Meghdoot: Content-Based Publish/Subscribe over P2P Networks*

Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science
University of California at Santa Barbara
{abhishek,odsahin,agrawal,amr}@cs.ucsb.edu

Abstract. Publish/Subscribe systems have become a prevalent model for delivering data from producers (publishers) to consumers (subscribers) distributed across wide-area networks while decoupling the publishers and the subscribers from each other. In this paper we present Meghdoot, which adapts content-based publish/subscribe systems to Distributed Hash Table based P2P networks in order to provide scalable content delivery mechanisms while maintaining the decoupling between the publishers and the subscribers. Meghdoot is designed to adapt to highly skewed data sets, which is typical of real applications. The experimental results demonstrate that Meghdoot balances the load among the peers and the design scales well with increasing number of peers, subscriptions and events.

1 Introduction

Publish/Subscribe systems are used to deliver data events from *publishers* (data/event producers) to *subscribers* (data/event consumers) in a decoupled fashion. Publishers can be completely unaware of the subscribers and simply introduce data events into the system. A data event specifies values for a set of attributes associated with the event. Subscribers can register their interests with the system in the form of *subscriptions* which act as filters that are used by the system to deliver relevant events to the subscribers. Content-based subscriptions specify the subscription based on attribute properties of the data events. The publish/subscribe system, or equivalently the publish/subscribe middleware, manages the subscriptions and delivers the events to the matching subscriptions.

Publish/subscribe systems arise in many applications including online stock quotes, Internet games, and sensor networks. In the stock quote application, for example, the events are generated by various stock exchanges where trading occurs. The events contain information about the open, close, low, and high values of companies' stocks at various times. The subscribers are clients interested in trading, and they are usually interested in the values of the stocks they trade. Another interesting example is IBM's web service for information on events in

* This research was supported in parts by NSF grants EIA00-80134, IIS-0220152, and IIS-0223022.

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 254–273, 2004.

the 2000 Olympics at Sydney. Users could query events in sports of their interest or events relating to their countries by contacting the web service. According to IBM [13], there were 11.3 billion hits to the web server. This number is expected to grow higher in upcoming events. Users are interested in a notification service that can allow them to track events of interest. The information for such events can be distributed to users by utilizing a distributed publish/subscribe system.

Current solutions are either centralized or distributed. Centralized solutions based on a DBMS use triggers [11, 6], which have an inherent scalability problem as the number of events in the system increase. Specialized data structures have been proposed to overcome these scalability problems [7]. However, to ensure efficient processing, restrictions are placed on the types of subscriptions that the system can support. Distributed solutions are usually restricted to specific subject-based subscriptions [5, 31, 19] and hence do not support general content-based subscriptions. Alternatively, routing trees [4, 2] are used to support multicast to prevent communication bottlenecks.

Peer-to-peer systems are particularly attractive for supporting publish/ subscribe systems since they are flexible, modular, and scalable. In such systems, peers are used both to store subscriptions and to route events to other peers with relevant subscriptions. Peer-to-peer systems are very scalable since additional peers can contribute their machines, thus increasing the computational and storage resources in the system. Traditionally, peer-to-peer systems were designed to answer exact match queries. In this paper we present Meghdoot[1], a P2P based publish/subscribe system. Meghdoot uses a structured distributed hash table [1] to determine where subscriptions are stored and how to route the events to the subscriptions. Meghdoot is a scalable architecture for publish/subscribe systems that applies a flexible model of content-based publish/subscribe systems to structured P2P systems.

A particular challenge in peer-to-peer systems involves ensuring a uniform distribution of load among the different peers in the system. Traditional peer-to-peer systems are oblivious to the content of the data and hence use a uniform hash function to distribute the data among the different peers. However, in a content-based publish/subscribe system, we distribute the subscriptions and events based on their content. Most real world datasets tend to be skewed and hence will cause a non-uniform distribution of load on the peers. One important innovation of Meghdoot is the alternative methods it uses to balance the load among peers. Our experimental results clearly demonstrate that even when using highly skewed real world datasets, the system ensures that no peer is unduly loaded.

The rest of the paper is structured as follows. Section 2 presents a survey of related work. Section 3 presents the basic design of our proposed approach. This is followed by Section 4 which presents strategies for employing peers in the system in a load balanced manner, and presents some optimizations. Section 5 presents the experimental setup and results. Section 6 concludes the paper.

[1] The name Meghdoot originates from an ancient epic where clouds were used as messengers.

2 Related Work

Publish-subscribe systems are designed for disseminating information (events) to a subset of the clients (subscribers) who are actually interested in this information. The designs for publish/subscribe systems can be classified into two categories: *Subject-based* and *Content-based*. Subject-based publish/subscribe systems assign each event to one of a set of pre-defined subjects (also referred to as topics, channels, or groups). The events themselves specify the topics that are relevant to the event. A client subscribes to a set of subjects it is interested in and is notified of all the events that are associated with these subjects. Examples of these systems include research proposals ISIS [3], iBus [16] and commercial products Tibco [26], and Vitria [28].

Content-based publish/subscribe systems allow more complex subscriptions by enabling restrictions on the event content. A subscriber can specify multiple predicates as a subscription and only those events whose content satisfies all the predicates are notified to the subscriber. Subscriptions in these systems are more expressive, but the systems are harder to implement. Examples of distributed content-based systems include Elvin [22], Siena [4], and Gryphon [2]. Elvin uses a central server which stores all the subscriptions and evaluates the subscriptions affected by the events. Fabret et al. [7] proposed novel data structures and application-specific caching policies and query processing in centralized systems to support high rates of subscriptions and events in a content-based system. For scalability purposes, subscriptions in this system must contain at least one equality predicate. Other centralized approaches have been proposed for scalable matching of predicates in the context of database trigger processing [11] and continuous queries [6]. Siena and Gryphon are distributed systems, in which a network of broker nodes is created and the events are distributed within the network.

P2P systems have emerged as a popular technique for exchanging information among a set of distributed peers. Initial approaches used a centralized index and/or flooding for locating objects in the system (e.g. Gnutella [9], Napster [15], KaZaA [14]). Advanced P2P systems provide more efficient lookups using structure in the logical overlay network formed by the peers. They implement a hash table functionality distributed over the peers, and are referred to as *Distributed Hash Tables* (DHT's). CAN [18], Chord [23], Pastry [20], and Tapestry [30] are different examples of DHT's. Most P2P systems were designed for locating information based on exact names, e.g. names of files. However, recently several proposals have been made to extend P2P functionality to more complex queries, e.g., range queries [10, 21], joins [12], SQL [27], XML [8], etc.

