruning parts of the forest when searching for redundant filters.

The poset-derived forest or one of its variants can be used to replace the filters poset in routing tables. However, it can also be used to complement the FP – a poset-derived forest can be set up as a *local routing table* to manage the local clients of a router, while the FP still acts as an *external routing table* that keeps track of (un)subscriptions received through external interfaces. This allows for the FP to be updated only when the root set of the PF is changed, a convenient property when there are many local clients with frequent subscription changes, as the PF can handle insertions and deletions more efficiently than the FP. Furthermore, using a separate routing table for local clients relieves the subscribers of keeping track of covering relations between their subscriptions (recall that in Siena unsubscription semantics, several subscriptions may be cancelled by issuing one unsubscription that covers them all).

It has also been proposed that two or more poset-derived forests can be *chained* [Tar07]. Chaining forests allows for implementing complex routing structures, such as matching subscriptions with *profiles.* A profile can be viewed as a set of triggering rules based on context (such as time or location) or metadata. An example of matching subscriptions with profiles would be to deliver notifications to a subscriber only at a certain weekday defined by the profile of the subscriber.

6 The offloading algorithm

In this section, we present our algorithm, which aims to improve scalability of the content-based routing function by offloading the content routing cost from the router nodes to the clients (in this context, client may also mean a neighboring router). In particular, we are interested in the case when new subscriptions are added to the system. In this, we consider the case where routing tables are either Siena-style filter posets or poset-derived forests. We will first examine how the tasks should be divided amongst the clients. Obviously, some overlapping and redundancy is needed as clients may produce results slowly, disconnect while performing the computation or provide wrong results on purpose. Thus, the results produced by clients must be verified and, if they are correct, inserted into the routing table efficiently. Our goal is to relieve the router of computing covering relations between filters, which is usually the most time-consuming part in inserting new subscriptions.

The problem and approach addressed here are similar to *cookie* or *computational puzzle* techniques used in some protocols to prevent attacks. In the cookie approach, a server may send a stateless cookie to a client and refuse further processing of requests from that client until the client sends back a reply with the same cookie. This method prevents e.g. denial of service attacks from spoofed IP addresses and is utilized by, among others, the DTLS protocol [RM06]. In the puzzle (or *challenge*) approach, the server may issue the client some task that requires some computation by the client but whose solution is easily verified by the server. The client is not served until it has provided a correct solution. This way, mounting a DoS attack from a single source becomes unfeasible due to the overhead generated by the puzzles. This approach is used for example in the Host Identity Protocol (HIP) [MNJH08].

We will also describe the protocol over which the offloading process is done. We present two different options: the *routing table offloading protocol* (RTO), where a part of routing table is included in each computation request sent to a client and the *fixed filterset protocol* (FF), in which the clients are sent a set of predefined filters and their relations and the clients then place their subsequent subscriptions in relation to this set. We describe the message formats and give examples of how the extended protocol works in different scenarios. We also briefly address other issues, such as security (the protocol reveals parts of the routing tables). In general, however, we assume that the routing table data contains no confidential information and can be disclosed to clients. Detailed security considerations are left for future work.

6.1 Dividing the tasks

For dividing the routing table in the RTO protocol, we use a simple layer-based vector approach. The routing table (either filters poset or poset-derived forest) will be "flattened" into a vector layer by layer. A layer in this context means the set of filters that are on the same level in the routing table. Thus, the first layer (layer 0) consists of the filters in the root set, the second layer consists of the immediate successors of the root filters, and so on. Note that if the filters poset is used, a filter may belong to several layers (Fig. 7 has an example of a filter that belongs to two different layers). Each client will be issued a part of the filter vector and a set of filters to place in relation to the given vector. In order to fully determine a location in the routing table for a new filter, the vector parts sent to the clients must cover the whole routing table. Each clients returns three sets: (1) The filters in the assigned vector that cover at least one of the input filters, (2) the filters in the assigned vector that are covered by at least one of the input filters and (3) a filters poset formed from the input filters. In practice, it is convenient to represent sets (1) and (2) as key-value tables where keys are input filter index numbers and values are sets of routing table filter index numbers.

In order to minimize the negative impact caused by malicious, unresponsive or uncooperative clients, the same vector slice and input set can be sent to several clients. In our scheme, a whole layer is sent to a set of clients. This may leave some room for improvement, as layer sizes may vary greatly, meaning more work for some clients. It is usually desirable that the workload is evenly distributed to the clients. It is easy to amend our scheme so that large layers can be split into several vectors or small layers can be combined into one vector. In our RTO implementation, we used the former approach. However, in our presentation we use the simple approach.

The FF protocol is an alternative way of offloading content routing cost. In the FF protocol, the clients are provided with a fixed set of filters. Then, whenever a client issues a new subscription, it must place the subscription filter in relation to the fixed filter set. The router can then use this information to place the filter in its own routing table. Similar technique can be used also with neighboring routers. Also, there is less communication overhead, as input vectors need not be transmitted to the clients.

6.2 Computing and verifying the results

In the RTO protocol, it is relatively easy for the router to add filters to the routing table based on the results returned by the clients. The return vectors are traversed layer by layer starting from the root layer. If an input filter is marked covered by a filter in the vector, the search continues to the next layer. The process continues until no filters in the layer cover the input filter or there are no more layers. The filter can then be placed as an immediate successor of the last covering filter found in this process; the router needs not compute any covering relations between the filters. The details of this process are discussed below for both the filters poset and the poset-derived forest.

For verifying the results produced by the clients, the router makes use of the fact that the same filter vector was sent to several clients. It compares the results from the clients that were issued the same vector and discards results inconsistent with the majority of results. This does not ensure correctness but in the context of this work, this level of verification can be considered adequate; in any case, if majority of clients are malicious or malfunctioning, the scheme presented here will have a negative impact on the performance of the router. More thorough check may be done occasionally for some filters. This is done by verifying that the predecessor filter(s) suggested by the clients actually cover the filter and that all successors of the predecessor filter(s) are covered by the filter. If absolute correctness is required, this check can be performed on each filter for which the clients have computed a place in the routing table. The amount of computation required by the router remains still less than without offloading, but performing checks on this level is not usually required.

In the FF protocol, the clients position their filters in relation of some set of static filters instead of positioning input filters in relation to an input vector. In this case, the router may need to do some extra processing; namely, the static set may only form a small subset of the routing table, and the router may have to compute some covering relations after receiving the results. Thus, the static set approach is analogous to pruning the routing table so that some options need not to be considered when inserting a filter.

6.2.1 Computation in the RTO protocol

We now present the algorithm the router uses in order to determine the place in the routing table for a set of new filters based on the covering data returned by the clients. The process is explained both for the filters poset and the poset-derived forest. First stage of the algorithm, the *cleanup stage*, is common to both data structures. Its purpose is to produce the final result sets on which the filter placing process will be based. In this stage, incorrect results are discarded and if some results are missing or inconclusive, they are computed by the router. In the following, let L represent the number of layers in the routing table. The set of filters to be placed in the routing table is called the set of *input filters* and is denoted by F. The *result set* returned by each client contains three entries: $Pred$, $Succ$ and F_{FP}. Entries $Pred$ and $Succ$ consist of routing table filters that are predecessors and successors of at least one filter in F, respectively. The entry F_{FP} is a filters poset formed from the filters in I.

1. Initialize R to be an array of L elements.
2. Initialize F_I to be an empty filters poset.
3. For each layer $l \in [0, L-1]$, do
   (a) Compute the frequency of each $(Pred, Succ)$ pair returned by the clients that were assigned the vector representing layer l.
   (b) Let $(Pred, Succ)_\mu$ denote the pair that occurred more frequently than any other pair. If $(Pred, Succ)_\mu$ exists, set $R[l] \leftarrow (Pred, Succ)_\mu$.
   (c) Else compute the result set for layer l and set it as the value of $R[l]$.
4. Compute the frequency of each poset F_{FP} returned by the clients.
5. Let F_μ denote the poset that occurred more frequently than any other poset. If F_μ exists, set $F_I \leftarrow F_\mu$.
6. Else compute the filters poset from filters in I and set it as the value of F_I.

Note that in step 3a, some hash function may be used instead of directly comparing the sets. Also, the rule in steps 3b and 5 is only one metric of deciding which result set should be considered. It is relatively easy to come up with others. It is desirable that steps 3c and 6 are executed very rarely if at all as they are the most computation-intensive steps of the cleanup algorithm. They are the last resort if results from clients cannot be trusted or obtained at all.

