Infrastructure Free Content-Based Publish/Subscribe

Vinod Muthusamy and Hans-Arno Jacobsen *Member, IEEE*

***Abstract*—Peer-to-peer (P2P) networks can offer benefits to distributed content-based publish/subscribe data dissemination systems. In particular, since a P2P network's aggregate resources grow as the number of participants increases, scalability can be achieved using no infrastructure other than the participants' own resources. This paper proposes algorithms for supporting content-based publish/subscribe in which subscriptions can specify a range of interest and publications a range of values. The algorithms are built over a distributed hash table abstraction and are completely decentralized. Load balance is addressed by subscription delegation away from overloaded peers and a bottom up tree search technique that avoids root hotspots. Furthermore, fault-tolerance is achieved with a light-weight replication scheme that quickly detects and recovers from faults. Experimental results support the scalability and fault-tolerance properties of the algorithms: For example, doubling the number of subscriptions, does not double system internal messages, and even the simultaneous failure of 20% of the peers in the system requires less than two minutes to fully recover.**

***Index Terms*—content-based publish/subscribe, peer-to-peer, distributed hash table, pub/sub, P2P, DHT**

I. Introduction

This paper develops a content-based publish/subscribe (pub/sub) system by leveraging structured peer-to-peer (P2P) overlay networks. The objective is to support fine-grained infrastructure free data dissemination at Internet scale.

Large-scale pub/sub systems are increasingly common in industry. For example, Google and Yahoo use the GooPS [31] and Yahoo Message Broker [14] pub/sub systems, respectively, to integrate their Web applications; SuperMontage [37], a distributed system that uses a pub/sub model, is used to disseminate financial data and orders; retailers such as Wal-Mart and Target exchange supply chain information using the GDSN (Global Data Synchronization Network) pub/sub network [20]; and a pub/sub system developed by IBM was used to deliver tennis match scores to millions of users around the world [24]. Common to these applications is the selective dissemination of data to a very large number of geographically scattered entities.

While large-scale pub/sub applications, like those above, can be built today, they typically require a large company's resources to deploy and administer [24]. A system could require dozens or hundreds of servers placed at strategic points across the globe in order to handle the load, and trained personnel are needed to monitor the network and strategically deploy servers and network bandwidth to resolve the bottlenecks. For example, many popular Web sites on the Internet use the Akamai content distribution system to cache Web site content across the Internet. The inability or unwillingness of many large corporations to perform this task themselves underscores the complexity of managing Akamai's infrastructure of 56 000 dedicated servers in 950 networks across 70 countries [13]. Likewise, large Web hosting organizations deploy and administer 70 000 or more geographically distributed machines [26].

The proposed design in this paper virtually eliminates pub/sub infrastructure costs. Instead of requiring dedicated servers, the users' resources are automatically used to achieve infrastructure free scalability. In addition, the design self-organizes to adapt to bottlenecks and faults, so no personnel are required to administer the network.

It should be noted that the use of a P2P network in this design may make the pub/sub system inappropriate for certain applications. For example, latencies are higher than for comparable systems with dedicated resources. Similarly, failure recovery times are comparatively higher. Also, since any node may participate in forwarding messages, it is not possible to meter traffic, control how sensitive data is handled, and deal with situations, where nodes might maliciously (or mistakenly) cause denial of service. However, years of building applications over the public Internet have shown that the end-to-end principle can address many concerns of relying on an unreliable common network infrastructure.

While there are several proposals for building a pub/sub system over a P2P network [3], [5], [6], [12], [21], [29], [35], [36], their design decisions have implications that this paper seeks to address. An obvious drawback of some early proposals, such as Scribe [12], is they only provide a restricted channel-based pub/sub model which cannot support the more expressive requirements of some of the above applications. For example, an expected GDSN scenario [19] is one where suppliers need to closely monitor all pricing information related to their products, whereas retailers may only be interested in price swings larger than some threshold. Such fine-grained constraints cannot be expressed in channel-based pub/sub models. Even systems that are more expressive may not efficiently support range constraints [35], which are inherently difficult to express in structured P2P networks such as distributed hash tables (DHT). For example, consider a subscription with a constraint "alert = warning". A hash table is ideally suited for such exact value lookups, but a range constraint such as "price $<$ 10 AND alert = warning" may require separate lookups for each discrete value less than 10. This can be prohibitively expensive or impractical, especially, when the constraint is over a floating point data type. A possible workaround may require the application to transform its subscription, either into one with exact value lookups only, which loses the initially intended query semantics, or into one that sends the exponent of the normalized float alongside the float attribute, which would require end-node filtering, potentially increasing the traffic in the P2P network. Also stemming

V. Muthusamy is with the IBM T.J. Watson Research Center, Yorktown Heights, New York. The work in this paper was performed at the University of Toronto. (e-mail: vmuthus@us.ibm.com).

H.-A. Jacobsen is with the Department of Electrical and Computer Engineering and Department of Computer Science, University of Toronto, Toronto, Canada. (e-mail: jacobsen@eecg.toronto.edu).

from the friction between content-based constraints and hash table lookups, many proposed algorithms exhibit inefficient message propagation, such as flooding constraints over the network [36], delivering data individually to each interested entity instead of multicasting it [3], or being susceptible to nodes that may become message hotspots [29]. In addition, some existing approaches restrict the data types or require the data schema to be fixed and known to all participants [3], [21]. Such requirements limit the flexibility of the system, and are antithetical to the decentralized nature of P2P networks.

This paper addresses issues with existing work, including all those mentioned above. Our approach supports an expressive content-based pub/sub model [4], [10], [16], [17], efficiently indexes range constraints in a DHT network, performs multicast message dissemination, dynamically manages overloaded nodes, and allows applications to determine and modify their data schema as they see fit.

Most existing large-scale pub/sub applications can benefit from our design. In addition, an infrastructure free scheme opens the door to applications that would have otherwise been infeasible. One such class of applications is "grassroots" applications involving users with little resources or expertise, such as a small shareware developer wanting a system to selectively distribute software updates to her users, but who does not have the resources to purchase and manage such a system. Another class of applications is those that require lots of resources for a short period of time, such as disseminating real-time results of sporting events, such as Olympia. This scenario can have millions of users around the globe, requiring selective information dissemination capabilities, and a massive pub/sub network. However, as this network is only used for a few weeks, it might not be cost-effective to deploy such a system with a dedicated infrastructure. Yet another class of applications is those where the users do not have the expertise to, or do not want to bother with the trouble of, deploying a pub/sub network. Consider a corporation that routinely needs to deliver memos to appropriate employees. A pub/sub system is an improvement over using mailing lists as it allows more fine-grained delivery of memos to only interested users. Finally, fully decentralized, infrastructure free approaches are also sometimes advocated for scenarios that would like to prevent a single entity from owning and accessing its users' information.

Concrete examples for these classes of pub/sub applications are the globally operating Spotify music service that employs a P2P network to offload processing from its dedicated resources. Spotify uses a pub/sub network to discover and disseminate relevant songs among interested peers [33], with latency requirements in the seconds for the pub/sub functionality. Decentralized social networking efforts, like Tribler.org and the open *Tent* specification [38], support pub/sub functionality without dedicated resources, including pub/sub capabilities for presence and status update dissemination [39] as well as selective information dissemination and recommendations [15], all of which require fine-grained filtering capabilities with operating latencies in the range of seconds. A further non-latency sensitive example is "cloudless" data storage that selectively synchronizes files across multiple devices among definable groups of end-points [25]. Finally, the provision for pub/sub capabilities in the widely used *Extensible Messaging and Presence Protocol* (XMPP) [39] which is "a protocol extension for generic publish-subscribe" and the *Personal Eventing Protocol* (PEP) [39], which "provides a presence-aware profile of PubSub" used "for rich presence, microblogging, social networking, ..." [39] constitute examples. XMPP, for example, is used in the federated GoogleWave protocol, where all communication except wavelet updates are sent via pub/sub [8].

The common theme here is that a pub/sub system running on a self-configuring, infrastructure free P2P network lowers the bar on the costs of using a pub/sub system, and this can lead to applications that are currently infeasible for reasons of financial cost, technical expertise, or administrative inconvenience. Furthermore, most of the above listed pub/sub scenarios have an operating latency requirement for the dissemination of subscriptions and events that ranges in seconds, fully in-line with the capabilities of today's P2P networks.

The main contributions of this work are as follows:

1) We evaluate alternative P2P-based pub/sub systems in a qualitative comparison. This analysis identifies several properties that can be used to distinguish these systems and highlights the strengths and weaknesses of the approaches in the literature.
2) We devise a distributed content-based pub/sub matching algorithm, referred to as *Distributed Multi-dimensional Matching* (DMM). The design of DMM is agnostic to the P2P substrate used, only leveraging the common DHT primitives. The algorithm is designed to avoid hotspots, and manage them when they do occur. In addition, pub/sub semantics are extended to allow publications with ranges, a unique property of this algorithm.
3) An extension of the above algorithm is devised, referred to as *DMM with Attribute Roots* (DMM-AR), that imposes no fixed global schema on the system. This is a difficult problem to solve in a DHT-based P2P network and is a key benefit of our algorithm as opposed to related approaches. As well, a light-weight path caching mechanism is constructed to reduce the publication delivery latency.
4) DMM and DMM-AR are fully implemented and evaluated, and we demonstrate their scalability and fault-tolerance properties.

Section II first presents a background on the pub/sub model and P2P networks. Section III then describes related work in building pub/sub systems over a DHT, and presents a detailed qualitative analysis of the relevant approaches. In Section IV, the distributed pub/sub matching algorithm is developed, and Section V addresses distributed pub/sub issues other than matching. Section VI evaluates the algorithm, and Section VII completes the paper with some concluding remarks and discussion of future work.

II. Background

To keep this paper self-contained, this section presents a brief overview of the pub/sub model and P2P networks.

A. Publish/Subscribe

Pub/Sub is a data dissemination model with three entities: The *publisher* is the data producer, the *subscriber* is the consumer, and the *broker* mediates between the two. There may be one broker or a set of distributed brokers. For example, in a stock quote dissemination application, the publisher would be the stock exchange, and the consumer could be a stock broker interested in tracking certain stocks. A subscriber S expresses his interest in these stocks by sending a *subscription* message s to the broker. The publisher P communicates the latest stock updates by sending a *publication* message p to the broker. Upon receipt of p, the broker forwards p to those subscribers with matching subscriptions.