Several application level multicast systems have been proposed employing an underlying DHT, and can be easily adapted for subject-based publish/subscribe. These systems inherit the scalability and fault tolerance properties from the underlying DHT structure. Scribe [5] is built on top of Pastry [20]. Scribe assigns a unique groupId to each topic and the node whose nodeId is numerically closest to the groupId becomes the rendezvous point for this topic. For each topic, a multicast tree, that is rooted at the rendezvous point, is created by com-

bining the paths from each subscriber to the rendezvous point. The messages (events) associated with the group (subject) are disseminated along the corresponding multicast tree starting from the root. A Pastry [20] based P2P overlay is used in [17] to provide support for a type based publish/subscribe system using similar rendezvous mechanism as Scribe [5]. The P2P overlay is also used by [17] for installing content based filters close to the publishers. Bayeux [31] uses Tapestry [30] as the underlying structure. Similar to Scribe, each group is associated with a root node based on its unique ID and a multicast tree rooted at that node is used for data dissemination. CAN-based multicast [19] is an application level multicast system based on CAN. Unlike the approaches above, CAN multicast does not build a distribution tree for each group. Instead, the members of a group form a separate group specific CAN and the multicasting is achieved by flooding over this separate CAN.

Our work implements a content-based publish-subscribe system over a DHT based on CAN [18]. There are several similar efforts [25, 24]. Terpstra et al. [25] partition the event space among the peers in the system. Chord [23] is used to broadcast events and subscriptions to all nodes in the system. These broadcasts are attenuated by filters installed at peer nodes due to the subscriptions stored in the system. This approach may require all peers in the system to be contacted to install a subscription. Unlike Terpstra et al., in our solution subscriptions are installed by routing through $O(dN^{\frac{1}{d}})$ peers, where N is the number of peers in the system and d is the dimensionality of the logical DHT space. Tam et al. [24] extend an existing subject-based system (Scribe [5]). Ordered collections of a set of selected attributes are used for indexing subscriptions into a subject-based publish/subscribe system. Thus a subscription can be submitted into the system only if it specifies all attribute values for at least one of the indices. The domains of attribute values are partitioned into intervals. If a subscription specifies a range over some attribute, it is decomposed into subparts according to the domain intervals. Meghdoot is a content-based publish/subscribe system that imposes no restrictions on the subscriptions or the events. It provides scalability by leveraging the P2P design, thus allowing flexible addition of peers into the system to match the demand. It uses innovative load distribution techniques to maintain the load balance in presence of highly skewed real world data.

3 Meghdoot Design

In this section, we start with a description of the schema and the representation of the events and the subscriptions in the system. Next, we describe the storage model for subscriptions and the event delivery mechanism.

3.1 Publish/Subscribe Model in Meghdoot

We consider a content-based publish/subscribe system with multiple attributes. The model and definitions are based on the model proposed by Fabret et al. [7]. The schema for the system can be described as: $\mathbb{S} = \{A_1, A_2, \ldots, A_n\}$, where each A_i corresponds to an attribute. Each attribute has a name, type and domain, and can be described by the tuple {`Name: Type, Min, Max`}. The attributes

are identified by their unique names. The data types we consider are integer, floating point and character strings. The values `Min` and `Max` describe the range of domain values taken by the given attribute. All the peers participating in the publish/subscribe system use the same schema $\mathbb{S}$.

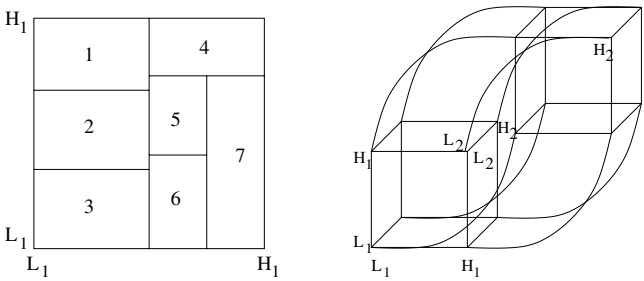
A *subscription* is a conjunction of predicates over one or more attributes. Each predicate specifies a constant value (using $=$) or a range (using $<, >, \leq, \geq$) for an attribute. If a subscription needs to specify multiple predicates over the same attribute, we can model such a subscription as a combination of multiple subscriptions, each of which specifies one continuous range over the attribute. For simplicity of presentation, henceforth, we assume each subscription specifies a continuous range over an attribute. An example subscription is $S = (A_1 \geq v_1) \wedge (v_2 \leq A_3 \leq v_3)$. An *event* is a set of equalities over the attributes in the schema. Therefore, an event can be represented as $e = \{A_1 = c_1, A_2 = c_2, \ldots, A_n = c_n\}$. In general, events may specify values for a subset of the attributes.

An event e matches a subscription S if each predicate of S is satisfied by the value of the corresponding attribute specified by the event e. The publish/subscribe system is required to store the subscriptions specified by the users and given an event, find all subscriptions matching the event and deliver the event to the subscribers.

The schema based clustering algorithms in [7] require that there be at least one equality predicate in each subscription. Applications may be interested in range predicates over all attributes specified in a subscription. For example, a subscription in the case of a stock quotes application can ask for all quotes where the volume of trade is greater than 100,000 on any day, without specifying any equality predicate. An event at the end of a day needs to specify values for stock name, its open, high, low and close values, and the volume of trade in that day. Our model is general and does not restrict the subscriptions and allows them to specify ranges over all attributes or a subset.

3.2 Logical Space Construction

In this section, we describe the construction of the logical space used for maintaining the distributed hash table. Given a schema $\mathbb{S} = \{A_1, A_2, \ldots, A_n\}$ with n attributes, we create a cartesian space with $2n$ dimensions. The mapping for the construction of the logical space as described below has been adapted from Sahin et al. [21]. Attribute A_i with domain range $[L_i, H_i]$ corresponds to dimensions $2i - 1$ and $2i$ of the cartesian space. Intuitively, the predicates of a subscription specify ranges of interest over the attributes, and the ranges are represented by a point in the logical space. The start of the range over the ith attribute is mapped to dimension $2i - 1$ corresponding to attribute A_i, and the end of the range is mapped to the dimension $2i$. Therefore the domain of the $2i - 1$ and $2i$ axes in the cartesian space is bounded by $[L_i, H_i]$. This logical space is partitioned among the peers present in the system, and each peer is responsible for one of the partitions. The partitions are referred to as *zones*, and if a peer is responsible for a partition, we say the peer *owns* the zone.



