

# Prefix Forwarding for 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

We present a prefix forwarding algorithm for content-based publish/subscribe systems. Our algorithm performs only one content-based match per message regardless of the number of routers (hops) traversed from the source to the destination. Moreover, prefix forwarding preserves the decoupling properties of publish/subscribe system. Prefix forwarding does not put any restriction on the content of the messages. The presented algorithm does not introduce any false negatives and allows to tune the false positive rate to balance the bandwidth and processing overheads. We provide experimental results confirming the properties of the proposed approach.

## 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

## 1. INTRODUCTION AND MOTIVATION

It has been recently shown [11] that content matching problem can be used to solve to the point dominance problem, which itself is a special case of range-searching problem, for which no worst case computationally efficient solution exist. This implies that content matching also does not have a worst case efficient solution. This has been also formally proved in [7], which states that the hardness of content matching is equivalent to the Partial Match [6] problem. Currently, in most content-based publish/subscribe systems content-based matching has to be performed at every node. Therefore, it is desired to limit the number of necessary

content-based matches to the minimum in order to improve the efficiency of publish/subscribe systems.

In this paper, we propose a prefix forwarding mechanism for publish/subscribe. Our mechanism is based on the observation that content-based matching needs to be performed only once, when a message enters the publish/subscribe network. The result of this match is a prefix assigned to a message. All subsequent nodes use this prefix to route the message further, which saves the effort of content-matching and simultaneously does not violate the decoupling principles of publish/subscribe systems [5]. Moreover, the proposed prefix structure allows to trade-off the precision of the content-based matching versus the number of false positives<sup>1</sup>. Specifically, the proposed solution does not introduce any false negatives<sup>2</sup> with respect to the original content-based event dissemination of the SIENA publish/subscribe system. Our mechanism does not impose any limitations neither onto the type nor onto the number of the messages used.

We present a proof of concept implementation based on the freely available SIENA [2] content-based, publish/subscribe system. We provide also simulation results for the prefix forwarding and its efficiency.

## 2. RELATED WORK

There exist two basic types of publish/subscribe systems: (1) filter-based systems and (2) multicast-based systems. Multicast-based systems partition the event space into multiple groups. Participants interested in the events of one group are formed into a multicast tree for the given group. Every event matching the group is then delivered to every participant in the group. Examples of such systems might be peer-to-peer systems [12], centralized [9] systems or topic-based [4] systems. Filter-based systems construct the event distribution trees on the fly by matching the events with the registered interest of the participants. Examples include hierarchical [8] and non-hierarchical [17] content-based systems.

All of the above mentioned approaches suffer from either lack of scalability, high false positives rates and lack of expressiveness (topic-based systems) or from high message pro-

<sup>1</sup>Messages delivered to subscribers, though not subscribed for.

<sup>2</sup>Messages not delivered to subscribers, though subscribed for.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DEBS '07, June 20-22, 2007 Toronto, Ontario, Canada  
Copyright 2007 ACM 978-1-59593-665-3/07/03... \$5.00

cessing overhead (content-based systems). Message processing overhead are caused by the need to perform a content-based matching for every message at every router [2] or it is an indirect result of the updates of the routing structures [3].

An attempt to cope with the above limitations has been proposed in [13]. Authors propose to use Bloom filters to aggregate, filter and match information in subscriptions, thus creating per broker subscription summaries compacting subscription information. However, the proposed approach requires a known, predefined and limited set of possible publication and subscription attributes, thus limiting the expressiveness of the content-based pub/sub.

Content-based publish/subscribe systems have no alternative as far as expressiveness, flexibility and precision are concerned. On the other hand, it has been recently stressed [11] that content-based matching is a technique for which no worst case computationally efficient solution exist. Therefore, recent efforts try to limit the number of content-based matches in pub/sub systems.

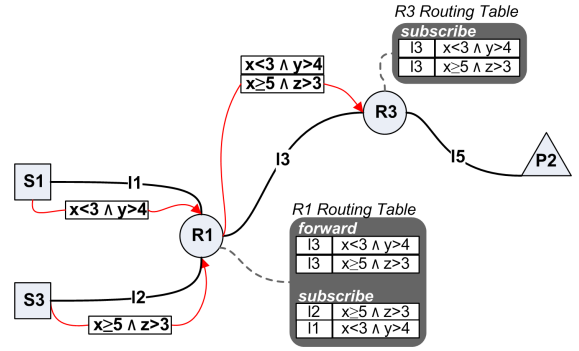
The authors of [1] propose to combine content-based and multicast-based approach. The presented solution suffers however from the fact that every server has to be aware of all other servers in the network. This limitation can be alleviated by the introduction of partitions. However, the implicit server cluster management and content space partitioning imposes new overheads. Specifically, the content space partitioning described in [15] requires a predefined and known set of attributes for every event, thus substantially limiting the expressiveness of the content-based system.

A similar approach to efficient routing has been presented in [16] where authors combine Bloom filter based subscription summaries and on-demand multicast routing protocol. Our approach differs in that we do not require a periodic broadcast of advertisements for route creation and we do not require the creation and maintenance of groups for a given publication.

Another approach has been proposed in [10], where the content-based matching has been combined with hash-based matching: matching results from the upstream routers are reused. The presented approach can however produce false negatives whenever it relies only on the hash-based matching. To ensure lack of false negatives, hash-based matching can only be used to quickly eliminate a subset of subscriptions to match with content-based matching having to be performed on the remaining ones.

### 3. BACKGROUND

The prefix forwarding semantics and proof-of-concept implementation are based on the SIENA publish/subscribe system. In the following we will use the terminology introduced with the SIENA system. In SIENA there are three types of participants (see Figure 1): (1) subscribers, shown as squares, (2) publishers, shown as triangles and (3) routers, interconnecting the subscribers and publishers, shown as circles. In order to receive information subscribers issue subscriptions (also called *filters*) which are broadcasted into the system (see Figure 1). Subsequently, publishers issue publications (also called *events*) which are actual pieces of infor-



**Figure 1: Routing of subscriptions.** Router  $R1$  forwards subscription  $x < 5 \wedge y > 3$  to both routers  $R2$  and  $R3$ . Each router stores the received and forwarded subscriptions table along with the corresponding links in its routing table.

mation. Subscriptions and publications are composed of attribute constraints. Attribute constraints are tuples consisting of an attribute name, an operator and an attribute value. Attribute constraints in filters are connected by conjunctions, while attribute constraints in events are connected by disjunctions. This implies that filter  $x > 5$  matches event  $x = 7 \vee y = 6$  and filter  $x > 5 \wedge y < 6$  does not match event  $x = 7$ .

Every router, upon receiving a publication, compares its content against stored subscriptions content (content-based matching) and whenever it encounters a match (e.g., subscription  $x < 5 \wedge y > 3$  and publication  $x = 1 \vee y = 5$ ) it forwards the publication on the reverse path of the subscription. This process is repeated until a subscriber is reached.

In order to cope with potentially large bandwidth overhead when disseminating filters SIENA introduces the concept of *coverage*. Filter  $f1$  covers filter  $f2$  if  $f1$  matches a superset of events matched by  $f2$ . An example might be  $f1 = x < 3 \wedge y > 4$  covered by  $f2 = x < 5 \wedge y > 3$ . Covered filters are not forwarded by the routers, thus saving bandwidth.

