Fully Dynamic Balanced and Distributed Search Trees with Logarithmic Costs *

Adriano Di Pasquale[1] **Enrico Nardelli**[1,2]

1. Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia. E-mail: nardelli@univaq.it
2. Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italia. – **CONTACT AUTHOR**

Printed on: August 15, 1999

Abstract

In this paper we consider the dictionary problem in a message passing distributed environment. We introduce a new version of an order-preserving distributed search tree, called BDST for *Balanced and Distributed Search Tree*, capable to both grow and shrink as long as keys are inserted and deleted. This is the first distributed data structure to explicitly support both insertion and deletion with logarithmic costs, i.e. a key can be searched, inserted and deleted in $O(\log n)$ messages, where n is the number of servers. Moreover a range query can be performed in $O(\log n + \lceil \frac{k}{b} \rceil)$ messages, where k is the number of items returned by the search and b is the capacity of each server. Since balance is explicitly maintained, the structure is able to adapt itself to any input distribution and does not depend on any uniformity assumption to obtain logarithmic performances.

Keywords: distributed data structure, fully dynamic, order preserving, message passing environment, range queries.

1 Introduction

With the striking advance of communication technology it is now easy and cost-effective to set up distributed applications running on a network of workstations. The technological framework we make reference to is the so called *network computing*: fast communication networks, in the order of 10-100MB/sec, and many powerful and cheap workstation, in the order of 50-100 MIPS. Many organizations have this kind of computing power: large organizations have easily a cumulative amount of main memory in the order of tenths of GB.

In this work we consider the dictionary problem in a message passing distributed environment. Litwin, Neimat e Schneider [3] were the first to present and to discuss for this environment a data structure paradigm called SDDS (*Scalable Distributed Data Structure*). The main properties of SDDS paradigm are:

*Research partially supported by the European Union TMR project "Chorochronos".

1. Keep a good performance level while the number of managed objects changes.
2. Perform operations locally.

The distributed environment we make reference to is constituted by a set of *sites* (processor or nodes) connected by a network. Every site in the network is either a *server*, that manages data, or a *client*, that requests access to data. Each server manages data items belonging to some parts of the data domain. Sites communicate by sending and receiving *point-to-point* messages. We assume network communication is free of errors. Every server can store a single block (called *bucket*) of at most b data items, for a fixed number b. The overall data organization scheme we consider is a search tree: servers manage both nodes containing data items (*leaf nodes*) and nodes guiding the search process (*internal nodes*).

The data distribution and management policy determines how data are distributed among the servers; there are no preconditions as to where the data can be stored. New servers can be added as the volume of data increases to maintain the performance level. The clients are not, in general, up-to-date with the evolution of the structure, in the sense they have some local indexing structure, but do not know, in general, the overall status of the data structure. Different clients may therefore have different and incomplete views of the data structure.

The fundamental measure of the efficiency of an operation in this distributed context is the number of messages exchanged between the computers of the network. In the literature various kinds of SDDSs have been proposed: LH* [3], RP* [4], DRT [5], lazy k-d-tree [7, 9], RBST [8].

All previous proposals but RBST considered explicitly only the semi-dynamic case, that is the case where keys are only inserted and never deleted. In this work we focus on the extensions of binary search trees to the distributed case (like DRT and RBST) and consider a fully dynamic context, i.e. keys can be both inserted and deleted.

The theoretical study of the characteristics of scalable distributed search trees conducted by Kröll e Widmayer [10] showed that if all the hypothesis used to efficiently manage search structures in the single processor case are carried over to a distributed environment then a lower bound of $\Omega(\sqrt{n})$ holds for the height of balanced search trees.

Nardelli *et al.* devised in [8] an SSSD, called RBST, where some of these hypothesis, related to the way the search process is executed, are relaxed, yielding a cost of $O(\log^2 n)$ messages for search and update operations, where n is the number of servers in the structure.

In this paper, we relax some other hypothesis, related to the kind of synchronization between servers and clients of the structure, and show that a distributed search trees can be maintained balanced in a distributed environment so that search and update operations can be executed with $O(\log n)$ messages. Hence we present the first balanced distributed search structure to be fully dynamic and order-preserving. We also show how to efficiently answer range queries in $O(\log n + \lceil \frac{k}{b} \rceil)$ messages, where k is the number of items returned by the search and b is the capacity of each server. Since the structure is explicitly kept balanced, logarithmic performances are obtained without relying on uniformity assumptions. A preliminary version of this paper was presented in [1].

The paper is structured as it follows. In Section 2 we describe the context of our work and the main idea our data structure is based on. Section 3 describes the search process, while Section 4 discusses how insertion and deletions are managed to keep the structure balanced. Sections 5 and 6 give details on algorithms for insertion and deletion, respectively. Their correctness is proved in section 7, while Section 8 discusses issues related to rotations used

to keep the structure balanced. Section 9 concludes the paper with final remarks and future work.

