Dynamically Reconfigurable Filtering Architectures

Mathieu Valero, Luciana Arantes, Maria Gradinariu, Pierre Sens

LIP6 - University of Paris 6 - INRIA

Abstract. Distributed R-trees (DR-trees) are appealing infrastructures for implementing range queries, content based filtering or k-NN structures since they inherit the features of R-trees such as logarithmic height, bounded number of neighbors and balanced shape. Interestingly, the mapping between the DR-tree logical nodes and the physical nodes has not yet received sufficient attention. In previous works, this mapping was naively defined either by the order physical nodes join/leave the system or by their semantics. Therefore, an important gap in terms of load and latency can be observed while comparing the theoretical work and the simulation/experimental results. This gap is partially due to the placement of virtual nodes. A naive placement that totally ignores the heterogeneity of the network may generate an unbalanced load of the physical system. In order to improve the overall system performances, this paper proposes mechanisms for placement and dynamic migration of virtual nodes that balances the load of the network *without* modifying the DR-tree virtual structure. That is, we reduce the gap between the theoretical results and the practical ones by injecting (at the middleware level) placement and migration strategies for virtual nodes that directly exploit the physical characteristics of the network. Extensive simulation results show that significant performance gain can be obtained with our mechanisms. Moreover, due to its generality, our approach can be easily extended to other overlays or P2P applications (e.g. multi-layer overlays or efficient P2P streaming).

1 Introduction

From the very beginning of the theoretical study of P2P systems, one of the topics that received a tremendous attention is the way these systems are mapped to the real (physical) network. Even early DHT-based systems such as Pastry [19] or CAN [18] included in their design the notion of geographical locality. Later, middlewares designed on top of DHT-based or DHT-free P2P systems exploit various degrees of similarity between peers following criteria such as the geographical vicinity or the semantic of their interests. One of the most relevant example in that sense are content-based publish/subscribe systems. These communication primitives completely decouple the source of events (*a.k.a. publishers*) from their users (*a.k.a. subscribers*). Their efficient implementations in P2P settings are optimized with respect to a broad class of metrics such as latency or load balancing. To this end the similarity between different subscribers is fully exploited and various logical infrastructures have been recently proposed. Most exploited architectures are tree-based due to their innate adaptability to easy filter. In the design of tree-based publish/subscribe there is a trade-off between the maintenance of an optimized tree infrastructure following various criteria (e.g. bounded degree, max/min number of internal nodes or leaves) and the placement of logical nodes on physical machines in order to optimize the overall system load or latency.

Our paper follows this research direction by addressing the placement of virtual nodes in tree-filtering architectures. In particular, we address the case of DR-trees overlays where the quality of service of the overlay greatly depends on the way virtual nodes are mapped on the physical nodes. DR-trees [2] are a distributed P2P version of R-trees [11] which are used to handle objects with a poly-rectangular representation. DR-trees are P2P overlays with a bounded degree and search/diffusion time logarithmic in size of the network. By their natural construction they are adapted to represent subscriptions with rectangle shape in content based systems. The efficient construction of a DR-tree overlay raises several problems. In particular, it should be noted that the design of DR-trees may bring a single physical machine to be responsible for several virtual nodes. Therefore, a wrong choice in the placement of virtual nodes may have a huge impact on overall system performances; in terms of latency, bandwidth etc...

Our contribution tends to address this issue. We investigate the problem of placement and migration of virtual nodes. We define valid migrations (with respect to the logical topology of the DR-trees) and, using some techniques borrowed from transactional systems, we propose strategies for solving conflicts generated by concurrent migrations.

2 Related work

The placement of virtual nodes has been evoked in several works addressing different ad hoc issues. In publish-subscribe systems such as Meghdoot [10], Mirinae [7], Rebeca [20], SCRIBE [6] or Sub-2-Sub [17], virtual nodes correspond to subscriptions and they are mapped on the physical node that created them. In BATON [13] and VBI [14], two AVL based frameworks, virtual nodes are divided in two categories: leaves and internal nodes. The former are used for storage while the latter are used for routing. A structural property of AVL ensures that there there are roughly as much leaves as internal nodes: each physical node holds one leaf and one internal node. In a DR-tree, due to its degree ($m : M$, with $m > 1$), there are fewer internal nodes than leaves; each physical node holds exactly one leaf and may hold one or more internal nodes.

Many publish/subscribe are based on multicast trees where network characteristics are exploited to build efficient multicast structures in terms of latency and/or bandwidth ([3, 1, 8, 5]). Our approach does not require any extra overlays or structures. Network characteristics are taken into account through a mechanism of virtual node migrations; this mechanism was not conceived for a particular evaluation metric. Metrics can be defined independently making our work more general.

Brushwood [4] is a kd-tree based overlay targeting locality preservation and load balancing. Each physical node holds one virtual node which corresponds to a k-dimensional hyperplane. When a physical node joins the network, it is routed to a physical node. If the joined physical node is overloaded, it will split its hyperplane and delegate half of it to the new node. In addition, Brushwood provides a sporadical load evaluation mechanism: overloaded physical nodes may force underloaded ones to reinsert themself and thus benefit from the join mechanism.

Chordal graph [15] is a range queriable overlay. Efficiency of queries is measured in terms of distance (that could be expressed in terms of latency, bandwidth etc. . .) between physical nodes. Similarly to SkipNet [12, 9], each physical node belongs to different rings. For each ring, the physical node holds a range of values which can be dynamically resized to balance load. During joins, physical nodes are routed according to their relative distance. Conceptually, hyperplanes and ranges are virtual nodes. While Chordal graph [15] and Brushwood [4] modify virtual nodes during hyperplane splitting or range scaling while our approach neither modifies virtual nodes nor the DR-tree structure.

