PUBLISH/SUBSCRIBE TECHNIQUES FOR P2P NETWORKS

Cuong Pham, University of Massachusetts, Boston, MA 02125, USA
Duc A. Tran, University of Massachusetts, Boston, MA 02125, USA

Abstract – As P2P is a popular networking paradigm in today's Internet, many research and development efforts are geared toward services that can be useful to the users of P2P networks. An important class of such services is that based on the publish/subscribe paradigm to allow the nodes of a network to publish data and subscribe data interests efficiently. This chapter is focused on the techniques that enable these services in P2P networks.

Keywords – Publish/subscribe, P2P, search, content-based search, DHT

INTRODUCTION

A publish/subscribe networking system is one in which the nodes can serve the role of a publisher or a subscriber to publish data or subscribe for data of interest, respectively. The publish/subscribe model differs from other request/response models in that a query of the former model is submitted and stored in advance, for which the result may not yet exist but the query subscriber expects to be notified if and when the result later becomes available. The publish/subscribe 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.

Enabling publish/subscribe services in peer-to-peer (P2P) networks is a topic that has received a lot of attention in recent years. As P2P can be adopted for distributed networking as an effective way to share resources, minimize server costs, and promote boundary-crossing collaborations, a publish/subscribe functionality should be useful to these networks. For example, a monitoring operator in a P2P-based geographical observation network [Teranishi et al. (2008)] 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 network [Shalaby & Zinky (2007)], a subscriber will be notified when new scientific information is published.

Usually, a publisher node does not know who is interested in its data, and, vice versa, a subscriber node does not know where in the network its data of interest is available. Thus, a challenging problem is to design mechanisms for the subscribers and publishers to find each other quickly and efficiently. A simple way is to broadcast each query to all the nodes in the network or to employ a centralized index of all the queries subscribed and information published. This mechanism is neither efficient nor scalable if applied to a large-scale network.

Consequently, a variety of distributed publish/subscribe mechanisms have been proposed. They follow two main approaches: gossip-based and structure-based. The first approach is designed for any unstructured networks, in which the subscriber nodes and publisher

nodes find each other via exchanges of information using the existing peer links, typically based on some form of randomization. The other approach organizes the nodes 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 (e.g., CAN [Ratnasamy et al. (2001)], Chord [Stoica et al. (2001)]). The gossip-based approach's advantage is its applicability to any unstructured network, while the structure-based approach is favored for better efficiency.

This chapter provides a survey on the publish/subscribe techniques for P2P networks. First, we will provide some necessary preliminaries. We then discuss several representative techniques in each of the following categories: structure-based, gossip-based, and a hybrid of these two. We conclude the chapter with some remarks.

PRELIMINARIES

Peer-to-Peer Networks

A P2P network is a decentralized network of equivalent-role nodes. A node can serve in either a "server" role or a "client" role, or both, depending on circumstances. Unlike traditional client/server networks, P2P networks have no limit for growth and no single point of failure. The capability to share resources and the freedom to join and leave the network at any time are among the properties that make P2P networks very popular on the Internet today.

There are two main types of P2P networks: structured and unstructured. In unstructured P2P networks, e.g., [Gnutella] and [Freenet], the links between nodes are formed in an ad hoc manner without any predefined structure. Unstructured P2P networks are easy to maintain under network dynamics. They are fully decentralized with a high degree of fairness. However, they are not efficient in search operations. Search in a unstructured network usually requires broadcasting of the query, thus incurring a high communication cost.

Structured P2P networks are designed for better search operations. In such a network, the nodes are arranged in an overlay structure which provides efficient routing and lookup mechanisms. Distributed Hash Tables (DHT) is the most popular structure for structured P2P networks (e.g., CAN [Ratnasamy et al. (2001)], Chord [Stoica et al. (2001)], Pastry [Rowstron & Druschel (2001)], and Tapestry [Zhao et al. (2004)]). As CAN is used later in this chapter, let us describe briefly how it works. In CAN (abbr. of "Content Addressable Networks"), the network is viewed virtually as a multi-dimensional geometric space, called the CAN space, in which each node is assigned a location. The CAN space is partitioned into rectangular zones and the node location assignments are determined such that there is only one node in each zone. An overlay neighbor link is created between two nodes if their zones are adjacent. The control overhead for each node is therefore small because a node only needs to keep track of its neighbors and the number of these neighbors is at most twice the CAN space dimensionality which is a constant.

Data storage and retrieval in CAN work as follows. Each data object is hashed into a location in the CAN space and its index is stored at the node whose zone contains this hash location. When a query for an object is initiated, the hash location of the object is computed and the query is sent to the node owning this location and thus can find the object there. Routing to a location in the CAN space is based on geometry-based greedy routing via the overlay neighbor links: in each forwarding step, the message is always forwarded to a neighbor node that is closer to the destination node, geometrically. The distance between two nodes, which defines "close-ness", is computed using their corresponding CAN locations.

