String Attribute Query Processing over DHTs: The Publish-Subscribe Case [1]

Ioannis Aekaterinidis and Peter Triantafillou
Research Academic Computer Technology Institute and
Department of Computer Engineering and Informatics, University of Patras, Greece
{aikater,peter}@ceid.upatras.gr

Abstract

In this paper, we present and study solutions for the efficient processing of queries over string attributes in a large P2P data network implemented with DHTs. The proposed solutions support queries with equality, prefix, suffix, and containment predicates over string attributes. Currently, no known solution to this problem exists.

We propose and study algorithms for processing such queries and their optimizations. As event-based, Publish/Subscribe information systems are a champion application class where string attribute (continuous) queries are very common, we pay particular attention to this type of data networks, formulating our solution in terms of this environment. A major design decision behind the proposed solution is our intention to provide a solution that is general (DHT-independent), capable of being implemented on top of any particular DHT.

1. Introduction

The peer-to-peer (P2P) paradigm is appropriate for building large-scale distributed systems/applications. P2P systems are completely decentralized, scalable, and self-organizing. All nodes participating in those systems have equal opportunities and are providing services where information is exchanged directly with each other. A popular class of them is the "structured" P2P systems where the data placement and the topology within the network are tightly controlled.

A large body of research is currently targeting the extension and employment of DHTs for efficient data query processing. The nature and functionality of the DHT-based P2P can guarantee the efficient managing of queries with equality predicates. However, it is difficult to perform queries with range predicates over numeric attributes and/or prefix, suffix, and containment predicates over string attributes. As far as we know, there is currently no applicable solution for processing string attribute queries over DHTs. This kind of functionality can be used by data management systems built using the Publish/Subscribe (Pub/Sub) paradigm.

With our work in this paper we propose a solution that supports rich queries with string attributes (and string operations like prefix, suffix and containment) over a DHT-based P2P infrastructure. Given the popularity of the pub/sub technology, we primarily focus on it and formulate our solution in terms of a pub/sub infrastructure.

[1]This work is partly supported by the 6th Framework Program of EU through the integrated project DELIS (#001907) on Dynamically Evolving, Large Scale Information Systems.

2. Background and contribution

2.1. Distributed Hash Tables (DHTs)

Distributed Hash Tables (DHTs [8], [9], [10], [12], [20]) are becoming increasingly popular for structuring overlay topologies of large-scale data networks. In a DHT each node has a unique identifier (nodeID) selected from a very large address space. Each data item can be associated with a key which is a unique identifier of the same type as nodeID. DHTs can efficiently locate and route a <data item, key> pair based on the key identifier.

As an example of a popular DHT we outline how Chord operates. Chord [10] is a fairly simple, structured peer-to-peer network based on a DHT. Compared to unstructured peer-to-peer networks like Gnutella and MojoNation where neighbors of peers are defined in rather ad hoc ways, Chord is "structured" because of the way peers define their neighbors, forming a ring topology. Chord provides an exact mapping between node identifiers (nodeID) and keys associated with data items using consistent hashing [13]. NodeIDs and keys are mapped to a large circular identifier space, e.g. $[0, 2^{160})$ for 160-bit IDs. Values in this space can be viewed as positions in the ring defining the name/identifier space. Thus, given a key, Chord maps it to the (ring position) node whose nodeID is equal to the key. If this node does not exist, the key is mapped to the first *successor* of this node on the ring.

Similarly to all DHT-based networks, Chord has a bounded performance in terms of hop count. It efficiently determines the successor of an identifier (key) in ½log(N)

hops on average (and in $O(\log(N)$ hops in the worst case), in the steady state (where N is the maximum number of nodes in the network). Each node maintains routing information for up to $O(\log(N))$ other nodes. Adding or removing a node from the network can be achieved at a cost of $O(\log^2(N))$ messages. Chord has become very popular and has been used as a building block for several large-scale distributed systems, as it's simple and its performance is guaranteed.

2.2. The publish/subscribe paradigm

There are two popular types of publish/subscribe systems: *topic-based* and *content-based*. Topic based systems are much like newsgroups. Content-based systems are preferable as they give users the ability to express their interest by issuing continuous queries, termed subscriptions, specifying predicates over the values of a number of well defined attributes. The matching of publications (events) to subscriptions (interests) is done based on the content (values of attributes).

Building a centralized publish/subscribe may result in scalability problems as the number of publications and subscriptions increases. Thus, a decentralized approach is deemed necessary. The main challenge in a distributed environment is the development of an efficient distributed matching algorithm. Distributed solutions have been provided for topic-based publish/subscribe systems [1], [2], [3]. More recently, some attempts on distributed content-based publish/subscribe systems use routing trees to disseminate the events to interested users based on multicast techniques [4], [5], [15], [16]. Some other attempts use the notion of rendezvous nodes which ensure that events and subscriptions meet in the system [14].

Some approaches have also considered the coupling of topic-based and content-based systems. In [6] the publications and the subscriptions are automatically classified in topics, using an appropriate application-specific schema. The main drawbacks of this attempt are the design of the domain schema and the false positives that may occur. Our previous work reported in [17] proposes a solution for numeric attributes with equality and range predicates in a DHT-dependent (Chord) way.

Recently there have been algorithms for filtering and matching queries on continuous queries databases, based on the AWP data model [18]. In [19] keyword searching is supported by applying a multi-level partitioning scheme on top of the SkipNet P2P infrastructure [21]. However both works do not support string attributes with prefix, suffix, and containment constraints. As far as we know, this is the first work that leverages DHT research to build large scale content-based pub/sub systems while supporting subscriptions with a rich set of constraints on string attributes.

2.3. Contribution: String processing over DHTs The publish/subscribe case

Until now, it still remains a challenge to leverage DHTs as a dominating technology for constructing efficient scalable overlay networks in creating a content-based pub/sub infrastructure. In particular, an efficient solution to string attribute support over DHTs is very much lacking.

Our solution on string attribute processing over DHTs is independent of the type of DHT. It is easily applicable to every DHT that can efficiently locate an object based on its key identifier. However, for simplicity of presentation of algorithms and to study their performance impact we will on occasion use Chord because of its simplicity, guaranteed performance, and popularity within the peer-to-peer community. By leveraging DHTs we allow ourselves the luxury of not having to be concerned with the development of an infrastructure for topology establishment that provides dynamic topology maintenance, fault tolerance, scalability, and efficient routing.