3 Background

In this section we recall some generic definitions and the main characteristics of the DR-trees [2] overlay. Moreover, we discuss the main issues related to the virtual nodes distribution.

3.1 Distributed R-trees

R-trees were first introduced in [11] as 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. A R-tree is characterized by the following structural properties:

– Every non-leaf node has a maximum of M and at least m entries where $m \leq M/2$, except for the root.
– The minimum number of entries in the root node is two, unless it is a leaf node. In this case, it may contain zero or one entry.
– All the leaf nodes are at the same level.

Distributed R-trees (DR-trees) introduced in [2] extend the R-tree index structures where peers are self-organized in a balanced virtual tree overlay based on semantic relations The structure 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 the dynamism of the system.

Physical machines connected to the system wille be further referred as *p-nodes* (shortcut for physical nodes). A DR-tree is a virtual structure distributed over a set of p-nodes. In the following, terms related to DR-tree will be prefixed with “v-”. Thus, DR-trees nodes will be called *v-nodes* (shortcut for virtual nodes). The root of the DR-tree is called the *v-root* while the leaves of the DR-tree are called *v-leaves*. Except the v-root, each v-node n has a v-father (*v-father(n)*), and, if it is not a v-leaf, some v-children (*v-children(n)*). These nodes are denoted $v-neighbors$ of n.

The physical interaction graph defined by the mapping of a DR-tree to $p-nodes$ of the system is a communication graph where there is a $p-edge$ (p,q), $p \neq q$, in the physical interaction graph if: there is a v-edge (s,t) in the DR-tree, p is the p-node holding v-node s, and q is the p-node holding v-node t.

Figure 1 shows a representation of a DR-tree composed of v-nodes $\{n0,\ldots,n12\}$ mapped on p-nodes $\{p1,\ldots,p9\}$. Dashed boxes represent nodes distribution. There is a p-edge $(p1,p5)$ in the interaction graph because there is a v-edge $(n0,n6)$ in the DR-tree, $p1$ is the p-node holding v-node $n0$, and $p5$ is the p-node holding v-node $n6$.

The key points in the construction of a DR-Tree are the join/leave procedures. When a p-node joins the system, it creates a v-leaf. Then the p-node contacts another p-node to insert its v-leaf in the existing DR-tree. During this insertion, some v-nodes may split and then Algorithm 1 is executed.

Algorithm 1 *void* onSplit(n:VNode)

```
1: if $n.isVRoot()$ then
2:     $newVRoot = n.createVNode()$        ▷ $newVRoot$ is held by the same p-node than $n$
3:     $n.v-father = newVRoot$
4: end if
5: $m = selectChildIn(n.v-children)$
6: $newVNode = m.createVNode()$        ▷ $newVNode$ is held by the same p-node than $m$
7: $n.v-children, newVNode.v-children = divide(n.v-children)$
8: $newVNode.v-father = n.v-father$
```

Distribution invariants: The following two properties are invariant in the implementation of DR-tree proposed in [2]:

- $Inv1$: each p-node holds exactly one v-leaf;
- $Inv2$: if p-node p holds v-node n, either n is a v-leaf or p holds exactly one v-children of n.

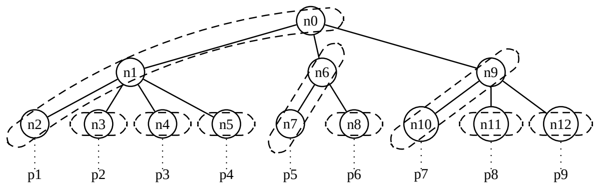
We denote the top and $bottom$ v-node of a p-node the v-node which is at the top and bottom of the chain of v-nodes kept by the p-node respectively. The above invariants ensure that the communication graph is a tree:

- The p-root is the p-node holding the v-root;
- A p-node p is the p-father of the p-node q (*p-father(q)*) if p holds the v-father of the v-node at the top of the chain of v-nodes held by q.

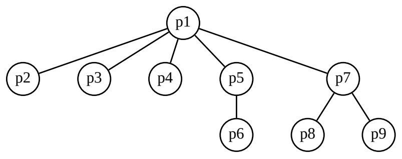
For instance in Figure 1a, *p-father(p5) = p-father(p7) = p1*. The above distribution invariants also guarantee that p-nodes have a bounded number of p-neighbors. In a system with N p-nodes and a DR-tree with degree $m-M$, the DR-tree height is $log_m(N)$; the p-root holds $log_m(N)$ v-nodes. Since each v-node has up to M v-neighbors, the p-root may have up to $M * log_m(N)$ p-neighbors.

3.2 Virtual Nodes Distribution

A DR-tree is a logical structure distributed across a set of physical machines. That is, each v-node is assigned to a p-node of the system. Figures 1 and 2 show the same DR-tree differently distributed over the same set of physical nodes $\{p1,\ldots,p9\}$. The

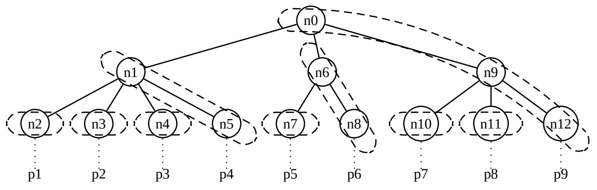


(a) A distribution of a given R-tree

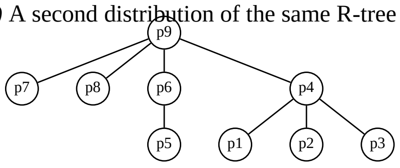


(b) Corresponding physical interaction graph

Fig. 1: A distribution of a given DR-tree and its corresponding physical interaction graph



(a) A second distribution of the same R-tree



(b) Corresponding physical interaction graph

Fig. 2: A second distribution of the same DR-tree leading to a different physical interaction graph

distribution of DR-tree nodes determines the communication interactions between the physical nodes and thus has a strong impact on system performances.

