

Infrastructure-less Content-Based Publish/Subscribe

Vinod Muthusamy[†] and Hans-Arno Jacobsen^{†‡}

Middleware Systems Research Group

[†]Department of Electrical and Computer Engineering

[‡]Department of Computer Science

University of Toronto

{vinod,jacobsen}@eecg.toronto.edu

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 grows as the number of participants increases, scalability can be achieved without managing or deploying additional infrastructure. This paper proposes an algorithm for supporting content-based publish/subscribe in which subscriptions can specify a range of interest, and publications a range of values. The algorithm is built over the Pastry distributed hash table and is 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. Simulations support the scalability and fault-tolerance properties of the algorithm.

I. INTRODUCTION

This paper addresses the problem of developing a fully content-based publish/subscribe (pub/sub) system on top of a structured peer-to-peer (P2P) overlay network. The objective is to support infrastructure-less data dissemination applications at Internet scale. Similar objectives are set forth by projects such as PIER [21] and PePeR [14]. PIER aims to build traditional database functionality on top of a P2P network, while PePeR supports range queries of data tuples stored in a P2P network. However, neither of these approaches realize pub/sub semantics, which are fundamentally different from database query semantics. In pub/sub, a large number of “queries” must be continuously matched against a stream of incoming data. Unlike databases, which index the data, in pub/sub, it is the queries that must be efficiently indexed in the P2P network. Projects such as Scribe [13], support the less expressive channel-based pub/sub model on top of a P2P network. However, the channel-based model is severely restricted and can not encompass the content-based model underlying our work.

Content-based pub/sub [17], [16] is a model whose proven scalability and ability to decouple data producers from consumers lends itself well to large-scale information dissemination applications on the Internet. There has been much research regarding both pub/sub matching algorithms [17], [3], [32], [15] and distributed pub/sub routing protocols [10], [7].

IBM, for example, used a pub/sub system to deliver real-time tennis match scores to millions of users around the world [22]. Another possible application is an auction service such as eBay to notify buyers and sellers of bids on certain

items. Other applications that can be built with the pub/sub model include distributed gaming, the distribution of software patches, the synchronization of distributed caches, and the customization of content on Web portals. Common to these applications is the selective dissemination of data to a very large number of geographically scattered users.

Large-scale pub/sub applications currently require a large company’s resources to deploy and administer [22]. 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 the Web site contents across the Internet. The inability or unwillingness of many large corporations to perform this task themselves underscores the complexity of the Akamai infrastructure management.

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-less scalability. In addition, the design self-organizes to adapt to bottlenecks and faults so no personnel is required to administer the network.

This paper develops algorithms for implementing content-based pub/sub semantics on top of a distributed hash table (DHT) peer-to-peer overlay network. The challenge is that the DHT interface only supports exact name lookups. The problem now becomes one of supporting content-based pub/sub matching semantics on top of the DHT interface. Interests in content-based pub/sub, expressed through subscriptions, contain a conjunction of predicates, where a predicate can express interest ranges, constraints, or Boolean conditions over value ranges. Supporting these kinds of queries solely based on name look-up is a difficult problem. For example, consider a subscription with a constraint “alert = warning”. A hash table is ideally suited to such exact value lookups. On the other hand, a range constraint such as “price < 10 AND alert = warning” requires separate lookups for each discrete value less than 10. This can be prohibitively expensive or impossible when the constraint is over a floating point data type.

The main contributions of this work include the following:

- 1) A distributed content-based pub/sub matching algorithm is developed based on 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.

- 2) An extension of the above algorithm is devised that imposes no fixed global schema on the system. This is a key benefit of this protocol as opposed to related approaches.
- 3) The above algorithms are evaluated in a simulation environment and demonstrate the scalability and fault-tolerance properties of the algorithms.

Section II presents a background on the pub/sub model and P2P networks. Section III describes related work in building pub/sub systems and databases over a DHT. In Section IV, the distributed pub/sub matching algorithm is developed, and Section V describes 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 a conjunction of {attribute, operator, value} tuples, where the operator can be one of {=, <, >, ≤, ≥}. 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 [28], [26], [24], [31], [1], [19], called structured P2P networks, are based on a distributed hash table (DHT) interface. These DHTs are self-organizing, load balanced, fault-tolerant, and provide statistical guarantees on the bandwidth usage and node state requirements.

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 [26] 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 around the identifier circle.

Pastry 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.

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 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, as well as protocols that attempt to support database query semantics on P2P networks.

A. P2P publish/subscribe

Scribe [13] 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 [23] is a content-based pub/sub system built over the Pastry DHT. It essentially assigns a channel to each publication and subscription and the matching algorithm degenerates to that of Scribe. Unlike the algorithm in this paper, Hermes' initial matching algorithm does not discriminate based on content, and does not address the issue of an overloaded channel root peer.

Terpstra [30] 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.