### 4. THE PREFIX FORWARDING

Proposed solution extends the existing implementation of the SIENA publish/subscribe system. In what follows we present the necessary modifications to the SIENA content-based publish/subscribe system that enable it to be used with the prefix forwarding. The first prerequisite for pre-

Input Filters	Resulting Normalized Filters
$x > 5 \wedge x > 2 \wedge y > 3$	$x > 5 \wedge y > 3$
$z = 9 \wedge z < 10 \wedge z > 2$	$z = 9$
$x = * \wedge x < 4$	$x < 4$
$x > 5 \wedge x < 3$	invalid filter

**Table 1: Normalization of filters for prefix forwarding.**

fix forwarding is the normalization of the existing SIENA filters. Filters in SIENA can be composed of arbitrary attribute constraints. This might result in possibly logically

inconsistent filters, e.g.,  $x > 3 \wedge x < 2$  or unnecessarily complex ones, e.g.,  $x > 3 \wedge x > 4 \wedge x < 6 \wedge x > 5$ . In order to normalize the filters, we evaluate the boolean expressions within filters and thus obtain a concise and coherent set of attribute constraints. The filter normalization is performed at the subscriber side by the publish/subscribe layer. Normalized filters are then propagated in the publish/subscribe network and are used for the construction of the Routing Tree.

#### 4.1 Construction of the Routing Tree

Subscriptions (Filters) in our publish/subscribe system are disseminated in a way which is similar to the SIENA approach. Filters are disseminated throughout the network to all routers. Filters are not disseminated further whenever they are covered in any of the routers – see Section 3. What is different with respect to the SIENA approach is the construction of the Routing Tree.

The Routing Tree (RT) holds complete routing information which is used to forward events and route filters. It is maintained by every router in the publish/subscribe network. This might seem contrary to the intuition behind the edge concept, as one might expect that only edge routers use this tree. However, we assume that every router  $r$  is an edge router with respect to the publishers which connect to the pub/sub network via  $r$ . The core task of the edge router is the maintenance of the RT. The maintenance task is limited to updates to the RT due to new filters arrivals and unsubscriptions.

The RT stores filters in a disjoint form – see Figure 2. Each node of a tree represents a single attribute constraint. Upon arrival of a new filter it is inserted into the RT on a per attribute constraint basis. Within a filter, attribute constraints are deterministically sorted in alphabetical order by the attribute names. In case of two identical attribute names, sorting considers the operators in those predicates. In case of two equal attribute names and operators predicates are sorted according to the attribute values<sup>3</sup>.

Insertion of a new filter starts at the root of the RT and progresses down the tree according to the algorithm in Listing 1. The first attribute constraint of the currently inserted filter is stored in the *Filter.first* data member. It is tested against the current node of the RT for the equality, coverage and intersection – Listing 1, lines 5–17. This reflects all possible relations two attribute constraints ( $ac1$  and  $ac2$ ) with the same names can be in: (1)  $ac1$  can cover  $ac2$ , (2)  $ac1$  can be covered by  $ac2$ , (3)  $ac1$  can be equal to  $ac2$  and (4)  $ac1$  can intersect  $ac2$ .

Attribute constraints intersection is a new concept introduced in this paper. Attribute constraint  $ac1$  intersects attribute constraint  $ac2$  if the  $ac1$  attribute name equals the  $ac2$  attribute name and the set of values selected by  $ac1$  ( $SAC1$ ) intersects the set of values selected by  $ac2$  ( $SAC2$ ):

$$SAC1 \cap SAC2 \neq \emptyset \quad (1)$$

Some examples might be:  $x < 2$  and  $x > 0$  pair – *inter-*

<sup>3</sup>Due to normalization such situation is only possible for the = operator.

**Listing 1: Updating RT with new Subscription.**

```

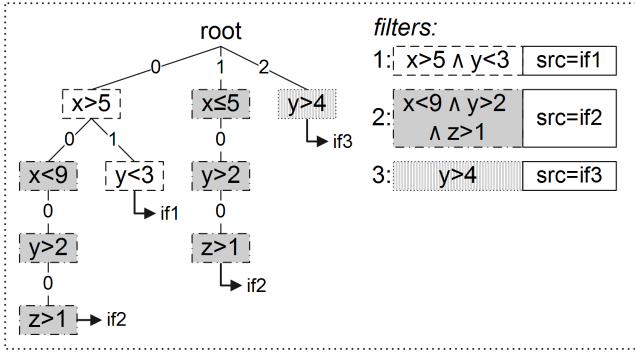
1 void insert(Filter f, Node n){
2   if(f.first==null)
3     break; //end of recursion
4
5   Node cr=null; // covered by f.first
6   Node cs=null; // covers f.first
7   Node in=null; // intersects f.first
8   Node eq=null; // equals f.first
9   for(child c : n.children){
10    if(intersects(f.first, c)) {
11      in = c; break;}
12    if(f.first == c) {
13      eq = c; break;}
14    if(covers(c, f.first)) {
15      cs = c; break;}
16    if(covers(f.first, c)) cr = c;
17  }
18
19  if(in){ //split and reinsert
20    Filter spl = f.split(in);
21    insert(spl, n);
22    insert(f, in);
23  }
24  else if(eq){ //step down
25    f.removeFirst();
26    insert(f, eq);
27  }
28  else if(cr){ //store as parent
29    n.removeChild(cr);
30    Node st = n.store(f.first);
31    st.addChild(cr);
32    f.removeFirst();
33    insert(f, st);
34  }
35  else if(cs){ //step down
36    insert(f, cs);
37  }
38  else { //store as child of n
39    Node st = n.store(f.first);
40    f.removeFirst();
41    insert(f, st);
42  }
43 }

```

*section* or  $x > 3$  and  $x > 4$  pair – *no intersection*. Intersection ensures that all attribute constraints with the same attribute name on the given level of the RT specify disjoint (non-overlapping) scopes in a one dimensional domain of all possible values. This property is used for event forwarding and further discussed in Section 4.2.

After the determination of the relation between the currently inserted attribute constraint and the set of children attribute constraints of the current node appropriate part of the recursive algorithm is executed. In case of the intersection (*intersects(f.first, c)*) algorithm performs a split of the inserted filter – Listing 1, lines 19-23. Split operation ensures that at the given level of the Routing Tree no two attribute constraints intersect.

One can visualize the filters as subspaces of a multidimensional space of all possible attribute constraints. An example is shown on the Figure 3. Two filters  $x < 5 \wedge y < 3$  and  $x > 1 \wedge y > 1$  select two subspaces in the two dimensional space of all possible values for  $x$  and  $y$ . If we were con-



**Figure 2: Filter insertion into the Routing Tree. Filters arrive in the order indicated by numbers 1:, 2: and 3:.**

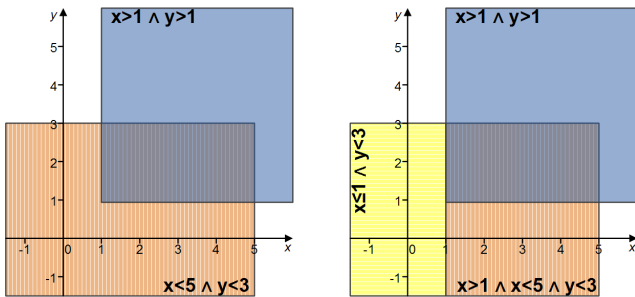
Considering three attribute names  $x$ ,  $y$  and  $z$ , the filter space would be three dimensional. Specifically, in content based publish/subscribe the dimensionality of the filter space (and thus implicitly event space) is not restricted.