Figure 1 provides an illustration of indexing and retrieval in a two-dimensional CAN network. Here, the CAN space is a square [0, 1] x [0, 1]. Each node (1→9) owns a zone, which is a rectangle. Node 1 has a file associated with key K. The pointer (K, 1) (meaning that node 1 has a file with key K) is stored in the node whose zone contains h(K) - the hashing location of K, which is node 4. Once a node 3 request a file with key K, it needs to route the query to location h(K) and thus will reach node 4, where every file with key K will be found. Routing from node 3 to node 4 is by relaying via neighbor nodes, greedily getting as close geometrically to the destination as possible.

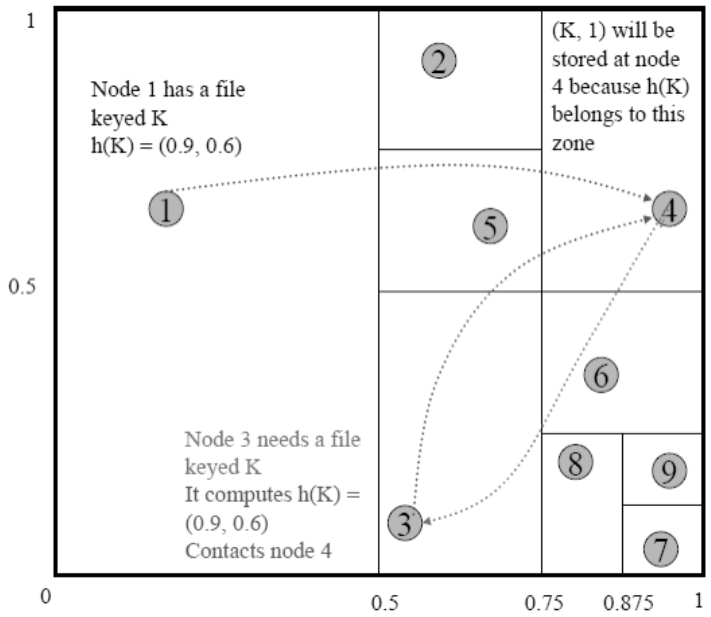

Fig. 1. Indexing and retrieval in CAN.

Figure 2 illustrates the construction and routing procedures of CAN. Suppose that node 10 want to join the CAN network. This node chooses a random location, which, we suppose, currently resides in the zone of node 9. Node 10 then contacts node 9 and asks for a share of the latter' zone. Node 9 halves its zone into two smaller zones, retaining one and giving the other to node 10.

Routing in CAN is even simpler. Suppose that node 1 is looking for some data, whose hashing location is in the zone of 7. Node 1 chooses among its neighbors the one node that is geometrically closest to 7 to forward the message; in this case, node 2. Node 2 then repeats the same procedure. Eventually, the routing path from node 1 to node 7 is 1→2→3→6→7.

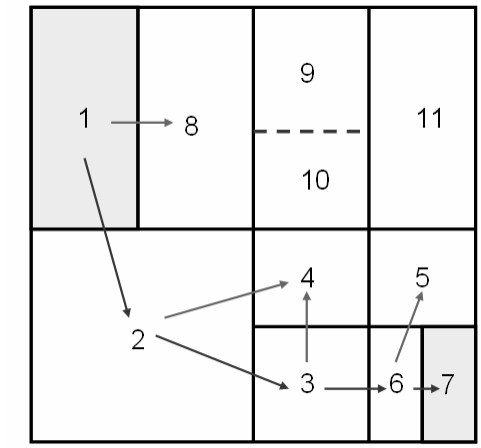

Fig. 2. CAN constructing and routing

The advantage of structured P2P networks is their scalability and lookup efficiency. However, they incur the cost of constructing and maintaining the overlay structure, most of which are due to node departures, arrivals, or failures.

Publish/Subscribe Services

Event and Query Representation

An event in a publish/subscribe system is usually specified as a set of *d* attribute-value pairs {($attr_1$, v_1), ($attr_2$, v_2), ..., ($attr_d$, v_d)} where *d* is the number of attributes, {$attr_1$, $attr_2$, ..., $attr_d$}, associated with the event. For example, consider a P2P network to monitor fire in a large region. Each peer records temperature, humidity, wind speed, and air pressure in its area over time, thus *d* is four – representing these four data attributes: $attr_1$ = *'temperature'*, $attr_2$= *'humidity'*, $attr_3$ = *'wind speed'*, *and* $attr_4$ = *'air pressure'*. In generic expression, the constraints in a query can be specified in a predicate of disjunctive normal form – a disjunction of one or more condition clauses, each clause being a conjunction of elementary predicates. Each elementary predicate, denoted by ($attr_i$ *?* p_i), is a condition on some attribute $attr_i$ with *'?'* being the filtering operator. As used in the literature of publish/subscribe techniques, a filtering operator can be a comparison operator (one of {=, <, >}) or a string operator such as "*prefix of*", "*suffix of*", and "*substring of*" if the attribute is of *string* type. Thus, to be notified of all locations with temperature above 100°F, humidity below 20%, and wind speed above 50 mph, no mater what air pressure, the query can be expressed as (*'temperature' > 100*) AND (*'humidity' < 20*) AND (*'wind speed' > 50*). An event *x* satisfies a query *q*, denoted by *x*

$\in q$, if and only if x satisfies all the elementary predicates specified in at least one condition clause of q.

