Stabilizing Peer-to-Peer Spatial Filters

Silvia Bianchi
University of Neuchâtel, Switzerland
silvia.bianchi@unine.ch

Ajoy Datta
UNLV, School of Computer Science, USA
datta@cs.unlv.edu

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Maria Gradinariu
LIP6, INRIA-Université Paris 6, France
maria.gradinariu@lip6.fr

Abstract

In this paper, we propose and prove correct a distributed stabilizing implementation of an overlay, called DR-tree*, optimized for efficient selective dissemination of information. DR-tree copes with nodes dynamicity (frequent joins and leaves) and memory and counter program corruptions, that is, the processes can connect/disconnect at any time, and their memories and programs can be corrupted. The maintenance of the structure is local and requires no additional memory to guarantee its stabilization. The structure is balanced and is of height $O(log_m(N))$, which makes it suitable for performing efficient data storage or search.*

We extend our overlay in order to support complex content-based filtering in publish/subscribe systems. Publish/subscribe systems provide useful platforms for delivering data (events) from publishers to subscribers in a decoupled fashion in distributed networks. Developing efficient publish/subscribe schemes in dynamic distributed systems is still an open problem for complex subscriptions (spanning multi-dimensional intervals). Embedding a publish/subscribe system in a DR-trees is a new and viable solution. The DR-tree overlay also guarantees subscription and publication times logarithmic in the size of the network while keeping its space requirement low (comparable to its DHT-based counterparts). Nonetheless, the DR-tree overlay helps in eliminating the false negatives and drastically reduces the false positives in the embedded publish/subscribe system.

Keywords: Content-based routing, publish/subscribe, peer-to-peer, self-organization, stabilizing dynamic R-trees.

1 Introduction

Unlike conventional routing, where packets are routed based on a limited, fixed set of attributes (e.g., source/destination IP addresses and port numbers), content-based publish/subscribe systems route messages on the basis of their content and the interests of the message consumers. Consumers typically specify *subscriptions*, indicating the type of content that they are interested in, using some predicate language. For each incoming message, a *content-based router* matches the message contents against the set of subscriptions to identify and route the message to the (sub)set of interested consumers. Therefore, the "destination" of a message is generally unknown to the data producer and is computed *dynamically* based on the message contents and the active set of subscriptions.

Traditional content routing systems are usually based on a fixed infrastructure of reliable brokers that filter and route messages on behalf of producers and the consumers. This routing process is a complex and time-consuming operation, as it often requires the maintenance of large routing tables on each router and the execution of complex filtering algorithms (e.g., [2, 10, 15]) to match each incoming message against every known subscription. The use of summarization techniques (e.g., subscription aggregation [7, 9]) alleviates those issues, but at the cost of significant control message overhead or a loss of routing accuracy.

Another approach to content routing is to design it free of broker infrastructure, and organize publishers and consumers in a peer-to-peer overlay through which messages flow to interested parties. By using an adequate structure and gathering consumers with similar interests to form *semantic communities*, messages can be quickly disseminated within a community without incurring significant filtering cost [11].

Obviously, for such techniques to be efficient, one needs to properly structure the overlay to: avoid *false negatives* (a registered consumer failing to receive a message it is interested in); minimize the occurrence of *false positives* (a consumer receiving a message that it is not interested in); and *self-adapt* to the dynamic nature of the systems, with peers joining, leaving, and failing.

In this paper, we propose to use *R-tree* based spacial filters to construct a peer-to-peer overlay optimized for se-

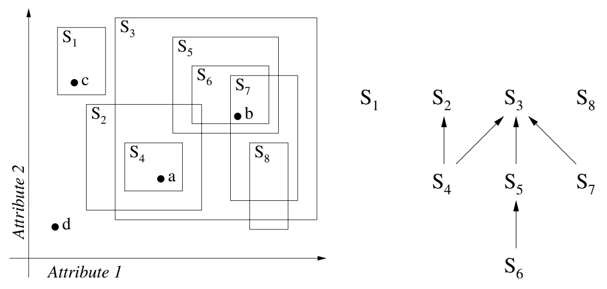

Figure 1. Sample subscriptions ($S_1, \ldots, S_8$) and events ($a, \ldots, d$) with two attributes (left) and associated containment graph (right).

lective dissemination of information. Our overlay extends the classical R-tree structures in several ways: First, it is distributed and completely decentralized. Second, it is specifically designed to meet the requirements of the publish/subscribe model. In particular, the DR-tree is constructed to preserve the containment relationship that may exist between different subscriptions. Third, it includes self-stabilizing protocols that guarantee consistency despite failures and rapid changes in the peer populations. In short, in this paper we make the following contributions: we design a novel DHT-free, stabilizing overlay and extend it to embed publish/subscribe systems with complex (spacial) filters. Our overlay guaranties publish/subscribe in logarithmic time. The recovery time in face of joins/leaves or memory corruptions is logarithmic in the size of the network. The overlay self-organizes in a balanced tree with a memory cost per node polylogarithmic in the size of the network. We prove the correctness of our overlay in spite of transient faults, joins, and leaves. Additionally, we analytically analyze the overlay resistance to churn.

The rest of the paper is organized as follows: Section 2 introduces the considered publish/subscribe model, Section 3 revisits the R-trees characteristics and includes the design and the correctness analyze of our DR-tree overlay. Section 4 discusses related work and concludes the paper.