Splitting filters in  $n$  dimensional attribute space is a challenging task, therefore in the Routing Tree the split is limited to the attribute name (dimension) of the current node, as every node holds only one attribute constraint. Figure 3 shows the split with regard to the  $x > 1$  attribute constraint of the  $x > 1 \wedge y > 1$  filter. This way we greatly reduce the splitting complexity and simultaneously do not put any restrictions on the number of different attribute names (dimensions) in the content-based publish/subscribe systems.

Input		Resulting Split	
$f$	$in$	$f$	$spl$
$x < 9 \wedge y > 2$	$x > 5$	$x < 9 \wedge y > 2$	$x \leq 5 \wedge y > 2$
$x > 1 \wedge z < 7$	$x \leq 6$	$x > 1 \wedge z < 7$	$x > 6 \wedge z < 7$

**Table 2: Filter  $f$  (Input columns) is being split with respect to predicate  $in$ . The results of the filter  $f$  split are two new filters  $f$  and  $spl$  presented in Resulting Split columns.**

Considering the algorithm presented in Listing 1 and Table 2 it can be observed that the  $split()$  method returns a whole new filter  $spl$ . It is presented as the second split product



**Figure 3: Example of the split operation of the filter  $x < 5 \wedge y < 3$  with respect to the  $x$  dimension of the filter  $x > 1 \wedge y > 1$ .**

in Table 2). Split operation creates thus a copy of the remaining filter  $f$  – equal to the first split product in Table 2. The  $spl$  filter is then subsequently inserted in parallel to the splitting node  $in$ , while  $f$  is inserted as its child. Intuitively, the split operation allows to make a deterministic decision as to which branch of the RT to follow when inserting new filter or matching a new event by not allowing for overlapping predicates to be placed at the same level of the RT. This property ensures also an optimal matching process of publications – see Section 4.2.

When inserting the last attribute constraint into the RT it is tagged with the source address (source interface) of the whole filter – see Figure 2. This allows to distinguish the last attribute constraint of the filter and is used in the forwarding process to obtain the reverse path which should be followed by the events. Please note that whenever there are two identical subscriptions arriving at the given node from two different sources the last attribute constraint for both of them will be represented by the same attribute constraint in the RT. This attribute constraint holds a list of two source addresses for both filters. These lists can grow arbitrarily large with the number of unique connections.

From the above, we can see that filter routing process is somewhat similar to the poset based routing of SIENA. This means that filters are routed further whenever insertion of the filter into the RT produces new root child. However, unlike SIENA's poset the RT allows for a fine grained interaction between filters on the attribute constraint level and ensures non-overlapping partitioning of the event space at each level of the RT.

## 4.2 Event Matching and Forwarding

When a new event enters our publish/subscribe system the first router it reaches matches it against the router's Routing Tree. The result of this match is a Forwarding Prefix Tree (FPT) which is subsequently assigned to (and piggybacked by) the event. Assignment of the FPT is executed only once in the edge (from the perspective of the event) router. All subsequent routers on the event's path use the assigned FPT to forward the event. Hence, the forwarding process consists of single content-based matching operation and multiple forward operations.

We can observe that attribute constraints placement in the Routing Tree reflects logical conjunctions and disjunctions between them. Traversing the Routing Tree from the root to its bottom along any of the paths equals to creating a filter consisting of the conjunction of all encountered attribute constraints. Adding another path to that filter results in a logical disjunction between the two paths. This observation is also the key for the event matching.

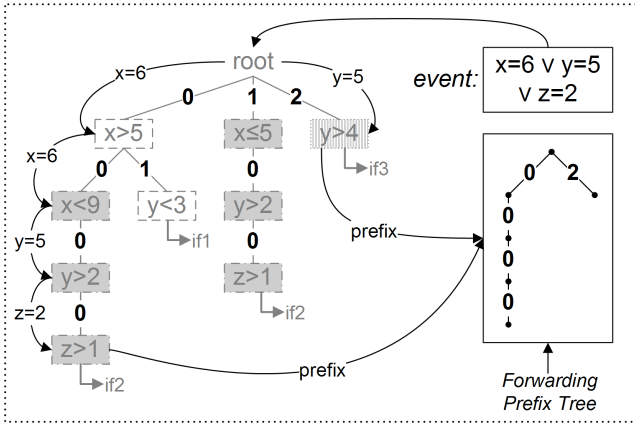
The event matching is carried out in the similar way to the filter insertion – see Listing 2. Attribute constraints in events are deterministically sorted by their names and values. This is performed at the publisher side by the publish/subscribe layer. Sorting of attribute constraints ensures that they are matched in the order of the filters' insertion into the RT. Upon arrival of the new event  $e$  a router matches it with the Routing Tree, starting with the first RT node  $n$ . The result of the matching is the Forwarding Prefix Tree consisting

**Listing 2: Event matching – creation of the FPT.**

```

1 match(Event e, Node n, FPTNode fN, int idx){
2     int i=0;
3     //for every child node of the RT
4     for(Node c : n.children) {
5         int myidx = idx;
6         //for every Attr Constr of the event
7         while(myidx < e.size()) {
8             if(covers(c, e.getAC(myidx))) {
9                 //step down the FPT and RT
10                FPTNode newFN = fN.add(i);
11                match(e, c, newFN, myidx);
12            }
13            ++myidx; //current event AC
14        }
15        ++i; //child number
16    }
17 }

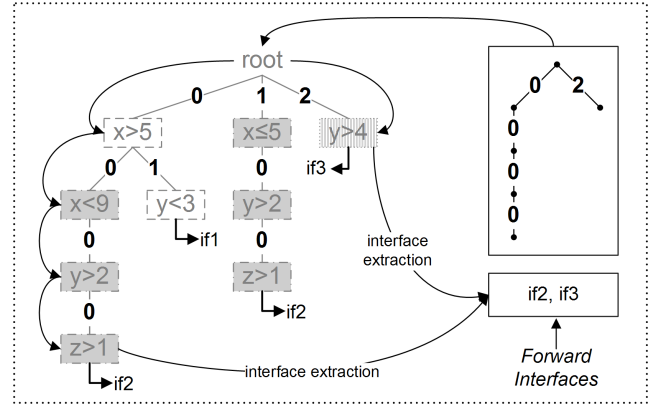
```



**Figure 4: The event matching process. An edge router creates the Forwarding Prefix Tree and assigns it to the event.**

of the Forwarding Prefix Tree nodes *FPTNode*. The matching starts by going through the child nodes (*n.children* – Listing 2, line 4) of the current RT node and comparing them against every attribute constraint (*e.getAC()* – Listing 2, line 8) in the event. If the event attribute constraint matches a filter attribute constraint (*covers()*) this path is added to the FPT (*fN.add()* – Listing 2, line 10) and the matching continues. Figure 4 visualizes the process of the event matching and the Forwarding Prefix Tree assignment. It can be seen that the FPT has actually a form which reflects the paths a given event traverses when matched against the router's Routing Tree. Arrows pointing from root towards the branches of the RT indicate the process of matching of event attribute constraints. Arrows with the *prefix* caption indicate the insertion of the new prefix into the FPT.

