Studying the properties of a distributed decentralized b+ tree with weak-consistency

By

Khaled Ben Hafaiedh

A Thesis submitted to the Faculty of Graduate Studies and Postdoctoral Studies in partial
fulfillment of requirement for the degree of

Master of Applied Science
in
Electrical & Computer Engineering

Ottawa-Carleton Institute for Electrical & Computer Engineering

Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

Université d'Ottawa

Abstract

Distributed computing is very popular in the field of computer science and is widely used in web applications. In such systems, tasks and resources are partitioned among several computers so that the workload can be shared among the different computers in the network, in contrast to systems using a single server computer. Distributed system designs are used for many practical reasons and are often found to be more scalable, robust and suitable for many applications.

The aim of this thesis is to study the properties of a distributed tree data-structure that allow searches, insertions and deletions of data elements. In particular, the b- tree structure *[13]* is considered, which is a generalization of a binary search tree. The study consists of analyzing the effect of distributing such a tree among several computers and investigates the behavior of such structure over a long period of time by growing the network of computers supporting the tree, while the state of the structure is instantly updated as insertions and deletions operations are performed. It also attempts to validate the necessary and sufficient invariants of the b-tree-structure that guarantee the correctness of the search operations.

A simulation study is also conducted to verify the validity of such distributed data-structure and the performance of the algorithm that implements it. Finally, a discussion is provided in the end of the thesis to compare the performance of the system design with other distributed tree structure designs.

Dedication

To the four most loved people in my life

My parents and my two sisters. Without you, my life would fall apart.

Dad, thank you for all that force of love and success for your family and children that you want to them more than you want it to you , thank you for teaching me to stand firm until the end, and never surrender, thank you for making me what I am today.

Mom, thank you for all the love you keep inside you, thank you for all this sweetness and tenderness that emerges from your humble soul. I always think of you.

Nahloula, thank you for your support and encouragement for what I do, I wish you all the success for your professional career, I also wish you much love and happiness with your husband Mourad and for your family career.

Nounou, thank you for always being there for me when I need it, you've always took care of me since we have been living under the same roof, your presence is very important to me. I wish you all the best lovely sister

Acknowledgments

I would like to express gratitude toward my supervisor, Dr Gregor Bochmann, for his academic support during my study, and for his helpful suggestions and advices. This thesis would not be possible without his guidance. I would like to express my gratitude also for his financial support during the school session.

My gratitude also goes to Mr Asaduzzaman for the paper that inspired this thesis and for his time during the school session. His guidance is much appreciated. I would also like to thank all the people at the University of Ottawa I have had the pleasure to meet.

The most special thanks go to my parents and my two sisters who are always there for me and pushing me to work hard for my education.

Finally, a special thanks to all my friends in Tunisia, North Africa, Europe and America for their encouragement.

Table of contents

Abstract.....2

Dedication.....3

Acknowledgments.....4

Table of contents.....5

List of Figures.....9

Chapter 1. Introduction.....12

1.1 Context of the Thesis.....12

1.2 Contribution.....14

1.3 Thesis organization.....16

Chapter 2. B-tree structure.....17

2.1 System model.....17

2.2 Assumptions.....19

2.3 Possible operations.....19

2.3.1 Key search.....19

2.3.2 Range search.....21

2.3.3 Key insertion.....22

2.3.4 Key deletion.....22

2.3.5 Update operations.....22

2.3.5.1 Split operation.....23

2.3.5.2 Merge operation.....26

Chapter 3 Decentralized b-tree……..…….....28
3.1 Motivation….....28
3.2 Overview of Strong-consistency models.....30
3.3 System model of “consistent decentralized b+ trees”…….....33
3.4 Assumptions.....35
3.5 Search in the consistent decentralized b-tree.....36
3.5.1 Key search…….....36
3.5.2 Range search…….....37
3.6 Updates in the consistent decentralized b-tree.....39
Chapter 4 Decentralized b+ trees with weak-consistency…….....40
4.1 Weak-consistency invariants……40
4.2 Split and merge operations with weak-consistency……..……42
4.2.1 Split leaf node.....43
4.2.2 Split non-leaf node.....44
4.2.3 Merge of two leaf nodes….....48
4.2.4 Merge of two non-leaf nodes….....49
4.3 Challenges…..……..……50
Chapter 5. Studying the validity of the b+ tree with weak-consistency 51
5.1 Purpose of modeling the decentralized b+ tree with weak-consistency.....51
5.2 Introduction to the Alloy specification language…….....52
5.3 Modeling the decentralized b- tree.....56
5.3.1 System model.....56
5.3.2 Model instance creation.....63
5.3.3 Search operation…….....67

5.4 Weak-consistency invariants……73
5.4.1 Formalization……73
5.4.2 Inconsistent b- tree structure ……75
5.5 Conclusions and limitations……78
Chapter 6 Revised updates with weak-consistency ……79
6.1 Transfer of responsibility mechanisms……79
6.1 .1 Ping-Pong strategy……79
6.1.2 Ping-Only strategy……84
6.2. Modified version of the split and merge algorithms……87
6.2.1 Split with weak-consistency……87
6.2.1.1 Leaf node split……87
6.2.1.2 Non-leaf node split……88
6.2.2 Merge operation……90
6.2.2.1 Leaf node merge……90
6.2.2.2 Non-leaf node merge……92
Chapter 7 Simulation of the b+ tree with weak-consistency……93
7.1 Programming environment……94
7.2 Design of the b+ tree simulation system……98
7.3 Implementation choices……113
7.3.1 Balancing the Ranges ……113
7.3.2 Node/Entry trade-off……115
7.3.3 Adapting the program to a real P2P system……115
Chapter 8. Results of the simulation……116
8.1 Growth of the distributed decentralized b+ tree……116
8.1.1 The depth of the b+ tree……117
8.1.2 Mean number of back-pointers……118
8.2 Comparing the exploration mechanisms……120

8.2.1 Comparing the message complexities……121
8.2.2 Comparing the order of traversal……..123
8.2.3 Comparing the levels of the suitable nodes found……125
8.3 Stationary system state with insertion and deletion periods……127
8.3.1 Distribution of empty nodes……..128
8.3.2 Distribution of the number of back-pointers……129
8.3.3 Distribution of the order of traversals……..131
8.3.4 Distribution of the number of entries in nodes……...133
8.4 Uneven key insertions……..135
8.4.1 Distribution of empty nodes……135
8.4.2 Distribution of the number of back-pointers……137
8.4.3 Distribution of the order of traversals……138

8.5 Conclusion……..140

9. Conclusion of the thesis.....141

References……..…144

List of Figures

Fig. 2.1. B+ tree structure with 11 leaf nodes (a-k)……18
Fig. 2.3.1. Search operation for the key value 93 on the b+ tree structure of Fig. 2.1……20
Fig. 2.3.2. Search operation for the range [90,100] on the b+ tree structure of Fig. 2.1……21
Fig. 2.3.5.1.a. Before the split operation of node X in a b+ tree of order 2……24
Fig. 2.3.5.1.b. After the split operation of node X of Fig. 2.3.5.1.a……25
Fig. 2.3.5.2.a. Before the merge operation of node X1 and X2 into X1……26
Fig. 2.3.5.2.b. After the merge operation of node X1 and X2 into X1……27
Fig. 3.1. Centralized distributed b+ tree structure……28
Fig. 3.2.1. Node-wise decentralized distributed b+ tree structure ……30
Fig. 3.2.2. Scalable distributed b+ tree and the BigTable structures ……31
Fig. 3.2.3. Duplicated distributed b+ tree structure……32
Fig. 3.2.4. Example of assigning one branch to one peer in a consistent decentralized b+ tree……33
Fig. 3.3. The view of the tree from peer a and its corresponding routing table……34
Fig.3.4.. Consistent decentralized b+ tree of Fig. 2.1……35
Fig. 3.5.1. Search for the key 100 initiated by peer a……37
Fig. 3.5.2. Search for the range [90, 100) initiated by peer a……38
Alg. 4.2.1. Leaf node split algorithm as introduced in [1]……43
Alg. 4.2.2. Non-leaf node split algorithm as introduced in [1]……45
Fig. 4.2. Evolution of a weak-consistent b+ tree with asynchronous updates……46
Alg. 4.2.3. Leaf node merge algorithm introduced in [1]……48
Alg. 4.2.4. Non-leaf node merge algorithm as introduced in [1]……49
Alg. 5.3.1.1.The signatures, i.e. classes used to represent the b+ tree structure……56
Alg. 5.3.1.2. Statements for ordering the routing tables, the levels and the search states……58
Alg. 5.3.2. The facts, i.e. restrictions, used to represent the b+ tree structure……59
Alg. 5.3.4. Running an example of the distributed b+ tree structure with two routing tables……63

Fig. 5.3.2.1. Example of the distributed b- tree system with 2 routing tables, 6 entries, 4 levels, 4 nodes and 4 pointers (magic layout view)……64
Alg. 5.3.2.5. Restrictions used to balance the distributed b-tree nodes……68