Tam et al. [29] map content-based pub/sub to a channel-based one as in Scribe. 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 specified. Each publication and subscription is mapped to several index digests, one for each index. An index digest is a channel name comprised of 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, yet the indices need to be manually specified. Furthermore, manually specified, globally known indices are contrary to the P2P philosophy of minimal administration.

Meghdoot [18] 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. We feel that this is the wrong approach: pub/sub workloads typically have many more publications than subscriptions, and it does not make sense to optimize subscription state (a subscription is only stored at one peer in Meghdoot) at the expense of publication matching load and delay (publications are routed to multiple peers).

Baldoni et al. [6] 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, [6] requires a global static schema of known pub/sub attributes.

B. P2P databases

Motivated by similar arguments as for developing a pub/sub system over a DHT, such as infrastructure-less scalability, there have been attempts to build a relational database on top of a DHT P2P network. However, techniques from these database solutions are not readily applicable to pub/sub. The database problem of finding all data that match a specified query is in many ways the dual of the pub/sub problem of finding all “queries” (subscriptions) that match a given “data” (publication).

PIER [21], [20] is a database query engine built over a DHT interface. PIER relaxes traditional relation database semantics such as guaranteed consistency and atomic transactions in order to achieve a massively distributed database.

A technique to perform efficient database range queries on data stored in a DHT is developed in [4]. Intervals of the range being indexed are assigned to nodes in the network, and nodes are globally ordered in a d-dimensional CAN DHT [24]

such that neighboring nodes in the CAN network are assigned neighboring intervals. To process queries, requests are routed, in a controlled or brute-force manner, to those nodes whose interval intersects the query range.

In PePeR [14], each node stores records that fall within that range assigned to that node. Strategic links are maintained among the nodes to facilitate traversing to a node with the desired range. A limitation is that only one attribute can be indexed by this structure.

In [27], an algorithm to cache queries so they may be reused is devised. It creates a $2n$ dimensional CAN space to handle queries on a schema with n attributes. Consequently, it requires a fixed schema. As well, although not shown experimentally, it is not designed for cases when the number of attributes in a query is much smaller than n , which is probably common in many database applications.

Mercury [8] assigns sets of nodes to be *hubs* for each attribute in the system. These hubs index data tuples containing that attribute, and handle queries with that attribute. A limitation of the Mercury algorithm is that a multi-attribute query is decomposed into a set of single attribute queries that must be processed sequentially or in parallel.

In [2], the design of a P2P database focuses on issues of load balance and constructing large distributed indexes quickly. This differs from the traditional assumption that data is added to the database, and hence indexed, slowly over time. In the pub/sub case, we assume that the items to be indexed (the subscriptions) are stable relative to the frequency of publications.

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. Pub/Sub multicast is a simpler problem and is addressed in Section V-A.

The channel-based pub/sub problem has been addressed in P2P networks without the use of static trees [13], [25], [33]. However, the techniques used in channel-based P2P pub/sub cannot be trivially extended to content-based pub/sub in P2P networks. The problems become evident when subscriptions with range predicates are used.

Despite the benefits of structured P2P networks, it is non-trivial to build a content-based pub/sub system over a DHT. In particular, a hash table is not well suited for performing range queries. It is typically necessary to “walk” the range to find all matching entries in the hash table. For instance, a lookup for all keys in a hash table with value between 1 and 10 requires individual lookups for keys 1, 2, ..., 10. This problem is exacerbated when the data items are continuous (floating point) values. Finally, range queries in a distributed system introduce issues of data placement and query routing.

A. Distributed multidimensional matching

We develop a distributed data structure that can match multiple attributes simultaneously. This algorithm, referred to

