

Bloom Filter Based Routing for Content-Based Publish/Subscribe

Zbigniew Jerzak
Dresden University of Technology
Systems Engineering Group
D-01062 Dresden, Germany
Zbigniew.Jerzak@tu-dresden.de

Christof Fetzer
Dresden University of Technology
Systems Engineering Group
D-01062 Dresden, Germany
Christof.Fetzer@tu-dresden.de

ABSTRACT

Achieving expressive and efficient content-based routing in publish/subscribe systems is a difficult problem. Traditional approaches prove to be either inefficient or severely limited in their expressiveness and flexibility. We present a novel routing method, based on Bloom filters, which shows high efficiency while simultaneously preserving the flexibility of content-based schemes. The resulting implementation is a fast, flexible and fully decoupled content-based publish/subscribe system.

Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems—*Distributed Applications*; D.1.3 [Programming Techniques]: Concurrent Programming—*Distributed Programming*

General Terms

Algorithms, Design

Keywords

distribution, interaction, publish/subscribe, Bloom filter

1. INTRODUCTION

The ultimate goal of publish/subscribe (Pub/Sub) systems is to provide a (1) fast, (2) flexible and (3) decoupled infrastructure for information exchange [9]. The Pub/Sub communication model is inherently *asynchronous*: neither publishers (senders) nor subscribers (receivers) are blocked. Both subscribers and publishers are *anonymous* to each other and the system as a whole. This implies that there are no explicit source and destination addresses. There is also no requirement on the communication between any two components to take place *simultaneously*, which is a strengthened derivative of the asynchronous property. The three above mentioned properties allow for the creation of systems which are decoupled in *synchronization*, *space* and *time* domains. A Pub/Sub system facilitates publishers and subscribers to

communicate based only on the content of the messages they exchange. Specifically, this implies that the content of the messages fulfills the role traditionally played by the explicit source and destination addressing. Among many variants of such addressing the most prominent are: subject-, type- and content-based [18] ones. Among those three, content-based addressing is characterized by the highest flexibility, allowing communicating parties to express their interest in certain information by the terms of conjunction of arbitrary boolean predicates over the whole content of messages.

However, both flexibility and decoupling do not come without a price. In case of a content-based Pub/Sub systems, this is the speed of information routing. It is intuitive that addressing requiring evaluation of predicates over the content of the message must be slower than the addressing based on comparison of short sequences of numbers. More formally, it has recently been shown [22] that there does not exist any worst case computationally efficient solution for the content-based addressing problem¹. Moreover, it has been formally proven in [15], that the hardness of content matching is equivalent to the Partial Match [13] problem.

In the described work, we seek to provide an efficient solution to the issue of content-based addressing. The main contribution of this paper is the development of the scheme for efficient, content-based addressing based on the Bloom filters [2, 10]. The developed scheme does not violate any of the decoupling properties, nor does it impede the flexibility of the Publish/Subscribe paradigm. We provide a demonstration of the applicability of our approach using *XSienna* – a system developed by the authors, based on the well established SIENA[5] architecture.

The remainder of this paper is structured as follows: in Section 2, we present the related work covering previous attempts at solving the content-based addressing complexity problem. In Section 3, we give a brief introduction to the Pub/Sub systems along with the definitions of terms used further in the paper. Section 4 discusses the concept behind the proposed Bloom filter based routing and relates it to other approaches. Section 5 gives the details regarding the implementation of the proposed routing scheme and is followed by the Section 6 which presents the results of evaluation. We conclude with Section 7.

¹Content-based addressing problem can be reduced to the point dominance problem, which itself is a special case of range-searching problem.

2. RELATED WORK

The problem of fast content-based addressing has been already addressed in the past. We can divide the publications dealing with that issue into three coarse groups. First group, is composed of papers trying to improve the routing structures, by re-implementing or re-organizing them. The second group tries to tackle the problem by altering the way the routing process works, which simultaneously enforces the change of the routing structures. Finally, the third group tries to improve routing performance by implementing a routing overlay which reorganizes the brokers so as to exploit the subscription similarities.

Among the earliest publications from the first group one has to mention [6]. The authors propose to use an extension of a counting algorithm which handles both conjunctions and disjunctions of subscriptions. The main issue with the presented approach is the prohibitive cost of updates to the routing table. Another approach has been presented in [25] and [24] which extended the original content-based addressing scheme presented in [5] to handle and store messages more efficiently. In [25], the authors substantially improved over the original approach as far as storing and management of subscriptions is concerned, however, the event forwarding remained virtually unchanged. In [24], the authors addressed the problem of event forwarding, however, they have introduced a requirement for profiles summarizing the events content.

The first related set of publications from the second group is [26, 27, 1]. The authors propose to use Bloom filters [2] to aggregate, filter and match information in subscriptions, thus creating per broker subscription summaries compacting subscription information. However, the proposed approach requires a known, limited set of possible publication and subscription attributes, thus limiting the expressiveness of the content-based pub/sub. A similar approach (based on [26]) to efficient content-based addressing has been presented in [30]. The authors combine Bloom filter based subscription summaries and an on-demand multicast routing protocol, however, due to the assumptions shared with the [26], the limitations are also similar. Moreover, our approach differs in that we do not require a periodic broadcast of advertisements (specifically, we do not require advertisements at all) for route creation and we do not need to maintain groups for a given publication. The authors of [3, 4] propose to combine content-based addressing with a multicast-based approach. Presented solutions require however that every broker has to be aware of all other brokers in the network. This limitation can be alleviated by introduction of partitions. However, the partition management and content space partitioning imposes new overheads. Specifically, the content space partitioning (based on R-Trees [12]) described in the [29] requires a predefined and known set of attributes for every event, thus substantially limiting the expressiveness and flexibility of the whole content-based system. Another approach has been developed in [14] which uses prefix routing, the limiting factor of that solution is the need for a uniform layout of the routing structures for all brokers. Ensuring such a common layout might be too expensive and difficult in WANs. Another approach has been proposed in [21], where the content-based addressing has been combined with hash-based matching so that downstream brokers

can reuse the matching results from the upstream brokers. This approach can however produce false negatives whenever it relies on the hash-based matching only. To eliminate false negatives, hash-based matching can only be used to quickly select a subset of subscriptions which have to be matched using content-based algorithms.

The alteration of the broker network, the strategy from the third group, has been proposed in the [7]. The main problem with such a design is the need to avoid hot-spots in the broker network and the problem with calculation of the similarity metrics for proper broker arrangement. Another class of Pub/Sub systems in this group are the ones based on the DHTs [28, 20]. Although DHT-based approaches usually expose a superior matching times they require typed subscriptions and publications (thus limiting the flexibility of the content-based publish/subscribe), globally unique node ids, they require additional mechanisms to account for the locality of the events and they usually expose an additional overhead associated with the stretch of the underlying DHT.

3. BACKGROUND

Within this section, we introduce some basic concepts related to the Pub/Sub systems that are used in the remainder of this paper. We also introduce some terminology which is mostly derived from the terminology used in [5]. We highlight the relevant terms using *italics*.