(a) 2d-cartesian space (b) 4d-cartesian space

Fig. 1. Logical space construction.

The peers maintain a multidimensional Distributed Hash Table (DHT) as described in CAN [18]. Each peer maintains information about the coordinates of its own zone. In addition, the peers store coordinate information of their neighboring zones as well as the IP addresses of the peers owning those zones. This information is used for the purposes of routing in the overlay network formed by the peers.

Figure 1 shows examples of the cartesian space. Figure 1(a) is the logical cartesian space for the case when the schema has only one attribute. The bounds of both axes in this case correspond to the bounds $[L_1, H_1]$ of the attribute. The rectangular regions form a partitioning of the space. Each rectangle is a zone and is owned by a peer in the system. Figure 1(b) is an example logical cartesian space when the schema has two attributes. The first two dimensions in the figure are bounded by the domain $[L_1, H_1]$ of the first attribute, whereas the next two dimensions are bounded by the domain $[L_2, H_2]$ of the second attribute.

3.3 Subscription Installation

A user can specify a subscription S by defining the ranges or values over one or more attributes in the schema. A subscription S can be expressed in the following format:
$S = (l_1 \leq A_1 \leq h_1) \wedge (l_2 \leq A_2 \leq h_2) \wedge \ldots \wedge (l_n \leq A_n \leq h_n)$.
If the subscription is interested in a specific value v of an attribute A_i then both l_i and h_i are set to v. If the subscription S does not specify any range over an attribute A_i then l_i and h_i are considered to be the boundaries L_i and H_i of the domain of A_i. The subscription S is mapped to the point $\langle l_1, h_1, l_2, h_2, \ldots, l_n, h_n \rangle$ in the $2n$-dimensional space which is referred to as the *subscription point*. Note that all subscriptions are stored in the upper left side of the diagonal hyperplane. This is due to the fact that the coordinate value of the $(2i-1)$th dimension is smaller than or equal to the $(2i)$th dimension for a subscription point since they correspond to the start and end values of the range over the ith attribute. The zones in the bottom right of the diagonal hyperplane are primarily used for routing purposes. Later, we utilize these zones for fault tolerance.

When a user wishes to subscribe for some events, the user submits the subscription to a peer in the system. The origin peer P_o maps the subscription to its corresponding subscription point in the $2n$-dimensional space. The peer whose *zone* contains this point is referred to as the target peer P_t. P_o needs to route the subscription to the peer P_t. In order to route the subscription, P_o selects one of its neighbors, which has the closest Euclidean distance in the $2n$-dimensional space to the subscription point, and forwards the subscription to it. This process of forwarding is repeated until the subscription reaches the peer P_t. This process of routing the subscription to the peer responsible for managing it takes $O(dN^{\frac{1}{d}})$ overlay hops [18] on average, where N is the number of peers in the system and d is the dimensionality of the cartesian space. When P_t receives the subscription, it stores the subscription along with an identifier (e.g. IP address, user name, e-mail, etc.). Figure 2 shows an example routing path for a subscription in the case of a single attribute schema. The subscription is finally stored in the target zone and the subscription point is marked S.

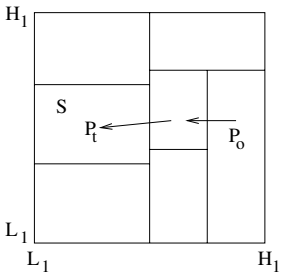


Fig. 2. Routing a subscription to its destination for installation.

3.4 Event Delivery

When an event is introduced into the system, Meghdoot is required to find all the matching subscriptions installed in the system, and deliver the event to the subscribers. Consider an event $e = \{A_1 = c_1, A_2 = c_2, \ldots, A_n = c_n\}$. The event e is mapped to the point $\langle c_{11}, c_{12}, c_{21}, c_{22}, \ldots, c_{n1}, c_{n2}\rangle$ in the $2n$-dimensional space, and is referred to as an *event point*. If the event specifies a value v for the ith attribute then $c_{i1} = c_{i2} = v$, otherwise $c_{i1} = L_i$ and $c_{i2} = H_i$. Note that the event points lie on the diagonal hyperplane of the space if all the attribute values are specified. A subscription $S = (l_1 \leq A_1 \leq h_1) \wedge (l_2 \leq A_2 \leq h_2) \wedge \ldots \wedge (l_n \leq A_n \leq h_n)$ is affected by the event e if the following property holds:

$$\forall i \in \{1, 2, \ldots, n\} \;\; l_i \leq c_{i1} \wedge c_{i2} \leq h_i$$

The shaded area in Figure 3 shows the region of event points in a $2d$ cartesian space corresponding to a single attribute schema that can affect a subscription S, because all the event points in the shaded region will satisfy the above property.

The shaded region in Figure 4 shows the region of subscription points of all subscriptions affected by an event e. In the $2n$-dimensional space, event e mapping to the point $\langle c_{11}, c_{12}, c_{21}, c_{22}, \ldots, c_{n1}, c_{n2}\rangle$ affects the hyper rectangular region defined by the following points:

$$\langle L_1, c_{11}, L_2, c_{21}, \ldots, L_n, c_{n1}\rangle \text{ and } \langle c_{12}, H_1, c_{22}, H_2, \ldots, c_{n2}, H_n\rangle.$$

When an event is introduced in the system at a peer P_o, it maps the event to the corresponding *event point* and routes the event to the peer P_t which contains the event point. Figure 4 shows the routing path of an event e from P_o to P_t and the region of affected subscriptions of e in a $2d$ space. The event is then propagated starting from P_t to all peers which are in the region affected by

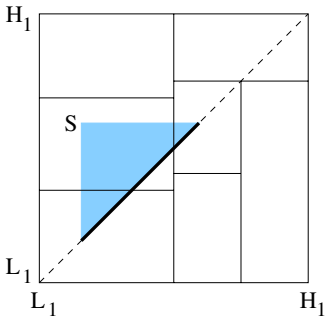


Fig. 3. Region of events affecting a subscription $S = \langle l_1, h_1 \rangle$.

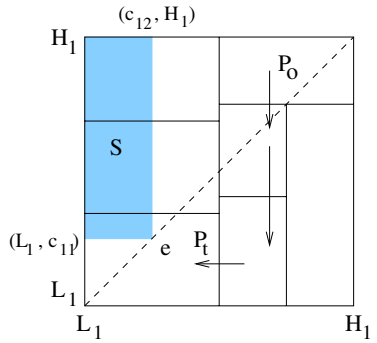


Fig. 4. Region of subscriptions affected by an event $e = \langle c_1, c_1 \rangle$.

the event. P_t sends the event to its immediate neighbors in the affected region, which in turn propagate the event to their neighbors in the affected region. This process repeats until all peers in the affected region have been notified of the event. The basic algorithm for event delivery is given below.