In the rest of the paper we will show how to build content-based pub/sub systems, able to support string-valued objects and their related operations.

3. Publish/Subscribe with string attributes over DHTs

Events and subscriptions are defined over an event schema that supports a number A of attributes. An event is defined to be a set of k values ($k \leq A$), one for each attribute, while a subscription is defined through an appropriate set of constraints over a subset of the A attributes of the schema.

3.1. The Event/Subscription Schema

The event schema is a set of typed attributes. Each attribute a_i consists of a name, type, and a value $v(a_i)$. Generally, the type of an attribute belongs to a predefined set of primitive data types commonly found in most programming languages. In this work we will focus on string attributes. Thus, from now on whenever we refer to an attribute of our schema, we assume that it is a string attribute. An event can be thought of as set of <attribute name, value> pairs. An example event with two attributes defined is:

Event1 = { Exchange : "NYSE"*, Symbol :* "OTE" *}*

The subscription schema is more general, allowing a rich set of subscriptions with all common string operators. An example of two subscriptions is:

*Sub1={SubID$_1$|Exchange: "*SE" , Symbol : "OT*"}*
*Sub2=SubID$_2$|Exchange: "N*E" , Symbol : "OTA*"}*

The allowed operators are: suffix (e.g. in Sub1, the *Symbol* attribute), prefix operator (e.g. in Sub1, the *Exchange* attribute), and the containment operator (e.g. in Sub2 the *Exchange* attribute).

An event matches a subscription *if and only if all* the subscription's attribute predicates/constraints are satisfied. A subscription can have two or more constraints for the same attribute which can be thought as if we had two or more different subscriptions with unique constraints over their attributes. Finally, an event can have more attributes than those mentioned in the subscription schema. Note that if an event has more attributes than a subscription and all subscription predicates are satisfied then the subscription is considered to be matched.

3.2. The Subscription Identifier

A subscription identifier (SubID) is the concatenation of three parts:

1. c_1: The id of the node receiving the subscription (i.e., where the subscription "belongs").
2. c_2: The key of the subscription needed to identify it among the subscriptions belonging to the same node.
3. c_3: The number of attributes on which constraints are declared by this subscription.

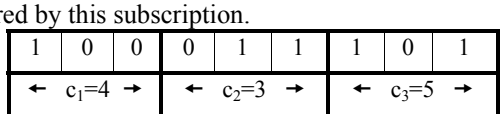

Figure 1. An example subscription id (SubID).

The subscription id depicted in Figure 1 identifies subscription 3 (c_2=3), belonging to a DHT node 4 (c_1=4), comprised of constraints on 5 attributes (c_3=5).

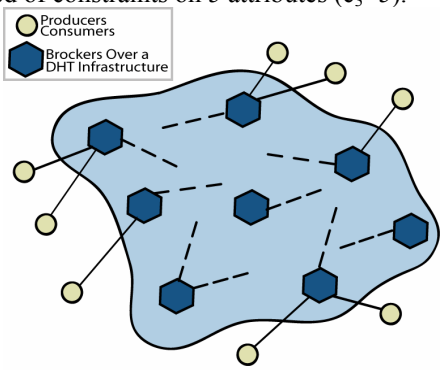

Figure 2. Network of nodes forming a DHT network.

3.3. System Architecture

Figure 2 depicts the intended pub/sub architecture. Client nodes are producers/consumers, issuing events/subscriptions, respectively. Each client is 'attached' to a broker node using any appropriate such mechanism (e.g., hashing the client's port and IP address with the DHTs hashing scheme). A broker node is a DHT node and is added/deleted from the DHT following the specific DHT related algorithms.

Moreover, we should note that for every subscription there is a node in the DHT network storing metadata information for it. That node is identified by the c_1 field of the subscription id and it keeps metadata information about the subscription (for example the IP address of the user that generated the subscription etc.).

4. Processing subscriptions

Consider an example pub/sub system supporting A string attributes (a_i, $1 \le i \le A$). Subscriptions specify a single value (equality) or string operators on values (prefix, suffix, containment) for each attribute a_i. The main idea behind our approach is to store the subscription ids (SubIDs) at those nodes of the DHT network that were selected by appropriately hashing the values of the attributes in the subscriptions. The matching of an incoming event can then be performed simply by asking those nodes for stored subscription ids.

4.1. Storing subscriptions

Storing subscriptions is done using the DHT hash function (*DHT.h*()). For example in Chord, this hash function *h*() (i.e., SHA-1) returns an identifier quasi-uniformly distributed in the address space used for the node identifiers. Thus, the result (say n) of this hash function for the value v(a_i) of the attribute a_i is quasi-uniformly distributed in the nodes' identifier address space (where $n = h(v(a_i))$).

The main idea behind the storing procedure, is to place the subscription id at the node whose id is the least id equal or greater to $n=h(v(a_i))$ (that is *successor*(n) from the Chord API). Therefore, in Chord, the SubID will be placed at the node: *successor*(h(v(a_i))).

We maintain three lists (initially empty) in every node for every string attribute a_i of our schema. The first two are the $L_{ai\text{-}pref}$ and $L_{ai\text{-}suff}$ where we store the SubIDs of the subscriptions that contain prefix or suffix operations, respectively, over the attribute a_i whose value hashed to this node.

The containment operation can be easily transformed into prefix and suffix operations and thus there is no need for a third list. More precisely, we break the containment string into prefix and suffix parts and we deal with the containment operation as a prefix and a suffix operation over the same attribute. Finally, there is a need for a list where the SubID is going to be stored in the case of an equality operation on a string attribute a_i. This is the L_{ai}

list. The procedure of storing subscriptions can be seen in Figure 3.

