Computer Networks 54 (2010) 1739–1749

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Enabling content-based publish/subscribe services in cooperative P2P networks

Duc A. Tran *, Cuong Pham

Department of Computer Science, University of Massachusetts, 100 Morrissey Blvd, Boston, MA 02125, USA

ARTICLE INFO

Article history:
Received 5 September 2009
Received in revised form 22 December 2009
Accepted 8 February 2010
Available online 12 February 2010
Responsible Editor: Qian Zhang

Keywords:
P2P
Publish/subscribe
Prefix tree
Search

ABSTRACT

P2P is a popular networking paradigm in today's internet. As such, many research and development efforts are geared toward services that can be useful to the users of P2P networks. This paper is focused on the content-based publish/subscribe service and our problem is to devise an efficient mechanism that enables this service in any given P2P network of cooperative nodes. Most techniques require some overlay structure added on top of the network. We propose an efficient solution called PUB-2-SUB which works with any unstructured network topology. In addition, multiple independent publish/subscribe applications can run simultaneously on a single instance of PUB-2-SUB. The proposed technique is based on two key components: the virtualization component and the indexing component. The virtualization component assigns to each node a unique binary string virtual address and, accordingly, a unique zone partitioned from the universe of binary strings. The indexing component hashes queries and publications to binary strings and, based on their overlapping with the node zones, chooses subscription and notification paths appropriately and deterministically. PUB-2-SUB works best for P2P-based cooperative networks such as data grid networks and institutional collaborative networks. Our theoretical findings are complemented by a simulation-based evaluation.

1. Introduction

Search applications can be categorized into two models: request/response and publish/subscribe. The former is the traditional, where a query is submitted on demand expecting immediate results; if none exists, a response indicating so is returned. Contrarily, a query in the publish/subscribe model is submitted and stored in advance, for which the results might not yet exist but the query subscriber expects to be notified when they later become available. This model is thus suitable for search applications where queries await future information, as opposed to the traditional applications where the information to be searched must pre-exist.

This paper is focused on the publish/subscribe model and our goal is to devise a mechanism that can be integrated into a given P2P network to enable applications of this model. In particular, we are interested in distributed networks where the participating nodes are cooperative, reliable, and rather static. In these networks, such as grid computing networks and institutional communication networks, P2P can be adopted as an effective way to share resources, minimize server costs, and promote boundary-crossing collaborations [1–4]. Such cooperative P2P networks can enhance applications in various fields, including education and academia [5], science [6], health [7], and business [8], to name a few. A publish/subscribe functionality is useful for these networks. For example, a monitor in a P2P-based geographical observation network [2] will be able to subscribe a query to receive alerts of fire occurrences so that necessary rescue efforts can be dispatched quickly; or in a P2P-based scientific information sharing

* Corresponding authors. Tel./fax: +1 617 287 6452 (D.A. Tran)
E-mail addresses: duc@cs.umb.edu (D.A. Tran), cpham@cs.umb.edu (C. Pham).

1389-1286/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2010.02.003

network [5], a subscriber will be notified when new scientific information is published.

To enable publish/subscribe applications, a simple way is to broadcast a query to all the nodes in the network or to employ a centralized index of all the queries subscribed and information published [9–11]. This mechanism is neither efficient nor scalable if applied to a large-scale network.

Consequently, a number of distributed publish/subscribe mechanisms have been proposed. They follow two main approaches: structure-based or gossip-based. The first approach [12–17] requires the nodes to be organized into some overlay structure and develops publish/subscribe methods on top of it. Examples of such an overlay are those based on Distributed Hash Tables [18–21] and Skip Lists [22]. The other approach [23–25] is for unstructured networks, in which the subscriber nodes and publisher nodes find each other via exchanges of information using existing peer links, typically based on some form of randomization.

The structure-based approach is favored for its efficiency over the gossip-based approach. However, when applied to a given network, the former introduces an additional overhead to construct and maintain the required overlay structure. Also, there may be practical cases where the new links, that are part of the new structure, are not allowed due to the policy or technicality restrictions of the given network.

The gossip-based approach's advantage is its applicability to any unstructured network without an additional overlay structure. But, due to the nature of gossiping, a query or a publication of new information must populate a sufficiently large portion of the network to be able to find each other at some rendezvous node with a high probability. The costs can be expensive as a result, including the communication cost to disseminate the query or publish the new information, the storage cost to replicate the query in the network, and the computation cost to evaluate the query matching condition.

We propose PUB-2-SUB – a publish/subscribe mechanism which, like the gossip-based approach, does not change the connectivity of the given network, but is aimed at a much better performance. PUB-2-SUB allows any number of independent publish/subscribe applications to run simultaneously. It is based on two key design components: the virtualization component and the indexing component. The virtualization component assigns to each node a unique binary string called a virtual address so that the virtual addresses of all the nodes form a prefix tree. Based on this prefix tree, each node is associated with a unique zone partitioned from the universe of binary strings. The indexing component hashes queries and publications to binary strings and, based on their overlapping with the node zones, chooses subscription and notification paths appropriately and deterministically.

Because PUB-2-SUB is based on directed routing, it has the potential to be more efficient than the gossip-based approach. Our evaluation study shows that PUB-2-SUB results in lower storage and communication costs than BubbleStorm [23] – a recent gossip-based search technique. In terms of computation cost, PUB-2-SUB requires only a node to evaluate its local queries to find those matching a given information publication. The proposed technique also incurs small notification delay and is robust under network failures.

The remainder of this paper is organized as follows. We discuss the related work in Section 2. We propose the details of the PUB-2-SUB mechanism in Section 3. We present our simulation results in Section 4. We conclude the paper in Section 5.

2. Related work

Many structure-based publish/subscribe techniques designed for P2P networks [12–17] use DHT [18–21] as the underlying overlay structure. This structure is favored because of its capability to grow the network and adapt to other network dynamics such as node departures and additions. When DHT is applied to enable publish/subscribe applications, the basic approach is to design a hash function that maps a subscription and a matching publication to rendezvous nodes that are either identical or in close proximity.

For example, Scribe [12], Hermes [26] (both based on Pastry DHT [20]), and Bayeux [27] (based on Tapestry DHT [21]) use topic-based hashing, thereby a subscription and a matching publication can find each other quickly because they must belong to the same topic and thus sent to the same node. Alternatively, the authors of [13–17] propose hashing based on the actual content rather than the topic, thus providing more flexible ways to express a subscription. For example, Meghdoot [13], which employs the CAN DHT [18], imposes no restrictions over subscriptions and allows them to be specified in terms of range predicates over all attributes in a schema. If this schema contains d attributes, the network is constructed as a $2d$-dimensional CAN overlay to disseminate queries and publish information. A technique that can operate atop any DHT structure is proposed in [15].

Structures other than DHT have also been used. In Sub-2-Sub [28], an overlay is built on top of the network to form "rings" of nodes, each ring containing similar subscriptions (based on some similarity measure). When a publication of new information reaches a ring, it can visit the nodes on this ring to find all the matching queries with a high hit rate. To expedite this procedure, a number of random links (connecting each node to randomly selected neighbors) and random overlapping links (connecting each node to randomly chosen nodes that share common interests) are also maintained in the overlay. Other non-DHT structure-based techniques include [17] which uses R-tree [29], and GosSkip [30] which uses Skip Lists [22], as the structure of the overlay.