2 Context

More formally, let T be a binary search tree with n leaves (and then with $n-1$ internal nodes). We call $f_1, \ldots, f_n$ the leaves and $t_1, \ldots, t_{n-1}$ the internal nodes. To each leaf a bucket capable of storing b data items is associated. Let $s_1, \ldots, s_n$ be the n servers managing the search tree. We define *leaf association* the pair (f, s), meaning that the server s manages the leaf f and its associated bucket, *node association* the pair (t, s), meaning that the server s manages the internal node t. In an equivalent way we define the two functions:

- $t(s_j) = t_i$, where (t_i, s_j) is a node association,
- $f(s_j) = f_i$, where (f_i, s_j) is a leaf association.

To each node x, either leaf or internal one, the interval $I(x)$ of data domain managed by x is associated.

In the centralized case a search tree is a binary tree such that every node represents an interval of the data domain. Moreover, the overall data organization satisfies the invariant that the interval managed by a child node lies inside the father node's interval. Hence the search process visit a child node only if the searched key is inside the father node's interval.

Kröll and Widmayer call this behavior the *straight guiding property* [10]. They observed that it is not possible, in the distributed case, to directly make use of rotations for balancing a distributed search tree while guaranteeing the straight guiding property. They proved that a lower bound of $O(\sqrt{n})$ holds for the height of balanced search trees if the straight guiding property has to be satisfied.

In [8] Nardelli *et al.* devised a distributed search tree, called RBST (for *Relaxed Balanced Search Tree*) where, by accepting a violation of the straight guiding property, the height of the tree is kept logarithmic and all update operations have a logarithmic cost, but the upper bound on the complexity of the search process is $O(\log^2 n)$.

In the following we relax the requirement of the straight guiding property, but by assuming a different synchronization mechanism between clients' local indexes and servers we show how to keep a distributed binary search tree balanced while all operations are maintained within a logarithmic upper bound.

2.1 Basic idea

In all previous works on SDDS, whenever a client index is introduced to improve performances, it is always built and managed to exactly reflect the global tree structure. This means that both clients and servers keep track of both node associations and leaf associations. Moreover it is assumed that the knowledge the client has of the global tree structure is partial and almost exact, in the sense it may possibly be incomplete and at a coarser level of detail than it is in the reality. A correction to a client index consists only in adding more detailed information.

If one wants to keep the overall structure balanced then rotations in the overall tree have to be used. But after a rotation in the overall tree has been performed, client indexes do not

represent any more, in general, a portion of the global tree in an exact way. The approach of sending messages from servers to all clients whenever a rotation is performed is clearly not an efficient solution.

Our basic idea to obtain logarithmic costs is to relax the synchronization between clients and server indexes. By accepting a structural mismatch between the overall index and the local indexes we can then use rotations to maintain the overall tree balanced. The straight guiding property is still violated but we are now able to keep a logarithmic upper bound on both search and update operations. We recently discovered that a similar approach was followed by Kröll [11].

To be more precise, we manage in different ways the two associations in the two types of indexes. Servers store in their index both node and leaf associations, while clients record only leaf associations. A rotation in the overall tree structure only affects node associations, since we never rotate leaves.

The global tree is therefore kept balanced and the search process is bounded by logarithmic costs. On the other side, client indexes will never have to be modified due to rotations, since they do not keep track of node associations.

2.2 The data structure

The distributed data structure we focus on is a binary search tree, where data are stored in the leaves and internal nodes contains only routing information. Every node has zero or two children. For a binary search tree T we denote with $h(T)$ the height of T, that is the number of internal nodes on a longest path from the root to a leaf. Every server s but one, with leaf node association (t, s) and leaf association (f, s), records at least the following information:

- An internal node $t = t(s)$ and the associated interval of key's domain $I(t)$,
- The server $p(s)$ managing the parent node $pn(t)$ of t, if t is not the root node,
- The server $l(s)$ (resp., $r(s)$) managing the left child $ls(t)$ (resp., right child $rs(t)$) of t, and the associated interval $I_l(t)$ (resp., $I_r(t)$),
- A leaf $f = f(s)$ and the associated interval of key's domain $I(f)$,
- The server $pf(s)$ managing the father node $pn(f)$ of f, if f is not the unique node of global tree (initial situation).

This information constitutes the local tree $lt(s)$ of server s (see figure 1). Since in a global tree of n nodes there are $n-1$ internal nodes, there is one server s' managing only a leaf association, hence $lt(s')$ is made up by only the two last pieces of information in the above list.

We say a server s is *pertinent* for a key k, if s manages the bucket to which k belongs. In our case if $k \in I(f(s))$. Moreover we say a server s is *logically pertinent* for a key k, if k is in the key interval of the internal node associated to s, that is if $k \in I(t(s))$. Note that the server managing the root is logically pertinent for each key. Note also that, due to the effect of rotations, it is not necessarily $I(f(s)) \subseteq I(t(s))$.