The publish/subscribe scheme above allows a flexible way to specify a query as a disjunction of any number of conjunctive clauses and the filtering operator '*?*' can be any of the aforementioned, including the string operators. However, for simplicity of implementation, most schemes assume that a query is a single conjunctive clause of elementary predicates that can only use the comparison operators {=, <, >}. This form of query can thereffore be called the *rectangular form* because if an event is modeled as a point in a *d*-dimensional coordinate system, each dimension representing an attribute, a query can be considered a *d*-dimensional box with the vertices defined based on the attribute constraint values provided in the query clause.

It is sometimes a tedious process to specify the lower and upper bounds for all the attributes of a query. In such cases, it is more convenient to provide an event sample as the query and request to be notified of all the events similar to this sample. For example, consider a camera remote surveillance network deployed over many airports to detect criminal suspects. If a particular suspect is searched for, his or her picture is submitted as a subscription to the network in hopes of finding the locations where similar images are captured. A query of this kind can be represented by a sphere, in which the sample is the center of the sphere and similarity is constrained by the sphere's radius. This query is said to have the *spherical form*.

Subject-based vs. Content-based

There are two main types of publish/subscribe designs: subject-based or content-based. In the subject-based design, events are categorized into a small number of known *subjects*. There must be an event attribute called '*subject*', or something alike, that represents the type of the event and a query must include a predicate ('*subject*' = *s*) to search only events belonging to some known subject *s*. The occurrence of any event of subject *s* will trigger a notification to the query subscriber. The subscription and notification protocols are mainly driven by subject match rather than actual-content match.

The content-based design offers a finer filtering inside the network and a richer way to express queries. A subscriber wants to receive only the events that match its query content, not all the events that belong to a certain subject (which could result in too many events). For example, many Bostonians are only interested in the Celtics and do not want to be bothered by any event published regarding the Lakers. A query with subject "NBA" would result in receiving all US professional basketball events including those about the Celtics and the Lakers. A content-based query is therefore more desirable. A node upon receipt of a query or event message needs to extract the content and makes a forwarding decision based on this content. We can think of the subject-based model as a special case of the content-based model and because of this simplification a subject-based system is less challenging to design than a content-based system is.

STRUCTURE-BASED PUBLISH/SUBSCRIBE TECHNIQUES

A popular approach among structure-based publish/subscribe techniques is to employ a Distributed Hash Table (DHT) to build an overlay structure on top of the P2P network. This overlay provides efficient methods to route queries and events to their corresponding nodes that are determined based on the hashing function. The goal is that the node storing a subscription and that receiving a satisfactory event are either identical or within a proximity of each other. Scribe [Castro et al. (2002)] uses Pastry [Rowstron & Druschel (2001)] to map a subscription to a node based on topic hashing, thus those subscriptions and data objects with the same topic are mapped to the same node. Instead of Pastry, the CAN [Ratnasamy et al. (2001)] and Chord [Stoica et al. (2001)] DHT structures are employed in [Gupta et al. (2004)] and [Terpstra et al. (2003)], respectively. A technique that can be used atop any such DHT structure was proposed in [Aekaterinidis & Triantafillou (2005)]. Non-DHT techniques also exist, such as Sub-2-Sub [Voulgaris et al. (2006)] and R-tree-based [Bianchi et al. (2007)].

To illustrate the structure-based approach, we discuss how publish/subscribe services can be deployed in a P2P network structured using the CAN overlay [Ratnasamy et al. (2001)] discussed earlier in this chapter. Although CAN is an efficient overlay for traditional retrieval in P2P networks, deploying a publish/subscribe service on top of CAN is not as straightforward. From the database perspective, because we typically model a data object as a point and a query as a range of points, we need to address the range indexing problem in publish/subscribe systems. From the networking perspective, due to its range, a subscription query may be replicated at multiple nodes to wait for notification of all possible matching data objects. Hence, the number of subscriptions stored in the network can be large, resulting in not only high communication cost to replicate the subscriptions, but also high storage cost for each node and long time to match an object against a subscription query. We need to minimize unnecessary replications, yet at the same time store the queries in the network intelligently so that data notification remains efficient.

Because of the low dimensionality of the CAN space, another challenge to a CAN-based publish/subscribe system is due to the mismatch between the CAN dimension and the data dimension. Data can, and usually, be of high dimension, such as in applications searching documents, multimedia, and sensor data, which normally are associated with many attributes. It is difficult to hash similar high-dimension data objects into zones in a low-dimension space which are close to each other, making the search for a continuous range of data highly inefficient.

Meghdoot [Gupta et al. (2004)] is a CAN-based publish/subscribe technique that works for multi-dimensional data space. In Meghdoot, each subscription query in d dimensions is mapped a point in $2d$ dimensions and the P2P network is virtualized in a CAN space of $2d$ dimensions. In the case that d is large, the CAN dimension is large, making CAN very inefficient. In addition, Meghdoot does not allow publish/subscribe applications with different data dimensions to run on the same CAN network.

