A Fast Content-based Data Distribution Infrastructure

Samrat Ganguly, Sudeept Bhatnagar, Akhilesh Saxena, Suman Banerjee, Rauf Izmailov

***Abstract*— We present Sieve – an infrastructure for fast content-based data distribution to interested users. The ability of Sieve to filter and forward high-bandwidth data streams stems from its distributed pipelined architecture. The complex message filtering task is broken-up into a sequence of light-weight filtering components resulting in high end-to-end throughput. Furthermore, since each component is assigned to a node based on its resource constraints, the queue buildup inside the nodes is minimal resulting in low end-to-end latency. Our experimental results based on real system implementation show that Sieve can sustain a throughput of more than 5000 messages per second for 100000 subscriptions with predicates of 10 attributes.**

***Index Terms*— Content-based Information Dissemination, Publish-Subscribe System, Event Stream Filtering**

I. Introduction

Content-based networking is an emerging data routing paradigm where a message is forwarded based on its content rather than specific destination addresses attached to it [1]. In this paradigm, data distribution to the users is based on the publish-subscribe model where publishers (sources) publish messages and subscribers (receivers) register their interest about the content. The content of the message has a list of attribute name and value pairs, such as (symbol="google"; price=196.8). The subscriber interest is usually expressed as a selection predicate, such as (symbol="google" & price $>$ 200 & volume $>$ 11M). A content-based network infrastructure enables selective data distribution from publishers to subscribers by matching the appropriate selection predicates.

However, along with the rich functionalities provided by content-based network infrastructure comes the high complexity of message processing stemming from parsing each message and matching it against all subscriptions. The resulting message processing latency makes it difficult to support high message publishing rates from diverse sites targeted to a large number of subscribers. For example, NASDAQ real-time data feeds alone include up to 6000 messages per second in the pre-market hours [2]; hundreds of thousands of users may subscribe to these data feeds.

The main contribution of this paper is design and evaluation of Sieve, a fast content-based data dissemination infrastructure to support high streaming rate of messages. The Sieve infrastructure is an overlay network with nodes spread over the Internet. The main design philosophy is to efficiently partition the complex multi-attribute subscription matching task and distribute the sub-tasks strategically among multiple servers while respecting the resource constraints at each server.

Sieve recognizes the importance of processing and bandwidth limitations of nodes as the core problems for any content-based filtering network. These limitations restrict the amount of filtering and forwarding that any node can perform. The specific objective of Sieve is to achieve a message processing latency that can match the message arrival rate while supporting rich content-based semantics. Sieve strives to partition the tasks so as to match both the network and the processing loads assigned to an overlay node with its corresponding processing capacity and network bandwidth. Fundamentally, the increased throughput of Sieve comes from the basic principles of pipelining, where the end-to-end throughput of the system increases by judicious partitioning of a task into smaller sub-tasks.

The message processing speed in a content-based router depends upon a variety of factors such as the subscription lookup data structure, predicate matching algorithm, and total space requirement. In existing distributed systems such as SIENA [1], [3], Gryphon [4] and others [5], each content-based router hosts a multi-attribute data structure to completely match the complex predicate for each subscription. The complete predicate matching cost coupled with large space requirement to hold the data structure in memory or processor cache increases the message processing latency.

Sieve differs from these systems by creating a distributed subscription-matching data structure over multiple overlay nodes. Using this data structure, Sieve provides the following advantages to achieve fast message processing: a) it allows for control over the space and forwarding bandwidth requirement, b) it allows for each node to participate in partial matching of predicates in a distributed way towards a global matching of the complete subscription predicate, and c) it allows for staging the subscription matching process such that the processing complexity at each node can match the message arrival rate.

Sieve architecture is composed of three stages of message forwarding to enable the filtering pipeline. Each stage is defined by a set of participating nodes and their well-defined roles. These stages create an efficient system that maximizes filtering pipelining while meeting the state space constraints and bandwidth limitations. This results in the following solutions that we propose in this paper:

- A consistent distributed data structure for data filtering and forwarding meeting the space requirements of a participating node.
- An efficient attribute-specific tree construction for mes-

Samrat Ganguly, Sudeept Bhatnagar, Akhilesh Saxena, and Rauf Izmailov are with NEC Laboratories America, Princeton, NJ. email: {samrat,sudeept,saxena,rauf}@nec-labs.com. Suman Banerjee is with Dept. of Computer Science, Univ. of Wisconsin, Madison. email: suman@cs.wisc.edu

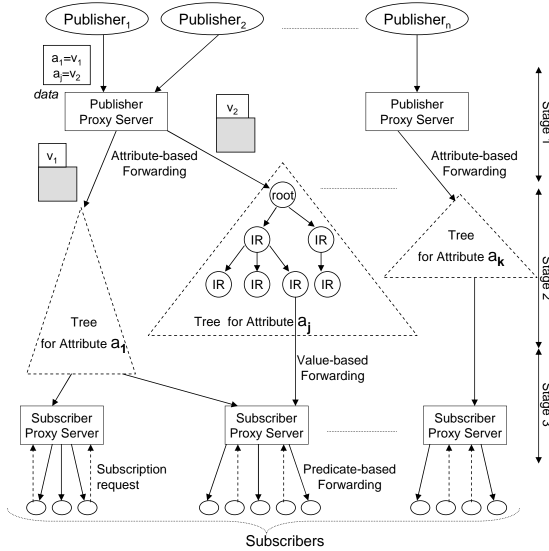


Fig. 1. Content-based Routing Architecture in Sieve

sage filtering and forwarding which can adapt to content popularity distribution and subscription profile. Given the space requirements, we provide algorithm for constructing an optimal filtering structure.

- Optimistic counting algorithm based on [6] for fast matching of complete subscription predicate.
- Since task distribution also leads to increase in number of messages in the system, we provide an approach adapting the task distribution based on content popularity and subscription interests.
- Label-based forwarding inside the network that limits the costly message-parsing operations [7] to the network edges.

Finally, micro-benchmark results from our real system implementation shows that Sieve can sustain a throughput of more than 5000 messages per second for 100000 subscriptions with predicates of 10 attributes.

II. Sieve Architecture

The Sieve architecture (shown in Figure 1) is based on the publish-subscribe [8] service model. A user registers its interest in certain content by means of a *subscription*. A content producer *publishes* a message into the Sieve network which sends it to all users with matching subscriptions. In this section, we describe the various architectural aspects of Sieve that help realize the above model.

A. Data model

In the Sieve infrastructure, an event notification from a publisher is associated with a message m containing a list of tuples $<$ *type, attribute name*(a), *value* $(v) >$ in XML format where *type* refers to data type (eg. float, string). Each subscription u (also in XML format) from a user is expressed as a selection predicate in conjunctive form as $u = P_1 \wedge \ldots \wedge P_n$. Each element P_i of u is expressed as $<$ *type; attribute name(a); value range(R)* $>$ where $R : (x_i, y_i)$. P_i is evaluated to be true only for a message containing $< a_i, v_i >$ such that $x_i \leq v_i \leq y_i$. A message m matches a subscription u if all the corresponding predicates are evaluated to be true based on the content of m. Disjunctions in the subscription predicates are handled by splitting the predicate into smaller conjunctive predicates and treating them separately.

B. Sieve Components

In order to efficiently distribute the parsing, filtering, and forwarding load, Sieve defines three logical roles that nodes play. In Sieve, a publisher contacts a *Publisher Proxy Server* (PPS) to publish its data. Similarly, a user contacts a *Subscriber Proxy Server* (SPS) to send subscription requests and to get notifications matching its subscriptions. There is one logical filtering tree corresponding to each attribute. Any message that contains a value for an attribute a is routed over the attribute tree for a. Each node in an attribute tree is called an *Intermediate Router* (IR).

Sieve is designed to operate on top of an overlay network with heterogenous characteristics. All nodes in the network can communicate with each other, however, the latency and bandwidth observed between different pairs of nodes varies. Each node in the Sieve network can be assigned one or more of the roles (PPS, SPS, IR) based on its processing and bandwidth capabilities. We assume a *coordinator node* (which could potentially be distributed over several physical nodes) controls the assignment of roles to the nodes. The publishers contact the coordinator to ask for nearest PPS, the PPS can contact it to find the roots of various attribute trees, and users contact the coordinator to find the SPS to which they should send their subscription. In the remainder of the paper, we do not detail the involvement of the coordinator node and focus on Sieve's content-based filtering framework.

C. Subscription process

A user sends its subscription containing its ranges of interest over multiple attributes to an SPS. The SPS of choice for a subscription is the one whose existing subscriptions have *maximum overlap* with the new subscription. The SPS stores the entire subscription for each user locally. It then subscribes to different range of values for different attributes on behalf of the users.