2 Background and Overview

2.1 System Model

We consider a distributed dynamic system composed of a finite yet unbounded set of processes. The network is described by a weakly connected graph, referred to as *communication graph*. Its nodes represent processes of the system and its edges represent established communication links between processes.

In order to connect to the system, a process executes an underlying connection (join) protocol. A process p is called *active* if there exists at least one process q which is already active in the system and aware of p. The set of *neighbors* of a process p is the set of processes q such that the link (p, q) is up (p and q are aware of each other and can communicate with each other through an underlying communication protocol). The logical graph defined by the neighbor relation is referred to as *overlay*. We assume the system to be subject to frequent and unpredictable changes: processes can join or leave arbitrarily often, and they can fail temporarily (transient faults) or permanently (crash failures).

Messages sent by publishers contain a set of *attributes* with associated values (similar to a dictionary or hash map). Each node in the system has associated a set of *subscriptions* or *content-based filters*. For the sake of simplicity, we initially assume that this set contains a single element. A content-based filter is a conjunction of predicates over the attributes field, i.e., $S = f_1 \wedge \ldots \wedge f_j$, where f_i is defined as a tuple $f_i = (n_i \; op_i \; v_i)$ with n_i the name of the attribute, op_i an operator, and v_i a constant value. The operator op_i can be chosen from a set of basic operators that depends on the attribute type. For example, possible operators for numerical attributes are $\{=, <, >, \leq, \geq\}$. In this work we consider complex filters expressed as the conjunction of two or more range predicates. For example a filter expressed on the attributes a and b may be of the form $(v_i < a < v_j) \wedge (v_k < b < v_l)$. Geometrically, these complex filters define poly-space rectangles. An event specifies a value for each attribute and corresponds geometrically to a point. Without restraining the generality, we illustrate our algorithms on two-dimensional filters corresponding to rectangles in a two-dimensional space. Figure 1 (left) shows a set of sample subscriptions and events defined on two attributes. Note that, if one attribute is undefined, then the corresponding rectangle is unbounded in the associated dimension.

Publish/subscribe systems can take advantage of the property of *subscription containment*, [1] which is defined as follows: subscription S_1 contains another subscription S_2 (written $S_1 \sqsupseteq S_2$) iff any message m that matches S_2 also matches S_1. Conversely, we say that S_2 is contained by S_1 and we write $S_2 \sqsubseteq S_1$. Note that the containment relationship is transitive and defines a partial order. Geometrically, subscription containment corresponds to the enclosure relationships between the poly-space rectangles. Figure 1 (right) shows the containment relationships for the sample subscriptions.

2.2 R-tree Index Structure

R-trees were first introduced in [18]. An R-tree is a height-balanced tree handling objects whose representation can be circumscribed in a poly-space rectangle. Each leaf-node in the tree is an array of pointers to spatial objects. Each entry in a leaf-array is labeled with the index of the pointed object. An R-tree is characterized by the following properties:

[1] The term *covering* is also commonly used in the literature.

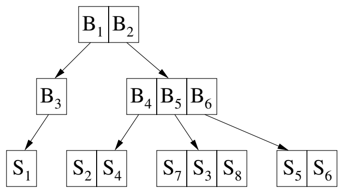

Figure 2. R-tree for the subscriptions of Figure 1.

- Every leaf node contains between m and M entries, each of which corresponds to a spatial object. Every entry is tagged with the smallest rectangle that contains the object.
- Every non-leaf node has between m and M children, except for the root that has at least two children. Each entry in a non-leaf node is tagged with the smallest rectangle—called *minimum bounding rectangle* (MBR)—that spatially contains the rectangles in the child node.
- The height of an R-tree containing N objects is $\lceil log_m(N)\rceil - 1$.
- The worst space utilization for each node except the root is m/M.

In a classical R-tree structure, the actual objects are only stored in the leaves of the tree and interior nodes only maintain MBRs. An R-tree constructed from to the sample subscriptions of Figure 1 is shown in Figure 2 and its spacial representation in Figure 3. Note that all subscriptions are stored in the leaves; the role of interior nodes $B_1, \ldots, B_6$ is to keep track of the bounding rectangles that contains their descendants. In distributed settings, obviously, interior nodes must be managed by specific peers in the system.

2.3 Selective Data Dissemination via R-trees

Consider a message traversing a distributed R-tree structure, where the nodes of the trees are mapped to the subscribers of a publish/subscribe system. An event received by a node will be forwarded to each of its children whose MBR contains the event. Obviously, as the event flows down the trees, it may be received by subscribers that are not interested, i.e., whose filter does not overlap with the message (even though the MBR does). We call such an event a *false positive*. If the filters of all the descendant of a subscriber n are contained within n's filter, then n's MBR will be identical to its filter and n will experience no false positive. It is therefore important to preserve the containment relationships, ideally embedding the containment graph in the R-tree (note that a full embedding is often impossible while at the same time maintaining the R-tree height balanced).

By construction of the MBRs, the R-tree structure does not produce *false negatives* during dissemination, i.e., all the subscribers that have subscribed for an event will receive the event.