Next, we describe a technique [Tran & Nguyen (2008)] using Random Projection (RP) to map queries and events to appropriate rendezvous nodes in the network. This technique can deploy a publish/subscribe application of any data dimensionality on any existing CAN network.

RP Based Publish/Subscribe

Suppose that the data (event) space D is d-dimensional, and CAN space is k-dimensional. The idea of RP is to project a d-dimensional data object in the original data space onto the CAN space to get a new data object in k dimensions such that the distance between two data objects after the projection remains within a small constant factor of the original distance.

Let $\{u_1,\ldots,u_k\}$ be k random vectors, each being a d-dimensional orthonormal vector. Consider a subscription query $Q = (s,r)$, which is a sphere centered at point $s \in D$ with radius r, asking for all events that are within a distance r from the sample event s. Projecting this sphere on the k random vectors, we obtain a k-dimensional hyper-rectangle $u(Q) = u_1(s, r) \times u_2(s, r) \times \ldots \times u_k(s, r)$, where each edge $u_i(s, r)$ is the interval $[<s, u_i> - r, <s, u_i> + r]$ (here, $<.,.>$ denotes the inner product of two vectors). The center of this rectangle is the point $center_u(Q) = (<s, u_1> , <s, u_2>, \ldots, <s, u_k>)$.

A strategy for query subscription is to store the query q in the nodes V_i whose CAN-zone, denoted by $zone(V_i)$, intersects Q's CAN-projection (i.e., $zone(V_i) \cap u(Q) \neq \varnothing$). This strategy can be implemented as follows:

1) Use CAN routing to send Q to the node V_Q such that $zone(V_Q)$ contains $center_u(Q)$.
2) Each node V that receives Q forwards this query to each neighbor node V' such that $zone(V') \cap u(Q) \neq \varnothing$; node V' follows the same procedure as V does.

When an event x becomes available, using CAN routing, we advertise x to the node V_x such that $zone(V_x)$ contains the point $u(x) = (<x, u_1> , <x, u_2>, \ldots, <x, u_k>)$. It is obvious that if x satisfies a query Q, then $u(x)$ must be a point inside rectangle $u(Q)$. Consequently, $zone(V_x)$ must intersect $u(Q)$ and the query Q must have been stored at node V_x. Thus, given an event x and subscription query Q, that match each other, they are guaranteed to always find each other at some rendezvous node.

The above strategy allows for quick and cost-effective event notification because each event is advertised to only one node. However, there might incur a large amount of subscription replicas in the network. Since subscription queries are likely to overlap, we should take advantage of this property to minimize their replication in the network. Observe that if query Q' covers query Q, it must be true that $u(Q')$ covers $u(Q)$. Therefore, if a new query Q is covered by an existing query Q', the nodes that the new query is mapped to must have already stored the existing query. Because those events that satisfy Q' will be returned to notify Q' anyway, which can be filtered to match Q, there is no need to replicate query Q further. Based on this observation, a more efficient

strategy is proposed in [Tran & Nguyen (2008)], which differs from the aforementioned strategy in that a query Q is not replicated at a node if this node has stored an existing query Q' such that $u(Q')$ contains $u(Q)$. The query subscription for a query Q works in detail as follows:

1) Use CAN routing to send Q to the node V_Q such that $zone(V_Q)$ contains $center_u(Q)$.
2) If there does not exist a query Q' currently stored at node V_Q such that $u(Q')$ covers $u(Q)$:
    a) Node V_Q will store Q and is called the *home node* of Q, denoted by $home(Q)$.
    b) Forward Q to adjacent nodes whose zone intersects Q. Such a node V' performs the following steps:
        i. Store Q at V'
        ii. For each existing query Q' stored at V' such that $u(Q)$ covers $u(Q')$, remove query Q' from node V'
        iii. Forward Q to adjacent nodes whose zone intersects Q as in step b)
3) If node V_Q stores a query Q' satisfying the condition of 2), query Q will be routed to and stored at node $home(Q')$.

To illustrate this revised strategy, we consider the following scenario. In Figure 4, queries q_1, q_2, q_3 are submitted into the network at times in that order. Query q_1 is submitted first, which intersects the zones of nodes 7, 13, 11, 9, 10, 8, and 2. Therefore, q_1 is stored at these nodes and the home node of q1 is node 7 (because the projection center $center_u(q_1)$ lies in the zone of node 7). When q_2 is submitted, it is sent to node 10 whose zone contains the projection center of q_2. Because node 10 already stores query q_1 and $u(q_1)$ covers $u(q_2)$, query q_2 will be stored at the home node of query q_1, i.e., node 7 only; hence, a significant reduction in subscription load. When query q_3 is submitted, because its projection is not covered by any other's, it is stored at nodes 2 and 3 because their zones intersect $u(q_3)$, node 2 serving as the home node of query q_3.