In [2] the distribution of the DR-tree depends both on the join order of machines and on the implementation of the join/split procedures (Algorithm 1). This approach has two main drawbacks. Firstly, the characteristics of the physical machines are not taken into account in the distribution of the DR-tree nodes. Secondly, this distribution is static. Virtual nodes are placed at their creation or during join/split operations. Their location is not changed even if system performances degrade. The first point is problematic in heterogeneous networks, where system performances are highly related to the distribution of the DR-tree root (and its "close" neighborhood). For example, if we evaluate the quality of the system in terms of bandwidth, if $p1$ has a bad one and $p9$ a good one, the mapping proposed in Figure 2a is better than the one proposed in Figure 1a. Static distribution is problematic if the quality of the communications in the physical interactions graph evolves over time. Therefore, virtual nodes that are close to the root of the DR-tree and that are placed on the best possible machines at a given time may induce poor system performances if their guest machines become less performant.

To address these issues, we propose a mechanism for dynamic migration of DR-tree nodes over the physical network. It allows to dynamically modify the DR-tree distribution (without any modification of the logical structure) in order to match the performance changes of the physical system. Our migration mechanism uses feed-backs from some cost functions (e.g. load of nodes) based on the physical interaction graph and also exploits the virtual relations defined by the DR-tree logical structure.

4 Migration mechanism

In this section we analyze the migration of a v-nodes while the DR-tree overlay evolves over time. We start by explaining which v-nodes are candidate to migrate and which destination p-nodes can accept the former with respect to distribution invariants described in section 3.1. Then, we present our migration protocol and discuss when it is triggered.

In the following, we denote $p \xrightarrow{n} q$ the migration of the v-node n from p-node p to p-node q.

4.1 Migration policy

Our migration policy prevents migrations that would violate the two invariants of DR-tree distribution. Note that the invariants can be violated by the wrong choice of both the v-node to migrate and the destination p-node where the v-node will be placed.

The p-node p can migrate its v-node n provided that the distribution invariants will still hold if p no longer keeps n, i.e., if the migration of n takes place.

Following migrations of the v-node n would violate one of the two invariants:

– if p holds exactly one v-node n: if p migrates n, p would no longer hold exactly one leaf ($Inv1$);
– if p holds at least two v-nodes: if p migrates its $bottom$ v-node, p would no longer hold exactly one leaf ($Inv1$); if p migrates a v-node n which is neither its $bottom$ nor its top v-node, p would no longer hold exactly one child of v-father(n) ($Inv2$).

However, if p decides to migrate its top v-node n, the distribution invariants continue to be verified if the top v-node of the destination p-node is the v-child of n. Then, we define a *migrable* v-node as follows:

Definition 1. *A v-node n of p-node p is migrable if p holds at least two v-nodes and n is the* top *v-node of p.*

For instance, in Figure 1a, only $n0$, $n6$, and $n9$ are *migrable*.

Let's now discuss how to chose a destination p-node, denoted a *valid* destination, which ensures that if the migration of n happens the two invariants will still be verified.

Consider that n is a *migrable* v-node of p and let q be a candidate destination p-node to receive n. Following cases would violate one of the two invariants:

– if q holds no v-neighbor of n: n would not be a v-leaf of q, and q would hold no v-children(n) ($Inv2$);

– if q holds v-father of n: n would not be a v-leaf of q, and q would hold two v-children(n) ($Inv2$).

However, if q holds $m \in$ v-children(n) and if n was migrated to q, then n would become q's top v-node. Therefore, we define valid destination of a *migrable* v-node n as follows:

Definition 2. *A p-node q is a valid destination for v-node n held by p, if q holds a v-children of n.*

In Figure 1a, $p1$ can migrate $n0$. Valid destinations are p-nodes which hold a v-children of $n0$: $p5$ and $p7$.

Since the degree of DR-tree is m-M (with $m \geq 2$), the p-root may chose between 1 and M-1 migrations while a p-node holding more than one v-node may chose between m-1 and M-1 migrations.

4.2 Migration conflict solver

Our migration policy guarantees a "local" coherency of migrations. However, two concurrent migrations could lead to invalid configurations. For instance, in Figure 1a, $p1$ can decide to execute $p1 \xrightarrow{n0} p5$ while $p5$ concurrently decides to execute $p5 \xrightarrow{n6} p6$. These two migrations are possible according to our migration policy. On the other hand, executing them concurrently will lead to a configuration where the set of v-nodes held by $p5$ is $\{n0, n7\}$, which violates $Inv2$: $n0$ is not a v-leaf but $p5$ holds no v-children(n0).

When a v-node n is migrated from p-node p to p-node q, p and q are obviously concerned in the migration protocol since they exchange some information for ensuring the migration policy (q can accept or not the migration). In order to avoid incoherent configurations due to concurrent migrations as the one described above, p-nodes holding some v-neighbors of n must also be involved in the migration protocol. Therefore, the principle of our migration protocol is that a p-node p that wants to perform $m = p \xrightarrow{n} q$ should ask permission for it to p-nodes that could concurrently execute some migration conflicting with m. The protocol is basically a distributed transaction where the destination p-node and all p-nodes involved in possible concurrent migrations must give their agreement in order to commit the migration; otherwise it is aborted.

