PastryStrings: Content-Based Publish/Subscribe over DHT Networks

Ioannis Aekaterinidis and Peter Triantafillou

RA Computer Technology Institute and Dept. of Computer Engineering and Informatics,

University of Patras, Greece, {aikater,peter}@ceid.upatras.gr

Abstract

In this work we propose and develop a comprehensive infrastructure, coined PastryStrings, for supporting rich queries on both numerical (with range, and comparison predicates) and string attributes, (accommodating equality, prefix, suffix, and containment predicates) over DHT networks utilising prefix-based routing. As event-based, publish/subscribe information systems are a champion application class, we formulate our solution in terms of this environment.

1 Introduction

Peer-to-peer (p2p) data networks are appropriate for building large-scale distributed systems and applications since they are completely decentralised, scalable, and self organising. All participating nodes have equal opportunities and are providing services where information is exchanged directly between them. There are two families of p2p networks: *structured*, where the data placement and the network topology are tightly controlled, and the *unstructured* ones. The most prominent *structured* p2p networks are built using a Distributed Hash Table (DHT [1, 18–21, 30]). A special class of DHTs employ prefix-based routing based on Plaxton's et al. Mesh [16] (Tapestry [30], Pastry [20], Bamboo [19]). DHT architectures provide scalable resource look-up and routing with $O(log(N))$ complexity in a N-node network.

A large body of research is currently targeting the extension and employment of p2p data network architectures over either *unstructured* or *DHT-based* p2p networks ([6, 10–12, 14, 17, 29]). Related work has provided solutions for a large number of problems, from architectures and algorithms for searching relevant data, to range query processing and data integration, and has started to examine how to support join and aggregate queries. This fact testifies to the importance the distributed systems community at large is giving to being able to support data-intensive applications over large-scale network infrastructures.

Supporting a *'rich'* set of queries (queries involving prefix, suffix, containment, and equality predicates on strings, and range and comparison predicates on numerical-typed attributes) in p2p data networks may be very useful to a number of applications. A representative class of such distributed applications is systems built using the publish/subscribe (pub/sub) paradigm. With our work in this paper we contribute a comprehensive infrastructure, coined *PastryStrings*, supporting efficiently and scalably a rich set of operators on string and numerical-typed attributes. Given the popularity of the pub/sub paradigm, we focus on it and formulate our solution in terms of this environment.

2 Background and contributions

2.1 Plaxton's mesh and Pastry

Pastry [20], as well as Tapestry [30] and Bamboo [19], are all based on location and routing mechanisms introduced in [16].

Plaxton et. al. present in [16] a distributed data structure (a.k.a. Plaxton Mesh) optimised for routing and locating objects in a very large network with constant size routing tables. Assuming a static network, routing tables consist of multiple levels, where in each level i there are pointers to nodes whose identifiers (or node ids) have the same i-digit long suffix with the current node's id. The routing of messages is achieved by resolving one digit of the destination id in each step i and looking at the $i+1$ level of the local routing table for the next node. This mechanism ensures that a node will be reached in at most $m = log_{\beta}(N)$ logical hops, where N is the namespace size, β is the base of ids, and m the number of digits in an id. The size of the routing table is constant and equal to $\beta \times log_{\beta}(N)$.

Pastry [20] offers a robust, scalable, and self-organising extension to Plaxton's Mesh under a dynamic environment. The routing scheme in Pastry, is similar to the one proposed by Plaxton et. al. with routing tables of size $\beta \times log_{\beta}(N)$ (with $log_{\beta}(N)$ levels/rows and β columns per level), resulting in $log_{\beta}(N)$ logical hops to locate a node. However, prefix (instead of suffix) matching is performed in each routing step towards the destination node, while routing table entries point to the closest node with the appropriate id prefix in terms of a proximity metric (such as round-trip time, RTT). Moreover, in order to achieve reliable routing, there is the notion of a *leaf set* for each node consisting of L pointers to nodes with id numerically close to the current node's id. In Pastry there is also the notion of neighbouring nodes, which is a set of M pointers to nearby nodes according to a proximity metric and used for maintaining locality properties. Tapestry [30] and Bamboo [19] are DHTs with similar routing functionality.

2.2 The publish/subscribe paradigm

In the pub/sub paradigm, subscribing users are interested in particular events, comprising a small subset of the set of all events that publishing users may generate. Pub/sub systems [8] are separated in two major categories, according to the way subscribers express their interests; the *topic-based* and the *content-based* pub/sub systems. *Content-based* pub/sub 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 (a.k.a. events) to subscriptions (a.k.a. interests) is done based on the content (values of attributes).

The main challenge in a distributed pub/sub environment is the development of an efficient distributed matching algorithm and related efficient algorithms to store subscriptions in the network. Distributed solutions have been provided for topic-based pub/sub systems [5]. More recently, some attempts on distributed content-based pub/sub systems use routing trees to disseminate the events to interested users based on multicast techniques [3, 4, 7, 12, 22, 23, 25]. Typically, processing subscriptions and/or events in these approaches requires $O(N)$ messages in N-node networks. Additionally, there exist techniques for subscription summarization that significantly reduce the complexity [25, 26].

Some other attempts use the notion of rendezvous nodes which ensure that events and subscriptions meet in the system [15]. Some approaches have also considered the coupling of topic-based and content-based systems [31] where events/subscriptions are automatically classified in topics. However, none of these works supports string attributes with prefix, suffix, and containment predicates.

Finally, some techniques found in the literature for string indexing may also be relevant to our goals. The most promising is the technique relying on n-grams [11] which can be applied for substring matching. However, deciding on the right value of n of $n-grams$ is difficult. Thus, typically, several values of n are used, which has a multiplicative effect on the overheads associated with $n-grams$. A relevant to *PastryStrings* work is presented in [27] where pub/sub functionality is offered on top of the Chord DHT using an attribute-value model, called *AWPS*. In [13] a balanced tree structure on top of a p2p network is presented, which can handle equality and range queries.

Prior research aiming to address relevant issues of processing 'rich' queries as typified by those in a pub/sub environment and which is closest to this work includes our previous work to support numerical-attributes in a DHT-based pub/sub environment [24]. In this work we showed how to exploit DHTs and order-preserving hashing to process range subscriptions (with range R of size $|R|$), with worst-case message complexity $O(|R| + log(N))$ and events in $O(log(N))$, in an N-node network. Our more recent work in [2] presented an approach able to support string-attribute predicates with message complexity $O(l \times log(N))$ (where l is the average length of string values) for events and $O(log(N))$ for subscriptions. Both of these works were DHT-independent, relying on the underlying DHT's lookup functionality for routing events and subscriptions. In the same spirit, the work in [17] proposed a distributed trie structure called *Prefix Hash Tree* (PHT) which is built on top of a DHT p2p network and can support range queries and prefix string queries. PHT, like [2] and [24] enjoy universal applicability (as they are based solely on the DHT's lookup function). However, it too suffers from a number of drawbacks regarding its performance and particularly the message complexity of processing range and string queries. Adapting PHT to the pub/sub paradigm we would observe that range query (subscription) processing would require $O(log(N)+|R|\times log(N))$ messages. This is similar to the performance of [24], only because of the use of order-preserving hashing the latter work has $O(|R| + log(N))$ complexity (since peer nodes storing neighbouring values are network neighbours due to the order-preserving data placement). With respect to processing events matching prefix-string subscriptions (in general) PHT would exhibit a message complexity of $O(l \times log(N))$, similar to [2], since one DHT lookup is needed per character of the string. The reader should note that the value of $|R|$ can be large and that l is typically in the order of $log(N)$.

2.3 Contribution

What is very much lacking in the literature is a single unified, comprehensive DHT-based, pub/sub architecture that can support with the same structures both string and numerical-attribute events and subscriptions effectively. This implies that it is highly desirable to offer logarithmic event and subscription processing performance for both string and numerical attributes.