Pub/Sub systems allow for a decoupled communication between *subscribers* and *publishers*. The communication is message based, with subscribers issuing *subscriptions*² and publishers issuing *events*. We assume a popular paradigm used by many publish/subscribe systems [5, 17, 20, 31, 11, 25, 23], where a subscription s is a conjunction of n *predicates*:

$$s = \{p_1 \wedge \dots \wedge p_n\} \quad (1)$$

A predicate p_i is a function which evaluates to either true or false for a given argument. In Pub/Sub systems, predicates consist of *attribute names* (an) and *attribute constraints* (ac):

$$p_i = \{an, ac\} \quad (2)$$

An attribute constraint consists of an *operator* and an *attribute value* (av). An event e is a disjunction of attribute name and attribute value pairs:

$$e = \{\{an_1, av_1\} \vee \dots \vee \{an_n, av_n\}\} \quad (3)$$

Events do not need to store operators, which are always implicitly an equality. Attribute name and value pairs from events are arguments for the predicate functions. A sample subscription can be represented as: $\{x > 5; y > 3\}$; a sample event has the following form: $\{x 0; z 3\}$.

The actual fabric that connects and simultaneously decouples subscribers and publishers is the set of *brokers* forming a network. Brokers are responsible for subscription *routing* and event *forwarding*. In order for the communication to take place subscriptions need to be routed and stored by all brokers in the Pub/Sub network. Subscriptions are

²We deliberately choose not use the other common term – *filter* so as to avoid confusion with Bloom filters.

stored in *routing tables*, which are used for event forwarding. Forwarding is performed by evaluating the predicates of stored subscriptions over attribute name and value pairs of the incoming events. This process is referred to as *matching*. The result of matching an event against a set of stored subscriptions is the *forwards set* containing addresses of brokers/subscribers which should have the event forwarded to them. Performing a matching operation in every broker will eventually lead to the delivery of the event to the subscriber which issued a subscription.

An important optimization of content-based routing and forwarding is the concept of the *binary coverage relation* between subscriptions. Subscription s_1 covers subscription s_2 if it matches the superset of events matched by s_2 . We might also say that s_1 is *more general* than s_2 . The usefulness of this concept is manifested by the fact that covered subscriptions do not need to be routed on the links which had covering subscriptions sent previously over them. This reduces both bandwidth consumption and storage overhead. The coverage relation, denoted as \succ is:

- reflexive: $s_1 \succ s_1$ always holds
- antisymmetric: if $s_1 \succ s_2$ and $s_2 \succ s_1$ then $s_1 = s_2$
- transitive: if $s_1 \succ s_2$ and $s_2 \succ s_3$ then $s_1 \succ s_3$

However, it is not total, which implies that it creates a partial order for all subscriptions. Therefore, the routing tables, storing subscriptions in brokers, are sometimes referred to as *posets* – partially ordered sets.

In this paper, we make extensive use of *Bloom filters*, which are a compact method to represent a set of n elements using a vector of m bits. A Bloom filter uses k independent hash functions which take as parameter an element to store and produce a hash of m bits. Storing an element equals to setting k bits in the Bloom filter, which correspond to the return values of the k hash functions. Bloom filters are characterized by a certain *false positives probability* which implies a probability for two different elements having k identical hashes. Intuitively, the false positives probability is proportional to the number of elements n stored in the Bloom filter and inversely proportional to the size m of the bit vector.

4. ROUTING

The traditional routing approach outlined in Section 3 has two main issues: (1) the evaluation of predicate functions over events is slow and (2) it has to be performed by every broker an event is forwarded to. Thus, the more brokers an event passes on its way, the higher the delay and processing overhead. Section 2 briefly outlined other approaches to solving above issues, however, they either did not succeed to preserve the flexibility of Pub/Sub communication scheme or introduced additional overheads. In this section, we outline our approach to coping with the speed of content-based addressing, while preserving its flexibility.

4.1 Principles

When building our system, we have focused on two main issues: (1) possible elimination of the content-based addressing at all but the incoming edge broker (or, the publisher)

and (2) increasing the speed of content-based matching. In order to achieve this, we have to consider the relation between the predicates of a subscription and attribute name and value pairs of an event. We can formally write that the event e matches the subscription s (denoted by $e \lesssim s$) iff:

$$\forall p \in s \quad \exists_{\{an, av\} \in e} : p(\{an, av\}) = \text{true} \quad (4)$$

where $p()$ is a predicate function. Since a broker contains a set of \mathbb{S} subscriptions, the above process has to be repeated for all $s \in \mathbb{S}$. An event e is then forwarded to a given address if:

$$\exists_{s \in \mathbb{S}_{addr}} : e \lesssim s \quad (5)$$

where \mathbb{S}_{addr} is a set of subscriptions which arrived from given address. From that, we can conclude that the event matching problem can be split in two parts: (1) evaluation of predicate functions with the attribute name and value pairs ($p(\{an, av\}) = \text{true}$) and (2) calculation of a disjunction of conjunctions of the predicate function's results – $\exists_{s \in \mathbb{S}_{addr}}$ and $\forall p \in s$. We also know that subscriptions are always propagated to all brokers in the network. Combining these two facts allows us to come up with an approach where the process of predicate evaluation must be carried out only once for each event in the first (edge) broker an event traverses. Let us assume that we are able to store the result of this evaluation and propagate it along with the event. This would imply that in all subsequent brokers, we only need to evaluate the conjunction of the transmitted predicate results (forming a single subscription s) followed by a disjunction over the set \mathbb{S}_{addr} of all subscriptions. This in turn would deliver us a valid forwards set for an event without the need to evaluate the predicate functions.

However, an important issue we need to consider is the coverage. Due to the coverage relation, not all brokers will receive all subscriptions. Specifically, if the Pub/Sub system has reached a stable routing state (subscription traffic has ceased) all brokers will only share a common set of the most general subscriptions. This implies the impossibility of the edge routing in case where covering relation is used to optimize the dissemination of subscriptions, which has also been implicitly shown in [14]. Therefore, we propose to use a dual routing strategy: (1) edge routing can be used when covering relation is not optimizing the subscription propagation, and (2) fast content-based matching is performed when the subscription traffic is limited by the covering relation. Another issue we have approached is the efficient representation of the conjunction over the set of predicates in a single subscription followed by a disjunction over the set of all subscriptions stored in the broker. We will address these issues by presenting three routing structures we have developed: the **bfposet** used for the storing and encoding of the evaluation of the predicate functions and **bftree** (along with its optimized version **sbstree** – see Section 4.5) used for the representation of the disjunction of conjunctions of predicate values.

4.2 Overview

In this section, we give a brief overview of the routing process adhering to the principles presented in Section 4.1. The details of the routing process are explained in Sections 4.3, 4.4 and 4.5.