Notation

SubID : subscription identifier
nodeID : node identifier
a_i : attribute *i*
L_{ai} : List of *SubIDs* for attribute a_i with equality operation
$L_{ai\text{-}pref}$: List of *SubIDs* for attribute a_i with prefix operation
$L_{ai\text{-}suff}$: List of *SubIDs* for attribute a_i with suffix operation
$v(a_i)$: value of attribute a_i
DHT.h() : DHT's function returning a key (SHA-1 in Chord)
$v(a_i)$***.prefix()*** : prefix operation of a containment on $v(a_i)$
$v(a_i)$***.suffix()*** : suffix operation of a containment on $v(a_i)$

Storing Subscriptions

1. For every string attribute a_i in subscription
2. If a_i has an equality constraint
3. store *SubID* in node with *nodeID =DHT.h*(v(a_i)) in the L_{ai} list.

4. If a_i has a prefix constraint
5. store *SubID* in node with *nodeID=DHT.h*(v(a_i)) in the $L_{ai\text{-}pref}$ list.

6. If a_i has a suffix constraint
7. store *SubID* in node with *nodeID=DHT.h*(v(a_i)) in the $L_{ai\text{-}suff}$ list.

8. If a_i has a containment constraint
9. break a_i into v(a_i).prefix() and $v(a_i)$.suffix()
10. store *SubID* in node with *nodeID=DHT.h*(v(a_i).prefix()) in the $L_{ai\text{-}pref}$ list.
11. store *SubID* in node with *nodeID=DHT.h*($v(a_i)$.suffix()) in the $L_{ai\text{-}suff}$ list.

Figure 3. The procedure of storing subscriptions.

Example 1: Storing subscriptions

Suppose that a user expresses her interests with the following subscriptions over a Chord DHT network:
*Sub1={SubID$_1$|Exchange: "*SE" , Symbol : "OT*"}*
*Sub2={SubID$_2$|Exchange: "N*E" , Symbol : "OTA*"}*

As we can see the *Symbol* attribute (a_2) is defined as a suffix with value OT* (that is v(a_2)= "OT*"). The storing procedure for this subscription involves the storage of $SubID_1$ in the list $L_{ai\text{-}suff}$ of the node with nodeID=*successor*(*h*(v(a_2)))=*successor*(*h*("OT")).

Similarly, for the *Exchange* attribute (a_1) with prefix value *SE (that is v(a_1)= "*SE") we store the $SubID_1$ at the $L_{ai\text{-}pref}$ list of the *successor*(*h*(v(a_1)))= *successor*(*h*("SE")) node. In Sub2 the *Exchange* attribute is declared as a containment operation with value N*E. The procedure for storing this type of subscription requires the transformation of containment into prefix and suffix operations. Thus the prefix operation of containment v(a_1) is v(a_1).prefix()="*E" while the suffix operation is v(a_1).suffix()="N*". Then we store the $SubID_2$ in the $L_{ai\text{-}suff}$ list of node with nodeID=*successor*(h("N")) and in the $L_{ai\text{-}pref}$ list of node with nodeID=*successor*(h("E")). By processing accordingly the a_2 attribute of the second subscription we are done with the storing procedure and we are ready to proceed to the matching process on every incoming event. ❑

4.2. Updating subscriptions

Updating a subscription involves a procedure during which the values of all attributes contained in the subscription are updated using the standard API of the DHT. In the case of equality, prefix, and suffix operation only two nodes are affected. On the one hand, the node that is mapped to the old, stale value is forced to delete the SubID for the attribute that belongs to the subscription with identifier SubID. On the other hand, a new node is going to store the SubID, depending on the id returned from the DHT's hash function passing the new updated value. In the case of the containment operation there is the need to contact twice as many nodes compared to the other operations, since as we said before, we break the containment operation into suffix and prefix operations.

Deleting subscriptions is done as explained above since the updating procedure includes a deleting step.

5. Event processing and matching

We define the functions *prefix*(string x, integer j) and *suffix*(string x, integer j) that return the j-characters-long prefix and suffix of the string x, respectively. For example, *prefix*("OTE", 2) = "OT" and *suffix*("OTE", 1) = "E".

Suppose now, that an event arrives at the system with $N_{a\text{-}event}$ attributes defined. The *SubID Lists Collection Phase* (Figure 4) starts by processing each attribute separately. It first tries to find the node which stores SubIDs for the value v(a_i) of the attribute a_i (e.g., in Chord *n=successor*(*h*(v(a_i))). Those subscriptions have defined an equality operation over the attribute a_i. The algorithm, then, retrieves the list of unique SubIDs found to be stored in node n in the list L_{ai} designated for the attribute a_i and stores it in the $L_{ai\text{-}EQUALITY\text{-}DELIVERY}$ list in order to inform the interested users for possible matching[2].

Apart from the equality operation, the algorithm must also check for any possible matches with subscriptions that have defined prefix, suffix and containment predicates. In order to find the subscriptions that may have declared a

[2] It should be noted that in the $L_{ai\text{-}*}$ lists we store the (SubID, v(a_i)) pairs so as to collect only those SubIDs that are stored there because of the value v(a_i). Consider for example the case where the SubIDs of two different subscriptions $SubID_1$ and $SubID_2$ that have defined different values for the attribute a_i ($v_1(a_i) \neq v_2(a_i)$), are both stored in the same node (that is *successor*(*h*($v_1(a_i)$)) = *successor*(*h*($v_2(a_i)$))). In the case where an event arrives with a value equal to $v_1(a_i)$ for the attribute a_i , we should collect only $SubID_1$ and not $SubID_2$. For the sake of simplicity and easy reading of the described algorithms we omit this detail.