3 PastryStrings architecture and rationale

The two primary design choices that best describe *PastryStrings* are (i) a tuned Pastry (or any other Plaxton-like DHT) network with an alphabet base[1] β appropriately defined so as to map string values (every possible spoken word) to nodes and (ii) a tree structure (known as *string trees*) on top of Pastry dedicated for storing subscriptions and matching events to subscriptions using prefix-based routing a la Pastry.

Each tree in the *string tree* forest is dedicated to one of the β characters of our alphabet. For string queries starting with a specific character we will first locate the appropriate tree dedicated to that character and follow a path towards the "rendezvous node" inside that tree where events and subscriptions will meet. Each node in a *string tree* uses the Pastry nodes' local routing tables as a hint for the tree construction.

The architecture of *PastryStrings* consists of clients that are producers/consumers, issuing events/subscriptions, respectively. Each client is "attached" to a Pastry network node using any appropriate mechanism. Each Pastry node hosts one or more *string tree* nodes responsible for holding and processing events and subscriptions.

Consumers publish their interests with subscriptions that are stored in specific *string tree* nodes, the "rendezvous nodes". Producers generate events that are delivered only to interested consumers by collecting and 'activating' the already stored subscriptions in the "rendezvous nodes". For simplicity of presentation, we will concentrate in this section on a single-attribute event/subscription schema.

3.1 String trees

There are two types of nodes in *PastryStrings*. Network (Pastry) nodes (referred to as simply "nodes") and *string tree* nodes (referred to as $Tnodes$). A node in general hosts several $Tnodes$. Nodes have ids (assigned by Pastry) while $Tnodes$ have labels for identifying them. A $Tnode$'s label is in general a prefix-string that is meant to identify a specific $Tnode$ that is responsible for storing subscriptions matching the string label. In this case, the $Tnode$ with label 'lbl' is denoted as $Tnode_{lbl}$. Each *string tree* is denoted by T_i where $i \in S = \{a \mid a\ is\ one\ of\ the\ \beta\ characters\ of\ the\ alphabet\}$ is the character for which T_i is responsible for. This means that every label in the T_i tree starts with the same character i. Each T_i has a maximum depth (root's depth is zero) equal to the maximum allowable string-length.

A first attempt regarding the *string tree* construction is to take advantage of the routing table of each node in the Pastry

[1]The digit base is equal to 64 as a result of the $2 \times 26 = 52$ characters of the English alphabet (uppercase and lower case), the 10 numerical characters and two special symbols: space and period.

network (e.g. with id digit base $\beta = 64$)[2] and use those routing tables for (prefix-based) routing the queries to rendezvous $Tnodes$. However, this turns out not to be a good idea due to the complications introduced by the maintenance functionality in the presence of churn. Consider, for example, the case where a Pastry node A changes the entry in its routing table which was pointing to B, to point now to C because of $B's$ departure. Then, the entire subtree rooted at B would become unreachable. In this case, either this subtree should be moved to hang from C, (which implies that every $Tnode$ in the subtree would have to be replaced and be hosted by a Pastry node reachable from C) or $C's$ routing table should be updated, which implies that we would interfere with the way the Pastry network is constructed.

Thus, a better idea is to maintain an additional routing table for each $Tnode$ of our T_i trees of constant length equal to β with entries pointing to the $Tnode$'s children. The construction of this routing table for a $Tnode$ is done based on the routing table of the Pastry node hosting the $Tnode$. The routing table at each Pastry node holds $\beta \times log_\beta(N)$, entries. Each $Tnode$ uses as a hint one of the $log_\beta(N)$ levels in the routing table of the Pastry node, where the $Tnode$ is hosted, in order to build its own routing table. More precisely, if a $Tnode$ lays in depth d then it is going to use the $(d+1)\ mod\ log_\beta(N)$ level of the host's routing table. Since the alphabet for Pastry node ids and $Tnode$'s labels is the same, the aim is to do prefix-based routing over $Tnode$'s labels utilising the Pastry infrastructure for doing prefix-based routing over node ids, so as (i) to leverage the Pastry self-organisation logic and (ii) achieve short RTT where possible.

String trees are created dynamically as new requests for storing subscriptions with string-valued predicates, arrive. Since there are many *string trees* we locate the root of a specific T_i by hashing the first character of the given string with a uniformly-distributed hash function like SHA-1. Then, the node with that id will host (or already hosts) the root of T_i. The following example illustrates how *string trees* are created.

Example 1 ***Simple subscription storing.*** *In Figure 1 a simplified snapshot of the PastryStrings infrastructure is presented. In this example, we use 3-character long ids with alphabet of base $\beta = 2$. Suppose now that a user expresses her interests with two subscriptions (with identifiers $SubID_1$ and $SubID_2$) on the same attribute, setting the attribute's value to '010' and '00' respectively.*

In general, the string tree forest consists of two trees: the T_0 and T_1. Both subscriptions in this example concern T_0. To process the storage request for value '010' we should first locate the Pastry node responsible for hosting the root of the T_0 tree (say node with id '000) by hashing the character '0' and locating the Pastry node with id equal (or close) to the hashing result. We construct there the root, $Tnode_0$ and since its routing table is empty we use the host's routing table at level 1 as a hint (recall that the root lays in depth $d = 0$ and thus we look at level $(d + 1)\ mod\ 3 = 1$). The routing table of the host of $Tnode_0$ at level 1 has two entries (since we have a binary alphabet) which are copied to $Tnode_0$'s routing table. The first one is pointing to node '001' which will host $Tnode_{00}$, while the second entry points to '011' which is going to host $Tnode_{01}$. Now that we have filled $Tnode_0$'s routing table we further process the request by checking the second character of the string, '1'. We look at the root's routing table, at column '1', and select the record that matches the next digit, sending the request to the node '011'. Since initially node '011' does not host any $Tnode$, we construct there $Tnode_{01}$ and fill its

[2]If, for instance $\beta = 64$ and $strLength$, the maximum string length, is 20, then the namespace size is $64^{20} \cong 2^{120}$. If $\beta = 64$ and $strLength = 30$ then the namespace size is $64^{30} \cong 2^{180}$. Please note, that typical DHTs are reported to have a namespace size in the range of $2^{128} \ldots 2^{164}$.

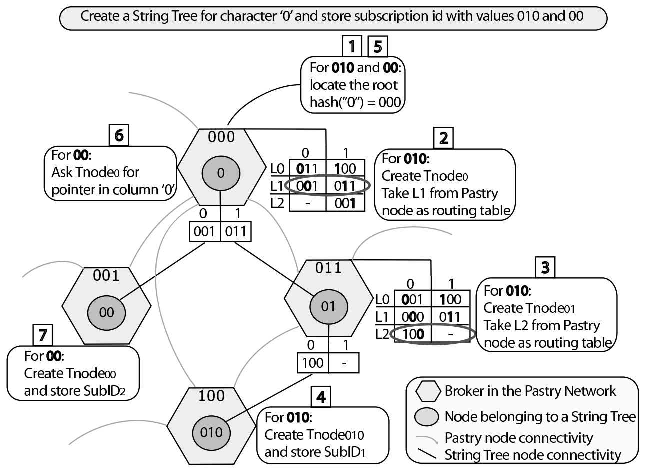
Create a String Tree for character '0' and store subscription id with values 010 and 00
1 5
For 010 and 00: locate the root hash("0") = 000
000
0
6
For 00: Ask Tnode0 for pointer in column '0'
2
For 010: Create Tnode0 Take L1 from Pastry node as routing table
L0 011 100
L1 001 011
L2 - 001
001 011
001
00
011
01
3
For 010: Create Tnode01 Take L2 from Pastry node as routing table
L0 001 100
L1 000 011
L2 100 -
100 -
7
For 00: Create Tnode00 and store SubID2
100
010
4
For 010: Create Tnode010 and store SubID1
Broker in the Pastry Network
Node belonging to a String Tree
Pastry node connectivity
String Tree node connectivity

Figure 1. String tree construction over a Pastry network with $\beta = 2$ and 3-characters long ids. The tree is constructed on demand as there is a store request for values '010' and '00'.