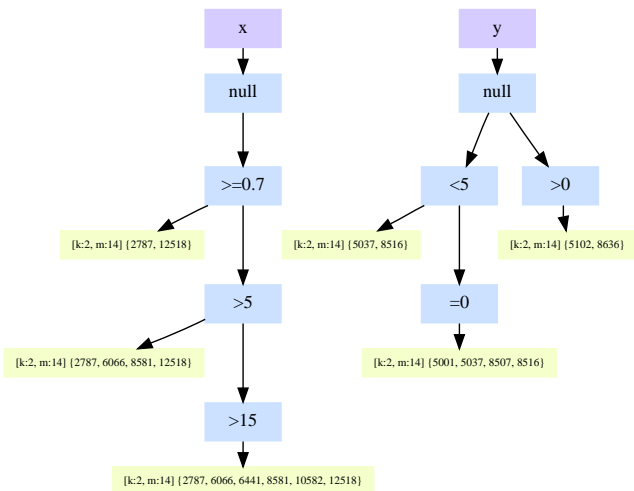


Figure 1: The bfposet layout as produced by dot

The routing processes requires the presence of subscriptions. Subscriptions arriving at the broker are stored in two data structures: the **bfposet** and **bftree/sbstree**. **bfposet** stores the predicates of the subscriptions while **bftree/sbstree** stores a disjunction of conjunctions of encoded subscription predicates. We encode subscriptions using Bloom filters. An event incoming at the edge broker has to be matched based on its content with the **bfposet**. The result of the content-based match is the Bloom filter encoding the matching predicates, which is assigned to the event. This Bloom filter is subsequently passed to the **bftree/sbstree** which computes the disjunction of conjunctions over the encoded subscriptions predicates and returns the set of interfaces to forward the event on. If no covering relation is used to optimize subscription traffic, subsequent brokers encountered by the event only evaluate the attached Bloom filter using the **bftree/sbstree**. Otherwise, the matching in the **bfposet** has to be repeated.

4.3 bfposet

The **bfposet** stores subscriptions as disjunction of single predicates. This allows us to abstract away the conjunctive form of a subscription allowing for a more efficient storage structure. The **bfposet** (see Figure 1) is a hybrid data structure. The top level of the **bfposet** is constructed as a sorted map. We use the attribute names of the stored predicates as the keys for the sorted map – in case of the Figure 1 these are x and y . Every key (attribute name) points to a partially ordered set, beginning with a virtual root node $\{null\}$, containing the attribute constraints sharing the same attribute name. The structure of the attribute constraints reflects the partial order from the covering relation. Every attribute constraint has a Bloom filter associated with it. Bloom filters are created when attribute constraints are inserted into the **bfposet**, with the hash function evaluated over the attribute name and the attribute constraints, thus ensuring that every attribute constraint in the **bfposet** has a unique Bloom filter. Only in rare case of a false positive can two attribute constraints have two identical Bloom filters. In Figure 1, attribute constraint Bloom filters are represented by a tuple stating the Bloom filter size in bits and the num-

ber of hash functions used, followed by the indexes of the bits equal to 1. We can observe that 2 hash functions were used and the size of the filter was 2^{14} bits – $[k : 2, m : 14]$. The indexes of bits which are set are enclosed in the curly braces.

An important property which is maintained in the **bfposet** is the coverage relation of the attribute constraints, represented by the Bloom filters. Every attribute constraint which is covered by another attribute constraint reflects this fact in its own Bloom filter by performing an *OR* operation with the Bloom filter of the covering attribute constraint. A single Bloom filter reflects than the whole chain of covering attribute constraints. The **bfposet** presented on Figure 1 contains subscriptions presented in Table 1. We can ob-

Table 1: The set of subscriptions used in examples

Subscription	Source	Bloom filter
$\{x > 5\}$	$f1@dot.com$	6066,8581
$\{x \geq 0.7\}$	$f2@dot.com$	2787,12518
$\{x > 15\}$	$f3@dot.com$	6441,10582
$\{y < 5\}$	$f4@dot.com$	5037,8516
$\{y = 0\}$	$f5@dot.com$	5001,8507
$\{x > 15, y > 0\}$	$f6@dot.com$	5102,6441,8636,10582

serve that every covered subscription contains the bits from the covering Bloom filters. An interesting property of the **bfposet** is that, in some cases, it can reduce the number of stored predicates in comparison to a structure storing subscriptions in conjunctive form. A simple example are the subscriptions $\{x > 15, y > 0\}$ and $\{x > 15\}$ from the above table, which in the **bfposet** are represented as two predicates, while, e.g., a SIENA poset would require the storage of three predicates. The removal of subscriptions from the **bfposet** using a counting Bloom filter [10] is also straightforward.

The matching of an incoming event with the **bfposet** is executed in the following way: for every attribute name in the event an appropriate partially ordered set for which the attribute name is a key is selected. Subsequently, the attribute constraint matching the attribute value of the event is selected. The selected attribute constraint is the first attribute constraint when moving from the leaf to the root of the partially ordered set, i.e., it is the most general attribute constraint matching the attribute value. In the final step, the Bloom filter of the selected attribute constraint (which contains the bloom filters of all covered attribute constraints) is added to the Bloom filter of the event by the means of an *OR* operation.

4.4 bftree

The main task of the **bftree** is to represent the disjunction of conjunctions of predicate values. Moreover, the **bftree** itself does not store any predicates, instead, it works exclusively with Bloom filters representing subscriptions and events. In contrast to the **bfposet**, **bftree** stores subscriptions in their conjunctive form. A conjunctive form of a subscription is expressed using a Bloom filter. We describe **bftree** to explain the basic principles of event and subscrip-

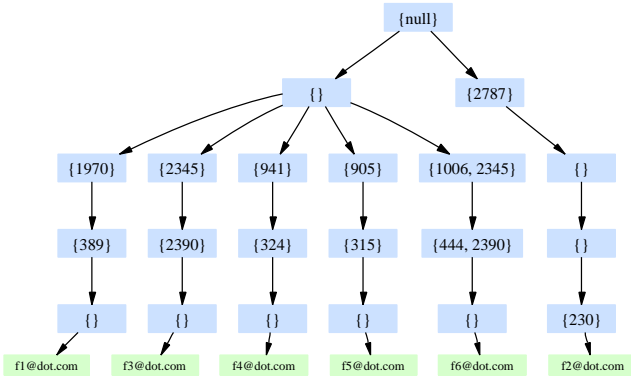


Figure 2: The bftree layout as produced by dot

tion matching, however, in our implementation we use an improved version (**sbftree**) which is described in Section 4.5.

The disjunction of the predicates of a single subscription can be expressed with a single Bloom filter using an *OR* operation over Bloom filters of the single predicates. However, this approach cannot be used to represent a conjunction of subscription predicates which could be matched with an event. Therefore, we use the **bftree** structure to cope with that issue. Figure 2 shows the layout of the **bftree** created by the **dot**³ program. The **bftree** contains the Bloom filters of subscriptions shown in Table 1.