Defining which p-nodes could execute a conflicting migration with $p \xrightarrow{n} q$ is in close relation with our migration policy. As the latter restricts migration, a p-node can only receive migration requests from its p-father. Therefore, migrations that could conflict with $p \xrightarrow{n} q$ are:

– $p - father(p) \xrightarrow{v-father(n)} p$
– $q \xrightarrow{m \in v-children(n)} r$ (where r holds a v-children of q)

There is some locality in potential conflicts: when p-node p tries to perform a migration, concurrent conflicts may happen with its p-father or with nodes having p as p-father.

In order to prevent the concurrent execution of conflicting migrations, our protocol adopts the following strategy: whenever p wants to execute $p \xrightarrow{n} q$, the latter is performed if p is the p-root; otherwise p must have the permission of its p-father before performing $p \xrightarrow{n} q$.

Figure 3 illustrates our migration protocol for $p \xrightarrow{n} q$ which is in fact a best-effort protocol. p is a non root p-node.

try($p \xrightarrow{n} q$**)** is invoked by the migration planner process (described bellow) when the latter wishes to migrate the v-node n from p-node p to the p-node q.

The migration request will be dropped (DROP CASE) if either p is already executing the migration protocol due to a previous v-node migration request or the $p \xrightarrow{n} q$ lasts too long (to avoid disturbing too much the application that run on top of it). Dropping the request will prevent further conflicts. Notice that migrations are expected to be sporadic and not very frequent for a given p-node. Thus, the drop of a migration request will be quite rare.

If $p \xrightarrow{n} q$ is in conflict with another concurrent migration that p has already allowed, the migration protocol is aborted (ABORT CASE 1). In other words, if p has given its permission to $q \xrightarrow{m \in v-children(n)} r$, p must abort $p \xrightarrow{n} q$ in order to avoid the concurrent executions of these two conflicting migrations. On the other hand, if there is no conflict, p invokes **permission(**$p \xrightarrow{n} q$**)** asking its p-father if the latter is not trying to perform any migration that would conflict with $p \xrightarrow{n} q$. If it is the case, p will receive a negative answer (ABORT CASE 2); otherwise, p-father(p) adds $p \xrightarrow{n} q$ to its set of granted migrations and returns a positive answer to p which means that it allows p to execute the migration for which it asked permission. Remark that ABORT CASE 2 happens when p-node attempts to execute $p \xrightarrow{n} q$ but its p-father does not gives it permission for such a migration, i.e., p-father(p) is concurrently executing $p - father(p) \xrightarrow{v-father(n)} p$.

Upon receiving its p-father's permission, p invokes **execute(**$p \xrightarrow{n} q$**)** for actually performing the migration $p \xrightarrow{n} q$. If some error happens during $p \xrightarrow{n} q$ (e.g. technical issues, timeouts, etc.), the migration protocol aborts (ABORT CASE 3). Otherwise, the migration is accomplished.

Finally, **complete(**$p \xrightarrow{n} q$**)** is invoked by p in order to inform its p-father that the migration $p \xrightarrow{n} q$, which it has allowed, was correctly performed. p-father(p) then removes $p \xrightarrow{n} q$ from its set of granted migrations.

4.3 Migration planner

Migration planner defines the exact time the migration should be executed, i.e., when the migration protocol must be invoked. Such an invocation can either take place periodically or whenever there is an event that changes the logical structure of the DR-Tree. Furthermore, an evaluation of the added value of the migration will have on the system must be taken into account in both cases.

In the first case, the migration planner can be implemented as a separated process. Some cost functions (e.g., load of nodes) may be periodically evaluated and when a given value is reached, the planner process is woken up in order to invoke the migration

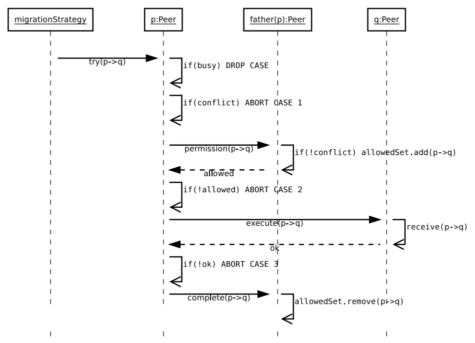


Fig. 3: A straightforward execution of the migration of n from p to q

conflict solver. In the second case, the invocation of the migration conflict solver is triggered in reaction to some event that changes the DR-Tree. For instance, when a v-node split occurs, the migration planner should call the migration conflict solver protocol in order to try to map the new v-nodes to the most suitable p-nodes. This approach can also be exploited for the initial placement of the DR-tree.

The migration planner will invoke the migration provided this one improves the performance of the system. Thus, a cost function is evaluated periodically or whenever a DR-Tree event might induce a migration. If this function signals that a proposed migration will degrade the system, the migration will not take place.

A cost function can concern one or more different metrics (e.g., message traffic, capacity of machines, etc.) and $p \xrightarrow{n} q$ involves p-nodes holding v-neighbors of n, as described in section 4.1. The *cost* function called by p is thus evaluated based on the information about the cost of p's p-neighbor. p can dynamically update this information by spoofing or sporadically evaluating the network traffic. An example of *cost* function is presented in Section 6.

The lack of global knowledge of a p-node cost function (e.g. it just exchanges information with its neighbors at the physical interaction graph) may limit the effectiveness of our migration mechanism, i.e., some performance enhanced configurations might not be reachable. For instance, let suppose that in the configuration of Figure 3.2, $p2$ has the best bandwidth; it should hold v-nodes of higher levels of the DR-tree since the latter are usually more loaded nodes than v-leaves. However, if, according to $p1$'s cost function, both $p1 \xrightarrow{n0} p5$ and $p1 \xrightarrow{n0} p7$ would degrade the bandwidth of the communication graph, $p1$ does not migrate $n0$. Therefore, while $p1$ holds $n0$, $p1 \xrightarrow{n1} p2$ is forbidden since it violates the distribution invariants. The local view of cost functions will not allow the migration of v-nodes to $p2$, even if such migrations would would enhance the bandwidth of the communication graph.