Among the techniques that do not require any structured overlay, the common approach is gossiping. In these techniques (e.g., [23–25]), a node can communicate only with its neighbors via neighbor links that are defined by the original P2P network. Because broadcasting a subscription or a publication to all the nodes is too expensive, gossip-based techniques try to be more efficient by limiting the scope of broadcasting: the goal is to reach a number of nodes, not all but large enough so that the dissemination

paths of a subscription and that of a matching publication intersect somewhere with a high probability. A recent technique demonstrative of this approach is BubbleStorm [23]. In an unstructured network of n nodes, BubbleStorm replicates each query at q nodes $(q = O(\sqrt{n}))$ and sends each publication to $d = c^2 n/q$ nodes where c is a certainty factor. The probability that there is a common rendezvous node for any (query, publication) pair is $r = 1 - \exp(-c^2)$ (e.g., c = 3 corresponds to a hit probability of r = 99.99%). Another gossip-based technique is Quasar [24] which enables group-based publish/subscribe applications especially for P2P social networks.

Our technique (PUB-2-SUB) does apply an overlay structure on the underlying network. However, the difference with the earlier structure-based techniques is that the PUB-2-SUB structure uses only existing links provided by the underlying network, not introducing any new links between the nodes. Thus, PUB-2-SUB can work with any unstructured P2P network. Compared to gossip-based techniques, PUB-2-SUB is instead based on directed routing to disseminate query and publication messages more efficiently. We compare PUB-2-SUB to BubbleStorm in Section 4.

3. The PUB-2-SUB technique

Consider a cooperative P2P network $\{S_1, S_2, \ldots, S_n\}$ that is constructed and maintained according to its built-in underlying protocols. The nodes are supposed to remain functional as much as possible although there may be failures that cannot be avoided, and whenever a new node joins or a failure occurs we assume that this network can re-organize itself. A requirement is that we cannot modify the existing connectivity of the network; all communication must be via the provided links.

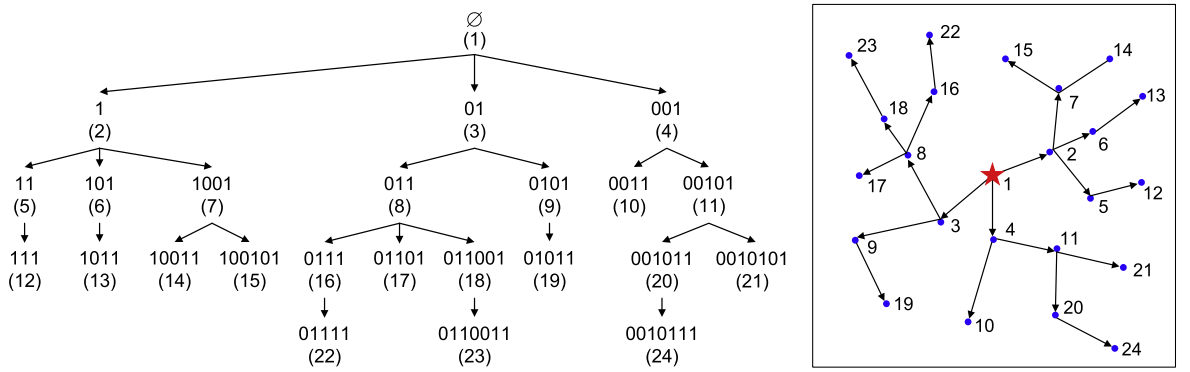
3.1. Basic idea

PUB-2-SUB is based on two key design components: the virtualization component and the indexing component. The virtualization component assigns to each node a unique virtual address. The indexing component determines the corresponding subscription and notification paths for given queries and publications, in which routing is based on the virtual addresses of the nodes.

3.1.1. Virtualization

A virtualization procedure can be initiated by any node to result in a "virtual address instance" (VA instance), where each node is assigned a virtual address (VA) being a binary string chosen from $\{0,1\}^*$. Suppose that the initiating node is S^*. In the corresponding VA instance, denoted by INSTANCE (S^*), we denote the VA of each node S_i by $\mathrm{VA}(S_i : S^*)$. To start the virtualization, node S^* assigns itself $\mathrm{VA}(S^* : S^*) = \varnothing$ and sends a message inviting its neighbor nodes to join INSTANCE (S^*). If a neighbor S_i is already part of the instance, it ignores this invitation; otherwise, by joining, S_i is called a "child" of S^* and receives from S^* a VA that is the shortest string of the form $\mathrm{VA}(S^* : S^*)+$ '0^*1' unused by any other child node of S^*. Once assigned a VA, node S_i forwards the invitation to its neighbor nodes and the same VA assignment procedure repeats. In general, the rule to compute the VA for a node S_j that accepts an invitation from node S_i is: $\mathrm{VA}(S_j : S^*)$ is always the shortest string of the form $\mathrm{VA}(S_i : S^*) + $ '0^*1' unused by any other child node currently of S_i.

Eventually, every node is assigned a VA and the VAs altogether form a prefix-tree rooted at node S^*. We call this tree a VA-tree and denote it by $TREE(S^*)$. For example, Fig. 1(a) shows the VA-tree with VAs assigned to the nodes as a result of the virtualization procedure initiated by node 1. Fig. 1(b) shows this tree as a spanning tree rooted at node 1 covering all the nodes. It is noted that the links of this spanning tree already exist in the original network (we do not create any new links). In these two figures, the nodes' labels (1,2,...,24) represent the order they join the VA-tree. Each time a node joins, its VA is assigned by its parent according to the VA assignment rule above. Thus, node 2 is the first child of node 1 and given VA(2:1) = VA(1:1) + '1' = '1', node 3 is the next child and given



(a) The VAs of all the nodes form a prefix tree

(b) The nodes with parent/child relationships form a spanning tree

Fig. 1. Virtual address instance inititated by node 1.

VA(3:1) = VA(1:1) + '01' = '01', and node 4 last and given VA(4:1) = VA(1:1) + '001' = '001'. Other nodes' VAs are assigned similarly. For example, consider node 18 which is the third child of node 8 (VA '011'). The VA of node 18 is the shortest binary string that is unused by any other child node of node 8 and of the form VA(8:1) + '0*1'. Because the other children 16 and 17 already occupy '0111' and '01101', node 18's VA is '011001'.

A VA-tree resembles the shortest-delay spanning tree rooted at the initiating node; i.e., the path from the root to a node should be the quickest path among those paths connecting them. It can be built quickly because only a single broadcast of the VA invitation is needed to assign VAs to all the nodes. The length of any VA cannot exceed $degree_{max} \times h$ where $degree_{max}$ is the maximum nodal degree in the network and h the height of the VA-tree. In a typical P2P network, this height should be at most $c\sqrt{n}$ where c is a small constant.

In $INSTANCE(S^*)$, each node S_i is associated with a "zone", denoted by $ZONE(S_i : S^*)$, consisting of all the binary strings str such that: (i) $VA(S_i : S^*)$ is a prefix of str, and (ii) no child of S_i has VA a prefix of str. In other words, among all the nodes in the network, node S_i is the one whose VA is the maximal prefix of str. We call S_i the "designated node" of str and use $NODE(str : S^*)$ to denote this node. For example, using the virtual instance $TREE(1)$ in Fig. 1(a), the zone of node 11 (VA '00101') is the set of binary strings '00101','001010', and all the strings of the form '0010100...', for which node 11 is the designated node.