Alg. 5.3.2.6. The predicate executing the latter search operation……..……..……..69
Fig. 5.3.2.2. State0 of the look up operation on the distributed b+ tree (magic layout view)..…71
Fig. 5.3.2.3. State 1 of the look up operation on the distributed b+ tree (magic layout view)….72
Assert 5.4.1.1. Assertion to verify the invariant of universe……..……73
Assert. 5.4.1.2. Assertion to check the navigability invariant…….……74
Assert. 5.4.1.3. Assertion to check the Invariant of disjoint local range……..75
Alg. 5.4.2. Fact removed from the constraints of the b+ tree structure…….……....76
Fig. 5.4.2. Inconsistent b+ tree structure……..76
Alg. 6.1.1. Ping-Pong algorithm……..……...80
Fig. 6.1.1.a. Ping-Pong method of order 1 for transferring the range [25, 50] in RT_A^l……82
Fig. 6.1.1.b. Ping-Pong method of order 2 for transferring the range [40, 60] of RT_A^l……83
Alg. 6.1.2. Ping-Only algorithm……..……84
Fig. 6.1.2. Ping-Only method for transferring the range [40, 60] from RT_A^l…….……86
Alg. 6.2.1.2. Revised non-leaf node split algorithm……..89
Alg. 6.2.2.1. Revised leaf node Merge algorithm……91
Alg. 6.2.2.2. Revised non-leaf node Merge algorithm……92
Alg. 7.1.1. Methods used by the simulated processes to implement the interface Process…....95
Alg. 7.1.2. Methods used by the class Sim……...96
Fig.7.2.1. Interaction between the main classes in the Simulation environment……98
Fig. 7.2.2. Complete distributed decentralized b+ tree Class diagram……..……...100
Alg.7.2.1. Methods used by peers to implement the interface ServiceInt……..……...101
Alg. 7.2.2. Attributes and methods used by message senders to implement the interface Process……103
Alg. 7.2.3. Attributes and methods used by message receivers to implement the interface Process……104
Alg. 7.2.4. Attributes and methods used by the message updater to implement the interface Process……106
Alg. 7.2.5. Attributes and methods used by the user to implement the interface ServiceCallBack…….……109
Fig.8.1.1. Growth of the distributed decentralized b+ tree structure….……..….117
Fig.8.1.2. Effect of growing the distributed decentralized b+ tree on the mean number of back-pointers……..…….……..118
Fig 8.2.1. Comparing the message complexities when the b+ tree is growing…….……...121

Fig 8.2.2. Comparing the order for traversal by Ping-Pong algorithm and Ping-Onlyalgorithm……..……..……..123
Fig 8.2.3. Comparing the level distribution of non-empty nodes found by Ping-Pong algorithm and Ping-Only algorithm……..……125
Fig.8.3.1. Distribution of the number of empty nodes per peer in the stationary system state.....128
Fig.8.3.2. Distribution of the number of back-pointers per non-empty node in the stationary System state…… ……129
Fig.8.3.3. Distribution of the order of traversal in the stationary system state……..……131
Fig.8.3.4. Distribution of entries per non-empty node in the stationary system state……..…..133
Fig.8.4.1. Distribution of the number of empty nodes per peer for random insertions scenario and specific key insertions scenario……..……135
Fig.8.4.2. Distribution of the number of back-pointers per non-empty node for random insertions scenario and specific key insertions scenario……137
Fig.8.4.3. Distribution of the order of traversal for random insertions scenario and specific key insertions scenario……..138

1. Introduction

1.1Context of the thesis

Cloud computing typically operates over a huge number of computers or interconnected networks and requires some decentralized approaches to overcome the traffic overflow that may occur when permanently accessing a certain number of processing units more often than others. Choosing the appropriate decentralized indexing structure for data organization is crucial in the modern communication systems such that the high volume of dynamic data is efficiently accessed and updated without any performance bottleneck.

We consider the b-tree structure *[13]* which is a tree data-structure that keeps data sorted and allows searches, sequential access, insertions, and deletions. The b-tree consists of a root node, branch nodes and leaf nodes. The latter contains the indexed data values associated with a key. Each node has a routing table that administers its communication with the other nodes in the tree. Accordingly, each routing table maintains a set of entries where each entry describes a specific range of key values and a link to another node in the tree. The b-tree can be centralized i.e. implemented on one peer where all communications are routed through one central hub, or can also be distributed on many peers who make it more efficient for data organization and retrieval. Using a b-tree structure has many advantages since it is well-understood and has a straightforward mechanism for updating the nodes of the trees for data insertions and deletions as well as handling unbalanced loads. However, applying such structure on a massive-scale computing system may be problematic due to the high traffic that may occur for some nodes, which reduces the efficiency of the system. Indeed, the intuitive method for distributing the b-tree data structure is to allocate one or more neighboring nodes to one peer. Although, this allows the update algorithms on the structure for data insertion/deletion to be similar to the centralized version, the peers holding the root may get over-loaded with the high traffic, while other peers may remain idle for a long period of time.

The discussion on the cloud computing research agenda *[2]* has identified the overhead and the high cost of the typical distributed system models with strong-consistency as one of the biggest concerns in cloud computing. Therefore, it is preferable to design distributed data structures that can allow some degree of inconsistency but still work properly. This study consists of designing a distributed implementation of the b-tree data structure *[13]* that works with weak-consistency among its replicated nodes but provides strong-consistency in terms of search semantics.

1.2 Contribution

The description of the alternative distribution of the b-tree structure*[1]* consists of replicating the higher level tree nodes in proportion to their usage. So, instead of assigning the responsibility of one node to one peer, one branch of the tree, i.e. the path from the root to a leaf node is assigned to one peer. As the data structure needs to be updated when keys are inserted or deleted to keep the workload balanced among the peers, the b-tree data structure grows with key-insertions by splitting a node when the number of entries overflows, and shrinks with key-deletions by merging two sibling nodes. Hence, the main goal of this work is to practically prove that such structure guaranties that the workload is equally distributed among the peers during these updates operations even for a larger-scale distributed network.

We start by expressing the required properties of the decentralized b-tree with weak-consistency *[1]* using the Alloy modeling language *[10]* in order to validate the weak-consistency properties that are necessary and sufficient for maintaining the b-tree structure suitable for the search operation. This validation explicitly proved that it is possible to construct the distributed b-tree by just invoking the core facts and certain constraints that keeps the search operations performing properly in all situations.

In addition, the major challenge of the update algorithm under the week consistency conditions *[1]* is to find an existing peer whose routing table already contains some entries covering some common range of key values. Our contribution consists of defining several algorithms which show that finding another peer responsible for the requested range is feasible.

We also found that there is a dependency between the number of entries and nodes in each routing table when one branch of the tree, i.e. the path from the root to a leaf node is assigned to one peer in the distributed b-tree structure.

Therefore, a study is conducted to optimize the system so that the nodes and entries remain equally distributed and balanced among the b-tree. Studying the characteristics of such approaches let us determine which settings should be used to reach the highest performance of the distributed b- tree with weak-consistency in terms of number of exchanged messages and execution time.

This study showed also that the depth, i.e. the minimum number of traversed nodes by a search query from the root to the leaf nodes of the tree, obtained after a long period of tree update operations can be maintained at the optimal level of $L = \log_p(N)$ where N is the number of peers in the system and p is the number of entries in each node. We were also able to determine the average number of peers involved in a single update operation, and perform a more thorough comparison between the two decentralized b-tree organizations with full and weak-consistency, to determine the performances in the two settings in terms of time and message complexity.

1.3 Thesis organization

The thesis is organized as follows: Chapter 2 gives a brief overview of the b-tree structure and its possible operations, Chapter 3 introduces the decentralized b-tree with strong-consistency, and it also illustrates the system model, its assumptions and the updates under these strong-consistency conditions. A discussion is then conducted to estimate how much consistency is needed and sufficient for correct search and update operations on the distributed b+ tree structure. These weak-consistency constrains are also introduced. Chapter 4 gives a brief introduction to decentralized b+ tree with weak-consistency and the split and merges update operations. In Chapter 5, we validate the weak-consistency properties by modeling the decentralized b+ tree with weak-consistency using the Alloy modeling language. Chapter 6 describes the revised updates under the weak-consistency conditions and the modified version of the split and merges update algorithms. Chapter 7 exposes our conducted simulation of the proposed b+ tree with weak-consistency. We first describe our system design and its working mechanism; we then discuss the implementation choices that were used for the simulation. In chapter 8, we present the results of the simulation by comparing the efficiency of the exploration mechanisms and studying the behavior of such structure for some particular situations. A discussion is also provided to evaluate the system performance and efficiency.

2. B-tree structure

2.1 System model

The typical purpose of a b-tree is storing entire records from a database system and serving as an in-memory representation of the collection of records, as well as storing indices of the collection of records in sorted files. The modernization of the communication systems gave birth to new applications of the b-tree structure such as fast accessing, reading and writing of disk blocks. Indeed, b-trees have been used for storing data elements identified by keys for efficient retrieval in file systems where data elements are stored in an array and individual data elements can be selected by an index that is usually a non-negative scalar integer. The b-tree algorithm marginalizes the number of times a medium is accessed to locate a desired data element, thereby speeding up the process. A b-tree in certain aspects is a generalization of a binary search tree *[16].* The main difference is that nodes of a b-tree may have more than two children rather than being limited to only two. Using a larger fan-out, i.e. having many tree branches at each node, reduces the height of the tree and thus speeds up the search operation whenever we are trying to locate records.

We consider the b+ tree variant of the b-tree *[13],* which is possibly the most widely used variant of the b-tree structure. A b+ tree is a height-balanced search tree that represents sorted data stored only in the leaf nodes in a way that allows for efficient insertion and removal of data elements. In this section, we assume a centralized b+ tree structure where all search and update operations are routed through one single hub: the root of the tree. Each tree-node comprises a set of entries that gives more information about a given record while the keys and records are stored in the leaf nodes of the b+ tree.