5 Fault tolerance

This section investigates different strategies to preserve a DR-tree structure when some p-node fails. In the classical DR-tree implementation proposed in [2] when a p-node fails all its subtrees are reinserted in the non-faulty structure (p-node by p-node) in order to guarantee the DR-tree invariants.

In the following we study an alternative strategy that exploits internal v-nodes replication.

5.1 Replication pattern

We propose a simple pattern of internal v-nodes replication:

- the p-root holds no replica
- each p-node holds a replica of the v-father of its top v-node

Each non leaf v-node is replicated on each p-node holding one of its v-children. For a DR-tree of degree m-M ($m \geq 2$), it ensures that the v-root is replicated on 1 to M-1 p-nodes and that each internal v-node is replicated on m-1 to M-1 p-nodes.

In Figure 1a, DR-tree has four non leaf v-nodes; $n0$, $n1$, $n6$ and $n9$. The following table shows on which p-nodes they are replicated:

internal v-node	$n0$	$n1$	$n6$	$n9$
replicated on p-nodes	$p5$, $p7$	$p2$, $p3$, $p4$	$p6$	$p8$, $p9$

When a p-node p fails each p-node holding the leftmost replica of a v-node held by p replaces that v-node. The restoration of a given v-node n concerns p-nodes holding a replica of n: the $m-1$ to $M-1$ p-node holding v-children of n. Moreover, the distribution invariants ensure that with N p-nodes, no p-nodes holds more than $\lfloor log_m(N) \rfloor$ v-nodes.

In Figure 1a, if $p1$ fails, two internal v-nodes have to be restored; $n0$ and $n1$. The former concerns $p5$ and $p7$ while the latter concerns $p2$ $p3$ and $p4$.

5.2 Replication cost

Replicas are created during joins and modified during joins and splits. When a v-node is modified its holder should notify the p-nodes which hold its replicas.

In the sequel we'll consider a DR-tree with N participants, a degree of $m : M$ and that updating one replica costs one message.

Figure 4 illustrates what happens at the R-tree level when a v-node n splits. Basically n creates a new v-node n' and delegates half of its childs to n'. n' is added to f childs. If that way f gets more than M childs, it splits according the same algorithm.

To evaluate the cost of replicas update we have to evaluate how many v-nodes are modified during a split. Figure 4 shows that when a v-node n splits, half of its childs and its father are also modified. Each split modifies $m+2$ v-nodes. As each v-node has $M-1$ replicas and that each update costs one message we have:

$$split\_cost = (m+2) * (M-1)$$

A p-node joining the system triggers between 0 and $\lfloor log_m(N) \rfloor$ splits. Its v-leaf is added to the v-children of another v-node that we will call in the sequel the joined v-node. The joined v-node has:

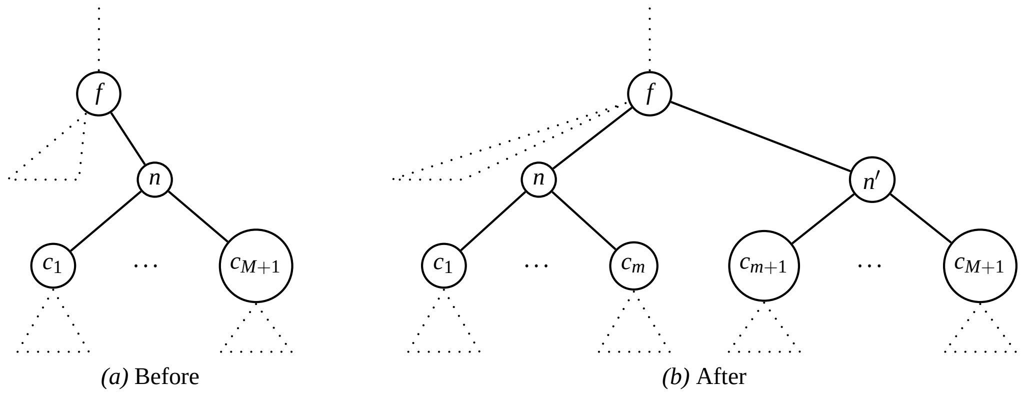


Fig. 4: R-tree split of v-node *n*

- between m and M v-children
- $\lceil log_m(N) - 1 \rceil$ v-ancestors
- between $m-1$ and $M-1$ replicas

In the following calculation we will define an upper bound on the number of updates considering that each v-node has $M-1$ replicas.

A v-node may have m to M v-children and thus has $M-m+1$ possible v-children quantities. It splits only when it has exactly M v-children. The probability p for a v-node to split is:

$$p = \frac{1}{M-m+1}$$

The probability for a p-node to trigger k splits is the probability p_k that the joined v-node and its $k-1$ first v-ancestors have exactly M v-children.

$$p_k = p^k * (1-p)$$

The average cost of replicas updates when a p-node joins a DR-tree is thus:

$$replication\_cost = \underbrace{p_0 * (M-1)}_{no\ split} + \underbrace{\sum_{k=1}^{\lfloor log_m(N) \rfloor} (p_k * k * split\_cost)}_{some\ splits}$$

The first term corresponds to the case where no splits are triggered; $M-1$ replicas of the joined v-node are to be updated. The second term corresponds to other cases. We could have distinguished the case where the v-root splits. Its splitting probability is different; it has $M-1$ possible v-children quantities. But for $m > 2$ this probability is smaller than p so we just majorate it to keep things simple.