The SPS aggregates the individual user subscriptions u into *non-overlapping ranges* for each attribute. Consider a set U_a with all the user subscriptions u interested in attribute a. Each of these subscription is characterized by a range (x_i, y_i) on attribute a. If there are n such subscriptions in U_a, their ranges can intersect at no more than $2n$ distinct points over the content space. For example, consider two subscriptions with ranges $< 20, 70 >$ and $< 10, 50 >$ for a given attribute. These ranges intersect the content space at the distinct points $(10, 20, 50, 70)$. Let the ranges be defined by two adjacent intersection points (l, m) in the ordered set of intersection points. For the above example, the ranges are $(10, 20), (20, 50), (50, 70)$. SPS assigns a unique subscription-id (denoted as S with or without subscripts) to each of these ranges and sends it to the corresponding attribute tree root (as

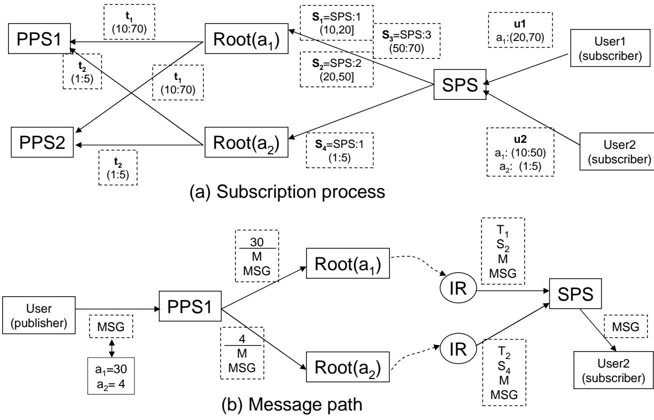


Fig. 2. Subscription movement and message routing

shown in Figure 2.a). Note that this subscription aggregation process ensures that the SPS subscribes to a value if and only if *some* user subscription is interested in it.

Each attribute tree root receives subscription S from one or more SPSs. The root finds the minimum and maximum value $t : (x_{min}, y_{max})$ in value space covered by union of all ranges in S from all SPSs. The root informs the range t to all the PPS.

D. Message filtering and forwarding

The goal of Sieve is efficient and correct delivery of messages to interested subscriber using content-based routing. In order to provide a high end-to-end throughput Sieve uses the concept of *filter pipelining*. Fundamentally, a pipeline of well-designed components increases the end-to-end throughput.

Sieve divides the complex task of event filtering and routing into three stages (Figure 1): a) *Attribute-based forwarding:*used for forwarding a message based on attribute name; b) *Value-based forwarding:* used for filtering the messages based on values of specific attributes and forwarding it to the correct SPSs; c) *Predicate-based forwarding* used for matching entire subscription based on the compound predicates and notifying the users. We now give a brief overview of the content-based routing of a message over the three stages as shown in Figure 1. The detailed description of these techniques is deferred to the subsequent sections.

Attribute-based forwarding (at PPS): Each publisher is assigned an attribute-based forwarding server (PPS). A publisher sends a new message (containing multiple attribute-value pairs) to its PPS. The PPS parses an incoming message to identify its attribute names. The message is forwarded to *all* attribute trees T_i corresponding to the attributes a_i present in the message (Figure 1). As an optimization, the PPS does not send a message to an attribute tree for a_i if the corresponding value v_i is not in the subscription bound (given by t_i set in subscription process). Before forwarding the message to T_i, PPS attaches the value v_i of attribute a_i as a *label* (Figure 2.b). As v_i can be of any length, labels can be a pointer to the location of v_i in the message. The PPS also attaches a unique *message-id* M to the message. M is common to all copies of the message irrespective of the attribute trees they are forwarded on.

Value-based forwarding (at IR): All IRs on an attribute tree use value-based forwarding. Upon receiving a message at the root of an attribute tree T_j, the objective is to deliver this message to all SPSs that have a subscription that matches value v_j. The value-based forwarding is implemented as a hierarchical forwarding tree structure using *Intermediate Router* (IR) providing two functionalities: message filtering and multicasting. Filtering restricts the multicast of a message to only those SPSs which have at least one subscription matching the value v_j contained in the message. In order to execute message filtering, each IR has a set of non-overlapping filters $f_{a,b}$ defined as

Definition: A *filter* f, is defined by range (a, b) (where $a, b \in$ content space V) and an associated set of nodes $\mathcal{N}$.

A filter f allows only those messages m to pass through whose label value v lies in its range, i.e., $v \in (a, b]$. A message that passes through f is forwarded to all the nodes in $\mathcal{N}$. Each IR has a list of filters (Figure 3). In an IR, a message is matched to a filter f using a range search on its label value v and then replicated and forwarded to the next hop nodes (IRs or SPSs) associated with f as shown in Figure 3. The construction of the value-based forwarding tree and the filters at each IR is based on the aggregated subscription profiles generated from each SPS. The IRs do not parse the message as the entire operation uses the value v attached as a label.

Predicate-based forwarding (at SPS): Each SPS maintains a list of complete subscriptions sent by the users and hosts a predicate-based forwarding server for matching user subscriptions. Based on messages received from the attribute trees, SPS matches the compound predicates of the subscriptions. For each subscription match, the message is forwarded to the corresponding user. The SPS uses an *optimistic counting algorithm* to avoid parsing the message while determining the final recipients of each message it receives. The SPS also maintains an efficient data structure to minimize the number of subscriptions that are considered for matching.

Example: Figure 2(b) shows a publisher generating a message which has two attributes a_1 and a_2. This message is published into the Sieve network by sending it to PPS2. From there the message is sent to the root of the two attribute trees with corresponding labels. The IRs forward these message to the SPS after adding labels that identify the tree-id T, the SPS subscription-id S that matched the value (in the label), and the message-id M. The SPS on receiving it performs complete matching to user subscriptions and forwards it to the interested user (User2).

In summary, the PPS performs filtering based on the *attributes* in the message. An attribute's filtering tree performs filtering on the *values* for that attribute in the message. The SPS combines the results from various attribute trees to identify the subscriptions that completely match a message.

III. VALUE BASED FORWARDING

As discussed in the previous section, the PPS nodes receive a message from the publisher, parses it to determine the

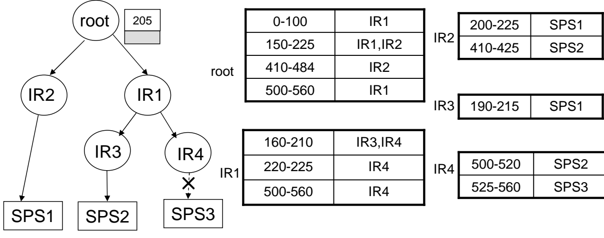


Fig. 3. Content-based Routing Architecture in Sieve

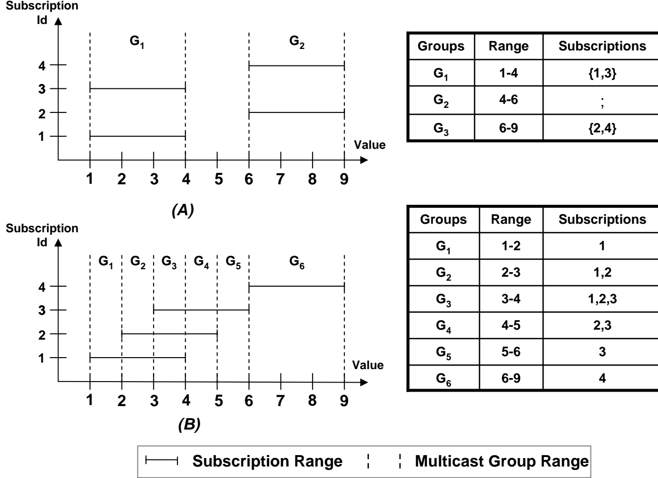


Fig. 4. Partitioning the content space into multicast group ranges based on subscription profiles

attributes it contains, and sends it to the corresponding attribute trees. Before sending the message to tree T_k, it assigns a unique message-id M, and a label containing the value v_k corresponding to the attribute a_k. We now detail the value-based forwarding operation in an attribute tree T_k.

An attribute tree contains subscriptions from multiple SPSs expressed as $(\text{SPS}_i : S_j)$. SPS_i is the unique-id of the SPS which subscribed to this attribute. S_j is the subscription-id that SPS_i assigned to the corresponding subscription range. T_k matches the value v_k in an incoming message's label to the subscription range that covers it, stamps it with the S_j of matching subscription, and sends it to SPS_i. SPS_i will assume that any message stamped with S_j matches the corresponding subscription range.