This makes it easy to maintain accurately updated lists of data that can be accessed from the root. Such architecture imposes a certain boundary restrictions to maintain the b+ tree balanced which requires maximum and minimum bounds on the number of entries in each node of the tree. The figure below shows an example of a b+ tree structure:

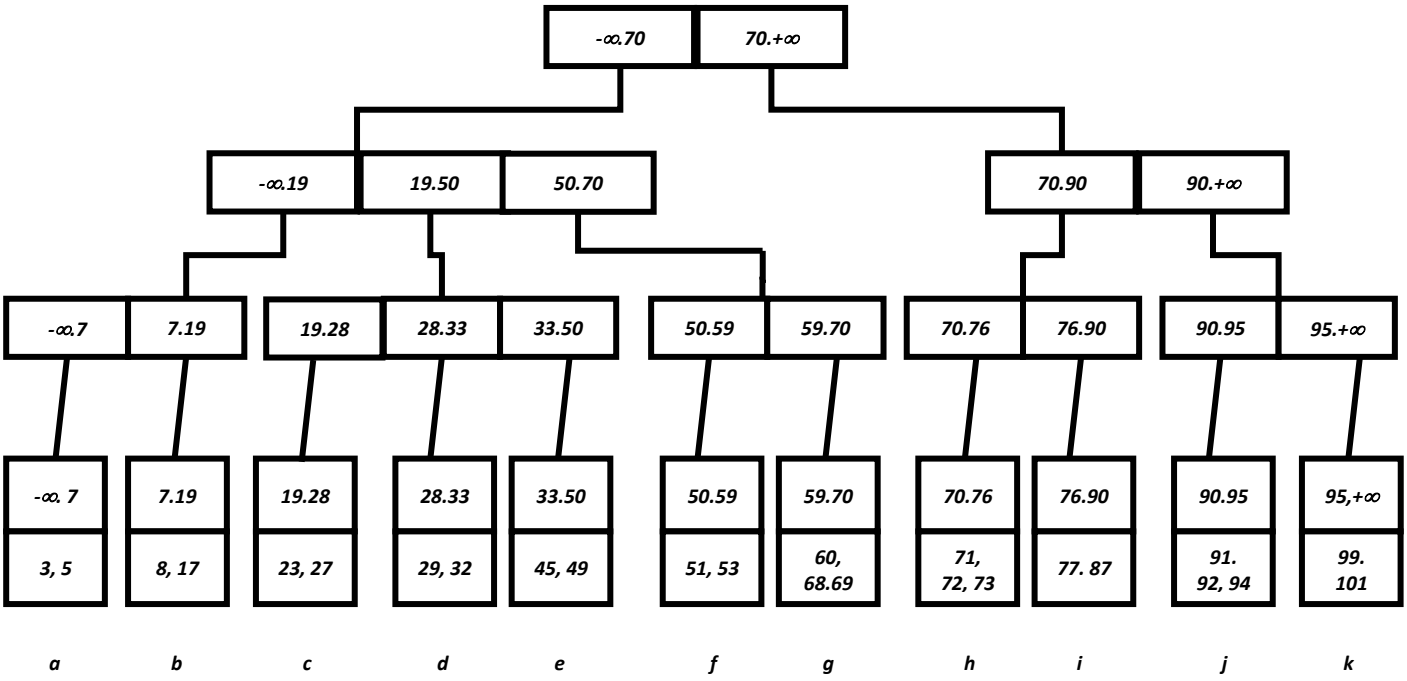


Fig. 2.1. B+ tree structure with 11 leaf nodes (a-k)

In the b+ tree structure, all nodes except the leaf nodes have the same structure. Each non-leaf node *i* in the tree maintains a routing table RT_i^l , also called a *node*, and each node comprises a set of entries. Each entry is a tuple <*r,j*> where *r* is a range of key values and *j* is a pointer to a child node in the tree. Each leaf node contains a single range, called the local range of the leaf-node, and holds the associated records with key values within the range.

2.2 Assumptions

The b-tree is designed for indexing one-dimensional data spaces. So, the ranges holding the key values are expressed by non-negative integer numbers and key values are stored in the nodes in non-decreasing order, i.e. sorted in lexicographical order, in contrast to the generalization of b-tree for key-spaces of arbitrary dimensions where the data spaces are expressed by lines or rectangles for two-dimensions as in Quad-tree *[7]* or R-tree *[8].*

For disambiguation, we denote the union of all the ranges of a non-leaf node *i* by $range(i)$. LR_i refers to the single range held by a leaf node *i*. We assume a perfect successiveness of the ranges in the entries contained in each non-leaf node, i.e. for each two successive entries in a given non-leaf node, their corresponding ranges must be successive without any gap. Additionally, the root node of the b+ tree describes the universe of key values, denoted *U*, i.e. the union of all key values in the b+ tree, and each non-leaf node other than the root maintains a sub-range of *U*. The size of this sub-range is reduced as we move to a lower level in the b+ tree. Thus, the lowest level describes the local key range of the stored data. Moreover, we assume that each child node *i* in the tree knows his parent node p by means of a back-pointer denoted BK_i^l.

2.3 Possible operations

The algorithms for the search insert and delete operations as well as the data update operations are described below:

2.3.1 Key search

The key search operation consists of finding the data associated with a given key value. Starting at the root, the tree is recursively traversed from top to bottom by sending the

search query from a parent node *i*, having an entry <*r,j*> whose range *r* includes the requested key, to the routing table (node) j pointed to by the routing table entry . This procedure is repeated until a leaf node is reached and the corresponding data element is then retrieved and the search operation is terminated.

Figure 2.3.1 illustrates an example of a search operation for the key value *100* performed on the b+ tree illustrated by *Fig 2.1.* Starting from the root of the b+ tree *(1),* the root localizes the entries that includes the key value *100* and sends the search query to its child node using the link associated with the range *[70,+∞] (2).* Similarly, this node localizes the entry *[90, +∞]* including the key value *100* and forwards the query to its child node associated with the latter range. The same procedure is performed in *(3)* and the non-leaf node directs the query to the leaf node k by means of the link associated with the range *[95, +∞]* which includes the key value *100 (4).* Finally, the data associated with the key *100* in the leaf node is retrieved and the search operation is terminated.

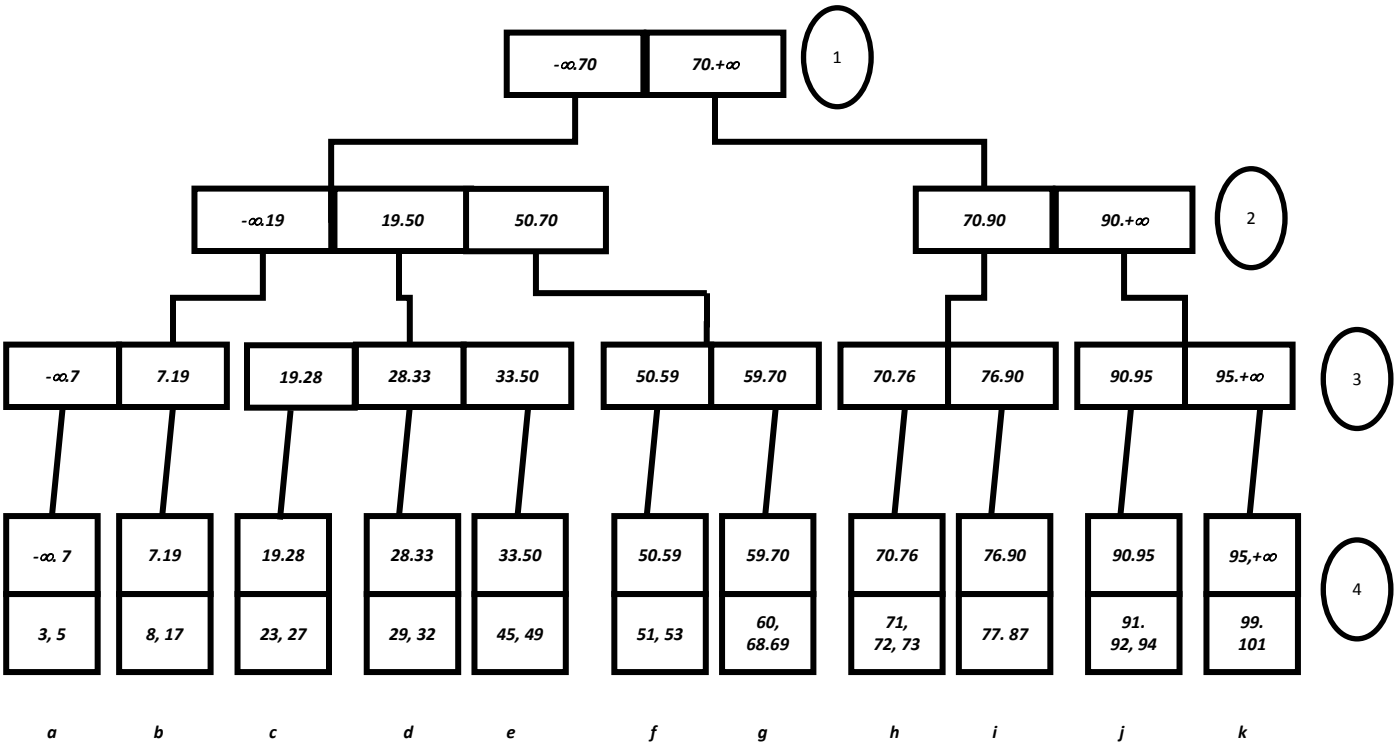


Fig. 2.3.1. Search operation for the key value 100 on the b+ tree structure of Fig. 2.1

2.3.2 Range search

In the case of a range search, by analogy to search key, the operation is performed by subdividing the range of each non-leaf node containing the requested range into sub-ranges to include each entry $<r_k,j_k>$ accordingly. The query is thus redirected to the corresponding child *jk* that hold a sub-range r_k. The navigation proceeds to the next child nodes, in parallel, by pointing at each hop of the navigation from a parent nodes to all child nodes whose range intersect with the range of the search, until each query gets to the leaf node holding the target sub-range in its local range. *Fig. 2.3.2* describes step by step the search for the target range *[90, 100).* Starting from the top of the tree *(1),* the root directs the query to the child node associated with the entry *[70, +∞) (2).* Similarly, this node redirects the query to the corresponding child node *(3).* The latter node checks if *[90, 100)* is included in its range *[90, +∞)* and the sub-range *[90, 100)* is then partitioned into 2 distinct sub-ranges *[90,95]* and *[95,100)* to include the two entries *<[90,95], j>* and *<[95,+∞), k>* and the query is redirected to the two corresponding leaf nodes j and k accordingly, in where the corresponding key values are stored *(4).*