The algorithm **Deliver Event** is initiated at event e's target peer, P_t, which owns zone z. Lines 1-4 check for all affected subscriptions that are stored at zone z and deliver the event to those subscriptions. Predicate $matchedSubscription(S, e)$ is true if the subscription point corresponding to subscription S is contained in the affected region of event e within zone z. Lines 5-8 find all neighbors, n, of zone z that are affected by event e and are in the upper left region of the zone z itself. The predicate $eventRegion(n, e)$ is true if the region of zone n intersects with the affected region of event e. The predicate $upperleftNeighbor(n, z)$ evaluates to true if neighbor n lies in the upper left region of zone z. We use the bottom right point of zone z as the reference point for the upper left region of z to ensure that none of the affected neighbors are missed. For example, in Figure 1(a) zones 1, 2, and 4 are in the upper left of zone 5, but zones 3, 6, and 7 are not. This predicate is required to prevent back propagation of events. In Figure 5, all (dashed and solid) arrows represent the propagation of the event. Predicate $upperleftNeighbor(n, z)$ prevents messages from propagating in the reverse order of the arrows. For example, it prevents zone z_4 from sending a message back to zones z_1, z_2, and z_3.

The event propagation algorithm can be further optimized to prevent the same event from being delivered to a zone by multiple neighbors. Figure 5 shows only zones of interest for illustration and the dashed region is the region of subscription points affected by event e. In the figure, zones z_2 and z_3 do not need to send the event message to the zone z_4 because zone z_1 sends the message to z_4. These messages can be prevented in the following way. When propagating an event to a neighbor z_n, a zone z checks if any of its bottom right neighbors could have already delivered the message to that neighbor, in which case it would not propagate the event to that neighbor. In our implementation we also use this optimization. Dashed arrows in Figure 5 are the duplicate messages that can be avoided by this optimization.

```
Algorithm: Deliver Event(z, e)
/* Executed at zone z for event e. */
 1. for all subscriptions S stored at z
 2.     if (matchedSubscription(S, e))
 3.         notifySubscriber(S, e)
 4. end for
 5. for all neighbors n of z
 6.     if (eventRegion(n, e) ∧
 7.         upperleftNeighbor(n, z))
 8.         DeliverEvent(n, e)
 9. end for
```

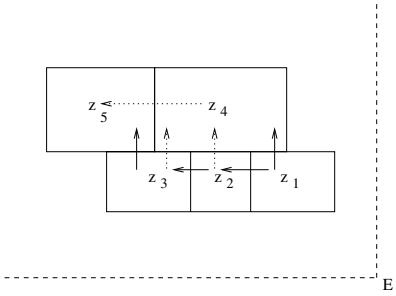


Fig. 5. Preventing repetitive propagation.

3.5 Example Applications

In this section, we present some examples of applications that can be implemented using Meghdoot. As an example consider an application where distributed sensors are used for gathering environmental parameters such as temperature, humidity, illumination, etc. The sensors in various geographical locations send the measured data to their local base stations. Users can specify continuous queries over the data, and these queries are stored at the base stations. This application can be modeled using Meghdoot where the base stations form a P2P network. The user queries are stored as subscriptions on the base stations and the sensor readings can be directed to all the affected queries using the event delivery mechanism.

Meghdoot can also be utilized in critical systems for event monitoring applications, for example power distribution system. The power distribution system consists of *Power Stations* which generate power which is sent to *Transmission Substations*. These transmission substations use high voltage transmission lines to convey power to various *Power Substations* which are located in different geographical areas. The power substations step down the power voltage and distribute it to the residential locations. The power system has sensors that measure the amount of power generated (in MW), transmitted and consumed (in KW) at various points in the system. Sensors also measure the voltage in the transmission lines. Monitoring agents can specify continuous queries that detect anomalies, like sudden drop in voltage over transmission lines or a trip in power generation. This enables early detection of events that can lead to serious problems, e.g. power outages. According to a recent report, the August 14th 2003 blackout in the North East USA and Canada, the cause of the large scale failure was due to lack of timely information about individual failures.

4 Peer Management in Meghdoot

In this section we discuss the mechanisms used by peers to join and leave the system. We start with a description of how a peer joins the system. In the simplest case, a new peer P_n can use the algorithm described in CAN [18] for joining

the system. Peer P_n contacts some existing node P_e in the system and requests P_e to locate a randomly generated point in the logical space. Once the peer P_t whose zone contains the random point is located, P_n submits a join request to P_t. The peer P_t then divides its zone space into two halves and assigns one half to P_n. The two peers then update their neighbor information and also inform the neighbors of the new zone coordinates. The CAN system [18] was originally designed for storing files and multimedia objects with syntactic identifiers. To distribute this information uniformly in the multidimensional identifier space, a uniform hash function was used, thus ensuring a balanced load among the peers. However, in a data-driven environment, as in our approach, the distribution of the data among the peers is content-based. Hence, if a set of events are popular, then the subscription distribution will be skewed in that region of the space. This is due to the fact that Meghdoot uses the content of the subscriptions, rather than a uniform hash function to place data on peers. Furthermore, unlike a standard DHT based P2P system, in a publish/subscribe setting a data event needs to be forwarded to all subscriptions interested in this event. Hence, a direct application of the original CAN join procedure to a publish/subscribe setting can lead to significant load imbalance among the peers. In this section, we study the characteristics of load and develop strategies for load balancing.

4.1 Load Characteristics

The peers in the system need to store subscriptions associated with the subscribers. In addition, they need to propagate the events to all peers in the affected region. Therefore, load on a peer is due to both subscriptions and events, and they have different characteristics, which we described below. Our proposed strategies for admitting new peers in the system exploit these varying characteristics.

Subscription Load. When a user installs a subscription, the subscription is mapped to its corresponding point, and is stored at the peer which owns the zone containing the subscription point. Thus a zone owned by a peer can contain various subscription points. Since the peer owning the zone is responsible for delivering the events it receives to the affected subscriptions it stores, the load on the peer due to subscriptions is proportional to the number of stored subscriptions in the zone. In particular, the load is reduced on a peer, if the subscriptions in its zone are reduced. Therefore, the load due to subscriptions on a peer can be reduced by dividing the spatial extent of a zone so that the number of stored subscriptions is evenly divided with the joining peer. Figure 6 shows an example in 2d space.

