On Efficient Content Matching in Distributed Pub/Sub Systems

Weixiong Rao* Lei Chen♯ Ada Wai-Chee Fu*
*Department of Computer Science and Engineering
The Chinese University of Hong Kong
{wxrao,adafu}@cse.cuhk.edu.hk
♯Department of Computing Science
Hong Kong University of Science and Technology
{leichen}@cse.ust.hk

***Abstract*—The efficiency of matching structures is the key issue for content publish/subscribe systems. In this paper, we propose an efficient matching tree structure, named COBASTREE, for a distributed environment. Particularly, we model a predicate in each subscription filter as an interval and published content value as a data point. The COBASTREE is designed to index all subscription intervals and a matching algorithm is proposed to match the data points to these indexed intervals. Through a set of techniques including selective multicast by bounding intervals, cost model-based interval division, and COBASTREE merging, COBASTREE can match the published contents against subscription filters with a high efficiency. We call the whole framework including COBASTREE and the associated techniques as COBAS. The performance evaluation in simulation environment and PlanetLab environment shows COBAS significantly outperforms two counterparts with low cost and fast forwarding.**

I. INTRODUCTION

Recently, Content based Publish/Subcribe System (CBPS) has emerged as a new paradigm to provide fine-grained content publishing and filtering services [5], [18]. In CBPS, consumers (subscribers) declare their interested contents through "subscriptions", producers (publishers) publish content messages, and pub/sub servers (brokers) match the subscriptions with content messages and deliver the subscribers with the expected documents. Usually in CBPS, contents are structured by pairs of <attribute, value> [9], [5]. The subscribers can specify selective predicates over content attributes as subscriptions, which are also called *filters*.

Due to the geographic distribution of publishers and subscribers, CBPS is often applied in a distributed environment, some examples are distributed multiplayer game [3], and system monitoring [5] etc. A distributed CBPS system usually consists of a set of pub/sub servers that manage filters and forward content messages to interested subscribers. Thus, the key issue of a distributed CBPS system is how to design an efficient matching structure to organize filters stored in distributed pub/sub servers. A well designed matching structure can efficiently forward content messages to relevant filters; and meanwhile, filters themselves can be effectively propagated within the matching structure.

To design an efficient matching structure for the distributed CBPS, two different approaches have been attempted:

i) *Multiple single-dimensional indexes*: Suppose there are d attributes in the content schema, d single-dimensional indexes are created. With d single-dimensional indexes, each publication creates d content copies to reach all indexes. Obviously, the high dimensionality value of d directly indicates *massive message traffic volume*. In addition, if a filter with predicates over multiple attributes is distributed in multiple single-dimensional indexes, the counting algorithm in [10] has to wait for the arrivals of all matching result, resulting in *high filter matching latency*.

ii) A *single multi-dimensional index* with dimensionality of d: this approach suffers from the problem of *dimensionality curse* due to the high dimensionality of d, i.e. the lookup performance over the multi-dimensional index steeply degrades with the dimensionality. Correspondingly, during the content matching, the high dimensionality in the single multi-dimensional index produces the *massive message traffic volume* and *high filter matching latency*. Moreover, the number of attributes that appear in the filters is usually much less than that in the contents, we call this phenomenon "*dimensionality mismatch*". In order to build a full-space indexing structure, the whole domain ranges of the missing attributes in the filter predicate are used which results in elongated boundaries and less efficient in the later on content matching.

In this paper, we model each predicate in a subscription filter as an interval, and content value as a data point. We arrange the intervals into a data structure called the COBASTREE. In order to handle the overloading problem and single point of failure in distributed CBPS system, we avoid forwarding all published messages towards the root of the COBASTREE. Instead, we propose a bottom-up matching algorithm and a leaf node in the tree is a start point to process the content matching. We call a node in the tree a logical node (*lnode*), in contrast to the physical node (*pnode*) which may store the contents of multiple *lnode*s.

We propose three techniques to overcome the challenges in CBPS (i.e. massive message traffic volume and high filter matching latency):

1) Instead of a tree structure for each dimension, or a single multi-dimensional tree, we propose to use a limited number of trees, each covering a subset of the dimensions. To this end, single dimension COBASTREES are *adaptively merged* and the total number of COBASTREES is decided by an integrated cost model (which incorporates the forwarding cost and filter maintenance cost).

2) We propose to replace single "long" interval by multiple "short" segments. The purpose of *interval division* is to solve

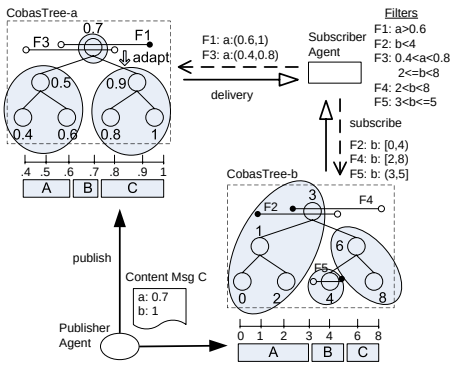


Fig. 1. Basic COBAS Framework: two COBASTREEs respectively for attribute a and b composed by 5 filters, and three pub/sub servers A, B and C.

the bottleneck problem caused by long intervals since they match most published content.

3) We propose a technique of *selective multicast*. A published content is multi-casted from the leaf node to and only to the non-leaf nodes containing filters that are interested in the publication.

We call the proposed distributed CBPS framework COBAS (abbreviation for Content based pub/sub system). The performance of COBAS is evaluated in a Peer-to-Peer (P2) overlay network, using the open source code of Mercury [2] for implementation. The experimental results based on the simulation environment and PlanetLab show that COBAS can significantly reduce the network traffic volume with low latency.

The rest of this paper is organized as follows: Section II shows the basic frame work of COBAS; as the principle of from easy to complex, we first describes selective multicast (Section III), then interval division (Section IV), and finally adaptive merging (Section V). Section VI summaries related works, and Section VII evaluates COBAS and two counterparts. Finally, Section VIII concludes this paper.

II. BASIC FRAMEWORK

A. Data Model

Given a data schema $M = \{A_1, ..., A_d\}$, we model the predicate condition of attribute A_i, $l_i \leq A_i \leq u_i$, as an *interval* $[l_i, u_i]$ over A_i. For example in Figure 1 where the domain range of attribute a is $[0, 1)$, the predicate condition in F_3 like $0.4 < a < 0.8$ is treated as an interval $a : (0.4, 0.8)$; the half coverage like $a > 0.6$ in F_1 as an interval $a : (0.6, 1.0)$. For the equality condition like $a = 0.5$, we can treat it as $a : [0.5, 0.5]$ where the lower bound is equal to the upper bound.