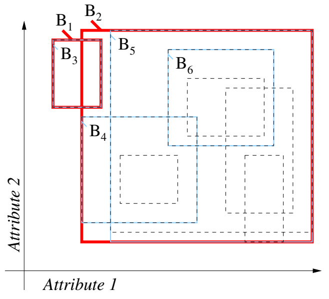

Figure 3. Spatial representation of the R-tree of Figure 2.

3 Self-stabilizing Dynamic R-trees

In this section we extend the R-trees index structures to self-stabilizing peer-to-peer content-based spatial filters. That is, subscribers self-organize in a balanced virtual tree overlay based on the semantic relations between their filters. We refer to the resulting distributed structure as *DR-tree*. In order to simplify the presentation we consider that each filter is a rectangle and can be represented using coordinates in a two dimensional space. This representation well captures the range filters difficult to express in the most popular publish/subscribe systems (e.g., [1, 19, 7, 14]). The extension to complex filters represented with poly-space rectangles is straightforward. The overlay preserves the R-trees index structure features: bounded degree per node and search time logarithmic in the size of the network. Moreover, the proposed overlay copes with nodes dynamicity, as well as memory and counter program corruptions.

Unlike the traditional R-trees, each node in our structure is under the responsibility of a physical process. The DR-tree structure is defined by the logical links between subscribers depending on the relations between their filters. A subscriber responsible for an internal node of the tree filters events for all subscribers in its subtree. In order to maintain the balanced nature of the tree (as will be explained further in the overlay description) a subscriber is present in all the levels of its subtree. More precisely, a subscriber p is recursively its own child in the subtree rooted at p. Therefore, a subscriber has different children sets for each level where it is active. Consequently, a process may have to maintain more than one parent link and children set.

Considering again the sample subscriptions of Figure 1, a possible logical organization of the subscriptions (i.e., processes) in the system is shown in Figure 4. Some subscriptions are both leaf and interior nodes of the DR-tree. The choice of which subscriptions are promoted as interior nodes will be discussed shortly. The logical tree has a single virtual root (subscription S_3) that appears at all level. The

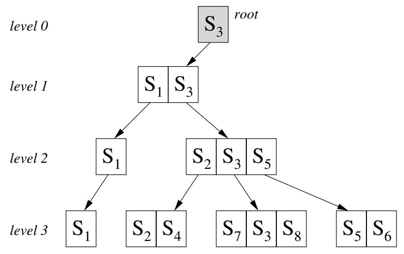

Figure 4. DR-tree for the subscriptions of Figure 1.

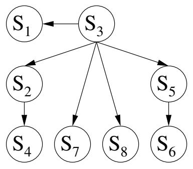

Figure 5. Communication graph for the DR-tree of Figure 4.

physical organization of the subscribers is shown in Figure 5. Each node is neighbor with its children and parent in the logical tree.

Events flow through the tree according to the values of the MBRs of the nodes at each level: an interior node forwards the event to each of its children whose MBR contains the event. An event produced by a node n is disseminated along all subtrees for which n is a root; further, it it propagated upwards the root of the DR-tree and down every sibling subtree encountered on the path to the root. Consider for instance the production of event a (see Figure 1) by the process associated to node S_2 in the DR-tree of Figure 4. The instance of node S_2 at level 2 sends the event down to child S_4, and upward to S_3. Node S_3 checks if the event is contained by the MBR of any of its children at levels 1 and 0 and, as this is not the case, does nothing. Therefore, the event is received only by S_2, S_3, and S_4, thus producing no false positive and necessitating only 2 messages.

3.1 Overlay Organization

As previously mentioned, the DR-tree structure guarantees that no false negative occurs during event dissemination, i.e., every consumer receives the events it is interested in. However, the organization of the subscribers has a strong influence on the routing accuracy and the number of false positives in the system, i.e., the messages erroneously received by consumers. In the worst case, the propagation of an event may degenerate into a broadcast reaching all consumer nodes irrespective of their interests. It is therefore essential to organize the nodes carefully so as to minimize the occurrence of false positives.

A straightforward approach to avoiding false positives is to organize the subscribers according to containment relationships, such that the filter of a node contains the filters of its descendants. Indeed, if an event matches the containee, it *has* to match the container; conversely, if it does not match the container, it cannot match the containee.

A direct mapping of the containment graph to a tree structure [11] is often inadequate. First, it requires a virtual root with as many children as subscriptions that are not contained in any other subscription. Second, depending on the subscription workload, the resulting tree might be heavily unbalanced with a high variance in the degrees of internal nodes.

Another approach consists in building one containment tree per dimension and add a subscription to each tree for which it specifies an attribute filter [3]. This solution tends to produce flat trees with high fan-out and may generate a significant number of false positives.

Instead, we need to organize the subscriptions in the DR-tree structure while preserving existing containment relationships. In particular, we want to preserve the following property:

Property 3.1 (Weak Containment Awareness) *Given two filters S_1 and S_2 with $S_1 \sqsubseteq S_2$, then (the topmost instance of) S_1 is not an ancestor of (the topmost instance of) S_2 in the DR-tree.*

This property guarantees that a containee filter will not be a parent of a container filter, as it would degrade routing accuracy. This property is guaranteed by our root election mechanism, which promotes as parent the node whose MBR has the largest coverage area (see below). Observe that the DR-tree of Figure 4 preserves this property.