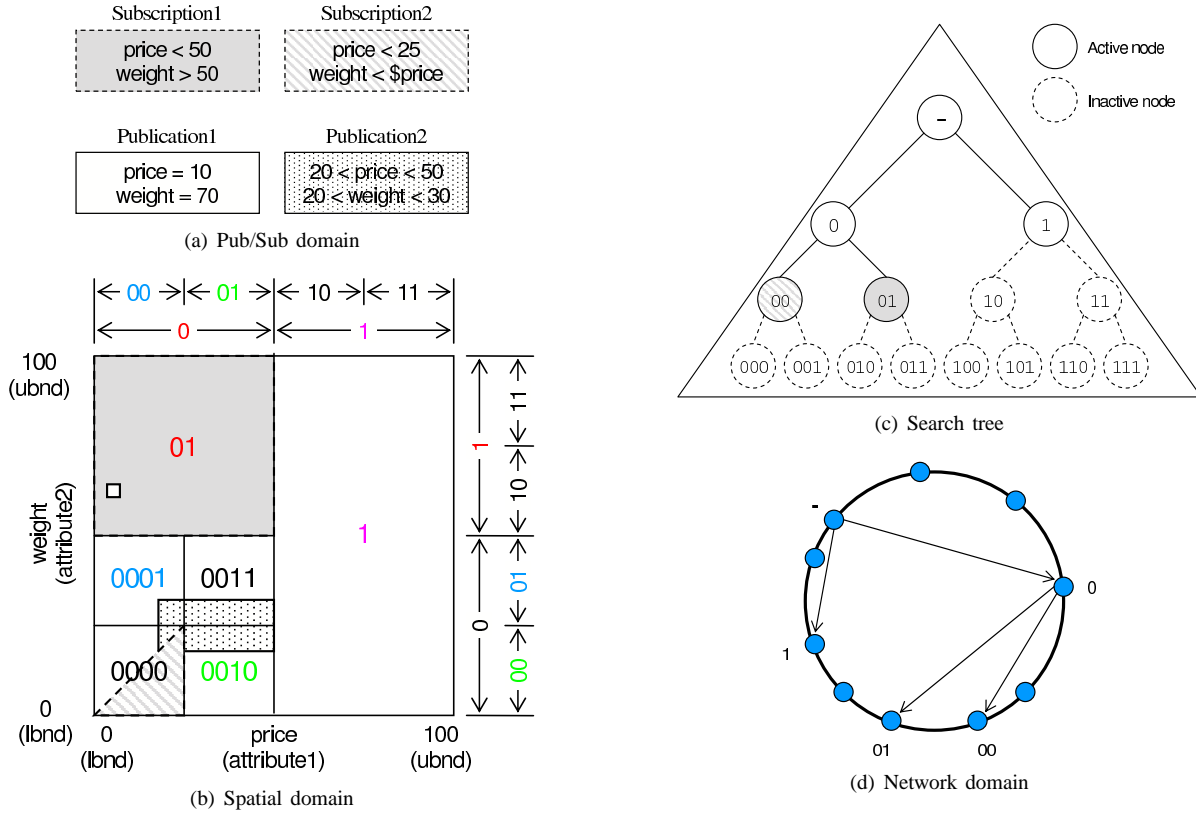


Fig. 1

MAPPING FROM PUB/SUB TO SPATIAL TO NETWORK DOMAIN

as distributed multidimensional matching (DMM), maps the pub/sub matching problem to a multidimensional indexing problem.

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 now be bound to another attribute’s value in the subscription. For example, a subscription can express interest for publications with “height > 3 ” and “length $< 2 * \text{height}$ ”.

1) *Mapping pub/sub to multidimensional indexing*: As a running example, we will use the four subscriptions and publications in Figure 1(a). Notice in the second subscription the ability to constrain the value of the “weight” attribute to the value of the “price” attribute. Also notice that the second publication allows attributes to have a range of values instead of a single value. That is, unlike traditional pub/sub semantics that require publications to be a set of {attribute, value} pairs, we allow publications to be a set of {attribute, operator, value} tuples. This allows for more expressive publication semantics. For example, a temperature reading can easily specify the precision of the measurement, by indicating a range of values for the temperature.

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 and their direction cycles through the d possibilities. 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. Figure 2 presents the algorithm to convert an integer attribute to a z-code. 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 of a 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.

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

Algorithm *IntAttributeToZCode*(*lval, uval, lbnd, ubnd*)
 (* Value is [*lval,uval*] and bounds are [*lbnd,ubnd*] *)

1. $l \leftarrow lbnd$
2. $u \leftarrow ubnd$
3. $zc \leftarrow \text{nil}$
4. $lastIter \leftarrow \text{false}$
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** $l = \lceil m \rceil$
12. $zc \leftarrow zc.Append(1)$
13. **else** (* Doesn't fit in either half. *)
14. **stop**
15. $lastIter \leftarrow l == u$
16. **until** $lastIter$
17. **return** zc

Fig. 2

Z-CODE OF AN INTEGER

Algorithm *AttributesToZCode*(*attrs*)
 (* Return the z-code of the specified attributes *)
 (* Determine the minimum z-code of all attributes. *)

1. $minlen \leftarrow MinZCodeLength(attrs)$
2. $minlen \leftarrow MinZCodeLength(attrs)$
- 3.
4. (* Weave zcodes into one. *)
5. $zc \leftarrow \text{nil}$
6. **for** $i \leftarrow 0$ **to** $minlen - 1$
7. **do for** $attr \in attrs$
8. **do** $z \leftarrow ZCodeOf(attr)$
9. $bit \leftarrow GetBit(z, i)$
10. $zc \leftarrow zc.Append(bit)$
11. **return** zc

Fig. 3

Z-CODE OF A SET OF ATTRIBUTES

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 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 $z_w(r) = 00$. As mentioned above, $z(r)$ is derived by interleaving $z_p(r)$ and $z_w(r)$.

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 leafs 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 subscription to these nodes.