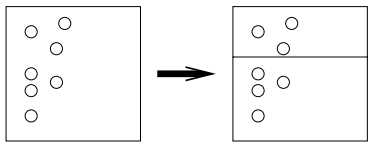
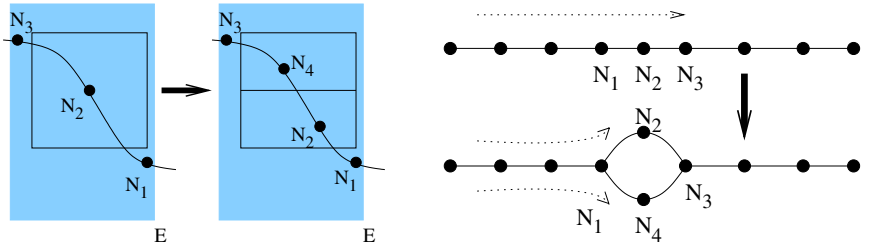


Fig. 6. Splitting a zone divides subscription load.

Event Load. An event generates load because it needs to be propagated to all zones that are in the affected region of the zone. If a zone is loaded because

it falls in the propagation path of too many events, splitting the zone in two will not help, because the zone will still remain in the path of those events. For example, in Figure 7(a), lets say that the zones owned by peer nodes N_1, N_2, and N_3 are in some event propagation path, and node N_2 is overloaded due to event propagation. If a new peer N_4 joins the system and splits the zone owned by N_2, now both N_2 and N_4 are in the propagation path, and hence this splitting does not reduce the load on N_2. In order to reduce the load due to event propagation, we need to create alternate propagation paths and select one of the available paths to propagate each event. Thus, not every event will propagate through the same set of nodes, and overload them. This can be achieved by replicating a zone that is overloaded due to event propagation. In the context of P2P systems partitioning is typically used for load distribution. Our approach is different, in that, we are proposing replication for load distribution.



(a) Splitting does not divide event propagation load.

(b) Replicating a zone divides propagation load.

Fig. 7. Event propagation load.

Figure 7(b) shows how this will be implemented. The straight line on top represents the original situation along a propagation path. The black dots represent the peers owning the zones in the path. Peer N_2 in the center was overloaded due to event propagation. When a new peer N_4 wishes to join the system, it can replicate the exact zone as peer N_2 along with all its subscriptions. The neighbors N_1 and N_3 need to store the IP addresses of both peers N_2 and N_4 associated with the zone coordinates. Effectively the neighbors have two paths to propagate the event through the replicated zone. When a neighbor needs to propagate an event to the zone, it picks one replica peer out of the list of replicas for the zone in a round robin fashion. This will distribute the propagation load on the old peer responsible for the zone.

4.2 Peer Join

When a new peer P_n wishes to join the system, it contacts a known peer P_e in the system. P_e tries to locate a heavily loaded peer in the system. After P_e finds a heavily loaded peer P_h, it forwards P_n to P_h. In order to locate a heavily loaded peer, each peer in the system maintains information about its current

load as well as its neighbors' load when it last heard from them. In addition, each peer maintains an estimated list called *loadedPeerList* of k most heavily loaded peers in the system that they ever heard about. The peers can decide a local value of k depending on available memory.

Peers in the system periodically update their neighbors about their load statistics. The peers also propagate their *loadedPeerList* to their neighbors. A peer merges its *loadedPeerList* with the received list. When required to locate a heavily loaded peer, the load information about the neighbors is utilized by the peers to perform a distributed hill climbing algorithm to locate a local maxima. The initiating peer sends a probe for a heavily loaded node to one of its neighbors that has the heaviest load. The probe is forwarded to the heaviest loaded neighbor by each receiving peer until it reaches a peer which has a load higher than any of its neighbors. This peer is referred to as a local maxima. The new peer is notified of the heavily loaded peer.

When contacted by the new peer, the heavily loaded peer chooses to either split its zone or replicate it based on its load conditions. If the peer is loaded due to event propagation it hands over a copy of its zone to the new peer, thus creating a replica of the zone. The neighbors are informed of the existence of this new replica. If the peer is loaded because it is managing too many subscriptions, it splits its zone in such a way that the two partitions have even distribution of subscriptions between them, and hands over one partition to the new peer.

4.3 Peer Departure or Failure

When a peer wishes to depart from the system, it first checks if there are any replicas for its zone. If one or more replicas exist, it simply informs its replicas as well as the neighbors of its decision to leave, and leaves the system. The neighbors and replicas update their information. If there are no available replicas for a peer leaving the system, it contacts its neighbors and finds a neighbor willing to take over its zone. Then it transfers all its subscriptions to this neighbor and leaves the system.

Since peers periodically send load messages to their neighbors, a neighbor that discovers a failed peer needs to either take up the zone of the failed peer or find some other peer to hand over the zone. In the case of failure, the information about the subscriptions stored at the failed peer can be lost, unless replicas exist due to event load balancing. To avoid this problem, we use the following replication scheme. As described in Section 3.3, all subscriptions are stored in the upper left side of the diagonal hyperplane. We exploit this property of our design for fault tolerance. When installing a subscription at subscription point $\langle l_1, h_1, l_2, h_2, \ldots, l_n, h_n \rangle$, we also store a copy of the subscription at point

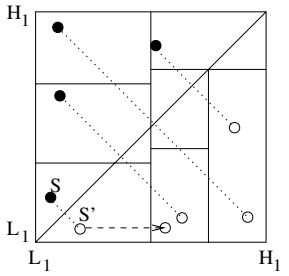


Fig. 8. Replicating subscriptions at mirror image.

$\langle h_1, l_1, h_2, l_2, \ldots, h_n, l_n \rangle$. This point is the reflection of the subscription point in the diagonal hyperplane of the $2n$-dimensional cartesian space. This is illustrated by Figure 8 for a $2d$ space. The filled dots represent the subscription points and the circles are their reflections in the diagonal. Therefore, if a peer fails and there are no replicas for the zone, then the lost subscriptions can be recovered by querying the reflection of the zone's coordinates in the diagonal hyperplane.

If the reflection of a subscription point falls into the same zone, for example subscription S in Figure 8 has its reflection S' in the same zone, we move along the increasing values of all odd dimensions until we find a neighbor zone with higher odd dimensional coordinates, and store the copy at this zone. For example S' is actually stored in the zone to its right in the Figure 8. In case any of the odd dimensional coordinates wrap around before reaching a neighboring zone, we start again with the reflection point and move towards the decreasing even dimensional coordinate values until we find a neighboring zone, and store the copy there. In 2 dimensions, this is equivalent to first moving right and if no neighbor is found, then moving down from the reflection point. If a zone splits, it has to accordingly move the reflection subscriptions to the left or top neighbors.