In addition, it is desirable to implement a stronger variant of the containment awareness property:

Property 3.2 (Strong Containment Awareness) *Given two filters S_1 and S_2 with $S_1 \sqsubseteq S_2$, then either (the topmost instance of) S_2 is an ancestor or sibling of (the topmost instance of) S_1 in the DR-tree, or there exists S_3 such that $S_1 \sqsubseteq S_3$, $S_2 \not\sqsubseteq S_3$, $S_3 \not\sqsubseteq S_2$, and (the topmost instance of) S_3 is an ancestor or sibling of (the topmost instance of) S_1 in the DR-tree.*

This property ensures that a containee filter is a descendant of its containers. Because of the height-balancing mechanism, it might not be possible to register a containee deep enough in the tree as child of one of its container; in that case, it can be inserted as a sibling of the container. The second clause of the property deals with the case of a filter having two container filters that do not cover each other (remember that the containment relationship is a partial order). Therefore, the containee may become a descendant of either of its container. This case is illustrated in Figure 1, with S_4 being contained in both S_2 and S_3. The DR-tree of Figure 4 preserves the strong containment awareness property

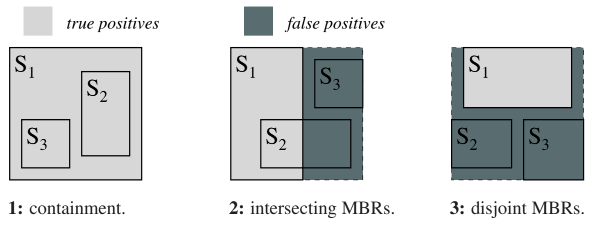

Figure 6. Principle of root election. In all cases, S_1 is the best candidate to be elected as root.

but, in the general case, the order of node insertion and removal may lead to sub-optimal configurations in which this property is occasionally violated.

3.2 Overlay Maintenance

Data Structures. Each process p in the overlay maintains constant non-corruptible data representing its subscription

$$filter_p = S_p = ((\underline{x}_p, \underline{y}_p), (\overline{x}_p, \overline{y}_p))$$

where $\underline{x}_p$ and $\underline{y}_p$ represent the minimal abscissa/ordinate, and $\overline{x}_p$ and $\overline{y}_p$ the maximal abscissa/ordinate of the rectangle that circumscribes the process filters. Additionally, each process p maintains the following variables:

- C_p^l is the set of children of p at level l. This set is periodically updated whenever new processes join or existing children leave the structure at level l. This set is also updated during the reshuffling periods (i.e., periods when nodes of the tree compact their children set).
- $mbr_p^l = ((\underline{x}_p^l, \underline{y}_p^l), (\overline{x}_p^l, \overline{y}_p^l))$ represents the minimum bounding rectangle that includes all the MBRs of all children at l, and is computed as:

  $$((\min_{q \in C_p^l}(\underline{x}_q^{l+1}), \min_{q \in C_p^l}(\underline{y}_q^{l+1})), (\max_{q \in C_p^l}(\overline{x}_q^{l+1}), \max_{q \in C_p^l}(\overline{y}_q^{l+1})))$$

  The MBR of a leaf node is identical to its subscription.

- $parent_p^l$ is the parent of node p for each level l where p is present. The parent of the DR-tree structure root process is the process itself.
- $underloaded_p^l$ is a boolean flag that indicates whether node p is underloaded at level l. The value of this flag is *true* if the size of the children set is less than m, i.e., $|C_p^l| < m$. It turns to false whenever the size of the children set at l is at least m.

Joins. We assume that, at connection time, a subscriber invokes an oracle that accurately provides a subscriber already in the structure. The join process may start from any node, but the odds of finding a good position for the new subscription are best when starting from the root. Therefore, the joining subscriber is recursively redirected upward the tree until it reaches the root.

Upon reception of a connection request, a subscriber p already in the structure adjusts its MBR in order to include the new subscription (if adjustment is needed) and chooses in its children set the child whose MBR needs the less adjustment to encompass the filter of the joining subscriber. In addition, it pushes the request to the chosen child. This downward propagation process stops on the last non-leaf level.

Assume that the join request reaches subscriber q on the last non-leaf level. If the number of alive children of q is less than M, then q adds the new subscriber to its children set. Otherwise, q executes a split-children module that divides its children set in two groups, each having at least m elements (note that m must be chosen such that $M \geq 2m$). The invocation of this module aims at preserving the maximal and the minimal bounds on the nodes degrees. One of the subtree returned by the split procedure stays as the children of the invoking subscriber, p, and this process adjusts its MBR accordingly. The other subtree is pushed backward to p's parent. If the size of the parent children set is less than M, then the parent adds the root of the sub-tree to its children list. Otherwise the parent recursively invokes a split-children module. Note that this process eventually stops with the split of the root, which generates the creation of two subtrees and the election of a new root.

There are three classical methods for splitting a children set, which are supported by our DR-tree structure:

- The *linear* method [18] chooses two children from the overflowing node such that the union of their MBRs waste the most area and place each one in a separate node. The remaining children are assigned to the nodes whose MBR is increased the least by the addition. This method takes linear time.
- The *quadratic* method [18] chooses two children from the overflowing node such that the union of their MBRs would waste the most area if they were in the same node, and place each one in a separate node. The remaining MBRs are examined and the one whose addition maximizes the difference in coverage between the MBRs associated with each node is added to the node whose coverage is minimized by the addition. This method takes quadratic time.
- The *R*-tree* splitting method [5] attempts to reduce not only the coverage, but also the overlap. Instead of just splitting the node when it overflows, it also tries to allocate some entries to a better suited node through reinsertion.

