Proceedings of the First Southern Symposium on Computing
The University of Southern Mississippi, December 4-5, 1998

EFFICIENT SPATIAL DATA MANAGEMENT USING BALANCED AND DISTRIBUTED SEARCH TREES

ADRIANO DI PASQUALE* AND ENRICO NARDELLI†

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, 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.

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

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 [2] were the first to present and to discuss for this environment a data structure paradigm called SDDS (*Scalable Distribuited Data Structure*). The main properties of SDDS paradigm are:

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* [2], RP* [3], DRT [4], lazy k-d-tree [6, 8], RBST [7].

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.

* Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia.
† Dipartimento di Matematica Pura ed Applicata, Univ. of L'Aquila, Via Vetoio, Coppito, I-67010 L'Aquila, Italia, (nardelli@merlino.iasi.rm.cnr.it). Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Viale Manzoni 30, I-00185 Roma, Italia.

The theoretical study of the characteristics of scalable distributed search trees conducted by Kröll e Widmayer [9] 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.

In the RBST [7] some of these hypothesis, related to the way the search proces is exectued, 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 dinamic and order-preserving.

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* [9]. 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 [7] we 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.

3. 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.

To be more precise, we manage in different ways the two associations. Servers manage both node and leaf associations, while clients manage 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.

4. 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. 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 father node $pn(t)$ of t, if t is not the root node,
- The server $l(s)$ (resp., $r(s)$) managing the left (resp., right) son $ls(t)$ (resp., $rs(t)$) 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 4.1). Since in a global tree of

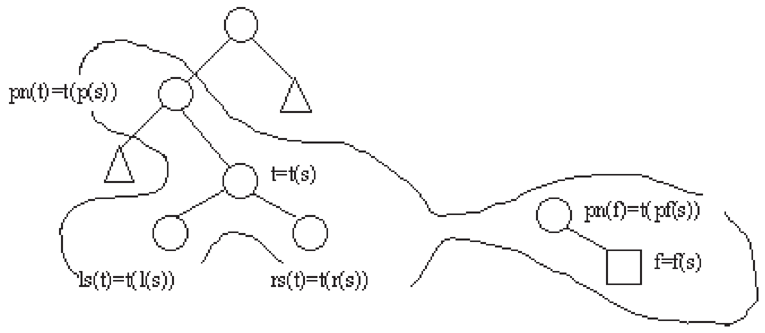

FIG. 4.1. *The local tree of server s.*

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 $k \in I(n(s))$. Note that the server managing the root is logically pertinent for each key.

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

5. 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 root r 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 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 5.1 (left).

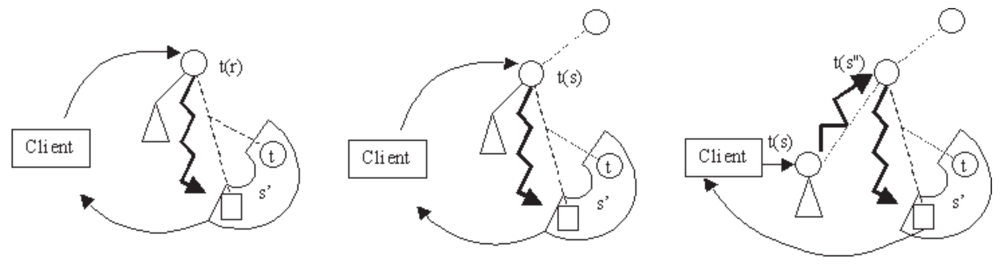

FIG. 5.1. *Searching queries from a new client (left) and from a client with addressing error (center and right).*

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 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 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 5.1 (center).

If s is not logically pertinent 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 5.1 (right).

THEOREM 5.1. *Let T be a BDST and let h denote its height. 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 5.2). In total we need 2h+2 messages. ☐

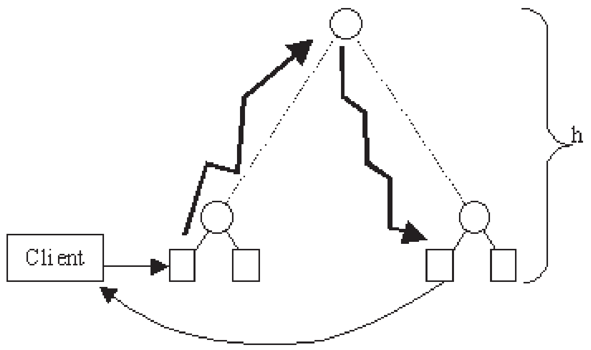

FIG. 5.2. *The worst case for searching.*

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

6. 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 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, 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 perform 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 $O(\log n)$.

6.1. 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 s must perform a function called *split*. This function is similar to the synonimous one described in [2, 4]. 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 6.1). The old interval $I(f)$ is divided in the new intervals $I(f_1)$ and $I(f_2)$, such that $I(f_1) \cup I(f_2) = I(f)$.