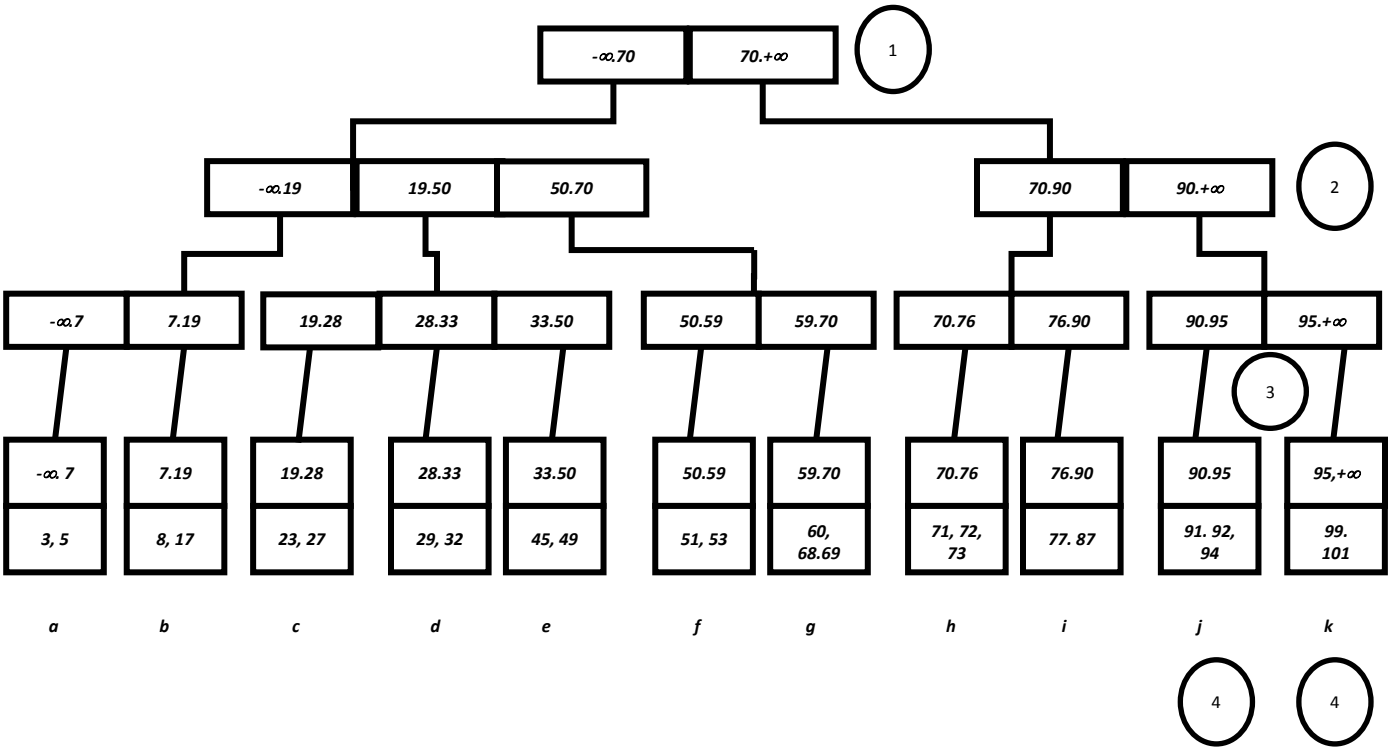


Fig. 2.3.2. Search operation for the range [90,100] on the b+ tree structure of Fig. 2.1

2.3.3 Key insertion

The insertion operation inserts a new data element associated with a given key value. The key value should not be in use prior to the insertion. The operation starts by performing the key search operation. Once the lowest level of the b+ tree is reached, the key is checked and the data is inserted. In the case when the leaf node becomes over-loaded, the node is split into two distinct leaf nodes and the keys and their corresponding data elements are accordingly distributed between the two leaf nodes in order to keep them balanced.

2.3.4 Key deletion

The delete operation consists of locating a given key value by performing the search operation, and then deleting the data element that may be associated with that key value from the corresponding leaf node. Once the delete operation is achieved, the corresponding leaf node may remain under-loaded. Thus, the leaf node may be joined with another one to form one single leaf node.

2.3.5 Update operations

While performing any insertion or deletion operation, a certain number of updates may have to be made to the b+ tree structure, contrary to the search operations that maintain the b+ tree structure unchanged. One goal of the updates is to keep the b+ tree structure as much balanced as possible so that the system can reach its highest search performance. The internal nodes may have a variable number of entries within some pre-defined limits. When data is inserted or removed from a node, the number of child nodes may change.

In order to maintain the tree balanced, adjacent internal nodes may be joined to create a single node if they are under-loaded, and one node may be split into two nodes if it is over-loaded. The lower and upper bounds on the number of children are typically fixed for a given b+ tree structure. Basically, the lower and upper bounds on the number of child nodes are parameterized by an integer number called the order of the tree and denoted by *p* referring to maximum number of entries allowed per node. A b+ tree of order *p* satisfies the following properties:

- The root node has at least *2* child nodes (if it is not a leaf node)
- All nodes other than the root have at least ceiling *(p/2)* entries, where *ceiling(x)* is the smallest integer not less than *x*.
- All non-leaf nodes have at most *p* entries.
- All leaf nodes are at the same level, i.e. have the same depth from the root.

2.3.5.1 Split operation

The split operation occurs when a leaf node or a non-leaf node is over-loaded. When a leaf node split is performed, a new node is introduced and the local range of the split leaf node is repartitioned into two distinct local ranges LR_1 and LR_2 .

The latter node keeps one of the ranges LR_1 while LR_2 is associated with the new node. In order to keep the search, delete and insert operation performing properly, the parent node i.e. the node located at the next higher level in the b+ tree and having the original leaf node as a child has to split its entry pointing to the leaf node into two entries pointing to the original and new leaf nodes. Therefore, an additional entry is introduced at this non-leaf node. This node may become over-loaded if the number of its entries is more than p and a split of this non-leaf node is performed similarly to the leaf node split operation. However, since a non-leaf node is assumed to have more than one entry, the entries are split into two sets of entries; the first set is keep within the original non-leaf node while the second set is associated with the new non-leaf node. Additionally, all the child nodes associated with the second set are allocated to the introduced node. Moreover, the entry of the parent of the original node is also split into two distinct entries and the latter node is updated as necessary.

Fig. 2.3.5.1.a and Fig. 2.3.5.1.b show an example before and after the split operation of node *X* in a b+ tree of order *2*.

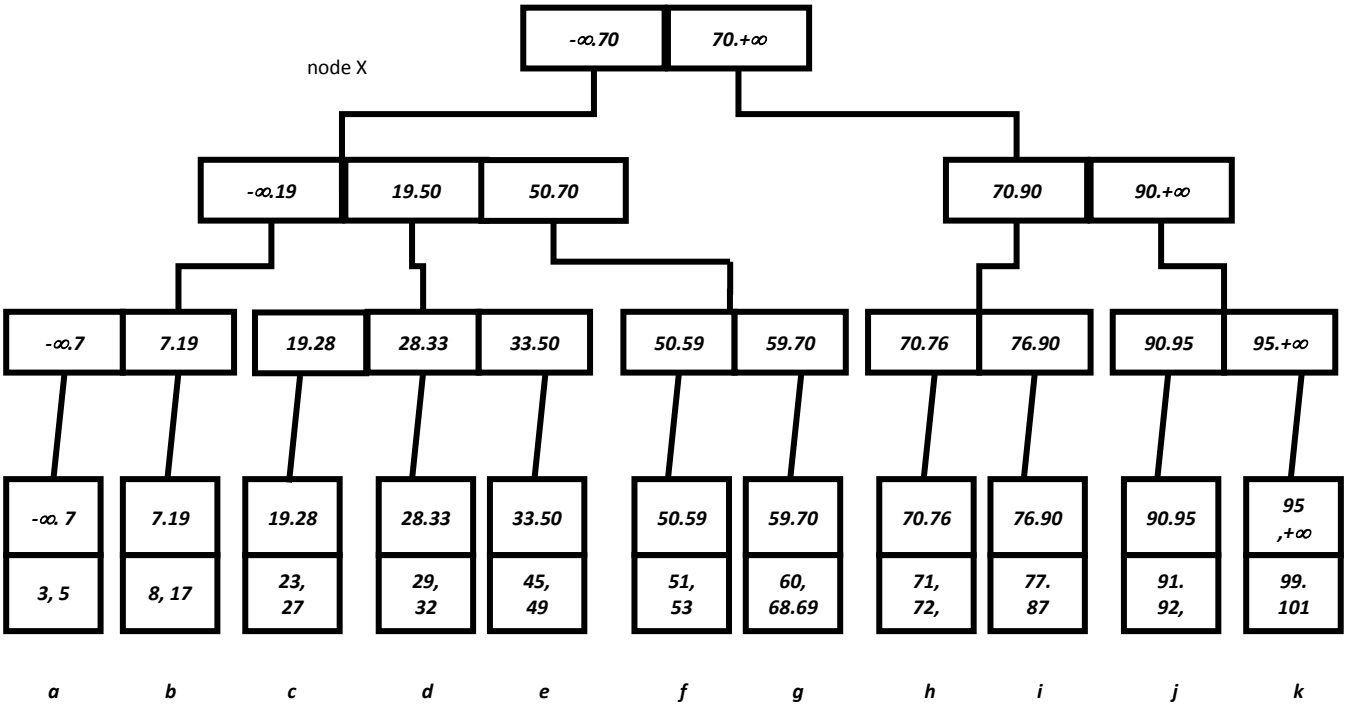


Fig. 2.3.5.1.a. Before the split operation of node X in a b+ tree of order 2

Since the node *X* has *3* entries and hence has exceeded the maximum number of entries allowed per node, a new non-leaf node is introduced and node *X* is partitioned into two distinct nodes *X1* and *X2.* The entries of node *X* are split into two sets of entries; the first set corresponding to the range *[-∞, 19]* is associated with node *X1* while the second entry having the range *[19, 70]* is associated with node *X2*. All the child nodes associated with the first set are now allocated to the node *X1* while the child nodes associated with the second set are allocated to the node *X2.* Additionally, the entry of the parent node that was associated with the split node *X* is also split into two distinct entries accordingly.