Which node is elected as parent of a subtree has influence on the routing accuracy of the resulting DR-tree, even

```
**Compute_MBR**$(p, l) \equiv$
  $mbr^l_p \leftarrow ((\min_{q \in \mathcal{C}^l_p}(\underline{x}^{l+1}_q), \min_{q \in \mathcal{C}^l_p}(\underline{y}^{l+1}_q)), (\max_{q \in \mathcal{C}^l_p}(\overline{x}^{l+1}_q), \max_{q \in \mathcal{C}^l_p}(\overline{y}^{l+1}_q)))$

**Is_Better_MBR_Cover**$(p, q, l) \equiv$
  **return** $|mbr^{l+1}_q| > |mbr^{l+1}_p|$

**Is_Good_MBR**$(p, l) \equiv$
  **return** $mbr^l_p = ((\min_{q \in \mathcal{C}^l_p}(\underline{x}^{l+1}_q), \min_{q \in \mathcal{C}^l_p}(\underline{y}^{l+1}_q)), (\max_{q \in \mathcal{C}^l_p}(\overline{x}^{l+1}_q), \max_{q \in \mathcal{C}^l_p}(\overline{y}^{l+1}_q)))$

**Adjust_Parent**$(p, q, l) \equiv$
  $parent^l_q \leftarrow parent^l_p$
  **forall** $s \in \mathcal{C}^l_p$ **do** $parent^{l+1}_s \leftarrow q$
  $\mathcal{C}^l_q \leftarrow \mathcal{C}^l_p$
  $Compute\_MBR(q, l)$

**Adjust_Children**$(p, q, l) \equiv$
  $mbr^l_p \leftarrow mbr^l_p \bigcup mbr_q$
  $\mathcal{C}^l_p \leftarrow \mathcal{C}^l_p \bigcup \{q\}$
  $parent^l_q \leftarrow p$
```

Figure 7. Functions used by the Join, Leave, and Repair modules

```
**upon receive** JOIN$(q, l)$ at node $p$
  **if** $\neg$Is_Leaf(Choose_Best_Child($p$, $filter_q$, $l$),$l+1$) **then**
    $mbr^l_p \leftarrow mbr^l_p \bigcup filter_q$
    send JOIN $(q, l+1)$ to Choose_Best_Child($p$, $filter_q$, $l$)
  **else**
    send ADD_CHILD $(q, l)$ to $p$

**upon** receive ADD_CHILD$(q, l)$ at node $p$
  **if** $|\mathcal{C}^l_p| < M$ **then**
    Adjust_Children$(p, q, l)$
    **if** Is_Better_MBR_Cover$(p, q, l)$ **then**
      Adjust_Parent$(p, q, l)$
  **else**
    $(left, right) \leftarrow$ Split_Node$(p, q, l)$
    **if** Is_Root$(left, l)$ **then**
      Create_Root$(left, right)$
    **else**
      send ADD_CHILD $(right, l-1)$ to $parent^l_{left}$
```

Figure 8. Pseudo-code of Join Phase executed at node p

```
**upon receive** LEAVE$(q, l)$ at node $p$
  send CHECK_CHILDREN$(l)$ to $p$
  **if** $q \in C^l_p$ **then**
    $C^l_p \leftarrow C^l_p \setminus \{q\}$
    Compute_MBR$(p, l)$
  send CHECK_PARENT$(l)$ to $p$
  **if** $|C^l_p| < m \wedge \neg$ Is_Root$(p, l)$ **then**
    send CHECK_STRUCTURE$(l-1)$ to $parent^l_p$
```

Figure 9. Pseudo-code of Leave Phase executed at node p

though it does not affect the size of the MBR of the new root. In order to preserve the containment awareness properties and minimize the likeliness for false positives, we elect as root the node whose current MBR is largest, i.e., which provides most coverage over the MBR of the new root. If one filter covers all the others, then it trivially becomes the new root (case 1 in Figure 6): the containment awareness properties is preserved and we have no false positives. If filters intersect or are disjoint, we elect the node with the largest MBR (cases 2 and 3 in Figure 6) in order to minimize the size of the area corresponding to false positives.

The detailed pseudo-code of the join process is shown in Figures 7 and 8.[2]

Dynamic Reorganizations. There are two situations where nodes may dynamically reorganize to improve the accuracy of the underlying DR-tree structure. First, each internal node in the tree periodically checks if it is the best cover for sub-tree. If one of its children provides better coverage (e.g., because its MBR has grown after the insertion of a new node), then the nodes exchange their position. This scenario can occur during join, splitting, and if the tree is corrupted (to be discussed later).

Second, under bias event workloads, it may happen that the organization of the DR-tree (computed statically so as to minimize MBR coverage) may perform poorly because small false positive regions are hit by many events while larger areas see none. To deal with such situations, each node computes its number of false positives, and the number of false positives that each of its children would have experienced if it had been in its place. If the former is higher than the latter, i.e., the child is interested in many events that its parent does not care about, then both nodes exchange their positions.