In order to convert the disjunction of subscription predicates encoded into a Bloom filter into a conjunction, a Bloom filter is stored in the **bftree** split into h disjoint parts $p_0^s \dots p_{h-1}^s$, with each part p_i^s representing an equal portion of the Bloom filter’s bits. For example, a Bloom filter with the width of $m = 2^{14}$ bits can be split into $h = 2^2$ parts with each of them having 2^{12} bits. Each such part is subsequently inserted into the **bftree**. The insertion algorithm starts from the virtual root node $\{null\}$ and inserts the part number p_{i+1}^s as a child node of the previously inserted part p_i^s creating a new **bftree** node n_{i+1} . If a part p_i^s to be inserted as a child of the node n_{i-1} is identical with one of the node’s children n_i , then p_i^s is not inserted and the algorithm tries to insert p_{i+1} as the child of the node n_i . This optimizes the space usage by not duplicating identical parts of the bloom filter. Such representation of the tree implies that its height is always equal to the value of h . The height of the **bftree** in Figure 2 is constant with the leaf nodes representing the source addresses of the subscriptions. Moreover, every branch has $k|\mathbb{P}|$ bits set, where \mathbb{P} is the set of all predicates in a subscription forming the branch. Since the **bftree** does not store predicates, subscriptions can be recognized by their source addresses in the leaves of the tree.

The forwarding process for an incoming event follows the similar algorithm. An incoming event has been matched with a **bfposet**. Hence it has been assigned a Bloom filter which represents a disjunction of all predicates which evaluated to true with the attribute name and value pairs of the event. The main issue forwarding process has to solve is the enforcement of the conjunction of the predicates of a single

subscription presented in Equation 4. When we consider a Bloom filter of an event \mathbb{B}_e and a Bloom filter of a subscription \mathbb{B}_s we can enforce the conjunction of the predicates in the subscription by requiring that:

$$\mathbb{B}_s \cap \mathbb{B}_e = \mathbb{B}_s \quad (6)$$

The actual implementation of the Equation 6 is straightforward given the storage format of the Bloom filters of subscriptions in the **bftree**. The Bloom filter of the event is partitioned into h parts $p_0^e \dots p_{h-1}^e$ using the same algorithm as for the partitioning of the Bloom filter of a subscription. Subsequently, starting at the virtual root node $\{null\}$ and the first part p_0^e of the event Bloom filter algorithm finds a child node n_1 which fulfills Equation 6, i.e., $p_0^e \cap n_1 = n_1$. If such a node has been found, the algorithm recurses in that it replaces $\{null\}$ with n_1 and p_0^e with p_1^e . The termination condition is expressed as either not finding a node fulfilling Equation 6 or reaching a leaf node with a source address, which is added to the list of matching addresses.

4.5 sbstree

Considering the properties of the **bftree** structure, we have noticed an issue which has forced us to change its design and create a tree structure (**sbstree**) based on the sparse bit set representing a Bloom filter. The issue of the **bftree** structure is the fact that a **bftree** is characterized by worst case complexity of $\mathcal{O}(2^m)$. Consider a **bftree** storing Bloom filters with the width of m bits. If we assume that every node in the **bftree** holds only one bit, we can see that the tree has a total height of m with the number of nodes N equal to:

$$N = \sum_{i=1}^m 2^i = 2^{m+1} - 2 \quad (7)$$

The total number of nodes is simultaneously equal to the total number of comparisons one has to perform in order to match an event which has a Bloom filter with all bits set. Hence, the worst case complexity is $\mathcal{O}(2^m)$. With this in mind, we have decided to use a structure based on sparse bit sets – the **sbstree**. A bit set represents a Bloom filter as an array of 64 bit integer values:

$$\underbrace{0 \dots 63}_0, \underbrace{0 \dots 63}_1, \dots, \underbrace{0 \dots 63}_{\frac{m}{64}} \quad (8)$$

Setting a bit number b in a bit set based Bloom filter corresponds to setting a bit number b *AND* 63 in the integer number $b \gg 64$, where *AND* is a bitwise AND operation and \gg is a bitwise shift right operation. A sparse bit set represents a Bloom filter using an array of integers representing the indices of bits which are set. Intuitively, there is a tradeoff in space efficiency, with a sparse representation being more efficient if a Bloom filter is holding less than $\frac{m}{\log_2(m)}$ elements, where m is the size of the Bloom filter.

For a sparse bit set using $\log_2(m) = 32$ bit integer values this corresponds to a fill ratio of 3% – i.e., the number of bits set in the Bloom filter divided by the total numbers of bits m . Figure 4 shows the fill ratios of event Bloom filters under different types of workload. Even with the highly skewed Pareto workload, we can observe that for a Bloom filter with size of $m = 2^{14}$ and 30,000 subscriptions, we get an average fill ratio of 1%, which is three times more space

³<http://www.graphviz.org/>

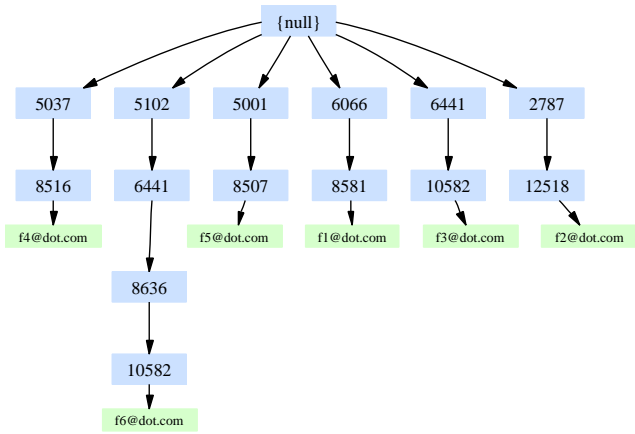


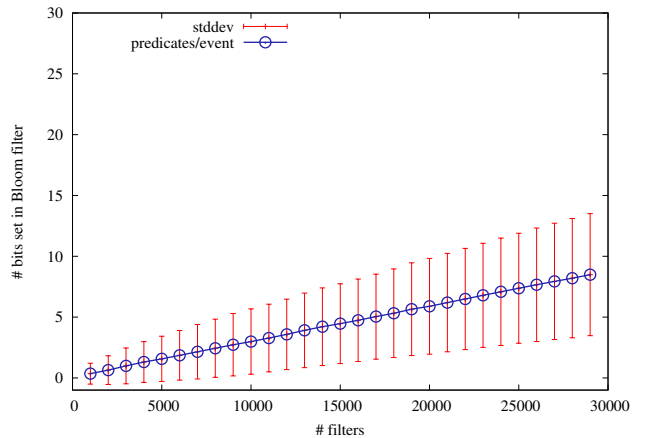
Figure 3: The sbstree layout as produced by dot

efficient than a bit set - even if we use 32 bit integers in the sparse bit set.