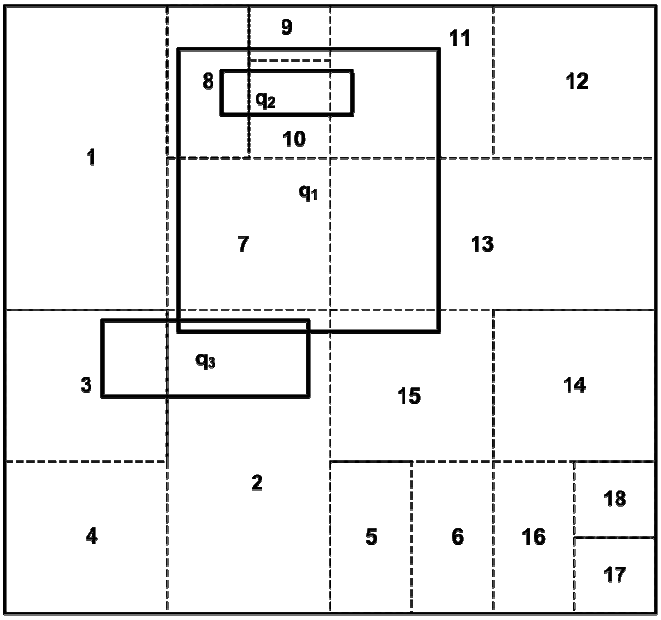

Fig. 4. Query subscription and event notification.

When an event *x* becomes available, the notification procedure works as follows:

1) Use CAN routing to advertise *x* to the node V_x such as $u(x) \in zone(V_x)$
2) For each query *Q* stored at node V_x such that $u(x) \in u(Q)$, forward *x* to node *home(Q)* – the home node of *Q*. At each home node *V* that receives *x*, search for and notify all the queries that match *x* and that call *V* home.

For example, continuing the previous illustration with Figure 6, suppose that an event *x* satisfying query q_2 is available such that *u(x)* lies in the zone of node 11. An advertisement will be sent to node 11, where it is found that $u(x) \in u(q_1)$. Consequently, the advertisement of *x* is routed to the home node 7 of query q_1, where we will find query q_2.

GOSSIP-BASED PUBLISH/SUBSCRIBE TECHNIQUES

Structure-based techniques are capable to grow the network and adaptable to network dynamics, but they incur an additional cost to maintain the overlay. In addition, some of these techniques require different overlays for different publish/subscribe applications. The gossip-based approach [Terpstra et al. (2007)], [Wong & Guha (2008)], [Gkantsidis et al. (2006)] can work with any unstructured networks and thus does not have these limitations.

The word "gossip" gives the intuition of its use in publish/subscribe techniques: subscriber nodes and publisher nodes find each other by gossiping with their respective neighbors. An example is to use random walks. A query follows a random walk in the network and is replicated at each node visited. Another random walk is used to publish an

event. If these two random walks are long enough, they have a good chance to intersect. As such, there is a high probability that an event will reach every query and thus find all the matching queries. The tradeoff, however, is due to the high cost (communication, storage) to disseminate each query or event. The main question, therefore, is how to design a gossiping mechanism that offers the best balance between efficiency and effectiveness. BubbleStorm [Terpstra et al. (2007)] is a recent technique aimed to address this challenge.

BubbleStorm

BubbleStorm replicates each query in a given number of nodes within a number of hops from the source. This set of nodes called a *query bubble*. Similarly, an event is also replicated in an *event bubble*. These bubbles need to be large enough to share at least a rendezvous node where the query and event can find each other.

To reach a given bubble size, nodes chosen to disseminate a query (or event) should be independent and the bubbles should be formed without cycles. For this purpose, a random multi-graph is proposed, in which self-loops and double-edges are allowed and a node's degree is proportional to its bandwidth. Figure 5(a) illustrates a random multi-graph of 7 nodes. The degree of node 1 is 3, of node 7 is 4, etc.

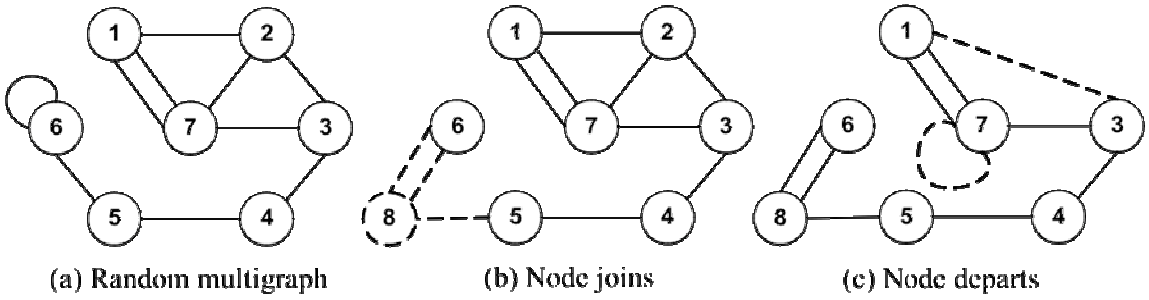

Fig. 5. Random multi-graph example

When a new node joins the network, it firstly contacts the bootstrap node and then uses a random walk to find a proper edge. The chosen edge will be split and the node is inserted between the vertices connecting this edge. Suppose that node 8 contacts its bootstrap node, which is node 3, and eventually found an edge between node 5 and 6. The creation of new links is illustrated in Figure 5(b). If a node leaves, its neighbors need to adjust their connections to maintain degrees. For example, in Figure 5(c), node 2 departs from the network and as a result new links (2-3, 7-7) are created to maintain the degrees of nodes 2, 3, and 7.