In Figure 1(c), Subscription1 and Subscription2 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. Subscription2, for instance, would have to be moved to both nodes 010 and 011 since Subscription1 intersects the regions corresponding to both these nodes. On the other hand, when node 00 becomes overloaded, Subscription2 only needs to be moved to its left child, node 000, since Subscription2 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 addition 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 tree. The information of each node 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 [26], indicating the peers responsible for the active nodes in the DMM search tree in Figure 1(c), and ordered edges between these peers illustrating the parent-child relationships in the tree. Notice that neighbors in the search tree are not necessarily neighbors in 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 from 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 towards the DMM tree. In this state, every peer along the subscription path stores an entry in its subscription table; the reverse path of these subscriptions is used to build the multicast tree. Once the subscription has found a node in the 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 matches 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 it 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.

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. 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 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.

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.

We create a multidimensional space S for every combination of attributes in the system. 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 $d = 3$ 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 subscription is shown 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 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. Determining the optimal attribute root is left for future work.

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.

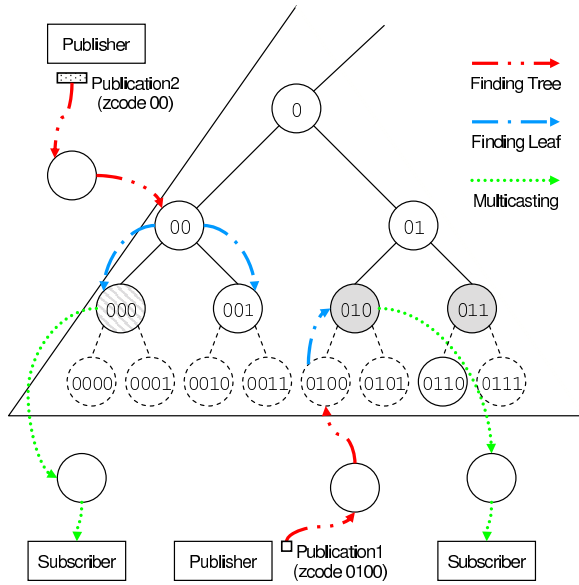


Fig. 4
PUBLICATION ROUTING

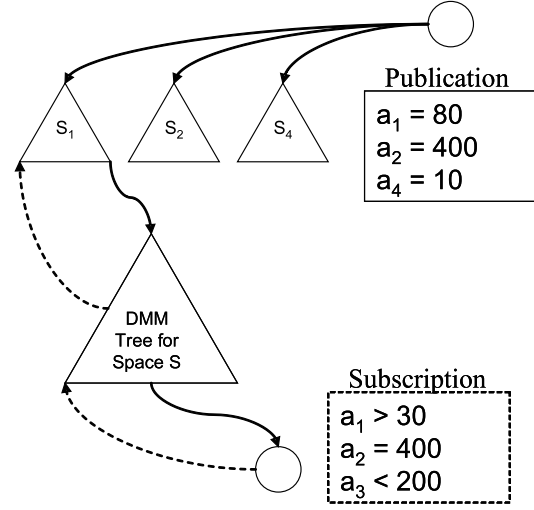


Fig. 5
DMM-AR MESSAGE PROPAGATION

V. OTHER DESIGN ISSUES

Section IV described a 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 borrow multicast techniques from traditional content-based pub/sub systems such as Siena [11].

Consider a subscription s from subscriber p_0 that is stored at k peers p_{h_j} in the DMM tree, where $0 \leq j < k$. The path of s from p_0 to p_{h_j} is $p_{0_j}, \dots, p_{i_j}, \dots, p_{h_j}$. Note that in the DMM algorithm, $p_{i_j} = p_{i'_j}$ for every $0 \leq i, i' < h$ and $0 \leq j, j' < k$. That is, s follows a single path until the hop before p_{h_j} at which point the path fans out to each p_{h_j} . So, without loss of ambiguity, p_i is used to refer to any p_{i_j} .

To achieve multicast, every peer p_{i_j} where $0 < i \leq h$ and $0 \leq j < k$, stores (s, p_{i-1_j}) in a subscription table T_S to remember the peer that sent it the subscription. These tables build a multicast tree from a DMM tree node to all subscribers that have sent an event to the tree. When an event e is received at a peer p_{i_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.

1) *Replica selection*: We employ a replica selection algorithm: the replicas of a peer are chosen to be its r successor and predecessor peers in the identifier circle, where r is the number of replicas per peer. For example, in Figure 6, with $r = 2$, 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 not necessarily that 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

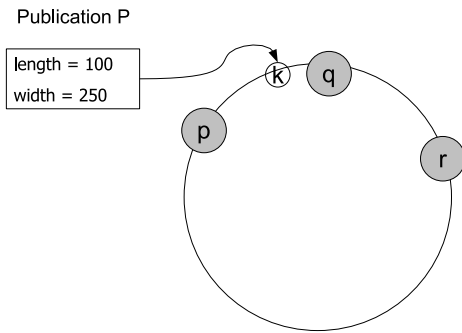


Fig. 6
REPLICAS

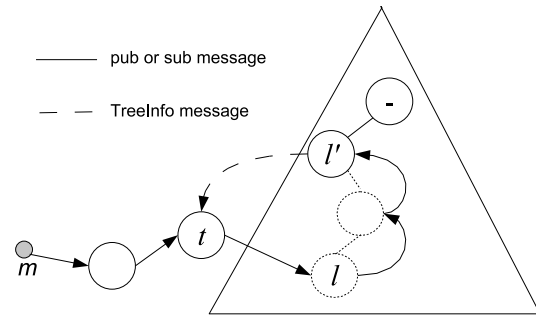


Fig. 7
TREECACHE OPTIMIZATION

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 [13], [12]. SimPastry is a discrete event simulator implemented in C# that simulates a network generated from the Georgia Tech Internet 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.