When a server sends a message, it always adds its local tree to it. This is useful to increase the knowledge about the global structure in the client receiving the message. As soon as a client receives an answer from a server, it uses the received local tree to update its local index,

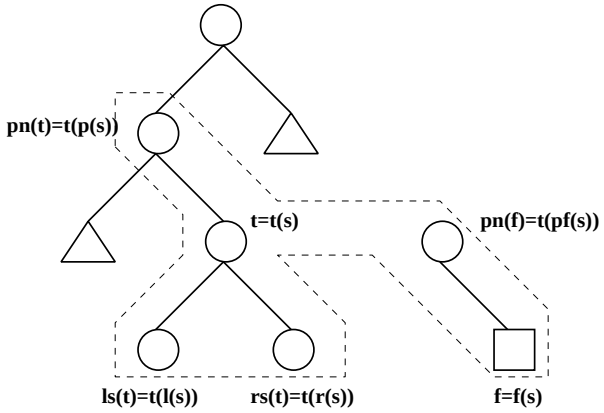

Figure 1: The local tree of server s.

where only leaf associations are stored. A client uses its local index to better address its queries.

2.3 The client index

Every client manages an index to reduce addressing errors. This is a collection, in general incomplete, of leaf associations. Since our complexity measure is the number of messages on the network, then it is not important which is the structure used to store the associations. It can be a list or a search tree. If it is a search tree, its structure is, in general, different from the structure of the global tree.

A client uses its index to find the server s which should answer to a query so to issue a point-to-point message to s. If this server is not found, then the client must send the query to the server managing the root of the global tree. This is true, in particular, for a new client, whose index is empty.

When a client issues a query, it receives in the answer message a certain number of servers's local trees (owned by the servers involved in the search process). It uses these local trees to improve the knowledge about the overall structure recorded in its index. If the client has a leaf association (f, s) stored in its index, it knows that server s manages interval $I(f(s))$. In the reality it may be that either s has been released due to an underflow or s is managing a sub-interval of $I(f(s))$. Local trees received as part of the answer to search queries are used to update this information.

Note that it may happen that a client can send a request to a server that has been released. In this case the client has to use some timeout mechanism. When the timeout period expires it sends the request to the server managing the root like if it were a new client.

3 The search process

We now describe how to search in our structure, called BDST for *Balanced and Distributed Search Tree*. We examine which events can occur and algorithms to treat them.

Event 1. A query from a new client. A new client is a client that never issued a query to the structure and then has no knowledge about it. Such a client, say c, always send the request of a key k to the server r managing the root of global tree. If r is the pertinent server for k, then r manages the request and answers to c, else it chooses between the servers $l(r)$ and $r(r)$ managing its left and right sons the pertinent or logically pertinent one for k and sends it the request. Note that one of the two has to be at least logically pertinent. The process continues until the request arrives to the pertinent server s' for k. s' manages the request and answers to c, see figure 2 (left).

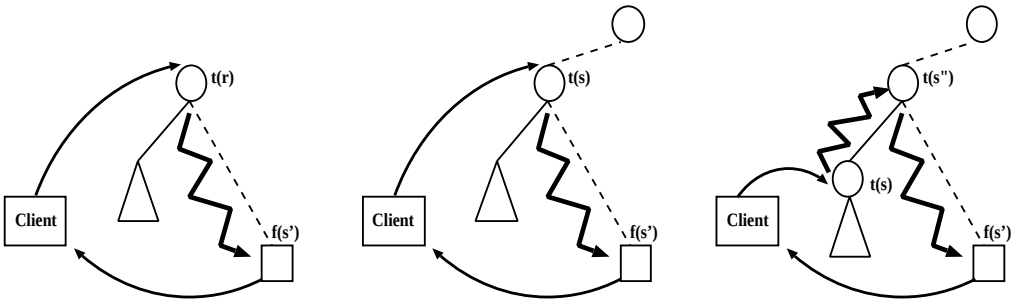

Figure 2: Searching queries from a new client (left), from a client with addressing error sending its request to: a logically pertinent server (center) and a non logically pertinent server (right).

Event 2. A query from a client without addressing error. A client c sends the request for a key k to a server s which is the pertinent server for k. s manages the request and answers to c.

Event 3. A query from a client with addressing error. A client c sends the request for a key k to a server s, but s is not the pertinent server for k.

If s is logically pertinent (see figure 2 center) for k then s chooses between the servers $l(s)$ and $r(s)$ managing left and right sons the pertinent or logically pertinent one for k and sends it the request. Note that one of the two has to be at least logically pertinent. The process continues until the request arrives to the pertinent server s' for k. s' manages the request and answers to c.