Controlled Departures. We now describe the repair algorithm executed whenever a subscriber leaves properly the system by sending a leave message to the parent of its topmost instance in the DR-tree. Upon receiving a leave message the parent removes the subscriber from its children set. For simplicity, we rely on the stabilization mechanisms (to be presented shortly) for repairing the subtree rooted at the departing node. Much more efficient variants are possible if the leave module drives the repair process and reconnects whole subtrees. If, due to the removal, the children set drops below the m limit, then the node sends a CHECK_STRUCTURE message to its parent. The algorithm executed upon the reception of this message is presented next. The detailed pseudo-code of the departure process is shown in Figure 9.

3.3 Overlay Stabilization

The overlay stabilization process implements the self-stabilization of the mbr variables and the tree structure, and checks that the overlay respects the DR-tree specification. This verification is performed periodically due to the dynamicity of the environment. That is, at each subscriber in

[2]The implementation of functions *Choose_Best_Child* (select the subtree in which to insert a new node) and *Split_Node* (separate a leaf set into two sets and return both parents) are not shown as they depend on the splitting method and type of structure being used (linear, quadratic, *R**).

the DR-tree, the following events are triggered periodically for each level where the subscriber is active: CHECK_MBR, CHECK_PARENT, CHECK_CHILDREN, CHECK_COVER and CHECK_STRUCTURE.

Correction of the MBR values. In a correct state, the MBR of a leaf node equals its filter while the MBR of a non-leaf node is the smallest rectangle that covers the MBRs of its children. Upon the reception of a CHECK_MBR event each subscriber checks the correctness of its mbr value and repairs it in case of anomaly (see Figure 10).

```
upon receive CHECK_MBR($l$) at node $p$
  if Is_Leaf($p, l$) $\wedge mbr_p^l \neq filter_p$ then
    $mbr_p^l \leftarrow filter_p$
  if $\neg$ Is_Leaf($p, l$) $\wedge \neg$Is_Good_MBR($p, l$) then
    Compute_MBR($p, l$)
```

Figure 10. Repair MBRs at node p

Correction of the DR-tree structure. Transitory faults or uncontrolled departures may have a dramatic impact on the DR-tree structure. Therefore, we reinforce the system by adding modules that deal with the different scenarios of corruption. The DR-tree structure is corrupted if: (a) the variable $parent$ or the children set are corrupted; (b) the child of a node has a filter with better cover than its parent; or (c) the size of the children set drops under the limit m. Each one of these situations is corrected by one of the modules shown in Figures 11, 12, 13 and 14 as explained in the following. Note that the insertion of new nodes in the structure cannot create corruptions of the DR-tree state.

Due to a transient fault the $parent$ variable may have any value. In order to stabilize the DR-tree structure in this respect, each node verifies via module 11 if it is present in the list of children of its parent. If that is not the case, then the node sets itself as parent and initiates a join process.

The module shown in Figure 12 checks the status of the variable "underloaded" which may have been corrupted during execution or may have an incorrect value due to sudden departures. In case of corruption the variable is reset to a correct value. Furthermore, if a node discovers that one of its children has another parent, then it simply discards the child.

```
upon receive CHECK_PARENT($l$) at node $p$
  if $p \notin \mathcal{C}^{l-1}_{parent^l_p}$ then
    $(n, l) \leftarrow$ Get_Contact_Node()
    send JOIN $(p, l)$ to $n$
```

Figure 11. Repair Parent at node p

Due to some modifications in the tree structure, the child of a node may better cover the node sub-tree than the node itself. In that case, the node and the child exchange their roles. Note that the MBR of the new parent must be updated, as well as the MBRs of all ancestor nodes on the path to the root. This correction is performed by the module proposed in Figure 13.

```
upon receive CHECK_CHILDREN($l$) at node $p$
  while $\exists q \in \mathcal{C}_p^l, parent_q^{l+1} \neq p$ do
    $\mathcal{C}_p^l \leftarrow \mathcal{C}_p^l \setminus \{q\}$
    Compute_MBR($p, l$)
  if $|\mathcal{C}_p^l| < m \wedge \neg underloaded_p^l$ then
    $underloaded \leftarrow$ true
  if $|\mathcal{C}_p^l| > m \wedge underloaded_p^l$ then
    $underloaded \leftarrow$ false
```

Figure 12. Repair Children Set at node p

```
upon receive CHECK_COVER($l$) at node $p$
  if $\exists q \in \mathcal{C}_p^l$, Is_Better_MBR_Cover($p, q, l$) then
    Adjust_Parent($p, q, l$)
```

Figure 13. Repair Cover at node p

By specification, each node has to have at least m children. Due to uncontrolled departures, the children set of a particular node may drop below the m limit. In the following this node is refered to as underloaded. In order to avoid the creation of an underloaded tree, each node verifies periodically if it has underloaded children (Figure 14). If such children exist the node starts a compaction process. That is, the subtrees corresponding to underloaded nodes are compacted in a unique subtree and MBRs are recomputed accordingly. If, after the compaction process, a node p still has at least one underloaded child q, then the children of q are dispatched to one of p's unsaturated children (nodes that have less than M children). If that is not possible, a reinsertion message is sent to q's children that couldn't be reinserted, so that they execute again the join process.

3.4 Overlay correctness

In this section we present the correctness of the computed overlay. We first show that join and controlled departure operations do not corrupt the DR-trees structure. Then we show the stabilization of the structure once the local variables are corrupted by a transient fault or disconnection of subscribers without notification. Proofs can be found in a companion technical report [6].