Event Processing and Matching [1/2]

Notation

SubID : subscription identifier
nodeID : node identifier
a_i : attribute *i*
$N_{a\text{-}sub\text{-}i}$: number of attributes defined in a subscription i
$N_{list\text{-}sub\text{-}i}$: number of collected lists that $SubID_i$ is stored
L_{ai} : List of *SubIDs* for attribute a_i with equality operation
$L_{ai\text{-}pref}$: List of *SubIDs* for attribute a_i with prefix operation
$L_{ai\text{-}suff}$: List of *SubIDs* for attribute a_i with suffix operation
$v(a_i)$: value of attribute a_i
DHT.h():DHT's function returning a key (in Chord SHA-1)
prefix(x,j) : j-characters-long prefix of string x
suffix(x,j) : j-characters-long suffix of string x
$L_{ai\text{-}PREFIX}$: the union of all $L_{ai\text{-}pref}$ lists
$L_{ai\text{-}SUFFIX}$: the union of all $L_{ai\text{-}suff}$ lists
Delivery Lists:
$L_{ai\text{-}EQUALITY\text{-}DELIVERY}$: list with candidate SubIDs for equality matching
$L_{ai\text{-}PREFIX\text{-}DELIVERY}$: list with candidate SubIDs for prefix matching
$L_{ai\text{-}SUFFIX\text{-}DELIVERY}$: list with candidate SubIDs for suffix matching
$L_{ai\text{-}CONTAINMENT\text{-}DELIVERY}$: list with candidate SubIDs for containment matching
Matching List:
$L_{matching}$: list with SubIDs that generated a match to the incoming event

SubID lists collection phase

1. for every $event_j$ arriving at the system
2. for every attribute a_i in the $event_j$
3. go to node *DHT.h*($v(a_i)$)
4. retrieve the list L_{ai} of SubIDs found there and store it to $L_{ai\text{-}EQUALITY\text{-}DELIVERY}$
5. for j=1 to character length of $v(a_i)$
6. go to node *DHT.h*(*suffix*($v(a_i)$,j))
7. retrieve the list $L_{ai\text{-}pref}$ of SubIDs found there and store it to $L_{ai\text{-}PREFIX}$
8. go to node DHT.*h*(*prefix*($v(a_i)$,j))
9. retrieve the list $L_{ai\text{-}suff}$ of SubIDs found there and store it to $L_{ai\text{-}SUFFIX}$
10. for every attribute a_i in $event_j$
11. for every $SubID_j$ in the $L_{ai\text{-}PREFIX}$ and $L_{ai\text{-}SUFFIX}$ lists
12. if $SubID_j$ is found ONLY in prefix lists
13. move $SubID_j$ to $L_{ai\text{-}PREFIX\text{-}DELIVERY}$ list
14. if $SubID_j$ is found ONLY in suffix lists
15. move $SubID_j$ to $L_{ai\text{-}SUFFIX\text{-}DELIVERY}$ list
16. if $SubID_j$ found in BOTH prefix and suffix lists
17. move $SubID_j$ to $L_{ai\text{-}CONTAINMENT\text{-}DELIVERY}$

Figure 4. Collecting the SubIDs of subscriptions that are candidate for matching the event.

prefix operation on the attribute a_i we should ask the nodes in the DHT in a similar to equality constraint way. For example, in the Chord ring we ask those nodes that have nodeID=*successor*(*suffix*($v(a_i)$,j)) for j ranging from 1 to *l* (where *l* is the length of string $v(a_i)$). From those nodes we retrieve the $L_{ai\text{-}pref}$ lists. Thus we collect *l* lists with SubIDs of subscriptions that may have declared a prefix operation.

A similar procedure is followed for the suffix operation and we finally collect the *l* $L_{ai\text{-}suff}$ lists from the DHT. After finishing with this step we have collected *l* lists in order to check prefix matching, *l* lists for suffix matching and one list for equality matching, for every string attribute. We then merge all the $L_{ai\text{-}suff}$ and $L_{ai\text{-}pref}$ into the $L_{ai\text{-}SUFFIX}$ and $L_{ai\text{-}PREFIX}$ lists respectively.

Event Processing and Matching [2/2]

Matching phase

1. for every $SubID_k$ found in the *Delivery Lists*
2. retrieve the number of attributes defined from the c_3 field: $N_{a\text{-}sub\text{-}k}$
3. count the number of Delivery lists where $SubID_k$ is stored: $N_{list\text{-}sub\text{-}k}$
4. if $N_{a\text{-}sub\text{-}k}$ equals $N_{list\text{-}sub\text{-}k}$ we have a match
5. remove $SubID_k$ from all lists
6. store $SubID_k$ in the Matching list $L_{matching}$

Delivery phase

7. for every $SubID_i$ in the $L_{matching}$ list
8. contact the node that keeps the subscription. (Its nodeID is the c_1 field of the $SubID_i$) in order to inform (deliver the event to) the interested client

Figure 5. The matching algorithm after collecting the SubIDs for potential evnent matching.

The next step (*Matching Phase*, Figure 5) is actually the event-subscriptions matching process. In order to discover a prefix, suffix or containment match on attribute a_i we check the $L_{ai\text{-}PREFIX}$ list along with the $L_{ai\text{-}SUFFIX}$ list. The SubIDs that are found to be in the prefix list but *not* in the suffix list are considered to produce a prefix matching and thus they are transferred in the $L_{ai\text{-}PREFIX\text{-}DELIVERY}$ list. On the other hand, the SubIDs that are found to be in the suffix list but *not* in the prefix list are considered to produce a suffix matching and are stored in the $L_{ai\text{-}SUFFIX\text{-}DELIVERY}$ list. Finally, those SubIDs that are found to be in both suffix and prefix lists are considered to produce a containment match and are stored in the $L_{ai\text{-}CONTAINMENT\text{-}DELIVERY}$ list.

Until now, we have collected four *delivery* lists with subscription identifiers for the a_i attribute. These are: $L_{ai\text{-}EQUALITY\text{-}DELIVERY}$, $L_{ai\text{-}PREFIX\text{-}DELIVERY}$, $L_{ai\text{-}SUFFIX\text{-}DELIVERY}$, and $L_{ai\text{-}CONTAINMENT\text{-}DELIVERY}$.

It should be noted that among the above four lists for the attribute a_i there are no duplicate SubIDs. Suppose, now, that a subscription $SubID_k$ is found to be in at least one of the $N_{a\text{-}event} \times 4$ lists that were collected when the event arrived at the system. Assume that this subscription consists of $N_{k\text{-}sub}$ attributes ($N_{k\text{-}sub}$ is obtained from the field c_3 of the SubID defined in section 3.2). Then, the subscription with identifier $SubID_k$ is considered to match the event if it appears in exactly $N_{k\text{-}sub}$ lists collected from

the network. Those SubIDs are then transferred to the *Matching list* $L_{matching}$ where they are processed further in order to inform the subscribers that are interested for the incoming event (*Delivery Phase*, Figure 5). More precisely, for every SubID in the $L_{matching}$ list, we contact the node that actually holds the subscription. This nodeID can be easily retrieved from the c_1 field of the SubID.