The communication primitive used by BubbleStorm to replicate queries and events in the bubbles is called BubbleCast. BubbleCast defines a split factor f which controls how many neighbor nodes should receive a forwarded query or event. Suppose that a query, starting at a node V needs to be replicated at n_q nodes. Node V will store a replica of the query and forwards the query to f neighbor nodes, chosen randomly. Each such a

neighbor node is responsible for $(n_q - 1)/f$ remaining replicas of the query; the same replication procedure as at node *V* is applied. The dissemination of an event is similar.

Figure 6 provides an example of how BubbleCast works. In this example, suppose the number of replicas for a query is 17, and the split factor is 2. Each time, the number of replicas is reduced by 1 and then divided by split factor 2. From the initial subscriber, the number of remaining replicas is 17-1 = 16, divided by split factor of 2. Each new forwarder will continue with 8 as the number of replicas, including itself. The process continues until the number of replicas becomes 1. So the procedure will end up with totally 17 replicas as the desired size of the bubble. This process is the hybrid of random walks and flooding.

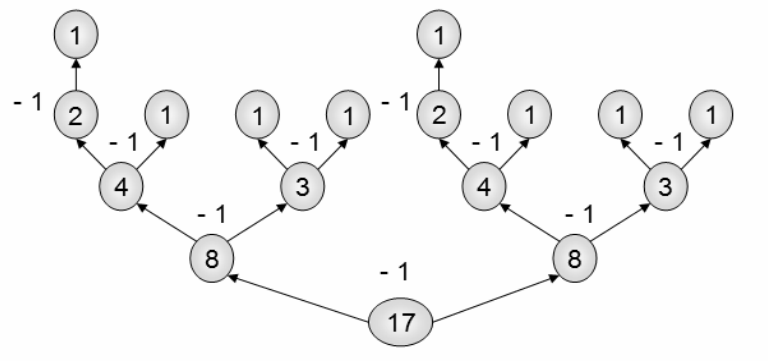

Fig. 6. How BubbleCast works.

Choosing a desirable bubble size is a key factor of BubbleStorm. It is proposed that if a query is replicated at $n_q = O(sqrt(n))$ nodes and an event is disseminated to $c^2 n/n_q$ nodes, the probability that the query and the event can find each other at an intersection of their bubbles is $1\text{-}exp(c^2)$, where *c* is the certainty factor (e.g., $c = 3$ → probability = 99.99%).

HYBRID PUBLISH/SUBSCRIBE TECHNIQUES

The gossip-based approach offers more flexibility than the structure-based approach because the former requires no overlay structure in advance and allows queries and events to be expressed in any format. On the other hand, as queries and events are spread randomly in the network, there is no guarantee that they will meet each other. To make any gossip-based system effective, we need to publicize the queries and events widely in the network, leading to the trade-off between efficiency and effectiveness. In addition to that, the guarantee that every query meets every event is unnecessarily strong. Indeed, we only need to guarantee that ever query meets every *matching* event.

In this section, we introduce Pub-2-Sub [Tran & Pham (2010)], a technique combining the strengths of both structure-based and gossip-based approaches. Pub-2-Sub can be considered a hybrid approach that can work in any unstructured P2P network, yet having the efficiency of structure-based techniques. It allows any number of independent publish/subscribe applications to run simultaneously on the same underlying P2P network. Because Pub-2-Sub is based on directed routing, it has the potential to be more

efficient than the gossip-based approach. Pub-2-Sub results in lower storage and communication costs in comparison to BubbleStorm. In terms of computation cost, Pub-2-Sub requires only a node in the network that needs to evaluate its local queries to find those matching a given published event. The technique also incurs small notification delay and is robust under network failures.

Pub-2-Sub

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 events, in which routing is based on the virtual addresses of the nodes.

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 $VA(S_i : S^*)$. To start the virtualization, node S^* assigns itself $VA(S^*: S^*) = \varnothing$ and sends a message inviting its neighbor nodes to join $INSTANCE(S^*)$. A neighbor S_i ignores this invitation if already part of the instance. 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 $VA(S^*: S^*) + '0^*1'$ 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 continues repeatedly. In general, the rule to compute the VA for a node S_j that accepts an invitation from a node S_i is that $VA(S_j : S^*)$ is always the shortest string of the form $VA(S_i : S^*) + '0^*1'$ 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, Figure 7 shows the VA-tree with VAs assigned to the nodes as a result of the virtualization procedure initiated by node 1. 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 $VA(3 : 1) = VA(1 : 1) + '01' = '01'$, and node 4 last and given $VA(4 : 1) = VA(1 : 1) + '001' = '001'$. Other nodes are assigned VAs 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 $VA(8 : 1) + '0^*1'$. The other children 16 and 17 already occupy *'0111'* and *'01101'*, therefore the VA of node 18 will be *'011001'*.