	2	4	8	16	32
10					
100					
1000					
10000					
100000					

Worst case In the worst case, the joined v-node and all its v-ancestors (including the v-root) split. The probability of that case is:

$$(p_{split})^{\lceil log_m(N) \rceil - 1} = \left(\frac{1}{M-m+1}\right)^{\lceil log_m(N) \rceil - 1}$$

$\lceil log_m(N) \rceil$ v-nodes are modified. Thus the overcost in terms of messages due to the replication in a join triggering $\lceil log_m(N) \rceil$ splits is:

$$\lceil log_m(N) \rceil * (M-1)$$

Average cost The average overcost of messages caused by replication during a join is:

$$\sum_{k=0}^{\lceil log_m(N) \rceil} p_{k-split} * cost_{k-split}$$

According previous calculations, this sum may be decomposed into two terms:

–
$$\sum_{k=0}^{\lceil log_m(N) \rceil - 1} \left[(p_{split})^k * (1-p_{split}) * (k+1) * (M-1) \right]$$

–
$$(p_{split})^{\lceil log_m(N) \rceil - 1} * \lceil log_m(N) \rceil * (M-1)$$

Replication implies no time overcost. Its overcost is the quantity of additional messages sent during joins. That quantity depends on the number of participant to the system and on the DR-tree degree.

6 Evaluation

Our evaluation experiments were conducted on a discrete ad-hoc simulator. At the start of a cycle, a p-node may check if it can try to improve the performance of the system by migrating its *migrable* v-node, if it has one. To this end, the p-node calls a score function which evaluates the benefit of a possible migration of this v-node.

A migration $m{=}p \xrightarrow{n} q$ concerns p-nodes holding v-neighbors of *n*; let note this set of p-nodes $concerned(m)$. We assume that *p* has some knowledge about the *cost* between its p-neighbors. To decide whether *m* is worth or not, we use the following *score* function:

$$score(m) = \sum_{r}^{concerned(m)} cost(p,r) - cost(q,r)$$

If p computes that $score(m) > 0$ then, according to p, m may enhance system performances. Having scores of possible migration of n, p will try to execute the migration with the best positive score. In our experiments, the metric related to the cost function is the latency between p-nodes.

One thousand simulations were performed with different configurations of DR-tree, with the following parameters:

– 500 p-nodes
– DR-tree degree with m = 2 and M = 4
– each experiment lasts 500 cycles
– the probability that a p-node checks for migrations is 1/10

DR-trees were populated with v-leaves represented by 50*50 2D rectangles randomly distributed in a 1024*1024 map. In order to simulate latency between p-nodes, we use the Meridian data set [16]; it is a latency matrix that reflects the median of round-trip times between 2500x2500 physical nodes spreads on Internet. For our experiments, we extracted a 500x500 sub-matrix.

6.1 Impact of migrations in all DR-tree configurations

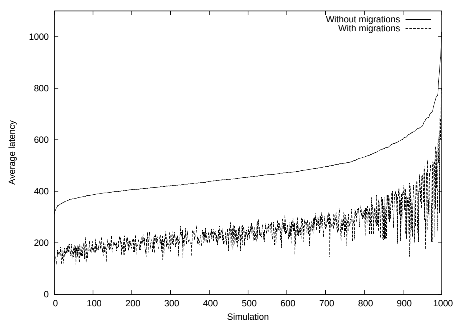
Impact on latency. We firstly studied the impact of migrations on the latency between two p-nodes of the interaction graph.

Figure 5a shows the average latency gain due to migrations. Simulations with different DR-tree configurations were ordered by increasing average latency without migration (X-axis). Y-axis corresponds to the average latency in millisecond in the communication graph when the system is stabilized, i.e., no more migration takes place.

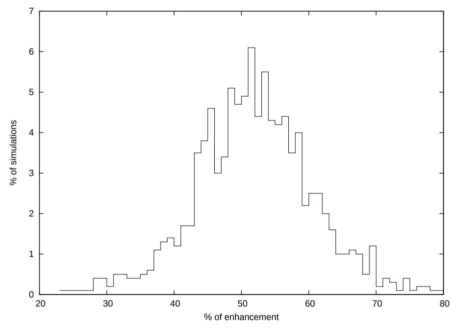
The average latency obtained without migrations -and thus without taking latency heterogeneity into account- is highly dependent on the mapping of v-root and its close neighborhood. In a small number of cases, this mapping was well-suited (resp. bad-suited), leading to an average latency of 300ms (resp. 1000ms). However, in almost 80% of simulations, the average latency was between 350ms and 600ms.

As we can observe in the same figure, migration of v-nodes clearly enhances average latency. The peaks of the curve can be similarly explained: in a small number of cases, many and/or very effective (resp. few and/or less effective) migrations were applied, leading to an average latency of 150ms (resp. 800ms). In most cases, the gain is around 50%.

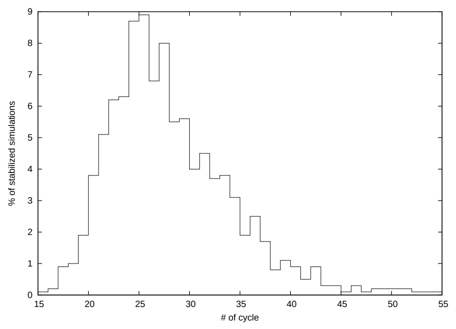
Figure 5b shows the gain distribution. X-axis is the percentage of enhancement reached during a simulation. Y-axis is the percentage of simulations where x gain has been reached. The gain distribution is quite gaussian: 78.7% of simulations raise a gain between 40% and 60%.