The following is true for the zone assignments:

1. $ZONE(S_i : S^*) \cap ZONE(S_j : S^*) \neq \emptyset$, for every $i \neq j$.
2. $\bigcup_{i=1}^{n} ZONE(S_i : S^*) = \{0,1\}^*$.
3. $\bigcup\{ZONE(S' : S^*) | S'$ is S_i or a descendant of $S_i\} = \{str \in \{0,1\}^* | VA(S_i : S^*)$ is a prefix of $str\}$, for every i.

These properties are important to designing our indexing component.

3.1.2. Indexing

For each publish/subscribe application under deployment, the information of interest is assumed to have a fixed number of attributes called the dimension of this application. PUB-2-SUB supports any dimension and allows multiple applications to run on the network simultaneously, whose dimension can be different from one another. We use the term "event" to refer to some new information that a node wants to publish. The queries of interest are those that specify a lower-bound and an upper-bound on each event attribute. For ease of presentation, we assume for now that events are unidimensional. Later in Section 3.4, we will discuss how PUB-2-SUB enables applications of any dimensionality.

Without loss of generality, we represent an event x as a k-bit binary string (the parameter k should be chosen to be larger than the longest VA length in the network). A query Q is represented as an interval $Q = [q_l, q_h]$, where $q_l, q_h \in \{0,1\}^k$, subscribing to all events x belonging to this interval (events are "ordered" lexicographically). As an example, if $k = 3$, the events matching a query ['001','101'] are {'001','010','011',100','101',111'}.

Assuming that every node has been assigned a VA as a result of a virtualization procedure initiated by a node S^*, we require that (i) each query Q is stored at every node S_i such that $ZONE(S_i : S^*) \cap Q \neq \emptyset$; and (ii) each event x is sent to $NODE(x : S^*)$ – the designated node of string x. It is guaranteed that if x satisfies Q then Q can always be found at node $NODE(x : S^*)$ (because this node's zone must intersect Q). The dissemination algorithms to subscribe a query and publish an event are presented below in Algorithms 3.1 and 3.2, respectively.

Algorithm 3.1 (*Query Subscription*). Considering a query Q:

- Initially, the subscription starts at the subscriber node of Q.
- At each node S_i that receives Q.
  1. Quit if S_i already received this query.
  2. Let $Z = \{str \in \{0,1\}^k | VA(S_i : S^*)$ is a prefix of str}.
  3. IF (Z does not overlap Q) .
     (a) Forward Q to the parent node of S_i in $TREE(S^*)$.
  4. ELSE
     (a) Q at S_i if $ZONE(S_i : S^*)$ intersects Q.
     (b) Forward Q to all children of S_i in $TREE(S^*)$.
     (c) IF (Z does not contain Q), forward Q to parent of S_i in $TREE(S^*)$.

Algorithm 3.2 (*Event Notification*). Considering an event x:

- Initially, the notification starts at the publisher node of x.
- At each node S_i that receives x.
  1. IF ($VA(S_i : S^*)$ is not a prefix of x) THEN.
     (a) Forward x to the parent node of S_i in $TREE(S^*)$.
  2. ELSE
     (a) Find the child node S_j such that $VA(S_j : S^*)$ is a prefix of x.
     (b) IF (S_j exists) THEN Forward x to S_j.
     (c) ELSE Search node S_i for those queries matching x.

Fig. 2 shows an example with $k = 7$. Suppose that node 12 wants to subscribe a query Q = ['0110001','0110101'], thus looking to be notified upon any of the following events
{'0110001','0110010','0110011','0110100','0110101'}.
Therefore, this query will be stored at nodes {8,17,18}, whose zone intersects Q. For example, node 8's zone intersects Q because they both contain '0110001'. The path to disseminate this query is 12 → 5 → 2 → 1 → 3 → 8 → {17,18} (represented by the solid arrow lines in Fig. 2). Now, suppose that node 22 wants to publish an event x = <'0110010'>. Firstly, this event will be routed upstream to node 8 – the *first* node that is a prefix with '0110010' (path 22 → 16 → 8). Afterwards, it is routed downstream to the designated node $NODE$('0110010':1), which is node 18 (path 8 → 18). Node 18 searches its local queries to find the matching queries. Because query Q = ['0110001', '0110101'] is stored at node 18, this query will also be found.

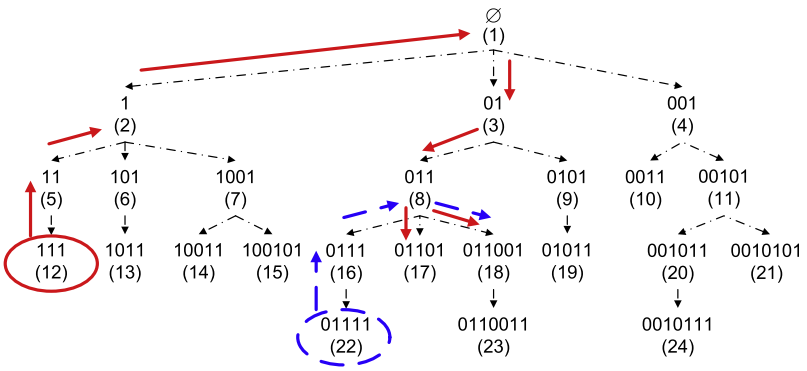


Fig. 2. Solid-bold path: subscription path of query ['0110001','0110101']; dashed-bold path: notification path of event < '0110010'>.

The storage and communication costs for a query's subscription depend on its range; the wider the range, the larger costs. For an event, the communication cost measured as the number of hops traveled to publish an event is $O(h)$ where h is the tree height ($h = O(\sqrt{n})$ in most cases). The delay to notify a matching subscriber is the time to travel this path; hence, also $O(h)$. The computation cost should be small because only one node – the designated node $NODE(x : S^*)$ – needs to search its stored queries to find those matching x. Our evaluation study in Section 4 indeed finds these costs reasonably small.

3.2. Update methods

There may be changes in the network such as when a new node is added or an existing node fails. PUB-2-SUB is adaptable to these changes. It can update the network virtualization accordingly together with the effort to avoid query loss and notification failure. Supposing that the network is virtualized according to the VA instance $INSTANCE(S^*)$, we present the update methods below.

3.2.1. Node addition

Consider a new node S_{new} that has just joined the network according to the network's underlying join protocol. As a result, it is connected to a number of neighbors. We need to add this node to $INSTANCE(S^*)$. First, this node communicates with its neighbors and asks the neighbor $S_{neighbor}$ with the minimum tree depth to be its parent; tie is broken by choosing the one with fewest children. This strategy helps keep the tree as balanced as possible so its height can be short and workload fairly distributed among the nodes. The neighbor will then assign S_{new} a VA that is the shortest unused binary string of the form $VA(S_{neighbor} : S^*) + '0^*1'$. For example, suppose that $VA(S_{neighbor} : S^*)$='1001' and the children of $S_{neighbor}$ in the tree $tree_{VA}(S^*)$ other than S_{new} have already obtained the following VAs: '10011','100101','10010001'. The VA for S_{new} would be $VA(S_{new} : S^*)$='1001001'.

Because $ZONE(S_{neighbor} : S^*)$ is changed, the next task is for the parent node $S_{neighbor}$ to delete those queries that do not intersect $ZONE(S_{neighbor} : S^*)$. Also, this parent node needs to forward to S_{new} the queries that intersect $ZONE(S_{new} : S^*)$.

3.2.2. Node removal

When a node fails to function, it is removed from the network according to the underlying maintenance protocol. This removal however affects the connectedness of the VA instance in place. Because the VAs of the child nodes are computed based on that of the parent node, the child nodes of the departing node need to find a new parent so the VA instance remains valid.