An implementation of the DMM and DMM-AR algorithms has 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 seen by the user, 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. 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 beacons every 30s, and expire after 60s. Tree cache information is beacons every 15s and expires after 30s. Each simulation run begins with a subscription phase during which subscription messages spaced 10ms apart are sent for each subscription in the system. 40s after the final subscription is sent, each publisher begins to publish at some random time within 10s, and then continues to periodically publish every 10s for a total of 100s. 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].

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 loss. 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-A.3) and hence are more likely to experience a send omission failure.

2) *Message cost*: Figure 9 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 sub-linearly with 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 in 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-A.3, the effectiveness of the cache is diminished when information about the relatively larger DMM tree needs to be cached.

Figure 10 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 grows sub-linearly with the number of subscriptions—a ten fold increase in subscriptions from 1000 to 10000 results in a less than five fold increase in messages.

To emphasize the sub-linear increase in message cost Figure 11 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

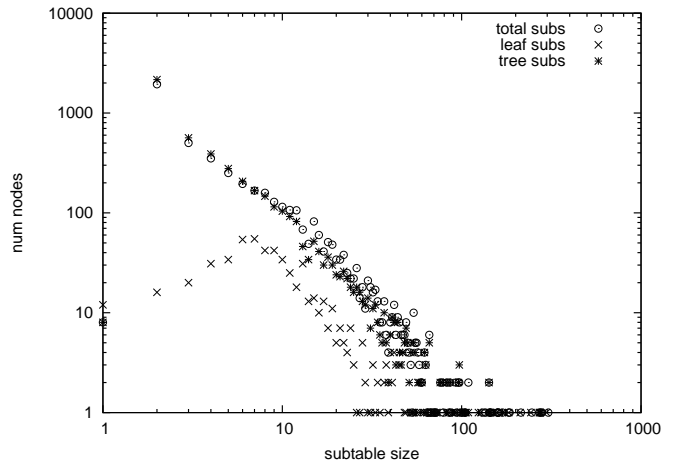


Fig. 14
SUBSCRIPTION STATE

more subscriptions, and hence will require more messages to be delivered to all subscribers. Figure 11 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.

a) *Comparison with Hermes*: Hermes [23] is a content-based pub/sub system that requires publications and subscriptions to be assigned to a topic. Publications and subscriptions are forwarded to the topic root, as in Scribe [13]. The key difference with Scribe is that publications are filtered based on their content as they propagate. 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. Due to lack of space, we point the reader to Figure 9 where the Hermes algorithm’s message cost is almost identical to that of the “DMM cache” case.

3) *Publication delivery latency*: Figure 12 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 15s 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 13 shows that the TreeCache optimization both works quickly and has a large impact. It only requires about 20s to populate the cache sufficiently to drastically reduce the delivery latency of most publications from about 15s to about 3s. We also see that the average delivery latency is very close the minimum, so there is little room for improvement to this optimization.