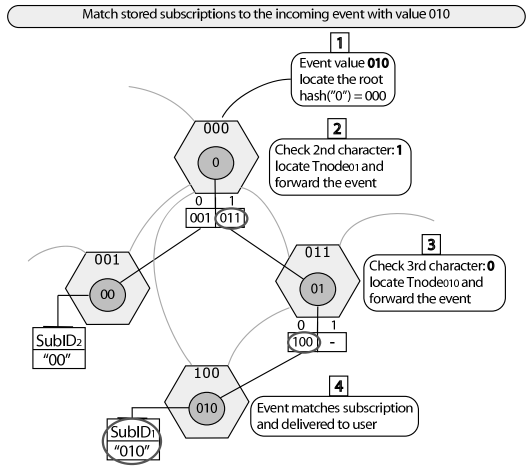
Match stored subscriptions to the incoming event with value 010
1
Event value 010 locate the root hash("0") = 000
000
0
2
Check 2nd character: 1 locate Tnode01 and forward the event
001 011
001
00
SubID2 "00"
011
01
3
Check 3rd character: 0 locate Tnode010 and forward the event
100 -
100
010
4
Event matches subscription and delivered to user
SubID1 "010"

Figure 2. Simple matching. An event arrives with value equal to '010'. Two subscriptions with values '00' and '010' are already stored in PastryStrings.

routing table with the level 2 of its host's routing table. Then we process the storage request, by examining the final character of the string, '0'. We again ask $Tnode_{01}$ for its pointer in column '0' of its routing table and we forward the request to node '100'. We construct there $Tnode_{010}$ and store $SubID_1$. The second storage request is handled similarly, as you can see in Figure 1. $\Diamond$

3.2 Event and subscription rendezvous

When an event arrives defining a value in our simple single-attribute event/subscription schema, we locate the root of the appropriate T_i and forward the event towards the $Tnode$ that has the same label as the string value in the event, by resolving one character at a time. All subscriptions found there, are considered to match the event since they have declared the same value as the event.

Example 2 ***Simple event-subscription rendezvous.*** *Figure 2 shows a snapshot of PastryStrings with two subscriptions already stored from the previous example. Suppose that an event defining the value '010' arrives at the system. We first locate the root node, $Tnode_0$, of the string tree responsible for the character '0' and send there the event. $Tnode_0$ will look-up its routing table in column '1' for the pointer to the nearby $Tnode_{01}$ and will send there the event. $Tnode_{01}$ will look-up its own routing table in column '0', for the $Tnode_{010}$ where the subscription $SubID_1$ is stored.* $\Diamond$

Having introduced the notion of *string trees*, T_i, and how event routing is performed, we see that two different routing schemes coexist in *PastryStrings*. Specifically:

- ***Pastry Routing***: is done based on Pastry's routing tables and offers the common API functions described in [20]. Pastry Routing is necessary for locating the T_i trees, and for creating *string tree* paths.

- ***String Tree Routing***: is performed within a T_i tree and exploits the $Tnodes$' routing tables in order to forward the requests towards the leaves. A typical API function is $Tc\_forward(msg, key)$, performed locally at each $Tnode$, forwarding a message (msg) to the $Tnode$ that is responsible for the value key.

We stress that Pastry routing is unaffected by *PastryStrings*. *String tree* routing uses the $Tnodes$'s routing tables. A $Tnode$'s routing table in essence constitutes another routing level, having one entry for each possible string character value.

3.3 Supporting complex string and numerical predicates

In this section we will show how to support prefix (e.g. 'abc*') and suffix (e.g. '*abc') predicates on string attributes as well as range (and comparison $\leq, \geq, \neq$) queries on numerical attributes over the *PastryStrings* infrastructure.

3.3.1 Storing subscriptions and processing incoming events: string-typed attributes

First note that a suffix operation can be easily transformed into a prefix operation if we simply proceed to examine the string from its last to its first character. Thus, without loss of generality, we shall only present how a prefix operation on string values can be applied in *PastryStrings*.

Suppose that we have a subscription with a prefix predicate. In order to appropriately store the subscription we follow the same methodology as if we had an equality predicate. Now when an event arrives, we locate the appropriate T_i tree which is responsible for the first character of the event's string value (e.g. T_a for predicate abc*) , and we then traverse a specific path of the tree (from the root $Tnode$ towards the leaves) until we find the $Tnode$ whose label is that value. During this traversal and since we perform prefix-based routing, all $Tnodes$ belonging to this path may be storing subscriptions matching a prefix of the event's value.

Example 3 ***Storing subscriptions.***

Suppose that a subscription arrives with the string predicate '00$*$'. We first locate the appropriate tree for character '0', T_0 , (in Figure 3, node '0111' hosts the root of T_0 , $Tnode_0$). Then the subscription's id is forwarded and stored in the $Tnode$ whose label equals '00' ($Tnode_{00}$).

Now suppose that an event arrives at the system with value '001'. The root of T_0 , $Tnode_0$ will be located. Then the event will be forwarded to the node hosting $Tnode_{001}$. At each $Tnode$ in the path from the root to the $Tnode_{001}$, the incoming event 'activates' the stored subscriptions if any. As you can see, there is a stored subscription in $Tnode_{00}$ and thus it is collected by our algorithm as a matched subscription. $\diamondsuit$

In addition to prefix and suffix predicates, our scheme can also support a "containment" predicate (e.g. 'a*c'). This containment operator can be easily decomposed to prefix/suffix operations. The main idea is that, for example, 'a*c' can be viewed both as prefix (i.e. 'a*') and suffix (i.e. '*c') predicates and with appropriate post-processing we can conclude on possible matching. Due to space limitations we omit the detailed methodology for this, which is straightforward extension given the support for prefix/suffix predicates.

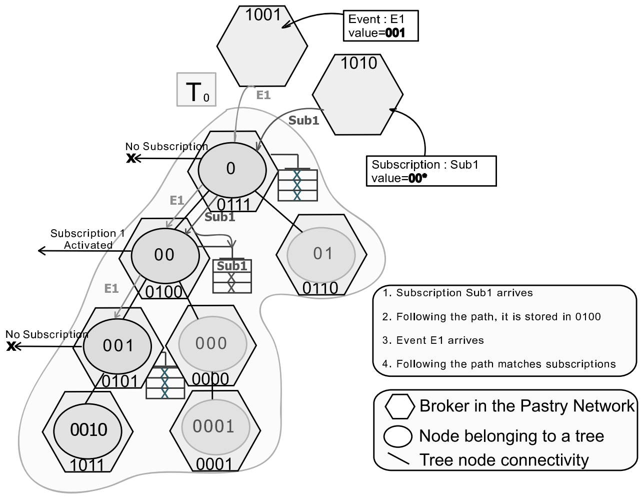

Figure 3. Storing subscriptions and performing the prefix-based matching with incoming events.

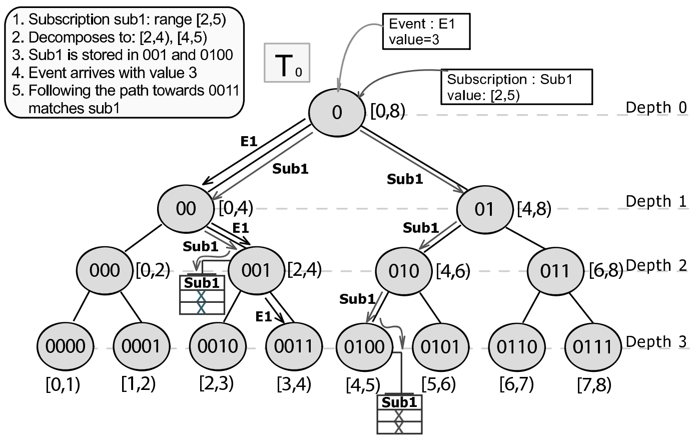

Figure 4. T_0 tree with stored $SubIDs$ for range [2,5). Event arrives with value 3 and follows the path from root to leaf, matching subscription.