Consider such a child node S_{child}. This node selects a new parent among its neighbors. The new parent, say node S_{parent}, computes a new VA for S_{child} (similar to node addition). Then, S_{child} re-computes the VAs for its children and informs them of the changes. Each child node follows the same procedure recursively to inform all its descendant nodes downstream. The query transfer/forwarding from S_{parent} to S_{child} and, if necessary, from S_{child} to the descendant nodes of S_{child} is similar to the case of adding a new node.

In addition, because each descendant node S_i of the removed node is now assigned a new VA and thus a new zone, the queries that are stored at S_i before the VA adjustment might no longer intersect its new zone. These queries can be either deleted or re-subscribed to the network depending on the priority we can set at the first time they are subscribed to the network. If a query is marked as "high-priority", it is stored in the network permanently until the subscriber determines to unsubscribe it (the unsubscription procedure is similar to the subscription procedure). On the other hand, if a query is marked as "low-priority", it is associated with a lease time after which the query will expire and be deleted. How to implement these two types of priority is determined by the application developer.

Although PUB-2-SUB is designed for cooperative networks in which the number of node failures should be small, it is still desirable to reduce the costs involved in a query's re-subscription resulted from a failure. One way is to choose a proper parent S_{parent} for node S_{child} above. Fortunately, in a network that is richly connected, which is the case for many P2P networks, the former grandparent of S_{child} may already be a neighbor of S_{child} and thus can serve as the new parent for S_{child}. As a result, many queries that have been stored at S_i also intersect the new zone of S_i and thus do not need to be re-subscribed.

The worst case is when the root node fails in which we have no way to recover other than rebuilding the entire VA instance. To avoid this unfortunate case, we propose that the root of a VA instance be a dedicated node deployed by the network administrator. In actual implemention, it is possible to deploy another reliable node serving as the backup for the root node in case the latter fails.

3.3. Multiple VA-instances

Because query subscription and event notification procedures are based on the VA-tree, the root node and those nearby become potential hotspots. To alleviate this bottleneck problem, a solution is to build, not one, but multiple VA-instances. We can build m VA-instances initiated by dedicated nodes randomly placed in the network $\{S_1^*, S_2^*, \ldots, S_m^*\}$. After m virtualization procedures, each node S_i will have m VAs, $\text{VA}(S_i : S_1^*), \text{VA}(S_i : S_2^*), \ldots$, and $\text{VA}(S_i : S_m^*)$, respectively corresponding to the m VA-instances.

In the presence of multiple VA-instances, each query is subscribed to a random VA-instance and each event is published to every VA-instance. A node near the root of a VA-tree might be deep in other VA-trees and so the workload and traffic are better shared among the nodes. Using multiple VA-instances also increases reliability. Because an event is notified to every VA-tree, the likelihood of its finding the matching queries should remain high even if a path this event is traveling is disconnected because of some failure.

Although the storage and communication costs per query should not increase, the computation and communication costs per event increase linearly with the number of VA-instances. We will discuss these effects in our evaluation study.

3.4. Multidimensionality

In the description of PUB-2-SUB we have expressed an event as a unidimensional k-bit binary string and a query as a unidimensional interval. In practice, however, an event can have multiple attributes and as such it is usually represented as a numeric value in d dimensions where d is the number of attributes. To specify a subscription, a query is often specified as a d-dimensional rectangular range of values. PUB-2-SUB can work with events and queries of this general form.

First, we need a hash mechanism f that hashes a d-dimensional value x to a unidimensional k-bit binary string $x^f = f(x)$ and a d-dimensional range Q to a unidimensional interval $Q^f = f(Q)$ of k-bit strings such that if $x \in Q$ then $x^f \in Q^f$. For this purpose, we propose to use a $(k/2)$-order Hilbert Curve mapping [31]. This mapping preserves not only the containment relationship but also the locality property. Thus, small Q in the original space is mapped to small Q^f in the unidimensional space with a high probability. Then, to subscribe a query Q we follow Algorithm 3.1 using the hash interval Q^f. Similarly, to publish an event x we route it to the designated node of x^f according to Algorithm 3.2. When the event x reaches this node, locally stored queries are evaluated to find those matching x; the query evaluation with the event is based on the original values of the query and event (Q and x), not the hash values (Q^f and x^f).

4. Evaluation study

We believe that PUB-2-SUB can be simulated with other simulators for P2P networks, such as PeerSim [32], P2PSim [33], and OverSim [34]. However, because our purpose is to substantiate the efficiency of PUB-2-SUB, we chose to develop a home-grown simulator which was simpler to develop, more flexible to tune, yet still guaranteeing correctness. Our simulator was event-driven and written in C. The simulated network consisted of 1000 nodes whose topology was a Waxman random graph generated with the BRITE generator [35]. Unlike highly dynamic free-to-grow P2P networks, a cooperative P2P network (e.g., grid networks) should be designed so that the node degrees are not so skewed as in the power-law distribution (a similar argument is given in [36]). Thus, although PUB-2-SUB can work with any topology, we choose to discuss in this section the results for the simulated network as a uniform, not power-law, random graph. This network consists of 2766 peer-to-peer links; hence, 5.5 neighbors per node.

An event was represented as a k-bit string and a query an arbitrary interval of k-bit strings (as discussed in Section 3.4, other event/query models can be transformed to this model). A query or event was initiated by a random node chosen uniformly. To cover a large domain of possible events, we set k to 50 bits, thus able to specify up to 2^{50} different events. From this domain, 10,000 events were chosen uniformly in random. The subscription load consisted of 10,000 queries, each having a range chosen according to the following Zipf's law: in the set of possible ranges $\{2^0, 2^1, \ldots, 2^{49}\}$, range 2^i is picked with probability $\frac{1/i^\alpha}{\sum_{j=1}^{50}(1/j^\alpha)}$. The number of events belonging to a query thus could be as large as half the entire domain size (2^{49}) or just one (exact match). We considered two query range models: $\alpha = 0$ representing the uniform distribution, and $\alpha = 0.8$ representing a heavy-tail distribution where a vast majority of the queries are specific, i.e., short range, with

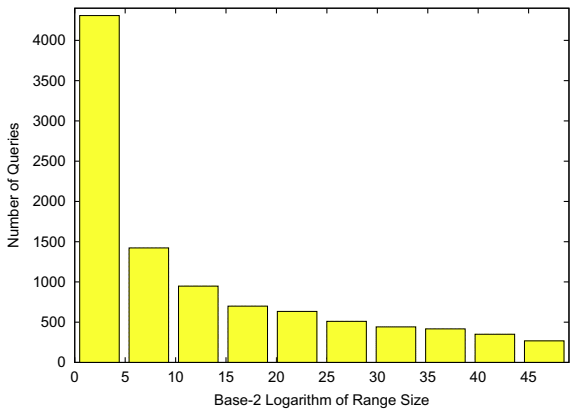
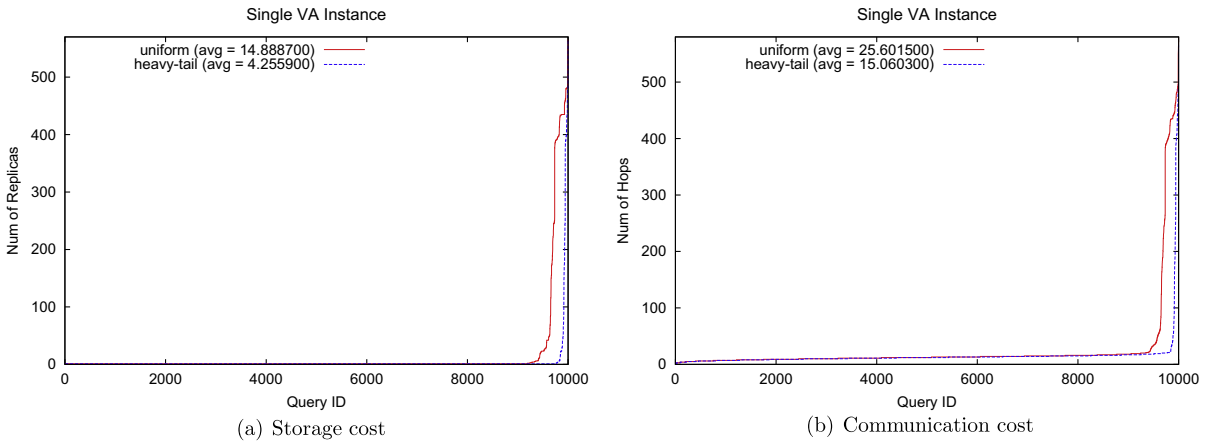