In content-based pub/sub, publications consist of a set of {attribute, value} pairs, and subscriptions are typically conjunctions of {attribute, operator, value} tuples, where the operator can be one of $\{=, <, >, \leq, \geq\}$ and include operations over strings. This allows subscriptions to discriminate based on the *content* of the publications. This is a more expressive model than channel-based pub/sub in which a publication is sent to a channel and delivered to all subscribers belonging to the channel.

B. Peer-to-peer

P2P networks are characterized by the direct sharing of resources among the peers in the network. P2P protocols can loosely be classified as unstructured and structured.

The first generation of P2P networks such as Napster, Gnutella and Kazaa were unstructured and provided no performance guarantees. The second generation of P2P networks [1], [22], [30], [32], [34], [41], called structured P2P networks, are based on a distributed hash table (DHT) interface. These DHTs are self-organizing, load balanced, fault-tolerant, provide statistical guarantees on the bandwidth usage and node state requirements, and offer an upper bound for the hop count to reach a queried nodeId.

A DHT is a distributed version of a hash table. It stores a (key, value)-pair at some node in the network, with the core operation of a DHT protocol being the ability to map a key to a node and efficiently route messages to this node.

The Pastry [32] DHT stores (key, value)-pairs where the $value$ is a sequence of bytes and the key is a 128-bit number. Each peer in the DHT network is addressed by a 128-bit *node identifier*. The key and nodeId are a sequence of 2^b digits and belong to a circular 128-bit identifier space. The keys and nodeIds are generated by the SHA-1 cryptographic hash to ensure an even distribution of keys and nodeIds on the identifier circle.

Pastry, much like other approaches, maps keys to the node with the numerically closest nodeId in the identifier circle such that in a network with N nodes and K keys, each node stores K/N keys with high probability. In Pastry, a prefix routing algorithm ensures each hop of a message is sent to a node that matches the destination nodeId by at least one more digit. This routing algorithm, as well as many other DHT-based algorithms, is able to route a message to the destination in $\lceil \log_{2^b} N \rceil$ overlay hops, and requires $O(\log N)$ node state to achieve this routing performance.

III. Related work

In this section, we compare our work to other pub/sub systems implemented over a DHT interface and include a detailed analysis of the qualitative properties of these approaches.

Scribe [12] is a *channel-based* pub/sub system built over the Pastry DHT. Scribe treats a channel name c as a key in the DHT which is stored at peer r called the channel root. Subscriptions are sent towards r, and their reverse path builds a multicast tree from the channel root r to the subscribers. Publications are also sent to the channel root, and then follow the multicast tree to the subscribers in the channel. As mentioned in Section II-A, channel-based pub/sub has limited filtering capabilities.

daMulticast [5] builds a hierarchical channel-based pub/sub system. The algorithm limits subscription state stored at nodes by dynamically grouping nodes based on their subscriptions and efficiently routing messages between groups. As with Scribe, daMulticast's channel-based pub/sub model is less expressive than a content-based model.

Hermes [29] is a pub/sub system built over the Pastry DHT. Hermes employs rendezvous nodes (RN), which are setup through event type messages submitted prior to publishing. An RN ensures that advertisements and subscriptions for a given event type meet; both are routed towards the same RN, setting up a routing path along their propagation path in the overlay for subsequent publication dissemination. Similar to the channel-name in Scribe, here, the event type name serves as the identifier for determining the location of the RN via the DHT. Next to channel-based semantic, content-based pub/sub semantic is achieved through matching publications along the propagation path by storing predicate filters next to routing information for subscriptions and advertisements. However, unlike the algorithms in this paper, Hermes relies on event type names to create a distributed matching index and does not address the issue of an overloaded channel root peer.

Tam *et al.* [35], where among the first, to map content-based pub/sub to a channel-based pub/sub model. A globally known schema that specifies the attribute names, types, and values in the system is required, and indices, each consisting of strategically chosen attributes, must be statically specified for each application deployed on the system. Each publication and subscription is mapped to several index digests, one for each index. An index digest is a channel name comprising the concatenation of the name, type, and value of the attributes in the corresponding index. In this way, content-based pub/sub semantics can be achieved from a channel-based one. The system's performance is sensitive to the specification of these indices, and moreover the indices need to be manually specified. Furthermore, manually specified, globally known indices are contrary to the P2P philosophy of minimal administration.

Meghdoot [21] is a content-based pub/sub system built over the CAN DHT. Meghdoot requires a static, globally known schema of the pub/sub attributes in the system. For a system with k attributes, it constructs a CAN space of dimension $2k$, with subscriptions mapped to a point in the CAN space and stored at the responsible node. Publications then traverse all regions with potentially matching subscriptions. Since pub/

Property	DMM	DMM-AR	Scribe [12]	Hermes [29]	Meghdoot [21]	PastryStrings [3]	Terpstra [36]
Query language	content	content	channel	content	content	content	content
Global schema	yes	no	no	no	yes	yes	no
DHT agnostic	yes	yes	yes	yes	no	no	no
Simultaneous attribute matching	yes	yes	n/a	yes	yes	no	yes
Multicast delivery	yes	yes	yes	yes	yes	no	yes
Sub indexing	1	2	1	1	1	1	flood
Match/storage bias	match	match	match	match	storage	match	match
Load balance match	yes	yes	no	no	yes	no	no
Load balance storage	yes	yes	no	no	yes	yes	no
Fault-tolerance	yes	yes	yes	yes	yes	yes	yes
Hotspot avoidance	yes	yes	no	no	yes	yes	no

TABLE I
QUALITATIVE COMPARISON OF RELATED WORK

sub workloads typically have many more publications than subscriptions, it is not as advantageous to optimize the design for subscription state (a subscription is only stored at one peer in Meghdoot) at the expense of publication matching load and delay (publications have to be routed to multiple peers).

Baldoni *et al.* [7] map publications and subscriptions to bit strings. However, publications and subscriptions are mapped to multiple nodes, with large ranges generally mapping to many nodes with the requisite need to be indexed by more nodes. We, however, present a protocol that indexes subscriptions at only one node, and replicates this index for load balance only when the index node becomes overloaded. Also, as in Meghdoot, [7] requires a global, static schema of known pub/sub attributes.

PastryStrings [3] indexes string predicates in a forest of β search trees of degree β and depth D, where β is the size of the alphabet and D the maximum string length. To handle numeric predicates, the domain of each subrange's predicate is mapped to nodes in the forest, with larger subranges mapping to higher nodes in the hierarchy. Events traverse down the search trees and retrieve matching predicates. PastyStrings matches each predicate individually and uses a counting algorithm to find matching subscriptions, whereas the algorithms in this paper simultaneously index and match all attributes in a subscription. PastryStrings also does not focus on the multicast delivery of events to interested subscribers.

Yang *et al.* [40] subdivide a content space into subregions in a manner similar to this paper. However, the system still assumes a fixed, known schema, and requires a top-down publication matching strategy that may overload the root of the subscription indexing tree.

Terpstra [36] builds a content-based pub/sub system over the Chord DHT. The algorithm creates a separate multicast tree rooted at each broker. The multicast trees are built by essentially flooding subscriptions, which may be drastic. To address this, flooding is somewhat quenched by covering and merging the subscriptions as they propagate.

This paper is related to prior work [28] that studied pub/sub in the context of small-scale P2P networks (i.e., enterprise system scales). An interesting result in that paper is that protocols designed for large-scale P2P networks may not scale down to small networks (of up to several dozens of nodes). That previous work and this paper build on techniques developed earlier [27].

Table I qualitatively compares a number of the related P2P pub/sub approaches. The first two approaches—DMM and DMM-AR—are the two algorithms presented in this paper. Other than Scribe, the comparison focuses on content-based pub/sub systems. A number of approaches, such as Meghdoot require a fixed and globally known schema of attributes in the system, making them inappropriate for systems where new applications may be deployed dynamically. Some algorithms, such as Hermes, only rely on the basic DHT lookup interface and can be automatically used with any new or improved DHTs. Some approaches, such as PastryStrings, collect matching predicates individually and then try to determine which subscriptions have been fully matched. This may not scale with the number of predicates per subscription or attributes per event. After finding matching subscriptions, most of the algorithms have the ability to multicast the publication to matching subscribers. The subscription indexing row in the table concerns the number of peers where the subscription is stored for indexing (as opposed to routing) purposes. For example, in Scribe, only the subscription stored at the channel root is strictly necessary during matching; the subscriptions stored along the path to the channel root are used to build the multicast tree and are used during the delivery of the publication. This count also ignores subscriptions that may be stored at multiple peers to achieve load balance or replication. Most of the algorithms index a subscription at one peer, except for DMM-AR which stores it at two peers, and Terpstra *et al.* which essentially floods them to all peers. A common assumption in pub/sub applications is that the number of publications is far larger than the number of subscriptions, and hence it is reasonable to optimize the algorithms for matching publications as opposed to indexing subscriptions. Most of the algorithms are indeed biased towards publication matching except for Meghdoot which stores subscriptions at a single peer but routes publications to many peers. Note that in some other approaches, such as DMM and PastyStrings, publications must traverse a path in a tree to find subscriptions and hence do visit multiple peers to perform matching, but (subjectively) the bias is still towards optimizing matching, compared to Meghdoot which must visit a potentially large number of peers in the system. Some of the algorithms take steps to dynamically balance the load on peers resulting from having to match too many publications or storing too many subscriptions.

IV. ARCHITECTURE

There are two main facets to a distributed pub/sub algorithm: Matching publications with subscriptions and multicasting publications to interested subscribers. This section focuses on the former problem in the context of a P2P network. Multicasting publications is a simpler problem and is addressed in Section V-A.

In our approach, we map the pub/sub matching problem to a distributed multidimensional indexing problem. In particular, publications and subscriptions are mapped to regions in a multidimensional space such that the intersection of these regions implies a match of the corresponding publications and subscriptions. The multidimensional space is recursively partitioned into regions and indexed by a (distributed) search tree, nodes of which are managed by peers in the network. The indexed regions, as well as subscriptions and publications, are uniquely labeled with keys (below referred to as z-codes), which serve to identify peers in the network that manage the corresponding search tree nodes. The keys are designed to allow a search tree node to easily determine the keys of its parent and child nodes, which, again, serve as keys to the underlying DHT to find the relevant peers for these nodes. A novel aspect of the algorithm is that the search tree can be traversed bottom up, thereby avoiding overloading the root of the tree.