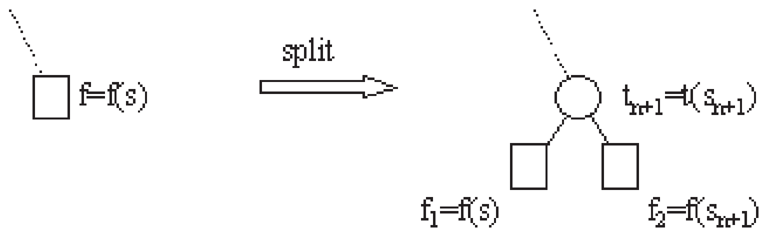

FIG. 6.1. *Insertion of an element in an overflowing bucket*

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

THEOREM 6.1. *Insertion in a BDST constituted 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 5.1 this costs $O(\log n)$ messages.
Step 2: A constant number of messages is needed to perform the split function (see [2, 4]).
Step 3: From the assumptions above we have a cost of $O(\log n)$ messages. □

6.2. 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 the server s must perform a function called *merge*. 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 $t(b)$. This function is constituted by the following sub-steps (see also figure 6.2):

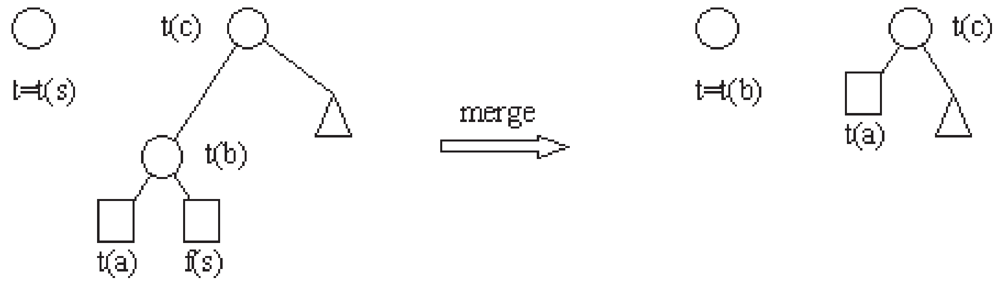

FIG. 6.2. *Deletion of an element from an underflowing bucket*

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

The new value of interval $I(t(a))$ becomes the union of the value of $I(t(a))$ before the deletion and of the value of $I(f)$.

There are two special cases: in the first case f is the root, the BDST is composed by one node and then no actions are performed. In the second case, the BDST is composed by the root r and two leaves f and x, hence there are only two servers s and s'. Then s is released and after the communication to s' and the deletion of r and that x become the root of BDST.

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

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

LEMMA 6.2. *The merge function is correct with respect to the local tree of the servers involved.*

Proof. 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. □

LEMMA 6.3. *The merge function costs $O(1)$ messages in the worst case.*

Proof. From lemma 6.2 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. □

THEOREM 6.4. *Deletion in a BDST constituted 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 5.1 this costs $O(\log n)$ messages.

Step 2: From lemma 6.3 this costs $O(1)$ messages.
Step 3: From the assumptions above we have a cost of $O(\log n)$ messages. □

7. 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 individuate the server s which should answer to a query so to issue a point-to-point message to s. If this server is not individuated, 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 information recorded in its index: for a server s of a leaf association present in its index, the client knows that s manages an 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 $I(f(s))$.

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 is a constant cost 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 realize a rotation in a distributed environment. Without loss generality, let us consider figure 8.1 (top-left), and suppose that node a must rotate with node b:

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 8.1 (top-center), to acknowledge the lock state.
4. a notifies to all servers involved in the rotation which modifications are needed and after all have been confirmed a releases all locks, see figure 8.1 (top-right).
5. When locks are released the situation is shown in figure 8.1 (bottom) and all servers restart to route messages.

It is simple 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 server involved).

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 low.

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* [10] uses a great number of rotations.

It is more convenient to use a data structure like a *red-black-tree*, 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 [1, 5], but this regards the concurrent, shared-memory case.

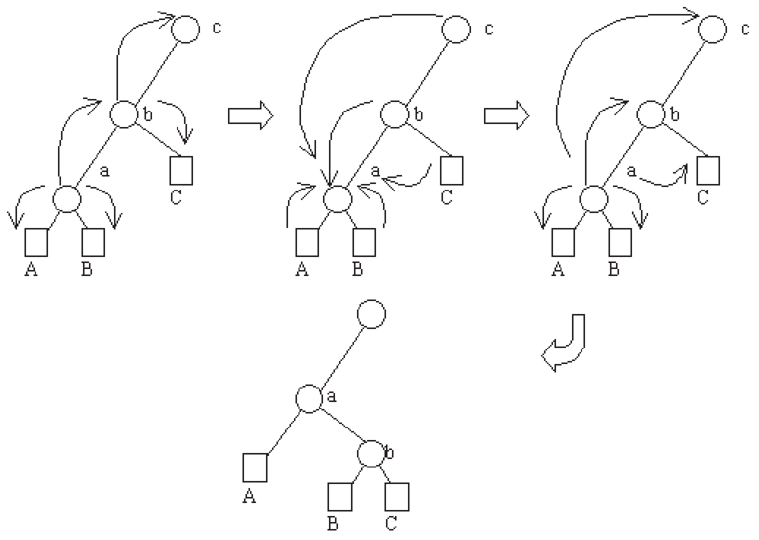

FIG. 8.1. *Locking messages during a rotation*

There is a big difference between this kind of work and the distributed tree studied here. In [1, 5] 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 balanced a distributed binary search tree, 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 implement in a message-passing distributed environment as efficiently, namely with a $O(\log n)$ worst case bound, as in the single processor case.

Acknowledgments. This research was partially supported by the European Union TMR project "Chorochronos".

REFERENCES

[1] J. ECKERLE, O. NURMI, Technical Report Aug17-7, Technical University of Munich, 1994.
[2] 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.
[3] 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.
[4] 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.
[5] 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.
[6] E. NARDELLI Distribuited k-d trees, in *XVI Int. Conf. of the Chilean Computer Scienze Society (SCCC'96)*, Valdivia, Chile, November 1996.
[7] 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.

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

[9] 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.

[10] D.D. Sleator, R.E. Tarjan. Self-Adjusting Binary Search Trees, JACM 32(3):652-686, 1985.