In the *filter insertion stage*, the router inserts the input filter(s) into its routing table. Depending on the data structure used, either the procedure `filter_insert_fp()` (for filters poset) or `filter_insert_pf()` (for forest) is called. Both procedures take in as an input the array R and poset F_I obtained from the cleanup stage and a boolean value *check* that indicates if filter placements should be verified by computing the covering relations between it and its predecessors and successors. In normal operation, we usually want to set *check* to $false$ and maybe occasionally set it to *true*. If *check* has the value *true* on every call, the correctness of the routing table is ensured in all situations but a significant overhead is introduced. The purpose of this procedures is to find the place in the routing table for filters in F_I without having to compute any covering relations. After a correct location is found for a filter, some measures may still be needed, such as manipulating the *subscribers* sets in the filters poset (see Section 5.1.1).

In the following, let $Pred(l, f)$ denote the set of routing table filters that are predecessors of a filter $f \in F_I$ at layer l. $Succ(l, f)$ is defined similarly. The procedure `filter_insert_fp()` is defined as follows:

1. Initialize F_{ins} to an empty set. This set will hold the input filters that have been inserted into the routing table already.

2. For each layer l in L, do

   (a) If layer $l-1$ exists, set $(Pred_{l-1}, Succ_{l-1}) \leftarrow R[l-1]$. Otherwise, set $Pred_{l-1} \leftarrow Succ_{l-1} \leftarrow \emptyset$

   (b) Set $(Pred_l, Succ_l) \leftarrow R[l]$

   (c) If layer $l+1$ exists, set $(Pred_{l+1}, Succ_{l+1}) \leftarrow R[l+1]$. Otherwise, set $Pred_{l+1} \leftarrow Succ_{l+1} \leftarrow \emptyset$

   (d) Determine the set of filters in F_I that belong to layer l. Let F_l denote this set. A filter f belongs to layer l, if one of the following conditions is met (note that filters belonging below the bottom layer are a special case that will be handled later):

      - The set $Pred(l, f)$ is empty and f is not yet in the set F_{ins}.
      - Layer $l-1$ exists and there exists a filter g' at layer l that has a predecessor f' at layer $l-1$ such that $f' \in Pred(l-1, f)$ and

$g' \notin Pred(l, f)$.

(e) Using the data in F_I, $Pred_{l-1}$ and $Succ_{l+1}$, set the immediate predecessor and successors relations for the filters in F_l, effectively adding them in the routing table:

- If layer $l-1$ exists, filter f' at layer $l-1$ is added as an immediate predecessor of filter $f \in F_l$ if $f' \in Pred(l-1, f)$ and there exists no filter $g \in F_I$ for which $g \in F_l$ and g covers f and $f' \in Pred(l-1, g)$. As an additional check, if an input filter f happens to belong to two or more layers, and l is not the last one it belongs to (i.e. $Pred(l, f)$ is nonempty), then filters whose immediate successors are also predecessors of f are *not* added immediate predecessors of f.
- The links to immediate successors of f' that are in $Succ(l, f)$ are removed as they are now covered by f and are no longer immediate successors of f' (this phase can also be executed in the end if modifying the relations in the routing table is not desirable at this point, which may be the case if all relations are to be checked).
- For each filter $f \in F_l$, filter $g \in Succ(l, f)$ is added as an immediate successor of f unless f covers some $f' \in F_l$ for which it also holds that $g \in Succ(l, f')$.
- For each filter $f \in F_{ins}$, if there is a filter $g \in Succ(l, f)$, whose immediate predecessors (that can be either routing table or input filters) are not covered by f, add f an as immediate predecessor of g.
- If $check = true$, each added predecessor and successor link must be verified by computing the covering relation. If this check fails for some filter $f \in F_l$, that filter is added into the routing table using the normal insert operation of the filters poset.

(f) Add to F_{ins} each filter f for which $Pred(l, f)$ is empty; these filters cannot belong to any lower layer, but they may have some immediate successors at the lower layers.

3. Insert the filters that belong below the bottom layer $L-1$. A filter f is inserted below the bottom layer if it is not in F_{ins} and $Pred(L-1, f)$ is nonempty. All filters in $Pred(L-1, f)$ are added as immediate predecessors of f, unless they cover some other filter in F_I that covers f and also belongs below the bottom layer.

4. Compute which of the immediate covering relations in F_I are retained in the result poset. This is done by comparing the successors and predecessors (in the routing table) of each filter pair with an immediate covering relation in F_I. If the intersection of those sets is empty, the immediate covering relation is retained in the routing table poset also.

The procedure `filter_insert_pf()` is essentially a simplified version of the procedure `filter_insert_fp()`. For example, the condition when a filter f belongs to a layer l can be relaxed; it is sufficient to consider only the case where the set $Pred(l, f)$ is empty. This way, also sibling-purity (see Section 5.2) in the tree can be maintained. Also, inserting the filter is simpler because in a poset-derived forest, each node has at most one parent. If several filters are being inserted on the same layer, some of their relations (found in F_I) may need to be dropped in order to maintain the properties of the forest. The procedure `filter_insert_pf()` can also be easily extended to work with multiple interface and interface-balanced forests.

6.2.2 Computation in the FF protocol

In the FF protocol, the router assigns the clients some predefined set of filters and a set of their covering relations (either in the format of a filters poset or a poset-derived forest). Any client that has received this set will then place their subsequent subscriptions in relation to this filterset and deliver the information along the subscription message. In order to do this, the client must compute covering relations between the subscription filter and the filters in the predefined filterset. The placement data delivered to the router consists of two sets: the immediate predecessors $ImPred(f)$ and the immediate successors $ImSucc(f)$ of the subscription filter f. Either or both sets may be empty. Furthermore, it is easy to extend the protocol to handle the insertion of several filters at once, as is the case in the RTO protocol.

The router uses the data provided by a client in pruning the routing table data structure. Having provided with the immediate predecessors and successors in the fixed filterset with the subscription, the router is able to exclude from further consideration any filter covered by the immediate successors and all filters covering the immediate predecessors of the subscription filter. Because the fixed filterset represents only a subset of all filters in the routing table, there may be some filters (unknown to the client) between the predecessors and successors proposed by the client. Thus, the router still needs to compute some covering relations between the subscription filter and the filters in the routing table. The covering relation to filters in the following sets must be computed in a filters poset:

1. Filters that do not cover a filter in $ImPred(f)$ and either share a layer with a filter in $ImPred(f)$ or are at a higher layer. These filters are potential immediate predecessors of f.

2. Filters covered by a filter in $ImPred(f)$ and covering a filter in $ImSucc(f)$, if both sets are nonempty, or filters covered by a filter in $ImPred(f)$, if $ImSucc(f)$ is empty, or filters covering a filter in $ImSucc(f)$, if $ImPred(f)$ is empty. These filters may be immediate successors or predecessors of f (or unrelated).

3. Filters that are not covered by filter in $ImSucc(f)$ and either share a layer with a filter in $ImSucc(f)$ or are at a lower layer. These filters are potential immediate successors of f.

Again, when using a poset-derived forest, not all of these checks are needed; it is sufficient to compute relations to the sibling candidates in order to ensure sibling-purity.

Obviously, the benefits of the FF scheme greatly depend on how much of the actual interest space the fixed filterset covers. If the filterset is only a small subset of all filters in the routing table, the router needs to compute many covering relations. At the same time, the filterset should be relatively static so that continuous reissuing messages and other complications can be avoided. It is outside the scope of this work to address this problem in detail.

6.3 The protocol

In this section, we describe the functionality and messages needed to implement our offloading scheme. We consider the request and response messages needed in the RTO protocol and address some issues in collecting incoming subscription filters into *batches* rather than sending out a computation request for every subscription separately. The RTO protocol must also take into account the fact that some clients may perform slowly or be uncooperative. Therefore, a timeout mechanism is needed. After a timeout occurs, the router no longer waits for computation responses rather than starts working with the responses received so far.

We also describe the message used by the routers in the FF protocol to issue the filterset to the clients. The same message can be used to replace the filterset with a new one should changes occur. The FF protocol also requires some changes to the subscription messages, namely including the $ImPred$ and $ImSucc$ sets for the subscription filter in relation to the fixed filterset.

6.3.1 RTO computation request

An RTO computation request is sent out by a router to a subset of its clients. It consists of one layer of the routing table and the set of input filters, both encoded into index-filter pairs. The index numbers are integers and filters are strings. An example request is shown in Fig. 9.