Another important observation is that the FPT preserves the relation between attribute constraints of the filters matching the event. Let us assume that event *e1* has been assigned a Forwarding Prefix Tree *fpt1* = 0010 and event *e2* has been assigned a FPT *fpt2* = 001. From the above we can reason that every subscriber interested in *e1* is also interested



**Figure 5: The event forwarding process. Router extracts the interfaces on which it will forward the event containing the given FPT.**

in *e2*. Moreover, in order to draw this conclusion, we need only to look at the FPT and do not need to consider the event's content. This property is further explored in the Section 4.3.

Once an event has been matched against the RT and assigned the FPT the same router executes the forward operation. During the forward operation FPT is used to traverse the RT – see Figure 5. Whenever during the traversal of the RT a node containing a list of source addresses (interfaces) is encountered (Listing 3, lines 4-9), those addresses are copied to the set of addresses an event should be forwarded to – forward interfaces (*fi*). When *fi* set equals all known addresses of the node neighbors (*oi*) or the last nodes of the FPT have been reached the forwarding process stops. Subsequently the event is disseminated to all neighbors in the *fi* set. The

**Listing 3: Event forwarding – creates and returns a set of the forward inrefaces (*fi*). The output interfaces set (*oi*) contains all known neighbors of the node.**

```

1 forward(CFPTNode fN, Node n, Set fi, Set oi) {
2     //for each child node of the current FPT node
3     for(CFPTNode currFN: fN.children()) {
4         //get corresponding RT node
5         Node currN = n.getChild(currFN.value());
6         if(currN.hasInIf()) {
7             //add the filter's forward interfaces to fi
8             fi.addAll(currN.getAllInIf());
9             if(fi.equals(oi)) {
10                //broadcast the event
11                break;
12            }
13        }
14        forward(currFN, currPN, fi, oi);
15    }
16 }

```

forwarding of the event in subsequent routers requires only the forwarding step to be executed, i.e., extracting the addresses of the next hops.

### 4.3 The Prefix Size

Prefix forwarding allows to disseminate events performing only one content-based match per event. It can be seen, however, that with the increasing number of different subscriptions stored in the Routing Tree the average Forwarding Prefix Tree will also grow in size. In order to limit the overhead, one can exploit the property mentioned in the Section 4.2. Instead of matching the event until the whole RT has been completely explored one can limit this process to a given level (height) of the tree.

This idea is based on the fact that the Prefix Tree is constructed exploiting the semantics of the SIENA filters. Specifically, whenever we consider an attribute constraint  $ac$  in the Prefix Tree  $pt$  we can say that all attribute constraints which are its descendants represent filters being a specialization of a filter constructed by traversing the path from the root of the  $pt$  to the  $ac$ .

As an example, let us again consider Figure 4. Instead of creating the complete FPT, we can abort the matching algorithm at level 1. This will result in the FPT having just two entries, apart from the root, – the 0 and the 2. The forwarding algorithm using such FPT can be outlined as: (1) perform the normal forwarding algorithm until the last node of the FPT has been reached, (2) if the corresponding PT node has any subtrees extract all source addresses from these subtrees and add them to the FI set, (3) forward the event to all routers in FI set.

The presented approach allows to trade off the size of the Forwarding Prefix Tree for the number of false positives. Intuitively, it is possible that some of the subtrees of the PT node corresponding to the last FPT node does not contain filters that match the event. This is the case for the event  $x = 6 \vee y = 5 \vee z = 2$  depicted in Figure 4. If we limit the depth of the FPT tree to one it will be also routed towards the subscriber of the  $x > 5 \wedge y < 3$  filter, although it does not match its filter. This will result in a false positive. Specifically, limiting the height of the FPT will never result in a false negative.

## 5. THE TREE OPTIMIZER

Correct forwarding of an event with attached FPT is only possible if the paths in the RT selected by the FPT preserve the attribute constraints conjunctions from the edge router which created the FPT and attached it to the event. Hence, all routers on the event path from the publisher to the subscriber must forward the event using its FPT and identical Routing Trees. In what follows, we formulate the set of conditions that must be met by the Routing Trees in order to achieve identical distribution of attribute constraints: (1) all Routing Trees must be composed of the same set of filters, (2) filters must be inserted into all Routing Trees in the same order and (3) filter insertion must be deterministic, i.e., inserting two identical filters  $f1$  and  $f2$  into two identical routing trees  $rt1$  and  $rt2$  must result in two new routing trees  $rt1'$  and  $rt2'$  which are also identical.

The last condition is already satisfied by the insertion algorithm presented in Listing 1. The second condition ensures that split operations in Listing 1 are executed in the same order. If filter  $f1$  is split with respect to  $f2$  the resulting RT might be different from the one created when  $f2$  is split w.r.t

$f1$ . This problem might be circumvented by performing splits with respect to some predefined value. Assume that a predefined value would be 0. If the RT already contains  $x < 6$  attribute constraint and a split should be performed due to arrival of the  $x > 3$  filter, then instead of splitting  $x > 3$  w.r.t  $x < 6$  a split of  $x < 6$  w.r.t  $x > 3$  should be performed as the  $> 3$  is closer the predefined 0 value than the  $< 6$ . This method might be however impractical as it involves a potentially large overhead resulting from the re-ordering of the RT caused by the child nodes of the  $x < 6$  node.

The first condition is almost never satisfied in a dynamic loosely coupled publish/subscribe system. Let us consider two routers  $r1$  and  $r2$  with two Routing Trees  $rt1$  and  $rt2$ . If an event  $e1$  has entered the publish/subscribe network via the  $r1$  router,  $r1$  will assign a Forwarding Prefix Tree  $fpt1$  to  $e1$ . This FPT has been created by matching the event against the  $rt1$ . When  $e1$  is subsequently forwarded to the  $r2$ , router  $r2$  uses  $fpt1$  to traverse its Routing Tree  $rt2$ . If  $rt1$  and  $rt2$  are different prefixes in the  $fpt1$  used with  $rt2$  might map to the different conjunctions of attribute constraints. Specifically, some of the prefixes might not be matched at all, since  $rt2$  might be missing nodes present in the  $rt1$ . Such problems might occur whenever two subscribers have a different set of subscriptions due to network delays, packet drops or message processing delays.