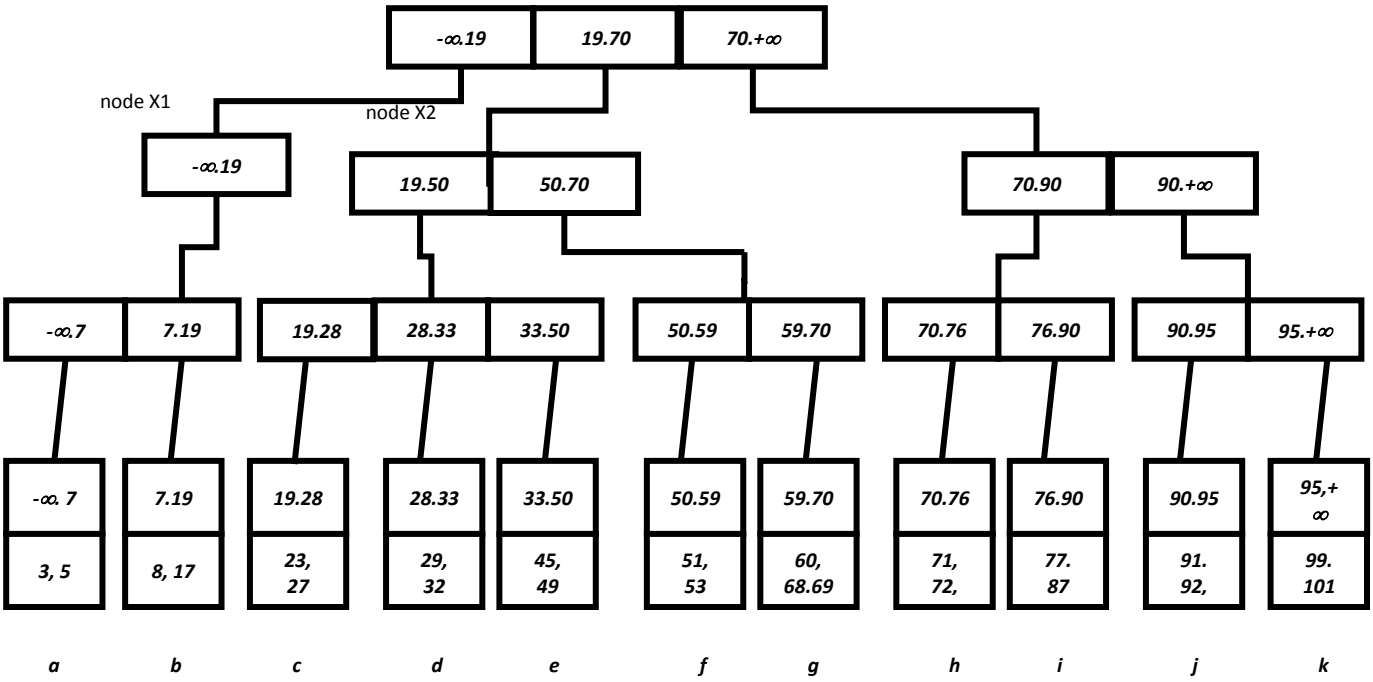


Fig. 2.3.5.1.b. After the split operation of node X of Fig. 2.3.5.1.a

2.3.5.2 Merge operation

In contrary to the split operation, the merge operation occurs when two adjacent sibling leaf nodes or non-leaf nodes become under-loaded. A leaf node merge consists of combining the local ranges of two leaf nodes LR_1 and LR_2 into a single local range that will be associated with the merged leaf node while the other leaf node becomes empty and is released. The parent node, i.e. the node located at the next higher level in the b+ tree and having the leaf nodes as children has also to merge its corresponding entries into a single one. Therefore, the number of entries comprised in the parent node is reduced and the node may become under-loaded if the number of its entries becomes less than ceiling *(p/2)*. Hence, a non-leaf node merge operation may be performed similarly to the leaf node merge operation. The same procedure is applied to the parent nodes as necessary.

Fig. 2.3.5.2.a and *Fig. 2.3.5.2.b* illustrated below show an example of a b-tree before and after the merging of nodes *X1* and *X2*.

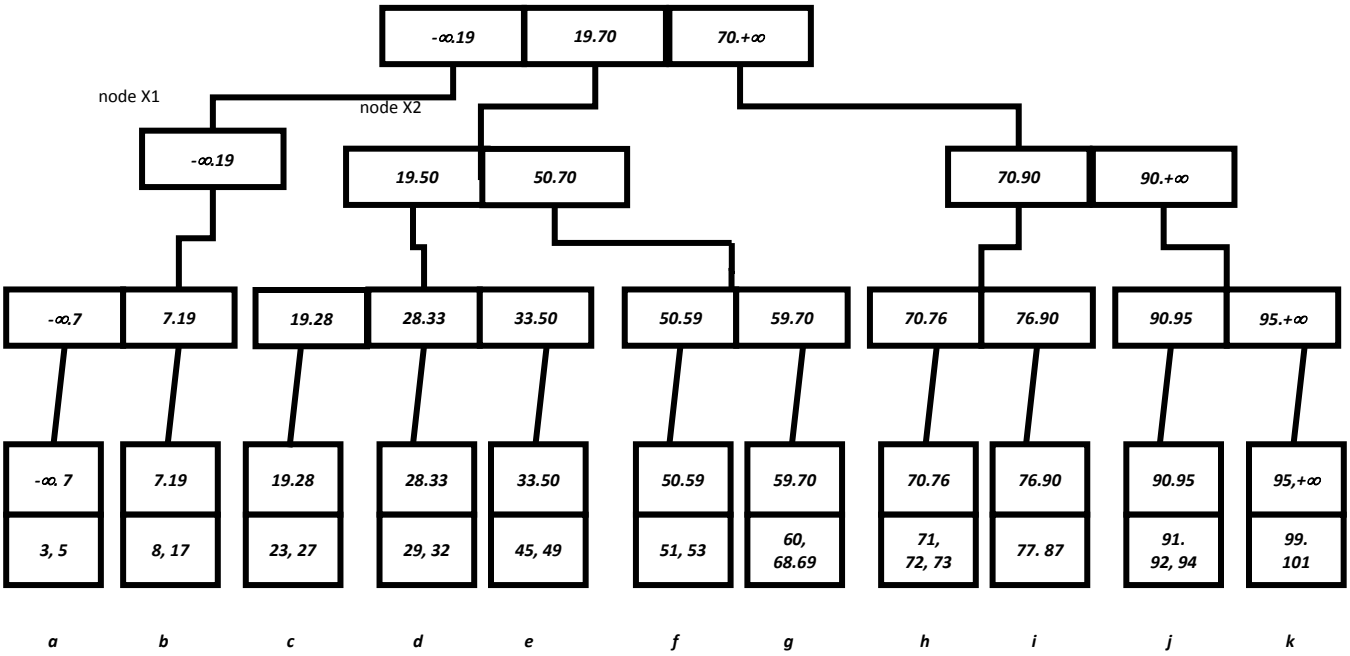


Fig. 2.3.5.2.a: Before the merge operation of node X1 and X2 into X1

The merge operation can be seen as the inverse of the split operation. The entries of node *X1* and *X2*, i.e. *[-∞, 19]* and *[19, 70],* are merged into one entry *[-∞, 70]* into node *X1*. All the child nodes associated with *X2* are now associated with node *X1* accordingly. Node *X2* is thus released. Additionally, the two entries of the parent node, which is the root, who were associated with the node *X1* and *X2*, i.e. *[-∞, 19]* and *[19, 70],* are also merged into one single entry *[-∞, 70]* and the number of entries contained in the root is reduced to *2*.