Layer vector	*Input filters*
$0 : p \in [10, 90], 1 : p \in [70, 95], 2 : name = A$	$0 : p = 80, 1 : p \geq 60, 2 : name = B$

Figure 9: An RTO computation request

6.3.2 RTO computation response

The response to an RTO computation request, returned by an individual client, consists of the representations of result set entries $Pred, Succ$ and F_{FP}. Recall from Section 6.2.1 that the entries $Pred$ and $Succ$ contain the predecessors and successors of the input filters in relation to the layer vector and F_{FP} is a filters poset formed from the input filters. The sets $Pred$ and $Succ$ are represented as key-value tables where input filter indices act as keys and the value associated with a key is a set of indices in the layer vector. This organization is convenient for finding the $Pred(l, f)$ and $Succ(l, f)$ sets for a filter f at layer l. The poset F_{FP} is also represented using only filter indices; for each input filter, there is a key and an associated value. The value consists of two sets, namely the immediate predecessors and successors of the filter represented by the index.

Additionally, we require that all keys and the index numbers in the value sets are sorted in ascending order by their numeric values. This makes it easier to compare the results as strings (e.g. compute hashes of them) rather than sets in the cleanup phase, speeding up the process considerably.

$Pred$	$Succ$	F_{FP}
$0 : \{0, 1\}, 1 : \{\}, 2 : \{\}$	$0 : \{\}, 1 : \{1\}, 2 : \{\}$	$0 : [\{1\}, \{\}], 1 : [\{\}, \{0\}], 2 : [\{\}, \{\}]$

Figure 10: An RTO computation response to the request in Fig. 9. For each entry in F_{FP}, the first set represents the immediate predecessors of the filter and the second one represents the immediate successors.

6.3.3 RTO batch mode and timeouts

If the router receives a large amount of subscriptions in a short period of time, sending out an RTO request for each one may be inefficient in terms of bandwidth utilization and may cause an unnecessary load on the clients. Therefore, we will implement a configurable *batch mode* in the protocol. Incoming subscriptions are buffered and an RTO request is sent out when either a predefined time limit or a threshold set for the number of filters has been reached. Clearly, there is a trade-off between the frequency of RTO requests and the representativeness of the routing table; if incoming subscriptions are held too long before issuing the offloading request, some notifications may not reach the subscribers as their subscriptions have not been added to the routing table yet.

In order to ensure that slow clients and errors in communication do not freeze the routing table update process, a timeout mechanism is needed. Whenever a set or RTO requests is sent for some set of input filters, a timer is started. The router moves to the cleanup stage when all responses have been received or when the timer expires, whichever occurs first. Missing results are computed by the router during the cleanup stage.

Another important parameter in the RTO protocol is the number of clients one layer of routing table is sent to. Let C denote this number. If the routing table has L layers, placing a set of input filters into the routing table will require sending CL RTO computation requests. Increasing the value of C increases the probability of receiving (correct) results from the clients but it also increases bandwidth usage and overhead caused to the clients by the offloading scheme. Also, it is usually preferable that one client is computing at most one RTO request at any given time, i.e. increasing the value of C may be problematic if the router has very few clients as there may not be enough clients available for serving the requests.

6.3.4 FF (re)assignment message

In the FF protocol, a router assigns its clients with the fixed filterset. This is done by the FF assignment message. Upon receiving such message, the client must discard any filtersets received before. Thus, the assignment message can be used also in changing the fixed filterset. The message consists of string filters and their immediate covering relations (in the case of filters poset) or parent and child relations (in the case of poset-derived forest). Filters are indexed and referred to as integers. An example message is shown in Figure 11.

Fixed filterset
$0 : [p \leq 100, \{\}, \{1, 2\}], 1 : [p \in [10, 90], \{0\}, \{\}], 2 : [p \in [70, 95], \{0\}, \{\}]$

Figure 11: An FF assignment message. Each entry has a filter, the set of immediate predecessors (or a parent) and the set of immediate successors (or children), respectively.

6.3.5 Subscriptions in FF protocol

The FF protocol defines an extension to the regular subscription messages. In addition to the fields in the subscription messages, there are two new fields, $ImPred$ and $ImSucc$. Both contain a set of indices that represent the immediate predecessors and successors, respectively, of the subscription filter in relation to the fixed filterset. Figure 12 shows an example of the subscription extension.

Filter	$ImPred$	$ImSucc$
$p \in [75, 80]$	{1,2}	{}

Figure 12: Part of a subscription message in the FF protocol. The filter and $ImPred$ and $ImSucc$ sets are shown. The indices refer to filters in Fig. 11.

6.4 Other considerations

In this section, we address some issues left outside the scope of this work or to be tested experimentally. These include some security considerations, performance issues, applying offloading also to unsubscriptions, alternative ways to implement the batch mode in the RTO protocol, and applying the FF scheme in inter-router communication.

6.4.1 Security

The RTO scheme we have proposed has some security issues. It may not be desirable that the interests of other clients and the structure of the routing table are revealed in the offloading process. Information about the subscribers is not sent in the computation request messages, but sometimes we do not wish to reveal even the interest space. The offloaded version of the routing table may be obfuscated to some extent by simple transformations. For example, all numeric values can be shifted (by addition or subtraction) by some constant value and strings may be transformed. This somewhat masks the interest space but information about the routing table structure is still revealed. In the FF scheme, routing table exposure is somewhat more limited. We do not issue these security considerations further. We assume that these simple obfuscations are adequate means of maintaining client privacy. More advanced solutions are outside the scope of this work.

6.4.2 Performance

We observe that the RTO scheme may in some cases limit the scalability of a content-based pub/sub system rather than improve it. This happens if there are very few clients or a large number of unresponsive, slow or malicious clients. By experiments, we try to find the threshold where the proposed offloading scheme is no longer feasible.

Alternative ways of dividing the routing table should also be considered. Assigning a whole layer to a set of clients may lead to unbalanced burden amongst the clients. Combining and splitting of layers may be needed.

The RTO scheme could also be expanded further to handle identical filters; the clients would then report if some of the input filters are identical to one of routing table filters or to each other. With this addition, the computational effort required from the clients would grow, but routing table sizes would in turn decrease.

6.4.3 Offloading unsubscriptions

An obvious question that follows from the offloading of subscriptions is if the same approach can be used with unsubscriptions also. In a filters poset, the deletion operation is quite straightforward and would not benefit from offloading. The poset-derived forest does however require the computing of covering relations during the delete operation, if sibling-purity is to be maintained. Thus, the forest may benefit from offloading.

6.4.4 Batch mode adjustments

In our scheme, the RTO batch mode assigns the whole batch of input filters to clients once some predefined batch size has been reached or some fixed amount of time has passed. Instead, it may be optimal to divide a batch into smaller parts and offload each part separately. This process might be combined with some preprocessing step in which the router would determine a good way to partition the input filter batch, possibly by computing some covering relations amongst the batch beforehand (determining an optimal partitioning may be computationally infeasible).

6.4.5 Extending the FF scheme

The FF scheme could also be used between neighboring routers when forwarding subscriptions. As a further extension, the filterset used may be dynamic and constructed by each router individually by modeling the interest space of their neighbors. The routers could then anticipate the needs of their neighbors and provide some initial data to reduce the load caused to their neighbors by forwarded (un)subscriptions.

7 Experimentation

We now present the experimental results for our algorithm. We will test the performance and correctness of our offloading scheme under varying conditions in the cases where routing tables are implemented as filter posets and poset-derived forests. We also compare the results against filter posets and poset-derived forests without offloading. Section 7.1 describes the test setting in detail, including the parameters of used filters. In Section 7.2, we describe the test scenarios and in Section 7.3, we provide the outcome of the tests. A brief summary of the tests and their results is given in Section 7.4.

7.1 Implementation and environment

For the tests, we implemented the server-side functionality of the RTO protocol and executed it in an environment that simulates a situation where there is a sufficient number (in terms of dividing the workload) of reliable clients. The implementation was done in the context of the Fuego middleware[1]. A layer-based implementation was made both for the filters poset and poset-derived forest. The Fuego implementations of these data structures were used as the reference in performance and correctness tests.

The server-side operation of the RTO consists of five phases. The time taken by each phase was measured. In the *preprocessing phase*, the set of new filters is examined in order to remove duplicates and filters that have already been in the data structure. Also, some elimination of interfaces (see Section 5.1.2) is done in this phase. Only syntactically identical filters are detected in this phase. The preprocessing phase is followed by the *layerization phase*, in which a layered representation of the data structure is formed. This representation is needed in order to issue the RTO computation requests to the clients.

In the *offloading phase*, the computation of covering relations between the input filter(s) and the routing table filters is executed by the clients. In the test setting, this was simulated by executing the computation locally in sequential fashion and taking into account only the longest time taken by a ”client”. The purpose of this was to easily simulate the concurrent environment, where the total time taken by the offloading is determined by the longest time an individual client takes to perform the computation. Communication latency is thus not *not* included in the test results. Also, the results obtained are always correct.