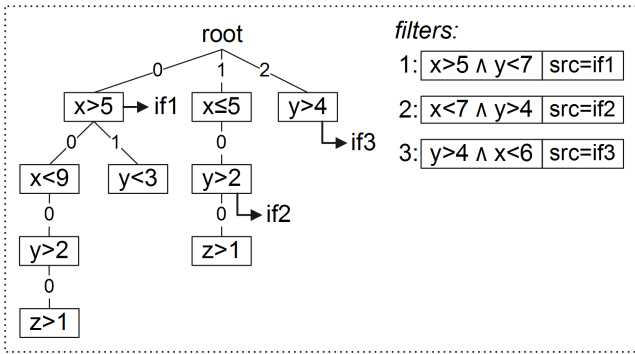
Therefore, we introduce the notion of the Tree Optimizer. Tree Optimizer (TO) is a node of the publish/subscribe system to which the task of the construction of the Routing Tree is delegated. TO takes over the routers responsibility to create and manage the Routing Trees and thus delivers the necessary determinism for the RT construction. It is important to notice that the Tree Optimizer itself obeys and uses the semantics of the content-based publish/subscribe system.

### 5.1 Construction of the Routing Tree with the Tree Optimizer

The Tree Optimizer creates the Routing Trees by first subscribing to subscriptions of the subscribers. This way events are routed towards it with the decoupled and asynchronous semantics of Publish/Subscribe maintained. Another important consideration is that the Tree Optimizer can be easily distributed in that each distributed Tree Optimizer subscribes to a part of the Routing Tree for which it is responsible. However, for simplicity let us now consider one Tree Optimizer which subscribes to all subscriptions.

The Tree Optimizer is the only entity in the publish/subscribe network which is allowed to change (add and remove) nodes in the Routing Tree. Hence, we ensure the necessary determinism for the operations regarding the Routing Tree. This in turn requires the introduction of the following changes to the Routing Tree management (presented in Section 4.1): (1) routers, upon reception of a new filter forward it also to the Tree Optimizer<sup>4</sup> while filters are in-

<sup>4</sup>Please note that typical broadcasting of subscriptions is not necessary any more. However, the current design allows both techniques to co-exist as the additional dissemination adds to the fault tolerance of the whole design.



**Figure 6: The filter insertion using `insertR()` algorithm. The existing Routing Tree is not modified. Arrows indicate the filters source addresses.**

serted into the Routing Tree using new `insertR()` function and (2) routers receive Routing Tree updates from the Tree Optimizer and map their own view of the network on the received RT.

In a typical scenario with delegated management of the Routing Tree one might expect routers to send new filters to the Tree Optimizer and wait for the Routing Tree updates from the TO containing new filters incorporated into the new RT. The main problem with such approach is the fact that new events which might be matched by the new filters will not be forwarded to subscribers until the new Routing Tree is disseminated by the Tree Optimizer. We are able to cope with that problem in that we again make use of the Routing Tree properties described in the Section 4.3. Instead of waiting for the updates to arrive we insert new filters using the modified `insert()` algorithm from the Listing 1. The new `insertR()` algorithm, shown in Figure 6, performs the same actions as the `insert()` algorithm, except for the creation of new nodes. When a new node needs to be created the `insertR()` algorithm stops and stores the return address of the filter (source interface) in the current RT node.

The so inserted return addresses allow to match a superset of events matched by the inserted filter. Moreover, we are able to route all events independently of the frequency of updates from the Tree Optimizer, the only trade-off being false positives. Whenever a Routing Tree update from the Tree Optimizer arrives the number of false positives is automatically reduced as the RT becomes more precise again.

In order to maintain the space decoupling in publish/subscribe the Tree Optimizer is responsible only for handling the contents of the filters in the Routing Tree, specifically it does not consider the return addresses of filters it receives. This leads to an observation that the Routing Tree management is decoupled between the Tree Optimizer and routers, with Tree Optimizer managing the filters in the Routing Tree and routers managing the return addresses. The Tree Optimizer creates the Routing Tree without inclusion of the filter return addresses, inserting only attribute constraints. When the difference between the current Routing Tree maintained in the Tree Optimizer and the Routing Tree maintained by the routers exceeds certain threshold an update

to the RT is disseminated to the routers. In order to calculate the difference the Tree Optimizer uses a copy of the previous Routing Tree. Currently the difference  $wdiff$  between two Routing Trees  $rt1$  and  $rt2$  is calculated with a following weighted algorithm:

$$wdiff_{rt1,rt2} = \frac{d(\text{root}_{rt1}, \text{root}_{rt2})}{s(\text{root}_{rt1}) + s(\text{root}_{rt2})} \quad (2)$$

where:

$$s(i) = \forall_{c \in C(i)} \sum \frac{1}{\text{level}(c)} + s(c) \quad (3)$$

$$d(i, j) = \forall_{k \in C(i), l \in C(j)} \begin{cases} k = l : & d(k, l) \\ k \neq l : & s(k) + s(l) \\ k = null : & s(l) \\ l = null : & s(k) \end{cases} \quad (4)$$

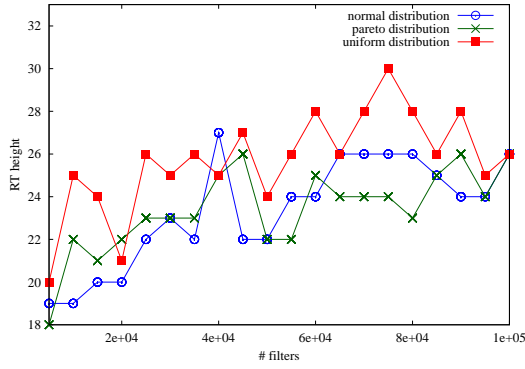
and  $C(i)$  is the set of all children of the RT node  $i$ . The weighted algorithm emphasizes changes closer to the root as those will have more significant impact on the potential false positives. Routers upon reception of the new RT updates insert outstanding filters (since last update) using the `insertR()` algorithm. This way they create a local mapping of the filter return addresses onto the Routing Tree preserving the decoupling principles.

Another issue arises when we consider that it is possible for some events to be still underway in the publish/subscribe network when a Routing Tree update is processed by routers. This might result in a situation when the FPT of the event was created with the Routing Tree  $rt1$ , however the subsequent forwarding would have to be performed with the Routing Tree  $rt1'$  being different (due to TO updates) from  $rt1$ . We solve this problem by implementing a monotonically increasing virtual clock in the TO. This virtual clock is incremented with every publication of the RT update. Upon update publication TO attaches the clock's value on the update so that it can be saved along the new nodes by every Router receiving an update. This allows for a construction of a Routing Tree whose nodes are tagged with the virtual creation time. An event being matched against a Routing Tree is assigned a Forwarding Prefix Tree along with the corresponding virtual clock time of the youngest (highest virtual clock time stamp) RT node FPT stores a reference to. Subsequent event forward operations consider only the Routing Trees which are composed of the nodes which are tagged with virtual clock time stamps time less or equal to that of the FPT.

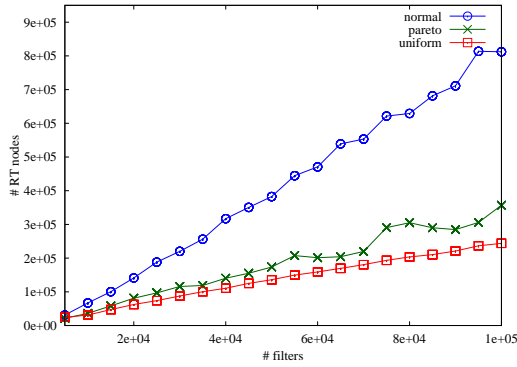
## 6. RESULTS

Our implementation is based on the SIENA publish/subscribe system. We have used existing SIENA filters and events and extended them to implement the functionality necessary for the prefix forwarding. Our implementation of prefix forwarding and filter routing has completely replaced the old `poset`-based SIENA source code. All tests were performed in a dedicated discreet event simulator, conceptually and architecturally based on the OMNeT++ [14] simulator by András Varga. For the random number generators we have used parts of the Stochastic Simulation in Java (SSJ)<sup>5</sup> library developed by Pierre L'écuyer.