If s is not logically pertinent (see figure 2 right) for k then s sends the request to $p(s)$, i.e. the server managing the father of $t(s)$. From $p(s)$ the search may proceed further upwards. There is certainly a node t'' in the path between $t(s)$ and the root such that its managing server s'' is pertinent or logically pertinent for k. If s'' is pertinent then it behaves like s'. If s'' is only logically pertinent then it chooses between the servers managing left and right sons and continues as in previous case, see figure 2 (right).

Theorem 3.1 *Let T be a BDST and let $h = h(T)$. Searching for a given key requires in the worst case $O(h)$ messages.*

Proof. If event 1 happens, a chain of messages departs from the root and arrives to a leaf. In the worst case, the chain is composed by h messages. Counting also request and answer messages, $h + 2$ messages are needed.

If event 2 happens, only O(1) messages are needed (namely, the request and answer message).

If event 3 happens, then we distinguish two cases. In the first case, s is logically pertinent, and $h+2$ messages are needed. In the second one, s is not logically pertinent, hence we must go up in the global tree to found the logically pertinent server. In the worst case we depart from a leaf at height h and arrive to the root, then we go down again to another leaf of height h (see figure 3). In total we need $2h+2$ messages. □

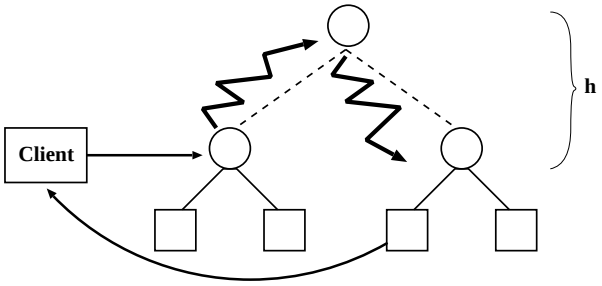

Figure 3: The worst case for searching.

Now, if we keep the global tree balanced during updates by using rotations, the height h always remains bounded by $O(\log n)$, hence also the cost of search process is bounded by $O(\log n)$.

3.1 Range queries

Now we describe how to perform a range query (see figure 4).

A client searches in its index for a server with a leaf interval internal to the range of the query and sends it the request. This server sends the request from f upwards until the lowest node t' that covers the query range is found. In the case of a new client (i.e., no information is stored in the client index) or in the case of an address error (i.e., when the request from the client is addressed to a server with an interval outside the range of the query), to reach t' we operate like in the case of exact search.

Then t' sends the request downward in the tree towards each of the $\lceil \frac{k}{b} \rceil$ servers within the range. All these servers answer to the client.

To reach t' we follow an upward path from a leaf to t'. The length of this path is $O(\log n)$. Then to reach the leaves from t' we use one message for each edge of the sub-tree with k leaves rooted at t', therefore a range query has a cost of $O(\log n + \lceil \frac{k}{b} \rceil)$ messages in the worst case.

4 Insertion and deletion

We now describe how to perform insertion and deletion in a BDST. Please note that in a distributed environment insertion and deletion refers, respectively, to the creation of a new server that receives part of the keys previously managed by an existing server that is now in overflow and to the release of an existing server that is now in underflow and sends all its keys to an existing server. Insertion and deletion of data items that do not cause, respectively,

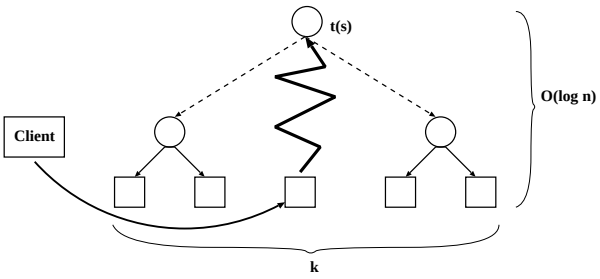

Figure 4: The range query.

overflow and underflow, do not require any rebalancing action, and their complexity analysis is the same of searching data items. When overflows and underflows occur, we must perform some actions to keep the structure balanced and a binary search tree (i.e. each node has either zero or two children).

The balance actions must affect only internal nodes and never change the leaves, since rotating the leaves would force to transfer the whole bucket content to another server and this is not efficient. This means that during balancing only node associations change while leaf associations remains the same. Therefore a leaf can change its father, but can never become an internal node. It is possible to use any balancing technique which satisfies these assumptions and keeps the costs logarithmic.

In the description of algorithms for insertion and deletion we assume that a server is able to execute a function, called *balance_bdst*, which performs the action that may be needed to keep the BDST balanced after an update. We assume *balance_bdst* uses at most $O(\log n)$ messages, where n is the number of servers managing the BDST, and that before the execution of the algorithms described below the BDST is already balanced, i.e. h, the height of BDST, is bounded by $O(\log n)$.