Definition 3.1 *The DR-tree is in a legal state iff the following conditions are verified :*

- *each non-root and non-leaf node in the tree*[3] *has at most M and at least m children;*
- *for each process p in the overlay, the parent and children variables are coherent:*
  - *if p is the parent of node q at level l then q belongs to the children set of p at $l-1$;*
  - *if q is the child of p at level l then q has parent variable set to p for level $l+1$;*
- *for each node p at level l there is no child q such that q offers a better cover for the subtree of p at l;*

[3]Note that a process can be responsible for several nodes in the tree.

```
Search_Compaction_Candidate $(p, q, l) \equiv$
  send CHECK_CHILDREN$(l+1)$ and CHECK_MBR$(l+1)$ to all $k \in \mathcal{C}_p^l$
  $cand \leftarrow \varnothing$
  while $\exists t \in \mathcal{C}_p^l \setminus \{q\}, |\mathcal{C}_t^{l+1} \bigcup \mathcal{C}_q^{l+1}| \leq M$ do
    $cand \leftarrow cand \bigcup \{t\}$
  if $cand = \varnothing$
  then return $\perp$
  else return $t \in cand, |mbr_t^{l+1} - mbr_q^{l+1}| = \min_{s \in cand} |mbr_s^{l+1} - mbr_q^{l+1}|$

Best_Set_Cover$(set, s, t, l) \equiv$
  $mbr\_set \leftarrow ((\min_{q \in set}(\underline{x}_q^l), \min_{q \in set}(\underline{y}_q^l)), (\max_{q \in set}(\overline{y}_q^l), \max_{q \in set}(\overline{y}_q^l)))$
  if $|mbr\_set - filter_s| < |mbr\_set - filter_t|$
  then return $s$
  else return $t$

Elect_Leader$(s, t, l) \equiv$
  send CHECK_CHILDREN$(l+1)$ and CHECK_MBR$(l+1)$ to all $k \in \mathcal{C}_s^l \bigcup \mathcal{C}_t^l$
  return Best_Set_Cover$(\mathcal{C}_s^l \bigcup \mathcal{C}_t^l, s, t, l)$

Merge_Children$(s, t, l) \equiv$
  $\mathcal{C}_s^l \leftarrow \mathcal{C}_s^l \bigcup \mathcal{C}_t^l$
  $parent_t^l \leftarrow \perp$
  forall $k \in \mathcal{C}_t^l$ do $parent_k^{l+1} \leftarrow s$
  Compute_MBR$(s, l)$

Compact$(s, t, l) \equiv$
  if Elect_Leader$(s, t, l) = s$ then
    Merge_Children$(s, t, l)$
  else
    Merge_Children$(t, s, l)$

upon receive CHECK_STRUCTURE$(l)$ at node $p$
  send CHECK_CHILDREN$(l+1)$ to all $q \in \mathcal{C}_p^l$
  while $\exists q \in \mathcal{C}_p^l, underloaded_q^{l+1}$ do
    $cand \leftarrow$ Search_Compaction_Candidate$(p, q, l)$
    if $cand = \perp$
    then send INITIATE_NEW_CONNECTION$(l+1)$ to $q$
    else Compact$(q, cand, l+1)$

upon receive INITIATE_NEW_CONNECTION$(l)$ at node $p$
  if $\neg$ Is_Leaf$(p, l)$ then
    send INITIATE_NEW_CONNECTION to all $q \in \mathcal{C}_p^l$
  else
    $(n, l) \leftarrow$ Get_Contact_Node()
    send JOIN $(p, l)$ to $n$
```

Figure 14. Repair DR-tree structure at node p

- *the MBR value of each non-leaf node p at level l is the union of the MBR values of its children at level $l+1$.*

Definition 3.2 (legitimate configuration) *Let S be the system executing the algorithms described in section 3.2. The system is in a legitimate configuration iff the virtual structure defined by the $parent$ variables and the children sets is a legal DR-tree.*

Lemma 3.1 *In a legitimate configuration the height of the DR-tree is $O(\log_m(N))$ while the memory complexity for the structure maintenance is $O(M \log^2(N)/\log(m))$.*

Lemma 3.2 (stabilization after joins) *Let c be a legitimate configuration and let p be a subscriber joining the system in c. Let c' be the new configuration of the system after p executes the join operation (Figure 8). c' is a legitimate configuration and is reached in $O(\log_m(N))$ steps.*

Lemma 3.3 (stabilization after compaction) *The system executing a compaction action reaches a legitimate configuration in $O(N \log_m(N))$ steps.*

Lemma 3.4 (stabilization after controlled leaves) *Let c be a legitimate configuration and let p be a subscriber leaving the system via a controlled departure in c. Let c' be the new configuration of the system after p executes the controlled departure operation (Figure 9). c' is a legitimate configuration and is reached in $O(N \log_m(N))$ steps.*

Lemma 3.5 (stabilization after uncontrolled leaves) *Let c a legitimate configuration and let p be a node leaving the system via an ucontrolled departure (failure) in c. The system reaches a legitimate configuration c' in a finite number of steps. It stabilizes in $O(N \log_m(N))$.*

Lemma 3.6 (stabilization after memory corruption) *Let c be an initial arbitrary configuration of the system. The system reaches a legitimate configuration c' in a finite number of steps.*

The DR-tree integrates modules that repair the overlay as soon as a corruption is detected. The recover process is totally dependent on the value of the "timeout" and the stabilization time of the structure. As shown in the previous lemmas, for most of the faults the recovery time is $O(N \log_m(N))$ (note again that the recovery time can be significantly reduced by reconnecting whole subtrees after a failure). However, in environments prone to high churn the structure may never be able to self-repair. Therefore, it is interesting to study the limits of our overlay. The following lemma computes the bound on the expected time the DR-tree gets disconnected due to frequent departures. We recall that joins have no impact on the overlay connectivity.