<sup>5</sup><http://www.iro.umontreal.ca/~simardr/ssj/>



(a) Routing Tree height

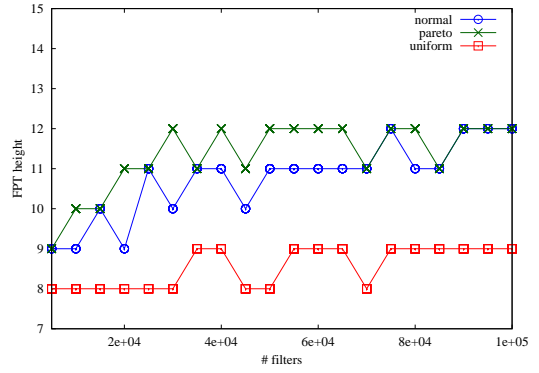


(b) Routing Tree size

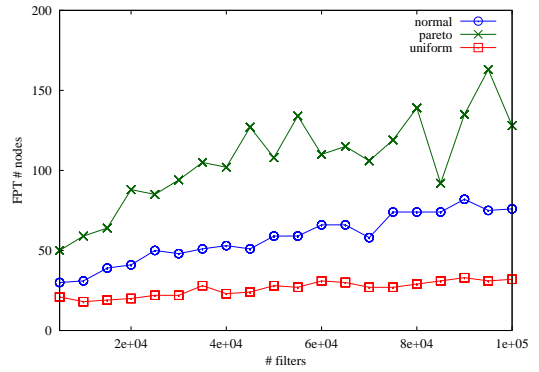
**Figure 7: Number of nodes (RT size) and height (RT height) of the Routing Tree as a function of the number of stored filters (# filters)**

The first tests included the feasibility of prefixes for message forwarding. Specifically, we were interested whether the size of the Forwarding Prefix Tree will allow it to be used with the events. For that purpose, we have created Routing Trees with increasingly large number of filters stored in them and computed the Forwarding Prefix Trees for random events matched against those trees. The results are presented in the Figures 7 and 8. Both figures show the development of the size of the Routing Trees and the Forwarding Prefix Trees as a function of number of filters stored in the Routing Trees. We have used three different filters and event generators. The differentiating factor was the distribution of the attribute names and attribute values for the constraints used in filters and events. We have decided to use two common distributions: Pareto and Gaussian. We have also added the uniform distribution for comparison. We have varied the amount of constraints in the filters from 2 to 8 and the number of constraints in the events from 2 to 12, both using normal distribution. The attribute values were selected from a continuous interval  $[-50, +50]$ .

We can observe on both Figures that although the count of attribute constraints in the Routing Tree approaches one million, the number of nodes (size) of the largest Forwarding Prefix Tree barely exceeds 150 nodes. This confirms that



(a) Forwarding Prefix Tree height



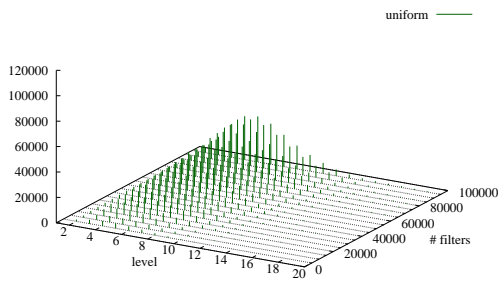
(b) Forwarding Prefix Tree size

**Figure 8: Number of nodes (FPT # nodes) and height (FPT height) of the Forwarding Prefix Tree as a function of the number of filters (# filters) stored in the Routing Tree. Average values for 100 events matched against the RT from Figure 7.**

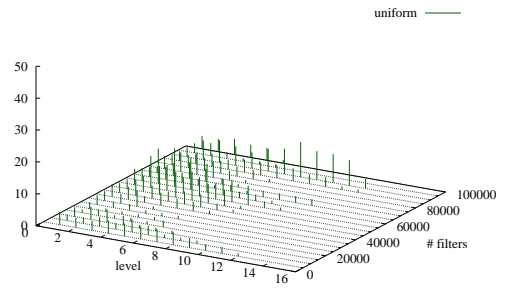
the size of the FPT exhibits small payload and allows it to be piggy-backed on the events. The count of the attribute constraints is higher than it would seem from the number of filters due to the fact that the split operation duplicates parts of filters and thus, for highly overlapping distributions (Gaussian and Pareto), the number of RT nodes is higher than for the uniform distribution. On the other hand identical attribute constrains in the filters contribute to the decrease in the overall size of the Routing Tree.

We have also plotted the distribution of the number of nodes per tree level (on the vertical axis) for both Routing Tree (Figure 9) and Forwarding Prefix Tree (Figure 10). We can see that Pareto and Gaussian distributions create wider Forwarding Prefix Trees which might be explained by the fact that Routing Trees are exposing similar characteristics due to the split operation of the intersecting filters.

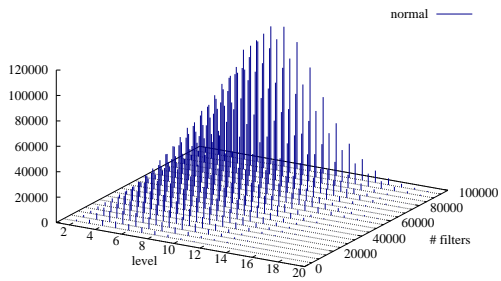
Finally, we have simulated the normal operations of a normal publish/subscribe system. We have created a network with 9 subscribers, 9 publishers and 27 routers. Each of the subscribers subscribed every 3 seconds for a random event with a filter generated using the Pareto distribution. Each



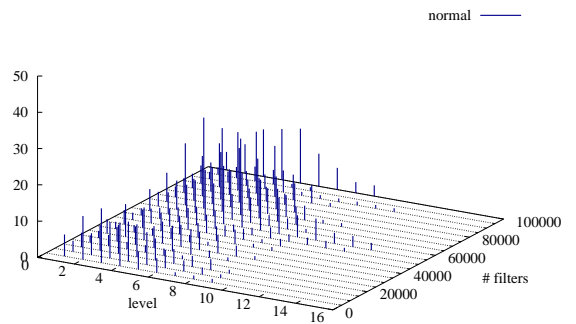
(a) RT nodes per level – uniform distribution



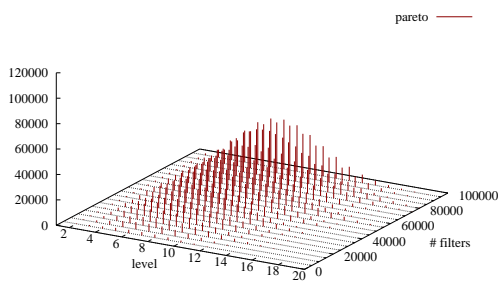
(a) FPT nodes per level – uniform distribution



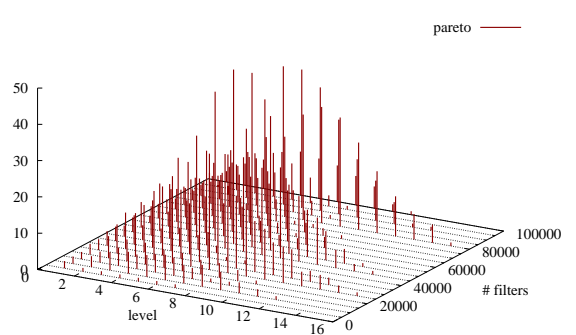
(b) RT nodes per level – normal distribution



(b) FPT nodes per level – normal distribution



(c) RT nodes per level – pareto distribution



(c) FPT nodes per level – pareto distribution

**Figure 9:** Distribution of nodes per level (vertical axis) for the Routing Tree as a function of the number of stored filters.