As a preliminary remark please note that an important problem in this distributed, fully dynamic context is the following hysteresis situation.

One server receives an insert operation, goes in overflow and splits itself in two to return within the size bound. Just after the split, it receives a delete operation. Now its bucket goes under $\frac{b}{2}$ keys and then it has to manage the underflow, and eventually performs a merge. This sequence of split and merge can be possibly repeated any number of times, with a corresponding degrade of the structure's performances.

This is a well known problem in the theory of file structures. One way to tackle it is to avoid to split an overflowing server or to release an underflowing server by exchanging keys between adjacent leaves.

Another way is to assign to the underflow a lower threshold than the one for the overflow, e.g. at $\frac{b}{3}$ instead that at $\frac{b}{2}$.

In general, we may to deal with this problem by applying any strategy known from the file structure theory.

The approach based on checking for a possible transfer of keys is carried out by the function *transfer_keys*. This is composed by the following sub-steps (see figure 5):

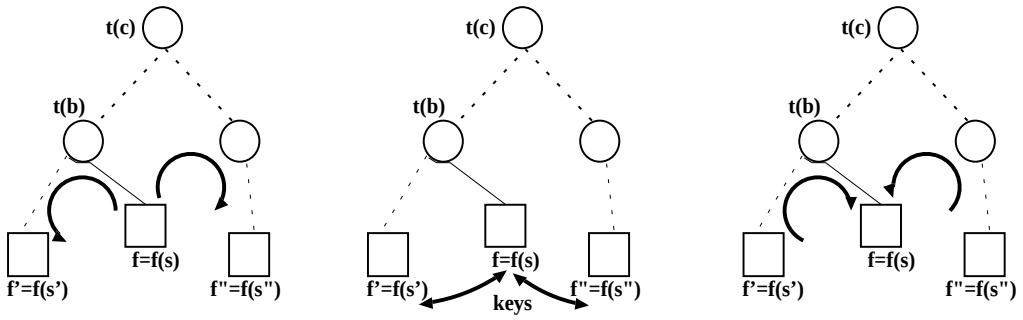

Figure 5: Transfer keys among adjacent leaves

1. **Looking for the adjacent leaves** (figure 5, left). The server s starts a chain of *search* messages toward the adjacent leaves $f' = f(s')$ and $f'' = f(s'')$ of $f(s)$.

2. **Decide the policy** (figure 5, center). On the basis of information received by s' and s'', s decides if it is possible to exchange keys among them without splitting/ releasing the server s. If it is possible, there will be a transfer of keys between s' and s and/or between s'' and s. How many keys to transfer and whether to involve in the transfer both s' and s'' or not are further issues to be considered for performance optimization purposes.

3. **Interval change and key transfer** (figure 5, right). If keys have been transferred, then from f' and f'' a chain of *change* messages starts and follows the same path of *search* messages and return to f. Each node reached by these messages, changes its key interval.

5 Algorithm for insertion

Step 1: Insert – We search for the leaf where the new key has to be inserted and insert it. We assume that this insert generates an overflow, that is the key to be inserted is the $(b+1)$-th key assigned to that bucket.

Step 2: Manage the overflow – Leaf f, managed by server s, goes in overflow. In this case we have to decide whether s has to be split or if it is possible to transfer its keys to adjacent nodes. Details about this aspect have been discussed in the previous section. Assume then the decision was to split the node. Then s must perform a function called *split*. This function is similar to the synonymous one described in [3, 5]. Leaf f splits in two new leaves f_1 and f_2. A new internal node t_{n+1} replaces f in the tree. A new server s_{n+1} is called to manage the new internal node and one of the new leaf. Server s releases the old leaf f and manages the other new leaf.

In conclusion we delete leaf association (f, s) and add two leaf associations (f_1, s) and (f_2, s_{n+1}) and one node association (t_{n+1}, s_{n+1}) (see figure 6). The old interval $I(f)$ is divided in the new intervals $I(f_1)$ and $I(f_2)$, such that $I(f_1) \cup I(f_2) = I(f)$.

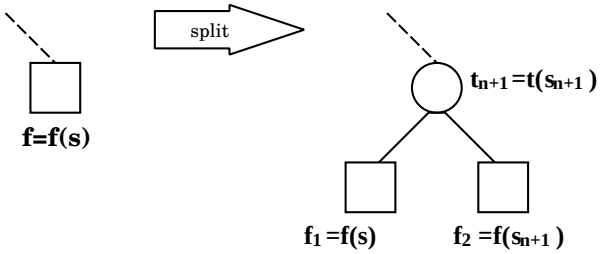

Figure 6: Insertion of an element in an overflowing bucket

Step 3: Balance the BDST – Perform the *balance_bdst* function starting from t_{n+1}.