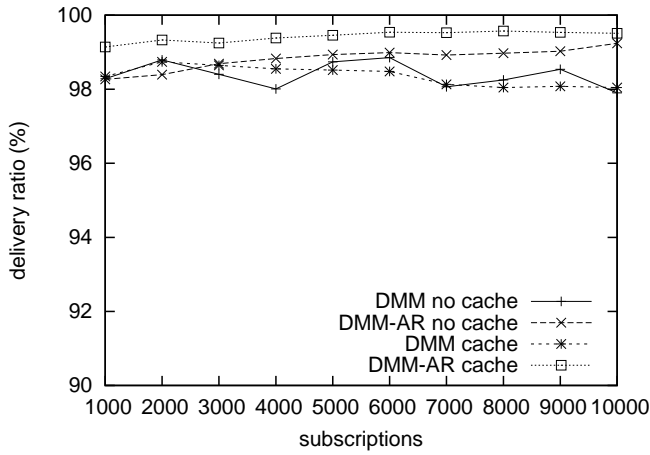


Fig. 8
DELIVERY RATIO

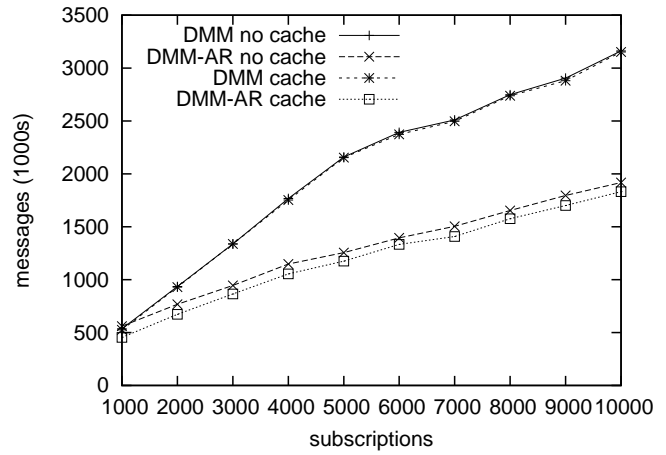


Fig. 9
MESSAGE COST

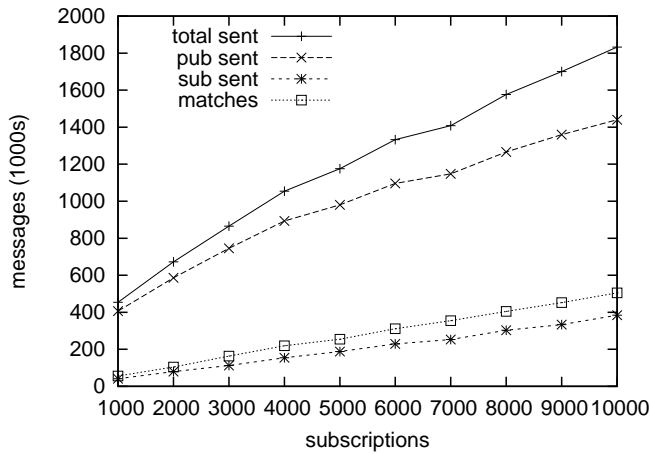


Fig. 10
MESSAGE COST (DMM-AR TREECACHE)

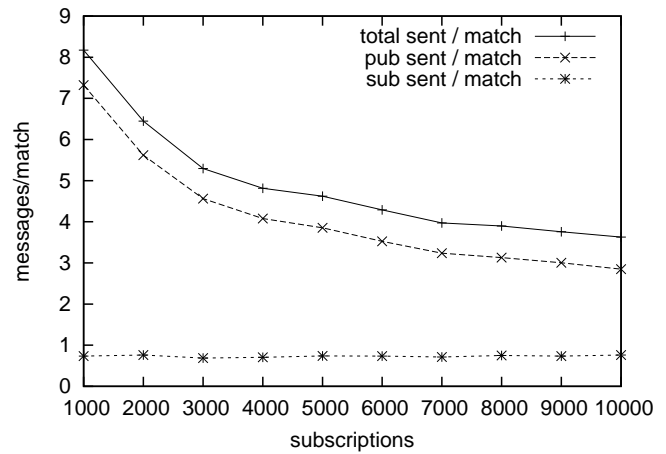


Fig. 11
NORMALIZED MESSAGE COST (DMM-AR TREECACHE)

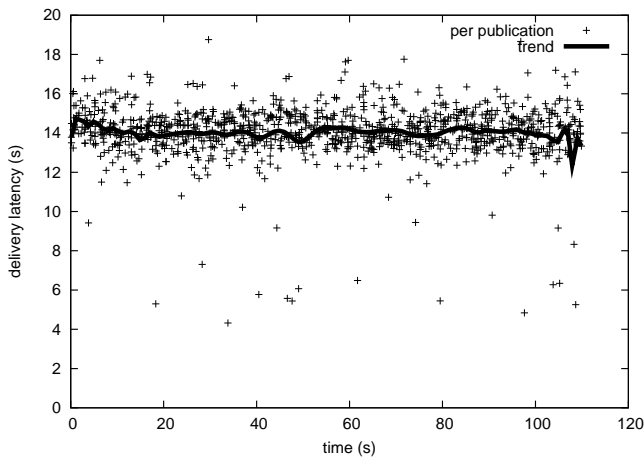


Fig. 12
DELAY WITHOUT TREECACHE

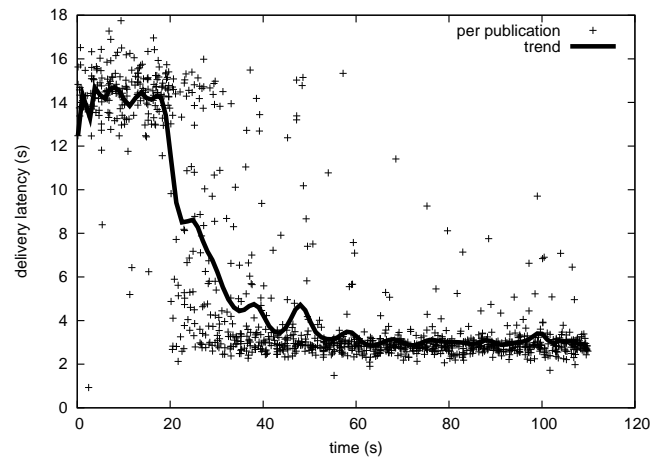


Fig. 13
DELAY WITH TREECACHE

4) *Subscription state*: Figure 14 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 subscription 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 increase 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. 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 attributes. A subscription with more attributes 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 attributes in the subscription. This is done by randomly choosing the lower and upper bounds of the first attribute in each subscription as usual, with the remaining attributes having bounds that cover the entire range. Hence, only the first attribute acts to discriminate among publications; the other attributes will always match. In addition, publications always have 10 attributes all whose values are chosen randomly. This workload results in an equal number of matches per subscription regardless of the number of attributes per subscription.