**Figure 10:** Distribution of nodes per level (vertical axis) for the Routing Tree as a function of the number of stored filters.

of the publishers issued an event every 3 seconds. Figure 12 shows the false positives rate as perceived by the subscriber. We can clearly see that with the growth of the routing trees the rate of false positives falls achieving less than 1% after 15000 events publications. We can also see that the results are similar for varying threshold values at the Tree Optimizer. We can recall that threshold value determines the difference between the current Routing Tree and the most recently disseminated Routing Tree which triggered the new Routing Tree dissemination.

We have also counted the number and frequency of Routing Tree updates performed by the Tree Optimizer. From the Figure 11 we can see that the number of updates is relatively low (84 for the threshold of 30%) and as the Routing Trees grow, the number of necessary updates decreases. This is due to the fact that the larger the Routing Tree grows the more new filters it takes to reach the update threshold. One way to avoid too infrequent updates for large Routing Trees is to add a second, time-based threshold.

Figure 13 shows the false positives count for the three distributions we have used in our experiments. The best results are obtained for the Gaussian and Pareto distributions. The normal distribution is almost an order of magnitude worse. This is caused by a higher probability for a filter to be placed closer to the root in the Routing Tree, and thus for the `insertR()` function to add the return address of a filter to the root node. However, we assume that the normal distribution is unlikely in an environment accepting large scale human-based input.

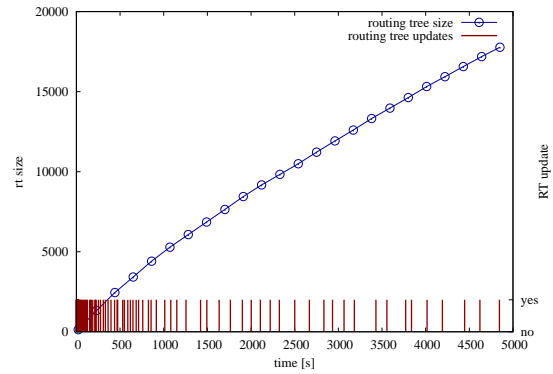
## 6.1 Testing Based on AOL/Google Data

Testing publish/subscribe systems has always been a challenging task, mainly due to the lack of real life data. There have been a few examples of Pub/Sub systems being implemented in a real life environment (e.g., Gryphon has been used in US Tennis Open, Ryder Cup, and Australian Open<sup>6</sup>), however, the community has never had the opportunity to evaluate the data gathered and processed by those systems. Therefore, authors (including authors of this paper) have usually tested their systems by creating their own filter and event generators and feeding the system with the so generated data using various distributions.

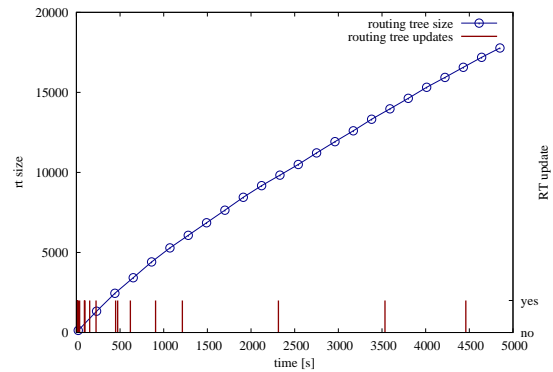
In order to cope with this problem, we have decided to use alternate source of data, namely the search engine logs. Queries issued by users of a search engine bear a certain similarity to subscriptions. A query contains data which summarize the interest of a given user, thus it can be regarded as a subscription. Such *subscriptions* are matched by the search engine against its database of all known publications (web pages) and results (links to content) are delivered to a user as a web page.

Therefore, being given a log containing user queries issued to a search engine one can easily build a set of subscriptions. In our experiments we have used the data released by the AOL on August 4, 2006 containing 20,000,000 search keywords for over 650,000 users over a 3-month period between March 1, 2006 and May 31, 2006. We have extracted two sets of data:

<sup>6</sup><http://www.research.ibm.com/distributedmessaging/>



(a) Threshold set at 30% – 84 updates



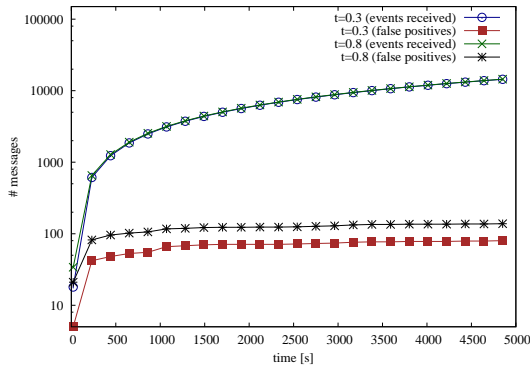
(b) Threshold set at 80% – 19 updates

**Figure 11: Number of nodes (rt size) in the Routing Tree maintained by the Tree Optimizer and the Routing Tree updates (RT update).**

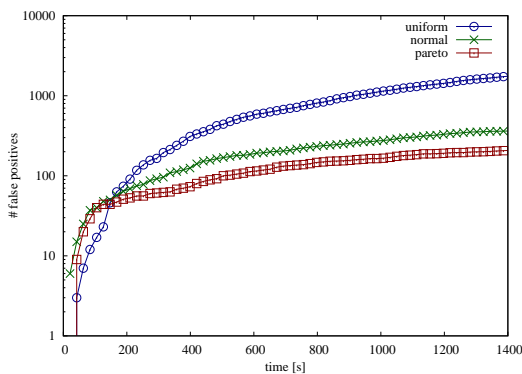
- (1) SET A: 43238 queries from 28 users with more than 4000 queries issued per user and (2) SET B: 77301 queries from 11276 users with exactly 10 queries issued per user.

When creating subscriptions we have used only one predicate with attribute name set to *query*, operator set to *substring* and attribute value set to the content of subscription. Being certainly not perfect, this approach allowed us for an easy integration of queries into SIENA filter and event model. In what follows, we will outline the main issues we have encountered and our rationale for handling them. At this point, however, we would like to stress that our testing routine is still under development.