3.3.2 Storing subscriptions and processing incoming events: numerical-typed attributes

PastryStrings also supports numerical attributes with range, $\leq$, $\geq$, $\neq$, and $=$ predicates. The key idea here is that every possible range of integer values may be appropriately mapped to a number of $Tnodes$ based on their labels and their location in the T_i tree. There are several ways to do so. Here we adapt the Range Search Tree (RST) approach presented in [9], and encapsulate its functionality within *PastryStrings*.

The required functionality consists of: (i) assign subranges of numerical values to $Tnodes$ and (ii) partitioning a given range into appropriate subranges. Given this functionality, when a subscription arrives declaring a range of values, we first decompose the range to proper sub-ranges. Then we locate the appropriate for each subrange $Tnode$ in T_i and store there the $SubID$. When an event arrives declaring a specific value, we transform the value to β-ary string representation, locate the appropriate T_i tree and follow the path from the root to the leaf with the same label as the given β-ary string. Each $Tnode$ in this path may have stored $SubIDs$ declaring ranges including the value of the event.

Recall that each T_i tree has a depth D (D equals the maximum string length). d denotes the depth of $Tnode_{lbl}$. We also denote with $num(lbl)$ the numerical representation of the label string lbl. For instance, given $\beta = 2$ and $lbl = 0010$ then $num(‘0010') = 2$.

With the proposed scheme, domains of size up to β^D may be easily mapped to $Tnodes$ in the *PastryStrings* infrastructure. Each $Tnode_{lbl}$ (in depth d in a T_i tree) responsible for the prefix lbl is assigned by RST to hold a specific range of values belonging to the following set:

$$[\, num(lbl) \times \beta^{D-d-1} \,,\, (num(lbl)+1) \times \beta^{D-d-1} \,)$$

In Figure 4 we present a simplified version of the *PastryStrings* infrastructure for explaining the main functionality. In this example $\beta = 2$ and string length equal to 4 ($D = 4$). As you can see, $Tnode_{001}$, is going to store ranges belonging to the interval [2,4). Using the node to interval mapping above, we get: $num(\text{“001”}) = 1$, $\beta = 2$, $D = 4$ and $d = 2$ resulting in: $[1 \times 2^{4-2-1}, (1+1) \times 2^{4-2-1}) = [2, 4)$. Each leaf $Tnode$ holds the smallest possible sub-range while any non-leaf $Tnode$ holds the union of its children sub-ranges. The union of sub-ranges of all root $Tnodes$ of our schema, covers the maximum possible domain for integer values (with size β^D).

Up to now we set our T_i trees so as to easily discover the sub-range a $Tnode$ is responsible for, based on the $Tnode_{lbl}$ label. For storing a subscription with a range predicate, we should actually break the range into subranges, find the appropriate $Tnodes$ responsible for each subrange, and then store there the $SubID$ of the subscription with the range predicate. In [9], a specific algorithm for this purpose is developed with $O(log_\beta(|R|))$ complexity for a range R with length $|R|$. Using this algorithm (with small modifications) we can decompose for example the range $[2, 5)$ into $[2, 4)$ and $[4, 5)$ (in our example, Figure 4). In this case, we store the $SubID$ in $Tnode_{001}$ and $Tnode_{0100}$.

Given the $Tnodes$ labels, we showed above that we can identify the sub-range for which the node is responsible for. However, the inverse operation (i.e., given the range find the node label, lbl), is done with the following method. Based on the way sub-ranges are mapped to $Tnodes$, we can notice that for a given decomposed subrange r, the $Tnode$ responsible for that subrange lays in depth $d = D - log_\beta|r| - 1$. Since we know the $Tnode_{lbl}$'s depth, its label is the $d + 1$ prefix of the string representation in the $\beta - base$ alphabet of the lower bound of the subrange. That is: $lbl = prefix(d+1, \beta - ary(Bl))$, where $\beta - ary(int\ x)$ is the string representation of integer x and Bl is the lower bound of the subrange. For example, as you can see in Figure 4, the decomposed subrange [3,4) is mapped to a $Tnode$ in depth $4 - 0 - 1 = 3$, and since the binary representation of the lower bound, 3 in $D = 4$ characters long string is ‘0011’, $Tnode$'s label is the 4 (3+1) character long prefix of string ‘0011’, resulting in ‘0011’.

When an event arrives declaring a value, we first compute the smallest possible subrange that includes the value. If, for example, the value is 3, the smallest possible subrange including 3, is $[3, 4)$. Then, we compute the label of the $Tnode$ responsible for that range, ‘0011’, and route the event towards $Tnode_{0011}$. All stored $SubIDs$ from root to leaf $Tnode_{0011}$ are considered to match the event.

Given that we know in advance every attribute's domain bounds (low and high bounds, BL, BH) we can easily transform the predicates $\leq$, $\geq$, and $\neq$ into range predicates. For example, the predicate $\leq V$ can be thought as the range $[BL, V]$, the predicate $\geq V$ as $[V, BH]$ and the predicate $\neq V$ as $[BL, V)$ and $(V, BH]$.

3.4 Load balancing issues

A possible limitation of the approach already described is that a very small fraction of the nodes may become bottlenecks as they are expected to absorb the access load of incoming subscriptions and events. Nodes belonging to this category are all $Tnodes$ close to the root of each one of the T_i trees. In *PastryStrings* we adopt two widely used and complementary techniques for distributing the load: (i) replicating the forest structure among the network nodes and (ii) partitioning the stored subscriptions for a popular value. Further, we can achieve even more load distribution by applying *domain relocation*.

Replicating the Forest

Replicating the forest results in balancing the access load (for storing subscriptions or locating them when events arrive) across the network. We define the replication factor (RF), as the number of replicas for each one of the β *string trees*. For this to be done we could use a hash function returning randomly RF values for each specific input. Thus, during the T_i root look-up phase (when an event or subscription is looking for the T_i tree) we could use this special function so as to reach one of the RF different replica roots (and eventually trees) and then follow a path inside that replica tree.

During the subscription storing phase, we could either repeat the same $SubID$ storing process RF times (for each replica tree) or let the node which was chosen to hold the subscription to inform the corresponding replica nodes for storing the subscription. Both approaches are easy to implement and details are omitted for space reasons.

Partitioning the Storage Load

In real applications it is likely that some $Tnodes$ may become overloaded because of storing subscription ids defining a popular value. This kind of storage load hot spots may be avoided by defining a threshold for the number of stored $SubIDs$ which when it is exceeded the $Tnode$ chooses randomly another $Tnode$ for further storing $SubIDs$. Each $Tnode$ under this scheme maintains pointers to other $Tnodes$ holding $SubIDs$ for the same value, so as to be possible to collect all matched subscriptions for an incoming event.

Numerical Attribute's Domain Relocation

A typical *PastryStrings* configuration can support extremely large domains of integer values, Dm. However, each numerical-typed attribute, a_i , is expected to have a much smaller domain, Dm_i. Given that some ranges are expected to be very popular, a small set of $Tnodes$ are expected to absorb a great load of requests for storing and retrieving $SubIDs$. With this observation in mind, we propose to distribute each Dm_i in our domain Dm adding an attribute-specific base value, b_i, to each attribute's i value. This kind of relocation will result in spreading the attributes' values across a large number of *string trees* and $Tnodes$ and will then ameliorate load balancing problems.

3.5 Self-organisation

The self organisation of the *string tree* forest is required in highly dynamic p2p networks with frequent node arrivals/departures and failure/recoveries. In order to treat failures successfully we must ensure *string tree* connectivity, so we need extra routing state per $Tnode$. This extra state, consists of pointers to a descendant node (that is a child of a $Tnode$'s child) and two pointers to the left and right siblings in the T_i tree structure.