Two important issues govern the design of an attribute tree: the amount of *state space* required to store the different subscriptions and the corresponding *forwarding load*. A simple insight helps in understanding the impact of these issues: a message being sent to a bunch of subscriptions (based on its content) is essentially being multicast to a (possibly unique) group. For example, in Figure 4(A), there are 4 subscriptions for ranges $(1,4),(6,9),(1,4)$, and $(6,9)$ respectively. Using the unique end-points of the subscriptions, the content-space is partitioned into three multicast group ranges G_1 from $1-4$, G_2 from $4-6$, and G_3 from $6-9$. Of these groups, there is no interested subscription in G_2 whereas G_1 and G_3 have two interested subscriptions each. Thus, a message with value 2 will be multicast to subscriptions 1 and 3. Implicitly, this also means that multicast group G_1 acts as a filter that does not send the message to subscriptions 2 and 4.

Note that, a multicast group has to identify *all* subscriptions that must receive a message and *no* other subscription should get it. This requirement results in a large state space. In fact, the real magnitude of this problem emerges from the analysis of a simple case in Sieve: Consider 100000 subscriptions over a content space with values in range 1-100000. Consider 100000 subscriptions where subscription i is for range (i,100000). All values from 1 to 100000 form distinct end-points for this subscription sequence. Thus, we have a 100000 distinct groups with group G_i having i associated members. The state space for this example is $\sum_{i=1}^{100000} i = 5\text{x}10^9$ entries. Even with a mere 4 bytes (integer) to store the subscription-ids (group members), this requires 20GB of memory in the system. Furthermore, the state space explodes as n^2 for n subscriptions using the above example. This amount of state is infeasible to be stored at a single node. Furthermore, consider the amount of forwarding load explosion that this system can experience. For the above example, consider that we are getting a minuscule incoming message rate of 1 byte per second per value. In such a case, each of the $5\text{x}10^9$ entries will get one byte per second resulting a total outgoing traffic of 5GBps. Thus, forwarding load requirement for such a system can be huge due to the multicasting requirements.

Lastly, the solutions to minimize state space and forwarding load are highly dependent on the subscription distribution over the content space. For example, consider the two subscription cases in Figures 4(A) and 4(B) where we have 4 subscriptions each on the same content space. Suppose we characterize the total state space as the number of subscription-ids over all groups (the number of entries in the right-most column). Then the total state requirement in 4(A) is 4 units and 10 units in 4(B). Suppose we have a traffic of i units per second for value i. Then the total forwarding load at the node in Figure 4(A) to all the subscriptions is given by (1+2+3+4) mesg/sec to subscription ids $\{1,3\}$ and (6+7+8+9) mesg/sec to subscription ids $\{2,4\}$ resulting in 80 messages per second. However, for the subscription distribution in Figure 4(B), the total load is only 72 messages per second. Thus, while state space requirement is higher in Figure 4(B), the forwarding load is higher in Figure 4(A).

Our objective is to construct a hierarchical filtering structure by distributing both the forwarding space and load among multiple nodes. The construction is adaptive to the message value popularity distribution and subscription range distribution over the content space.

In order to capture the popularity distribution of different values, each node keeps aggregate arrival statistics for different values in form of a histogram. The histogram provides the distribution of values in a given unit interval over the entire content space. The histogram is updated using a sliding window average every time a message is received. From the histogram, one can easily compute for any range r in the value space, the fraction of traffic $p(r)$ with values in r. Note that the total traffic with values in range r is $\lambda p(r)$ where λ is the total message arrival rate.

A. Space and load partitioning

If the root node of the attribute tree cannot handle the space and forwarding load requirements, we distribute the filtering and forwarding task among multiple children. We observe that both the number of multicast group ranges and the associated forwarding load is an increasing function of the number of subscriptions. Thus by partitioning the subscriptions among multiple children, we can meet both space and forwarding constraints of each node. Each node can then serve a subset of all subscriptions by having a unique filter for each multicast group range thereby ensuring zero false positive delivery to the SPSs.

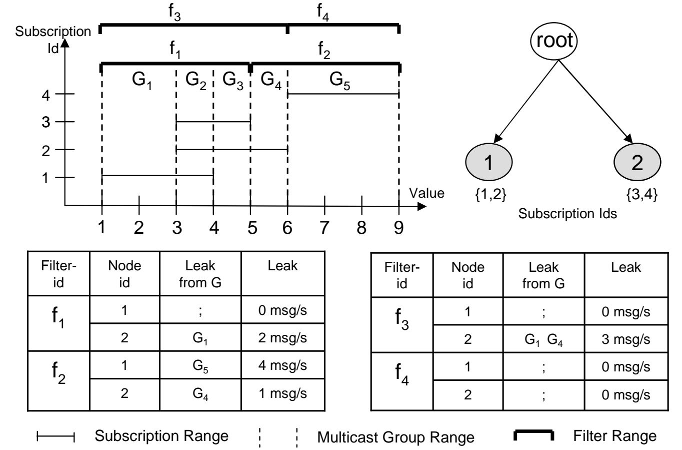
We assume that the available space and forwarding bandwidth for each node in a resource pool is given. Assume, in the current state, the set of all subscriptions is partitioned among k nodes all of which are children of the root. The subscription set in the node i is denoted as $\mathcal{S}_i$. The following subscription partition process is used when a new subscription request S is received at the root.

Subscription partition and movement: Consider a node i with a maximum forwarding capacity of $c(\mathcal{S}_i)$ and suppose that $l(\mathcal{S}_i)$ of its capacity is currently being used to forward messages to downstream nodes. $l(\mathcal{S}_i)$ is the forwarding load of node i. If the subscription S is assigned to node i, the increase in its forwarding load is $\delta_S^l = p(r)\lambda$ where r is the range of values subscribed in S. The above is true as each subscription identifies a unique SPS because of the user subscriptions being aggregated at SPS. One can now easily find the feasible set of nodes $\mathcal{N}$ for which $l(\mathcal{S}_i) + \delta_S^l \leq c(\mathcal{S}_i)$. Let g be the total number of multicast groups for a node j in $\mathcal{N}$ with subscription set $\mathcal{S}_j$. If S is assigned to node j, the total space requirement will increase by maximum $\delta_S^g(j) = g' + 2$ where g' is the total number of multicast group ranges spanned by the range of S. In order to minimize the increase in space requirement, S is assigned to node j for which $\delta_S^g(j)$ is minimum. However, if space requirement is not met by any of the nodes in $\mathcal{N}$, a new node is added to which the subscription is assigned. Note that the solution does not provide for load balancing; instead it tries to minimize the number of nodes while meeting their capacity constraints.

B. Hierarchical filtering

In order to have the filtering at each of the k leaf nodes, the root can forward each message to all of these nodes. However, simple forwarding leads to the following problems: a) The total outgoing message forwarding load λk at the root becomes high; b) the number of messages processed by each leaf node is high (number of message received is independent of the subscriptions handles by the leaf node) and c) increased overall network traffic. It is therefore worthwhile for the root to invest in the filtering process, albeit in a weak form, meeting the space constraint. In order to clarify weak filtering, we define *leak* as follows: A *leak* is the amount of extra traffic that is passed to a node with subscription set $\mathcal{S}$ which is not matched by $\mathcal{S}$ and thus should not have received it.

A leak occurs in a case where there is a *partial* overlap between a filter's range and an associated multicast group



Filter-id	Node id	Leak from G	Leak
f_1	1	;	0 msg/s
	2	G_1	2 msg/s
f_2	1	G_5	4 msg/s
	2	G_4	1 msg/s

Filter-id	Node id	Leak from G	Leak
f_3	1	;	0 msg/s
	2	G_1 G_4	3 msg/s
f_4	1	;	0 msg/s
	2	;	0 msg/s

Fig. 5. Mapping into multicast trees based on subscription profiles

range. Let us define $\mathcal{G}(f)$ as a set of multicast group ranges covered by f. Let the subset of group ranges in $\mathcal{G}(f)$ that is of interest to subscriptions in leaf node i be $\mathcal{G}(i)$. It follows that any message passed by f intended for multicast group ranges $\mathcal{G}(f) - \mathcal{G}(i)$ contributes to the traffic leak for node i. Therefore, the total leak for a filter f is given by

Definition: The leak of a filter f (denoted as L^f) is defined as the total traffic leak *caused by* f given as

$$L^f = \sum_{i=1}^{k} p(\mathcal{G}(f) - \mathcal{G}(i))\lambda,$$

where $p(\mathcal{G}(f) - \mathcal{G}(i))$ is the fraction of traffic with the values in the partition ranges $\mathcal{G}(f)$ and not in $\mathcal{G}(i)$ as obtained from the histogram. A filter f with a non-zero leak L^f is called a *weak filter*.