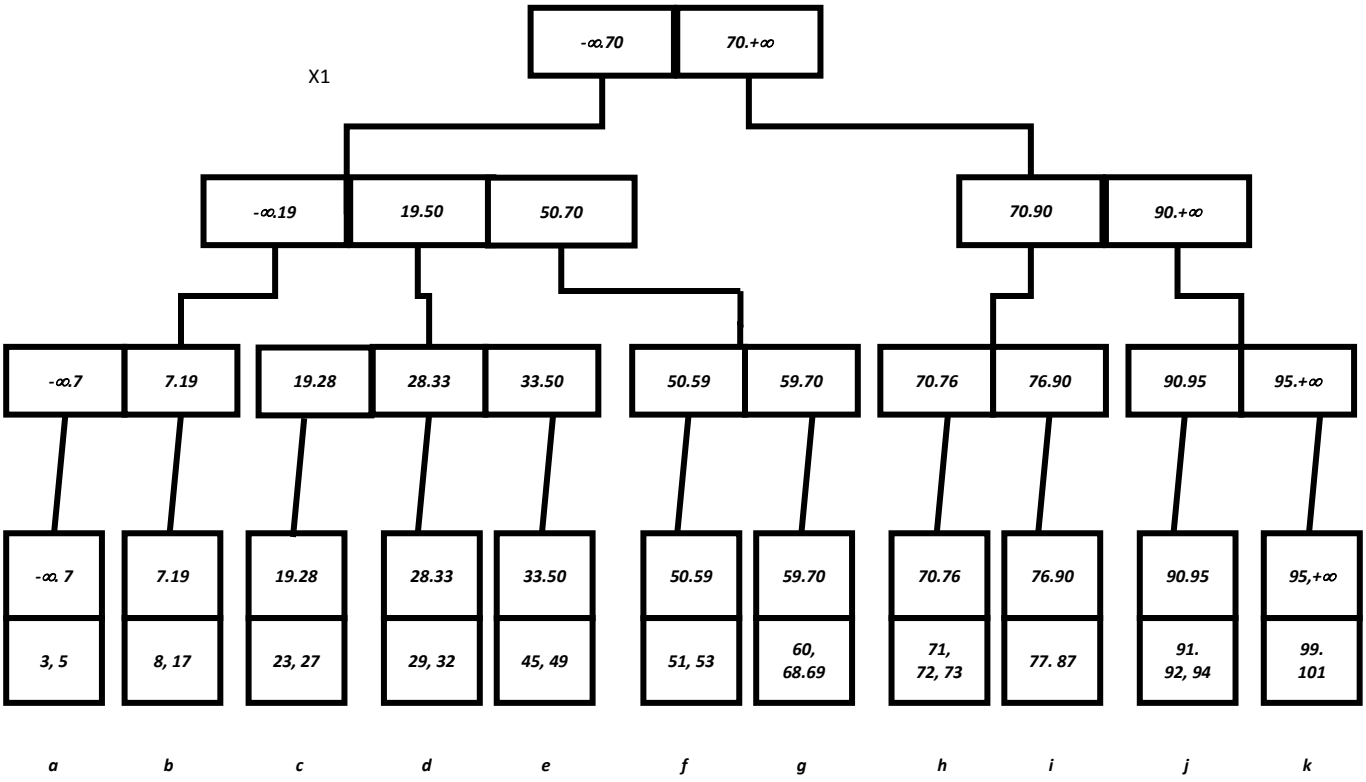


Fig. 2.3.5.2.b: After the merge operation of node X1 and X2 into X1

However, in practical communication systems, b+ trees usually have much bigger orders. In fact, a practical b+ tree system can hold millions or billions of nodes; the number of traversed nodes to reach a child node from the root, i.e. the depth of the tree, may be changed when update operations are performed. Typically, adding a large enough number of data elements will slightly increase the depth while deleting a large enough number of data elements will slightly decrease the depth. This ensures that the b-tree is maintained optimal for the number of nodes it contains.

3. Decentralized b-tree

3.1 Motivation

When a huge number of data records are indexed, for the sake of distributing the storage and access load, the data-structure is distributed on a large number of computers called peers. By saying distributed, we mean that the b-tree data structure will be dispersed over multiple interconnected peers, where the basic operations on the data structure such as search, insert, delete and load-balance are collaboratively performed. Typically, distributed systems tend to be fairly centralized; the client/server paradigm *[17]* can be used where the b+ tree is implemented on one peer (the server and all search and update operations are routed through this peer), and the storage devices are distributed among other interconnected peers (the clients). In this case, the system management resides within one administrative hub. We call this architecture a centralized distributed b+ tree.

The figure below illustrates an example of a centralized distributed b+ tree structure. The server is identified as an acting agent for all communications and must be a high capacity, high speed computer with a large hard disk capacity.

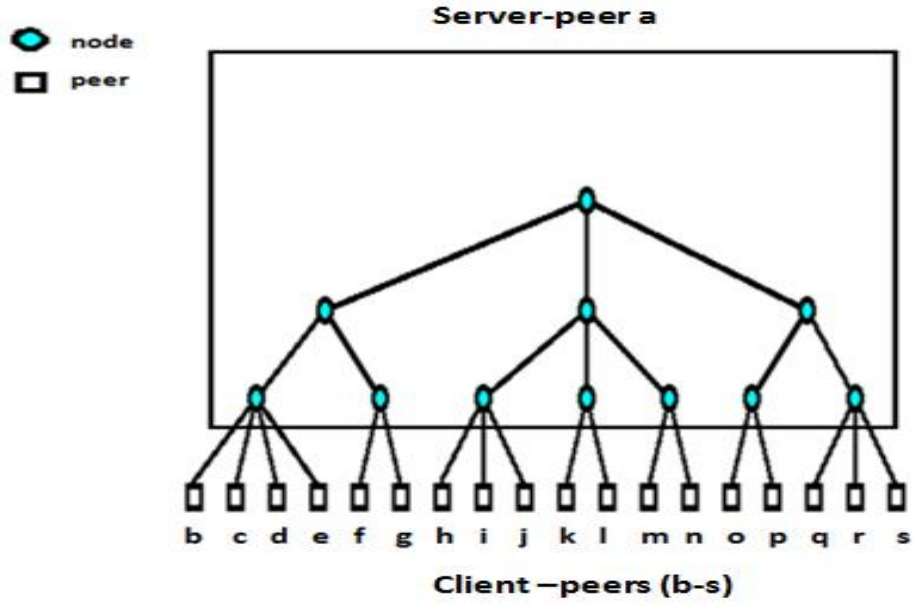


Fig. 3.1. Centralized distributed b+ tree structure

By contrast to the above centralized b-tree structure, a peer-to-peer network [12] is a decentralized distributed network in which storage devices are not all attached to a common hub, it may be dispersed over a network of interconnected peers where the system management is symmetrically distributed among all the peers. Such decentralized architecture insures that the systems control is cooperatively maintained by all the peers, unlike the centralized architecture which consists of a single controller or master-process in the system. In contrast to the traditional centralized approach, a peer-to-peer system is fully decentralized system in which each peer acts as both the client and the server. Peer-to-peer systems are found to be more scalable, robust and suitable for many applications. The term P2P is used to refer to Peer-to-peer systems. To distinguish between tree-nodes and processing nodes in the distributed b+ tree, we denote the latter as peers, while the term node refers to tree-nodes.

3.2 Overview of strong-consistency b+ tree models

An intuitive method for distributing a b+ tree within a P2P system is node-wise distribution that consists of placing each tree-node on a distinct peer. *Figure 3.2.1* shows an example of a b-tree that is node-wise distributed among peers. Nodes may be associated randomly with the peers or according to some data placement policy. For example, larger peers may be associated with the high-level nodes where higher communication traffic is expected.

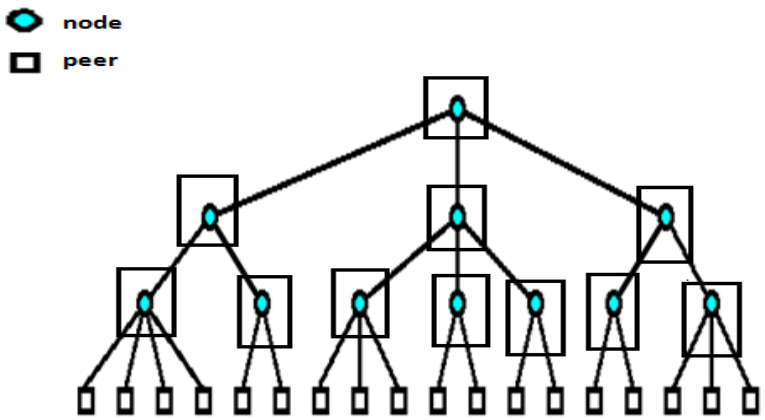


Fig. 3.2.1. Node-wise decentralized distributed b+ tree structure

Another distribution of the b+ tree structure allocates more than one node to each peer and larger peers may get more nodes than others. The scalable distributed b+ tree proposed by Aguilera et al. *[3]* and the tablet hierarchy in the internal representation of Google's BigTable *[6]* structure use such representations. Such distributed b+ tree comprises a set of peers and each peer is allocated more than one tree node. The updates of the tree-nodes are spread across multiple peers instead of a single one. For example, an Insert operation may have to split a b+ tree node, which requires modifying the node, stored on one peer, and its parent, stored possibly on a different peer.