Fig. 3. Histogram of query ranges in the heavy-tail query model.



(a) Storage cost (b) Communication cost

Fig. 4. Query subscription costs.

an average size of 2^{12} (see Fig. 3). The heavy-tail model reflects many real-world applications.

We evaluated PUB-2-SUB in the following aspects: subscription efficiency, notification efficiency, notification delay, failure effect, load balancing, and effect of using multiple VA-instances. We also compared PUB-2-SUB to BubbleStorm [23] – a recent search technique designed for unstructured P2P networks. Two versions of BubbleStorm were considered: (1) BubbleStorm-64%: each query or event is sent to $\sqrt{n}$ nodes, resulting in a 64% query/event matching success rate (when there is no failure), and (2) BubbleStorm-98%: each query or event is sent to $2\sqrt{n}$ nodes, resulting in a 98% success rate (when there is no failure). The evaluation results are discussed below.

4.1. Subscription efficiency

This efficiency is measured in terms of the storage cost and the communication cost. The storage cost is computed as the number of nodes that store a given query, and the communication cost as the number of hops (nodes) that have to forward this query during its subscription procedure.

Fig. 4 shows these costs for every query, which are sorted in non-decreasing order. It is observed for either cost that all queries result in a small cost except for a very few with a high-cost. These high-cost queries are those with long ranges. As such, they intersect the zones of many nodes and thus have to travel more to be stored at these nodes. Despite so, on average, a query is replicated at only 15 nodes (uniform case) and 4.3 nodes (heavy-tail case), resulting in a communication cost of 25.6 hops (uniform case) and 15 hops (heavy-tail case).

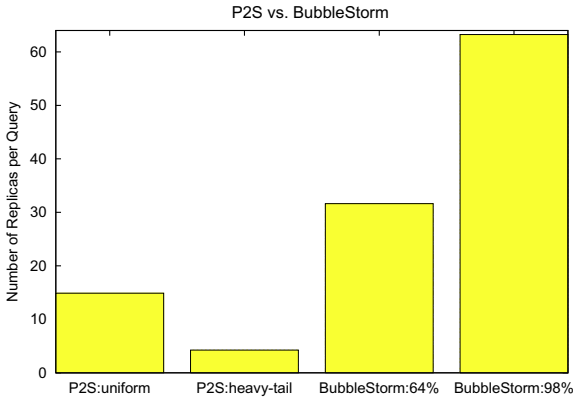


Fig. 5. PUB-2-SUB vs. BubbleStorm: storage cost.

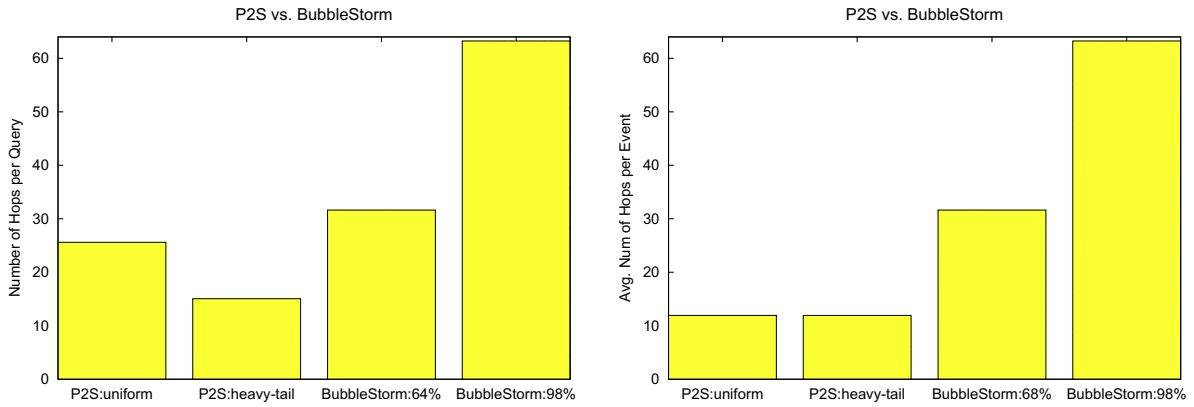
These costs are much lower than that incurred by BubbleStorm. Fig. 5 shows that BubbleStorm-64% stores an average query at 33 nodes, more than twice the storage cost of PUB-2-SUB (uniform) and eight times the cost of PUB-2-SUB (heavy-tail). The storage cost of BubbleStorm-98% is even higher. In terms of the communication cost, as seen in Fig. 6(a), a query in BubbleStorm-64% and BubbleStorm-98% has to travel 33 hops and 66 hops, respectively, which are also higher than the communication cost of PUB-2-SUB.

4.2. Notification efficiency

This efficiency is measured in terms of the communication cost and the computation cost. The communication cost is computed as the number of hops, i.e., nodes that have to forward a given event during its publication procedure, and the computation cost as the number of queries evaluated to match this event.

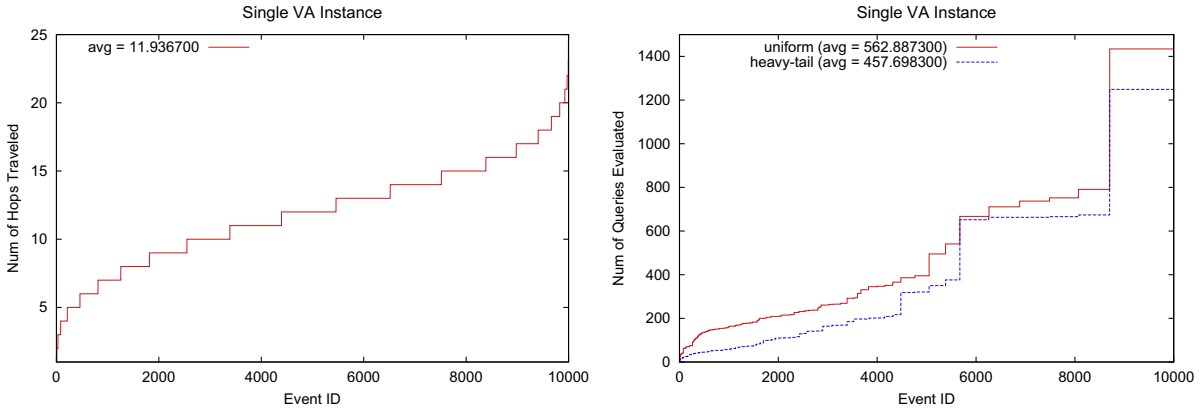
Because an event is routed based on the nodes' VAs, its communication cost is independent of the query model used, uniform or heavy-tail. Fig. 7(a) shows that this cost is distributed normally from zero hops (best-case) to 25 hops (worst case), having an average of 12 hops. The event communication cost is also much lower (by approximately three times at least) when compared to BubbleStorm (Fig. 6(b)). This study together with the study of the subscription efficiency are evident that PUB-2-SUB clearly outperforms BubbleStorm in both storage cost and communication cost.