A. Distributed multidimensional matching

We develop a distributed data structure that can match multiple attributes simultaneously. This algorithm, referred to as *distributed multidimensional matching* (DMM), maps the pub/sub matching problem to a multidimensional indexing problem.

We adapt database techniques from multidimensional indexing of objects and develop a distributed indexing structure that is embedded in a P2P network [18]. Other than existing indexing structures, like the kd-tree or the grid file [18], our index can be traversed bottom up or from any node in the tree (e.g., from a node that manages the smallest enclosing region for a given subscription and publication), alleviating the root of the tree from becoming a hot spot.

To facilitate the discussion, we temporarily require that the set of all attribute names and domains is globally known. We remove this constraint in Section IV-C.

Unlike traditional pub/sub semantics, the DMM algorithm also allows publications to support inequality constraints as in subscriptions. For example, it is possible for the "height" attribute in a publication to have a "value > 3". In addition, an attribute's value in a subscription can be bound to another attribute's value in the subscription. For example, a subscription can express interest for publications with "height $>$ 3" and "length $< 2 *$ height". This allows for more expressive language semantics. For example, we can express interval event semantics (i.e., events that extend over a period of time or space), model uncertainty in events, where the exact value is unknown, and support location-based services with imprecisely given location position information. Variable binding allows applications to formulate more precise subscriptions that would otherwise lead to too many or too few matches.

d	The number of dimensions in the spatial domain. Also the number of attributes in the pub/sub domain.
a_i	Attribute i in the pub/sub domain.
r	A region in the spatial domain.
$n(r)$	The node in the search tree corresponding to region r.
$z(r)$	The z-code of region r.
$z_i(r)$	The z-code of the i-th dimension of region r.
$p(r)$	The peer in the DHT responsible for node $n(r)$ in the search tree.

TABLE II
NOTATION USED IN THE PAPER

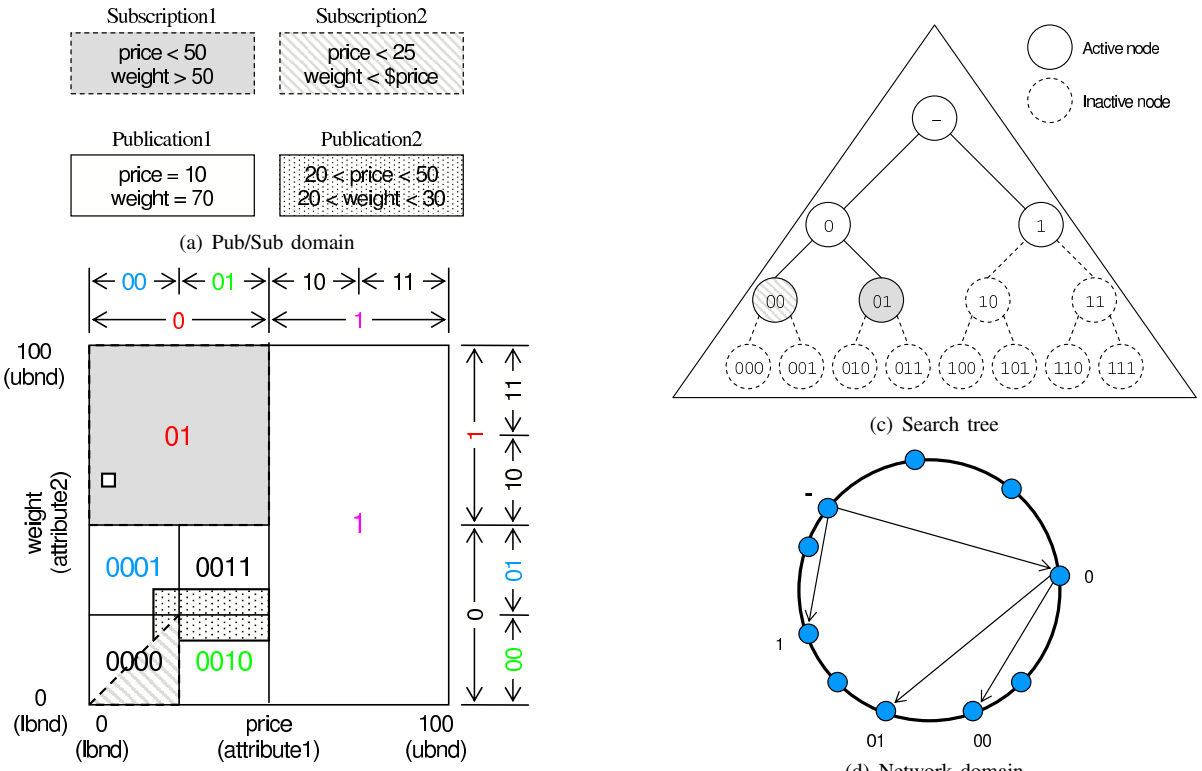
1) Mapping pub/sub to multidimensional indexing: As a running example, we use the four subscriptions and publications in Figure 1(a). Table II summarizes the key notation used in the following discussion.

The mapping from the pub/sub domain to a spatial domain for multidimensional indexing is as follows: (1) A d-dimensional space is created, where d is the number of unique attributes (name and data type) in the pub/sub domain. (2) Every attribute a_i in the pub/sub domain maps to a dimension d_i in the spatial domain.

A d-dimensional space S is managed by a binary *search tree* that represents a recursive subdivision of the universe into subspaces (regions) by means of $(d-1)$-dimensional hyperplanes. The hyperplanes are iso-oriented (i.e., aligned with their respective axes) and cycle through the d possibilities. For example, for $d = 3$, splitting hyperplanes are alternately perpendicular to the x-, y-, and z-axes, with each hyperplane dividing a region in half. Each region r has a corresponding node $n(r)$ in the search tree.

Each region is addressed by a bit string, called a z-code, and is associated with one node in the tree. The z-code of a region is computed by first computing the z-code for each dimension (i.e., attribute) of the region, and then interleaving the individual z-codes. Figure 2 presents the algorithm to convert an integer attribute to a z-code. It takes as input the range and bounds of the attribute, and looks for the smallest 1-dimensional region that encloses the attribute's range. Z-codes can be derived for any data type that can be mapped to a range. For example, the z-codes for string attributes can be computed by converting the string to an integer (assuming the maximum string length is known). The z-code for the entire region is computed by interleaving the z-codes of the corresponding ranges of each dimension that constitute that region, as shown by the algorithm in Figure 3. An interesting property of z-codes is that if a z-code, z_1, is a prefix of another z-code, z_2, then z_1 encloses (covers) z_2. This property serves to identify for a given publication (subscription), the smallest enclosing region, used in below message propagation algorithms. An example of applying these algorithms to derive z-codes is described next.

Figure 1(b) shows the spatial domain representation of the subscriptions and publications from Figure 1(a). Both the "price" and "weight" attributes have values in the range $[0, 100]$. For clarity, only some of the splitting hyperplanes are shown, and only some of the regions are labeled with their z-codes. For example, the large white region on the right has a z-code of 1, while the shaded upper-left region has a z-code



(a) Pub/Sub domain

(b) Spatial domain

(c) Search tree

(d) Network domain

Fig. 1. Mapping from pub/sub to spatial to network domain

Algorithm *IntAttributeToZCode*($lval$, $uval$, $lbnd$, $ubnd$)
($*$ Value is [lval,uval] and bounds are [lbnd,ubnd] $*$)
1. $l \leftarrow lbnd$
2. $u \leftarrow ubnd$
3. $zc \leftarrow$ **nil**
4. $lastIter \leftarrow$ **false**
5. **repeat**
6. $m \leftarrow \frac{l+u}{2}$
7. **if** $lval \leq m \wedge uval \leq m$
8. **then** $u = \lfloor m \rfloor$
9. $zc \leftarrow zc.Append(0)$
10. **else if** $lval > m \wedge uval > m$
11. **then** $l \leftarrow \lceil m \rceil$
12. $zc \leftarrow zc.Append(1)$
13. **else** ($*$ Doesn't fit in either half. $*$)
14. **stop** ($*$ Break out of loop. $*$)
15. $lastIter \leftarrow l == u$
16. **until** $lastIter$
17. **return** zc

Fig. 2. Z-code of integer attribute

Algorithm *AttributesToZCode*($attrs$)
($*$ Return the z-code of the specified attributes $*$)
1. ($*$ Find length of shortest z-code across all attributes. $*$)
2. $minlen \leftarrow \min_{a \in attrs} (|ZCodeOf(a)|)$
3.
4. ($*$ Weave z-codes into one. $*$)
5. $zc \leftarrow$ **nil**
6. **for** $i \leftarrow 0$ **to** $minlen - 1$
7. **do for** $j \leftarrow 0$ **to** $|attrs| - 1$
8. **do** $z \leftarrow ZCodeOf(attrs[j])$
9. $bit \leftarrow GetBit(z, i)$
10. $zc \leftarrow zc.Append(bit)$
11. **return** zc

Fig. 3. Z-code of a set of attributes

of 01. The top and right axes are labeled with the z-codes of the indicated portions of their respective dimensions.

Consider the region r with z-code $z(r) = 0010$. This region's "price" dimension has a z-code of $z_p(r) = 01$ as indicated by the label at the top, and its "weight" dimension has a z-code of $z_w(r) = 00$. The values of $z_p(r)$ and $z_w(r)$ can be computed with the algorithm in Figure 2. Then, the algorithm in Figure 3 is used to derive $z(r)$ by interleaving $z_p(r)$ and $z_w(r)$ (i.e., taking the first bit of $z_p(r)$ (0), appending the first bit of $z_w(r)$ (0), appending the second bit of $z_p(r)$ (1), and appending the second bit of $z_w(r)$ (0), resulting in $z(r) = 0010$).