Lemma 3.7 (DR-tree churn resistance) *Let Δ be an interval of time during which no stabilization operation is triggered and let λ be the rate of departures.*[4] *The expected time before the DR-tree disconnects is $\frac{\Delta}{N} e^{\frac{(N-\Delta\lambda)^2}{4\Delta\lambda}}$.*

4 Discussions and Conclusion

Publish/Subscribe systems have been studied extensively in the last few years starting with pioneering work of [1] and several surveys (e.g., [16]) report the contributions in this area.

Implementing Publish/Subscribe in dynamic environments where continuous service has to be guaranteed despite high churn (frequent connections/disconnections) remains an important challenge. Some recent systems have proposed to exploit the features of DHT-based peer-to-peer overlays [8, 21, 17]. The main goal is to offer guarantees with respect to the publish/subscribe latency and the resilience to high churn. The first objective was successfully achieved since DHT's offer logarithmic latencies. However, it was shown [4] that these overlays have limited scalability and low resistance to churn. Additionally, the

[4] We consider arrivals and departures modeled by a Poisson distribution.

mapping of complex filters to uni-dimensional name spaces results in poor performance. Consequently, several designs of DHT-free peer-to-peer publish/subscribe systems were proposed [12, 11, 3, 20]. The main advantage of these approaches is their scalability, although most of them suffer from two problems: the loss of accuracy (apparition of false negatives or false positives) and their poor latency for unfriendly scenarios of connection/disconnection. Note that none of the previously mentioned systems is self-stabilizing.

In this paper we designed a self-stabilizing overlay, DR-tree, that combines the best of both worlds. Our overlay deals with complex (multi-dimensional) filters via a data-structure similar to R-trees. DR-tree is a distributed and fault-tolerant implementation of R-trees structure, fully adapted to embed a publish/subscribe system with complex filters, which copes with the dynamicity of the system and memory corruption. The overlay is designed such as it eradicates the false negatives and drastically drops the false positives (our experiments show that the false positive rate is in the order of $2 - 3\%$ with most workloads [6]). Moreover, as its DHT-based counterparts, it provides logarithmic guaranties for the publish/subscribe operations using only polylogarithmic memory per node. We proved the correctness of our overlay under static and dynamic assumptions. Note that DR-trees generalize P-trees [13], which are the dynamic version of B+-trees.[5]

References

[1] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, and T.D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC 1999)*, pages 53–61, 1999.

[2] M. Altinel and M.J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Databases (VLDB 2000)*, pages 53–64, 2000.

[3] E. Anceaume, A.K. Datta, M. Gradinariu, G. Simon, and A. Virgillito. A semantic overlay for self-* peer-to-peer publish subscribe. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.

[4] R. Baldoni, S. Bonomi, A. Rippa, L. Querzoni, S.T. Piergiovanni, and A. Virgillito. Evaluation of unstructured overlay maintenance protocols under churn. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS 2006)*, 2006.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[6] S. Bianchi, A.K. Datta, P. Felber, and M. Gradinariu. Self-stabilizing peer-to-peer spatial filters. Technical report, LIP6, Universite Paris 6, 2007.

[7] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[5]B+-trees are ancestors of R-trees designed to handle ranges and inequalities.

[8] M. Castro, P. Druschel, A.M. Kermarrec, and A. Rowston. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 20(8):1489–1499, 2002.

[9] C.-Y. Chan, W. Fan, P. Felber, M. Garofalakis, and R. Rastogi. Tree pattern aggregation for scalable XML data dissemination. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002)*, 2002.

[10] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal, Special Issue on XML*, 1(4):354–379, 2002.

[11] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2005)*, 2005.

[12] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola. Epidemic algorithms for reliable content-based publish/subscribe: An evaluation. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, 2004.

[13] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-Trees. In *Proceedings of the the 7th International Workshop on the Web and Databases (WebDB 2004)*, pages 25–30, 2004.

[14] G. Cugola, E. Di Nitto, and A. Fugetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, 2001.

[15] Y. Diao, P. Fischer, M. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, 2002.

[16] P.Th. Eugster, P. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[17] A. Gupta, O.D. Sahin, D. Agrawal, and A. El Abbadi. Meghdoot: Content-based publish:subscribe over P2P networks. In *Proceedings of the ACM/IFIP/USENIX 5th International Middleware Conference (Middleware 2004)*, 2004.

[18] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[19] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with Elvin4. In *Proceedings of the AUUG2K*, 2000.

[20] S. Voulgaris, E. Rivire, A. Kermarrec, and M. van Steen. Sub-2-Sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[21] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of the International Workshop on Network and OS Support for Digital Audio and Video*, 2001.