In terms of computation cost, Fig. 7(b) shows that in the worst case about 1400 queries need to be evaluated to find all those that match a given event. This number is however, only 14% of the entire query population. On average, the computation cost is only 563 query evaluations (uniform



(a) Query subscription (b) Event notification

Fig. 6. PUB-2-SUB vs. BubbleStorm: communication cost.



(a) Communication cost (b) Computation cost

Fig. 7. Event notification costs.

case) and 458 query evaluations (heavy-tail case), corresponding to 5.63% and 4.58% of the query population, respectively.

4.3. Notification delay

When an event is published, there might be more than one query subscribing to this event. To represent the notification delay for each (event, query) matching pair, we compute the ratio a/b where a is the hopcount-based distance the event has to travel from the publisher node to the subscriber node and b is the hopcount-based distance directly between these two nodes. This ratio is at least 1.0 because even if the publisher knows the subscriber, it must already take b hops to send the event to the subscriber. In practice, because the publisher and the subscriber initially do not know each other, it is impossible to obtain a perfect 1.0 ratio.

Fig. 8 plots the histogram of notification delay incurred by PUB-2-SUB. Approximately, 70% of the notifications have a delay not exceeding 2.0 (i.e., twice the perfect delay) and 90% have a delay not exceeding 3.0 (i.e., three times the perfect delay). Thus, despite a few (event, query) pairs with high notification delay, a vast majority of events can notify their matching queries reasonably quickly.

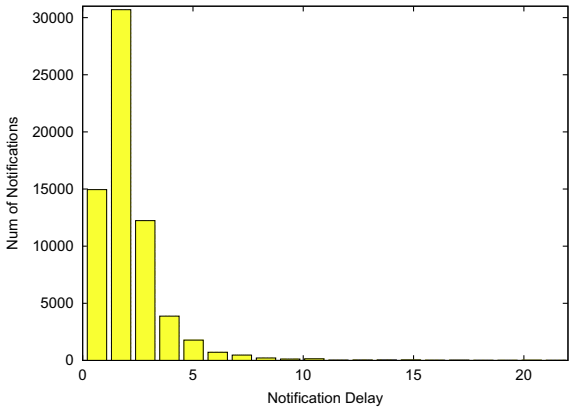
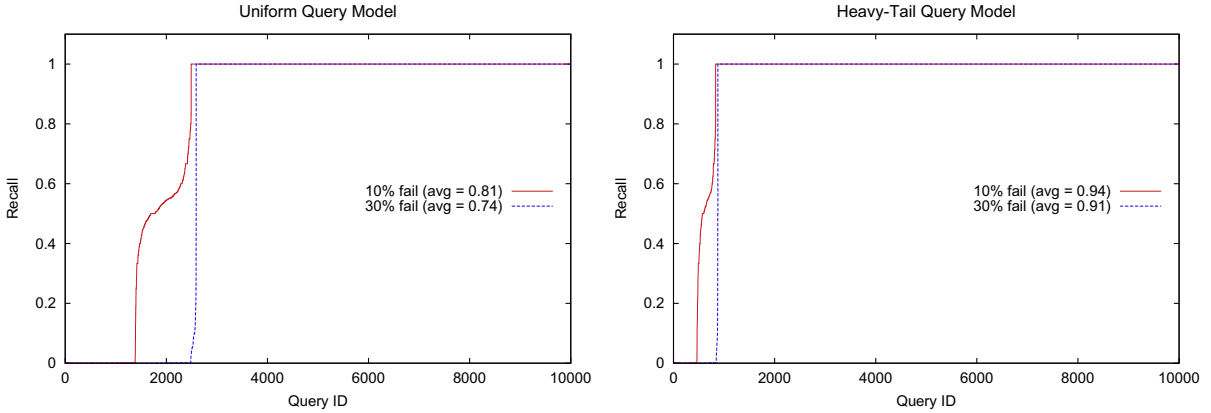


Fig. 8. Histogram of publisher-to-subscriber notification delay.

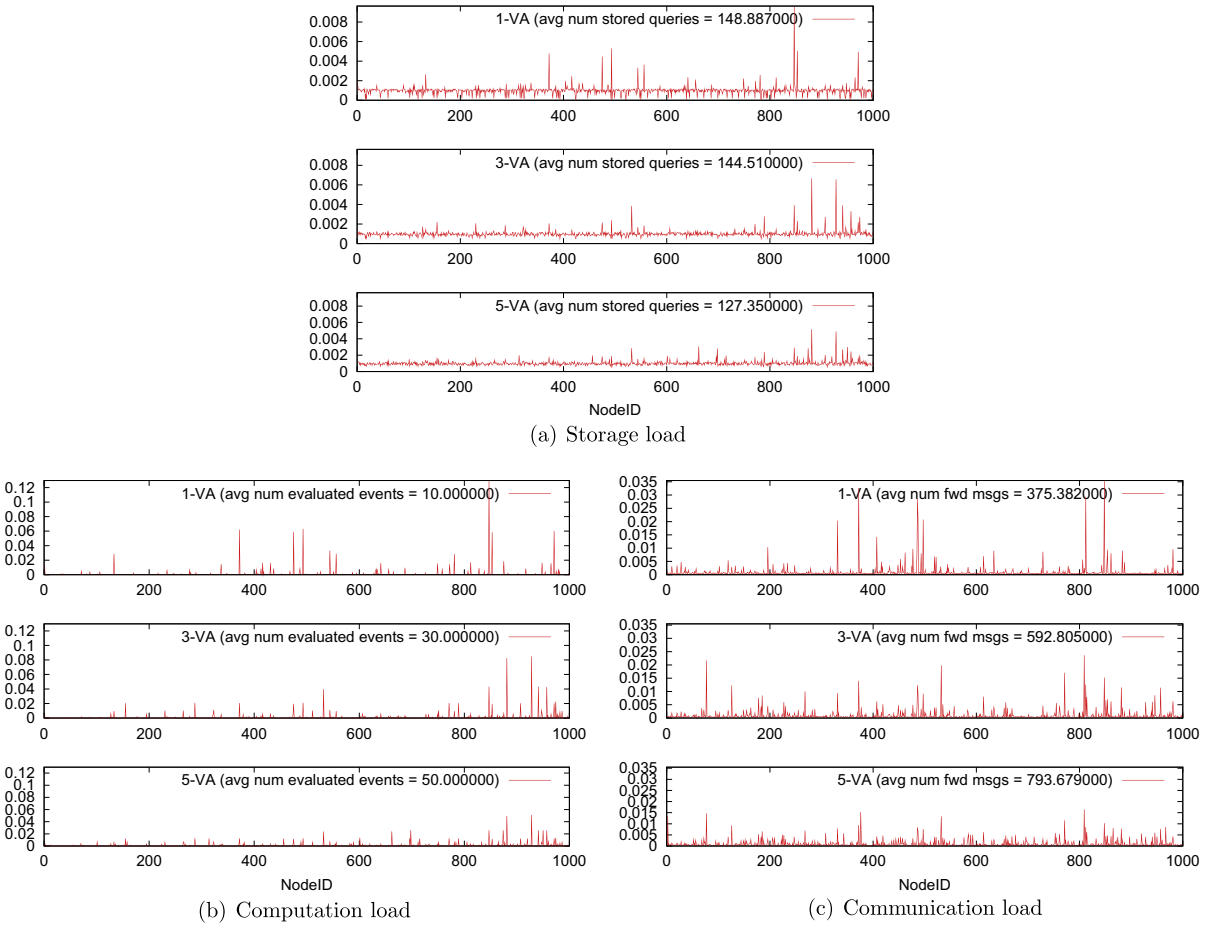
4.4. Failure effect

Although PUB-2-SUB is designed for cooperative P2P networks where all the nodes are supposed to be operational as much as possible, node failures are inevitable. When a node stops functioning, an event might fail to notify its subscribers. To evaluate PUB-2-SUB's effectiveness under such a failure, we compute "recall" – the percentage of the returned events that match a given query out of all the matching events. If there is no failure, the recall for