The scalable distributed b+ tree and the BigTable structures may be illustrated as follows:

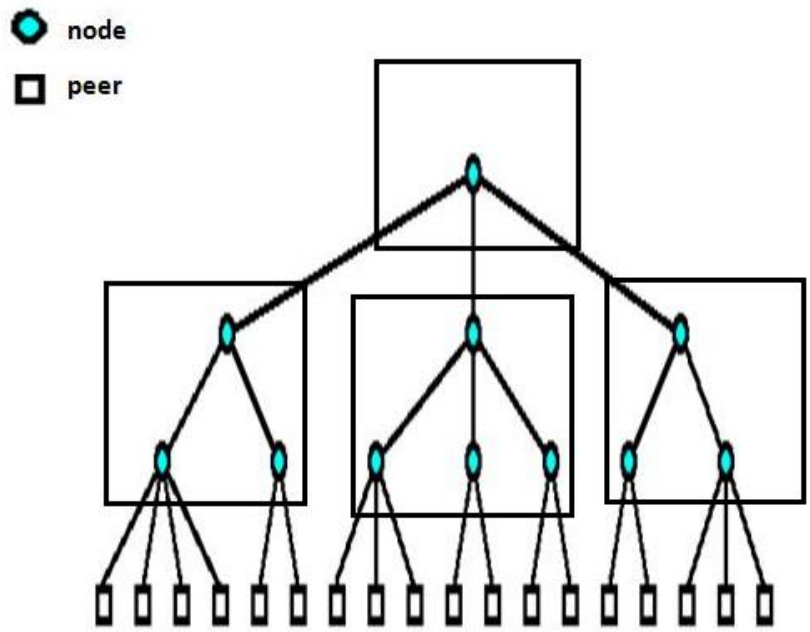


Fig. 3.2.2. Scalable distributed b+ tree and the BigTable structures

Although this allows the update algorithms on the structure for data insertion/deletion to be similar to the centralized version introduced in Section 3.1, the peers holding the root or the higher level tree-nodes get overburdened with search traffic. A typical solution to this problem, used in both *[3]* and *[6],* is caching or replicating the higher level nodes of the tree in the user or client computers, such that traversing higher level nodes can be avoided. An example of a duplicated distributed b+ tree structure is illustrated below:

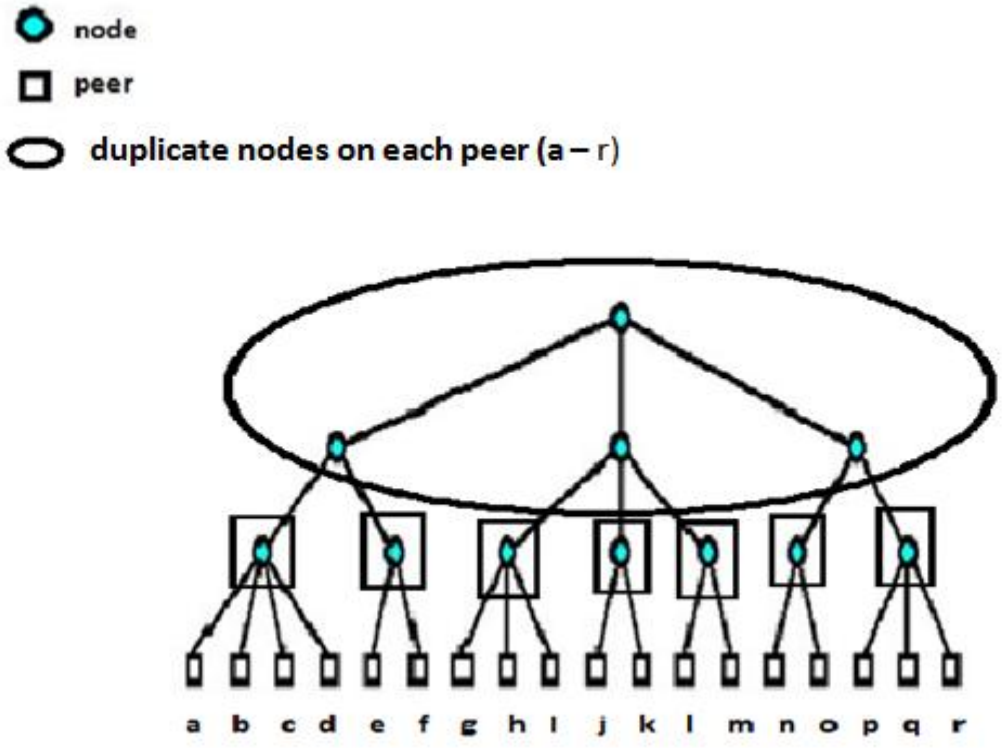


Fig. 3.2.3. Duplicated distributed b+ tree structure

However, this involves additional overhead for maintaining consistency among the replicas, and may not be suitable for highly dynamic data sets.

An alternative distribution of the tree structure is to replicate the higher level tree nodes in proportion to their usage. So, instead of assigning the responsibility of one more tree nodes to one peer, one branch of the tree, i.e. the path from the root to a leaf node, is assigned to one peer as described in *[1]*, and hence, the workload due to traversal operations is equally distributed among the peers. Thus, each peer is composed of multiple levels that are equivalent to multiple nodes of the classical b+ tree. The lowest level describes the local range of the peer in where the data is stored, similarly to a leaf node in the b+ tree structure. This way, the responsibility of each peer is prearranged symmetrically and the workload due to search and update operations on the tree structure is equally distributed among the peers.

We call such structure a consistent decentralized b+ tree. An example of the consistent decentralized b+ tree structure is illustrated below:

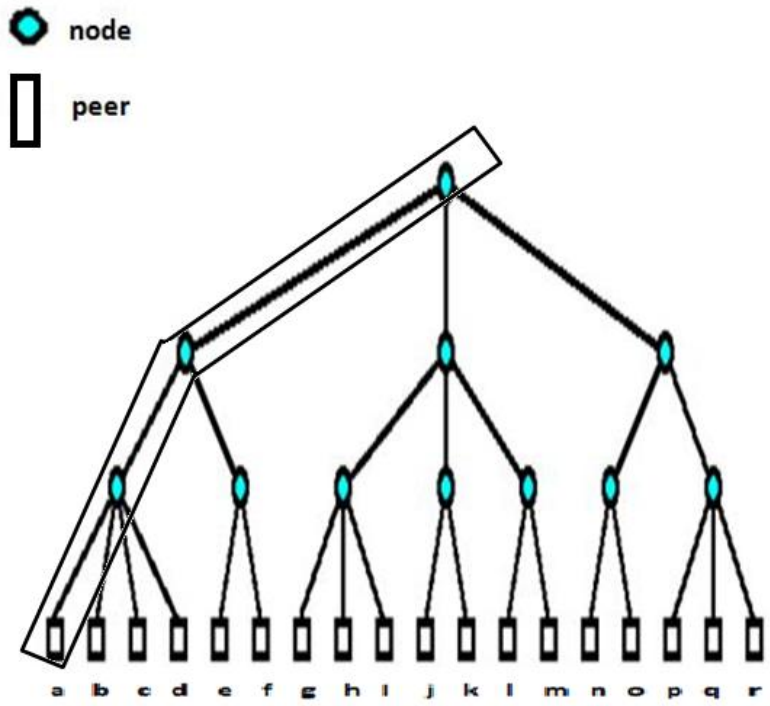


Fig. 3.2.4. Example of assigning one branch to one peer in a consistent decentralized b+ tree

3.3 System model of "consistent decentralized b+ trees"

In order to represent a branch of the tree, each *peer i* is composed of multiple levels in contrary to one single tree-node in node-wise b+ tree structure introduced in Section 2.1. RT_i^l refers to the routing table of a tree-node of the branch, at level l of the original b+ tree structure, that belongs to peer i. Notice that these entries may point to the local peer if the range of the entry includes the range of a node at the next-lower level within the same peer i.

Such range is called local view of the peer in a given level, and gives more information on the local range the peer holds among its responsibility, as we go down in each of the branches of the b-tree data structure. The lowest level RT_i^0 corresponds to a leaf node of the original distributed b+ tree, and stores the set of keys and data in the local range delegated to the branch of peer *i*. A back-pointer table is also required for updates: Each non-leaf node RT_i^l maintains a back-pointer table BK_i^l describing all its parent nodes RT_p^{l+1} pointing to node RT_i^l . *Fig.3.3* represents a consistent decentralized implementation of *peer a* in *Fig. 2.1.*