Similarly, we can also model the content value over attribute A_i as a *point*. For example, $C = \{\langle a, 0.7\rangle, \langle b, 1\rangle\}$, can be treated as $a : (0.7, 0.7)$ and $b : (1, 1)$, which can be intuitively seen as a point in 4-dimensional space. Given the above model, matching attribute value $\langle A_i, v_i\rangle$ against predicates $l_i \leq A_i \leq u_i$ can be seen as a stabbing query of point (v_i, v_i) against intervals $[l_i, u_i]$.

B. Basic COBASTREE

After the predicates of subscription filters are modeled as intervals, we organize all subscription intervals of attribute A_i into a logical tree of attribute A_i, and call it the COBASTREE of A_i. The COBASTREE is used to index the subscription filters for efficient content matching.

The COBASTREE is primarily a balanced binary tree, where each node is labeled by a unique *key value*, and each interval is stored in exactly one *lnode* with a key value that is contained in the interval. An interval $[l, u]$ is kept at the *highest lnode* it *covers*, i.e. the highest *lnode* with key value w for which $l \leq w \leq u$ holds when descending the tree from the root. In COBASTREE-b of Figure 1, though the key value of 4 is inside both interval $(3, 5]$ and $[2, 8)$, interval $[2, 8)$ is kept only at the higher level node with key 3. Also any interval registered to the node with key 4 must have a upper bound less than 6 and lower bound larger than (or equal to) 3; otherwise the interval should reside at a higher node. Each of the logic nodes (*lnode*) w is associated with *two lists* $L(w)$ and $U(w)$. $L(w)$ and $U(w)$ contain, respectively, sorted lists of the lower and upper bounds of the intervals stored at w.

For a COBASTREE with height H, each leaf node records H "*bounding intervals*" with one for each *lnode* along the bottom-up path from the leaf *lnode* to the root: among all upper bound values (i.e. $U(w)$) in each *lnode*, we record the maximal upper bound value, denoted as u^b; similarly the minimal lower bound value in $L(w)$, denoted as l^b. For u^b and l^b of each *lnode*, we may treat $[l^b, u^b]$ as the *bounding interval* of such a *lnode*. For *lnode* 3 in COBASTREE-b of Figure 1, there are two upper bounds: 4 and 8 and two lower bounds: 0 and 2; then u^b is 8 and l^b is 0. So *lnode* 3 has the bounding interval $[0, 8)$. Note that bounding interval $[l^b, u^b]$ is not necessarily a real interval stored in the *lnode*. In COBASTREE-b of Figure 1, no interval $[0, 8)$ is really stored in *lnode* 3. The bounding intervals can help the content matching, which will be shown later in Section III.

C. System Overview

In COBAS, each pub/sub server (i.e. the physical node, *pnode*) maintains *contiguous* data ranges for d attributes (dimensions) and all *pnode*s are connected as a P2P overlay network. In our implementation, Mercury [2] is used as the overlay network with $O(\log N)$ lookup complexity where N is the number of *pnode*s. A *pnode* responsible for some data range R_i *stores* all *lnode*s with key values inside R_i, and *resolves* the matching of content C with attribute value v_i inside R_i. In Figure 1 with two attributes a and b in schema M and three *pnode*s A, B and C: *pnode* A maintains data ranges $a : [0.4, 0.6], b : [0, 3]$, *pnode* B data ranges $a : (0.6, 0.7], b : (3, 5]$, and *pnode* C data ranges $a : (0.7, 0.1], b : (5, 8]$.

As a logical structure, the COBASTREE is physically distributed across the pub/sub overlay network. In Figure 1, for the COBASTREE of attribute b, the subtree of 4 nodes with key 0,1,2 and 3 is stored in *pnode* A, the node with key 4 in *pnode* B and the subtree of 2 nodes with key 6 and 8 in *pnode* C. Similar situation holds for the COBASTREE of attribute a. By this semantical locality property, the content matching in the whole COBASTREE may save on communication cost across the underlying *pnode*s. For example, for any content value

$v_b = 0$, all relevant filters are certainly located in the path from the *lnode* with key 3 to the leaf *lnode* with key 0 in COBASTREE-b, and no communication cost is consumed to traverse this path since all *lnode* are stored in *pnode* A.

D. Subscription

In COBAS, the registration of filter F is the process to insert predicate intervals in F to COBASTREEs. For F with predicates over only one attribute A_i, a subscriber can contact a $pnode$ as Subscriber Agent (SA) to send the subscription to the COBASTREE of A_i; after some *lnode* is found to insert F, F is physically stored in some *pnode* which stores the found *lnode*. When some content C matches F, this *pnode* delivers content C to SA, then the subscriber receives the subscribed contents via SA.

For a filter F with predicates over *multiple* attributes, we register F in *one* chosen COBASTREE. Here we do not registering F to d COBASTREEs to avoid sending duplicates to Subscriber Agent(SA). In Figure 1, F_3 with attributes a and b is inserted to only one COBASTREE: COBASTREE-a. In the implementation for simplicity, we use the random policy to chose one among total d COBASTREEs.

E. Publication

After injecting C to Cobas via some speical *pnode*, named publisher agent (PA), C will be matched and forwarded inside d COBASTREEs by the following steps:

1). With the underlying pub/sub overlay network, content C is routed to *pnode*s with data ranges R_i covering v_i where $i = 1, ..., d$. For simplicity, this step is called content routing in pub/sub overlay network;

2). By means of the COBASTREE, content C is forwarded to all *lnode*s that store the intervals covering $\langle A_i, v_i \rangle$. This step is called content forwarding in COBASTREEs;

3). The forwarded content C will locally match each retrieved filter F in last step (i.e. predicates of F except A_i); whenever C matches a filter F, C will be delivered to the subscriber agent(SA) of F.

Each content is associated with d pairs of $\langle A_i, v_i \rangle$, d copies of content C are routed to d *pnode*s. For example in Figure 1, two copies of content C are routed to *pnode* A and C because the data range of attribute a responsible by *pnode* C, $a : [0.8, 0.1]$, covers the value of attribute a in content C, 0.7; the data range of attribute b maintained by *pnode* A, $b : [0, 3]$, covers the value of attribute b in content C, 1.