5 Experimental Evaluation

In this section we evaluate Meghdoot using a model of stock quotes application. We start the discussion by explaining the schema for the stock quotes application and the datasets used for evaluation. Next, we present the evaluation results for various metrics.

5.1 Experimental Setup

We developed a simulator of Meghdoot in C++. We used the simulator in conjunction with a model of the stock quotes application with the following schema:

$$\mathbb{S} = \left\{ \begin{array}{l} \{\texttt{Date} : \texttt{STRING},\ \texttt{2/Jan/98},\ \texttt{31/Dec/02}\}, \\ \{\texttt{Symbol} : \texttt{STRING},\ \texttt{aaa},\ \texttt{zzzzz}\}, \\ \{\texttt{Open} : \texttt{FLOAT},\ 0,\ 500\}, \\ \{\texttt{High} : \texttt{FLOAT},\ 0,\ 500\}, \\ \{\texttt{Low} : \texttt{FLOAT},\ 0,\ 500\}, \\ \{\texttt{Close} : \texttt{FLOAT},\ 0,\ 500\}, \\ \{\texttt{Volume} : \texttt{INTEGER},\ 0,\ 310000000\} \end{array} \right\}$$

In the above schema, `Symbol` is the stock symbol for the corresponding company. The symbols are character strings of lengths 3 to 5. `Open` and `Close` are the opening and closing prices of the stock on a given day. `High` and `Low` are the highest and lowest price values attained by the stock in the given day. `Volume` is the total amount of trade in this stock on that day.

An example subscription in the stock market publish/subscribe system with the above schema is: $\{\texttt{Symbol} = aapl \wedge \texttt{High} \geq 45\}$, which subscribes for any events for the stock of Apple when the high value of the stock is greater than or equal to \$45. An example event is $\{\texttt{Date} = 30/Jan/98,\ \texttt{Symbol} = aapl,\ \texttt{Open} = 18.31,\ \texttt{High} = 18.87,\ \texttt{Low} = 18.25,\ \texttt{Close} = 18.31,\ \texttt{Volume} = 1450600\}$.

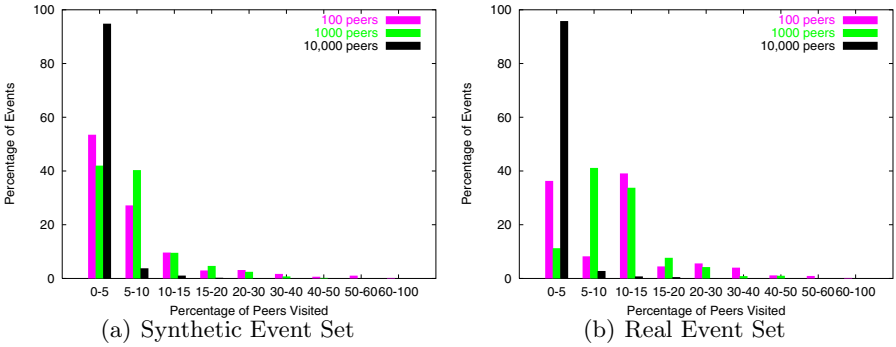
The input sets for the simulations consist of subscriptions and events drawn randomly. The subscriptions were generated using five template subscriptions. A template was picked at random with probability p and parameterized to generate a subscription. We assign general subscriptions low probabilities of occurrence. The reason for this choice is that in a real application subscribers are usually interested in specific events relevant to their narrow interests. Our standard subscription set included 14,029 subscriptions, however, in some of the experiments we increased the number of subscriptions to study the scalability of the system. The templates along with their probability of occurrence are described below:

- $\{(\texttt{Symbol} = P1) \wedge (P2 \leq \texttt{Open} \leq P3)\}$. Notify events for stock $P1$ when its open value is between $P2$ and $P3$. Probability = 20%.
- $\{(\texttt{Symbol} = P1) \wedge (\texttt{Low} \leq P2)\}$. Notify events where a certain stock $P1$'s value is less than or equal to $P2$. Probability = 35%.
- $\{(\texttt{Symbol} = P1) \wedge (\texttt{High} \geq P2)\}$. Notify events where a certain stock $P1$'s value is higher than or equal to $P2$. Probability = 35%.
- $\{(\texttt{Symbol} = P1) \wedge (\texttt{Volume} \geq P2)\}$. Notify events where a certain stock $P1$ was traded more than or equal to $P2$. Probability = 5%.
- $\{\texttt{Volume} \geq P1\}$. Notify if a stock is traded more than $P1$. Probability = 5%.

We have used two different sets of events. One of the event sets was created synthetically by generating uniformly random values in the domains of the attributes. This event set consists of 115,000 events (again in the scalability experiments we varied the number of events). The other event set is the real stock event data for 100 stocks. The data was obtained from Yahoo! Finance [29] by downloading the stock events on a daily basis starting from 2/Jan/ 1998 to 31/Dec/2002. This event set contains 115,353 events. The real event set is highly skewed because most of the stock prices ranged between the values of \$10 to \$30. This is typical of real world datasets.

We evaluated the scalability of the system by running the simulation with varying number of peers in the system. We performed measurements for 100, 1000 and 10,000 peers in the system. The simulations were initialized with one peer in the system. A new peer joins the system after each simulation event with a probability of 10%, until the total number of peers reaches the bound. We measured the number of peers contacted to deliver each event. To evaluate the load on the peers, we measured the total number of messages received by each peer during the run of the simulation. This includes messages due to routing of events and subscriptions, as well as messages due to propagation of events. These experiments do not include peer failure.

In the following sections we present the results of the evaluation. First we present the evaluation results for the scalability and load balancing of the system for the synthetic and real event sets. Later, we analyze the effectiveness of zone replication in the system.



(a) Synthetic Event Set (b) Real Event Set

Fig. 9. Scalability Performance of the System.

5.2 System Scalability

In this section we present the evaluation results for the scalability of the system when the number of peers in the system is varied. We also analyzed the effect of varying the number of subscriptions and the number of events on the system.

Figure 9 shows the distribution of the number of peers that were contacted in order to deliver events to relevant subscriptions. The x-axis in the plot represents the percentage of peers contacted to deliver an event out of the total number of peers that were present in the system when the event was generated. The buckets on the x-axis have been recalibrated because the values 60-100 on x-axis had no data points, and the range 0-20 has been expanded.