A subscription s is stored at all the leaf nodes $n(r_i)$ in the search tree such that r_i intersects s. Thus, the insertion or deletion of a subscription may require the traversal of multiple paths from the root to leaves of the tree. Note that a leaf node with an excessive number of subscriptions can create two children and move a subset of the subscriptions to each child.

The splitting/merging of regions is done dynamically. If the number of subscriptions stored at a node $n(r)$ reaches some threshold, then region r is split into r' and r'', and new nodes $n(r')$ and $n(r'')$ are created. The z-code of the new region r' (r'') is the z-code of r with bit 0 (1) appended.

Figure 1(c) shows the search tree corresponding to the subdivision of the space in Figure 1(b), and to simplify the presentation, the figure only contains four levels of the tree. In this figure, the solid and dotted nodes represent *active* and *inactive* nodes, respectively. An inactive node n_I is simply a node that has yet to be created by the search tree; that is,

neither n_I nor its descendants store any subscriptions. An active node n_A that is overloaded can activate its children nodes and delegate its subscriptions to these nodes.

In Figure 1(c), subscriptions S_1 and S_2 are stored at nodes 01 and 00, respectively. If node 01 becomes overloaded, it can activate nodes 010 and 011 and move its subscriptions to these nodes. S_1, for instance, would have to be moved to both nodes 010 and 011 since S_1 intersects the regions corresponding to both these nodes. On the other hand, when node 00 becomes overloaded, S_2 only needs to be moved to its left child, node 000, since S_2 does not intersect the region indexed by its right child, node 001. The intersection of subscriptions and regions can be determined visually from Figure 1(b) or algorithmically by computing whether the z-code of either the subscription or region is a prefix of the other. Figure 4 shows the state of the tree after both nodes 00 and 01 have delegated their subscriptions to their children. Note that this figure only shows the left subtree of the root, and has an additional level compared to Figure 1(c).

Notice that subscriptions are only stored at *active* leaf nodes in the tree. So even a subscription with very general constraints will only be stored in a fraction of the total (active and inactive) nodes in the tree.

2) Mapping multidimensional indexing to a DHT: Each region r with z-code z has a corresponding node $n(r)$ in the search tree. The information of each node (e.g., subscription table, status of child nodes (whether active or inactive), etc.) is stored at the peer $p(r)$ (as determined by the DHT) in the network. It is important to note that given the z-code of a region r, peers can independently find $p(r)$.

Figure 1(d) shows the peers organized in an identifier circle [32], indicating the peers responsible for the active nodes in the DMM search tree in Figure 1(c), and directed edges between these peers illustrating the parent-child relationships in the search tree. Notice that neighbors in the search tree are not necessarily neighbors in the DHT. Also, note that the search tree edges are only implicitly stored (i.e., they result from applying the nodes' z-codes as search key to the DHT).

We start out with a single root peer $p(S)$ for the entire space. In order for both publishers and subscribers to find this root, $p(S)$ can be the hash of the attributes in the system (which is known to all peers). This global requirement is removed in Section IV-C.

Subscriptions are sent to the root peer $p(S)$ and flow down to the appropriate leaf nodes. To avoid the root node becoming overloaded, an event e flows *up* the tree to find matching subscriptions. We can find the smallest region r that encloses e, and send e to $p(r)$. If $n(r)$ doesn't exist in the binary search tree, $p(r)$ forwards e to its first ancestor $p(r')$ in the tree.

3) Algorithm: The propagation of a subscription goes through two states. First, the subscription goes through the *finding tree* state in which it travels from peer to peer, mediated via DHT routing on the underlying substrate, towards the DMM tree (i.e., the peer in the network that holds the search tree's root node). In this state, every peer, along the subscription propagation path, stores an entry in its subscription table; the reverse path of these subscription entries is used to build the multicast tree used to disseminate publications to subscribers (cf. Section V-A). Once the subscription has found a node in the search tree, it then goes into the *finding leaf* state. In this state, the subscription travels up or down the tree searching for an active leaf node; the subscription is not stored in the subscription table in this state until it reaches an active leaf node.

Publication propagation is similar to that of subscriptions but has three states: *Finding tree*, *finding leaf*, and *multicasting*. Every hop of a publication in the finding tree state looks for subscriptions in the subscription table that match the publication. If one or more matches exist, a copy of the publication is made and sent to matching subscribers. This publication copy goes into the multicasting state, while the original continues in the finding leaf state. As with subscriptions, once a publication reaches a node in the DMM tree, it goes into the finding leaf state to search for an active leaf. No multicasting is done in the finding leaf state until the publication reaches an active leaf node.

The propagation of the publications from Figure 1(a) are illustrated in Figure 4. Notice how the publications are first routed to the node corresponding to their z-code. For publications without ranges, this will always be a leaf node (which may be inactive). Then, the publication traverses up or down the tree to find an active leaf, where it is matched with any matching subscriptions, and then multicast to the subscribers. For example, P_1 with z-code "0100" is routed via the DHT to the peer that holds information about the leaf node for the region enclosing the publication. In this case, the leaf for P_1 is inactive, which triggers the peer to pass P_1 up the search tree to the peer holding information about the leaf's parent node. This is accomplished by routing P_1 via the DHT on the key (z-code) "010" (stripping off the last bit of the z-code to find the enclosing region). This node is active, a matching subscription is found in the routing table, and the publication is dispatch from there via the DHT to the subscriber. Notice that the DMM matching algorithm matches all attributes in a publication simultaneously, instead of matching each attribute separately and combining the results.

B. Discussion

The DMM algorithm supports any attribute types that satisfy the following properties. These properties are not overly restrictive and the common data types, such as integers, floating point numbers, and strings can be supported.

First, attributes have a known domain. For example, a "weight" attribute may be known to only have values in the range $[0, 300]$. If the domain is not specified, the bounds of the data type are used by default. For example, the default bounds of a 32-bit signed integer are $[-2^{31}, 2^{31} - 1]$. There is also a known granularity for each attribute value. For example, a "length" attribute might have a granularity of 0.001m. The granularity is used to terminate the recursive indexing algorithm. Note that the actual values can be of finer granularity, but the values are rounded (temporarily) to the finest granularity for indexing purposes. As with the attribute domain, if no granularity is specified, the maximum precision of the data type is used as the default granularity. For

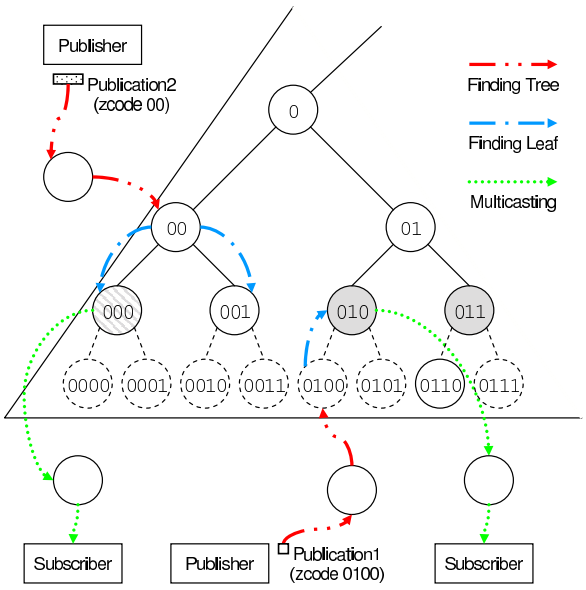


Fig. 4. Publication routing

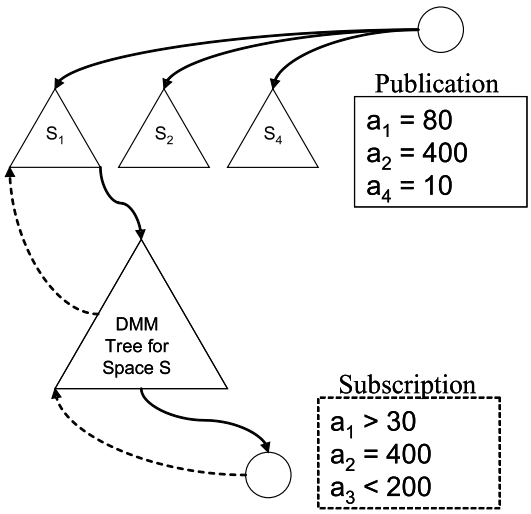


Fig. 5. DMM-AR message propagation

example, the finest granularity of a 32-bit integer attribute is one unit, and the granularity of a 6-digit floating point number is 0.000001. String attributes have a maximum number of characters. (There may be unbounded length string attributes in an event, but these attributes are not matched by DMM, although they can be filtered while the event is being multicast.) This allows string values to have known bounds. For example, an attribute with a maximum string length of four, has a lowest string value of "a" and highest value of "zzzz". Finally, there is a known global order of the attributes in the system. The global order can simply be the lexical ordering of all attribute names.

Any operators that constrain an attribute's value to a closed range are supported. This includes the Boolean operators $=, <, >, \leq, \geq$. Also operators such as string prefixes (e.g., "name = ab*") are allowed in subscriptions, since such constraints can be mapped to a closed range within the dimension corresponding to the "name" attribute.

Note that the DMM search tree only grows vertically when nodes become overloaded and delegate subscriptions to child nodes. This means that for a given workload, larger attribute ranges will not increase the height of the search tree. The growth of the tree is determined by the total number of subscriptions managed in the system and by the threshold that triggers a node to activate its child nodes in the search tree. However, the *maximum* height of the search tree grows logarithmically with the size of the attribute's domain, and since publications search the tree from the leaves up, the publication path length will increase logarithmically. This property also applies to the DMM-AR algorithm discussed next.

C. DMM with attribute roots (DMM-AR)

We now extend the matching capabilities of DMM to no longer need a global schema, and therefore not suffer from the problems associated with indexing high dimensional spaces.

A multidimensional space S is created opportunistically for any combination of attributes that appear in a subscription. These spaces are created dynamically as needed when the system sees new attributes, so there is no need to pre-specify all the attributes in the system. In each S with $d > 1$ we choose an attribute a_x to be the "primary" attribute of S. A node $n(r)$ in the tree representation of S is mapped to a peer by using the underlying DHT to hash the concatenation of the names of the attributes of S and the z-code of r. The root node of S has a null z-code.