Figure 15 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 16 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.

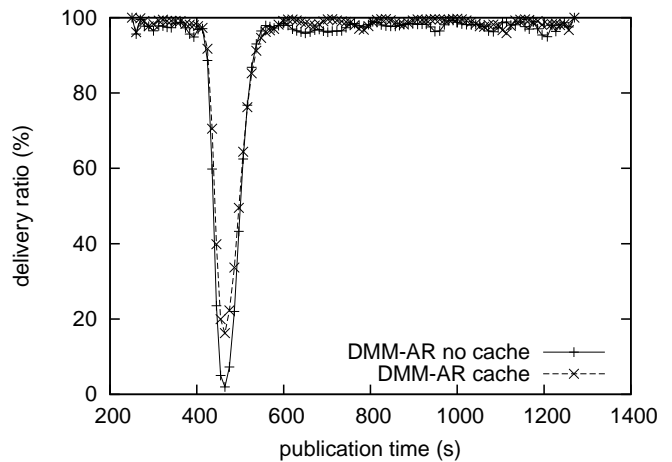


Fig. 17

DELIVERY RATIO WITH FAILURES

Together, Figures 15 and 16 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.

C. 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 400s mark, 1000 of the 5000 nodes in the network simultaneously crash, that is, they no longer send or receive any messages.

Figure 17 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 100s of the nodes crashing. (While in the figure it seems as though this recovery is closer to 200s, 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. The Pastry

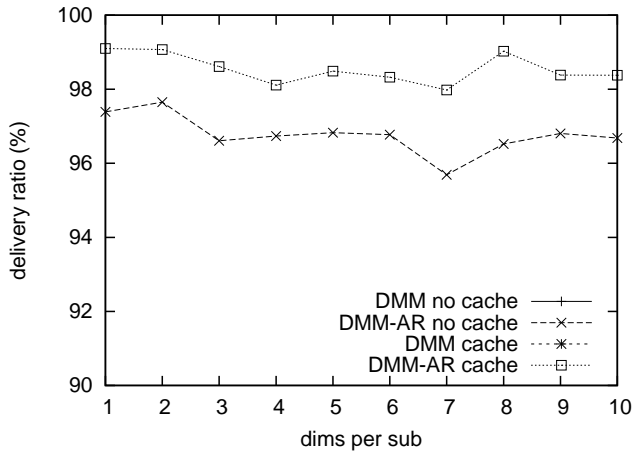


Fig. 15

DELIVERY RATIO

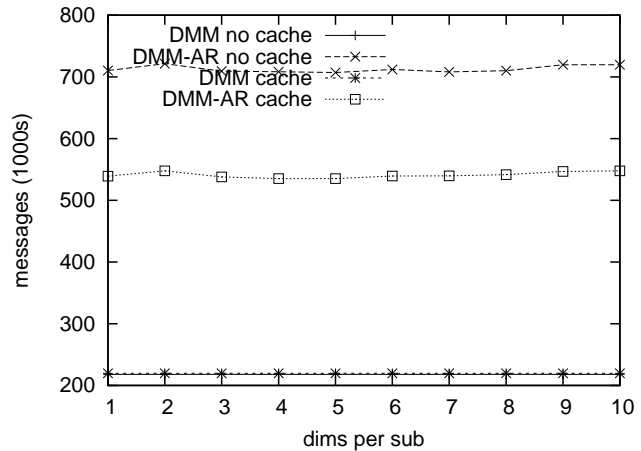


Fig. 16

MESSAGE COST

DHT is a P2P overlay that provides probabilistic performance guarantees. We develop 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. Also, extending traditional pub/sub semantics, the matching algorithm supports publications with range values.

Simulation experiments show that the algorithm scales with increasing number of subscriptions. The delivery ratio and latency are negligibly affected by increased subscriptions, the message cost scales sub-linearly, and the message cost per match actually decreases with increasing subscriptions. An optimization to cache information about the distributed search tree quickly reduces publication latency from about 15s to a nearly optimal 3s. 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 algorithm 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 100s.

For future work, we would like to support more complex data types in the publication and subscription language such as XML and XPath, add reliability guarantees to the algorithm (a stronger requirement than fault-tolerance), and take fairness considerations into account such as peer heterogeneity and incentives for peers to participate in the system. Other useful features include security concerns such as privacy, confidentiality, and resilience to malicious users.