Figure 9(a) shows the scalability of the system for the synthetic event set, where the events are distributed uniformly in the domain. In the case of 100 peers, more than half of the events were delivered to all the relevant subscriptions by contacting at most upto 5% of the peers only. For the case of 10,000 peers 95% of the events were delivered to all affected subscriptions by contacting less than 5% of the peers. In fact, almost all the events were delivered by contacting less than 10% of the peers. Overall, as the number of peers in the system increases, the peers needed to be contacted for delivering events scales very well.

Figure 9(b) shows the results for the simulations with the real event set which is highly skewed. Because of the skewed distribution of the event data, more zones are created in the vicinity of event points which leads to a small increase in the number of peers contacted to deliver events. Even with skewed events, in the cases of 100 and 1000 peers, 85%-90% are delivered by contacting at most 15% of the peers. For the experiment with 10,000 peers, almost all events are delivered by contacting less than 10% of the peers and 97% of those events contact less than 5% of the peers. This strengthens our conclusion that Meghdoot scales very well as the number of peers in the system increase.

We performed an experiment with 10,000 peers in the system by varying the number of subscriptions installed in the system from 25,000 to 150,000. We used the synthetic event set for this experiment which contained 115,000 events. Figure 10 shows the results. In all cases more than 95% of the events are

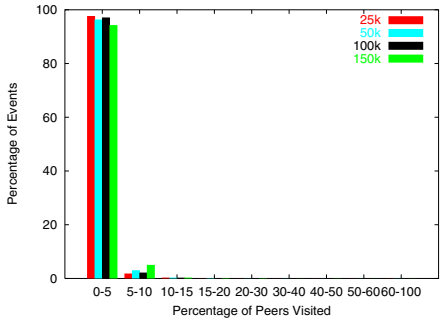


Fig. 10. Effect of Varying the Number of Subscriptions.

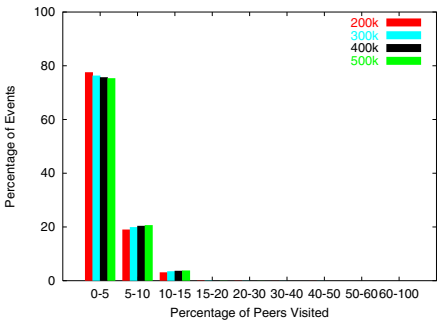


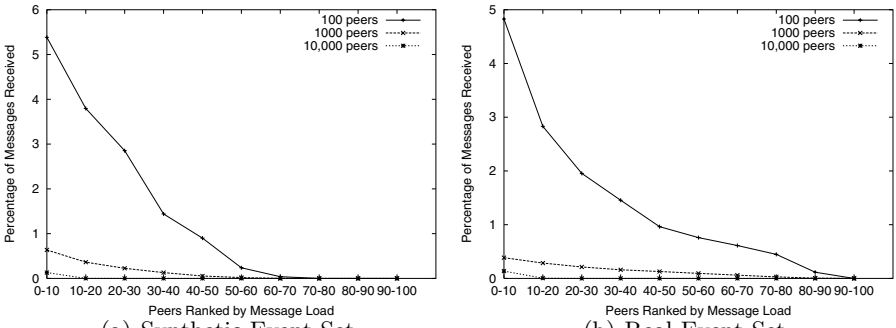
Fig. 11. Effect of Varying the Number of Events.

delivered by contacting less than 5% of the peers in the system. There is only a marginal degradation in performance even as the number of subscriptions stored in the system increases from 25,000 to 150,000 over 10,000 peers. This shows that Meghdoot scales well as the number of installed subscriptions increases.

In another experiment we varied the number of events from 200,000 to 500,000. These simulations had 10,000 peers in the system and 50,000 subscriptions. The events sets contained uniformly distributed events. Figure 11 shows the results of the experiment. With 200,000 events in the system more than 75% of the events are delivered by contacting at most 5% of the peers. For all cases, at most 15% of the peers are contacted to deliver the events. The results are very similar in all cases demonstrating that the system scales very well with the number of events.

5.3 Load Distribution

We measure the load on a peer as the ratio of messages the peer receives to the total number of messages processed in the system since the peer joined the system. Figure 12 shows the load distribution on the peers in the system. The peers were sorted in decreasing order of the load, and were grouped by their rank into groups of size 10% each. The plot shows the average load on each group.



(a) Synthetic Event Set (b) Real Event Set

Fig. 12. Load Distribution in the System.

Figure 12(a) shows the load distribution in the system for the synthetic event set where the events are distributed uniformly. Even in the case of 100 peers the maximum load is only 5.35% of the messages, which is very good. In the case of 10,000 peers the load is very evenly distributed among all the peers in the system. As the number of peers increase in the system, the load is evenly shared by the new peers. This shows that our load balancing schemes are quite effective.

Figure 12(b) shows the load distribution over the peers for the real event set which is highly skewed. In the case of 100 peers, the maximally loaded peers handle less than 5% of the total messages. As the number of peers increases, the load is well distributed among the new joining peers. The trend of load distribution is very similar to the case of uniformly distributed events. Therefore, our load balancing strategies are very effective and adapt very well dynamically to the distribution of subscriptions and events.

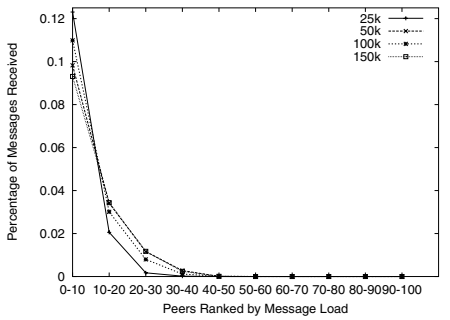


Fig. 13. Effect of Varying the Number of Subscriptions on Load Distribution.

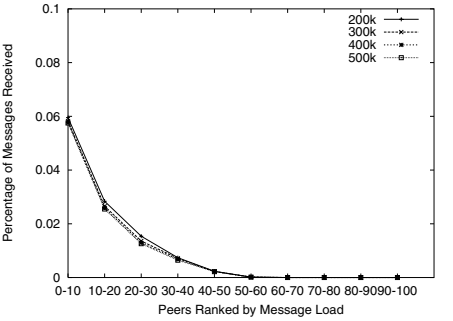


Fig. 14. Effect of Varying the Number of Events on Load Distribution.