The example shown in Figure 5 illustrates the leak from filters for a given set of subscriptions. In this example, we assume that each unit value interval $(i : i + 1)$ has traffic of 1 message/sec. In order to find the leak from f_1, consider leaf nodes 1 and 2. Suppose that node 1 hosts subscriptions 1&2 with ranges (1-4) and (3-6) respectively. Node 2 hosts subscriptions 3&4 with ranges (3-5) and (6-9). As shown in Figure 5, partitioning these ranges results in 5 multicast groups $G_1, \ldots, G_5$. We obtain $\mathcal{G}(f_1) = \{G_1, G_2, G_3\}$, $\mathcal{G}(1) = \{G_1, G_2, G_3\}$, $\mathcal{G}(2) = \{G_2, G_3\}$,. Therefore, the leak for node 1 from f is zero as $\mathcal{G}(f_1) - \mathcal{G}(1) = \emptyset$ while leak for node 2 from f is 2 messages/sec as $\mathcal{G}(f_1) - \mathcal{G}(1) = \{G_1\}$. Similarly, computing the same for filter f_2, we obtain the total leak from both filter f_1 and f_2 to be 7 messages/sec. The interesting point to note is that without any filter (equivalent to having one filter), the total leak is 7 messages/sec as well. However, with filters f_3 and f_4, the total leak is reduced to 3 messages/sec. Thus we observe that creating proper filters is important in order to exploit the benefit of filtering at root. Next we present an optimal polynomial time algorithm for filter construction.

C. Optimal Filter Construction

We are given the set of subscriptions and their location in one of the leaf nodes. Let F_i denotes a set of i filters and $\mathcal{L}(F_i)$

denotes combined leak from all filters in F_k, i.e, $\mathcal{L}(F_k) = \sum_{j=1}^{i} L^{f_k}$. We want to construct i non-overlapping filters such that:
1) They span the subscription space, i.e., any value that any subscription has subscribed to, must pass through one of the filters.
2) $\mathcal{L}(F_i)$ is minimized.

We now present an optimal dynamic programming algorithm for the above problem. The algorithm runs for $k-1$ iterations where k is the number of filters to be constructed. In the ith iteration, the algorithm computes the best filter set if we were allowed only $i+1$ filters (denoted by F_{i+1}). This is done using the filter sets generated in the $i-1^{th}$ iteration and adding a new filter strategically.

A key property of a filter's leak as defined above is that it is self-contained and independent, i.e, the total leak due to a filter f is computed using only the portions of subscriptions that overlap it and the value of the leak remains the same *irrespective of how the remaining filters are designed* (recall that the ranges of filters are non-overlapping). This key property is used in the dynamic programming approach.

Let the ordered set $V : \{v_1 \ldots v_n\}$ denote the set of distinct edge points in the value space in increasing order corresponding to either start or end of multicast group ranges. Let an ordered subset V_k denote $\{v_k \ldots v_n\}$. In order to take advantage of dynamic programming's book-keeping capability, we store some partial information after each iteration of the algorithm. This information is in form of a filter set over a *subset* V_k. We denote by $F_i^j(V_k)$ the filter set defined over V_k when we have i filters *such that* $1st$ *filter spans* $v_k, \ldots, v_{k+j}$ *and the rest* $i-1$ *filters are spread over the range* $v_{k+j+1}, \ldots, v_n$. Note that, if $n-j-k > i-1$ then the set F_i^j is meaningless since there are not enough values to assign the $i-1$ non-overlapping filters over that range. Let $F_i^*(V_k)$ define the optimal filter set of i filters over V_k such that total leak $\mathcal{L}(F_i^*(V_k))$ is minimized. This is obtained by finding value of j for which $F_i^j(V_k)$ is minimized.

The base step of the algorithm involves computing the filter sets assuming that there are two filters. [1] For this, we compute all the sets $F_2^1(V_k)$ to $F_2^n(V_k)$ and the corresponding leak $\mathcal{L}(F_2^j(V_k))$. $F_2^*(V_k)$ is obtained as $\min_j(\mathcal{L}(F_2^j(V_k)))$ for all $j = 1:n$. The base case is repeated for all n subsets $V_1 \ldots V_n$.

In the subsequent iterations i where we need to find the filter sets with i filters, we utilize the sets $F_{i-1}^*(V_1) \ldots F_{i-1}^*(V_n)$ instead of combinatorially testing all possible filter assignments with i filters. As mentioned above this is possible due to the independence of the filter-leaks with respect to the other filter leaks. We note that $\mathcal{L}(F_i^j(V_k)) = L^f + \mathcal{L}(F_{i-1}^*(V_{k+j}))$, where L^f is the leak for the first filter in $F_i^j(V_k)$ spanning $v_k \ldots v_{k+j}$. Thus we can find j for which $\mathcal{L}(F_i^j(V_k))$ is minimized giving us the filter set $F_i^*(V_k)$. Repeating the above steps for all k we get the optimal filter set $F_i^*(V)$ where $V = V_1$.

[1]The only case when just one filter covering the entire range suffices, is when there is only one subscription. With two or more distinct subscriptions, two or more filters will have a lower total leak than a single filter.

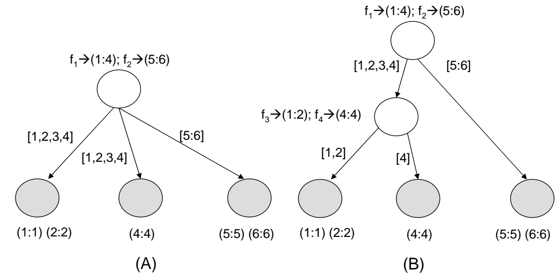


Fig. 6. Example showing the need for multi-stage filtering. In case (B) the amount of leak to leaf node is zero

D. Multi-stage filtering

Although the above solution minimizes the leak at the root, it does not completely eliminate it due to the space constraints. One can add multiple layers between the root and the leaf nodes to successively filter messages leading to zero leak (as shown in Figure 6). In the multi-stage arrangement, the total event space is partitioned and each partition assigned to a given node in that stage. Partitioning is done such that each partition has equal number of multicast group ranges. The number of partitions is determined by the amount of leak from the previous stage. Our experiments under most scenarios indicates that single stage framework is sufficient to handle the leaks.

IV. Predicate based forwarding at SPS

Each SPS contains various user subscriptions and is responsible to subscribe to appropriate attribute trees on behalf of these subscriptions. Based on the incoming messages, SPS performs predicate matching and forwards the message to users with matching subscriptions. However, matching a single message content against all subscription is inefficient. Furthermore, each end-user subscription predicate is multi-dimensional whereas the received message only corresponds to one dimension. We next present optimistic counting algorithm – a predicate evaluation mechanism based on an efficient data structure that achieves fast subscription matching.

A. Optimistic Counting Algorithm

Consider a single user subscription s with a selection predicate defined on n attributes. Assume that the SPS subscribes to *all* n attribute trees on behalf of the user subscription. If the SPS receives n copies of a message corresponding to each attributed tree, it implies that the user subscription is evaluated to be true. Any less than n copies would mean otherwise. Therefore, it is possible to establish whether a subscription is matched by simple counting. In essence, the algorithm recognizes that the messages reaching an SPS are already filtered along different attribute trees and tries to avoid further local matching. This simple observation serves as the basis for the optimistic counting algorithm.

The algorithm is considered optimistic because it assumes that all copies of the messages are definitely going to arrive if they are going to match the subscription and that they will arrive in a reasonably finite time.

In order for the above algorithm to be of practical use, there are several problems that need to be addressed: 1) Each subscription S of an SPS for a given attribute tree is a union of several user subscriptions s for that attribute. Therefore, in order to take any action for a message m, SPS must know which attribute tree it arrives from and the corresponding user subscriptions, without parsing the message content; 2) messages m can come asynchronously from different attribute trees; 3) the messages can be arbitrarily delayed. Hence, the SPS has to maintain the counts of multiple subscriptions for some duration; 4) Subscribing to multiple attributes for a subscription can result in extra overhead because of the multiple copies of the message received. Thus, a mechanism is required to curtail the number of attributes for which the SPS should subscribe. We address these issues in the remainder of this section.

B. Forwarding Structure at SPS

We describe forwarding mechanism at SPS based on the action taken on a given message and the corresponding data structures used. When a message m arrives at the SPS, it identifies the attribute tree id T and the subscription id S as shown in Figure 7 (these information were put as a label by the last hop IR node in the attribute tree). From the tree id T_i, the corresponding user subscription mapping table is accessed. There is one table for each attribute where a row is indexed by the subscription-id S_i used by the SPS to subscribe to the attribute tree. Using this table, we map the subscription-id S_i in m to a list of constituent user subscriptions-ids s. Thus with two lookup operations, we get the user subscription list who are interested in m (for example, u_1 and u_2 in Figure 7).

SPS also maintains a subscription matching table (Figure 7) indexed by the unique-id of the user subscription u_j. For each subscription u_j, the *attribute count* field contains the number of attributes in u_j for which the SPS has a corresponding subscription to some IR. The *Full Match Flag* indicates whether the *attribute count* represents the actual number of attributes of the subscription. A value of 1 in this field indicates that the SPS has subscribed to *all* attribute trees for this subscription. This implies that if the number of copies of m received at SPS is equal to *attribute count*, then m matches u_j whereas a 0 value for *Full Match Flag* would only indicate a partial match. The table contains a hash-queue of *pending message-ids and their counts* which contains the list of all messages which have matched along some (but not all) attributes indicated by them having a count greater than zero but less than *attribute count*. Merely counting the number of copies of a message received is not enough to match user-subscriptions. Only message arriving from specific trees should be able to increment the count for a specific subscription.