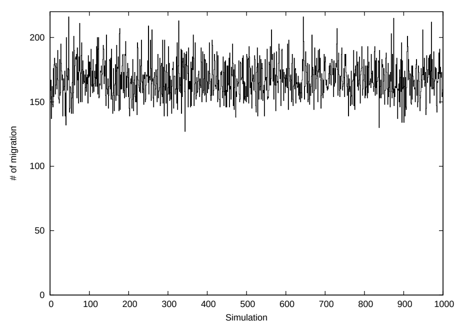
(a) Average latency enhancement



(b) Average latency enhancement distribution



(c) Stabilization time distribution



(d) Number of migrations per simulation

min	max	avg	stddev
15	55	27.494	5.84209

(e) Stabilization time distribution stat

min	max	avg	stddev
127	216	167.793	14.530

(f) Number of migrations per simulation stat

Fig. 5: Measurements on 1000 simulations

Stabilization time. Figure 5c shows the distribution of the stabilization time. X-axis is the last cycle where a migration was executed while Y-axis is the percentage of simulations where the last migration was executed at time *x*.

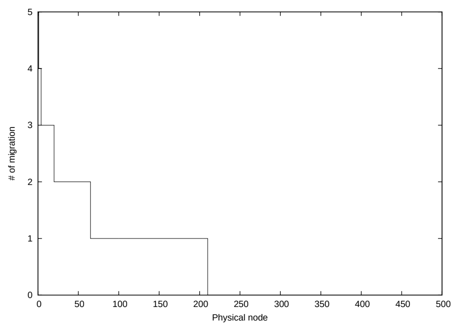
The distribution of stabilization time is also quite gaussian: 92.9% of simulations converge in a number of cycles between 20 and 40. Even with p-nodes “rarely” checking if they can perform migrations, stabilization is reached very fast: simulations last 500 cycles and 96.1% of them converge in less than 40 cycles (100% in less than 55 cycles).

Overall number of migrations. Figure 5d shows the number of migrations executed per simulation.

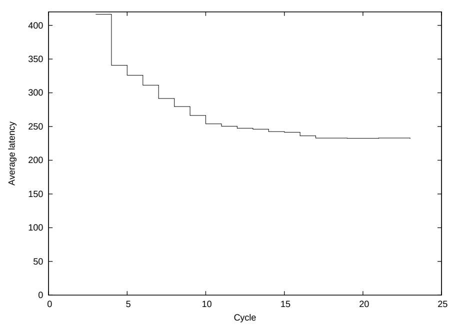
We observe a relatively stable number of migrations around a third of p-nodes with a low standard deviation. This suggests that different configurations of DR-tree with the same degree has low impact on the overall number of migrations. In fact, this number depends in fact on the distribution of latencies between p-nodes which is fixed in our experiments.

6.2 Analysis of migrations in a given DR-Tree configuration

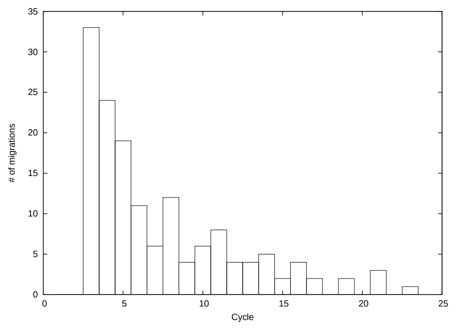
Among the previous simulations, we have chosen one where the corresponding DR-tree configuration presents a distribution enhancement of 50% (Figure 5b), a stabilization time of 25 cycles (Figure 5c), and 173 migrations executions (Figure 5d). Evaluation results presented in the following are based on this simulation.



(a) Number of migrations per p-node



(b) Average latency evolution over time



(c) Number of migrations per cycle

Fig. 6: Measurements on a very “average” simulation

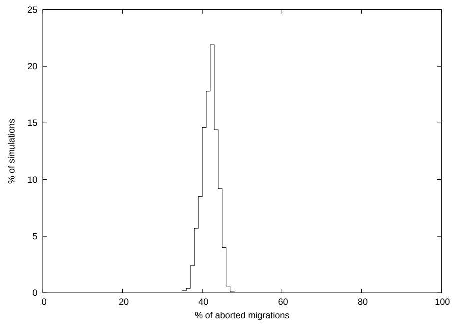
In Figure 6a, we can observe the number of migration per p-node during the referenced simulation. More than half of p-nodes do not participate in any migration and no p-node participates in more than five migrations. Furthermore, the distribution of per p-node migrations is relatively well-balanced since no p-node is overloaded by a high number of migrations and only 4 % of nodes perform more than two migrations.

Figure 6b shows the evolution of average latency between p-nodes in the interaction graph till stabilization time. All p-nodes start migration planner at the first cycle of the simulation. However, the first migrations actually occurs at the fourth cycle since our migration protocol takes four cycles to fully commit a v-node migration. Due to our migration protocol too, during the first cycles, migrations of high level p-nodes (and thus v-nodes closer to v-leaves) are likely to be aborted as their respective fathers are also likely to be executing other migrations. Therefore, the first executed migrations are more likely to concern lower level p-nodes (and thus low level v-nodes, close to the v-root) than higher level ones. Furthermore, the further from the v-root a v-node is, the longer the path to the v-root is, and thus its migration has a higher impact in the overall average latency. After these first migrations, the others are more likely to be small adjustments just inducing local communication enhancements which thus do not reduce average latency.

The number of migrations executed per cycle for the same simulation is given in Figure 6c. Y-axis is the number of executed migrations. The results shown in this figure confirm our previous analysis of Figure 6b: since all p-nodes start the migration planner during first cycle, many of them execute migrations during the fourth cycles. Since all p-node join the system during the first cycle, the higher the number of cycles executed, the smaller the number of executed migrations.