Then the metadata information about the specific subscription stored in that node can be used in order to deliver the event to the interested subscriber.

Example 2: Matching events with subscriptions

Suppose that we have the following subscriptions
*Sub1={SubID$_1$|Exchange: "*SE" , Symbol : "OT*"}*
*Sub2={SubID$_2$|Exchange: "N*E" , Symbol : "OTA*"}*
generated by two clients connected to brokers in a Chord DHT network and an event arrives with the following values:

Event1 = { Exchange : "NYSE", Symbol : "OTE" }

We start with the attribute *Exchange* and we try to locate the subscriptions that declared an equality constraint. Thus, we retrieve the list $L_{Exchange}$ from node successor(h("NYSE")). This list is empty since none of the two subscriptions declared an equality constraint. Hence, the list $L_{Exchange\text{-}EQUALITY}$ is empty.

We continue by collecting the prefix and suffix lists ($L_{Exchange\text{-}PREFIX}$ and $L_{Exchange\text{-}SUFFIX}$). We ask the nodes successor(h("N")), successor(h("NY")), successor(h("NYS")) and successor(h("NYSE")) for their stored suffix lists and we get only $SubID_2$ from the successor(h("N")) node as a result of the containment constraint of Sub2. We then ask the corresponding nodes for their stored prefix lists and we get $SubID_1$ from node successor(h("SE")) as a result of the prefix constraint of Sub1 and $SubID_2$ from node successor(h("E")) as a result of the containment constraint of Sub2. Thus after examining the attribute *Exchange* we get the lists:

$L_{Exchange\text{-}EQUALITY}$ ..... → {empty}
$L_{Exchange\text{-}PREFIX}$ ..... → **$SubID_2$, $SubID_1$**
$L_{Exchange\text{-}SUFFIX}$ ..... → **$SubID_2$**

In order to fill the *Delivery* lists (that contain SubIDs for possible matching) we check the lists already collected. Recall that those SubIDs that are found to be in both lists are candidates for containment matching. $SubID_2$ belongs to this category. Those SubIDs that are found only in the suffix list are candidate for suffix matching and as we can see there is no SubID belonging to this category. Finally, those SubIDs that are found only in the prefix list ($SubID_1$) are candidate for prefix matching. After this phase, we have collected the following *Delivery* lists for the *Exchange* attribute:

$L_{Exchange\text{-}EQUALITY\text{-}DELIVERY}$ ..... → {empty}
$L_{Exchange\text{-}PREFIX\text{-}DELIVERY}$ ..... → **$SubID_1$**
$L_{Exchange\text{-}SUFFIX\text{-}DELIVERY}$ ..... → {empty}
$L_{Exchange\text{-}CONTAINMENT\text{-}DELIVERY}$ ... → **$SubID_2$**

Repeating the above process for the *Symbol* attribute, we get the following *Delivery* lists:

$L_{Symbol\text{-}EQUALITY\text{-}DELIVERY}$ ..... → {empty}
$L_{Symbol\text{-}PREFIX\text{-}DELIVERY}$ ..... → {empty}
$L_{Symbol\text{-}SUFFIX\text{-}DELIVERY}$ ..... → **$SubID_1$**
$L_{Symbol\text{-}CONTAINMENT\text{-}DELIVERY}$ ..... → {empty}

Now we continue to the matching phase determining which one of the collected subscriptions are indeed matching the incoming event. From the c_3 part of the SubIDs of subscriptions 1 and 2 we can find out that both subscriptions have constraints over two attributes. Since $SubID_1$ is found in two lists, a match is implied and so we keep the $SubID_1$ in order to inform the node which generated the subscription about the matched event. On the other hand, subscription 2 is found to be in one list and thus we do not have a match. The next step is to inform the interested user. This is done by consulting the node storing the subscription (with nodeID equal to the c_1 field of the $SubID_1$) and holding metadata information for $SubID_1$, in order to locate the IP address of the client that generated the subscription. Then, the matched event is delivered to the interested client. ❑

5.1. Expected performance analysis

All DHTs have bounded performance of $O(log(N))$ [3] hops in order to contact a node. In this section we present a performance analysis of our algorithms

During the subscription storage procedure, the average number of hops needed to store a SubID is equal for all allowable operations on strings except the containment operation that requires more hops. This is so, since the SubID is stored in a single node in the case of an equality constraint. Thus, for a subscription, i, $O(log(N))$ hops are required in the worst case in order to store the $SubID_i$ for every attribute. Note that for the case of the containment operation the number of hops is twice as many, but obviously $O(log(N))$. Thus, the scalability of the processing/managing algorithm is guaranteed.

The matching process and more precisely the *SubID Lists Collection Phase*, requires contacting more nodes. More precisely, for every attribute in the event, we should contact one node in order to retrieve the lists of SubIDs that have declared equality for the given attribute. Now, suppose that the character length of the attribute a_i is l_i. Then in order to collect the prefix and suffix lists we should contact $2\times l_i$ nodes which results in $2\times l_i \times O(log(N))$

[3] N is the number of nodes participating in the DHT. N can be approximately computed in the case of the Chord DHT, by observing the distance between successors in the finger table.

hops for every attribute. In general, for an event with $N_{a\text{-event}}$ attributes $\sum_{i=1}^{N_{a-event}}[(1+2\cdot l_i)\cdot O(\log(N))]$ hops are required in the worst case which in fact results in bounded performance of $O(l \times log(N))$ where c is a constant.

During the event delivery phase suppose that we have to contact k nodes, (k=| $L_{matching}$ |). The event can be delivered to the brokers storing the matched SubIDs by choosing between two different delivery policies. The first one involves the contacting of k nodes separately which results in $k \times O(log(N))$ hops. With the second one we inform all nodes in the network, which results in $O(N)$ hops (this can be achieved by following a full circle path in the case of the Chord ring). Our choice on what police to use is based on the relation between the number of hops on average needed to contact separately k nodes, ($k \times ½ \times \log(N)$ for Chord) and the number of hops needed to contact all nodes in the network (N). If $k \times ½ \times log(N) < N$ we use the first policy which results in an overall hop count (*Collection and Delivery phase*) of $O((l+k)log(N))$. Following the second policy requires $O(N)$ hops which is the worst case.