In Figure 7, for the arriving message m, the list of pending messages for u_1, u_2 is traversed. Using the unique message-id M_1 of the message, we check if the message is already pending for either one of them. It increments the count for M_1 in both u_1, u_2. Since the count of M_1 for u_1 now goes to 3, it is a complete match as the *Full Match Flag* is 1 and

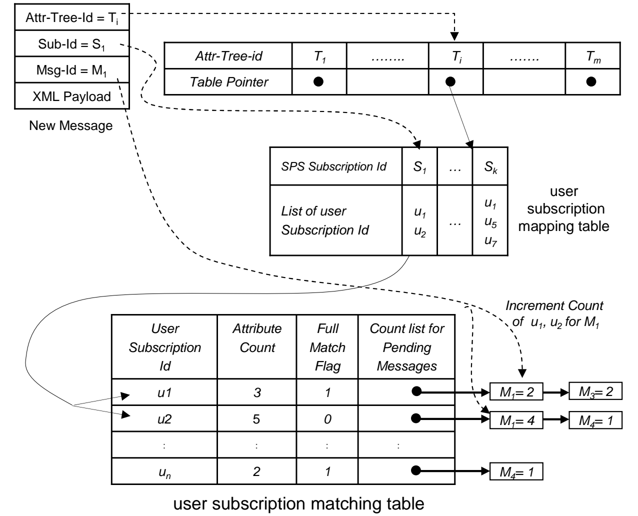


Fig. 7. Matching a message against the resident subscriptions using the Optimistic Counting Algorithm

the message is directly sent to u_1. Also, the count for M_1 for u_2 reaches 5 (which is the required attribute count) implying that M_1 matches u_2. However, this is a partial match since the *Full Match Flag* for u_2 is 0 indicating that there are more than 5 attributes in u_2. In this case, now the message is sent to the message cache (described next) to fully match its content against partially-matched subscriptions (like u_2).

C. Message Cache and Timer Management

An SPS can receive multiple copies of a message with each arrival possibly resulting in some partial matches. Without any special mechanism, this would require parsing the message each time to test for a complete match. We alleviate this overhead using a *message cache*. Whenever a new message arrives, it is added to the message cache. Any subsequent copies of the message are used only for the counting algorithm. Furthermore, the message is parsed lazily, i.e, only when the first partial match occurs.

A feasible implementation of the message cache requires the use of timers. There are two uses of timers in our setup. A timeout for a message in a subscription's pending queue indicates that the message did not match and the entry can be discarded. A timeout in the message cache indicates that a duplicate copy of the message is no longer expected and the message can be purged from the cache. Note that both timer expiry durations depend upon the maximum possible delay between different attribute trees. In practical scenarios this delay would be small.

In worst case, the SPS may have to start one or more timers with each message arrival. Thus, the number of active timers can quickly grow into an infeasible number. We control this overhead by grouping the expiry events into buckets and using a timer expiry to process all events in the corresponding bucket [9].

D. Selective Subscriptions

We allow SPS to subscribe to only a subset of attributes for each subscription to reduce the extraneous messages. The rationale for this choice is that certain attributes and values would be very common (especially in skewed distributions) vis-a-vis the others. By subscribing to a popular attribute, the SPS does not gain much in terms of filtering. For example, a subscription for the entire content space for a particular attribute, matches all messages having that attribute (implying no filtering). In such a case, the SPS decides to subscribe to an extra attribute only if it gets significant benefit with respect to the filtered traffic. Specifically, the *selectivity* of a subscription-range determines whether the SPS subscribes to it or not. Lastly, the SPS must subscribe to *at least one* attribute from each of its user-subscriptions.

While selectivity reduces extraneous traffic, it introduces a new problem. Now merely counting cannot ensure that a subscription matched completely with a message. The count can only ensure that the subscription matches in all attributes *that were subscribed to*. Unless the message is parsed, the SPS cannot determine whether the remaining attributes match or not. However, the utility of the selective subscription technique is that it reduces incoming message rate at SPS while still retaining the ability to identify the *non-matching subscriptions*. If k of a subscription's attributes have been subscribed to and the SPS receives at most $k-1$ copies of the message from the corresponding trees, then irrespective of the values of its remaining attributes, the message does not match the subscription. This reduces the number of subscriptions that need to be matched against a message.

E. Choosing Subscription Ranges

The selective subscription mechanism requires a technique to determine the ranges for each attribute to subscribe to. We present a simple strategy to solve this problem using the event arrival statistics, and the cost of matching and counter-incrementing operations.

We consider the SPS to consist of two separate units: 1) a matching unit which identifies the matching subscriptions for each message, and 2) a forwarding unit, which forwards the message to all matching users. Clearly, if the forwarding unit cannot handle the forwarding load, we need to move some subscriptions to other SPS as the forwarding load consists entirely of desired messages. Hence for this discussion we consider the matching unit.

Intuitively, adding an extra subscription is only useful if it increases the system throughput. Thus, if M *distinct* messages arrive at the SPS per unit time, then the SPS becomes a bottleneck if it matches less than M messages per unit time. Here matching a message refers to the message being sent to the forwarding unit along with all matched subscriptions. Using selective subscriptions, we can increase the throughput of the matching unit.

Let the average time taken to fully match a message against a subscription be t_f and the time taken to increment a message counter be t_c. We expect t_f to be much higher compared to t_c because a full-match involves parsing the message and matching all its attributes against all attributes in the subscription. The expected time T to process a message by the SPS is given by $n_1 * t_f + n_2 * t_c$ where n_1 is the number of full matches it has to do and n_2 is the number of counters it has to increment. If T is less than $1/M$ then the SPS is not the bottleneck. Otherwise, we can try to reduce T by subscribing to additional dimensions for some subscriptions.

If SPS subscribes to more attribute trees, it results in more incoming messages. That in turn increases the total counter-increment cost as each message would likely increment some counters. On the other hand, subscribing to an additional attribute tree for a subscription reduces the likelihood that it would have to be fully matched. This because then a full-match is required only when *an additional attribute has already matched in the counting domain*. There exists an optimal point beyond which the matching unit's throughput starts decreasing when we subscribe to extra attributes. We try to reach the optimal point by an incremental algorithm. The SPS calculates the effective throughput if it subscribes to a range with *least* extra traffic. If the effective throughput increases, it subscribes and checks the next candidate range, otherwise it stops. We show in the evaluation section that this simple strategy can significantly increase the throughput of the matching unit under different circumstances.

V. Evaluation

We now show the effectiveness of the Sieve architecture by running several experiments over a prototype implementation and some simulations.

We use zipf distribution to generate both the subscriptions and messages so that value i occurs with probability $i^{-\alpha}$ (normalized by number of values) where α is the zipf parameter. Since lower values of the value-space are more prevalent in the zipf generation, we permute them using a random permutation vector so as to disperse the popular values in the distribution. To generate the subscription ranges, we draw a number from the zipf distribution as the lower end of the subscription range. The higher end of the subscription is generated using a uniform width with a specified mean (that is varied in the experiments). Since a subscription exhibits interest in *all* values within its range, this technique allows us to have more subscriptions concentrated around the more popular values. Furthermore, using subscription width as a parameter allows us to control the overlap between multiple subscriptions giving us a wider test area for Sieve.

For the evaluation purposes, we need the following definitions:

Definition: The *per-message processing time* of a node is the mean time it takes for it to identify *all* the subscriptions that completely match a message.

Definition: The *throughput ratio* of a node is the ratio of the mean message inter-arrival time it sees and its per-message processing time. A throughput ratio of less than 1 is a must for a node to be able to seamlessly filter and forward an incoming message stream.

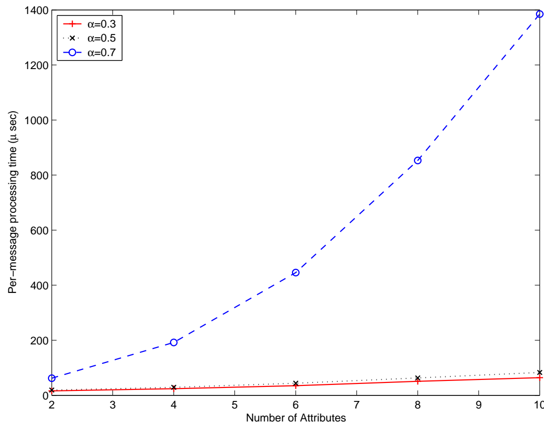


Fig. 8. Per-message processing time with same distribution for publication and subscription.