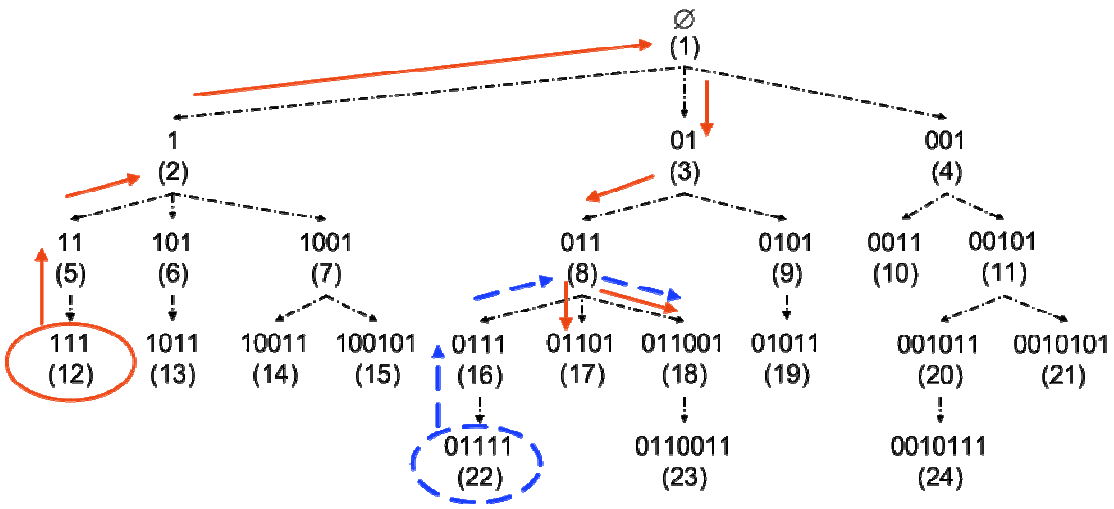

Fig. 7. Pub-2-Sub: Virtualization and Indexing

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 Figure 7, 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.

Indexing

Pub-2-Sub supports publish/subscribe applications that can have any data dimensionality and allows any number of them to run on the network simultaneously, whose dimension can be different from one another. For ease of presentation, we assume for now that events are one-dimensional.

In Pub-2-Sub, an event *x* is expressed 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'}*. Supposing that every node has been assigned a VA as a result of a virtualization procedure initiated by a node *S**, we propose 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 routing of queries and events to their destination nodes is facilitated by the VA structure based on the matching between the node VAs and query/event content.

Figure 7 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 with *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 Figure 11). 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. The storage and communication costs for a query’s subscription depend on its range; the wider the range, the larger costs.

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, $VA(S_i : S_1^*)$, $VA(S_i : S_2^*)$, ..., and $VA(S_i : S_m^*)$, 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 may likely 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.

Multi-Dimensionality

In the description of Pub-2-Sub we have expressed an event as an one-dimensional *k*-bit binary string and a query as an one-dimensional 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 an one-dimensional *k*-bit binary string $x^f = f(x)$ and a *d*-dimensional range *Q* to an one-dimensional 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 [Lawder & King (2000)]. 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 one-dimensional space with a high probability.

Then, to subscribe a query Q we use the hash interval Q^f. Similarly, to publish an event x we route it to the designated node of x^f. 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).

CONCLUSIONS

The publish/subscribe paradigm represents a large class of applications in P2P networks. Despite many existing techniques to implement this paradigm in P2P networks, there remains much room for future research. For a large-scale P2P network where broadcast-based and gossip-based approaches may not be the best fit, the routing design should be driven by the content of the message being routed so as to limit the scope of propagation. On the other hand, content-based routing if enabled by a structured overlay might incur considerable costs to maintain the structure. Since P2P networks may be of different types (small vs. large, unstructured vs. structured, static vs. dynamic) and the application to deploy may also have its own characteristic (low vs. high query rate, low vs. high event rate, subject-based vs. content-based, etc.), it is difficult to choose a publish/subscribe design that works well in every practical case. Thus, rather than trying to find a "perfect" design universally, it would be better to categorize the networks and applications into similarity-based groups and design the "best" technique for each group. For example, for P2P-based cooperative networks in which the nodes are supposed to be functional most of the time and failures should not happen too often, a technique like Pub-2-Sub presented in this chapter is a good design candidate. Data grid networks and institutional collaborative networks can take full advantage of this technique.

It is also important to develop a publish/subscribe middleware package that provides a set of common services to most publish/subscribe network/applications no matter their categories, and another set of services each customized toward a specific category. This middleware should provide convenient tools for the middleware designer to add new service components to the existing architecture, such as a new language for query and event description and a new implementation for routing, data aggregation, or an event matching algorithm. It should also give the application developer freedom and a convenient API to choose the publish/subscribe service configuration that is best for the context of the deployment. Middleware development for publish/subscribe applications in P2P networks remains ad hoc and isolated. It should be given high priority in the future research towards publish/subscribe services in P2P networks.