The main problem we were faced with, was the generation of publications. Being given a set of subscriptions, there is no direct way to devise a publication pattern, nor the content of publications. The main issue is the specification of the publication domain. However, when one considers the origin of data (search engine logs) it becomes clear that it is not necessary to obtain the whole publication domain – which would be equal to the mirroring of the Internet. It is enough to obtain the size of publication domain. In our case, the size of the domain translates directly to the index size of the AOL search engine, i.e., Google. We have assumed



**Figure 12:** False positives count as perceived by the subscribers as a function of the number of events received (# messages) for different Tree Optimizer threshold ( $t$ ).



**Figure 13:** False positives count as perceived by the subscribers as a function of the number of events received (# messages) for different filter and event generator distributions.

that AOL (Google) index equals that of Yahoo!, which is approximately  $2^{10}$ .

In next step, we have re-executed every query in the AOL log in order to obtain the number of hits each of them has got – a value not supplied with the original logs. From that and the size of the publication domain we can devise a probability that a given subscription will be matched, by dividing the number of results for this subscription by the index size of the search engine – under the assumption that a search engine has crawled the whole Internet. This in turn allows us to create corresponding publications with appropriate content by adding content to match subscriptions with a calculated probability. The results of our experiments are presented in Figures 14 and 15.

Figure 14(a) shows the Routing Tree after insertion of unique filters from the SET A, while figure 14(b) shows the Routing Tree after insertion of unique filters from the SET B. The lighter the color of the edge, the closer it is placed to root – edges from root to level 1 nodes are bright yellow. We can see that different users tend to have more contention in

their queries (substring operator), compared to more differentiated queries coming from the smaller user base.

Figure 15 shows the sizes of the FPTs obtained by matching publications with RTs obtained from SET A and SET B. The FPT size constitutes approximately 1% of the number of filters stored in the RT.

## 7. SUMMARY AND FUTURE WORK

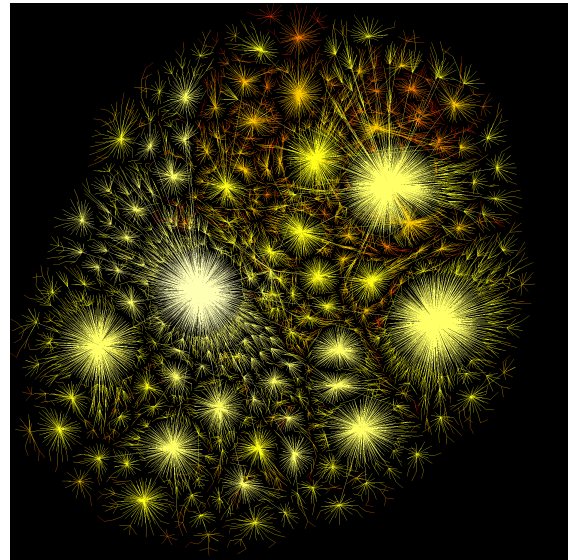
We have presented a prefix forwarding publish/subscribe system. Our system requires only one content match per delivered event. Moreover, our design is fully compatible with the loosely coupled nature of publish/subscribe systems and does not violate the decoupling properties of the SIENA publish/subscribe system it is based on. We have implemented a proof of concept system and evaluated its functioning in a simulation environment. We have shown that used prefixes exhibit low overhead and are reasonably sized with respect to the number of filters in the system. We have also shown that maintenance of Routing Trees by Tree Optimizers introduces little overhead as far as excess network traffic is concerned. The presented design exhibits good convergence as with respect to false positives count. Moreover, our system does not introduce any false negatives over those resulting from the normal operations of the traditional content-based publish/subscribe system.

The current outlook includes implementation and evaluation of the Prefix Forwarding in the PlanetLab and Emu-Lab environments. We also plan to further investigate the possibilities for trade-offs exhibited by our design. One of the interesting questions that needs further investigation is the influence of the length of the Forwarding Prefix Tree on the number of false positives and its interaction with the false positives introduced by the Tree Optimizer based management of the Routing Tree. Another issue worth investigating is the question whether by optimizing the layout of the Routing Tree itself one could gain benefits in terms of the false positive rate.

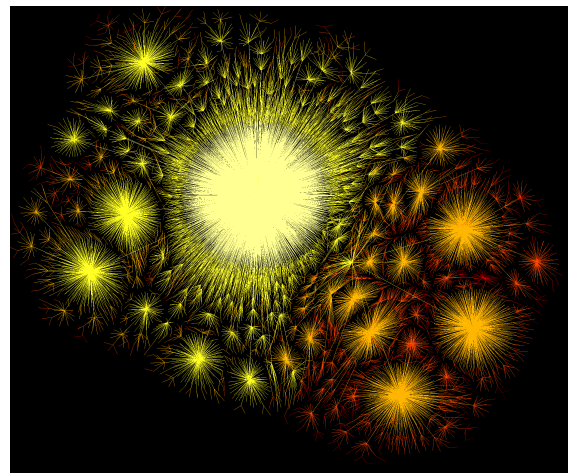
## 8. REFERENCES

- [1] Fengyun Cao and Jaswinder Pal Singh. Medym: Match-early with dynamic multicast for content-based publish-subscribe networks. In *Middleware*, 2005.
- [2] 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.
- [3] 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.
- [4] M. Castro, P. Druschel, A.M. Kermarrec, and AIT Rowstron. Scribe: a large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, October 2002.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] Robbert Van Renesse, Kenneth P. Birman, and Werner Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206, 2003.
- [9] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. *Proceedings AUUG2K, Canberra, Australia, June, 2000*.
- [10] Zhenhui Shen and S. Tirthapura. Faster event forwarding in a content-based publish-subscribe system through lookup reuseevent. In *Fifth IEEE International Symposium on Network Computing and Applications*, pages 77– 84, 2006.
- [11] 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.
- [12] W.W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A.P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8, 2003.
- [13] 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.
- [14] Andrs Varga. Using the omnet++ discrete event simulation system in education. *IEEE Transactions on Education*, 42(4):372, 1999.
- [15] 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.
- [16] 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.
- [17] Yuanyuan Zhao, Daniel Sturman, and Sumeer Bhola. Subscription propagation in highly-available publish/subscribe middleware. *Lecture Notes in Computer Science*, 3231:274 – 293, 2004.

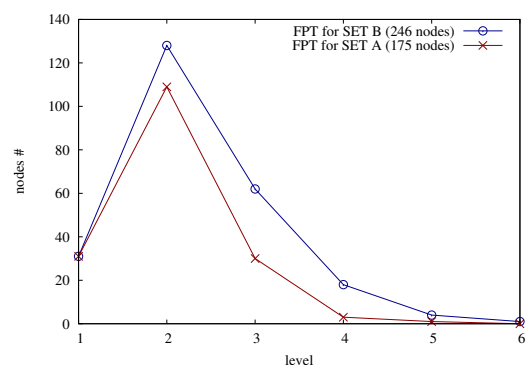


(a) RT after insertion of 25980 unique filters from SET B using the sub-string relation



(b) RT after insertion of 26124 unique filters from SET A using the sub-string relation

**Figure 14: LGL Plots of the Routing Trees after insertion of subscriptions created from the AOL log**



**Figure 15: Size of the FPT averaged over 1000 publications**