Consider a subscription s with predicates containing attributes a_1, a_2, and a_3. We construct our multidimensional structure with a three dimensional ($d = 3$) space S for these attributes, with say a_1, as the *primary attribute*. Subscription s is stored in this structure as described earlier. The node in space S that ends up storing subscription s then sends s to the one-dimensional space S_1 consisting of attribute a_1. This ends up creating a subscription chain from the subscriber to a node in structure S to a node in structure S_1. This propagation of s is shown as a dashed line in Figure 5.

Events are sent to the one-dimensional space for each attribute in the event. Consider event e with $a_1 = v_1$, $a_2 = v_2$ and $a_4 = v_4$. The publisher calculates the smallest region r_1 in the space S_1 that encloses v_1 (based on the pre-specified granularity), and sends e to $p(r_1)$. This will result in e being sent to the node in space S_1 that contains subscriptions whose a_1 attribute matches e. This node in turns sends e to the node in space S that contains subscriptions whose a_1, a_2, and a_3 attributes match e. This node then finally multicasts e to the known subscribers. The publisher, also sends e to some $p(r_2)$ and $p(r_4)$, and e might be forwarded to some other subscribers from there. The propagation of such a publication is illustrated in Figure 5.

During subscription propagation, the optimal choice of the primary attribute a_p among a set of attributes a_i ($i = 1..n$) is non-trivial. To maximize filtering, that is, to minimize the propagation of publications, a_p should be the most selective

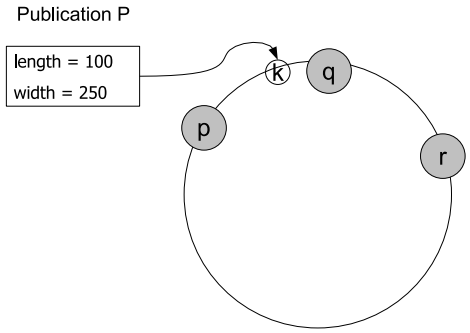


Fig. 6. Replicas

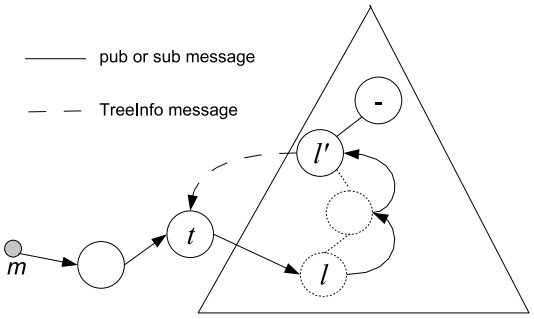


Fig. 7. TreeCache optimization

attribute. In other words, it should match the fewest number of publications. At the same time, it is desirable if the predicates associated with a_p exhibit a lot of covering. This way the propagation of subscriptions, and hence subscription state, is reduced. Hence, to maximize filtering, each subscription should select its most selective attribute, but to maximize covering, all subscriptions should select the same attribute. Due to such conflicting goals, as well as the need for global knowledge of the subscription and publication distributions, the optimal selection of the primary attributes is non-trivial.

It is important to emphasize that the DMM-AR algorithm creates spaces only for combinations of attributes that appear in subscriptions in the system. Furthermore, the creation of a space is very inexpensive—simply an additional entry in the subscription table at a node. For example, the first subscription s with attributes a_1 and a_2 is forwarded to the root peer $p(S)$ where S is the space consisting of attributes a_1 and a_2. This root peer "constructs" space S simply by storing subscription s in its subscription table.

V. Additional design decisions

Section IV described our distributed pub/sub matching algorithm. We now complete the design by discussing other concerns in a pub/sub system including the need to multicast publications to subscribers and fault-tolerance issues.

A. Multicast

We adapt multicasting techniques from traditional content-based pub/sub systems in the following manner.

Consider a subscription s from a subscriber at peer p_1 that is stored at n peers p_i in the DMM tree, where $1 \leq i \leq n$. Note that s follows a single path from p_1 while it is in the *finding tree* state after which point the path fans out to each p_i. Therefore, this first portion of the path from p_1 to any p_i will be common for all p_i. Let p_k be the last peer in the path that is in the *finding tree* state.

To achieve multicast, every peer in the path up to p_k stores (s, p_j) in a subscription table T_S to remember the peer p_j that sent it the subscription. Additionally, each peer p_i that indexes the subscription in the DMM tree, stores (s, p_k) in its T_S. Notice that only the last peer in the *finding leaf* state of the path to p_i adds an entry to its subscription table. These tables build a multicast tree from a DMM tree node to all subscribers that have sent a subscription to the tree. When an event e is received at a peer p_j, it forwards e to all peers p such that (s, p) is in T_S and event e matches subscription s.

B. Fault-tolerance

In a distributed pub/sub system, the only state that needs to be maintained are the subscription tables. We observe, however, that subscriptions can be retrieved from the subscriber peers, and hence even subscription state need not be fully replicated. This section designs a light-weight fault-tolerance scheme that exploits this observation.

1) Replica selection: We employ a replica selection algorithm: the replicas of a peer are chosen to be its $\mathbb{R}$ successor and predecessor peers in the identifier circle, where $\mathbb{R}$ is the number of replicas per peer. For example, in Figure 6, with $\mathbb{R} = 2$, q's replicas are p and r. Since nearby peers in the identifier circle are likely to be geographically dispersed, a replica is likely to survive localized network failures. While this choice of replicas is similar to that in DHTs such as Chord and CAN, as we describe below, we only require a fraction of the data at a node to be replicated.

2) Replication data: It was noted earlier that the only state that needs to be replicated is the subscription tables at the peers. However, the subscription table at a peer can be rebuilt from that of other peers, which in turn can eventually be rebuilt from the subscriptions at the original subscribers. Therefore, only the addresses of the peers from which a peer received its subscriptions need to be replicated. This information is cheaper to store and transmit than the subscriptions themselves.

3) Failure detection: The failure of a peer can be detected using heartbeat messages between the primary and its replicas, with a tradeoff between heartbeat frequency and failure detection speed. However, a cheaper method is possible. Consider peer q in Figure 6. Suppose publication P (which is hashed to node id k) would normally be sent to peer q in the DMM tree. Now, if q fails, P would get sent to p, the next closest peer to k. The receipt of publication P at peer p tells it that peer q has failed, and that it should take over for q. This algorithm requires at least one successor and one predecessor replica to ensure that any message originally intended for the primary will be sent to one of the replicas.

It is important to note that this failure detection algorithm also moves state to the correct peers as peers arrive and leave the system.

4) Failure recovery: When a replica p detects that a primary peer q has failed, it needs to recover the subscription state at q. It does this by requesting all of q's children to resend their subscriptions. Note that it is possible that not all of q's children will send their subscriptions to p. Some subscriptions may be

sent to a peer r that is closer to the hash of the z-code and attribute names of a subscription. This automatically ensures that subscriptions are migrated to the correct peers after a failure.

In summary, the fault tolerance algorithm described above uses a light-weight replication scheme that detects failures based on the presence of an unexpected message receipt. We call this active failure detection, as opposed to the passive failure detection used by the absence of periodic beacons from the primary peer.

C. DMM tree cache

In the DMM tree in Figure 7, a publication sent to this tree must traverse from an inactive leaf node l up until it reaches an active node l'. While this ensures that the root node is not unnecessarily overloaded with messages, the multiple hops that the publication travels can be expensive.

We introduce an optimization here that retains the benefits of searching from the leaf up, but alleviates the repeated bottom up traversal, by caching information about the DMM tree at various peers in the network.

In the TreeCache optimization, a publication or subscription message m stores the final hop t in the finding tree state. When m finally reaches the active leaf node l' in the DMM tree, the peer corresponding to node l' will send a TreeInfo message to peer t notifying it of the existence of node l'. Any future publications or subscriptions that traverse through peer t can be directly sent to node l'. Stale tree caches do not affect the correctness of the algorithm. If a publication is sent to a newly inactive node l in the tree, the regular publication handling algorithm will ensure that the publication finds an active leaf node.

The experiments in Section VI show that this optimization is effective and has little message overhead.

VI. Evaluation

The experiments are run on SimPastry, a Pastry simulator that has been used in published research on the Pastry protocol [11], [12]. SimPastry is a discrete event simulator implemented in C# that simulates a network generated from the Georgia Tech Internetwork Topology Model (GT-ITM) generator [9]. Note that only network latency costs are simulated in these experiments, under the assumption that local computation is dominated by network latencies.

The DMM and DMM-AR algorithms have been implemented on top of SimPastry. The implementation includes the matching algorithms described in Section IV and the multicast and fault-tolerance features outlined in Section V.

The main metrics are the load on the system, measured in terms of storage load on a peer, and the quality of service as seen by the subscribing entity, measured by the latency of delivering publications and the percentage of successfully delivered publications. The simulations were performed on a transit-stub network generated by the GT-ITM topology generator. In the simulations below, each router in a transit domain is connected to an average of 10 stub domains, with each stub domain having an average of 10 routers. Intra-domain connections have an average latency of 50ms, while inter-domain connection latencies average between 100ms and 500ms. In total there are 5050 transit and stub routers. Each peer randomly connects to one of the stub routers with a 1ms latency LAN connection. Unless otherwise stated, there are 5000 peers, 100 of which are publishers, and the remaining peers are subscribers with the subscriptions evenly distributed among these subscribers. To evaluate the performance under larger networks, some results under a 50 000 peer topology are also shown. It is assumed that 0.1% of the messages transmitted in the system are randomly lost.

Unless otherwise specified, the DMM-AR algorithm with subscription covering is used in the simulations. Subscriptions are beaconed every 30 s, and expire after 60 s. Tree cache information is beaconed every 15 s and expires after 30 s.

Each simulation run begins with a subscription phase during which subscription messages spaced 10ms apart are sent for each subscription in the system. 40 s after the final subscription is sent, each publisher begins to publish at some random time within 10 s, and then continues to periodically publish every 10 s for a total of 100 s. Each publication and subscription is generated as follows. There are 10 attributes in the system with each attribute having a unique name. Each attribute is an integer in the range [1,256] having finest granularity of 1; thus, each attribute has a maximum z-code of 8 bits. A subscription consists of a random choice of 1 to 5 of the 10 attributes. For each attribute, the lower and upper values are chosen randomly within the range [1,256]. A publication consists of a random choice of 1 to 10 of the 10 attributes. For each attribute, the lower and upper values are the same, and chosen randomly within the range [1,256]. Unless otherwise stated, uniform distributions are used in the randomly generated values in the experiments including the above parameters.