3.5.1 Node Arrival

The Pastry protocol facilitating new arrivals is briefly as follows: the new node with id X sends a 'join' message to an already known node A, which then routes the join message to node Z with id numerically closest to X. Then all nodes in the path from A to Z send their state tables to X, which then builds its own routing table and informs other nodes, if necessary, for its arrival.

In general, p2p networks are very sparse. Thus, a node may be responsible for processing messages that are sent to an

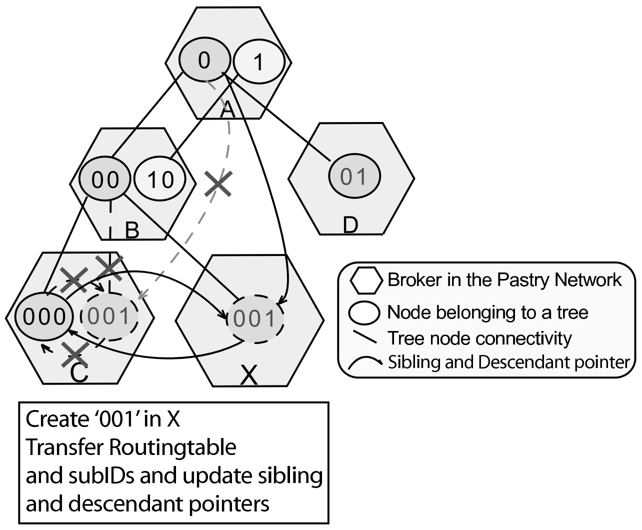

Figure 5. Node X arrives in *PastryStrings* **with T_0 and T_1 trees already deployed. T_0 reconstruction on arrival of X.**

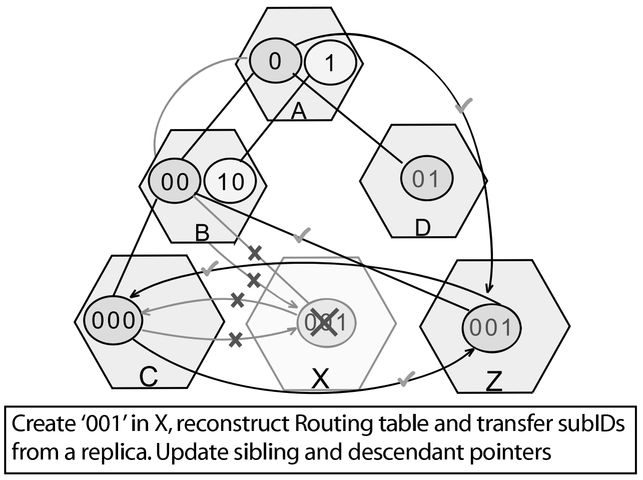
0
1
A
00
10
B
01
D
000
C
001
X
001
Z
Create '001' in X, reconstruct Routing table and transfer subIDs
from a replica. Update sibling and descendant pointers

Figure 6. Node X leaves *PastryStrings* **network. T_0 reconstruction on X's departure.**

absent node. In *PastryStrings* this results in nodes hosting more than one $Tnodes$. For example, in Figure 5 node C hosts $Tnode_{000}$ and $Tnode_{001}$. However, $Tnode_{001}$ belongs to an absent Pastry node. Upon the arrival of a new node, our system should reconstruct appropriately the string tree structure and supply the new node with the data ($SubID$ lists) that it is responsible for handling.

Suppose now, that a Pastry node B, (Figure 5), detects the arrival of node X and updates its routing table at level l in order to point to the new node. If there is any hosted $Tnode$ laying in depth $(l-1)\ mod\ log_{\beta}(N)$ in its own T_i tree (in Figure 5 $Tnode_{00}$), then node X will host a new child for that $Tnode$ (if the appropriate entry in $Tnode$'s routing table is not empty, as in this example, then the incoming node, X, should host the child hosted by the node pointed in the routing table). What we do in this case, is to update father's ($Tnode_{00}$) routing table to point to X. Then we should inform the child (in Figure 5 $Tnode_{001}$) that it should be hosted in X and thus to change host node. Changing host node, means that we create a new $Tnode$ in X, we transfer the routing table and the stored $SubIDs$ there and we update descendant and sibling pointers for each $Tnode$ that points to C so as to now point to X.

3.5.2 Node Departure

A node in Pastry may depart without warning due to a network failure or leave the network at its own will. In both cases the way the string tree structure is self-organised is almost as above. The only difference is in the way $SubID$ lists stored in the failing node, are recovered. Pastry provides mechanisms for perceiving if a node has failed. More precisely, when the immediate neighbours of a node in the node id space fail to communicate, then the node is considered failed. In this case, the routing state of nodes having a pointer to the failed node is repaired by finding a new node appropriate for handling messages designated for the failed one.

Suppose now that the failed node is X (Figure 6) and node B that hosts the father of one of the $Tnodes$ in X ($Tnode_{00}$ in Figure 6), notices its absence. Pastry-specific protocols will take control and update the routing table of B pointing to a new node, Z. This action will force the father $Tnode$ ($Tnode_{00}$ in Figure 6) to update his own routing table pointing to Z. The real challenge in coping with node failures is how to retrieve and deliver to Z the $SubID$ lists stored in the failed node as well as to reconstruct the routing table of $Tnode_{001}$ in Z. If node X leaves the network at its own will, then X may easily communicate with Z in order to deliver its stored lists. If X's departure is unexpected then the only solution for recovering the stored $SubID$ lists is to have at least one replica forest in order to contact the replica nodes and retrieve the lists. Regarding the routing table of $Tnode_{001}$, when node X leaves at its own will, it may inform node Z about the $Tnode_{001}$'s children by copying the routing table to Z. When X suddenly fails then we need one (or more depending on the fault tolerance level) extra routing pointer pointing to one of their descendants and two more pointers to their left and right siblings, per $Tnode_.$ Then when the father $Tnode_{00}$ in B is triggered upon the failure of X, will inform one of $Tnode_{001}$'s children (using the descendant pointer) about the new node (Z) hosting $Tnode_{001}$. Then that child will inform its siblings about the existence of Z which in turn will send a special message to Z in order to help Z reconstruct the missing routing table.

4 Multi-dimensional events and subscriptions

So far, we have presented *PastryStrings* under a single-attribute event/subscription schema. In real world pub/sub systems events and subscriptions are defined over a schema that supports A attributes. Each attribute a_i consists of a name, type, and a value $v(a_i)$. A $k-attribute$ $(k \leq A)$ event is defined to be a set of k values, one for each attribute. Similarly, a subscription is defined through an appropriate set of predicates over a subset of the A attributes of the schema.

The allowed operators are: (i) prefix (e.g. *abc**), (ii) suffix (e.g. **abc*), (iii) equality, and (iv) numerical range. An event matches a subscription *if and only if* all the attribute predicates of the subscription are satisfied.

The subscription identifier in our approach, $SubID$ is the concatenation of three parts: c_1, c_2, and c_3. c_1 represents the id of the node where the subscription arrived from a connected to that node client and keeps metadata information about the subscription, c_2 refers to the key of the subscription for identifying it among the stored ones at c_1, and c_3 is the number of declared attributes in the subscription.

4.1 Processing incoming subscriptions

We maintain four lists (initially empty) in every $Tnode$ for every attribute a_i of our schema. These are the $L_{ai-pref}$ and $L_{ai-suff}$ lists, where we store the $SubIDs$ of the subscriptions that contain prefix or suffix predicates on attribute a_i , respectively, the L_{ai-eq} list dedicated to equality predicates, and the L_{ai-num} dedicated to numerical predicates.

Storing subscriptions is done by appropriately storing the $SubID$ in at least c_3 nodes[3] using the methodology presented earlier. Briefly, we process each attribute a_i of the subscription and (i) when dealing with prefix predicate we store $SubID$ at

[3] If all attributes in the subscription involve predicates on strings then c_3 $Tnodes$ must be reached. However, if ranges are defined, then each range R, may be translated into $O(log_{\beta}(|R|))$ string values and thus the number increases.