(a) Uniform query model (b) Heavy-tail query model

Fig. 9. Effect of Failures: 10% of nodes fail and 30% of nodes fail.



(a) Storage load

(b) Computation load (c) Communication load

Fig. 10. Load balancing.

every query is 100%. Upon a failure, a high recall is desirable because it implies that the system remains effective. We consider the case where 10% of the nodes fail simultaneously and the case where 30% fail.

Fig. 9(a) shows the results for the uniform-query model case, where it is observed that 75% of the queries are successfully notified by all the matching events (i.e., recall = 100%) even when 30% of the nodes fail. The difference between the 10%-fail case and the 30%-fail case is that in the latter case most of the remaining queries (the remaining 25%) fail to receive any matching event while in the former case about half of the queries do not receive any matching event and the other half receiving at least some portion of the matching events. On average, the recall for the 10%-fail case is 81%, and for the 30%-fail case is 74%.

Higher recall is obtained when the query model is heavy-tail (see Fig. 9(b)). The average recall is 91% when 30% of the nodes fail and 94% when 10% fail. The results are encouraging because in practice the query range should follow the heavy-tail model more often than the uniform model. This study is demonstrative of PUB-2-SUB's sustainable effectiveness when a large portion of the network fails to operate.

4.5. Load balancing and effect of using multiple VA-instances

We compute for each node S_i the storage load $L_i^{store}/\sum_{j=1}^{n} L_j^{store}$, computation load $L_i^{comp}/\sum_{j=1}^{n} L_j^{comp}$, and communication load $L_i^{comm}/\sum_{j=1}^{n} L_j^{comm}$, and investigate the variation in each type of load. Here, L_i^{store} is the number of queries stored at node S_i, L_i^{comp} the number of events evaluated at node S_i, and L_i^{comm} the number of queries/events forwarded by node S_i.

When a single VA is used, the root node of the VA-tree and those nearby are likely to have higher workload than those deeper in the tree. Although this problem is not avoidable, Fig. 10 (top diagrams) demonstrates that such high-loaded nodes represent a minor network population. Among the remaining nodes the workload is quite balanced, especially for storage load (Fig. 10(a) (top diagram)). On average, a node has to store 149 queries (1.5% of all the queries; see Fig. 10(a) (top diagram)), evaluate 10 events (1% of all the events; see Fig. 10(b) (top diagram)), and forward 794 queries and events (4% of all queries and events; see Fig. 10(c) (top diagram)). The workload per node is therefore small.

As discussed in Section 3.3, using multiple VA-instances is a way to improve load balancing. This improvement is evident in Fig. 10 which shows significantly fewer nodes with peak workloads as we increase the number of VA-instances from one to three (Fig. 10 (middle diagrams)) to five (Fig. 10 (bottom diagrams)). Further, the storage cost also gets better. When five VA-instances are used, an average node stores 128 queries, less than 149 queries if a single VA is used.

The computation and communication costs per node are however proportionally increased if more VA-instances are used. This is understandable because each event is published to all the VA-instances. Using multiple VA-instances is thus a tradeoff between (i) better load balancing and storage cost versus (ii) worse communication and computation costs. If there are few events in the network compared to the number of subscription queries, we would rather use multiple VA-instances. Otherwise, a single VA instance should be used.

5. Conclusions

Many distributed computing networks have adopted P2P as an effective way to share resources, reduce server costs, and promote collaboration. Useful to these networks is a mechanism that enables publish/subscribe applications. We have proposed such a mechanism, called PUB-2-SUB, which can be integrated into any unstructured network. Using PUB-2-SUB, any number of content-based publish/subscribe applications can be deployed simultaneously. Unlike the gossip-based approach previously recommended for unstructured networks, the proposed technique is based on directed routing and incurs less storage and communication costs. This is evident in an evaluation study in which PUB-2-SUB is compared to a representative technique of the other approach. It is also found that our technique results in low computation cost and low notification delay and remains highly effective in cases when many nodes in the network stop to function.

We do not recommend PUB-2-SUB for use in highly dynamic networks with the nodes being on and off frequently. Instead, PUB-2-SUB works best for P2P-based cooperative networks in which the nodes are supposed to be functional most of the time and failures do not happen too often. Thus, data grid networks and institutional collaborative networks can take full advantage of the proposed technique.

Acknowledgment

This work was supported in part by the National Science Foundation under award number CNS-0753066. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the National Science Foundation.

References

[1] X. Sun, J. Liu, E. Yao, X. Chen, A scalable P2P platform for the knowledge grid, IEEE Transactions on Knowledge and Data Engineering 17 (12) (2005) 1721–1736. doi:http://dx.doi.org/10.1109/TKDE.2005.190.

[2] Y. Teranishi, H. Tanaka, Y. Ishi, M. Yoshida, A geographical observation system based on P2P agents, in: PERCOM'08: Proceedings of the 2008 6th Annual IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Washington, DC, USA, 2008, pp. 615–620. doi:http://dx.doi.org/10.1109/PERCOM.2008.63.

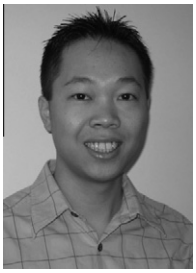
[3] N. Shalaby, J. Zinky, Towards an architecture for extreme P2P applications, in: Parallel and Distributed Computing and Systems Conference (PDCS), Cambridge, MA, 2007.

[4] M. Cai, M. Frank, J. Chen, P. Szekely, Maan: A multi-attribute addressable network for grid information services, GRID'03: Proceedings of the 4th International Workshop on Grid Computing, IEEE Computer Society, Washington, DC, USA, 2003, p. 184.

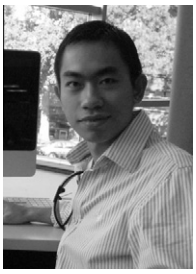
[5] U. Liebel, Sciencenet search-engine-based on-yacy-P2P-technology, January, 2008, http://liebel.fzk.de/collaborations/sciencenet-search-engine-based on-yacy-P2P-technology.

[6] T. Hey, A.E. Trefethen, Cyberinfrastructure for e-science, Science 308 (5723) (2005) 817–821.