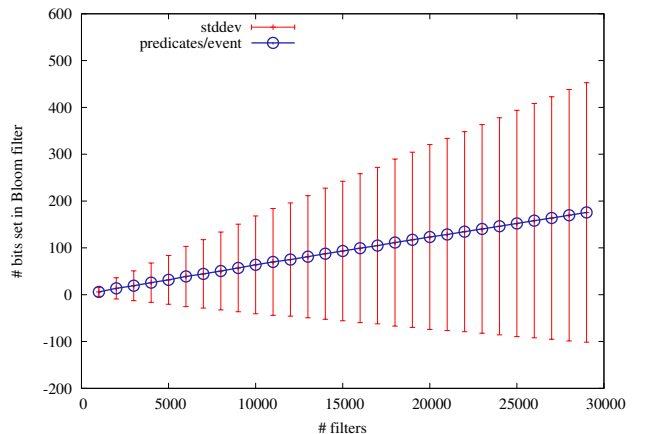
A sparse bit set not only provides advantages in terms of space complexity but it is also beneficial in terms of performance when used in the **sbstree**. The construction of the **sbstree** from a Bloom filter represented by a sparse bit set is carried out by inserting the integer values representing the set bits of the Bloom filter into the **sbstree** – see Figure 3. The insertion process starts at the virtual root node $\{null\}$ of the **sbstree** and searches for a node which has the same value as the first bit set in the sparse bit set. If such node is found, the process continues with this node as a new root node and the next bit from the sparse bit set. If such node has not been found the remaining bits from the sparse bit set are inserted as children of each other. Figure 3 shows the layout of the sparse bit set after the insertion of the set of subscriptions presented in Table 1. The matching process starts at the virtual root $\{null\}$ of the **sbstree**. For every integer representing a set bit from the sparse Bloom filter, a comparison with children of the root node is performed. In case of a match, the procedure is repeated starting with the matching node and the next set bit after the bit which has been matched. Even though, the **sbstree** has an inferior worst case complexity:

$$\sum_{i=1}^m i2^i = (m-1)2^{m+1} \quad (9)$$

in comparison to the **bftree** our evaluation has shown that it superior in both insertion and matching. A simple case can be constructed if we consider a comparison between two bit sets of width $m = 2^{14}$ with only one bit, number $2^{14} - 1$, set. Such comparison would require the traversal of $\frac{2^{14}}{bits}$ integer values, where *bits* is the number of bits used in the integer representation. Using a sparse bit set the overhead is reduced to a single comparison of two integer values. The argument is strengthened if we recall the Figure 4 illustrating the low number of predicates matching an average event. Furthermore, when storing integers in a sorted array both in the sparse bit set and in the **sbstree** nodes one can quickly eliminate whole ranges of **sbstree** child nodes during the matching process. The API of the sparse bit set is identical to the one presented in Listing 2, the only difference is that



(a) normal attribute name distribution



(b) Pareto attribute name distribution ($\alpha = 1, \beta = 3$)

Figure 4: The number of predicates matching a single event

IBFTree uses a sparse bit set instead of a Bloom filter.

5. IMPLEMENTATION

In order to evaluate the design presented in the Section 4, we have implemented the presented algorithms and incorporated them into our Pub/Sub system $\mathcal{X}Siena$. $\mathcal{X}Siena$ can be seen as a derivative of $SIENA$ system⁴ by Antonio Carzaniga. Among the most notable differences are: (1) the support for non-blocking I/O using the Apache Mina⁵, (2) support for acyclic graph topologies, and (3) modular design allowing for easy extension. $\mathcal{X}Siena$ is available for download from: <http://wwwse.inf.tu-dresden.de/xsiena>.

In order to compare the proposed approach with legacy routing strategies, we have used a standard Siena partially ordered set, further referred to as **siena** and a forest data structure proposed in [25], to which we refer as **forest**. The routing structures which are our contribution and the effect of the design presented in Sections 4.3 and 4.4 are further referred to as **bfpset** and **bftree**.

⁴<http://www.inf.unisi.ch/carzaniga/siena/software/>

⁵<http://mina.apache.org/>

Listing 1: Traditional Publish/Subscribe routing interface.

```

1 public interface IPoset {
2     //store subscription
3     Set<Addr> subscribe(Subscription s, Addr s);
4     //calculate forward set for an event
5     Set<Addr> publish(Event e);
6 }

```

Listing 2: Bloom filter based routing interface.

```

1 public interface IBFPoset {
2     //store subscription in routing structure
3     Set<Addr> subscribe(Subscription s, Addr s);
4     //calculate Bloom filter for an event
5     Bloom match(Event e);
6 }
7
8 public interface IBFTree {
9     //store the subscription's Bloom filter
10    void insert(Bloom filter, Addr s);
11    //calculate forward set for an event
12    Set<Addr> publish(Bloom filter);
13 }

```

5.1 API

The traditional stateful API for the Pub/Sub systems [19] is presented in Listing 1. Using this API, which is common to all implementations using a single partially ordered set, an incoming subscription is stored in the implementation of the `IPoset` interface using the `subscribe()` method. In case of *XSiena*, these implementations are the `siena` and `forest` structures. The returned set of addresses indicates contacts which should receive this subscription. An incoming event is matched with stored subscriptions using `publish()` method, which returns the forwards set for this event. The `publish()` invocation is performed on every subsequent broker which receives the event.

Listing 2 presents interfaces which are the result of the design presented in Section 4. An implementation of `IBFPoset` (`bfposet`) stores subscriptions decomposed into single predicates along with accompanying Bloom filters. A subscription arriving from address `s` is first subject to `subscribe()` method, which stores it and returns a set of addresses it has to be routed to. Subsequently, its Bloom filter is passed as a parameter to the method `insert()` from the implementation of `IBFTree` – `bftree`. It constructs the forwarding tree for matching with incoming events. An incoming event arriving at the edge broker is passed as parameter to `match()` method from implementation of `IBFPoset`. In case of edge routing, this method is invoked only once in a lifetime of an event and assigns it a Bloom filter, which is subsequently a parameter to the `publish()` method which returns the set of interested brokers and/or subscribers. The `publish()` method is always executed by all brokers an event passes on its way towards a subscriber.

5.2 Bloom Filters

One of the main issues, when implementing Bloom filters is the proper selection of the parameters which influence the Bloom filter behavior. These parameters are: size m of the

Bloom filter in bits, number of hash functions k used to hash the input data, number of elements n stored in the Bloom filter and the probability p of false positives. Based on [10], we can say that the relation between the parameters can be given as a set of equations:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \quad (10)$$

$$k = \frac{m}{n} \ln 2 \quad (11)$$

The value k is chosen so as to minimize the probability p of false positives. Using the value k from Equation 11 in Equation 10, we can obtain the probability as a function of only two parameters: m and n . This in turn allows us to easily calculate the desired size of the Bloom filter as a function of the false positives probability p and expected number of elements n . Figure 5 shows the relation between p , m and n for the value of k given in Equation 11. In

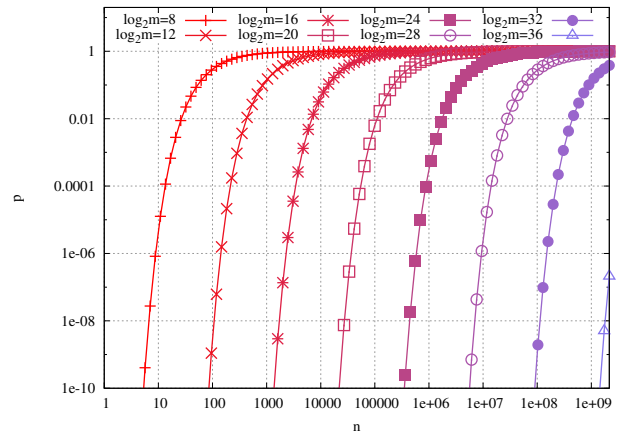


Figure 5: The false positive probability p as a function of number of elements n and the size m of a Bloom filter.

order to reduce the need for computation of possibly large number of different hash functions, we simulate additional hash functions using double hashing [8]:

$$g_i(x) = h_1(x) + ih_2(x) \quad (12)$$

The authors of [16] have shown that only two hash functions $h_1(x)$ and $h_2(x)$ are necessary to effectively implement a Bloom filter without any loss in the asymptotic false positive probability.