Storing Subscriptions

Notation

$SubID$: subscription identifier, a_i : attribute *i*, $v(a_i)$: value of attribute a_i , prefix(x,j) : j-characters-long prefix of string x
L_{ai-num} , L_{ai-eq} , $L_{ai-pref}$, $L_{ai-suff}$: List of $SubIDs$ for attribute a_i with numerical, equality, prefix, or suffix constraint
inv(x) : inverts the string x, (inv(abc) = cba), h(): DHT's hash function (e.g. SHA-1), $Tnode_{lbl}$: $Tnode$ in the T_i tree with label lbl

Function: LocateAndStore($v(a)$, $SubID$)

```
01. Create a node id with value h( prefix($v(a)$ , 1 ) )
02. Go there (the root of the tree) and follow the path towards $Tnode_{v(a)}$
03.     If attribute $a$ has an equality (or prefix, or suffix, or numerical) constraint
04.         store $SubID$ in that node in the $L_{ai-eq}$ (or $L_{ai-pref}$ , or $L_{ai-suff}$ , or $L_{ai-num}$ ) list.
04.     endif
```

Main Procedure

```
01. For every attribute $a_i$ in subscription $SubID_j$ loop
02.     If $a_i$ has a numerical constraint
03.         decompose $v(a_i)$ and translate subranges to labels
04.         for every label in the decomposed set loop
05.             LocateAndStore( label,$SubID_j$ )
06.         end loop
07.     else if $a_i$ has a suffix constraint
08.         LocateAndStore( inv($v(a)$ ),$SubID_j$ )
09.     else if $a_i$ has a prefix constraint
10.         LocateAndStore( $v(a)$ ,$SubID_j$ )
11.     end if
12. end Loop
```

Table 1. The procedure of storing subscription identifiers in *PastryStrings*.

$L_{ai-pref}$ of $Tnode_{v(a_i)}$ ($v(a_i)$ is the attribute's value), (ii) when dealing with suffix predicate we invert $v(a_i)$ and store $SubID$ at $L_{ai-suff}$ of $Tnode_{inv(v(a_i))}$, (iii) when dealing with equality predicate we store $SubID$ at L_{a-eq} of $Tnode_{v(a_i)}$, and finally (iv) when dealing with numerical values we decompose the range into subranges and following the methodology presented earlier we store $SubID$ at the L_{ai-num} of all appropriate $Tnodes$. The procedure of storing subscriptions can be seen in Table 1.

4.2 Event processing and matching

Suppose now, that an event arrives at the system with $N_{a-event}$ attributes defined. The $SubID$ *Lists Collection Phase* (Table 2), starts by processing each attribute separately. It first locates the root $Tnode$ of the appropriate tree and then the event is forwarded towards the $Tnode_{v(a_i)}$. In each $Tnode$ in the path towards $Tnode_{v(a_i)}$, we collect all stored lists for the given attribute and send them to the next $Tnode$ in the path. At each step of this process, we merge the previously collected lists of each kind resulting in four major lists which are finally returned back to the node where the event arrived where the matching is performed. Those lists are the $L_{ai-NUMERICAL}$, $L_{ai-EQUALITY}$, $L_{ai-PREFIX}$, and $L_{ai-SUFFIX}$ lists[4].

The next step, termed *Matching Phase*(Table 3), is actually the event-subscriptions matching process. Suppose, now, that a subscription $SubID_k$ is found to be in at least one of the collected lists. Assume that this subscription consists of $N_{a-sub-k}$

[4]In fact in order to collect the $L_{a\ -SUFFIX}$ list we should repeat the same procedure with the inverted string value of the event.

Event Processing and Matching

Notation

$SubID$: subscription identifier, a_i : attribute i, $v(a_i)$: value of attribute a_i, inv(x) : inverts the string x, (inv(abc) = cba)

$N_{a-sub-i}$: the number of attributes defined in a subscription i, $N_{list-sub-i}$: the number of collected lists that $SubID_i$ is stored

L_{ai-num}, L_{ai-eq}, $L_{ai-pref}$, $L_{ai-suff}$: List of $SubIDs$ for attribute a_i with numerical, equality, prefix, or suffix operation

$L_{ai-\{NUMERICAL,EQUALITY,PREFIX,SUFFIX\}}$: Delivery List with candidate $SubIDs$ for numerical, equality, prefix, and suffix matching

$Tnode_{str}$: $Tnode$ in the T_i tree with label str

SubID Lists Collection Phase

```
01. for every $event_j$ arriving at the system, loop
02.     for every attribute $a_i$ in the $event_j$, loop
03.         if $v(a_i)$ is numerical value
04.             translate it to the appropriate $Tnode$ label: $lbl$
05.             locate the right $T_i$ tree and for each node in the path towards $Tnode_{lbl}$
06.                 retrieve the $L_{ai-num}$ found there and merge it with the previously collected to $L_{ai-NUMERICAL}$
07.         end if
08.         if $v(a_i)$ is a string value
09.             locate the right $T_i$ tree and for each node in the path towards $Tnode_{v(a_i)}$
10.                 retrieve the $L_{ai-eq}$ found there and merge it with the previously collected to $L_{ai-EQUALITY}$
11.                 retrieve the $L_{ai-pref}$ found there and merge it with the previously collected to $L_{ai-PREFIX}$
12.             locate the right $T_i$ tree and for each node in the path towards $Tnode_{inv(v(a_i))}$
13.                 retrieve the $L_{ai-suff}$ found there and merge it with the previously collected to $L_{ai-SUFFIX}$
14.         end if
15.     end loop
16. end loop
```

Table 2. Collecting the SubIDs of subscriptions that are candidate for matching the event.

attributes ($N_{a-sub-k}$ is obtained from the field c_3 of the subscription identifier). Then, this subscription is considered to match the event if it appears in exactly $N_{a-sub-k}$ lists collected from the network, since an event matches a subscription if and only if all of the subscription's predicates are satisfied. 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 utilising field c_1.

A number of distributed algorithms for event matching can be used to avoid performance problems stemming from the use of a per-event coordinator for event matching. These are orthogonal issues and outside the scope of this paper. We refer the interested reader to [2].

4.3 Message complexity analysis

The Pastry infrastructure ensures that at most $O(log_{\beta}(N))$ messages are needed to reach any node in a system with namespace size N and node identifiers of base β.

During the subscription storage procedure, the average number of messages needed to store a $SubID$ is equal for all allowable operations on strings. Thus, for string-typed attributes we need $O(log_{\beta}(N))$ messages in order to reach the root of the appropriate T_i *string tree* (i.e. one DHT lookup) and then at most $O(log_{\beta}(N))$ messages in order to locate the $Tnode$ inside the *string tree* that will accommodate the subscription id (i.e. one message per string character), yielding a

Event Processing and Matching

Matching Phase

```
01. for every $SubID_k$ found in the Delivery Lists, loop
02.     retrieve the number of attributes defined from the $c_3$ field: $N_{a-sub-k}$
03.     count the number of Delivery lists where $SubID_k$ is stored: $N_{list-sub-k}$
04.     if $N_{a-sub-k}$ equals $N_{list-sub-k}$ we have a match
05.         remove $SubID_k$ from all lists
06.         store $SubID_k$ in the Matching list $L_{matching}$
07.     end if
08. end loop
```

Table 3. Matching collected SubIDs to event.

total of $O(log_\beta(N))$ messages. For numerical-typed attributes, if the mean size of ranges is $|R|$, the range decomposition and string translations used in RST results in $O(log_\beta|R|)$ string values. Thus $O(log_\beta|R| \times log_\beta(N))$ messages are required for storing a numerical range. $O(log_\beta|R|)$ is expected to be small compared to $O(log_\beta(N))$ in real-life pub/sub applications with range sizes ($|R|$) very small compared to N. Thus, we could view $O(log_\beta|R|)$ as a relatively small constant.