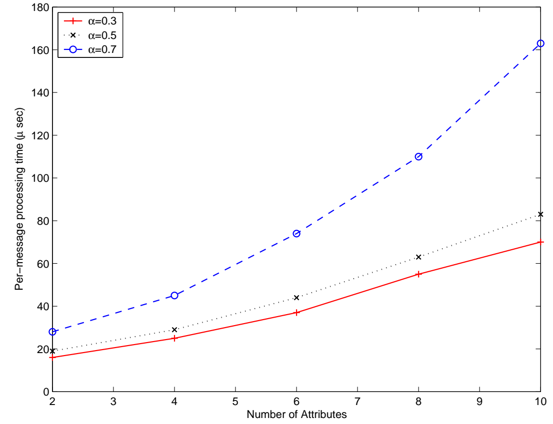


Fig. 9. Per-message processing time with different distribution for publication and subscription.

A. Implementation

We have implemented the Sieve system prototype using the optimal filters at IRs (defined in section III-C) and the optimistic counting algorithm at the SPS (defined in section IV-A). We show some performance results taken over a cluster of nodes. The key performance metric is the effective throughput of the *matching unit* of an SPS. We chose this as the performance metric of interest for two reasons: 1) the filtering cost at IRs is much lower compared to the SPS because the match is on a single attribute for aggregate subscriptions. 2) the actual throughput of the SPS depends on the subscriptions and the message arrival rate. Hence the capability of the system is better illustrated by the matching throughput rather than the forwarding throughput.

The following results were taken on a set of 13 Pentium-4 2.8Ghz machines with 1GB of RAM connected over 100Mbps Ethernet. One machine acts as both publisher and PPS thus generating the messages, attaching the value-based labels, and forwarding the messages to the appropriate IR nodes. We have one subscriber node that generates 100000 subscriptions and sends them to the single SPS node. The SPS computes its local tables and subscribes to the appropriate ranges over *all* attributes implementing the *full-match* version of the counting algorithm.

The publisher generates 10000 messages per second with each message carrying a payload of 512 bytes. The payload includes the XML message with varying number of attributes and the rest of it is padding data. The number of attributes and the traffic patterns are varied to test the system's performance. Our first experiment generates a stressful workload of subscriptions and publications. In this case, we generate subscriptions and publications using a zipf distribution with same value of α. Furthermore, we use the *same* permutation vector at both publisher and subscriber end to generate the case where the most number of subscriptions are for the most popular events. This results in a message from the range with maximum arrival rate matching a large number of subscriptions (thus resulting in a large number of counter increments). Hence, the performance of the matching unit in this scenario is expected to form a worst case for Sieve for the corresponding α. Figure 8 shows the actual time it takes for the SPS to match and forward *all* copies of a message from the system for varying number of attributes and different α. We can observe the following: 1) In the best case, the SPS needs only 16μsec per message of matching time. Thus, the system is not only capable of sustaining a throughput of 10000 messages per second, but has the capacity to scale upto 60000 messages per second (network permitting), 2) Increasing number of attributes increases the matching time due to more copies of messages arriving and thus increasing the number of counter-increment operations, 3) An increase in α results in a super-linear increase in the matching time. As discussed above, this is caused by our choice of having the same values popular both among publishers and subscribers.

The second experiment tests the system in a more general condition. For this experiment, we have different degrees of popularity of various values for subscribers and publishers. We keep α for the publisher at 0.5 and change the alpha for subscribers from 0.3 to 0.7. We still have the artificial sharing of popular values introduced by the identical permutation vectors so that only the relative interest level in a particular value changes (but the popularity index of a value amongst all values does not). Figure 9 shows the results of this experiment. We find that the message processing times have reduced significantly by changing the interest level in the values. This happens because a message carrying a popular value can now have less number of interested subscriptions (resulting in lower number of counter increments). This shows that in an average situation, Sieve is likely to perform very well.

An important measure of the pipelining effect is the amount of throughput the bottleneck node can provide with respect to its input traffic. Our experience with the system suggests that the bottleneck node in Sieve is invariably the SPS. The next experiment aims at quantifying the throughput that the SPS can achieve by evaluating its throughput ratio over diverse conditions. We set α to 0.5 for both publisher and subscriber with the same permutation vectors to generate subscriptions and messages having the same values more popular. Figure 10

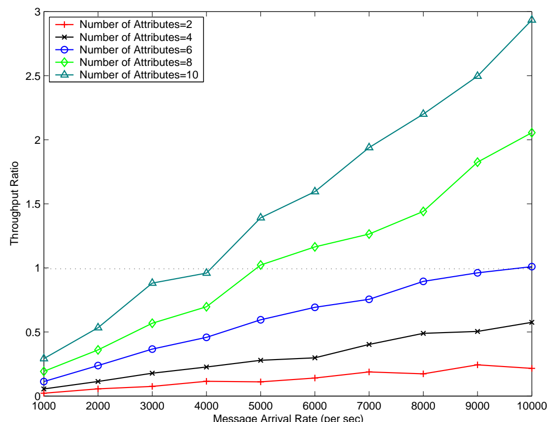


Fig. 10. Throughput Ratio of the SPS with increasing arrival rate for same value distribution for publications and subscriptions.

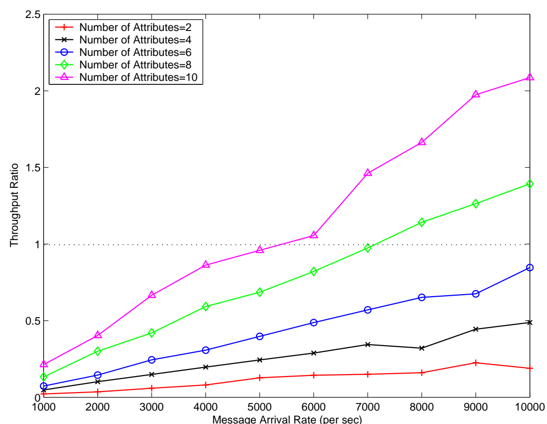


Fig. 11. Throughput Ratio of the SPS with increasing arrival rate for different value distribution for publications and subscriptions.

shows the throughput ratio of the SPS with different number of attributes in the system with increasing message arrival rates. As mentioned earlier, a throughput ratio of less than one is mandatory for a node to seamlessly handle its incoming traffic. The figure shows us that while supporting 10 attributes, we can handle around 4000 messages per second while serving 100000 subscriptions. Figure 11 shows the throughput ratio for the case when the subscription distribution is uniform with the publication distribution having an α of 0.5. In this case, we can see that one SPS can support around 5500 messages per second while supporting 10 attributes with 100000 subscriptions. These results strongly establish the viability of Sieve in supporting high-rate message streams.

Lastly, we show the impact of increase in number of subscriptions on the processing time for individual messages. For this experiment, both publication and subscription α were set to 0.5 and both had identical permutation vectors. Figure 12 shows the result. We see an almost linear increase in the total message processing time as the number of subscriptions increase. The reason behind this increase is that each new subscription is added to the table corresponding to each of its attributes thus increasing the number of subscriptions a

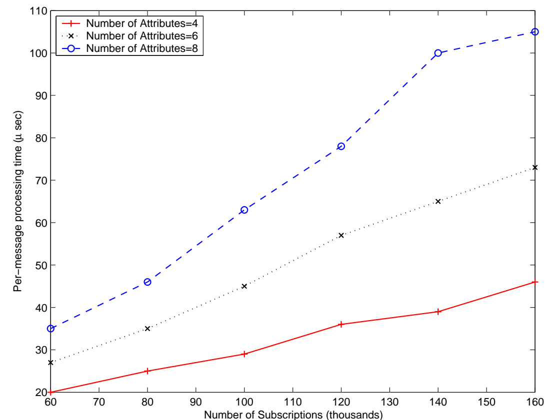


Fig. 12. Per-message processing time with increasing number of subscriptions.

message matches. However, our data structure at the SPS (with 3-level lookup) ensures that a subscription is only matched against a limited number of messages (those that match it in some attributes). This is why the increase in per-message processing time is marginal.

B. Comparison with Multi-dimensional Matching

While Sieve partitions the task of filtering among one-dimensional attribute trees, an alternate approach is to have multi-dimensional filtering at each node. We compare the Sieve approach with multi-dimensional matching by using R-tree[2] [10] to index subscriptions (as used in [5]). For this, we have 100000 subscriptions generated with $\alpha = 0.5$ as in the previous section. We divide the subscriptions among 10 nodes, so that each handles only 10000 subscriptions. A dispatcher node sends each message to all the 10 nodes which identify the individual subscriptions that the message matches. Since, our prototype of Sieve uses one IXR node per attribute, this gives the R-tree based system extra computing advantage unless the subscriptions are on 10 attributes.