The Local_Matching, i.e. the fourth step of publication, is necessary. For example in Figure 1, for F_3 in the COBASTREE of attribute a, though C satisfies predicate $0.4 < a < 0.8$, we must determine whether content C can match predicate $2 < b < 8$. In this paper, the classical algorithms in [9] are used for local matching.

From the publication steps, the publication performance depends on two factors: i) the communication traffics consumed by the first two steps of content publication (content routing and content forwarding); ii) the workload to locally process the matching between the contents and filters by the last two steps of content publication. This paper focuses on how to reduce the communication costs; it is important to reduce the processing workload of the local matching by fewer contents forwarded to the *pnodes*. The following sections optimize the messaging cost for forwarding and routing content messages.

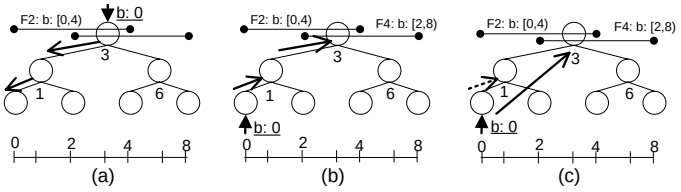


Fig. 2. Content Forwarding: (a)top-down (b)bottom-up (c)selective multicast

III. SELECTIVE MULTICAST

In this section, we consider content forwarding in the COBASTREE, which is the second step in publication.

A. Basic Content Forwarding

When content C with d pairs of $\langle A_i, v_i \rangle$ is forwarded to each COBASTREE of A_i, one may follow a top-down traversal from the root to the leaf nodes along the COBASTREE (see Figure 2 (a)). However, this overloads the *pnode* responsible for the root. To solve this problem, we consider a bottom-up approach traversing from a leaf towards the root as illustrated by Figure 2 (b) to forward a content with an attribute value $b := 0$.

The bottom-up forwarding need use FIND_LEAF(v) to find some leaf *lnode* which has the closet key value to v. FIND_LEAF(v) can be implemented by the basic lookup algorithm provided by the underlying pub/sub overlay network to find some *pnode* with the closest key vale to v. In our implementation of COBAS, the underlying overlay network Mercury [2] provides $O(\log N)$ lookup hops for FIND_LEAF(v).

B. Selective Multicast

For a COBASTREE with height H, the sequential traversal from the leaf *lnode* to the root produces $O(H)$ latency and H units of network traffic volume. In Figure 2(b), though there is no registered filter in *lnode* 1, the sequential approach still visits both *lnode* 1 and *lnode* 3, wasting 1 unit of traffic volume. Moreover, if *lnode* 3 does not contain F_4, the forwarding to *lnode* 3 is also unnecessary.

Here the "*bounding interval*" that is recorded for each leaf *lnode* (See Section II-B) becomes useful for avoiding the unnecessary messaging along the bottom-up path. We can multicast content C with attribute value v_i to and only to *lnode*s with *bounding interval* $[l^b, u^b]$ covering v_i. The key point is: *lnode*s with *bounding interval* covering some content value v_i, i.e. $l^b \leq v_i \leq u^b$, *must* contain a predicate interval covering the required value v_i. This property is given by Lemma below. Thus, forwarding contents to these *lnode*s will not generate unnecessary communication cost. We call this *selective multicast*

Lemma 1: For some *lnode* $\mathcal{N}$ in COBASTREE of A_i with *bounding interval* $[l^b, u^b]$, if attribute value v_i in content C is

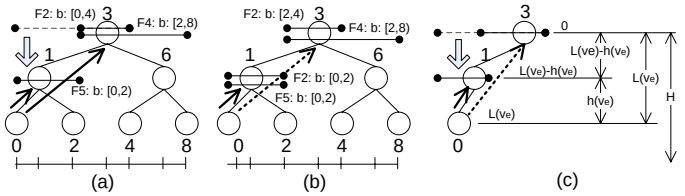


Fig. 3. Interval Division: (a) before dividing with two units of messages; (b) after dividing with only one unit of message; (c) setting up the level of v_e.

inside *bounding interval* $[l^b, u^b]$, *lnode* $\mathcal{N}$ certainly contains subscription filters f with predicate intervals covering the attribute value v_i. □

For forwarding latency, selective multicast in Figure 2(c) forwards content from leaf *lnode* 4 to *lnode* 3 with only 1 hop. For forwarding messaging cost, the traffic volume of selective multicast is decided by the number of *bounding interval*s covering the content value v_i, bounded by the height H of the COBASTREE.

IV. INTERVAL DIVISION

When H *bounding interval*s of *lnode*s along the bottom-up path cover v_i, all these *lnode*s receive the forwarded contents by selective multicast and produce H units of content copies. By interval division, the forwarding cost of selective multicast can be further reduced.

A. Basic Idea

We illustrate the basic idea in Figure 3. In Figure 3(a), by selective multicast, a stabbing query for $b := 0$ reaches three leaf *lnode*s respectively with key 0, 1 and 3. In Figure 3(b) $F_2 = [0, 4)$ is divided into two segments $[0, 2)$ and $[2, 4)$. The new segment $[0, 2)$ is reinserted to *lnode* 1. Now selective multicast only forwards contents from leaf node 0 to *lnode* 1. Note that filter F_2 is maintained in two *lnode*s: *lnode* 1 for segment $[0, 2)$ and *lnode* 3 for segment $[2, 4)$, resulting in more maintenance cost.

We can continue the above step to divide the interval $[0, 2)$ in *lnode* 1 to two segments $[0, 1)$ and $[1, 2)$, and the new segment $[0, 1)$ will be reinserted to *lnode* 0. When content having attribute value $b = 0$ reaches *lnode* 0, no multicast is required since LOCAL_MATCH can directly be used to match the content against segment $[0, 1)$ in *lnode* 0. Compared with Figure 3(b), the benefits is no communication traffic, but more maintenance cost is needed to maintain filter F_2 in three *lnode*s and to maintain F_5 in 2 *lnode*s.

We call the operation to divide one long interval to two shorter segments and to register one segment with a child *lnodes* with the remaining segment still in the original *lnode* a "*push-down*" operation.