The phase of most interest is the *insert phase*, which is a realization of the `filter_insert_fp()` and `filter_insert_pf()` (depending on used data structure) procedures described in Section 6.2.1. Last, in the *postprocessing phase*, an interface elimination procedure is called for each of the inserted input filters in order to ensure that the covered filters do not contain redundant interfaces.

As test data, we used three pre-generated, differently distributed sets of range filters. Each test case was repeated three times, and test filters were inserted into the

[1]Available at http://www.tml.tkk.fi/ starkoma/fc/

data structures in random order. Both one-dimensional and two-dimensional ranges were tested. After each insert operation, an integrity check was made to ensure the correctness of the resulting data structure. In this, a regular poset or forest containing the same filterset was used as a reference structure against which the layer-based structure was compared. The time to insert the filters to the reference data structure was also measured in order to obtain data for insert performance comparisons.

It soon became obvious that offloading the computation of relations for a whole layer is infeasible for large layers (the first 2–3 layers of the data structures tend to be very large compared to lower layers). Therefore, we modified the RTO scheme presented above to split large layers into vectors of 10 filters.

As the test hardware, we used a desktop computer with a 2.66 GHz Intel Core 2 Quad processor, 4 GB of main memory, Windows Vista and Java JDK 1.6.0.

7.2 Tests

As the first test scenario, we inserted 10000 filters into the test and reference structures. The purpose of this scenario was to measure the overall performance of the data structures and the effects of different batch sizes. The following parameters were used:

- Distribution of subscription space: Gaussian, Zipf and uniform.
- Batch size: 1, 5, 10, 25 and 50 filters.
- Filter dimension: 1d ranges ($x \in [a, b]$) and 2d ranges ($x \in [a, b] \wedge y \in [c, d]$).
- Covering checks enabled and disabled.
- Data structure: filters poset and poset-derived forest.
- The number of interfaces was fixed at 20, and inserted filters were assigned an interface in round-robin fashion.

The total number of test cases in this scenario was thus 96. Each test case was executed three times to mitigate effects of chance variation. At the same time, integrity checks were made on the test structures. All tests were successful in terms of data structure correctness. The cost of integrity checks is not included in the results

The second test scenario measured the scalability of the insert phase. In this scenario, 10000 filters (uniform distribution) were inserted again. This time, the number of interfaces was fixed at 10000, ensuring that the data structure size equals the number of inserted filters (as no interface-based elimination can be done). Covering checks were disabled. Batch size was fixed at 10. Tests were executed for both 1d and 2d data for poset and poset-derived forest.

In the third test scenario, we measured the effects to performance caused by larger offload sizes. Increasing the offload size simulates a situation where there are fewer clients available and the remaining clients must perform more computation. Again, 10000 uniformly distributed filters (both 1d and 2d ranges) were inserted in a poset and poset-derived forest. Batch size was fixed at 10 and number of interfaces at 10000. The offload size parameter (number of routing table filters in one offloaded vector) was assigned values of 10 (the value used also in the first two test scenarios), 100, 200 and 500. Note that for layers smaller than the offload size, the layer size was used instead, i.e. filters from two or more layers are never combined to one vector.

7.3 Results

The results for the first test scenario are shown in Figures 13 and 14. For the layered poset, the cost of inserting 10000 filters is lower than that of the reference structure for all batch sizes when using 2d range filters. Offloading each filter insertion separately (batch size 1) has a significant overhead over other batch sizes. With 1d ranges, small batch sizes perform significantly worse than the reference structure. Moreover, although for large batch sizes the offloading scheme seems to perform better than the reference structure, the scheme is still not feasible with 1d ranges when considering additional overheads (such as communication latency).

The situation is similar with poset-derived forests; with 1d range filters, we observe degraded or similar performance when compared to the reference structure. With 2d ranges, the performance differences between the test structure and the reference structure are not as large as with regular posets. Further, having covering checks enabled introduces a significant overhead to layered forest implementation. This is due to an implementation detail: in the forest, sibling-purity is enforced by recursively relocating the input filters further down the tree, with covering checks being made with every relocation operation, increasing the number of checks performed when compared to filters poset, in which the filters are placed in their final places directly.

The results for the second test scenario are shown in Fig. 15. This scenario revealed a scalability issue with the poset insertion phase. Even with uniform data, the data structure tends to grow in depth very fast with 1d ranges. The cost of the insert phase is strongly affected by the number of layers in the poset. The insert phase for the forest in much simpler and does not suffer from the same issue. With 2d ranges, the data structure grows slower in depth. The width of the data structure does not seem to have a significant effect on performance.

For other cases, the cost of the insert phase grows very moderately. This holds particularly in case of 2d range filters. With 1d ranges and forest, the cost of the whole operation grows considerably faster than the time taken by insert phase. This is due to the fact that the layerization phase is expensive and its cost grows linearly as data structure size grows. In the following section, we propose some improvements for creating layered representation of the data structures.

Figure 16 show the results for the third test scenario. As the offload size grows, the computational strain on an individual client (the offload time) grows. This test also measured the effect of client amount on the scheme. If the client base is large, offload size can be kept relatively small. Smaller client base (and larger offload size) has a negative effect on performance.

The results show that poset with 1d range data again suffers from the performance issue identified in the previous test case. For other cases, the results are consistent: the offloading time takes a larger proportion of the total operation time as offloading size grows. It seems that setting the offload size between 10 and 100 is a feasible option. There is a trade-off between bandwidth usage/communication latency and computational effort required from the clients: for larger offload sizes, fewer messages are needed to perform offloading, but an individual client is required to do more work to complete its task.

7.4 Summary

Above, we have described the test setting and experimental results for our RTO scheme. Our implementation is not a full realization of the protocols described in Section 6, but it should provide some indicators of the applications of offloading content routing cost to clients. Based on the test results, at least the following observations and suggestions for further improvement can be made:

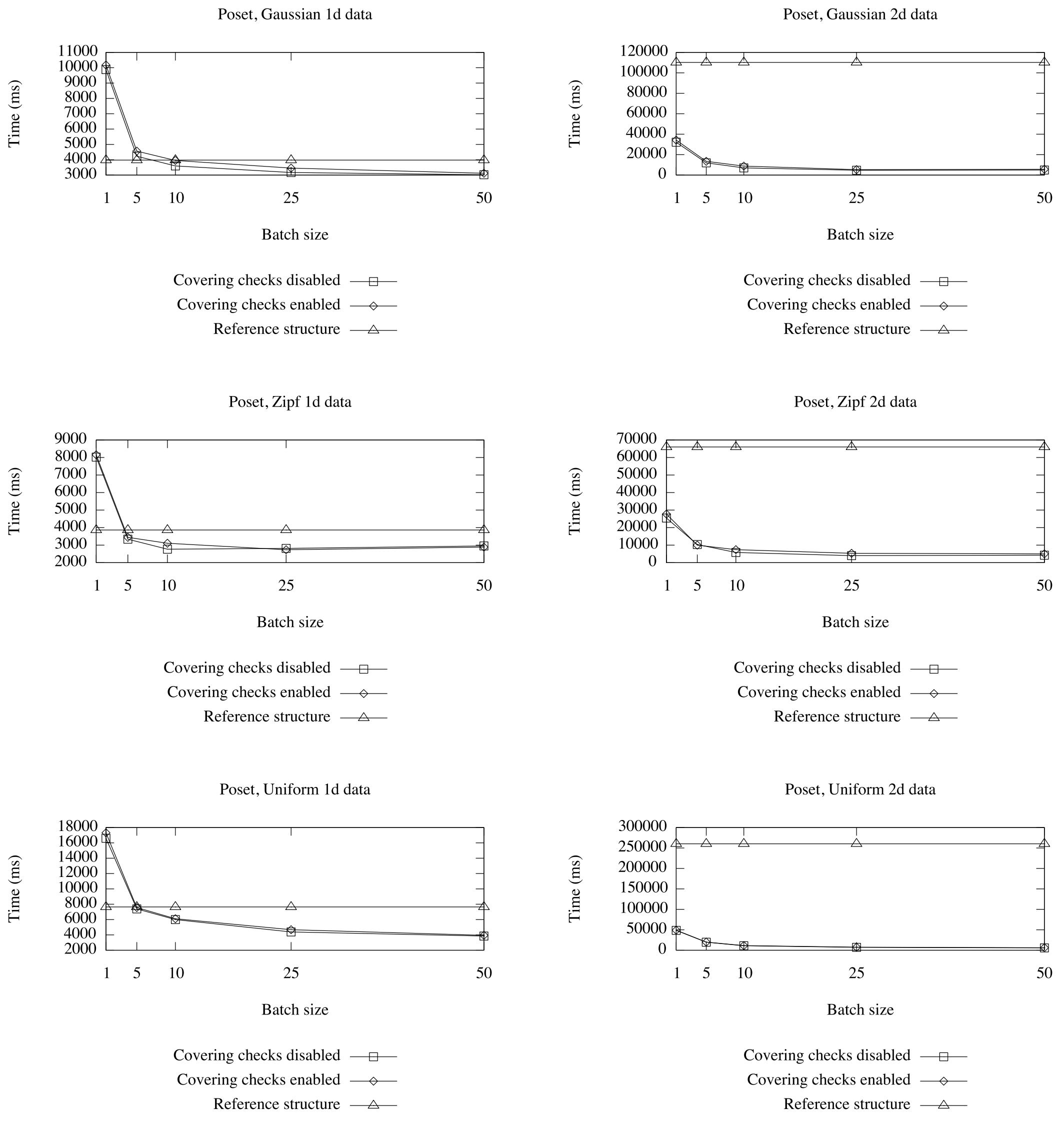


Figure 13: Results for adding 10000 filters into a filters poset (1d and 2d ranges, 20 interfaces).