[7] K.A. Peterson, P. Fontaine, S. Speedie, The electronic primary care research network (ePCRN): a new era in practice-based research, The Journal of the American Board of Family Medicine 19 (2006) 93–97.
[8] IBM, Ibm grid and grow express: a solution for competitive business, August (2005). http://www-03.ibm.com/linux/grid/gridandgrow.shtml.
[9] E.N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J.B. Park, A. Vernon, Scalable trigger processing, in: Proceedings of the 15th International Conference on Data Engineering, IEE Computer Society, Sydney, Austrialia, 1999. pp. 266–275.
[10] J. Chen, D.J. DeWitt, F. Tian, Y. Wang, Niagaracq: a scalable continuous query system for internet databases, SIGMOD Record 29 (2) (2000) 379–390. doi:http://doi.acm.org/10.1145/335191.335432.
[11] F. Fabret, H.A. Jacobsen, F. Llirbat, J. Pereira, K.A. Ross, D. Shasha, Filtering algorithms and implementation for very fast publish/subscribe systems, in: SIGMOD'01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, ACM Press, New York, NY, USA, 2001, pp. 115–126. doi:http://doi.acm.org/10.1145/375663.375677.
[12] M. Castro, P. Druschel, A. Kermarrec, A. Rowstron, SCRIBE: a large-scale and decentralized application-level multicast infrastructure, IEEE Journal on Selected Areas in Communications (JSAC) 20 (8) (2002) 1489–1499.
[13] A. Gupta, O.D. Sahin, D. Agrawal, A.E. Abbadi, Meghdoot: content-based publish/subscribe over P2P networks, in: Middleware'04: Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware, Springer-Verlag New York, Inc., New York, NY, USA, 2004, pp. 254–273.
[14] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, A.P. Buchmann, A peer-to-peer approach to content-based publish/subscribe, DEBS'03: Proceedings of the 2nd International Workshop on Distributed event-based systems, ACM Press, New York, NY, USA, 2003, pp. 1–8.
[15] I. Aekaterinidis, P. Triantafillou, Internet scale string attribute publish/subscribe data networks, in: CIKM'05: Proceedings of the 14th ACM International Conference on Information and knowledge Management, ACM Press, 2005, pp. 44–51.
[16] D.A. Tran, T. Nguyen, Publish/subscribe service in can-based P2P networks: dimension mismatch and the random projection approach, in: IEEE Conference on Computer Communications and Networks (ICCCN'08), IEEE Press, Virgin Island, USA, 2008.
[17] S. Bianchi, P. Felber, M. Gradinariu, Content-based publish/subscribe using distributed R-trees, Euro-Par, 2007, 537–548.
[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker, A scalable content addressable network, in: ACM SIGCOMM, San Diego, CA, 2001, pp. 161–172.
[19] I. Stoica, R. Morris, D. Karger, M. Kaashock, H. Balakrishman, Chord: a scalable peer-to-peer lookup protocol for internet applications, in: ACM SIGCOMM, San Diego, CA, 2001, pp. 149–160.
[20] A. Rowstron, P. Druschel, Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems, in: IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, 2001, pp. 329–350.
[21] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, J. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, IEEE Journal on Selected Areas in Communications 22 (1) (2004) 41–53.
[22] W. Pugh, Skip lists: a probabilistic alternative to balanced trees, Communications of the ACM 33 (1990) 668–676.
[23] W.W. Terpstra, J. Kangasharju, C. Leng, A.P. Buchmann, BubbleStorm: resilient, probabilistic, and exhaustive peer-to-peer search, in: SIGCOMM'07: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ACM, New York, NY, USA, 2007, pp. 49–60. doi:http://doi.acm.org/10.1145/1282380.1282387.
[24] B. Wong, S. Guha, Quasar: a probabilistic publish/subscribe system for social setworks, in: Proceedings of the 7th International Workshop on Peer-to-Peer Systems (IPTPS'08), Tampa Bay, FL, 2008.
[25] C. Gkantsidis, M. Mihail, A. Saberi, Random walks in peer-to-peer networks: algorithms and evaluation, Performance Evaluation 63 (3) (2006) 241–263. doi:http://dx.doi.org/10.1016/j.peva.2005.01.002.
[26] P. R. Pietzuch, J. Bacon, Peer-to-peer overlay broker networks in an event-based middleware, DEBS'03: Proceedings of the 2nd International Workshop on Distributed Event-Based Systems, ACM, New York, NY, USA, 2003, pp. 1–8.
[27] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, J. D. Kubiatowicz, Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination, NOSSDAV'01: Proceedings of the 11th international Workshop on Network and Operating Systems Support for Digital Audio and Video, ACM, New York, NY, USA, 2001, pp. 11–20.
[28] S. Voulgaris, E. Rivire, A.-M. Kermarrec, M. van Steen, Sub-2-sub: self-organizing content-based publish subscribe for dynamic large-scale collaborative networks, in: Fifth International Workshop on Peer-to-Peer Systems (IPTPS 2006), 2006.
[29] A. Guttman, R-trees: a dynamic index structure for spatial searching, in: ACM SIGMOD Conference on Management of Data, 1984, pp. 47–57.
[30] R. Guerraoui, S. Handurukande, K. Huguenin, A.-M. Kermarrec, F. Le Fessant, E. Riviere, GosSkip, an efficient, fault-tolerant and self-organizing overlay using gossip-based construction and skiplists principles, in: IEEE International Conference on Peer-to-Peer Computing, 2006.
[31] J.K. Lawder, P.J.H. King, Using space-filling curves for multidimensional indexing, in: BNCOD 17: Proceedings of the 17th British National Conference on Databases, Springer-Verlag, London, UK, 2000, pp. 20–35.
[32] M. Jelasity, A. Montresor, G.P. Jesi, S. Voulgaris, The Peersim simulator, http://peersim.sf.net.
[33] P2PSim, A simulator for P2P protocols, http://pdos.csail.mit.edu/p2psim.
[34] I. Baumgart, B. Heep, S. Krause, OverSim: a scalable and flexible overlay framework for simulation and real network applications, in: Ninth International Conference on Peer-to-Peer Computing (IEEE P2P'09), 2009, pp. 87–88. doi:10.1109/P2P.2009.5284505.
[35] A. Medina, A. Lakhina, I. Matta, J. Byers, Brite: an approach to universal topology generation, IEE Computer Society, Washington, DC, USA, 2001. p. 346.
[36] Q. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, Search and replication in unstructured peer-to-peer networks, in: ICS'02: Proceedings of the 16th ACM International Conference on Supercomputing, ACM Press, New York, NY, USA, 2002, pp. 84–95.



Duc A. Tran is an Assistant Professor in the Department of Computer Science at the University of Massachusetts at Boston, where he leads the Network Information Systems Laboratory (NISLab). He received a Ph.D. degree in Computer Science from the University of Central Florida (Orlando, Florida) in 2003. His interests are in the areas of networking and distributed systems, particularly in support of information systems that can scale with both network size and data size. The results of his work have led to research grants from the National Science Foundation (NSF), a Best Paper Award at ICCCN 2008, and a Best Paper Recognition at DaWak 1999. He has engaged in many professional activities, serving as a multi-time Review Panelist for the NSF, Guest-Editor for two international journals, TPC Chair for IRSN 2009 and GridPeer 2009, TPC Vice-Chair for AINA 2007, TPC member for 30+ international conferences, and referee and session chair for numerous journals/conferences.



Cuong (Charlie) Pham is a Ph.D. student in the Department of Computer Science at the University of Massachusetts at Boston and a research member of NISLab. He received a BS degree in Computer Science from Bowman Technical State University in Moscow, Russia in 2007. His research interests are P2P networks and wireless sensor networks. He has received a Student Travel Award from the NSF and a Research Excellence Award from the Department of Computer Science (UMass Boston), both in 2009.