In addition, so far we have not considered the problem of long intervals in subscriptions. A long interval tends to reside at a higher level *lnode* in COBASTREE and it attracts intensive querying since a lot of stabbing queries will stab the interval. This result in unbalanced workload and overloaded nodes. Interval division by push-down operation can also help to distribute the workload that is generated by long intervals.

B. Cost Model

Interval divisions helps to save on the forwarding cost and to avoid the overloading problem; however, there is a tradeoff in the maintenance cost. Here we introduce a cost model to find a good balance between the two odds.

We consider two main costs in our cost model: the content forwarding cost $Cost(C)$ and the filter maintenance cost $Cost(F)$. Interval division reduces $Cost(C)$ as shown in Section IV-A, but $Cost(F)$ is increased because the segments divided from the interval are registered to more *lnode*s in COBASTREE. Though the filter maintenance cost $Cost(F)$ includes the increased storage cost and the communication cost, we mainly consider the communication cost between subscriber agent (SA) and the *pnode* responsible for filters as $Cost(F)$, because memory is relatively cheap nowadays. In COBAS, the tradeoff between $Cost(C)$ and $Cost(F)$ will determine how predicate intervals are divided.

The frequencies of attribute values may be skewed, while some values appear in many contents, others rarely. We define the ***content density*** of v_e, denoted as γ_e, as the average number of published contents having value v_e. In addition, ***content popularity*** of v_e in the subscriber side, denoted by η_e, is defined as the total number of intervals covering v_e is η_e.

Our optimal solution to minimize the overall cost $Cost(A) = Cost(C)+Cost(F)$ for COBASTREE of A_i shows that:

- When v_e is frequent published (i.e. γ_e is large), intervals covering v_e should be "pushed-down" to *lnodes* at a *lower* level, to reduce $Cost(C)$, albeit having a higher $Cost(F)$.
- When v_e are *popular* subscribed so that many filter intervals covers v_e (i.e. η_e is large), without any optimization policy, dividing these intervals covering v_e may produce more smaller segments and increase the filter maintenance cost.
- To minimize the total cost $Cost(A)$, optimally COBAS favors the interval division in order to control $Cost(F)$ while trading off a higher $Cost(C)$.

V. MAINTAINING COBASTREES

This section describes a special tree merging mechanism to reduce the the messaging costs for content routing processing, that is the first step of content publication (See Section II-E).

Until now, in COBAS we assume there are d COBASTREEs, one for each attribute A_i. High dimensionality directly indicates more content copies, which significantly increases the network traffics. To reduce the above content copies in COBAS, building a multidimensional (MD) COBASTREE seems a good idea. However, a high dimensional COBASTREE may suffer from the well known problem of *dimensionality curse*. Instead, we propose the merge algorithm as follows.

Suppose COBASTREE of A_j is merged to the COBASTREE of A_i where $i \neq j$. Among all filters in the COBASTREE of A_j, some filter F_{ij} may contain predicate intervals over both A_i and A_j, then F_{ij} is inserted to the COBASTREE of A_i.

For a filter F_{kj} in COBASTREE of A_j which does not contain predicates over attribute A_i, it may contain a third

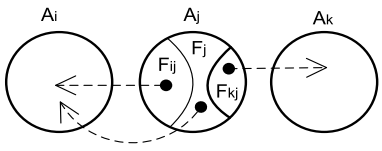


Fig. 4. Merging COBASTREEs

attributes A_k, then F_{kj} may be merged to the COBASTREE of A_k, instead of being merged to the COBASTREE of A_i, as shown in Figure 4. Only for some filter f_j containing a single attribute A_j would we consider using domain range $[L_i, H_i]$ as the predicate interval to merge f_j to the COBASTREE of A_i. After the merging, the original COBASTREE for A_j will be eliminated. This merging can be repeated for other COBASTREEs so that d' COBASTREEs remain in COBAS.

The goal of merging COBASTREEs are twofold: (1) the overall cost (including $Cost(A) = Cost(C) + Cost(F)$) after merging should be less than that before merging; (2) among all possible merging combination candidates, the one with the minimal overall cost should be chosen to form the final merging structure.

Since the cost model in Section IV-B is for a single COBASTREE, the cost model for the whole COBAS system is the aggregation result of each COASTREE. Obviously, the higher dimensionality d means the larger aggregation value for $Cost(C)$. We denote the aggregation result of the whole Cobas system as $Cost_w(A)$=$Cost_w(C)$ + $Cost_w(F)$ where $Cost_w(C)$ and $Cost_w(F)$ respectively is the aggregation of $Cost(C)$ and $Cost(F)$ of each COBASTREE in the whole COBAS system. And After merging, the cost of COBAS, denoted as $Cost'_w(A)$, will still be the aggregation result of each remaining COBASTREE. By merging we expect a larger positive value of $\Delta Cost_w(A) = (Cost_w(A) - Cost'_w(A)) > 0$ which means more gain of merging.

An exhaustive algorithm may find the minimal $\Delta Cost_w(A)$ among all possible merging instances, with exponential complexity. Instead, we use a greedy algorithm with a quadratic complexity to profit $\Delta Cost_w(A)$. □

After merging COBASTREEs, "domain range" like long intervals used to rebuild filters with single attribute(like F_j in Figure 4) are divided as described in Section IV to ensure the best result. Our experiment in Section VII shows that only merging COBASTREEs without interval division will produce more network traffic volume than COBAS. Due to the dynamic patterns of contents and filter, COBAS may periodically run the greedy algorithm to adaptively merge COBASTREES to achieve better performance.

VI. RELATED WORK

[2], [10], [19], [1] are the examples of *multiple single-dimensional indexing structures*. [2] creates one P2P ring overlay structure for each attributes and build the pub/sub service over the overlay network. [10] creates one filtering and forwarding tree for each attribute. Two problems with this approach are that the number of content message copies for each publication is equal to the dimensionality, which is typically high, and the response latency caused by asynchronously counting the intermediate matching results from multiple single-dimensional indexes. In Ferry [19], the home node for each attribute, as the rendezvous node for such attribute, may suffer from overloading and single point of failure issue. Also, for each attribute, PastryStrings [1] builds the prefix based string tree over Pastry [15].