6. RESULTS

This section presents the evaluation of the Bloom filter based routing and its comparison to current approaches. The evaluation is based on the *XSiena* system. We have compared our approach with two implementations of content-based routing. The first one is the traditional Siena routing algorithm based on the Poset structure [5]. The second one is a recently published improvement implementation based on the Forest data structure [25]. All three routing strategies were implemented in the *XSiena* system so as to create a uniform test environment operating within the same communications framework and with identical message semantics. All experiments have been executed on the same hardware and software: Java HotSpot(TM) Server VM (build

1.6.0_03-b05, mixed mode) running on a DELL Optiplex 745 with Core 2 Duo E6400 processor and 2GB of RAM. Regarding the workloads, we have assumed a similar approach as presented in [25], with a distinct exception that every subscription originates from a different subscriber/interface.

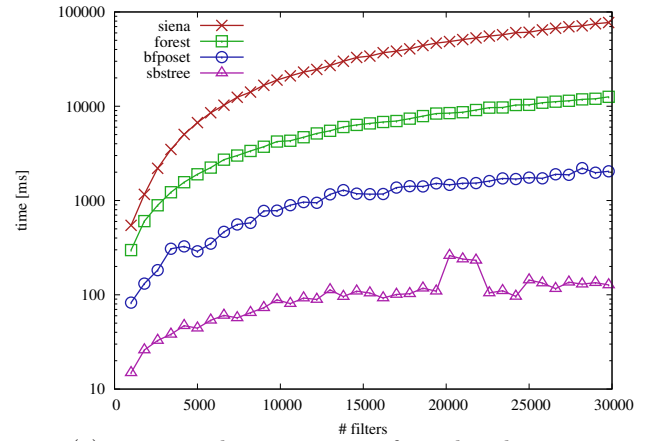
6.1 Routing

Figure 6 shows the time needed to insert a given number of subscriptions into different routing structures. The **siena** data series represents the original *SIENA* poset, the **forest** data series represents the improved implementation presented in [25] and **bfposet** with **sbstree** represent the Bloom filter based routing structures. In Section 4, we have shown that only after insertion into both structures a subscription can be matched against incoming events. We can observe that although the original *SIENA* data structure has the worst performance, the difference between the more optimal **forest** and **sbstree** combined with **bfposet** is still of an order of magnitude. Interestingly, also the tests with Pareto distribution, which mimics the social phenomena show a significant improvement over the **forest** and **siena** algorithms. This could be explained by the fact that additional time is lost in those structures when searching for less popular matches – in contrast to the uniform distribution with 1,000 unique attribute names. The bad performance of the *SIENA* poset is further stressed by the fact that its routing structure allows only for hierarchical networks of brokers, while both **forest** and Bloom filter based structures are designed to work with arbitrary acyclic graph topologies. The test with 1,000 attribute names used subscriptions having only one predicate, while the 20,000 attribute names used subscriptions with 2 to 5 attribute names (uniform distribution). The Pareto distribution used in all tests had the parameters $\alpha = 1$ and $\beta = 3$. Attribute names for both events and subscriptions are selected from the Automatically Generated Inflection Database (AGID)⁶, based on **aspell** word list, containing 112505 English words and acronyms. In all tests we have assumed a pessimistic scenario, i.e., that every subscription originates in a different subscriber.

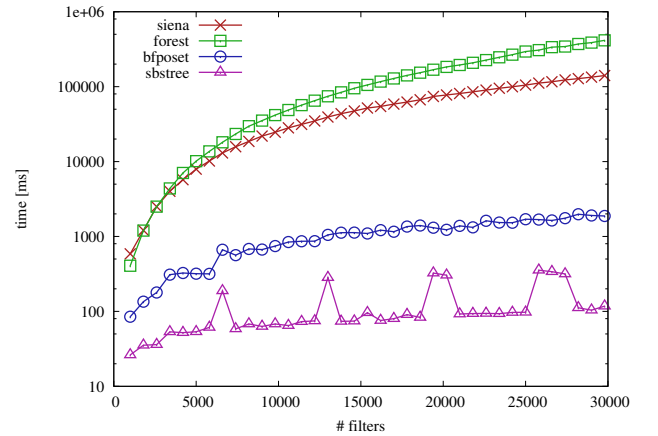
6.2 Matching and Forwarding

Figure 7 shows the event matching speed as a function of number of subscriptions stored in the routing structure of a broker. In case of traditional content-based routing, the matching time is defined as the time needed to calculate the forwards set for an event. In case of the **bfposet** it is a time needed by the broker to assign a Bloom filter to an event and in case of the **sbstree** it is a time needed to calculate the forwards set for an event given its sparse BitSet. Figure 7(a) shows results for a set of 1,000 unique attribute names, while Figures 7(b) and 7(c) show results for a set of 20,000 words using normal and Pareto distributions respectively.

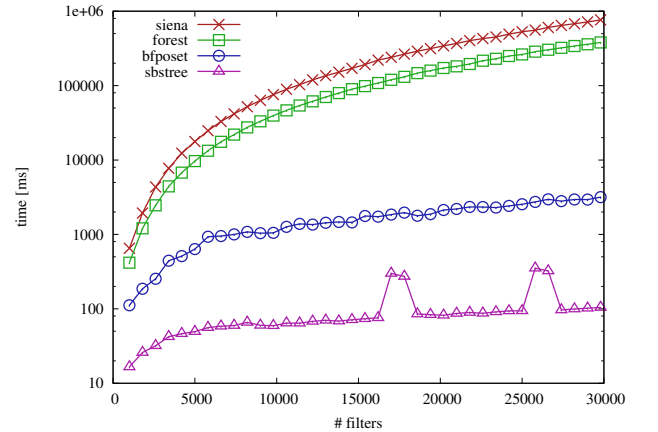
In all cases, the matching time using Bloom filter based structure (**bfposet**) is significantly faster than the matching time for traditional *SIENA* poset and **forest**. The difference in the performance improvement can be attributed to the way the **bfposet** is built: attribute names are stored in a hash set and serve as pointers to the lists of respective attribute constraints. Clearly, with increasing number