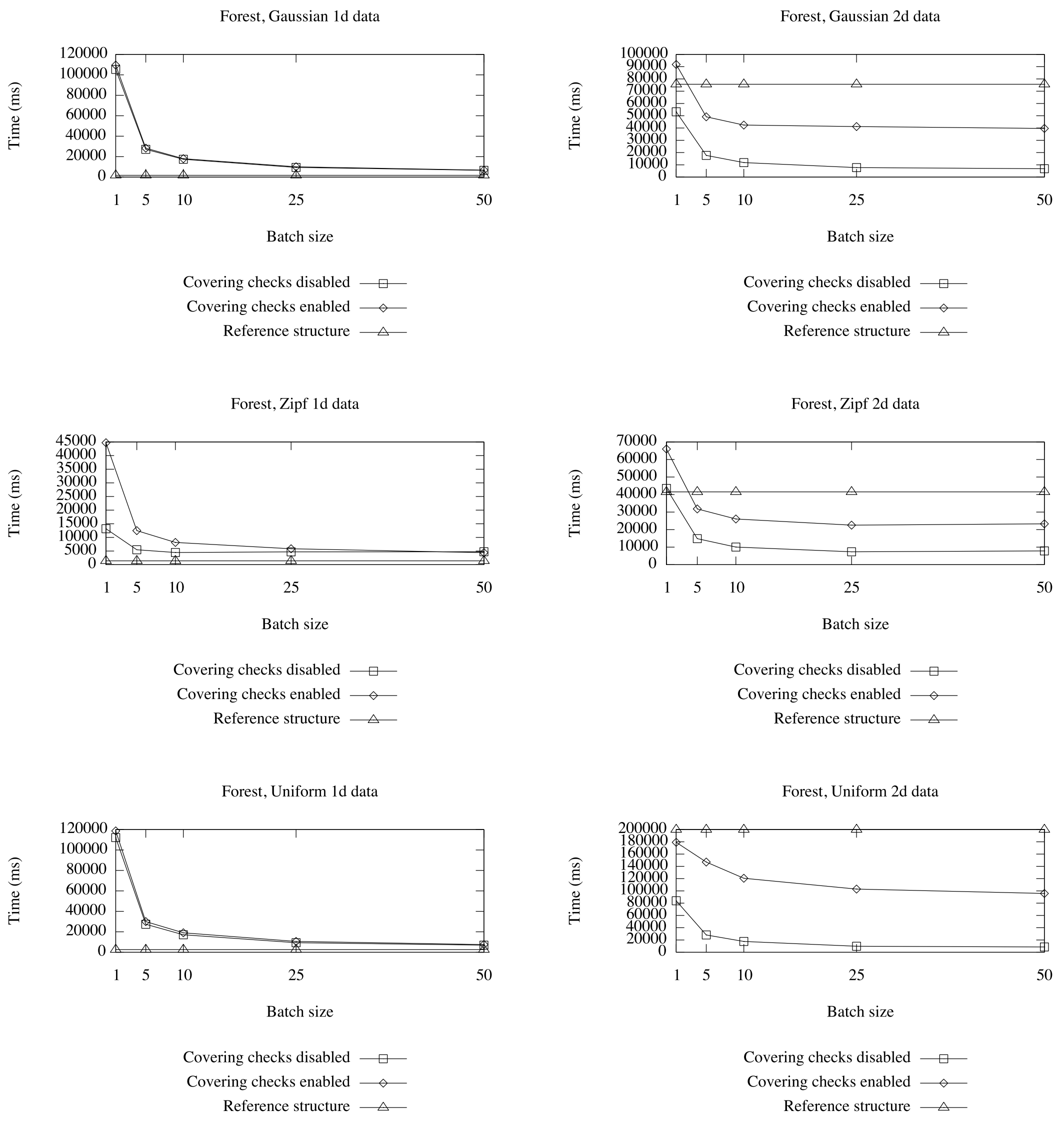


Figure 14: Results for adding 10000 filters into a poset-derived forest (1d and 2d ranges, 20 interfaces).

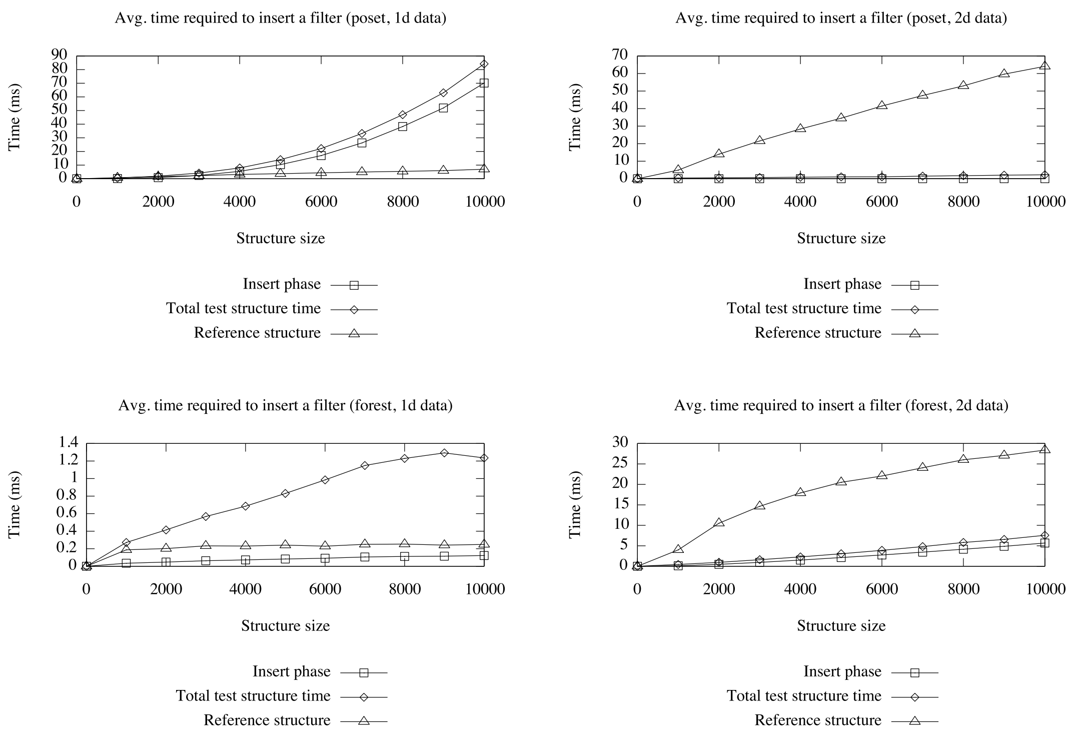


Figure 15: Cost of insert phase and total operation in comparison to reference structures as function of data structure size.

- If the subscriptions are simple (including only single range filter), offloading does not necessarily offer any performance or scalability boost. With the poset-derived forest, it may even degrade performance. This is mostly due to the facts that with a small number of interfaces, poset-derived forests tend to have more nodes than posets [TK06] and that insert operation in a regular poset-derived forest is generally quite fast.

- For more complex subscriptions, the performance gain is significant. Moreover, with complex subscriptions, the cost of the insert phase grows considerably slower than the cost of regular insert operation.

- In practice, the offloading scheme does not work well if the router has very few clients available for offloading. The cost of the offloaded computation for the clients is so high that it is preferable to offload only the computation of 10–100 relations per client. Offloading the computation for a whole layer to one client is not feasible. In future work, the communication cost (latency and bandwidth usage) needs to be also measured.