Theorem 5.1 *Insertion in a BDST made up by n servers costs in the worst case $O(\log n)$ messages.*

Proof. From the algorithm above we have in the worst-case the following costs for the various steps:
Step 1: From theorem 3.1 this costs $O(\log n)$ messages.
Step 2: A constant number of messages is needed to perform the split function (see [3, 5]).
Step 3: From the assumptions above we have a cost of $O(\log n)$ messages. □

6 Algorithm for deletion

Step 1: Delete – We search for the leaf where the key has to be deleted and delete it. We assume that this generates an underflow, that is by deleting that key the bucket has less than $\frac{b}{2}$ keys.

Step 2: Manage the underflow – The leaf f, managed by server s, goes in underflow. In this case we have to decide whether s has to be released or if it is possible to transfer keys from the adjacent leaves, without releasing s. Details on this aspect have been discussed in section 4. Assume then the decision was to release s. Then s performs a function called *merge*. This is its behaviour:

If f is the root, the BDST is composed by one node and then no action are performed.

If the BDST is composed by the root r and two leaves f and x, there are only two servers s and s'. Then s is released and after the communication to s' and the deletion of r, x become the root of BDST. All the keys of f are sent to x.

In the general case f is the leaf in figure 7. The case with f as left son is analogous. We assume b is the server such that $t(b)$ is the father node of $f(s)$ and c is the server such that $t(c)$ is the father node of $t(b)$. $t(a)$ can be a leaf or an internal node. In this case the function is constituted by the following sub-steps (see also figure 7):

1. Release server s and delete leaf $f = f(s)$.

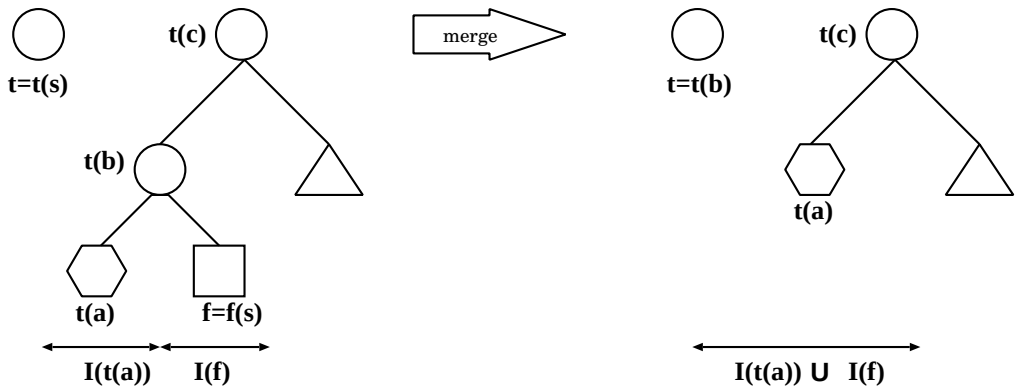

Figure 7: Deletion of an element from an underflowing bucket

2. Since node $t(b)$ has now one son, then delete $t(b)$ and replace it with $t(a)$ as the son of $t(c)$.

3. If s managed an internal node $t = t(s)$, then from now on t is managed by server b (note that b has just released its internal node $t(b)$).

Step 3: Balance the BDST – Perform the *balance_bdst* function starting from $t(c)$.

Theorem 6.1 *Deletion in a BDST made up by n server costs in the worst case $O(\log n)$ messages.*

Proof. From the algorithm above we have the following worst case costs for the various steps:
Step 1: From theorem 3.1 this costs $O(\log n)$ messages.
Step 2: From lemma 7.2 this costs $O(\log n)$ messages.
Step 3: From the assumptions above we have a cost of $O(\log n)$ messages. □

7 Proofs of correctness

In the next lemma we prove that every message needed to perform the *merge* and *transfer_keys* functions can actually be sent, i.e. every server searching in the local tree eventually finds the servers destination of messages.

Lemma 7.1 *The* merge *and* transfer_keys *functions are correct with respect to the local tree of the servers involved.*

Proof. For the *merge*: In step 2 server s has to notify to b that it has to release its internal node $t(b)$. This can be done since b is the father of $f = f(s)$ and then is in the local tree of s. Server b has to notify to servers a and c the change of, respectively, the father of $t(a)$ and the son of $t(c)$. This can be done since we can find a and c in the local tree of b. In step 3, if s managed an internal node t, then s has to notify to b the new internal node t to manage (this can also be performed in previous messages from s to b) and which are the father and the sons

of t. Then this change has to be notified to the servers managing the father and the sons of t. All the required information is in the local tree of s.

For the *transfer_keys*: Each server sends a *search* message or a *change* messages to its parent or to its child, therefore it finds the destination of message in its local tree. The *search* messages transport the address of s, therefore s' and s'' can exchange messages with s. □

Lemma 7.2 *The* merge *function costs* $O(1)$ *messages in the worst case. The* transfer_keys *function costs* $O(\log n)$ *messages in the worst case.*

Proof. For the *merge*: From lemma 7.1 we can see that step 2 needs one message from s to b, one from b to a, and one from b to c.