Figure 13 shows the load distribution in the system when we varied the number of stored subscriptions. This experiment had 10,000 peers and used the synthetic event set with 115,000 events. Even the most heavily loaded peers received only about 0.123% of the total messages generated in the system. When the number of stored subscriptions increases, the number of messages generated increases, but as is evident from the graph, this load is evenly distributed among the available peers. The load distribution in a 10,000 peer system when the number of events is varied from 200,000 to 500,000 is shown in Figure 14. There were 50,000 subscriptions in this experiment. The distribution of the load among the peers in the system remains quite stable with the growing number of events.

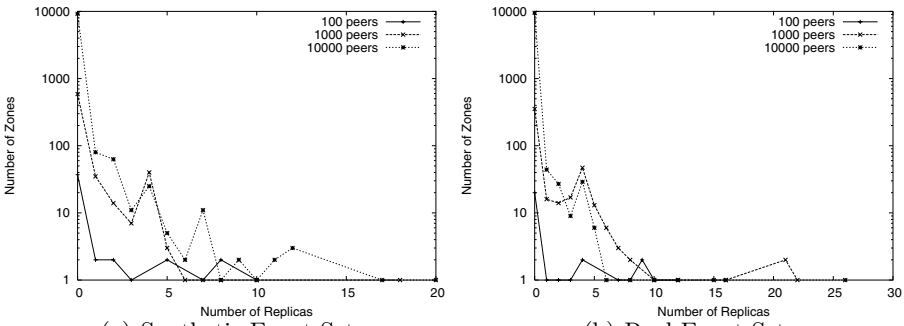
5.4 Replication of Zones

The system replicates zones that are overloaded due to event propagation as a mechanism to handle load. This has proved to be a very effective strategy as is evident from the previous analysis. In this section, we analyze the effect the distribution of events has on the degree of replication in the system.

Table 1. Number of Unique Zones.

#peers	Synthetic Events	Real Events
100	76	73
1000	652	470
10,000	9622	9323

Table 1 summarizes the number of unique zones that were created during the simulations with real and synthetic event sets with 14,029 subscriptions. Each zone in the system has at least one peer associated with it. The remaining peers are used for replication. Note that the number of peer nodes used for the replication of existing zones is higher in all cases when the real event set is used. This is the case because the real event set, as is typical of many real world datasets, is very skewed and a lot of events are propagated through the same zones which overloads them. Therefore, new joining peers are directed to those zones and they are replicated to reduce the propagation load.



(a) Synthetic Event Set (b) Real Event Set

Fig. 15. Replication of zones.

Figure 15 presents the distribution of the degree of replication in the system. The x-axis is the degree of replication and the y-axis is the number of unique zones that were replicated with that degree. The y-axis has been plotted on logscale because of the large variation in the number of non-replicated zones in the cases of different number of peers in the simulations. In Figure 15(a) for 10,000 peers, there was a zone which was replicated 20 times. From Figure 15(b), we can see that one of the zones was replicated to a high degree of 27. As mentioned earlier, in the real event set, most stock events had price values in the range of \$10-\$30. Thus a large subset of the events map to the event points that fall in a small number of zones. These zones therefore have a high degree of replication to overcome the load generated by these events.

6 Conclusions

We presented Meghdoot, a middleware for a content-based publish/subscribe system, which leverages peer-to-peer based Distributed Hash Tables for scalable dissemination of data events to subscribers distributed across the network. P2P design offers the flexibility of incorporating additional resources, thus providing performance scalability. Meghdoot imposes no restrictions over subscriptions and allows them to be specified in terms of range predicates over all attributes in a schema. Unlike most other P2P approaches, Meghdoot uses the semantics of the subscriptions and the events to store subscriptions and deliver matching events to them. Since real world datasets tend to be skewed, existing peer management techniques fail to distribute load well among the peers. Hence, unlike previous work, we use the characteristics of the load to determine how to distribute the load. Subscription load leads to zone splitting, while event propagation load leads to zone replication. We also exploit the underlying distributed hash structure to replicate subscriptions for fault tolerance in an innovative and systematically transparent way. Our extensive simulation experiments have verified the scalability and load balancing aspects of Meghdoot. In particular, the experiments show that the design scales very well upto thousands of peers in the system and can handle large numbers of subscriptions and events. They also demonstrate that both zone replication and splitting techniques are very effective in evenly distributing the load among the peers in the system.

References

1. H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, Feb. 2003.
2. G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th ICDCS*, pages 262–272, 1999.
3. K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, Dec. 1993.
4. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.
5. M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):100–110, 2002.
6. J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD*, pages 379–390, 2000.
7. F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record*, 30(2):115–126, 2001.
8. L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating data sources in large distributed systems. In *Proceedings of the 29th VLDB*, pages 874–885, 2003.
9. Gnutella. http://gnutella.wego.com/.

10. A. Gupta, D. Agrawal, and A. El Abbadi. Approximate range selection queries in peer-to-peer systems. In *Proceedings of the 1st CIDR*, pages 141–151, 2003.
11. E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *Proceedings of the 15th ICDE*, pages 266–275, 1999.
12. M. Harren, J. M. Hellerstein, R. Huebsch, B. T. Loo, S. Shenker, and I. Stoica. Complex queries in DHT-based peer-to-peer networks. In *Proceedings of the first International Workshop on Peer-to-Peer Systems*, pages 242–250, 2002.
13. IBM News. http://www.ibm.com/ibm/history/history/year_2000.html.
14. KaZaA. http://www.kazaa.com/.
15. Napster. http://www.napster.com/.
16. B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The information bus: an architecture for extensible distributed systems. In *Proceedings of the fourteenth ACM SOSP*, pages 58–68, 1993.
17. P. R. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *Proceedings of the 2nd DEBS*, pages 1–8, 2003.
18. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 ACM SIGCOMM*, pages 161–172.
19. S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of the 3rd International Workshop of NGC*, volume 2233, pages 14–29. LNCS, Springer, 2001.
20. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001.*
21. O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi. A peer-to-peer framework for caching range queries. In *Proceedings of the 20th ICDE*, pages 165–176, 2004.
22. B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Proceedings of AUUG*, 1997.
23. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM*, pages 149–160.
24. D. Tam, R. Azimi, and H.-A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *Proceedings of International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, 2003.
25. W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *Proceeding of the Second DEBS*, 2003.
26. Tibco. http://www.tibco.com/.
27. P. Triantafillou and T. Pitoura. Towards a unifying framework for complex query processing over structured peer-to-peer data networks. In *Fist International workshop DBISP2P 2003*, pages 169–183.
28. Vitria. http://www.vitria.com/.
29. Yahoo! Finance. http://finance.yahoo.com/.
30. Y. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, University of California at Berkeley, 2001.
31. S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the 11th ACM NOSSDAV*, pages 11–20, 2001.