[16], [11] are examples of using *a single multi-dimensional indexing structure*. After mapping content values and predicate conditions into $2d$-dimensional points, [16] utilizes the R-Tree structure to index the predicate conditions and the content matching is transferred to the R-Tree query processing; [11] utilizes the underlying P2P overlay network CAN [14] to reorganizes the filters in the d-dimensional Cartesian coordinate space.

[5], [4] focus on the generic graph-like network. Without particularly building the distributed matching structure, each pub/sub servers maintains the forwarding states about all subscriptions over multiple attributes, and the subscriptions are merged to reduce the maintained forwarding states. The "general" graph network favors the more reliability; however, a content message may be forwarded to each intermediate node along the forwarding path to the another endpoint, creating a high message traffic volume. Also instead of a tree structure, Sub-2-Sub [17] builds the unstructured overlay network for content pub/sub by an epidemic based algorithm; in particular, also modeled as intervals, the subscription predicates are clustered which is further used for publication.

For indexing structures, the main-memory based interval tree [8] and the segment tree [7] are both geometric data structures to index intervals. Both allow efficient retrieval of intervals covering a query point, however, they are designed for centralized control, therefore a search always begins at the root, making it a critical point that is not suitable for a distributed environment. The interval tree allows for $O(N)$ storage space given N intervals, compared to $O(NlogN)$ for segment tree. In the interval tree, long intervals are not broken up; since they can satisfy most querying, in a distributed publish/subscribe environment, a physical node that contains long intervals will be overloaded. Compared with interval tree and segment tree, the significant novelty of COBASTREE is the mechanism of interval division for the efficient content matching. Moreover, the bottom-up select multicast and adaptive merging algorithm are also unique in the COBASTREE.

VII. PERFORMANCE EVALUATION

A. Experimental Setup

Prototype Implementation of COBAS and Counterparts: To illustrate how COBAS works as a distributed CBPS system, we need to chose an overlay network to connect *pnode*s. For a given value v_i, such an overlay network is expected to lookup a *pnode* with local data range R_i containing v_i. The lookup can support the function FIND_LEAF that is used by COBAS.

There are many works like [6], [2] to connect distributed nodes as a Peer-to-Peer (P2P) overlay network. During our implementation, we chose Mercury[2] as the underlying pub/sub

overlay network due to its efficient lookup (comparable to other methods) and simple circular structure. In fact, other P2P networks supporting range query functions can also be used for COBAS.

Mercury handles multi-attribute queries by creating a routing *hub* for each attribute in the application schema. A logical collection of servers in the system are organized to a circular overlay, and data are placed contiguously on this ring, with each server responsible for a non-overlapping data range for the particular attribute. When each server maintains k links to other servers, the routing algorithm supports range-based lookups within each routing hub in $\mathcal{O}(\log^2 N/k)$ hops. To balance the load, random sampling is used to estimate the load distribution across the whole system.

To support COBAS, we extend the basic lookup algorithm in Mercury to find some data with the closest (nearest) value to some given value v_i, with $\mathcal{O}(\log N)$ hops by setting $k = \log N$ links per server. The random sampling algorithm of Mercury is used by COBAS to collect statistics information for the periodical optimizations of interval division and COBASTREE merging. To build the COBASTREE structure over Mercury, *lnode*s refer to the address information of the parent and child *lnode*s, if any. Besides, the level information of *lnode*s are periodically propagated across COBASTREE, and COBASTREE rotation is lazily balanced when the root of COBASTREE detects the level of either subtree is smaller than half of the level of another subtree.

Running Environments: We implement COBAS over the open source code of Mercury (version 0.9.2: http://www.cs.cmu.edu/∼ashu/research.html). In the open source code of Mercury, two running modes are supported by the same event driven scheduler: simulation mode and real communication mode. We use both modes to evaluate the performance of COBAS in two environments, respectively:

- *Simulation mode* by a single Linux machine with the 4-way 3.2GHz CPU machine with 2G RAM. We setup the number of *pnode*s that participate the pub/sub overlay network from 10^3 to $8 * 10^4$; all *pnode*s are logically organized to the circular overlay structure as Mercury [2]. Each *pnode* acts as the publisher agent (PA) to inject content messages to COBAS and total 10^4 content messages per minute are published with each content message carrying a payload of 256kb. Each *pnode* also acts as subscriber agent (SA) to register the generated filters in COBAS. The connection between SA and the *pnode* storing subscription filters (either original intervals or divided segments) is maintained by the one heartbeat message of size 128 bytes per 10 minutes.
- *Real mode* by PlanetLab (http://www.planet-lab.org). In PlanetLab, we deploy the COBAS prototype to around 70 physically distributed machines as *pnode*s of COBAS. Around 120 content messages (with 10 attributes) are published per minute, and total 10^5 filters are registered. Due to he limited bandwidth and low capacity of physical machines in PlanetLab, we set content arriving rate as 120 contents per minute and filter number as 10^5. The COBAS prototype has run in PlanetLab for 24 hours.

Counterparts: For comparison, we use two counterparts:

1) We extend Mercury pub/sub application [3] as the example of *multiple one dimensional structures* by *Counting* algorithm for content matching. For Counting approach, we forward published contents to each attribute and then count the matching result in the subscriber machine. Since [3] is the pub/sub application in Mercury, we directly use the pub/sub example application provided in Mercury open source code. For simplicity, we abbreviate this approach as "*multi-1D*".
2) R-Tree approach [16] as the example of *one multidimensional structure*. During the implementation of R-Tree approach, we first preprocess all subscription filters into the R-Tree structure in the full d dimensional space and use space-filling curve (Hilbert curve) to transform the multidimensional R-Tree to the single dimensional tree structure, which can be easily implemented in Mercury [2]. This kind of RTree implemented in the distributed environment (i.e. Mercury) is abbreviated as "*DRtree*".

With consideration of the skew filter popularity and content density, we use Mercury load balancing utility to adjust the data range of *pnode* to balance the (storage space and matching processing) load in each *pnode*. In addition, in two experiments (Figure 6(c)), we replicate the filter copies to multiple *pnode*s. According to the proportional replication principle [12], [13], the filters that covers more content values are replicated to more *pnode*s.