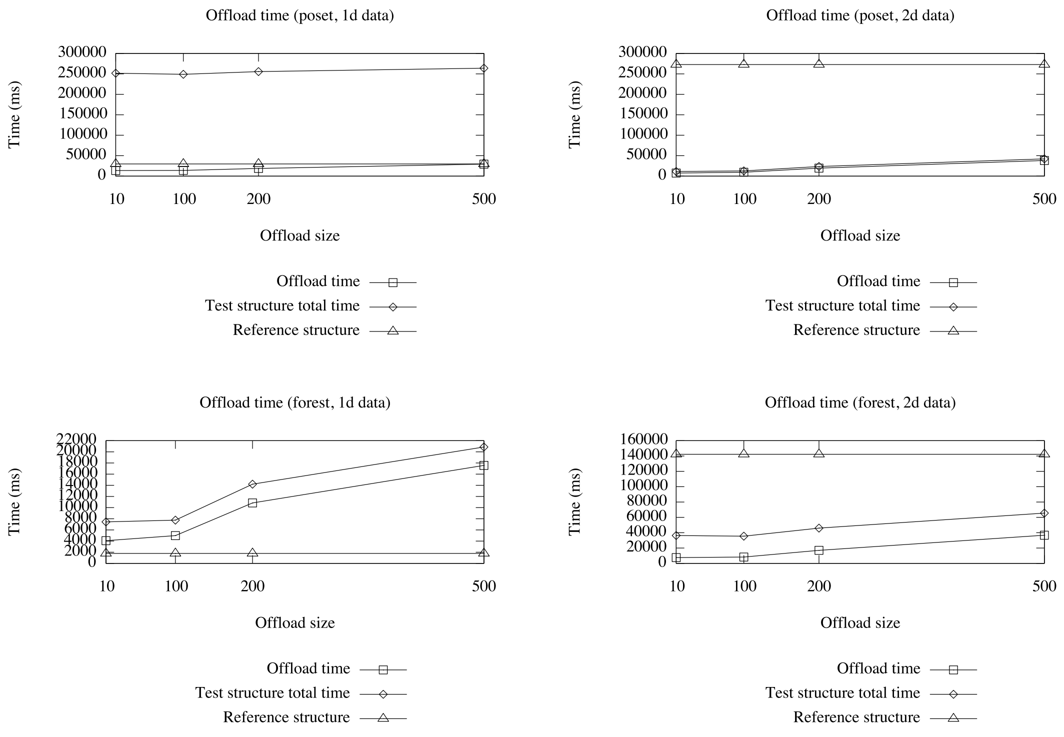


Figure 16: Cost of offloading phase and total operation in comparison to reference structures as function of offload size.

- It is beneficial to insert the filters in small batches; using offloading for the insertion of a single filter is too expensive. Using large batch size does not offer a significant performance improvement over smaller batch sizes and may have a negative impact on the freshness of the data structures as insertion of new subscriptions may be postponed for too long. A batch size of 5 or 10 subscriptions seems to be a good compromise.
- The cost of creating a layered representation of the data structure increases rapidly as the data structure size increases. In the current implementation, the layered representation is created in the beginning of each offload cycle. In our test setting, this cost was one of the most significant bottlenecks, especially so in the case of poset-derived forest. Maintaining the layered representation and updating it as the data structure is updated instead of re-creating the representation every time might increase performance.
- The cost of preprocessing and postprocessing phases seems not to be greatly affected by the data structure size; the dominant factor here is the batch size.

Thus, these operations scale very well.

- Enabling covering checks introduces only a relatively small overhead to the inserting phase for poset. However, it should be noted that in our tests, no filter actually failed a covering check. Failed filters may increase the cost considerably as they need to be inserted using the regular insert operation. For the poset-derived forest, a significant overhead was introduced if covering checks were enabled. This is because more covering checks per filter are made as the filter is being relocated in the forest in order to preserve sibling-purity.

We conclude that our RTO scheme is feasible, provided that the client base of a router utilizing the scheme is large enough. Further, our current implementation is not a complete realization of the scheme; implementation and tests of the actual protocol messaging between the clients and the RTO router is still needed. Future tests should primarily measure the effects to network traffic, such as latency and bandwidth usage, caused by the scheme. These are also some known performance issues in creating layered representation of the data structures; while it is a simple process, it requires iterating through all nodes in the data structure. Another scalability issue was identified in the implementation of the insert phase of the layered poset when the data structure is deep. Future work should also aim at solving these issues.

8 Related work on improving scalability

In the following, we take a look at alternative ways of building scalable content-based publish/subscribe systems. We discuss two quite recent approaches: systems that employ *distributed hash tables* (DHTs) and *Bloom filter based routing*. Examples of the former include systems such as Hermes [Pie04] and DHTStrings [AT05]. The latter approach [JF08] has yielded good experimental results. It does, however, come with a price: due to the probabilistic nature of Bloom filters, there is a chance of *false positives*, i.e. notifications being forwarded to clients who have not subscribed to them.

8.1 Systems based on distributed hash tables

As an example of an overlay network different from what we have used in this work, we discuss systems based on distributed hash tables. We start with introducing the concept of a distributed hash table. We then discuss Hermes [Pie04], a pub/sub system that employs DHTs in its overlay network.

DHT-based systems are usually very scalable and fault-tolerant and provide fast matching, but they somewhat limit the expressiveness of content-based pub/sub [JF08], as is the case with Hermes. Furthermore, DHT-based approaches have some additional requirements, such as globally unique node identifiers.

8.1.1 Distributed hash tables

DHTs [TDVK99] are widely used in implementation of large-scale peer-to-peer systems. As the name suggests, a DHT maps keys to values that are stored somewhere in the overlay. As usual, the key for a value is determined by applying a hash function on the value. In a DHT the key the hash function returns is the identifier of the node where the value is stored. The overlay is responsible of routing requests for keys to the corresponding nodes, also in situations where nodes leave and enter.

The details of constructing DHTs are omitted here, but we mention some considerations. First, any DHT needs some *keyspace partitioning scheme* that defines the ownership of the keys among the nodes in the DHT. A node owns all keys that are closer to its id than the id of any other node. The definition of ”close” varies per system, but in general it is required that there exists some distance metric between the keys. Second, the following property, or one similar to it, must hold for every node in any DHT overlay: for any key, the node must either own the key or be connected to a node that is closer to the key. This property makes it rather simple to write forwarding algorithms. Additionally, the topology of the DHT overlay must be organized so that the path between any two nodes is relatively short (so that requests for values by key are served quickly) while still keeping the degree (the number of neighbors) of the nodes as low as possible (in order to reduce maintenance and forwarding overhead).

8.1.2 Hermes

The peer-to-peer overlay network of Hermes [Pie04] is DHT-based. Events in Hermes are typed, but the system allows for clients to opt for additional content-based filtering of events (”type and attribute -based routing”). On top of the overlay network layer, there is a *event dissemination tree layer*. Event dissemination trees are constructed per event type and are used in routing events from publishers to subscribers.

Any broker (router) in Hermes can act as a *rendezvous node* that manages a particular event type. The purpose of rendezvous nodes is to ensure that the set of event dissemination trees for some event type remains consistent, that is, all brokers agree upon them. The rendezvous nodes must be globally known and available, a property satisfied by the DHT used in the overlay network; the rendezvous nodes can be found in the DHT by hashing the event type name. Further, the DHT takes care of replacing failed rendezvous nodes. There must be at least one rendezvous node for each event type. Hermes supports adding new event types at runtime. When a new type is added, it is also assigned a rendezvous node. The system supports special type creation messages that publishers can issue to their local brokers. The local broker then takes care of forwarding the message to the corresponding rendezvous node.

Subscriptions and advertisements for a particular event type are always routed towards the rendezvous node responsible for the event type. If a broker on a path from the source of the subscription/advertisement to the rendezvous node notices that a covering filter has been forwarded earlier, the message is dropped. Thus, Hermes effectively employs a covering-based routing scheme. When operating in strictly type-based mode, the coverage of two subscriptions is determined by the type hierarchy. In type and attribute -based mode, subscriptions are also forwarded towards advertisements in addition to being forwarded to the rendezvous node. This reduces the load on the rendezvous nodes as all events of the type managed by the node are not forwarded towards it, contrary to the type-based mode.

8.2 Bloom filter based routing

We now briefly discuss a routing scheme based on Bloom filters [JF08, Jer09]. First we present the basic idea of Bloom filters, and move on to describing the routing strategy and the Bloom filter -derived data structures needed in implementing it.