If s was not managing an internal node t then step 3 needs zero messages, else it needs one message from s to b, one from s to the server managing the father of t (zero if n is the root), and two from s to the servers managing the sons of t. This makes a total of 6 messages.

If b coincides with s then only two messages are needed. In the two special cases we have, respectively, zero and one messages.

For the *transfer_keys*: We follow at most four times a path in the tree from $t(c)$ to a leaf. The length L of this path is $O(\log n)$ in the worst case. Counting the remaining messages, we have $4L + 9$ messages. □

The servers involved in the *merge* function have to be locked, like in the *split* function case. In the next lemma we show that on the contrary the *transfer_keys* does not need to lock the servers.

Lemma 7.3 *The* transfer_keys *may be correctly executed without locking the involved servers.*

Proof. We want to prove that during every steps of the *transfer_keys*, each request of keys in $I(f)$ will be satisfied. We give a proof for the case where we transfer the keys from s to s'. The proof for the other case is analoguous.

We denote with R the requests of keys belonging to $I(f)$ (see figure 8).

1. As long as $f(s')$ has not received the keys and changed its interval, each request follows the path to $f(s)$ where the keys reside.

2. After $f(s')$ has received the keys, requests arriving to s' for keys in $I(f)$ do not go upwards like in the previous case, but are directly satisfied by s'. Requests arriving to s are forwarded to s'.

3. For each internal node $s*$ belonging to the path beetween $t(a)$ and $f(s')$, after $s*$ has been reached by the *change* messages and has changed its interval, the requests arriving to it for keys in $I(f)$ do not go upwards like before, but are sent downward to $f(s')$. These requests will be satisfied because s' now has the keys. The ancestors of $s*$, that have not yet been reached by the *change* messages, send the requests upwards to $f(s)$. These messages will be satisfied too, because when will arrive to s, s will forward them to s'.

4. When the *change* messages arrive to s, eventually s begins the *merge* function, locking the involved servers.

□

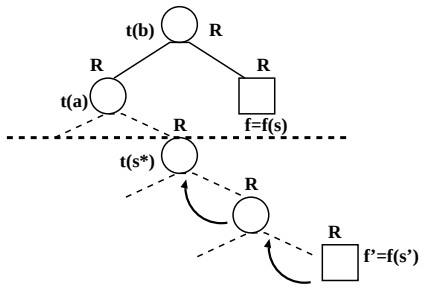

Figure 8: The chain of change messages

8 Rotations in a distributed environment

Rotations in a distributed environment are performed via message exchanges between servers. Since we are in a concurrency framework, in the sense that various clients independently manipulate the structure, each rotation must be preceeded by a lock of the servers involved. Then some messages are needed to create the lock, others to communicate the modifications and others to release the lock. Each rotation has therefore a cost in terms of messages. We can show that the cost of one rotation is a constant and then if a balancing strategy uses a logarithmic number of rotations for operation, then the overall cost is kept logarithmic.

We show by means of an example how to execute rotations in a distributed environment. Without loss of generality, let us consider figure 9 (first), and suppose that node a must rotate with node b. The flow of events is the following:

1. a sends messages to (client) nodes A, B and to (server) node b, to notify that a lock must be created. After having received these messages, nodes A, B, and b stop routing messages towards a and send a lock acknowledgement to a.

2. b sends messages to (client) node C and to (server) node c, to notify that a lock must be created and that acknowledgement must be sent to a. After this message, nodes C and c stop routing messages towards b.

3. Every server answers to a, see figure 9 (second), to acknowledge the lock state.

4. a notifies to all servers involved in the rotation which modifications are needed and after all servers have been confirmed a releases all locks, see figure 9 (third).

5. When locks are released the situation is shown in figure 9 (last) and all servers restart to route messages.

It is easy to prove that the example is correct with respect to the local tree of a server. We used 15 messages and 5 servers are involved. We note that in each rotation exists a server that does not need to be informed of the rotation, and then is not involved in the lock. In the discussed example this server is C. We can therefore improve the procedure and use only 12 messages (with 4 servers involved).

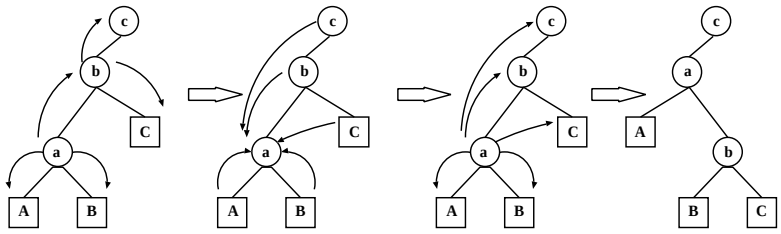

Figure 9: Locking messages during a rotation

Each lock, in a certain sense, reduces the degree of concurrency and this is a drawback in a distributed environment. It is then important to keep the number of locks small.