Now, compared to existing systems like Siena [6] where the subscription propagation as well as the event matching phase requires $O(N)$ hops, we can definitely say that our approach is overall preferable. More precisely, in our approach the subscription storing phase requires $O(l \times log(N))$ hops while in the Siena system $O(N)$ hops are required. During the event delivery phase, compared to Siena's $O(N)$ hop count, our approach requires $O(klog(N))$ hops when k is relative small and $O(N)$ in the worst case.

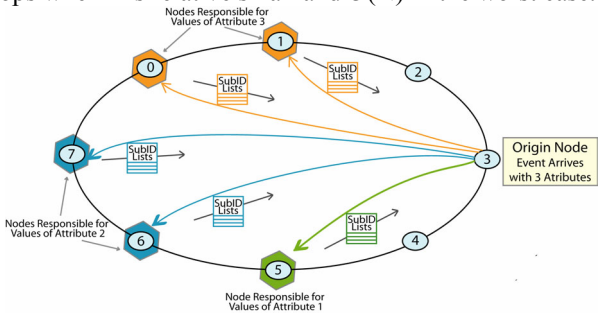

Figure 6. Coordinated matching.

6. Event processing optimizations

The optimizations that follow aim to reduce mainly the processing cost and study the related trade-offs with respect to the overall network traffic as a result of collecting, sending back to the origin node (the broker node where the incoming event arrived) the SubID lists, and finally processing them in order to compute and deliver the matched events to interested users. Our motivation is to distribute when possible/profitable the matching phase to a number of involved DHT nodes.

Coordinated Matching

The algorithm presented in section 5 starts by processing each attribute of the event separately, contacting a subset of nodes and retrieving the SubID lists as we can see in Figure 6. It is clear that the matching process is performed at the origin node where all lists have been collected.

Distributed Matching

A first idea trying to ameliorate the above process, is to perform the matching process in a distributed, step-by-step way, as can be seen in Figure 7. The key idea is to order the events' attribute-values based on their expected selectivity. This selectivity (i.e., the size of the SubID lists with subIDs matching the event's attribute value) depends on the popularity of the attribute (i.e., how many subscriptions are involving this attribute) as well as on the attribute values' popularity. This kind of ordering will lead in processing first the attributes that are likely to return a small result set and pass those relatively small lists to subsequent nodes in order to perform the matching. The problem of identifying the selectivity of an event's attribute value is a formidable one in general (since both popularity distributions mentioned above need to be estimated). However, fortunately, there exist applications where such information on event-attribute selectivities is readily deducible. For example, in a stock market application it is likely that attributes such as "stock exchange name" are associated with large results sets and thus should be processed as lately as possible.

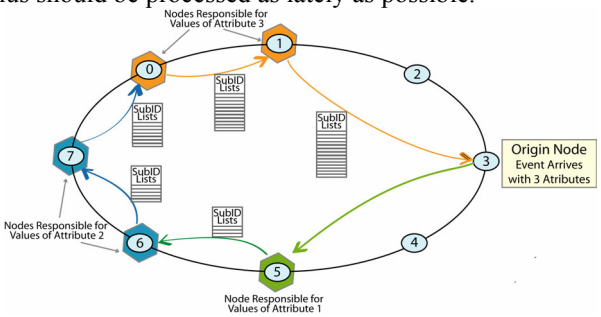

Figure 7. Distributed matching.

Another indication of how selective an event attribute is, could be the size of its value domain. Thus, we can order the k attributes of the event from largest domains (attribute a_1) to smallest ones (attribute a_k). Small domains indicate that the SubID lists stored at each node are going to grow in size as new subscriptions arrive at the system with values picked from this small set of values. Thus, attributes with small domains is preferred to be processed later during the SubID collecting phase.

Suppose now that an event arrives. We start by processing the attribute, a_1, with the estimated smallest selectivity and send a request for collecting SubIDs to all appropriate nodes, as illustrated before. Among those nodes we pick one that is responsible for collecting and merging all such lists, in a *GlobalSubIDList(1)* list. Then,

the current node sends this list to the next set of nodes that will process the event for the attribute that has the next smaller selectivity, a_2.

At this step, the second node has to process two lists of SubIDs. The list that was collected from the nodes that it is responsible for, $LocalList_2$, and the previously retrieved list, *GlobalSubIDList(1)*. From the $LocalList_2$ we drop those SubIDs that have declared[4] one or more attributes that we have already checked (in this case attribute a_1), and are not present in the *GlobalSubIDList(1)*. From the *GlobalSubIDList(1)* we drop those SubIDs that have declared the current attribute a_2 and are not present in the $LocalList_2$. The remaining SubIDs are merged to the *GlobalSubIDList(2)* list and are propagated to the node responsible for the third attribute, a_3. This process continues until we reach the last node (responsible for the last attribute of the event a_k) where the already matched list of SubIDs *GlobalSubIDList(k)* is sent to the origin node.

The weakness of *Distributing Matching* is that it is possible that many SubIDs that may already match the event, will be sent several times through the DHT network until it finally reaches the origin node. This is the case where a subscription does not declare any of the attributes that are going to be checked in later steps of the distributed matching process.

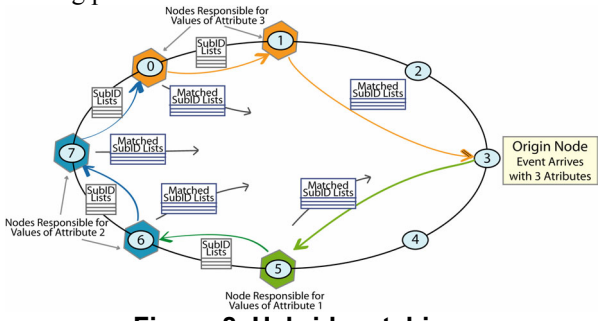

Figure 8. Hybrid matching.