In tests conducted by the authors of [JF08], the performance of Bloom filter based routing surpassed that of the Siena covering-based routing scheme (both with filters posets and poset-derived forests). Usage of Bloom filters introduces the chance of false positives, i.e. notifications that are delivered uninterested subscribers, but the amount of these seems to remain on a tolerable level.

8.2.1 Bloom filters

Bloom filters [Blo70] are probabilistic data structures that are used to represent a set of elements in a compact way. They are not to be confused with the filters we have used throughout this work. A Bloom filter has a fixed size of m bits. Querying the filter for the existence of an element e in the set may yield a *false positive*; the answer may be yes even if e was never inserted in the set. There are k independent hash functions associated with a Bloom filter, each of which returns values in the range $[0, m-1]$. Each value is equally likely.

When an element is inserted into the set, it is first input to each of the hash functions. Then the bits corresponding to the values returned by the hash functions are set to 1. When the filter is queried for an element, it returns true if and only if all bits in locations returned by the hash functions are 1. This is where a false positive can occur; one or more of previous inserts may have switched the corresponding bits on. It is clear that no false negatives can occur as bits are never switched off once they are set to 1. The probability of a false positive is proportional to the number of elements stored in the Bloom filter and inversely proportional to the size m of the filter [Jer09].

If we want to allow removing elements from a Bloom filter, the data structure needs to be somewhat modified. Simply setting the bits returned by the hash functions to 0 will not do, because the same bit can be shared by multiple elements (recall that false negatives are not allowed). This limitation can be worked around by using some extra memory and storing a counter with each of the m bits of the filter. The counter indicates how many times the bit has been set. When an element is added, the counter is incremented and when an element is removed, the counter is decremented and the bit is set to 0 only if the counter has the value 0. This also implies that if an element is added twice, it also needs to be removed twice.

8.2.2 Routing strategy

The basic idea of Bloom filter based routing and forwarding is the following. Let us first consider the case of simple routing where every broker in the network knows every subscription. Now, when an event is being forwarded from its source to its destinations through the network, it is matched with the subscriptions at each router in order to determine the set of next hop destinations. This process can be improved by using *edge routing*: the set of matched subscriptions is computed at the local router of the publisher and is forwarded together with the event. It is enough for intermediate routers only to compute the set of interfaces associated with the matched subscriptions. Edge routing obviously increases the size of the event messages. Using Bloom filters in this makes the overhead quite tolerable. This, however, introduces the possibility of false positives: a non-matched subscription may be flagged as matched by the Bloom filter and the event may forwarded to nodes not interested in it. Thus, additional processing at clients or their local routers is needed.

Unfortunately, edge routing is not possible when covering-based routing is in use. However, Bloom filters can still be used to improve routing and forwarding performance in the network. The authors of [JF08] have presented three data structures for this. The `bfposet` encodes the attribute constraints of subscriptions and is used to match incoming events with subscriptions. The `bftree` or its optimized version `sbstree` is used in finding the set of next hop interfaces for matched subscriptions provided by the `bfposet`. If edge routing can be used, the matching in `bfposet` is done only at the local router of the publisher and the result is forwarded with the event. Intermediate routers need only to do matching in `bftree`/`sbstree`. If covering-based routing is used, the matching process with `bfposet` needs to take place at every router.

8.2.3 The `bfposet`

The `bfposet` consists of several posets, one for each attribute name that has been present in a subscription received by the router. The top level of a `bfposet` is a map whose keys are attribute names and values are references to the root of a poset that stores Bloom filters for all constraints for the attribute. The root node of the poset is an imaginary root `null` and the relationships in the poset follow the immediate covering relation. The most general constraints are children of the imaginary root. A Bloom filter associated with a constraint consists of a Bloom filter computed of the

Subscription	Interface	Bloom filter
$(type = T \wedge p \leq 100)$	I_0	583, 900, 1415, 4146
$(p \in [10, 90])$	I_1	2561, 2800
$(p \in [70, 95] \wedge q = 100)$	I_2	54, 1985, 4877, 12702
$(type = T \wedge q > 30)$	I_3	583, 3981, 4146, 9046
$(p \in [90, 95])$	I_4	5902, 10875
$(q > 15)$	I_5	7621, 15762

Figure 17: A set of subscriptions and their corresponding interfaces. Bloom filters computed for the subscriptions are also shown.

constraint itself, combined with the Bloom filter of the covered constraint, yielded by a bitwise OR operation over the two filters. The Bloom filter associated with the imaginary root `null` is empty, i.e. consists of all zeros.

An example set of subscriptions and their Bloom filters is shown in Figure 17. Figure 18 shows the `bfposet` formed from the subscriptions. A Bloom filter is represented by a set of index numbers. These are the indices of the bits set to 1 in the filter, as returned by the hash functions. In this case, the Bloom filter size m is 2^{14} bits and the number of hash functions used is $k = 2$. Thus, the index numbers are from the interval $[0, 16383]$ and each root constraint has two index numbers. The Bloom filter of a subscription is a combination of the Bloom filters of its individual constraints.

The purpose of the `bfposet` is to form a Bloom filter for an incoming event. For each of the attributes in the event, the top-level map of the `bfposet` is looked up for the attribute poset. Then the poset is traversed depth-first, starting from the leaves, and the first attribute constraint matching the attribute value on each path is selected. The Bloom filters of the selected constraints are combined with the Bloom filter of the event by means of bitwise OR. This way, all constraints that cover the selected constraints are also selected (and matched), because their Bloom filters are stored as a part of the Bloom filter of the selected constraint. The resulting Bloom filter is then passed to `bftree`/`sbstree` for finding the interfaces to which the event should be forwarded. As mentioned above, if edge routing can be used, then the Bloom filter for the event needs to be computed only once and can then be forwarded together with the event.

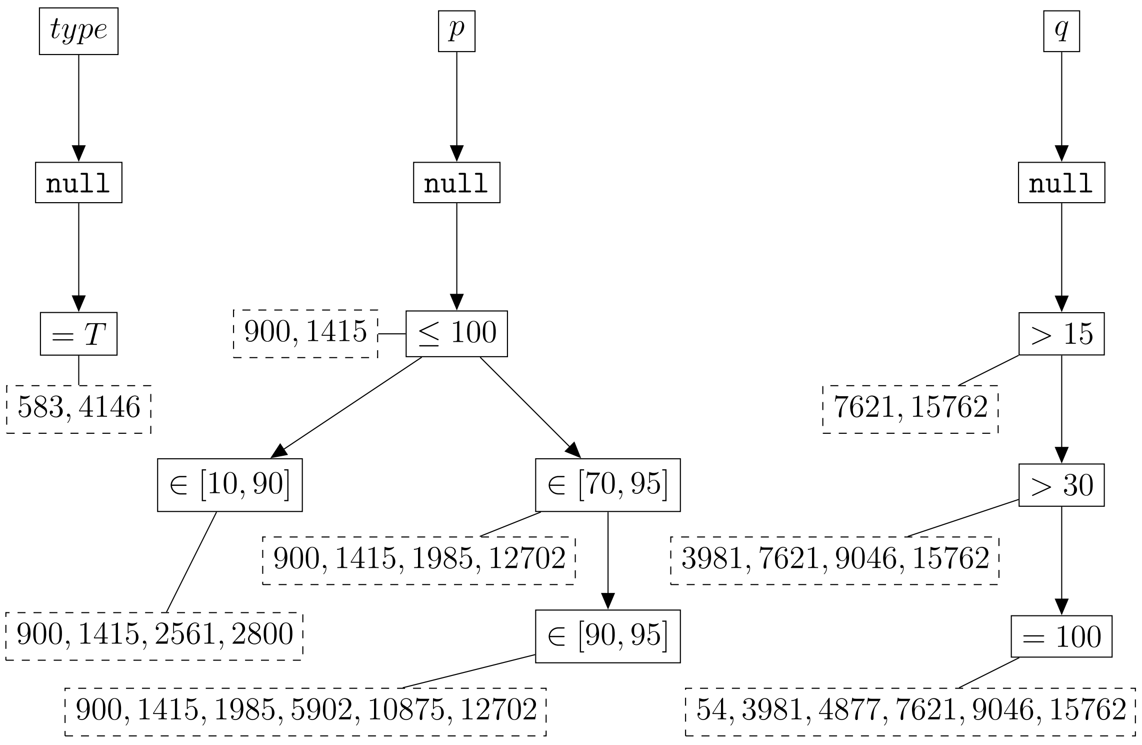


Figure 18: A `bfposet` formed from the constraints in subscriptions in Fig. 17. The Bloom filter for each constraint is represented as a set of index numbers in the dashed box next tp the constraint.