Acknowledgments

We would like to thank Microsoft Research for providing us with the source code for the SimPastry simulator.

REFERENCES

- [1] K. Aberer, "P-grid: A self-organizing access structure for p2p information systems," in *Proceedings of the 9th International Conference on Cooperative Information Systems*. Springer-Verlag, 2001, pp. 179–194.
- [2] K. Aberer, A. Datta, M. Hauswirth, and R. Schmidt, "Indexing data-oriented overlay networks," in *VLDB*, 2005, pp. 685–696.
- [3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra, "Matching events in a content-based subscription system," in *PODC*, 1999, pp. 53–61.
- [4] A. Andrzejak and Z. Xu, "Scalable, efficient range queries for grid information services," in *Second International Conference on P2P Computing*. IEEE Computer Society, 2002, p. 33.
- [5] S. Baehni, P. T. Eugster, and R. Guerraoui, "Data-aware multicast," in *DSN*, 2004, pp. 233–242.
- [6] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based publish-subscribe over structured overlay networks," in *ICDCS*, 2005, pp. 437–446.
- [7] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajao, R. E. Strom, and D. C. Sturman, "An efficient multicast protocol for content-based publish-subscribe systems," in *ICDCS*, 1999.
- [8] A. R. Bhambe, M. Agrawal, and S. Seshan, "Mercury: supporting scalable multi-attribute range queries," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 353–366, 2004.
- [9] K. L. Calvert, M. B. Doar, and E. W. Zegura, "Modeling internet topology," *IEEE Communications Magazine*, vol. 35, no. 6, pp. 160–163, June 1997.
- [10] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Achieving scalability and expressiveness in an internet-scale event notification service," in *PODC*, Portland, Oregon, July 2000, pp. 219–227.
- [11] —, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, Aug. 2001.
- [12] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron, "Proximity neighbor selection in tree-based structured peer-to-peer overlays," Microsoft Research, Technical Report MSR-TR-2003-52, 2003.
- [13] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE JSAC*, vol. 20, no. 8, oct 2002.
- [14] A. Daskos, S. Ghandeharizadeh, and X. An, "PePeR: A distributed range addressing space for peer-to-peer systems," in *DBISP2P 2003 (co-located with VLDB 2003)*, Berlin, Germany, September 2003, pp. 200–218.
- [15] Y. Diao, P. Fischer, M. Franklin, and R. To, "Yfilter: Efficient and scalable filtering of XML documents," in *Proceedings of ICDE2002*, 2002.
- [16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

- [17] F. Fabret, H.-A. Jacobsen, F. Lirbat, J. Pereira, K. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *SIGMOD*, 2001.
- [18] A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi, "Meghdoot: Content-based publish/subscribe over P2P networks," in *Middleware*, 2004, pp. 254–273.
- [19] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, "Skipnet: A scalable overlay network with practical locality properties," in *Proc. USITS Conf.*, Seattle, WA, March 2003.
- [20] R. Huebsch, B. N. Chun, J. M. Hellerstein, B. T. Loo, P. Maniatis, T. Roscoe, S. Shenker, I. Stoica, and A. R. Yumerefendi, "The architecture of PIER: an internet-scale query processor," in *CIDR*, 2005, pp. 28–43.
- [21] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica, "Querying the internet with PIER," in *VLDB*, Sept. 2003.
- [22] 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.
- [23] P. R. Pietzuch and J. Bacon, "Peer-to-peer overlay broker networks in an event-based middleware," in *DEBS*. ACM Press, 2003, pp. 1–8.
- [24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM Press, 2001, pp. 161–172.
- [25] S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-level multicast using content-addressable networks," in *Proceedings of the Third International COST264 Workshop on Networked Group Communication*. Springer-Verlag, 2001, pp. 14–29.
- [26] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *ICDSP (Middleware)*, Nov. 2001, pp. 329–350.
- [27] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi, "A peer-to-peer framework for caching range queries," in *ICDE*. IEEE Computer Society, 2004, pp. 165–176.
- [28] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.
- [29] D. Tam, R. Azimi, and H.-A. Jacobsen, "Building content-based publish/subscribe systems with distributed hash tables," in *DBISP2P (co-located with VLDB 2003)*, Berlin, Germany, September 2003.
- [30] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "A peer-to-peer approach to content-based publish/subscribe," in *DEBS*. ACM Press, 2003, pp. 1–8.
- [31] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," *IEEE JSAC*, vol. 22, no. 1, pp. 41–53, Jan. 2004.
- [32] Y. Zhao and R. Strom, "Exploiting event stream interpretation in publish-subscribe systems," in *PODC*. ACM Press, 2001, pp. 219–228.
- [33] 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 international workshop on Network and operating systems support for digital audio and video*. ACM Press, 2001, pp. 11–20.