First we show the processing time per-message at *one* R-tree node with publisher generating messages with same α. Note that since each node is effectively identical and operating in parallel, the system throughput is governed by the processing time at each node. Figure 13 shows that the throughput due to our counting-based algorithm is consistently higher than that of R-tree. We emphasize that this throughput only corresponds to the matching algorithms and not the network portion. With large number of attributes, Sieve almost doubles the throughput that R-tree can attain. This clearly shows that Sieve's approach of splitting the filtering task into multiple one-dimensional matching and then combining the results works well.

The second result in this set shows the impact of number of subscriptions on a multi-dimensional matching system. We vary the number of subscriptions from 6000 to 16000 on an R-tree node (equivalent to 60000 to 160000 subscriptions

[2] We used the publicly available implementation from http://www.cs.ucr.edu/ marioh/spatialindex/ with default parameters.

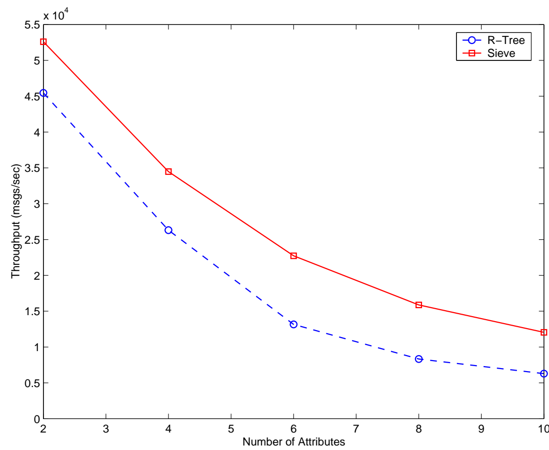


Fig. 13. Comparison of matching throughput of R-tree based matching with Sieve with increasing number of attributes.

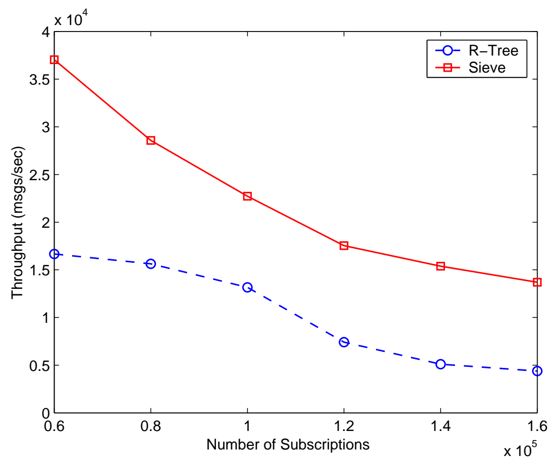


Fig. 14. Comparison of matching throughput of R-tree based matching with Sieve with increasing number of subscriptions.

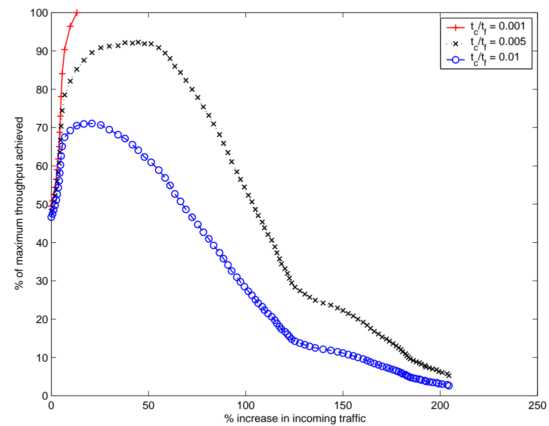


Fig. 15. Maximum throughput achieved by subscribing to additional attributes for different cost ratios t_c/t_f.

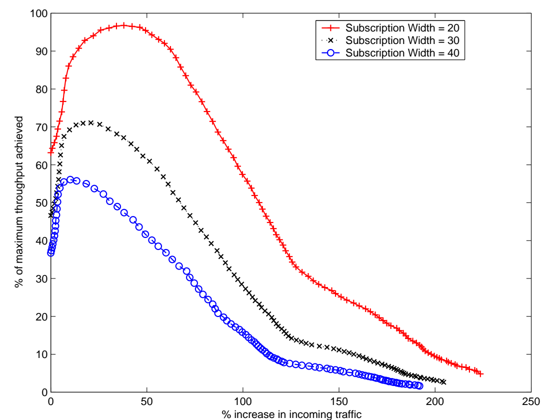


Fig. 16. Maximum attainable throughput by subscriptions with different having different width.

in Sieve with 10 attributes) with the same distribution of subscriptions and publications as above and with the subscriptions and messages having 6 attributes. Figure 14 plots the throughput that the matching process attains for the R-tree node in comparison with the Sieve system. Again we see that Sieve has much higher throughput even when the system has to match a large number of subscribers.

C. Selective Subscription

This set of experiments shows the benefits of selective subscription mechanism using simulations. We consider 10000 subscriptions interested in 5 attributes on an average. There are 50 different attributes in the system each having 10000 distinct values. The subscriptions ranges for each attribute are chosen independently using the method detailed above. The attributes and values are zipf-distributed but are are independently chosen. We have 20000 messages arriving per unit time.

We first show the viability of the selective subscription approach with different ratios of time for full-matching (t_f) and counter increment (t_c). The average t_f for matching a message with a subscription is set as 0.0001 units of time. The average range width of each subscription is 30 units. The ratio t_c/t_f is set to three different values 0.01, 0.005, and 0.001. Figure 15 shows the increase in matching unit throughput in these three cases. There are several things shown by the figure: 1) In the base case when SPS subscribes on only one attribute per subscription, the throughput of its matching unit is low making it the bottleneck. In this experiment, it is around 45% of the distinct message arrival rate. 2) As the SPS subscribes to more attributes per subscription, the matching unit throughput (and hence the system throughput) increases initially. However, beyond a certain point, the increased cost of counter increments outweighs the gains attained by reducing number of full-matches. 3) The smaller the ratio t_c/t_f, the higher the throughput we can attain. The reason being the ability to add extra subscriptions and reducing the number of full-matches without paying much in terms of increased counter maintenance cost. 4) In two of the three cases, the maximum reached throughput is less than 100% of the arrival rate. In Sieve, this serves as an indication that the SPS is overloaded and some of its subscriptions need to be off-loaded

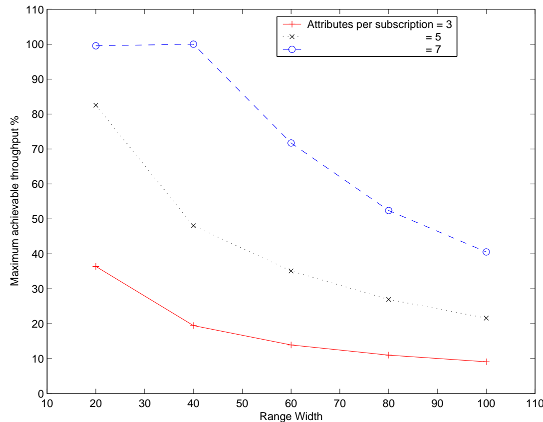


Fig. 17. Maximum attainable throughput by subscribing to additional attributes for different cost ratios t_c/t_f.

to another SPS.

The next experiment shows the impact of different subscription range widths on the attainable throughput. For this experiment, t_f is set to 0.0001 time units and the ratio t_c/t_f is 0.01. Figure 16 shows the results of the experiment. We see that as the width of the subscriptions increase, the maximum attainable throughput reduces. The reason for this reduction is that the messages matching the new subscription are likely to partially-match more number of subscriptions.

Our next experiment aims at identifying the impact of various parameters on the maximum attainable throughput. We set the ratio t_c/t_f to 0.005, the zipf parameter for subscriptions and publications is set to 0.7, the subscription range width is varied from 20 to 100, and the number of attributes on an average per-subscription is successively set to 3,5,7. Figure 17 shows the maximum attainable throughput for different parameters. There are two important observations from these figures: 1) With larger subscription width, the maximum attainable throughput decreases because each additional subscription results in a larger number of potential counter increment operations for the newly added traffic. 2) Larger the number of attributes in a subscription, the larger the possible throughput. This because we have more dimensions to add and improve the throughput.

VI. Related Work

In the recent past, a large body of work has emerged focussing on the problem of large scale selective dissemination of information to users. Several solutions were proposed based on using the multicast model. Using conventional multicast model is not scalable as the number of multicast trees can grow up to 2^n to capture all possible subscriber groups. The channelization problem formulated in [11] provides a solution to map sources and destinations to a limited set of multicast trees to minimize the unwanted message delivery. Another category of work [12], [13], [14], [15] creates a limited number of multicast trees by proper clustering of user subscription profiles. In the above solutions, filtering is done at the source, at the receiving point, or both. In contrast, authors in [16], [17] proposed the use of filters in the intermediate nodes in a given multicast tree for selective data dissemination. [17] provides a solution to filter placement and leak minimization problem. In multicast based approaches, the forwarding path of a message is restricted to pre-defined multicast tree topology. Although these approaches can apply well in topic/subject based or messages with single attribute, they are not suitable to support general predicates over multiple attributes.