Regarding the matching process and more precisely the $SubID$ Lists *Collection Phase*, for every attribute in the event, we should first locate the appropriate *string tree*, which requires $O(log_\beta(N))$ messages (i.e. one DHT lookup). Then we locate the right $Tnode$ and then collect the $SubID$ lists stored in all $Tnodes$ in the path from the root to $Tnode$. This step requires at most $O(log_\beta(N))$ messages. Thus, in general, for each attribute of the event, $O(log_\beta(N))$ messages are required in order to collect the stored $SubID$ lists.

5 Experimentation and performance evaluation

We performed a number of experiments in a 1000-broker *PastryStrings* simulated network with up to 140,000 $SubIDs$ stored and 160,000 generated requests for collecting $SubID$ lists (the exact number depends on the skewness of relevant distributions). We used a Zipfian[5] popularity distribution for attributes, which determines the actual number of attributes in an event or subscription, varying from 1 to 10. The popularity of values for each attribute also follows a Zipfian distribution. As the skewness of the values' distribution plays a key role here, we varied ϑ from 0.0 to 1.6 (to test for load imbalances). Regarding the distribution of numerical and string typed attributes in the subscriptions, half of the attributes are numerical (and the rest strings). Unless stated otherwise, half of the numerical attributes are declaring equalities while the other half ranges on integer numbers. The domain of each numerical attribute is $[0, 70000]$ and the size of each range defined in subscriptions, unless stated otherwise, was varied from 1 (equality) to 20 (following a uniform distribution). The β base of our alphabet was set to 15 and the maximum string length to 5.

[5]the frequency of occurrence of the n^{th} ranked item is defined to be $\frac{1}{n^\vartheta}$. Typical values of the parameter ϑ are: $0.0 \leq \vartheta \leq 1.6$, where large values of ϑ denote very skewed distributions and $\vartheta = 0.0$ yields a uniform distribution

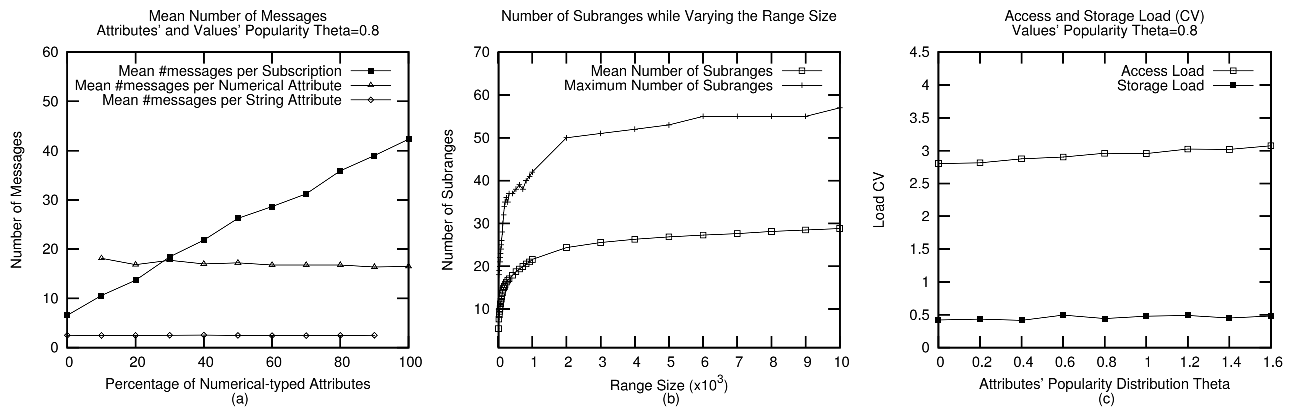

Figure 7. The effect of ranges in subscription processing and access/storage load balancing for different attributes' popularity skewness.

5.1 The effect of ranges on subscription processing performance

In the first set of experiments (Figure 7(a)) we varied the percentage of attributes defining a numerical value (equality or range) in each subscription from 0% (no ranges at all) to 100%. Our performance metric here is the number of messages needed to store a subscription. You can see that as the percentage of range predicates per subscription increases, so does the number of messages for subscription processing. In fact the message count for the only-ranges case is four times larger than the no-ranges case. This is as expected from our analysis since numerical attributes need to be decomposed into subranges for each of which a different $Tnode$ is responsible.

To have a clear picture with respect to the number of subranges a range is decomposed into, we varied the size of the range and counted the subranges produced. Then for each particular range size, $|R|$ we created $100,000$ ranges with size ranging from 0 to $|R|$ following a uniform distribution and counted the average and maximum number of subranges. As you can see in figure 7(b) as the range size increases so does the average number of subranges, but the increase is logarithmic. In the same figure you can also observe that the maximum number of subranges into which an 1000-size range is decomposed is 42 which is the trend of the expected number of subranges (and eventually the number of $SubID$ storing requests) in the case of an inequality ($\neq$) predicate under the existence of an attribute's domain size of up to 1000 integer values.

5.2 Load balancing

Our main objective with this set of experiments, is to observe how load balancing is affected by changing the skewness of the attributes' and values' popularity distribution.

5.2.1 The data distribution effect on access and storage load

Our specific performance metric here is the coefficient of variation (CV) of access and storage load. CV for storage load is defined as the ratio of the standard deviation of the number of stored $SubIDs$ in a network node across *PastryStrings*,

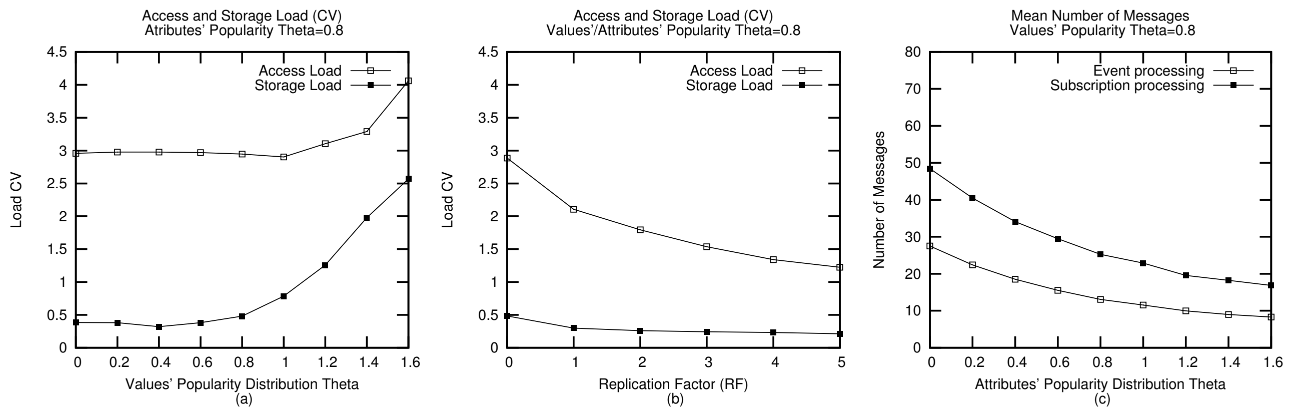

Figure 8. Access/storage load balancing and messages needed for event/subscription processing.

to the mean value of stored $SubIDs$ in a node. CV for access load involves both the number of requests for retrieving $SubIDs$ and routing requests to nodes. As you can see in Figure 7(c) the coefficient of variation of storage and access load slightly increases as the attributes' skewness increases because the number of attributes per event/subscription decreases (fewer popular attributes are defined per event/subscription). Having fewer attributes results in more workload for fewer nodes. However, the difference between skewed and uniform workloads seems to be small.

Figure 8(a) shows how access and storage CV increases as values become more popular. This can be explained by the fact that as ϑ increases fewer and popular values are defined in each subscription/event. This results in overloading a few $Tnodes$ in the *PastryStrings* infrastructure while others remain lightly-loaded. To study how storage load balancing can be improved by applying the attributes' domain relocation (recall the discussion in section 3.4) we made a number of experiments varying the attributes' values popularity distribution skewness and observed that the CV for storage is not affected by more skewed values' popularity distributions. This comes from the fact that range processing dominates string processing during the subscription storage phase (since ranges involve the contacting of many more $Tnodes$) and the domain relocation further distributes hot values and ranges to different $Tnodes$.