Although any balancing strategy with a logarithmic number of messages is good for the general objective, we must focus on those minimizing the number of rotations and then the number of locks. For example the *splay tree* [12] uses a large number of rotations.

It is more convenient to use a data structure like a *red-black-tree* [13], which has a constant number of rotations both for deletion and insertion operations.

Much work has been done about reducing the number of rotations while balancing a concurrent search tree [2, 6], but this regards the concurrent, shared-memory case.

There is a big difference between this kind of work and the distributed tree studied here. In [2, 6] every update operation can unbalance the structure, while in our case a great number of update operations do not cause an unbalance to the structure.

This is due to the fact that data are managed in buckets of size b. If a server s start with an empty bucket, b insert operations addressed to s do not cause an overflow and do not change the distributed tree's structure. More in general if we have k insert operations in a structure where each server manages $\frac{b}{2}$ keys (i.e. every server has just performed a split), then the number of overflows (and then of splits) is bounded by $\lceil \frac{2k}{b} \rceil$ (the bound holds when all k inserts are in the same server). Then if b is large, we have a low number of overflows. An analogous situation holds for underflows.

9 Conclusions

We have presented an approach to keep a distributed binary search tree balanced, enabling it to manage both insertion and deletion of data items in a message-passing distributed environment.

Hence we have shown that a fully-dynamic and order preserving distributed search structure, that is a structure that is able to grow and shrink as long as data items are inserted and deleted, can be implemented in a message-passing distributed environment as efficiently, namely with a $O(\log n)$ worst case bound, as in the single processor case. We have also shown how to answer range queries with $O(\log n + \lceil \frac{k}{b} \rceil)$, where k is the number of returned elements and b is the bucket size.

Our data structure keeps the same good level of performance for every distribution of the keys in the domain of values. There are other order-preserving distributed structure with good performances, but often only under the hypothesis of a uniform distribution of the keys. Future

work will focus on a thorough experimental analysis of our data structure behavior, also in comparison with its competitors.

References

[1] A.Di Pasquale, E. Nardelli: Balanced and Distributed Search Trees, *1st Southern Symposium on Computing*, Hattiesburg, Ma., December 1998.

[2] J. Eckerle, O. Nurmi: Technical Report Aug17-7, Technical University of Munich, 1994.

[3] W. Litwin, M.A. Neimat, D.A. Schneider: LH* - Linear hashing for distributed files, *ACM SIGMOD Int. Conf. on Management of Data*, Washington, D. C., 1993.

[4] W. Litwin, M.A. Neimat, D.A. Schneider: RP* - A family of order-preserving scalable distributed data structure, in *20th Conf. on Very Large Data Bases*, Santiago, Chile, 1994.

[5] B. Kröll, P. Widmayer: Distributing a search tree among a growing number of processor, in *ACM SIGMOD Int. Conf. on Management of Data*, pp 265-276 Minneapolis, MN, 1994.

[6] K. Larsen, E. Soisalon-Soininen, P. Widmayer: Relaxed balance through standard rotations, in *Workshop on Algorithms and Data Structures*, Halifax, Nova Scotia, Canada, August 1997.

[7] E. Nardelli: Distribuited k-d trees, in *XVI Int. Conf. of the Chilean Computer Science Society (SCCC'96)*, Valdivia, Chile, November 1996.

[8] F. Barillari, E. Nardelli, M. Pepe: Fully Dinamic Distribuited Search Trees Can Be Balanced in $O(\log^2 N)$ Time, Technical Report 146, Dipartimento di Matematica Pura ed Applicata, Universita' di L'Aquila, July 1997, submitted for publication.

[9] E. Nardelli, F.Barillari, M. Pepe: Distributed Searching of Multi-Dimensional Data: a Performance Evaluation Study, Journal of Parallel and Distributed Computation, 49, 1998.

[10] B. Kröll, P. Widmayer: Balanced distributed search trees do not exists, in *4th Int. Workshop on Algorithms and Data Structures (WADS'95)*, Kingston, Canada, (S. Akl et al., Eds.), Lecture Notes in Computer Science, Vol. 955, pp. 50-61, Springer-Verlag, Berlin/New York, August 1995.

[11] B.Kröll: Dynamisch verteilte Woerterbuecher. PhD thesis, ETH Zürich, Institute of Theoretical Computer Science, February 1997.

[12] D.D. Sleator, R.E. Tarjan: Self-Adjusting Binary Search Trees, Journal of the ACM 32(3):652-686, 1985.

[13] T.H. Cormen, C.E. Leiserson, R.L. Rivest: "Introduction to Algorithms", McGraw-Hill, New York, 1990.