(a) 1,000 attribute names, uniform distribution



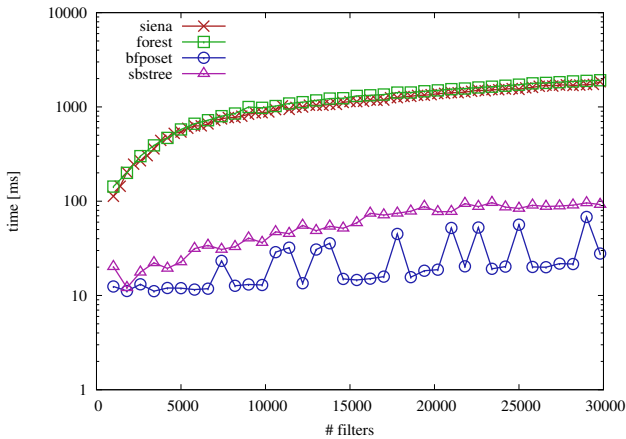
(b) 20,000 attribute names, uniform distribution



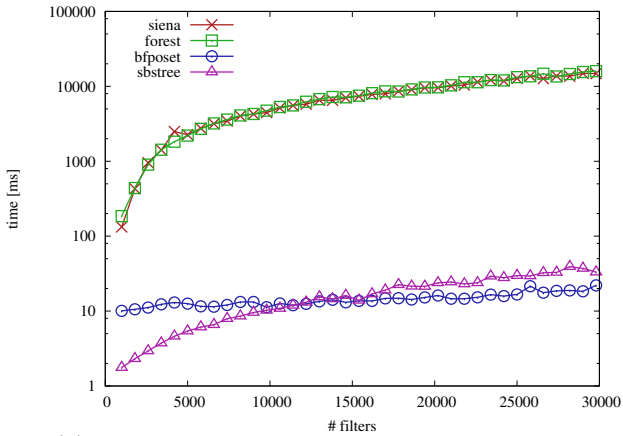
(c) 20,000 attribute names, Pareto distribution

Figure 6: Filter storage time as a function of number of filters to store.

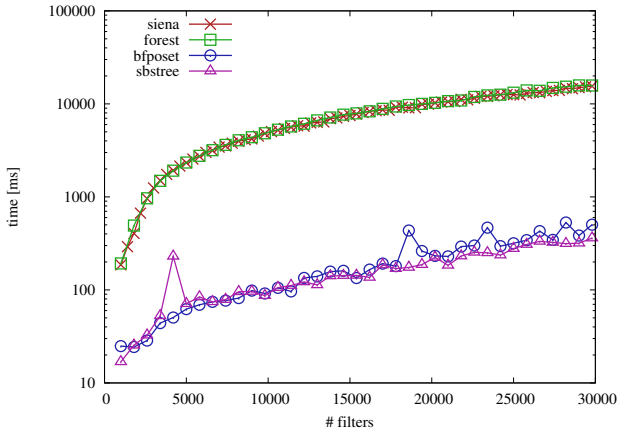
⁶<http://wordlist.sourceforge.net>



(a) 1,000 attribute names, uniform distribution



(b) 20,000 attribute names, uniform distribution



(c) 20,000 attribute names, Pareto distribution

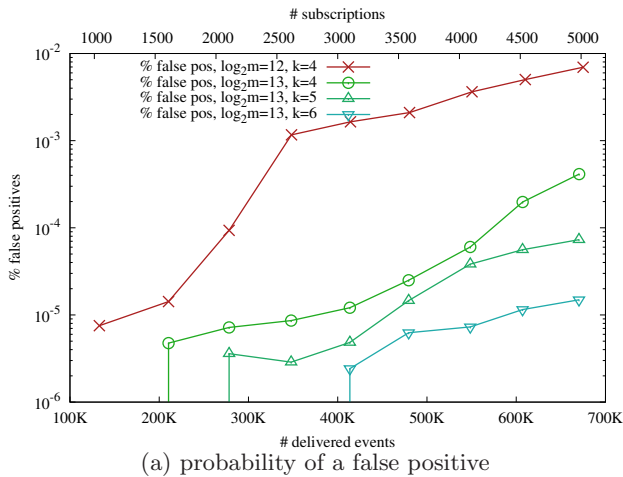
Figure 7: Matching time of 1000 events as a function of number of filters in the routing structure.

of different attribute names the advantage of the **bfposet** is more substantial as with single lookup in a hash set it can filter out more irrelevant predicates. We can observe that a twentyfold increase in the number of different attribute names, with constant number of subscriptions results in the approximate tenfold improvement in comparison between the **bfposet** and traditional routing structures. The same mechanism allows for the explanation of the performance of **bfposet** in the routing test – see Figure 6. We can also observe that the matching speed using the **sbstree** follows with the same performance. The higher matching time in case of the 1,000 attribute names test can be contributed to the higher cost of tree traversal, as opposed to the selection of matching elements on one level. The sporadic peaks in the measured matching time (and filter storage time) can be contributed to background tasks interfering with the experiment.

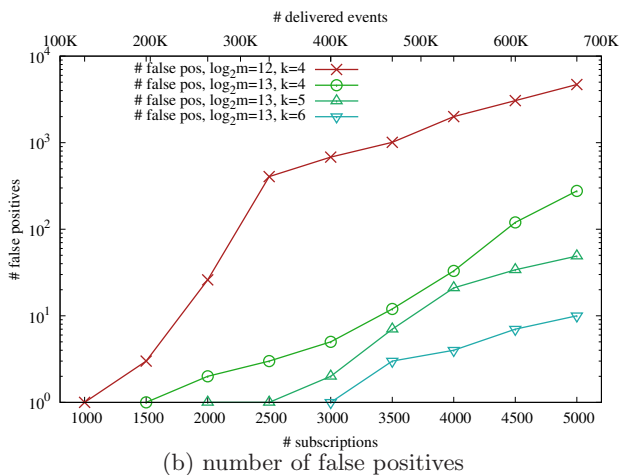
Figure 4 allows us to estimate the average overhead for the transmission of an event when using edge routing. We can observe that for the Pareto (worst case) attribute name distribution using 20,000 attribute names an average event will have up to 200 bits set in its sparse BitSet representation of the Bloom filter. Using short integer values (having the width of 16 bits) in the sparse BitSet this would result in the average overhead of 400 bytes per event. We believe this is an acceptable value in the case when edge routing should be used.

6.3 False Positives

The advantage of using Bloom filters to represent the sets of predicates matching an event is the fact that they are compact and can represent the whole universe of predicates – having all bits set. However, this feature does not come without a price. In case of the Bloom filters, this is the possibility of false positives. As illustrated in Figure 5 the probabilities for small filters are relatively high, even for a low number of elements. This in turn might have a negative impact on the memory requirements of the brokers. The false positives in a Bloom filter would result in false positives in the Pub/Sub system using Bloom filters for routing. In Pub/Sub systems we define a false positive as a delivery of an event which was not subscribed to. In order to verify that issue, we have performed a series of experiments where we have routed 10,000 events through a broker which stored an increasing number of subscriptions. After the delivery of events, we have checked whether they have been really subscribed for. For our experiment, we have assumed a pessimistic case where every subscription is issued by a different subscriber. In order to maximize the number of matches, in this experiment we have used subscriptions with one predicate selected using a Pareto distribution from 2,000 English words. Events were constructed in a similar way: every event contained from 1 to 10 attribute name and value pairs (uniform distribution), and attribute names were chosen from 2,000 English words using the Pareto distribution. In all experiments, according to the expectations, the number of false negatives was equal to zero. Figure 8(b) shows the number of false positives in the function of the number of subscriptions in the broker. At first we can observe that the number of false positives is lower than it would be assumed when looking at the Figure 5. For example the number of false positives events for 5,000 subscriptions, $\log_2 m = 12$