REFERENCES

Aekaterinidis, I., & Triantafillou, P. (2005) "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.

Bianchi, S., Felber, P., & Gradinariu, M. (2007) "Content-based publish/subscribe using distributed r-trees," in *Euro-Par*, 2007, pp. 537–548.

Castro, M., Druschel, P., Kermarrec, A., & Rowstron, A. (2002) "SCRIBE: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in communications (JSAC)*, vol. 20, no. 8, pp. 1489–1499, 2002.

Freenet. http://en.wikipedia.org/wiki/Freenet

Gkantsidis, C., Mihail, M., & Saberi, A. (2006) "Random walks in peer-to-peer networks: algorithms and evaluation," *Perform. Eval.*, vol. 63, no. 3, pp.241–263, 2006.

Gnutella. http://gnutella.wego.com

Gupta, A., Sahin, O. D., Agrawal, D., & Abbadi, A. E. (2004) "Meghdoot: content-based publish/subscribe over p2p networks," in *Middleware '04:Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. New York, NY, USA: Springer-Verlag New York, Inc.,2004, pp. 254–273.

Lawder, J. K., & King, P. J. H. (2000) "Using space-filling curves for multidimensional indexing," in *BNCOD 17: Proceedings of the 17th British National Conference on Databases*. London, UK: Springer-Verlag, 2000, pp. 20–35.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001) "A scalable content addressable network," in *ACM SIGCOMM*, San Diego, CA, August 2001, pp. 161–172.

Rowstron, A., & Druschel, P. (2001) "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, November 2001, pp. 329–350.

Shalaby, N., & Zinky, J. (2007) "Towards an architecture for extreme p2p applications," in *Parallel and Distributed Computing and Systems Conference (PDCS)*, Cambridge, MA, November 2007.

Stoica, I., Morris, R., Karger, D., Kaashock, M., & Balakrishman, H. (2001) "Chord: A scalable peer-to-peer lookup protocol for internet applications," in *ACM SIGCOMM*, San Diego, CA, August 2001, pp. 149–160.

Teranishi, Y., Tanaka, H., Ishi, Y., & Yoshida, M. (2008) "A geographical observation system based on p2p agents," in *PERCOM '08: Proceedings of the 2008 Sixth Annual*

IEEE International Conference on Pervasive Computing and Communications. Washington, DC, USA: IEEE Computer Society, 2008, pp. 615–620.

Terpstra, W. W., Behnel, S., Fiege, L., Zeidler, A., & Buchmann, A. P. (2003) "A peer-to-peer approach to content-based publish/subscribe," in *DEBS '03:Proceedings of the 2nd international workshop on Distributed eventbased systems*. New York, NY, USA: ACM Press, 2003, pp. 1–8.

Terpstra, W. W., Kangasharju, J., Leng, C., & Buchmann, A. P. (2007) "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*. New York, NY, USA: ACM, 2007, pp. 49–60.

Tran, D. A., & Nguyen, T. (2008) "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)*. Virgin Island, USA: IEEE Press, August 2008.

Tran, D. A., & Pham, C. (2010) "Enabling content-based publish/subscribe services in cooperative P2P networks," in *Journal of Computer Networks*. Elsevier, February 2010.

Voulgaris, S., Rivire, E., Kermarrec, A.-M., & van Steen, M. (2006) "Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks," in *5th Int'l Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

Wong, B., & Guha, S. (2008) "Quasar: A Probabilistic Publish-Subscribe System for Social Networks," in *Proceedings of The 7th International Workshop on Peer-to-Peer Systems (IPTPS '08)*, Tampa Bay, FL, February 2008.

Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., & Kubiatowicz, J. (2004) "Tapestry: A resilient global-scale overlay for service deployment," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, January 2004.

Cuong (Charlie) Pham is a PhD student in the Department of Computer Science at the University of Massachusetts at Boston and a research member of the Network Information Systems Laboratory (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 was a research intern working on distributed storage networks at EMC (USA) during summer of 2010. He received a Student Travel Award from the NSF and a Research Excellence Award from the Department of Computer Science (UMass Boston), both in 2009.

Duc A. Tran is an Assistant Professor in the Department of Computer Science at the UMass Boston, where he leads the Network Information Systems Laboratory (NISLab). He received a PhD in CS degree from the University of Central Florida (Orlando, Florida). Dr. Tran's interests are focused on data management and networking designs for

decentralized networks. His work has resulted in research grants from the National Science Foundation and two Best Papers (ICCCN 2008, DaWak 1999). Dr. Tran has served as a Review Panelist for the NSF, Editor for the Journal on Parallel, Emergent, and Distributed Systems (2010-date), Guest-Editor for the Journal on Pervasive Computing and Communications (2009), TPC Co-Chair for CCNet 2010, GridPeer (2009, 2010, 2011), and IRSN 2009, TPC Vice-Chair for AINA 2007, and TPC member for 40+ international conferences.