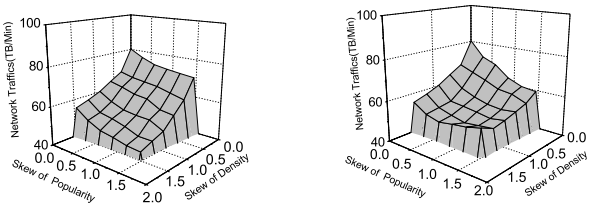
Data Set: we use Zipf distribution to generate both the predicate intervals and attribute values. For simplicity, all data types in schema M are *doubles* draw within domain range [0,1]. We produce the attribute values by a Zipf distribution within the domain range. For predicate intervals I, first we draw a number from the Zipf distribution as the interval median I_m; the half of the interval length,denoted as $l_{1/2}$, is drawn from range [0,I_m] if $I_m < 1 - I_m$; otherwise from range [0,$1 - I_m$]. The Zipf distribution of attribute values and the predicate intervals can respectively indicate the skew of *content density* and *filter popularity*. Given data scheme M with d attributes $A_1, ..., A_d$, we follow the Zipf distribution to pick $|A_f|$ attributes among $A_1, ..., A_d$ used as predicate attributes, where $|A_f| \in [1, d]$. The dimensionality of the produced contents is d. The related parameters including allowable range, probability distribution and default value can refer to Table I. Without special mention, all parameters use default values.

Evaluation Metrics: We mainly focus on the performance evaluation for:

- *the network traffic volume*: message volume consumed to forward content messages from the publisher machine to the subscriber agent machine, i.e. $Cost(C)$, and to exchange the heartbeat message between the subscriber agent maintenance and the machines responsible for the subscription filters, i.e. $Cost(F)$.
- *the content processing time*: the period between the mo-

Parameter	Range	Distribution	Default
N: # of *pnode*s	S:$10^3 - 8 * 10^4$, P:70		10^4
d: dimensionality	1-20	Constant	10
n: # of filters	S: 10^{5-8}, P: 10^5	Constant	10^7
I_m:interval median	[0,1.5]	Zipf, [0,1.5]	0.95
$l_{1/2}$: half interval length	[0,I_m] or [0,1-I_m]	Zipf, [0,1.5]	0.95
$\|A_f\|$: attribute # in filter f	[1,n]	Zipf, [0,1.5]	0.95
$\|C\|$: # of contents per sec	S: 10^4, P: 120	Constant	
γ_e: content density	[0,1]	Zipf, [0,1.5]	0.95
v_e: attribute value	[0,1.0]	Zipf, [0,1.5]	0.95

TABLE I
PARAMETERS USED IN EXPERIMENTS(S: SIMULATION; P: PLANETLAB)



(a) Anti-occurrence (b) Co-occurrence

Fig. 5. Effect of Content Density and Filter Popularity

ment of the content publication at publisher machine and the moment that satisfied contents arrive at the subscriber's machine.

B. Simulation Results

Effect by Content Density and Popularity: In the first experiment we study the performance of COBAS itself with the effect of several parameters: skew of *content density* (indicated by the Zipf parameter α to generate the value of γ_e), and skew of *filter popularity* (also indicated by the Zipf parameter α to generate I_m and $l_{1/2}$). We are particularly interested whether the skew of content density co-occurs with skew of filter popularity or not. The *co-occurrence* means: the attribute values, frequently published, are simultaneously covered (or subscribed) by the a large number of predicate intervals, and vice versa. The *anti-occurrence* means the frequently produced values are not necessarily covered (or subscribed) by many predicate intervals.

Figure 5(a) and (b) respectively plot the network traffics for both cases where the experiment is conducted in COBAS with only one COBASTREE. In the *anti-occurrence* case, the network traffic volume is reduced with increasing skews (i.e. larger values of both Zipf parameters α): from the maximal 80.25 TB/min to the minimal 55.3 TB/min; while in the *co-occurrence* case, the minimal 48.45 TB/min is achieved when content density is $\alpha = 1.2$ and filter popularity is $\alpha = 1.0$, not the most skew situation with $\alpha = 1.5$. The reason is: in the *anti-occurrence* case, the skew of interval popularity can be divided independently on the skew of content density. However, in *co-occurrence* case, the skew of interval popularity also means the skew filter popularity; thus covering highly dense attribute values, the intervals will be divided to more small segments; simultaneously, due to the high popularity, the number of those intervals is also high, producing an extremely large number of small segments and requiring high $Cost(F)$. Consequently, the most skew of filter popularity and content density do not produce the minimal network traffics in *co-occurrence* case.

Effect by Dimensionality: In the following four simulation experiments from Figure 6 to Figure 7, we compare COBAS with two counterparts (*DRtree* and *Multi-1D*). Figure 6(a) plots the network traffic volume per minute with the varied attribute number (dimensionality). In particular, we study: COBAS with both interval division and merging COBASTREEs, in short COBAS; COBAS having interval division but without merging COBASTREEs, in short COBAS WITHOUT MERGING (there are d COBASTREEs and long intervals are divided); Cobas having merging COBASTREEs but without interval division, in short COBAS WITHOUT DIVISION (d COBASTREEs are merged but the domain range like long intervals in the root are not divided).

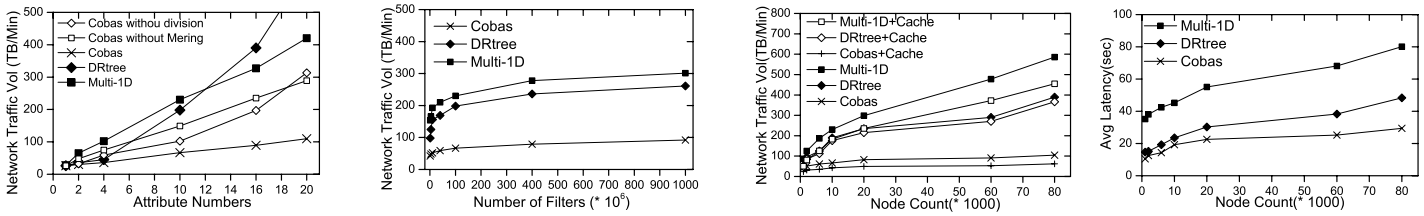
From Figure 6(a), we can find for *Multi-1D* and *DRtree* approach with a high dimensionality, the network traffic volume grows significantly. With a higher dimensionality, *Multi-1D* approach uses more content copies since each copy is forwarded for each attribute. When the dimensionality is low, *DRtree* approach does not cause high network traffic volume; but when the dimensionality is high (around larger 12), the network traffic volume of *DRtree* is larger than that of *Multi-1D* approach, due to the curse of dimensionality.