Hybrid Matching

Hybrid Matching takes advantage of this fact. When we reach at a point in the distributed matching where all the declared attributes of a subscription are already checked, the subscription matches the event and it is returned back directly to the origin node. As we can see from Figure 8, it is clear that *Hybrid Matching* borrows and combines the benefits of both *Coordinated* and *Distributed Matching*.

7. Performance evaluation

7.1. Performance of event processing optimization algorithms

[4] For this algorithm to work we should also know which attributes were declared from a subscription and not only the number of them. This can be easily done by replacing the c3 field of the subscription identifier with an m-bit vector that indicates which attribute is declared.

Hop count

By straightforward analysis one can easily find that the *Distributed Matching algorithm* is the best algorithm in terms of hop count, compared to the other two that have similar performance.

Suppose an event arrives at the system that involves the communication of k broker nodes in order to collect the SubIDs that are candidates for matching. In general, under the *Coordinated Matching* algorithm, the origin broker node (where the event arrives) has to send one DHT message to each of the k nodes and each of the k nodes have to send the origin node a DHT message with the SubID lists they posses. Thus, with *Coordinated Matching*, we have to perform 2k DHT node lookups.

Under *Distributed Matching*, k+1 DHT lookups in total need be performed. They are fewer compared to *Coordinated Matching* by k-1, because *Distributed Matching* gets rid of the communication of each node with the origin node except the last one that sends back the SubID lists.

Finally, with the *Hybrid Matching* algorithm we have to perform 2k lookups, exactly the same as in *Coordinated Matching*. They are k-1 more lookups compared to *Distributed Matching*, as all nodes except the last one have to contact the origin node in order to send back the already matched subscription identifiers.

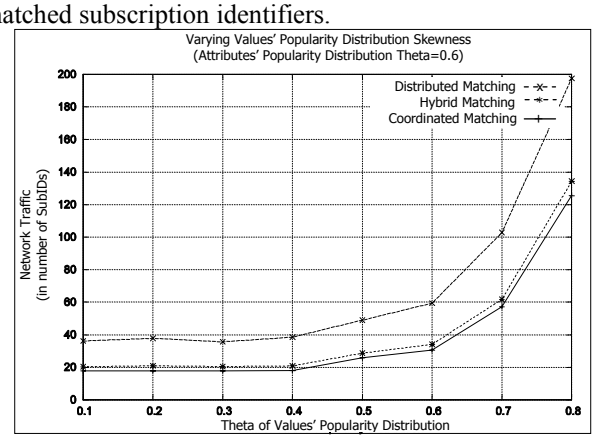

Figure 9. Varying the skewness of Values' popularity distribution.

Network Traffic

DHT hop counts, however, are just one indication. Perhaps a more important metric is the network traffic generated as a result of the above algorithms. Our specific performance metric here is the total number of SubIDs sent for the processing of each incoming event (when a SubID is sent r times, it is counted as r SubIDs). This metric is a clear indication of the bandwidth requirements during the event processing phases as they are affected by the result sizes being transmitted over the DHT.

In order to find out how the system performs in terms of generated network traffic under different skewness degrees of the popularity distributions of attributes and

their values, we have performed a number of experiments; we report the results on a series of experiments in a 128-broker network with 10,000 subscriptions and 30,000 generated events.

The value domain size of each attribute is large enough compared to the number of nodes. The number of attributes that an event or subscription can have, varies from 1 to 10 attributes (and depends on the attributes' popularity). The popularities of attributes as well as the values of each attribute follow a Zipf distribution with parameter θ, varying from 0.1 (more uniform) to 1.0 (more skewed).

In Figure 9 we vary the skewness of attribute values' popularity distribution while θ for attributes' popularity equals 0.6. We can see that the preferred algorithm is the *Coordinated Matching,* which is slightly better that *Hybrid Matching,* and considerably better than *Distributed Matching*.

Note that, despite that the two best algorithms have similar performance, with respect to the total result set sizes sent over the DHT, the matching process is performed in a distributed environment under *Hybrid Matching*. This is expected to alleviate problems related to performing the whole matching process centrally at a broker in *Coordinated Matching.*

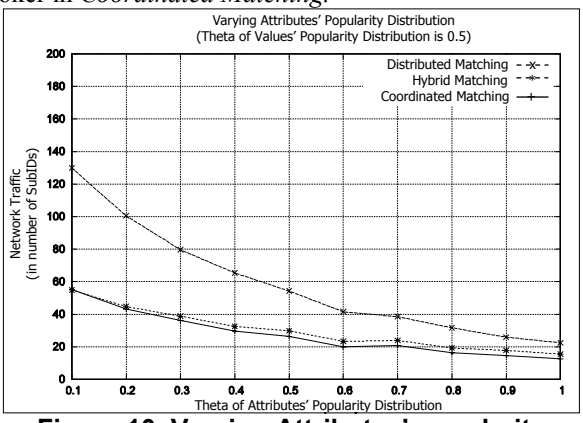

Figure 10. Varying Attributes' popularity distribution skewness with constant skewness (0.5) of values' distribution.

Distributed Matching is the worst, in general, because, as we said in section 6, SubIDs that may already match the event, are sent through the network until the last node involved in the SubID Collection phase.

In the next set of experiments we try to figure out under which circumstances the *Distributed* and *Hybrid Matching* algorithms can improve their performance compared to *Coordinated Matching*. We first change the attributes' popularity distribution with θ varying from 0.1 to 1.0. The values' popularity distribution remains the same for all value domains with θ = 0.5. The rest parameters of the experiment remain unchanged.

As you can see in Figure 10, *Coordinated Matching* is marginally better compared to *Hybrid Matching,* in all cases. Our intuition is that the filtering performed at each step of *Distributed* and *Hybrid Matching*, becomes more efficient as popular attributes that are going to return small result sets (their values' popularity is more uniform) are processed as early as possible. Moreover, the distributed matching should be performed in many steps which means that the attributes' popularity distribution should not be very skewed. In order to verify our thoughts we tuned our experimentation, so that popular attributes have uniform value distributions and less popular attributes have skewed popularity distributions for their possible values.