In Sieve, we use multicast model based approach in value based forwarding in the attribute tree. In our solution, we solve the joint problem of filter construction at each node and multicast tree creation, which is not explored in existing work.

The added advantage of associating subscriptions to a multicast tree is marginal as the complex predicate has to be finally matched either at source or receiver. Instead of restricting to a multicast model, a general model is to create a routing network composed of content-based routers as proposed in Siena [1], [3] and Gryphon [4]. A content-based router creates a forwarding table based on subscription profiles and performs both data filtering and forwarding based on predicate matching. As with any data distribution network, the speed of matching the subscription predicates at each content-based router determines the sustainable throughput. The goal of content-based routing is to provide processing latency meeting the wire-speed.

In both Siena and Gryphon, each router may need to keep states about all subscriptions. Even though Siena [3] proposed subscription merging to minimize states, the resultant benefit is not applicable with subscription deletion. In Sieve architecture design, we are particularly concerned about the subscription states as that determines the message processing speed and forwarding bandwidth. A large subscription state space cannot be accommodated in main memory or processor cache for fast processing [18]. We specifically provide solutions to move subscriptions among leaf nodes (IR) such that space constraints and forwarding capacity are met.

A significant amount of research [18], [19], [20], [6], [1], [21]has been done on finding better solutions for general predicate matching at a single node. As a centralized solution using a single node may not support the ever-increasing rate of information flow, Sieve tries to distribute the matching complexity among multiple nodes. Although Sieve architecture implements a distributed data structure to match a message, the solution is based on many ideas from the single node based solutions. The reverse indexing structure used by Sieve to map content-space to subscriptions is similar to those used in [6]. Both [6] and [1] proposed a variation of counting based mechanisms to match predicate in a single node. However, to apply the counting method to a distributed engenders new problems that we discuss in section IV-A.

In certain solutions such as Siena [1], [3], [4], the message dissemination path is coupled with the subscription movement path and therefore lacks the routing flexibilities. In contrast, solutions [22], [23], [24], [25], [26], [27], [28] based on indirection use rendezvous points in form of broker nodes where messages meet subscribers. Sieve infrastructure is also based on the indirection philosophy (expounded in [29]), however, the sequence of indirections is used to partition the

complex task of predicate matching by defining separate roles. Unlike the above solutions, the goal of Sieve is to match time complexity of each role with the message arrival rate in supporting high bandwidth data stream.

Content-based information dissemination over P2P network was proposed in XROUTE [30]. Their main concern is network bandwidth usage and minimizing the size of the routing tables. In this work, our primary objective fast end-to-end message processing and delivery.

The notion of weak filtering as discussed section III-B on hierarchical filtering has been used in summary based routing [5] and [19]. In contrast to the above, use of weak filters in Sieve is motivated by space constraint and the design objective is to minimize leak for which we provide optimal filter construction algorithm.

Another body of works such as topic-subject based SCRIBE [31] over P2P substrate and XML based mesh routing in [7] looks at the reliability and fault-tolerance issues. In supporting XML format information dissemination, several solutions have been proposed such as [7], [32], [33]. Sieve supports XML format as well, however, it restricts the XML parsing operation to the network edge (PPS and SPS).

VII. Conclusion

We presented Sieve as an infrastructure solution for content-based forwarding of high-rate message streams. The key insight enabling Sieve's handling of high-rate streams is the concept of filter pipelining. Sieve divides the filtering task into smaller components, each with low space requirement and fast processing operation. Collectively, these components form a filtering pipeline providing the basis for high end-to-end throughput. The fundamental benefit that Sieve exhibits is that partitioning of tasks expedites the filtering process. This is illustrated by a sustained throughput of more than 5500 messages per second with 100000 subscriptions over 10 attributes.

Our work represents a step towards accomplishing a universal content-based filtering and routing network. In order to make a complete system, we plan to address issues related to the fault-tolerance to broker node and reliable message delivery as future work. In its current form, Sieve is meant for performing distributed multi-dimensional subscription matching. We aim to support regular expression matching and text search as filtering functions. Furthermore, we plan on exploring the use of Sieve for resource discovery, event transformations, and event compositions. We believe that the distributed pipelining architecture of Sieve can serve as the basis for a unified content-based information dissemination network.

References

[1] A. Carzaniga and A. Wolf, "Forwarding in a content-based network," in *Proc. of ACM SIGCOMM*, Aug 2003.

[2] N. P.-M. Volume, "http://dynamic.nasdaq.com/ dynamic/premarket5dayvolume.stm."

[3] A. Carzaniga, M. Rutherford, and A. Wolf, "A routing scheme for content-based networking," in *Proceedings of IEEE INFOCOM*, Mar 2004.

[4] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra, "Matching events in a content-based subscription system," in *Symposium on Principles of Distributed Computing*, 1999.

[5] Y. Wang, L. Qiu, C. Verbowski, D. Achlioptas, G. Das, and P. Larson, "Summary-based routing for content-based event distribution networks," *Computer Communication Review*, 2004.

[6] T. Yan and H. Garcia-Molina, "Index structures for selective dissemination of information under the boolean model," in *ACM Transactions on Database Systems*, 1994.

[7] A. Snoeren, K. Conley, , and D. K. Gifford, "Mesh based content routing using xml," in *Proc. of SOSP*, 2001.

[8] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec, "The many faces of publish/subscribe," in *Tech. Rep. DSC ID:2001*, 2001.

[9] G. Varghese and A. Lauck, "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 824–834, 1997.

[10] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984.

[11] M. Adler, Z. Ge, J. Kurose, D. Towsley, and S. Zabele, "Channelization problem in large scale data dissemination," in *ICNP*, 2001.

[12] A. Riabov, Z. Liu, J. Wolf, P. Yu, and L. Zhang, "Clustering algorithms for content-based publication-subscription systems," in *Proc. of ICDCS*, 2002.

[13] F. Cao and J. Singh, "Efficient event routing in content-based publish-subscribe service networks," in *Proceedings of Infocom*, Apr 2004.

[14] O. Papaemmanouil and U. Cetintemel, "Semcast: Semantic multicast for content-based data dissemination," in *Proceedings of ICDE*, Apr 2005.

[15] Y. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. Wang, "Subscription partitioning and routing in content-based publish/subscribe networks," in *Proc. International Symposium on Distributed Computing*, 2002.

[16] M. Oliveira, J. Crowcroft, and C. Diot, "Router level filtering on receiver interest delivery," in *Proc. of 2nd Int. Workshop on Networked Group Communication*, 2000.

[17] R. Shah, R. Jain, and F. Anjum, "Efficient dissemination of personalized information using content-based multicast," in *Proc. of Infocom*, 2002.

[18] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *ACM SIGMOD*, 2001.

[19] P. Eugster, P. Felber, R. Guerraoui, and S. Handurukande, "Event systems: How to have your cake and eat it too," in *Proc. of DEBS*, 2002.

[20] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith, "Efficient filtering in publish-subscribe systems using binary decision," in *International Conference on Software Engineering*, 2001.

[21] B. Segall and D. Arnold, "Elvin has left the building: A publish/ subscribe notification service with quenching," in *AUUG*, 1997.

[22] P. J. Z. Ge, J. Kurose, and D. Towsley, "Min-cost matchmaker problem in distributed publish/subscribe infrastructures," in *OPENSIG*, 2002.

[23] P. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *DEBS workshop on Distributed event-based systems*, 2002.

[24] H. Yu, D. Estrin, and R. Govindan, "A hierarchical proxy architecture for internet-scale event services," in *Proc. of WETICE '99*, 1999.

[25] G. Cugola, E. Nitto, and A. Fugetta, "The jedi eventbased infrastructure and its application to the development of the opss wfms," in *IEEE Transactions on Software Engineering*, 2001.

[26] P. Pietzuch and J. Bacon, "Peer-to-peer overlay broker networks in an event-based middleware," in *DEBS workshop on Distributed event-based systems*, 2003.

[27] L. Cabrera, M. Jones, and M. Theimer, "Herald: Achieving a global event notification service," in *Proc. of HotOS-VIII*, 2001.

[28] G. Fox and S. Pallickara, "The narada event brokering system: Overview and extensions," in *Conference on Parallel and Distributed Processing Techniques and Applications*, 2002.

[29] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *ACM SIGCOMM*, 2002.

[30] R. Chand and P. Felber, "A scalable protocol for content-based routing in overlay networks," in *IEEE Symposium on Network Computing and Applications*, 2003.

[31] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," in *IEEE Journal on Selected Areas in communications*, 2002.

[32] Y. Diao, S. Rizvi, and M. Franklin, "Towards an internet-scale xml dissemination service," in *VLDB*, 2004.

[33] A. Gupta and D. Suciu, "Streaming processing of xpath queries with predicates," in *SIGMOD*, 2003.