5.2.2 Effect of replication on access and storage load

In this set of experiments we tried to measure how the replication of the *string tree* forest helps distributing evenly the storage and the access load during the $SubID$ storage phase and the $SubID$ collection phase, respectively, by varying the replication factor RF (the number of replica *string tree* forests).

As you can see in Figure 8(b) the coefficient of variation (CV) of the number of $SubIDs$ stored in network nodes (storage load) as well as the number of retrieval and routing requests for $SubIDs$ (access load) decreases and approaches 0 (fully balanced), as the replication factor increases. Another important observation is that the network is more imbalanced when dealing with access load compared to the storage load. This is due to the fact that subscriptions involving range predicates may generate a greater number of storage requests (recall that the range decomposition may result in storing the $SubID$ in $log_{\beta}|R|$ nodes). This number of $SubIDs$ stored is greater and have a more evenly distribution among brokers, compared to the distribution of event requests.

5.3 Event processing and matching

5.3.1 Number of messages

We also conducted a number of experiments in order to measure the number of messages per event needed to collect all $SubID$ lists and the number of messages per subscription in order to store $SubIDs$ as a function of the attributes' and values' popularity distribution skewness.

We observed that as the the skewness of the attributes' popularity distribution increases, fewer attributes are involved per event/subscription and thus the mean number of messages per event/subscription decreases (Figure 8(c)). Again, our main observation here is that subscription processing needs more communication overhead, compared to event processing. This is the right design choice. In real pub/sub systems events are expected to arrive in the system in rates much greater than subscriptions rates. Thus, it is deemed necessary to perform a fast and efficient event matching. We also increased the skewness of the values' popularity distribution and we have noticed that the number of messages per event/subscriptions is not affected by the value popularity.

5.3.2 Network traffic

Our specific performance metric here is the total number of $SubIDs$ sent for the processing of each incoming event.

We varied the skewness of the attribute values' distribution while ϑ for the attributes' popularity equals 0.8. Detailed results are omitted for space reasons. Briefly, we observed that as the values' popularity distribution becomes more skewed (varying ϑ of Zipfian popularity distribution from 0.0 to 1.6) the traffic increases by a factor of 5 since most of the incoming events, contact a small number of nodes that hold the majority of stored subscription identifiers. We also varied the attributes' popularity distribution (varying ϑ from 0.0 to 1.6) and we observed that when the distribution is skewed (ϑ approaches 1.6) the network traffic is decreased by a factor of 2.5, since fewer and popular attributes are chosen in every incoming event.

6 Conclusions

In this work we have contributed an architecture for building scalable, self-organising, well-performing systems that support queries with a rich set of predicates on string and numerical typed attributes. We specifically focused on and presented how our algorithms can be applied in a pub/sub environment with a broker network implemented using a DHT network. The distinguishing feature of our work is that is shows how to leverage specific DHT infrastructures to ensure logarithmic message complexity for both event and subscription processing, and for both rich string and numerical predicates. *PastryStrings* is DHT-specific, but does not interfere with the DHT internals; it simply leverages its key information.

Our experimentation results show that *PastryStrings* can handle subscriptions with rich string and numerical predicates efficiently and scalably, i.e., with small number of messages, good load distribution to network nodes, and small network bandwidth requirements.

References

[1] K. Aberer, *P-grid: A self-organizing access structure for p2p information systems*, CoopIS'01, 2001.

[2] I. Aekaterinidis and P. Triantafillou, *Internet scale string attribute publish/subscribe data networks*, 14th ACM Conference on Information and Knowledge Management (CIKM05), 2005.

[3] G. Banavar, T. Chandra, B. Mukherjee, and J. Nagarajarao, *An efficient multicast protocol for content-based publish-subscribe systems*, 19th ICDCS, 1999.

[4] A. Carzaniga and A. Wolf, *Forwarding in a content-based network*, Proc. SIGCOMM'03, 2003, 2003.

[5] 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 (2002).

[6] A. Crespo and H. Garcia-Molina, *Routing indices for peer-to-peer systems*, ICDCS'02, 2002.

[7] G. Cugola, E. D. Nitto, and A. Fuggetta, *The jedi event-based infrastructure and its application to the development of the opss wfms*, (2001).

[8] P. Th. Eugster, P. Felber, R. Guerraoui, and A. M. Kermarrec, *The many faces of publish/subscribe*, ACM Computing Surveys, 2003.

[9] Jun Gao and Peter Steenkiste, *An adaptive protocol for efficient support of range queries in dht-based systems*, 12th IEEE International Conference on Network Protocols (ICNP'04), 2004.

[10] A. Gupta, O. D. Sahin, D. Agrawal, and A. El Abbadi, *Meghdoot: Content-based publish subscribe over p2p networks*, Middleware'04, 2004.

[11] M. Harren, J. M. Hellerstein, R. Huebsch, B. Thau Loo, S. Shenker, and I. Stoica, *Complex queries in dht-based peer-to-peer networks*, IPTPS'02, 2002.

[12] R. Huebsch, J. M. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica, *Querying the internet with pier*, VLDB'03, 2003.

[13] H. V. Jagadish and B. C. Ooi and Q. H. Vu, *BATON: A Balanced Tree Structure for Peer-to-Peer Networks*, VLDB'05,2005.

[14] S. W. Ng, B. C. Ooi, K. L. Tan, and A. Zhou, *Peerdb: A p2p-based system for distributed data sharing*, ICDE'03, 2003.

[15] P. R. Pietzuch and J. Bacon, *Peer-to-peer overlay broker networks in an event-based middleware*, DEBS03, 2003.

[16] C. G. Plaxton, R. Rajaraman, and A. W. Richa, *Accessing nearby copies of replicated objects in a distributed environment*, ACM Symposium on Parallel Algorithms and Architectures, 1997.

[17] S. Ramabadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker, *Brief announcement: Prefix hash tree*, ACM PODC, 2004.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content addressable network*, ACM SIGCOMM'01, 2001.

[19] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz, *Handling churn in a dht*, Proceedings of the 2004 USENIX Annual Technical Conference (USENIX '04), 2004.

[20] A. Rowstron and P. Druschel, *Pastry: Scalable and distributed object location and routing for large-scale peer-to-peer systems*, ACM Middleware'01, 2001.

[21] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, *Chord: A scalable peer-to-peer lookup service for internet applications*, SIGCOMM'01, 2001.

[22] C. Tang and Z. Xu, *pFilter: Global Information Filtering and Dissemination*, IEEE FTDCS'03,2003.

[23] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann, *A peer-to-peer approach to content-based publish/subscribe*, DEBS'03, 2003.

[24] P. Triantafillou and I. Aekaterinidis, *Publish-subscribe over structured p2p networks*, DEBS 04, 2004.

[25] P. Triantafillou and A. Economides, *Subscription summarization: A new paradigm for efficient publish/subscribe systems*, IEEE ICDCS04, 2004.

[26] P. Triantafillou and A. Economides, *Subscription summaries for scalability and efficiency in publish/subscribe systems*, DEBS'02, 2002.

[27] C. Tryfonopoulos, S. Idreos, and M. Koubarakis, *Publish/subscribe functionality in ir environments using structured overlay networks*, In Proc. of ACM SIGIR, 2005.

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

[29] B. Yang and H. Garcia-Molina, *Improving search in peer-to-peer systems*, ICDCS'02, 2002.

[30] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, *Tapestry: A global-scale overlay for rapid service deployment*, IEEE Journal on Selected Areas in Communications (2003).

[31] D. Tam and R. Azimi and H-A. Jacobsen, *Building Content-Based Publish/Subscribe Systems with Distributed Hash Tables*, Proc. DBISP2P'03, 2003.