8.2.4 The `bftree` and `sbstree`

The `bftree` and its optimized version, `sbstree`, store Bloom filters for subscriptions. A `bftree` has fixed height $h+1$, where h such that the Bloom filter size m is divisible by h. In the `bftree`, the subscription Bloom filters (see Fig 17) are split into h parts so that the first part contains the first m/h bits of the Bloom filter, the second part contains the next m/h bits and so forth. Let $s_0, s_1, \ldots, s_{h-1}$ denote the h parts of some subscription Bloom filter. Parts are inserted into the tree in order starting from s_0. The insertion procedure starts from the imaginary root `null`. The part s_0 is inserted into the tree as a child of the root node and the new node is selected, unless an identical child already exists, in which case the existing node is selected. Then the procedure is repeated for the selected node and the part s_1 and again for each of the subsequent parts until a leaf is inserted or a leaf identical to the last part is found. Then the subscription source is stored in the leaf. As a result of this process, there are exactly h levels under the root in the `bftree`.

Finding the set of matched subscriptions from the `bftree` for the Bloom filter B_e of an incoming event (produced either by `bfposet` matching or edge routing) is based on the observation that an event matches a subscription s, represented by Bloom filter B_s, if $B_e \cap B_s = B_s$. An intersection of two Bloom filters can be interpreted as the set of index numbers present in both filters. The set of matched subscriptions is found by partitioning B_e into h parts $e_0, e_1, \dots e_{h-1}$ as above. Then a depth-first search is started from the root node. At each level i, the search continues to those child nodes s_{i+1} for which $e_{i+1} \cap s_{i+1} = s_{i+1}$. If the search reaches a leaf node, the interfaces stored at the leaf are added into the set of next-hop interfaces.

The number of comparisons made determines the time complexity of matching in the `bftree`. The worst case scenario occurs when an event matches all subscriptions at the router. In this case, the depth-first search has to go through all the nodes in the tree. Because of the representation of the Bloom filters, the number of nodes in the `bftree` may grow large, particularly when m and h are large. In practical applications, at least m needs to be large enough so that excessive numbers of false positives can be avoided [JF08]. By representing the Bloom filters using *sparse bit sets*, the size of the tree can be reduced considerably. The details of the representation are omitted here, but it resembles the notation used in Figures 17 and 18. The sparse representation saves space if the Bloom filter it represents has few bits set (about 1–3%) [JF08]. In practice, this condition usually holds. The `sbstree` uses the sparse representation and each node in the tree represents one bit of a subscription Bloom filter. The insert and matching algorithms are similar to the ones used in the `bftree`.

9 Conclusions

In this work, we have given a short introduction to the publish/subscribe paradigm. Our main focus has been on large-scale content-based publish/subscribe systems. We have given a detailed description of content-based networking and routing. We also addressed some data structures used in content-based routing, namely filters poset and poset-derived forest. These data structures store the subscription or advertisement filters by their covering relations. Computing these covering relations in a large data structure for complex filters is expensive.

To overcome some scalability issues related to routing tables implemented with posets or poset-derived forests, we proposed two schemes where part of the computational cost is offloaded by the router to neighboring clients. In the RTO scheme, a layered representation of the routing table is created. This representation is partitioned and sent to clients along with filter(s) to be inserted in the routing table. The actual insertion is made by the router based on the data provided by the clients. The FF scheme fixes a set of filters the clients use to positions their subscriptions.

We implemented and tested the core parts of the RTO scheme. Our implementation works correctly for posets and poset-derived forests. The test results propose that the scheme is feasible if the client based of the implementing router is large enough. Some issues, such as security and further implementation and testing, were left outside the scope of this work. Instead, we propose that these issues be addressed in future works.

We also briefly explored alternative ways of improving scalability in large-scale pub/sub systems. As examples, we used DHT-based systems and Bloom filter based routing.

References

ASS$^+$99 Aguilera, M. K., Strom, R. E., Sturman, D. C., Astley, M. and Chandra, T. D., Matching events in a content-based subscription system. *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 1999, ACM, pages 53–61.

AT05 Aekaterinidis, I. and Triantafillou, P., Internet scale string attribute publish/subscribe data networks. *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, New York, NY, USA, 2005, ACM, pages 44–51.

BCM$^+$99 Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R. E. and Sturman, D. C., An efficient multicast protocol for content-based publish-subscribe systems. *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Washington, DC, USA, 1999, IEEE Computer Society, pages 262–272.

Blo70 Bloom, B. H., Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13,7(1970), pages 422–426.

CBGM03 Crespo, A., Buyukkokten, O. and Garcia-Molina, H., Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering*, 15, pages 174–191.

CCC$^+$01 Campailla, A., Chaki, S., Clarke, E., Jha, S. and Veith, H., Efficient filtering in publish-subscribe systems using binary decision diagrams. *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, Washington, DC, USA, 2001, IEEE Computer Society, pages 443–452.

CRW00 Carzaniga, A., Rosenblum, D. S. and Wolf, A. L., Achieving scalability and expressiveness in an internet-scale event notification service. *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, 2000, ACM, pages 219–227.

CRW01 Carzaniga, A., Rosenblum, D. S. and Wolf, A. L., Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19,3(2001), pages 332–383.

CRW04 Carzaniga, A., Rutherford, M. J. and Wolf, A. L., A routing scheme for content-based networking. *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China, March 2004.

CW02 Carzaniga, A. and Wolf, A. L., Content-based networking: A new communication infrastructure. *IMWS '01: Revised Papers from the NSF Workshop on Developing an Infrastructure for Mobile and Wireless Systems*, London, UK, 2002, Springer-Verlag, pages 59–68.

CW03 Carzaniga, A. and Wolf, A. L., Forwarding in a content-based network. *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, 2003, ACM, pages 163–174.

EFGK03 Eugster, P. T., Felber, P. A., Guerraoui, R. and Kermarrec, A.-M., The many faces of publish/subscribe. *ACM Comput. Surv.*, 35,2(2003), pages 114–131.

FJL$^+$01 Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A. and Shasha, D., Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2001, ACM, pages 115–126.

Jer09 Jerzak, Z., *XSiena: The Content-Based Publish/Subscribe System.* Ph.D. thesis, Dresden University of Technology, 2009.

JF08 Jerzak, Z. and Fetzer, C., Bloom filter based routing for content-based publish/subscribe. *DEBS '08: Proceedings of the second international conference on Distributed event-based systems*, New York, NY, USA, 2008, ACM, pages 71–81.

JF09 Jerzak, Z. and Fetzer, C., Soft state in publish/subscribe. *DEBS '09: Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, New York, NY, USA, 2009, ACM, pages 1–12.

MFGB02 Mühl, G., Fiege, L., Gärtner, F. C. and Buchmann, A., Evaluating advanced routing algorithms for content-based publish/subscribe systems. *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, Washington, DC, USA, 2002, IEEE Computer Society, page 167.

Müh02 Mühl, G., *Large-Scale Content-Based Publish/Subscribe Systems.* Ph.D. thesis, Darmstadt University of Technology, 2002.

MNJH08 Moskowitz, R., Nikander, P., Jokela, P. and Henderson, T., Host Identity Protocol, RFC 5201 (Experimental), April 2008. URL `http://www.ietf.org/rfc/rfc5201.txt`.

Pie04 Pietzuch, P. R., *Hermes: A Scalable Event-Based Middleware.* Ph.D. thesis, Queens College, University of Cambridge, February 2004.

RM06 Rescorla, E. and Modadugu, N., Datagram Transport Layer Security, RFC 4347 (Proposed Standard), April 2006. URL `http://www.ietf.org/rfc/rfc4347.txt`. Updated by RFC 5746.

Tar07	Tarkoma, S., Chained forests for fast subsumption matching. *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, New York, NY, USA, 2007, ACM, pages 97–102.
Tar08	Tarkoma, S., Fast track article: Dynamic filter merging and mergeability detection for publish/subscribe. *Pervasive Mob. Comput.*, 4,5(2008), pages 681–696.
TDVK99	Tewari, R., Dahlin, M., Vin, H. M. and Kay, J. S., Design considerations for distributed caching on the internet. *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, Washington, DC, USA, 1999, IEEE Computer Society, page 273.
TK05	Tarkoma, S. and Kangasharju, J., Filter merging for efficient information dissemination. *Proceedings of the 13th International Conference on Cooperative Information Systems*, 2005, pages 274–291.
TK06	Tarkoma, S. and Kangasharju, J., Optimizing content-based routers: Posets and forests. *Distributed Computing*, 19,1(2006), pages 62–77.
YGM94	Yan, T. W. and García-Molina, H., Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19,2(1994), pages 332–364.