As we can see in Figure 11, for small values of θ (where many attributes are likely to be defined by events/subscriptions), *Hybrid Matching* performs better compared to *Coordinated Matching* because there are many filtering steps and the popular attributes with small result sets seem to further help the filtering. As the popularity of attributes becomes more skewed the mean number of attributes per event/subscription decreases and thus there are not enough steps for *Hybrid Matching* to show its worthiness. As we can see in Figure 11 all three algorithms tend to perform the same as the attributes' popularity becomes very skewed (θ value approaches 1.0).

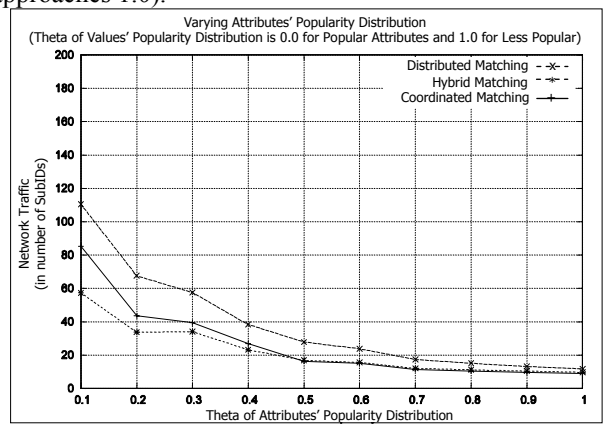

Figure 11. Varying Attributes' popularity distribution skewness. Value distribution is skewed for less popular attributes.

7.2. Load balancing

In our approach we store string values on the DHT network. In most real-world environments, attribute value and access distributions are not uniform. Such skewness may in general create storage and access load imbalances.

The intuition behind our conjecture that load imbalances are not a significant problem with our approach is based on the following observation: even though a skewed value/access distribution of an attribute can create load imbalances, in real world applications there will be tens of attributes. Further, each pub/sub

infrastructure is expected to support several applications (each with many attributes). As the total number of supported attributes increases, the load imbalances are disappearing.

By performing a number of experiments in a 128-node network we found out that even with a small, say 7-character long attribute, the domain size of each attribute is large enough to achieve 1.07 value for the maximum to minimum storage load ratio with only 6 attributes in the event/subscription schema. For smaller value domain sizes more attributes are needed in order to achieve adequate balancing. It should be obvious that the same results will be obtained regardless of whether the skewed access distributions refer to value-occurrence distributions (i.e., storage load) or value-access distribution (i.e. access load).

8. Conclusion

In this work we have shown how to leverage DHT-based P2P systems, towards building scalable, self-organizing, well-performing systems that support queries with a rich set of constraints on string attributes. We specifically focused on and presented how our algorithms can be applied in a publish/subscribe environment with a broker network implemented using a DHT. The proposed solution is DHT-independent and can be applied in every DHT infrastructure that provides the basic functionality of finding and reaching the node that stores an object with a specific key value. To our knowledge, this is the first work that shows how string attribute queries (with equality, prefix, suffix, and containment predicates) can be processed over a DHT infrastructure.

Using it, DHT-based pub/sub systems can be built, achieving better or comparable performance to traditional systems for both the subscription propagation and the event delivery phases (in terms of number of hops required for each task). Future work includes further reducing the network traffic overhead, and comparing it with that of non-DHT-based pub/sub systems.

9. Acknowledgment

We thank Ludger Fiege whose questions on a related work of ours [17] prompted us to develop the event-processing optimization algorithms of Section 6.

10. References

[1] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. "*Scribe: A large-scale and decentralized application-level multicast infrastructure*". Journal on Selected Areas in Communication, vol. 20, Oct. 2002.

[2] 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*". Proc. ACM NOSSDAV 2001.

[3] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. "*Application-level multicast using content-addressable networks*". Proc. 3rd International Workshop of NGC, vol. 2233, pages 14–29, LNCS, Springer, 2001.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. "*An efficient multicast protocol for content-based publish-subscribe systems*". Proc. 19th ICDCS 1999.

[5] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "*Design and evaluation of a wide-area event notification service*". ACM Transactions on Computer Systems, 2001.

[6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. *"Achieving scalability and expressiveness in an Internet-scale event notification service"*. Proc. ACM PODC 2000.

[7] D. Tam, R. Azimi, and H. Jacobsen. "*Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*". Proc. DISP2PC 2003.

[8] A. Rowstron and P. Druschel. "*Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*". In Proc. 18th IFIP/ACM DSP 2001.

[9] S. Ratnasamy,P. Francis, M. Handley, R. Karp, and S. Shenker. "*A scalable content addressable network*". Proc. ACM SIGCOMM 2001.

[10] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. "*Chord: A scalable peer-to-peer lookup service for internet applications*". Proc. SIGCOMM 01.

[11] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. "*Achieving scalability and expressiveness in an Internet-scale event notification service*". Proc. ACM PODC 2000.

[12] Y. B. Zhao, J. Kubiatowitcz, and A. Joseph. "*Tapestry: An infrastructure for fault-tolerant wide-area location and routing*". Tech. Rep. UCB/CSD-01-1141, Univ. of California at Berkley, Computer Science Dept. (2001)

[13] D. Karger, et al. "*Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web*". Proc. ACM STOC 1997.

[14] P. R. Pietzuch and J. Bacon. "*Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware*". Proc. DEBS'03.

[15] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, "*A Peer-to-Peer Approach to Content-Based Publish/Subscribe*". Proc. DEBS 2003.

[16] P. Triantafillou and A. Economides. "*Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems*". Proc. IEEE ICDCS 2004.

[17] P. Triantafillou and I. Aekaterinidis. "*Publish-Subscribe over Structured P2P Networks*". Proc. DEBS 2004.

[18] C. Tryfonopoulos, M. Koubarakis and Y. Drougas. "*Filtering Algorithms for Information Retrieval Models with Named Attributes and Proximity Operators*". Proc. ACM SIGIR 2004.

[19] S. Shi, G. Yang, D. Wang, J. Yu, S. Qu, and M. Chen "*Making Peer-to-Peer Keyword Searching Feasible Using Multi-level Partitioning*". Proc. IPTPS 2004.

[20] K. Aberer. "*P-Grid: A self-organizing access structure for P2P information systems*". Proc. CoopIS 2001.

[21] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. "*SkipNet: A Scalable Overlay Network with Practical Locality Properties*". Proc. USITS 2003.