(a) probability of a false positive



(b) number of false positives

Figure 8: False positives as a function of size of the Bloom filter and the number of hash functions used

and $k = 4$ is equal to 4695, which is less than 50% of the sent events. Looking at Figure 5 we might expect to receive almost exclusively false positives. This contradiction might be explained by three facts: (1) probability of false positives is lower if we consider multiple predicates forming a subscription, as their conjunction results in a multiplication of false positives probabilities; (2) the high number of matches in the experiment results in false positives being covered by non-false positives, thus reducing their number; (3) subscriptions generated with Pareto distribution will tend to have more identical predicates, reducing the number of different elements. Figure 8(a) shows the percentage of false positives as a function of the number of delivered events. The points in the graph correspond to the points in the Figure 8(b) in that we can observe that for 1,000 subscriptions in the broker we have approximately 130,000 subscribers notified, which given the number of 10,000 events sent, results in the average 13 subscriptions matched per event.

7. SUMMARY

We have presented a Bloom filter based approach to routing, which is a fast, flexible and decoupled form of information dissemination in content-based publish/subscribe systems.

We have provided a prototype implementation using an $\mathcal{X}Siena$ system, which is based on $SIENA$. We have evaluated the performance of the implemented routing algorithms and compared them with other existing approaches to the content-based routing. Our experiments have shown, that the Bloom filter based routing promises good performance under various traffic characteristics, especially when compared to the competing solutions. Apart from performance related issues, we motivate the use of the Bloom filters by their ability to naturally express the conjunction of predicates of subscriptions, and by their limited size, which allows them effectively to represent any number of elements – subscription predicates matching an event. We have shown, that according to the expectations, the Bloom filter based routing does not expose any false negatives, and false positives rates are acceptable – even for relatively small Bloom filters. In this paper, we have also shown that a compact representation of a Bloom filter using a sparse bit set allows for substantial saving both in processing time at the brokers and in transmission bandwidth – which has been proved with various workloads.

Our approach can be used for both edge and typical content-based routing, allowing to trade off speed of event dissemination vs the subscription traffic overhead. We believe that this would allow system designers to tune the performance depending on the application requirements and environmental constraints.

8. REFERENCES

- [1] I. Aekaterinidis and P. Triantafyllou. Publish-subscribe information delivery with substring predicates. *IEEE Internet Computing*, 11(4):16–23, August 2007.
- [2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [3] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribenetworks. In *Middleware*, 2005.
- [4] Fengyun Cao and J.P. Singh. Efficient event routing in content-based publish-subscribe servicenetworks. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, 2004.
- [5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.
- [6] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *Proceedings of ACM SIGCOMM 2003*, pages 163–174, Karlsruhe, Germany, Aug 2003.
- [7] Raphaël Chand and Pascal Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In Jose C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference*, volume 3648 / 2005, Lisbon, Portugal, August 2005. Springer Berlin / Heidelberg.
- [8] Peter C. Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *5th International Conference on Formal Methods in Computer-Aided Design*, 2004.

- [9] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- [10] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000.
- [11] Eli Fidler, Hans-Arno Jacobsen, Guoli Li, and Serge Mankovski. The padres distributed publish/subscribe system. In *Feature Interactions in Telecommunications and Software Systems*, pages 12–30, Leicester, UK, July 2005.
- [12] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, New York, NY, USA, 1984. ACM Press.
- [13] T. S. Jayram, Subhash Khot, Ravi Kumar, and Yuval Rabani. Cell-probe lower bounds for the partial match problem. *Journal of Computer and System Sciences*, 69(3):435–447, November 2004.
- [14] Zbigniew Jerzak and Christof Fetzer. Prefix forwarding for publish/subscribe. In *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, pages 238–249, Toronto, Ontario, Canada, June 2007. ACM Press.
- [15] S. Kale, E. Hazan, F. Cao, and J.P. Singh. Analysis and algorithms for content-based event matching. In *25th IEEE International Conference on Distributed Computing Systems Workshops*, pages 363–369, June 2005.
- [16] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *14th Annual European Symposium on Algorithms*, page 456467, 2006.
- [17] Gero Mühl, Ludger Fiege, F. C. Gärtner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *MASCOTS '02: Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 167–176, Washington, DC, USA, 2002. IEEE Computer Society.
- [18] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., 2006.
- [19] Peter Pietzuch, David Eysers, Samuel Kounev, and Brian Shand. Towards a common api for publish/subscribe. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 152–157, New York, NY, USA, 2007. ACM.
- [20] Peter R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Computer Laboratory, Queens' College, University of Cambridge, February 2004.
- [21] Zhenhui Shen and S. Tirthapura. Faster event forwarding in a content-based publish-subscribe system through lookup reuse event. In *Fifth IEEE International Symposium on Network Computing and Applications*, pages 77–84, 2006.
- [22] Zhenhui Shen, Srikanta Tirthapura, and Srinivas Aluru. Indexing for subscription covering in publish-subscribe systems. In *Proc. IEEE International Conference on Data Engineering (ICDE)*, pages 32–43, 2005.
- [23] Salman Taherian and Jean Bacon. State-filters for enhanced filtering in sensor-based publish/subscribe systems. In *International Conference on Mobile Data Management*, pages 346–350, May 2007.
- [24] Sasu Tarkoma. Chained forests for fast subsumption matching. In *DEBS '07: Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 97–102, New York, NY, USA, 2007. ACM.
- [25] Sasu Tarkoma and Jaakko Kangasharju. Optimizing content-based routers: posets and forests. *Distributed Computing*, 19(1):62–77, September 2006.
- [26] P. Triantafillou and A. Economides. Subscription summaries for scalability and efficiency in publish/subscribe systems. In *Proceedings of Workshops of 22nd International Conference on Distributed Computing Systems*, 2002.
- [27] P. Triantafillou and A. Economides. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *Proceedings of 24th International Conference on Distributed Computing Systems*, 2004.
- [28] Christos Tryfonopoulos, Christian Zimmer, Gerhard Weikum, and Manolis Koubarakis. Architectural alternatives for information filtering in structured overlays. *IEEE Internet Computing*, 11:24–34, 2007.
- [29] Yi-Min Wang, Lili Qiu, Dimitris Achlioptas, Gautam Das, Paul Larson, and Helen J. Wang. Subscription partitioning and routing in content-based publish/subscribe systems. In *16th International Symposium on Distributed Computing (DISC'02)*, 2002.
- [30] E. Yoneki and J. Bacon. Distributed multicast grouping for publish/subscribe over mobile ad hoc networks. In *IEEE Wireless Communications and Networking Conference*, New Orleans, USA, March 2005.
- [31] Yuanyuan Zhao, Daniel Sturman, and Sumeer Bhola. Subscription propagation in highly-available publish/subscribe middleware. *Lecture Notes in Computer Science*, 3231:274–293, 2004.