6.3 Abort rate

Our migration protocol is based on a best effort approach which implies that attempts of v-node migration may be aborted. Moreover, abortion of migrations has a cost due to the messages exchanged by the involved p-nodes till the protocol is aborted.



min	max	avg	stddev
51.929%	64.591%	58.130%	1.969

Fig. 7: Abort rate distribution

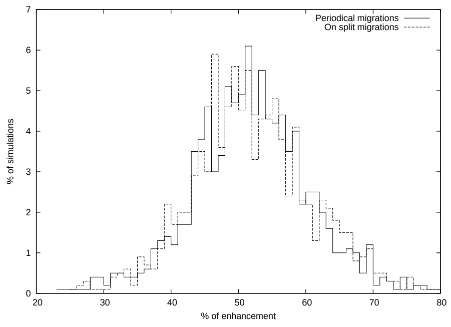
Figure 7 shows the abort rate distribution. X-axis is the percentage of aborted migrations while Y-axis is the percentage of simulations having *x*% of aborted migrations.

The rate of aborted migration is related to both the frequency of p-nodes attempts for migrating a v-node and the DR-tree degree. The higher this frequency is, the higher chances are that a p-node and its p-father try to execute concurrently conflicting migrations are. On the other hand, the higher the degree of the DR-tree is, the higher the chances are that a p-node can execute a migration and thus abort some of its p-neighbors migration attempts.

6.4 Migration scheduling impact

In this section, we compare two migration scheduling strategies for executing the migration planner: *periodical* strategy where the planner is executed periodically, and *triggered* strategy where the planner is executed whenever a v-node is split.

Each simulation is basically composed of two main phases: system building and system “lifetime”. The former starts at the simulation initialization and ends when the last p-node has joined the system while the latter starts after the last p-node has joined the system and ends at the simulation termination. In the periodically (resp. triggered) strategy, migrations only takes place during the system “lifetime” (resp. building);



	min	max	avg	stddev
Periodical	-23.173	-80.161	-51.998	8.460
Split triggered	-25.241	-80.168	-51.979	8.823

Fig. 8: Migration scheduling impact on average latency enhancement

Figure 8 shows the distribution gain of the two migration scheduling. X-axis is the percentage of enhancement reached during a simulation while Y-axis is the percentage of simulations where *x* gain has been reached. We can observe that both strategies are rather similar in terms of gain.

7 Conclusion

Our article has shown that it is very interesting to exploit the relation between the logical structure of a DR-tree [2] and its corresponding physical interaction graph. By expressing some invariants, a dynamic migration mechanism can modify the distribution of DR-tree v-nodes over the physical network without modifying its logical structure. Evaluation results have confirmed that even a very simple best-effort dynamic migration protocol substantially improves system performance in terms of latency. Finally, we should point out that our dynamic v-node migration mechanism could be applied to other logical structures. Concepts such as v-nodes, p-nodes, distribution invariants, migration policy, migration conflict management, migration planner, etc. can be easily generalized in order to satisfy other logical structures requirements.

References

1. S. Baehni, P. T. Eugster, and R. Guerraoui. Data-aware multicast. In *DSN*, page 233, 2004.

2. S. Bianchi, A. K. Datta, P. Felber, and M. Gradinariu. Stabilizing peer-to-peer spatial filters. In *ICDCS 2007*, page 27, 2007.
3. P. C. and V. K. A topologically-aware overlay tree for efficient and low-latency media streaming. In *QSHINE*, pages 383–399, 2009.
4. R. Y. W. C. Zhang, A. Krishnamurthy. Brushwood: Distributed trees in peer-to-peer systems. pages 47–57, 2005.
5. M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. I. T. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in cooperative environments. In *IPTPS*, pages 292–303, 2003.
6. G. Chockler, R. Melamed, Y. Tock, and R. Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *PODC*, pages 109–118, 2007.
7. Y. Choi and D. Park. Mirinae: A peer-to-peer overlay network for large-scale content-based publish/subscribe systems. In *NOSSDAV*, pages 105–110, 2005.
8. P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.
9. R. Guerraoui, S. B. Handurukande, K. Huguenin, A.-M. Kermarrec, F. L. Fessant, and E. Riviere. Gosskip, an efficient, fault-tolerant and self organizing overlay using gossip-based construction and skip-lists principles. In *Peer-to-Peer Computing*, pages 12–22, 2006.
10. A. Gupta, O. D. Sahin, D. Agrawal, and A. E. Abbadi. Meghdoot: content-based publish/subscribe over p2p networks. In *Middleware*, pages 254–273, 2004.
11. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, 1984.
12. N. J. A. Harvey and J. I. Munro. Deterministic skipnet. *Inf. Process. Lett.*, 90(4):205–208, 2004.
13. H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: a balanced tree structure for peer-to-peer networks. In *VLDB*, pages 661–672, 2005.
14. H. V. Jagadish, B. C. Ooi, Q. H. Vu, R. Zhang, and A. Zhou. Vbi-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *ICDE*, page 34, 2006.
15. Y.-J. Joung. Approaching neighbor proximity and load balance for range query in p2p networks. *Comput. Netw.*, 52(7):1451–1472, 2008.
16. H. V. Madhyastha, T. Anderson, A. Krishnamurthy, N. Spring, and A. Venkataramani. A structural approach to latency prediction. In *IMC*, pages 99–104, 2006.
17. J. Pujol Ahullo, P. Garcia Lopez, and A. F. Gomez Skarmeta. Towards a lightweight content-based publish/subscribe services for peer-to-peer systems. *Int. J. Grid Util. Comput.*, 1(3):239–251, 2009.
18. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *SIGCOMM*, pages 161–172, 2001.
19. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, 2001.
20. W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, pages 1–8, 2003.