For COBAS WITHOUT MERGING and COBAS WITHOUT DIVISION, dimensionality 17 becomes the turning point: for dimensionality lower than 17, COBAS WITHOUT MERGING consumes more message cost; but for the higher dimensionality, COBAS WITHOUT DIVISION consumes more cost. This is because: merging COBASTREEs can reduce the message copy number, however, without interval division, the "domain range" like long intervals are registered in the root of the merged COBASTREE. When the dimensionality is higher, the number of these long intervals becomes more and the root of merged COBASTREE is registered with more long intervals, resulting in following outcomes:

- the load balancing utility of Mercury will adjust the data range of *pnode* and redistribute these long intervals to more *pnode*s; intuitively, these long intervals are expanded to more *pnode*s and result in the larger value of m of the content matching complexity $\mathcal{O}(\log n + m)$ in COBASTREE.
- since these (domain range like) long intervals satisfy each content value, each content value suffers from the larger value of m. As a result, when the dimensionality is higher, COBAS WITHOUT DIVISION suffers from a higher cost even than COBAS WITHOUT MERGING.

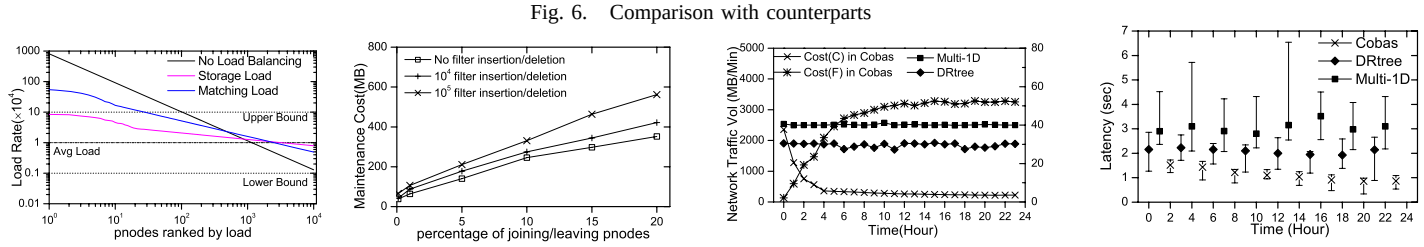
Finally, with minimal message cost, COBAS itself can benefit from the integrated solution of interval division and COBASTREE merging.

Effect by Filters: In Figure 6(b), we evaluate the network traffics with the varied number of subscription filters from 10^5 to 10^8. In general, the network traffic volumes of three approaches are increased, but with a relatively smooth growth. In COBAS the increased number of filters means the COBASTREE with a higher value of H; similar situation holds for *DRtree* approach. For *Multi-1D* approach by [3], the increased



(a) Effect by Attribute Number (b) Effect by Filter Number (c) Network Traffics by Node Count (d) Latency by Node Count

Fig. 6. Comparison with counterparts



(a) Load Distribution (b) Maintenance Cost (c) Network Traffic Volume (d) Latency

Fig. 7. Maintenance Result and PlanetLab Result

number of filters means more *pnode*s store the filters and the more forwarding cost. Among three approaches, COBAS has the least traffic cost with the increasing of subscriptions.

Effect by Nodes: In experiments in Figure 6(c) and (d), we study how the node count (i.e. the number of *pnode*s) and caching technique affect the network traffics.

Figure 7(a) plots the network traffic volume per minute when the node count is varied from 10^3 to 8×10^4. When node count becomes larger, the network traffic volumes of three approaches grow too. The more number of *pnode*s participated in the P2P network Mercury result in two outcomes:

- Mercury P2P routing algorithm (with $\log N$ hops for each lookup) will consume more lookup messages;
- the data range responsible by each *pnode* becomes smaller.

For *Multi-1D* approach, since the contents are forwarded to d COBASTREEs, the more hop number of Mercury lookup contributes the growth of content forwarding cost in the *Multi-1D* approach; in addition, the smaller data range of each *pnode* means each predicate interval will cover more data ranges of *pnode*s, resulting in each filter stored in more *pnode*s and higher filter maintenance cost.

For *DRtree*, with the smaller data range in each *pnode*, the logic bounding box in *DRtree* will cover more smaller data ranges of *pnode*s, resulting in more *pnode*s physically responsible for the logic bounding box; then each query in *DRtree* will consume more visits of *pnode*s when the bounding boxes in *DRtree* overlaps, which produces the high network traffic volume in *DRtree* approach. Among three approaches, COBAS suffers from the smallest growth when *pnode* number becomes larger. When caching is used to reduce Mercury lookup hop number, the message traffic volume caused by the P2P lookup used in each FIND_LEAF(v_i) is reduced. Furthermore, with d FIND_LEAF(v_i) operations in d Mercury hubs, *Multi-1D* approach benefits most from the reduced hop number by caching technique; with one Mercury hub, *DRtree* approach achieves the least benefits, and COBAS obtained benefits between those of *DRtree* and *Multi-1D* approach.

Figure 6(d) plots the average latency when *pnode* number grows. The *Multi-1D* algorithm suffers from the highest latency since the *Multi-1D* algorithm requires waiting for the matching result from distributed *pnode*s. The most late arriving matching result becomes the bottleneck time to evaluate the whole filter. When there are overlapped bounding boxes, *DRtree* requires exploring each possible leaf node until the real data point(the predicate interval) is found. The latency of *DRtree* approach is always related to the height of *DRtree*. Instead, the latency of COBAS is mainly decided by the routing process to FIND_LEAF(v_i); while the selective forwarding process across COBASTREE only requires 1 hop.

Maintenance Study: The final two simulations study the system maintenance aspects of COBAS. Figure 7(a) shows the load distribution in COBAS. We distinguish two types of loads: the *storage load* of a *pnode* is indicated by the rate of the filter number stored in the *pnode* over all filters in COBAS; the *matching load* of a *pnode* is indicated by the rate of incoming content number forwarded to a *pnode* over all contents. Here we consider the (either storage or matching) load within [avg_load*10,avg_load/10] as *allowable load*, otherwise the load higher than avg_load*10 means *overloading*. In Figure 7(a), without load balancing (provided by Mercury), 1.02% *pnode*s have to serve 47.12% (storage) load. After the load balancing utility is applied, for the storage load, all *pnode*s are safely inside the allowable load range; for the matching cost, around 0.27% *pnode*s are overloaded due to the dynamic publication of contents, but with only 5.25% overall load.