A. Subscription scalability

In this section we study the scalability of the algorithms with respect to the number of subscriptions in the system. The number of subscriptions is varied, and various metrics are studied.

1) Delivery rate: The most important metric is the delivery rate, which measures the percentage of publications successfully delivered to subscribers. Figure 8 shows that the DMM-AR algorithm with the TreeCache information delivers virtually all the publications. Some loss is inevitable due to message losses in the system. We do not show results with acknowledgments to make the difference in delivery rates among the algorithms more evident. The use of the TreeCache optimization improves the delivery rate in the DMM-AR algorithm. This is because the optimization reduces the number of hops a publication or subscription travels and thus reduces the likelihood of a message being lost. Furthermore, the DMM algorithm both with and without the TreeCache optimization has a worse delivery rate than DMM-AR. This is because publications travel longer paths in DMM compared to DMM-AR (see Section VI-A4) and hence are more likely to experience a send omission failure.

Figure 9 shows the same set of results under a larger topology of 50 000 peers with up to 300 000 subscriptions.

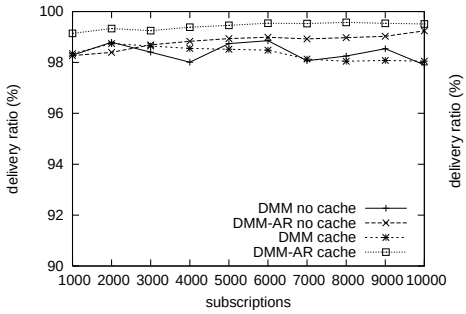


Fig. 8. Delivery ratio

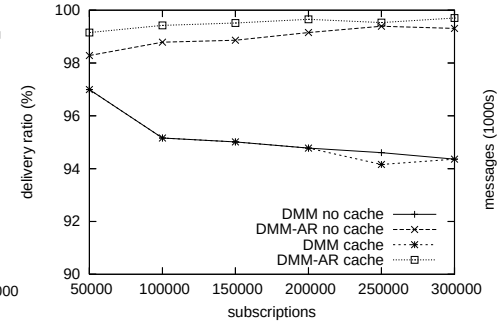


Fig. 9. Delivery ratio (50 000 peers)

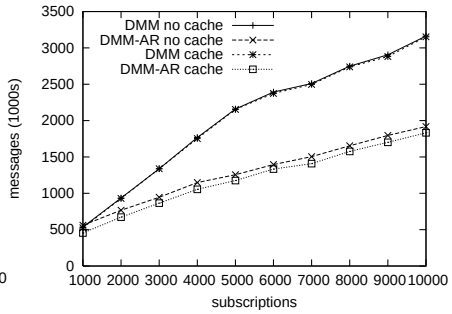


Fig. 10. Message cost

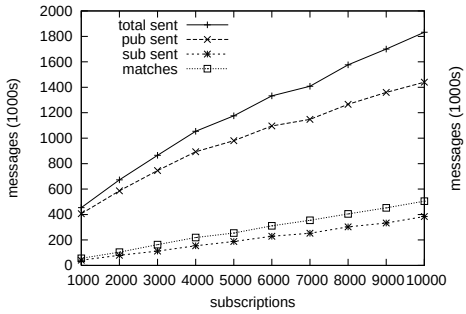


Fig. 11. Message cost (DMM-AR TreeCache)

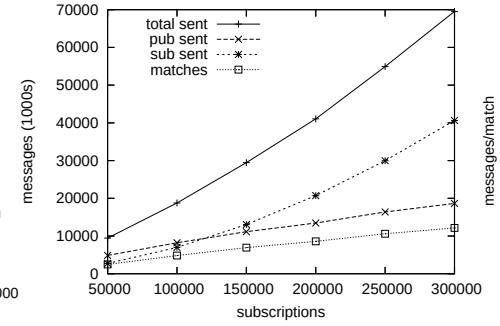


Fig. 12. Message cost (DMM-AR TreeCache with 50 000 peers)

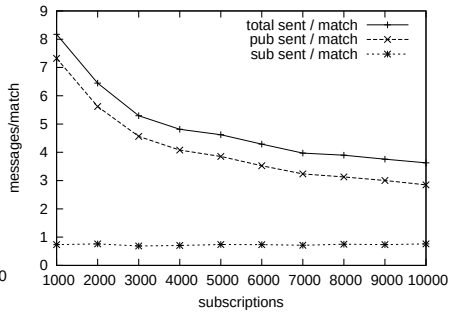


Fig. 13. Normalized message cost (DMM-AR TreeCache)

With more peers in the system, the underlying DHT requires more hops to deliver messages, and this results in a slight degradation in the delivery rate of the DMM algorithm. However, despite the longer DHT paths, the DMM-AR algorithm, especially with caching enabled, reduces the number of overlay hops a message travels to the point that it continues to deliver virtually the same delivery performance as in the smaller topology in Figure 8.

2) Message cost: Figure 10 shows the total message counts for the DMM and DMM-AR algorithms with and without the TreeCache optimization. First, note that the total number of messages grows at a slower rate than the number of subscriptions in the system, so the algorithm is scalable with respect to message cost. The DMM-AR algorithm has a much lower message cost than DMM. This is because the DMM-AR spaces are smaller, resulting in smaller trees that need to be traversed to find matching subscriptions. Also, we see that TreeCache helps to reduce the message cost in the DMM-AR algorithm, but has little effect on the DMM algorithm. The latter is because the DMM tree is much larger than the DMM-AR tree and therefore there are fewer cache hits with the TreeCache optimization. Also, the DMM algorithm suffers from a larger message cost than the DMM-AR algorithm. This again is due to the longer paths that publications take in the DMM algorithm. The TreeCache optimization does not improve this cost because, as explained in Section VI-A4, the effectiveness of the cache is diminished when information about the relatively larger DMM tree needs to be cached.

Figure 11 shows a breakdown of message costs for the DMM-AR algorithm with TreeCache. First, we note that the relative number of TreeCache messages is almost non-existent, and the periodic beaconing of subscriptions does not cause an excessive message load compared to the publications; message cost is dominated by publication messages. We note again that the message cost scales with the number of subscriptions—a ten fold increase in subscriptions from 1000 to 10 000 results in a less than five fold increase in messages.

The message costs under a larger 50 000 peer topology are shown in Figure 12. Compared to the smaller network in Figure 11, we also increase the number of subscriptions up to 300 000 but keep the number of publications constant, which explains the inversion in the relative number of publications and subscriptions in Figures 11 and 12. However, we see that the publication traffic in both cases scales with the number of subscriptions in the system.

To emphasize the scalable increase in message cost, Figure 13 plots the message cost normalized by the number of expected matches. We choose to normalize by the number of matches because we expect that a publication that matches more subscriptions will require more messages in order to be delivered to all subscribers. Figure 13 shows that with 1000 subscriptions, it requires about 7 publication messages to deliver the publication to each interested subscriber. It is important to note that this number counts the physical hops that a publication traverses; this is a very low count considering the fact that the DMM-AR algorithm requires an extra level of indirection to the attribute root, and that each overlay hop in the DHT substrate corresponds to several physical hops. The figure also shows that the incremental cost of delivering a publication to an additional subscription diminishes with the number of matching subscriptions in the system. This confirms that the DMM-AR algorithm still provides the advantages of multicast publication delivery.

3) Comparison with Hermes: Hermes [29] is a content-based pub/sub system that assigns publications and subscriptions to a topic. Publications and subscriptions are forwarded

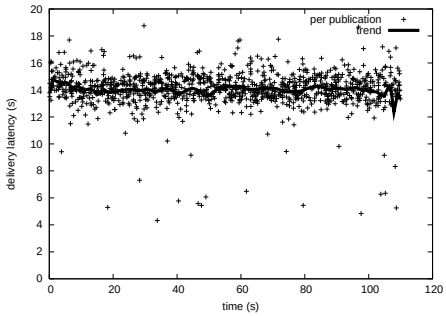


Fig. 14. DMM-AR delay without TreeCache

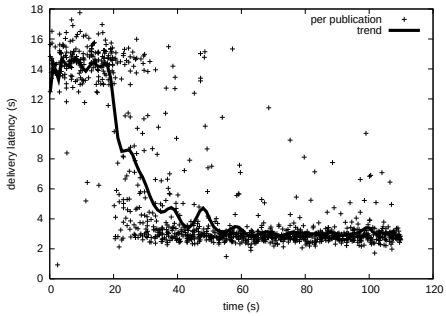


Fig. 15. DMM-AR delay with TreeCache

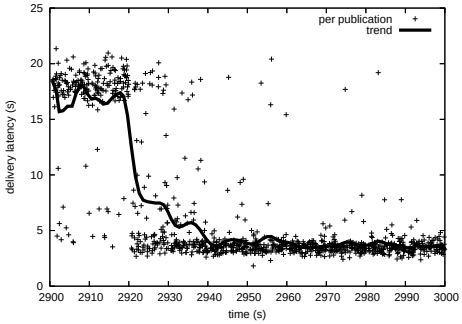


Fig. 16. DMM-AR delay with TreeCache (50 000 peers)

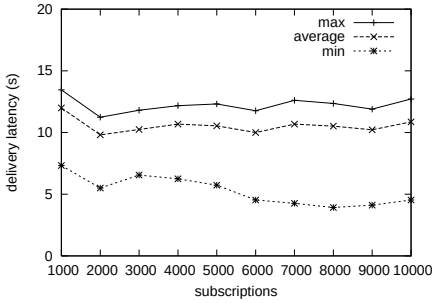


Fig. 17. DMM delay without TreeCache

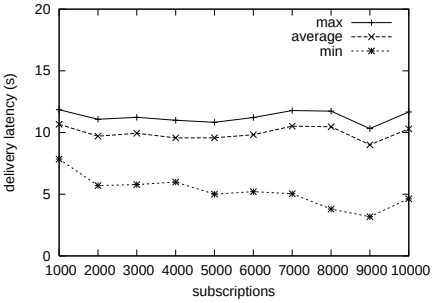


Fig. 18. DMM delay with TreeCache

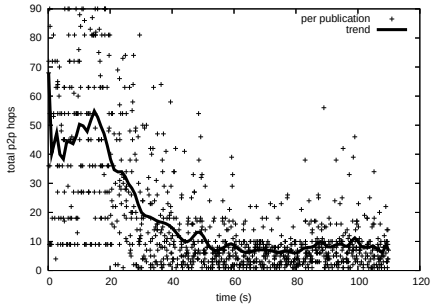


Fig. 19. Hops with TreeCache

to the topic root, as in Scribe [12]. The key difference to Scribe is that publications are filtered based on their content as they propagate along the routing paths. Although the algorithms are quite different, the message cost of Hermes with one topic is almost identical to that of the DMM algorithm with the TreeCache optimization. To avoid duplicating the results, we point the reader to Figure 10, where the Hermes algorithm's message cost is almost identical to that of the "DMM cache" case.