Figure 7(b) shows the maintenance cost of COBAS when *pnode*s dynamically join and leave simultaneously with filter insertion and deletion. The maintenance cost is measured by the consumed message traffics to re-structure and re-balance COBASTREE. By this figure we find compared with filter insertion/deletion, *pnode* join/leave may consume relatively more maintenance cost. It is not hard to explore: the *pnode* join/leave may result in the *lnode* lose and COBASTREE

restructuring; moreover, because *pnode*s are used to responsible for all CobasTrees in Cobas, the lost *lnode*s by *pnode* leave require repairing all CobasTrees. Instead, filter insertion/deletion may consume less cost.

C. PlanetLab Results

In PlanetLab, we measure the network traffic volume and latency of three approaches. Figure 7(c) plots the network traffic volume of Cobas, *DRtree* and *Multi-1D* approach, in every 1 hour to compute the average network traffic per minute. For Cobas, we specially plots the content forwarding cost $Cost(C)$, and filter maintenances cost $Cost(F)$ (for $Cost(F)$, we use the bottom-x and right-y axes; other curves use the default bottom-x and left-y axes). It is not supervised that *DRtree* and *Multi-1D* approach consume more traffics than Cobas in the final moment. For Cobas, it may be observed that: during the period from the beginning to the 4-th hour, the value of $Cost(C)$ drops quickly from the 2354.6MB/min to 361.8MB/min, due to the effect of merging CobasTrees and interval division; while the value of $Cost(F)$ grows from 2.083MB/min to 33.5MB/min. Since the overall cost $Cost(A)$ of Cobas is the sum of $Cost(C)$ and $Cost(F)$, the value of $Cost(A)$ also quickly drops from 2356.683MB/min to 395.3MB/min. In the later period from 4-th to the end of the experiment, the value of $Cost(C)$ is reduced from 361.8MB/min to 222.56MB/min with around 38.5% drop, but the value of $Cost(F)$ is increased from 33.5MB/min to 52.1MB/min. By this experiment, we find that merging CobasTree and interval division may significantly the value of $Cost(C)$ of Cobas.

Figure 7(d) shows the worst, best and average latency every three hour for three approaches. In this figure, Cobas may achieve the minimal average latency with around 1.1 seconds, and *Multi-1D* approach introduces the highest latency with 3.07 seconds. The latency of Cobas is relatively stable due to the selective multicast which is independent on other operations in Cobas like CobasTree merging or interval division. The worst case of *Multi-1D* approach is as high as 6.64 seconds because the *Multi-1D* approach waiting for the matching result from each attribute may suffer from the lowest latency in a PlanetLab like distributed system.

Discussion: 1) The range query function, random sampling, load balancing and other utilities of Mercury provide many facilities for Cobas implementation, thus we can focus on the function of Cobas as an application abover Mercury; 2) The detection of data pattern like the distribution of content density etc requires a relatively long period of the data sampling. Too frequent CobasTree merging operations produce the highly maintenance load to adjust the CobasTrees. The similar situation holds for interval division; 3) The tree structure is sensitive to the high node failure rate. The passive policy to allow the unbalanced tree structure consumes the less cost to adjust the tree shape, but more forwarding cost for such unbalanced tree.

VIII. Conclusion and Future Works

We present Cobas as a framework for efficient matching for distributed CBPS. The key component of Cobas is the novel CobasTree structure and a set of techniques like selective multicast, interval division and CobasTree merging. The experiments of Cobas in two running modes illustrate the efficiency of Cobas. In order to demonstrate the practical application of Cobas, from the semantic side, we plan to enhance the flexibility of the content filtering of Cobas by using more complicated content format and filtering mechanism; from the system side, we may consider reliable message delivery over the Cobas framework.

Acknowledgment

Funding for this work was partially provided by RGC Earmarked Research Grant of HKSAR CUHK 4118/06E, NSFC/RGC Joint Research Scheme N_HKUST602/08 and National Natural Science Foundation of China (NSFC) under Grant No. 60736013.

References

[1] I. Aekaterinidis and P. Triantafillou. Pastrystrings: A comprehensive content-based publish/subscribe dht network. In *ICDCS*, page 23, 2006.
[2] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.
[3] A. R. Bharambe, S. G. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *NETGAMES*, pages 3–9, 2002.
[4] A. Carzaniga, M. J. Rutherford, and A. L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.
[5] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.
[6] A. Datta, M. Hauswirth, R. John, R. Schmidt, and K. Aberer. Range queries in trie-structured overlays. In *Peer-to-Peer Computing*, 2005.
[7] M. de Berg, M. O. Marc van Kreveld, and O. Schwarzkopf. Computational geometry: Algorithms and applications. In *Springer-Verlag, Berlin*, 1997.
[8] H. Edelsbrunner. A new approach to rectangle intersections. In *International Journal Computational Mathematics 13*, pages 209–219 and 221–229, 1983.
[9] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.
[10] S. Ganguly, S. Bhatnagar, A. Saxena, R. Izmailov, and S. Banerjee. A fast content-based data distribution infrastructure. In *INFOCOM*, 2006.
[11] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: Content-based publish/subscribe over p2p networks. In *Middleware*, pages 254–273, 2004.
[12] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the internet. In *SIGCOMM*, 2004.
[13] W. Rao, L. Chen, A. W.-C. Fu, and Y. Bu. Optimal proactive caching in peer-to-peer network: analysis and application. In *CIKM*, pages 663–672, 2007.
[14] S. Ratnasamy, P. Francis, M. Handley, R. M. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.
[15] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.
[16] P. F. S. Bianchi, A.K. Datta and M. Gradinariu. Stabilizing dynamic r-tree-based spatial filters. In *ICDCS*, pages 447–457, 2007.
[17] S. Voulgaris, E. Riviere, A. Kermarrec, and M. van Steen. Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *IPTPS*, 2006.
[18] R. Zhang and Y. C. Hu. Hyper: A hybrid approach to efficient content-based publish/subscribe. In *ICDCS*, pages 427–436, 2005.
[19] Y. Zhu and Y. Hu. Ferry: An architecture for content-based publish/subscribe services on p2p networks. In *ICPP*, pages 427–434, 2005.