4) Publication delivery latency: Figure 14 shows a scatter plot of the average latency to deliver each publication to each matching subscriber with the DMM-AR algorithm. We see that the average publication takes about 15 s to be delivered. The large delay is primarily due to the need of publications having to traverse multiple hops from the bottom of the tree towards the root. This hypothesis is verified by Figure 15 which shows that the TreeCache optimization works quickly and has a large impact. It only requires about 20 s to populate the cache sufficiently to drastically reduce the delivery latency of most publications from about 15 s to about 3 s. We also see that the average delivery latency is very close to the minimum, so there is little room for improvement to this optimization.

Similar results are seen in Figure 16 where a larger 50 000 peer network is used. We expect that a larger DHT peer population will increase the average path length in the underlying DHT, and results show that without the TreeCache optimization, the delivery latency increases to about 17 s in the larger topology from about 15 s. With the optimization, the delay in the 50 000 network is reduced to about 4 s which is only marginally longer than the 3 s delay in the smaller network. As well, from the scatter plot in Figure 16, we see that, even in the much larger network, it again takes about 20 s to warm the distributed cache enough to reduce the delay to nearly the minimum.

The publication delivery delay results are primarily influenced by the number of hops publications travel in the DHT. Figure 19 shows the number of DHT hops taken by each publication in the DMM-AR algorithm with the TreeCache optimization enabled, and it has a close relationship with the delay results in Figure 15. We note such similarities between the delay and hop counts for the other algorithms as well. The average hop count of slightly less than 10 hops per publication in Figure 19 for the DMM-AR algorithm with the TreeCache enabled. Note that these are DHT hops, and include matching and delivery of the publication.

It is instructive to see how DMM performs relative to DMM-AR. Recall that the DMM algorithm constructs one large global space (and thus requires a global schema) and does not suffer from the additional step of attribute roots. Figure 17 shows the latency without the TreeCache optimization. Notice that the average latency with the DMM algorithm is about 10 s which is lower than that in DMM-AR; this is due to the absence of attribute roots, which reduces the number of hops a publication travels.

Interestingly, the use of the TreeCache optimization with the DMM algorithm has little effect; in Figure 18, the average publication delivery latency remains around 10 s. This is because the DMM algorithm sends publications towards a tree that represents a high-dimensional space. With the workload used here (10 attributes with an 8-bit z-code per dimension, leading to 80 bits in the z-code of a publication), the full tree contains 2^{80} leaf nodes.[1] A given publication is sent towards

[1]Note that, as explained in Section IV-A, while the tree *potentially* contains 2^{80} leaf nodes (and hence $2^{81}-1$ total nodes), only those that need to store a subscription will be active and need to be managed by a peer. The number of active nodes is likely a small fraction of the total nodes in the tree. For example, even if there were 2^{30} subscriptions, and we let each node only maintain 2^{10} subscriptions before delegating to a child, the fraction of active nodes is still only $\frac{2^{30}/2^{10}}{2^{81}-1} \approx 2^{-61}$.

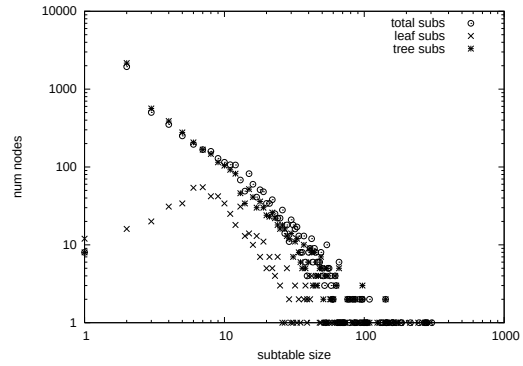


Fig. 20. Subscription state

one of these leaf nodes. Since the tree information is only cached at the hop preceding the intended destination leaf, the large number of leaves means that the tree cache is dispersed over many nodes and is not used frequently–that is, there are few cache hits

Compare this to the DMM-AR case, where publications are sent to an attribute root whose full tree only has 2^8 leaves. The DMM-AR algorithm interestingly also benefits from the fan-out of a publication to multiple attribute roots. This has the effect of increasing the publications in the system and thus more opportunities for the tree information cache to be updated.

It should be noted that although the delivery latencies in the order of seconds are high compared to pub/sub systems with a dedicated infrastructure, such latencies are not uncommon in P2P networks and are a consequence of the underlying DHT routing techniques. As a point of reference, however, database queries evaluated over a DHT offer response times of several seconds to hundreds of seconds [2], [23]. Many of the scenarios we reviewed in Section I do not require low latencies and are more concerned with large-scale, high throughput and fine-grained filtering requirements. Examples included information discovery and filtering, as in the Spotify service [33], infrastructure free file synchronization across devices and groups of end-points [25], decentralized social networking with needs for notifications [38], selective filtering, status, and presence update [39], and collaborative filtering for recommendations to peers [15]. Also, the more traditional sports ticker dissemination [24] and software patch distribution scenarios fall into the operating range that can be supported by our approach.

5) Subscription state: Figure 20 illustrates the distribution of the subscription state among the peers in the network. A peer's state is taken to be the number of subscriptions stored at that peer. The figure shows the number of peers that have a particular subscription table size. A point on the graph with an x-value of 10 and y-value of 100 means that there are 100 peers that store 10 subscriptions. Three sets of plots are shown: the total number of subscriptions, the number of subscriptions in the leaf state and the number of subscription in the tree state. Recall that subscriptions in the leaf state are stored in the DMM (or DMM-AR) tree for matching publications with subscriptions, while those in the tree state are stored to maintain multicast paths and are used to multicast publications. The figure shows that most peers have very little total state, while only a few peers have a large state. This effect is more pronounced for the tree state subscriptions compared to the leaf state subscriptions.

The skewed subscription state is due to the fact that the load balancing algorithms do not attempt to balance system load. Instead, each node individually determines when it is overloaded, and initiates subscription delegation if needed. In this way, more powerful nodes will assume greater load. This is desirable since unnecessary subscription delegation, while contributing to load balance, will increasing routing path lengths. The point is that load balance in itself is not always desirable; load balance is used here as a means to allow nodes to assume as much load as they desire.

B. Skewed message distribution

The following experiments evaluate workloads similar to that in Section VI-A but with skewed message distributions. More precisely, for both publications and subscriptions, the choice of attribute names and the values in each predicate are selected according to a Zipfian distribution characterized with an exponent $s = 1$.

The results show that the scalability of the system is not affected by a skewed subscription interest and publication content. For example, comparing the publication delivery ratios in Figure 21 with those in Figure 9, where a uniform distribution was used, we see an identical percentage of publications is delivered.

Likewise, the message overhead with a skewed distribution as shown in Figure 22 is similar to those seen earlier in Figure 12. There is slightly more publication traffic with the skewed distribution due to this workload having more matching publications (which is also plotted in Figure 22).

The publication delivery latencies were also little changed when the Zipfian workload was used, and the TreeCache optimization was just as effective. For example, the delay trends observed both before and after the TreeCache optimization in Figure 23 are comparable to those in the uniformly distributed workload in Figure 16.

C. Subscription dimensionality

In this experiment, the number of dimensions in a subscription is varied, to measure the effects of more "complex" subscriptions. Each subscription has a fixed number of predicates. A subscription with more predicates is more selective, and hence fewer publications will match it. This will, among other things, decrease the message load on the system. These secondary effects are removed by constructing a workload that has the same number of matches regardless of the number of predicates in the subscription. This is done by randomly choosing the lower and upper bounds of the first predicate in each subscription as usual, with the remaining predicates having bounds that cover the entire range. Hence, only the first predicate acts to discriminate among publications; the other predicates will always match. In addition, publications always have 10 predicates, all of whose values are chosen randomly. This workload results in an equal number of matches

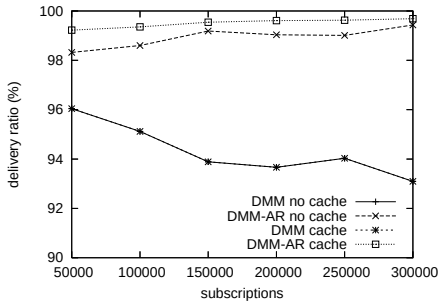


Fig. 21. Delivery ratio (Zipf distribution, 50 000 peers)

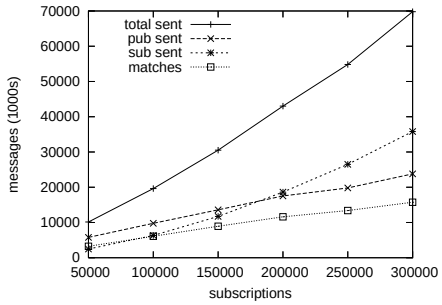


Fig. 22. DMM-AR message cost with TreeCache (Zipf distribution, 50 000 peers)

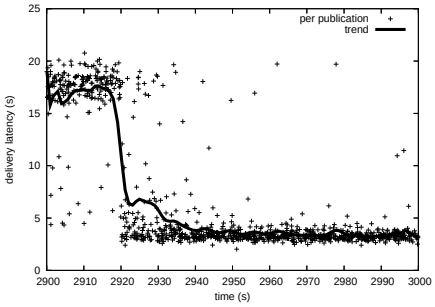


Fig. 23. DMM-AR delay with TreeCache (Zipf distribution, 50 000 peers)

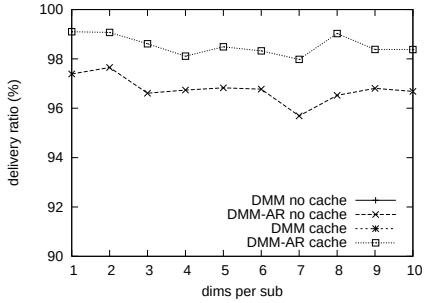


Fig. 24. Delivery ratio

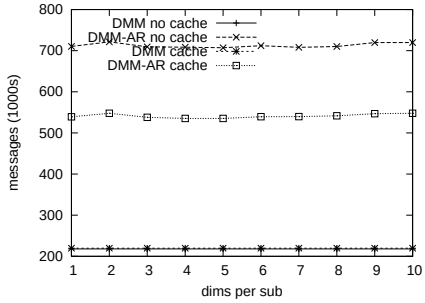


Fig. 25. Message cost

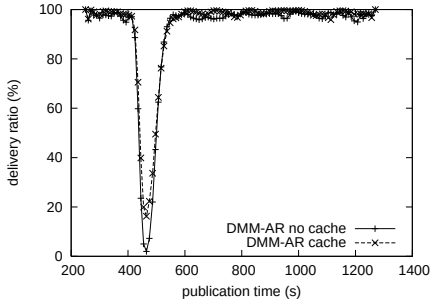


Fig. 26. Delivery ratio with failures

per subscription regardless of the number of predicates per subscription.

Figure 24 shows that the TreeCache optimization increases the delivery rate for the DMM-AR algorithm. Note that the delivery ratio for the DMM algorithms is less than 20% and does not appear on the graph. The reason for this poor performance is because the DMM trees are very large, and the long path from an inactive leaf to an active leaf in the tree makes it unlikely a message will survive the entire path without a message loss. Figure 25 shows that the DMM-AR message cost is reduced by the TreeCache optimization. We also see that increasing the number of dimensions of the subscriptions has negligible impact on the message cost.

Together, Figures 24 and 25 demonstrate that the matching algorithm is not affected by the complexity of the subscriptions. In addition, our results indicate that the publication latency does not vary greatly with increasing subscription dimension. So, the complexity of subscriptions has minimal effect on the quality of service (as measured by delivery latency) experienced by a subscriber. Also, as before, the TreeCache optimization helps to greatly reduce the delivery latency of publications.

The speed with which the TreeCache optimization reduces publication delivery latency, and the distribution of the subscription state is similar to the results in the previous section.

D. Fault-tolerance

In this section, we study the resilience of the algorithm to faults in the network. The workload consists of 1000 subscriptions, and an aggregate publication rate of 100 per minute. At the 400 s mark, 1000 of the 5000 nodes in the network simultaneously crash, that is, they no longer send or receive any messages.

Figure 26 plots the delivery ratio of the publications in the system over time. A Bezier curve is used to smooth the curve and highlight the trend. The delivery ratio is a ratio of the number of actual deliveries of a publication to the number of interested subscribers that had not crashed at the time of publication.

The results show that the correct delivery of publications resumes within about 100 s of the nodes crashing. (While in the figure it seems as though this recovery is closer to 200 s, this is an artifact of the Bezier curve approximation lagging the actual data.) This time is largely accounted for by the delay in the underlying DHT to reorganize around the faults. Note that despite the unreasonably large number of network failures—one fifth of the peers simultaneously crash—the network is still able to recover relatively quickly.

VII. Conclusions

Distributed pub/sub systems based on P2P networks can achieve scalability without dedicated infrastructure. Many P2P networks support a DHT interface. We developed an algorithm that implements a pub/sub system over a DHT, without requiring any centralized schema knowledge. This is done by mapping publications and subscriptions in the pub/sub domain into regions in a multidimensional space. This multidimensional space is indexed with a distributed search tree, which allows matching multiple pub/sub attributes simultaneously. The multidimensional index trades off storing subscriptions at multiple nodes in order to achieve a bottom up publication matching that prevents root hotspots in the index. Also, extending traditional pub/sub semantics, the matching algorithm supports publications with range values.

Our experiments show that the algorithms scale with increasing number of subscriptions. The delivery ratio and latency are negligibly affected by increased subscriptions: For example, message cost measured in terms of system-internal messages resulting from publishing does not double

as the number of subscriptions serviced doubles. Also, the message cost per match actually decreases with an increasing number of subscriptions. The search tree scales with increased subscription complexity (measured by the number of attributes in a subscription). Two fault-tolerance mechanisms—an active failure detector and periodic state refreshing—allow the algorithms to quickly recover from even a serious crash of the network. In one experiment even with one fifth of all nodes in the system crashing, the system recovered in about 100 s. Finally, an optimization to cache information about the distributed search tree quickly reduces publication delivery latency from about 15 s to nearly 3 s. Note that latencies of these magnitudes are not uncommon in P2P networks and the latencies exhibited by systems with dedicated infrastructure should not be expected. We reviewed classes of applications drawing from social networking, selective information dissemination, and infrastructure free file synchronization that match these latency requirements.

REFERENCES

[1] Karl Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *COOPIS*, 2001.
[2] Karl Aberer, Anwitaman Datta, Manfred Hauswirth, and Roman Schmidt. Indexing data-oriented overlay networks. In *VLDB*, 2005.
[3] Ioannis Aekaterinidis and Peter Triantafillou. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *IEEE ICDCS*, 2006.
[4] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.
[5] Sébastien Baehni, Patrick Th. Eugster, and Rachid Guerraoui. Data-aware multicast. In *DSN*, 2004.
[6] Roberto Baldoni, Roberto Beraldi, Leonardo Querzoni, and Antonino Virgillito. Efficient publish/subscribe through a self-organizing broker overlay and its application to SIENA. *The Computer Journal*, 2007.
[7] Roberto Baldoni, Carlo Marchetti, Antonino Virgillito, and Roman Vitenberg. Content-based publish-subscribe over structured overlay networks. In *ICDCS*, 2005.
[8] Anthony Baxter, Jochen Bekmann, Daniel Berlin, Soren Lassen, and Sam Thorogood. Google Wave Federation Protocol Over XMPP. Section 3.1: Connection Initiation and Lifetime, 2009.
[9] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6), 1997.
[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC*, 2000.
[11] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, 2003.
[12] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), 2002.
[13] Lucy Cherkasova. Next generation data centers: Management and configuration challenges. In *ICDCS (Invited talk)*, 2010.
[14] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2), 2008.
[15] R. Delaviz, J. A. Pouwelse, and D. H. J. Epema. Targeted and scalable information dissemination in a distributed reputation mechanism. In *ACM STC*, 2012.
[16] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 2003.
[17] Francoise Fabret, H.-Arno Jacobsen, Francois Llirbat, Joao Pereira, Kenneth Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD*, 2001.
[18] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.
[19] Kevin Graham. Hedging your bets: Currency fluctuations & the supply chain. www.supplyexcellence.com, 2008.
[20] GS1. www.gs1.org/docs/gdsn/gdsn_brochure.pdf.
[21] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: Content-based publish/subscribe over P2P networks. In *Middleware*, 2004.
[22] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. In *Proc. USITS Conf.*, 2003.
[23] Ryan Huebsch, Brent N. Chun, Joseph M. Hellerstein, Boon Thau Loo, Petros Maniatis, Timothy Roscoe, Scott Shenker, Ion Stoica, and Aydan R. Yumerefendi. The architecture of PIER: An internet-scale query processor. In *CIDR*, 2005.
[24] IBM. IBM serves on demand solutions at the Australian Open. http://www-8.ibm.com/e-business/au/australianopen/pdf/australian_open_case_study.pdf.
[25] BitTorrent Inc. BitTorrent Sync. Accessed May 2013: http://labs.bittorrent.com/.
[26] Rich Miller. Who has the most web servers? http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/, July 2010.
[27] Vinod Muthusamy. Infrastructureless data dissemination: A distributed hash table based publish/subscribe system. Master's thesis, University of Toronto, Toronto, 2005.
[28] Vinod Muthusamy and Hans-Arno Jacobsen. Small-scale peer-to-peer publish/subscribe. In *P2PKM Workshop at MobiQuitous*, 2005.
[29] Peter R. Pietzuch and Jean Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In *DEBS*, 2003.
[30] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.
[31] John Reumann. Pub/Sub at Google. CANOE Summer School, 2011.
[32] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.
[33] Vinay Setty, Gunnar Kreitz, Roman Vitenberg, Maarten van Steen, Guido Urdaneta, and Staffan Gimker. The hidden pub/sub of Spotify. In *ACM DEBS*, 2013.
[34] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1), 2003.
[35] David Tam, Reza Azimi, and H.-Arno Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *DBISP2P (co-located with VLDB 2003)*, 2003.
[36] Wesley W. Terpstra, Stefan Behnel, Ludger Fiege, Andreas Zeidler, and Alejandro P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.
[37] Tibco. TIBCO software chosen as infrastructure for NASDAQ's supermontage, 2001. ACCESSED May 2013: http://www.prnewswire.com/news-releases/tibco-software-chosen-as-infrastructure-for-nasdaqs-supermontage-72106332.html.
[38] Tent (TM). Distributed social networking protocol. ACCESSED May 2013: https://tent.io/.
[39] XMPP. XMPP Extensible Messaging and Presence Protocol. ACCESSED May 2013: http://xmpp.org/.
[40] Xiaoyu Yang, Yingwu Zhu, and Yiming Hu. A large-scale and decentralized infrastructure for content-based publish/subscribe services. In *ICPP*, 2007.
[41] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE JSAC*, 22(1), 2004.

Vinod Muthusamy is a member of the Service Integration and Analytics Group at The IBM T.J. Watson Research Center. His research efforts are primarily focused in the areas of process and service management, and event processing system. In the former his interests include process mining, distributed workflow engines, and service composition. The latter includes the language models, matching algorithms, and distributed protocols for publish/subscribe systems.

Hans-Arno Jacobsen is a professor of Computer Engineering and Computer Science and directs the Middleware Systems Research Group. His principal areas of research include the design and the development of middleware systems and distributed systems. His research revolves around publish/subscribe, content-based routing, event processing, and aspect-orientation.