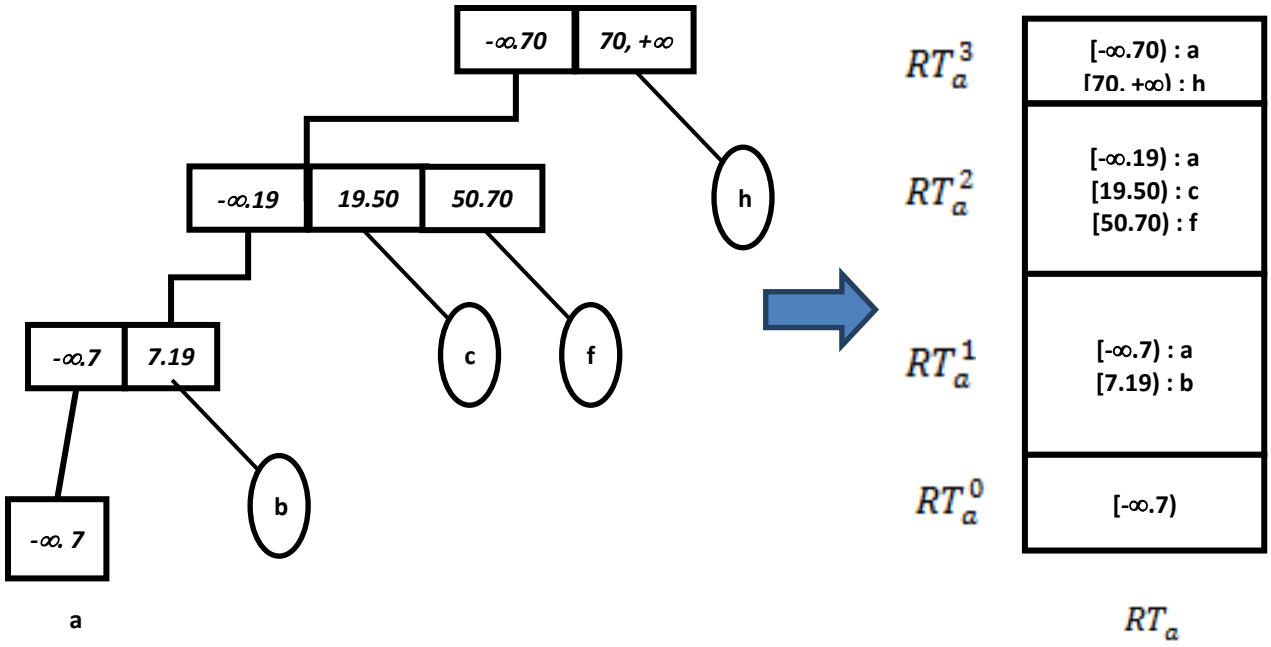


Fig. 3.3. View of the tree from peer a *and its corresponding routing table*

Fig.3.4 illustrates the consistent decentralized b+ tree of *Fig. 2.1.*

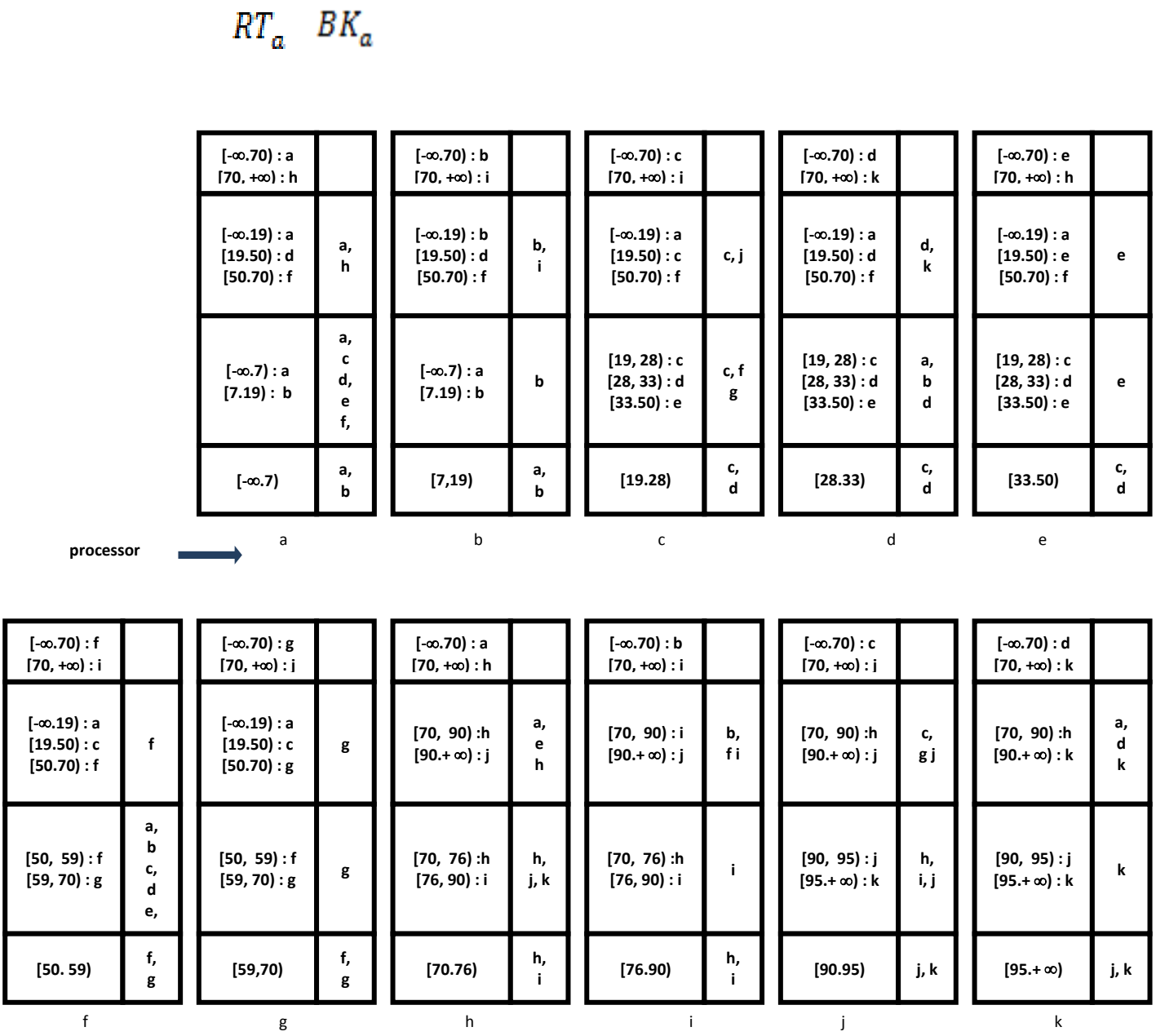


Fig.3.4. Consistent decentralized b+ tree of *Fig. 2.1.*

3.4 Assumptions

The following assumptions of the general b+ tree structure introduced in Section 2.2 are maintained: The ranges holding the key values are expressed by non-negative integer numbers and key values are stored in the nodes in non-decreasing order. We also

assume a perfect successiveness of the entries comprised in each non-leaf node. The root node of the b+ tree describes the universe of key values denoted *U*.

Additionally, we denote the union of all the ranges of a non-leaf node RT_i^l by $range(RT_i^l)$. LR_i refers to the local range in the leaf node of peer i. We assume that each child node RT_i^l in the tree knows its parent nodes p by means of a back-pointer table BK_i^l. Search and update operations can be initiated from any peer. We assume also a reliable, asynchronous communication system for exchanging messages between the different peers. We also assume that peers do not fail. Finally, the b+ tree model is considered to be a complete network model, where any peer is able to send messages to any other peer as long as the address of that peer is known.

3.5 Search in the consistent decentralized B-tree

3.5.1 Key Search

The search for a given key in the decentralized b+ tree is essentially the same as described in Chapter 2. The operation consists of exploring the tree from the top, starting from the peer initiating the search operation until reaching the peer k that holds the target key in its local range at RT_k^0. The main difference is that pointers point to peers rather than tree-nodes. *Fig. 3.5.1* describes step by step the search for the target key *100* and initiated by peer A. At each step of the search operation, the level of interest decreases by one. Notice that the back-pointer table described by the right-side box in each routing table is not needed for the search operations.

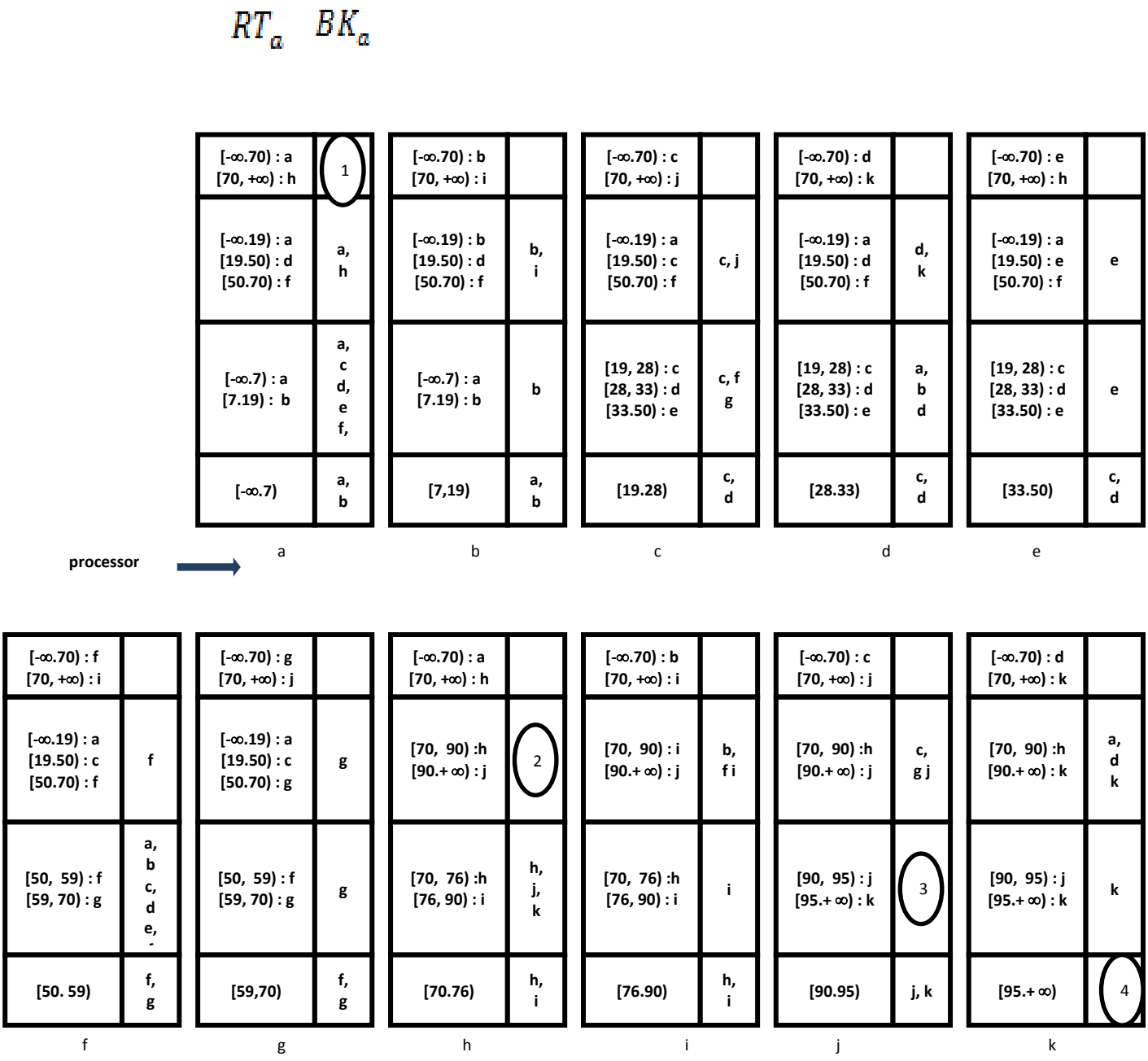


Fig. 3.5.1. Search for the key 100 initiated by peer a

3.5.2 Range Search

The search for a given range in the decentralized b+ tree is basically the same as described in Chapter 2. *Fig. 3.5.2* describes step by step the search for the target range *[90, 100)* initiated by peer A. The back-pointer table is also not needed for range search operations.

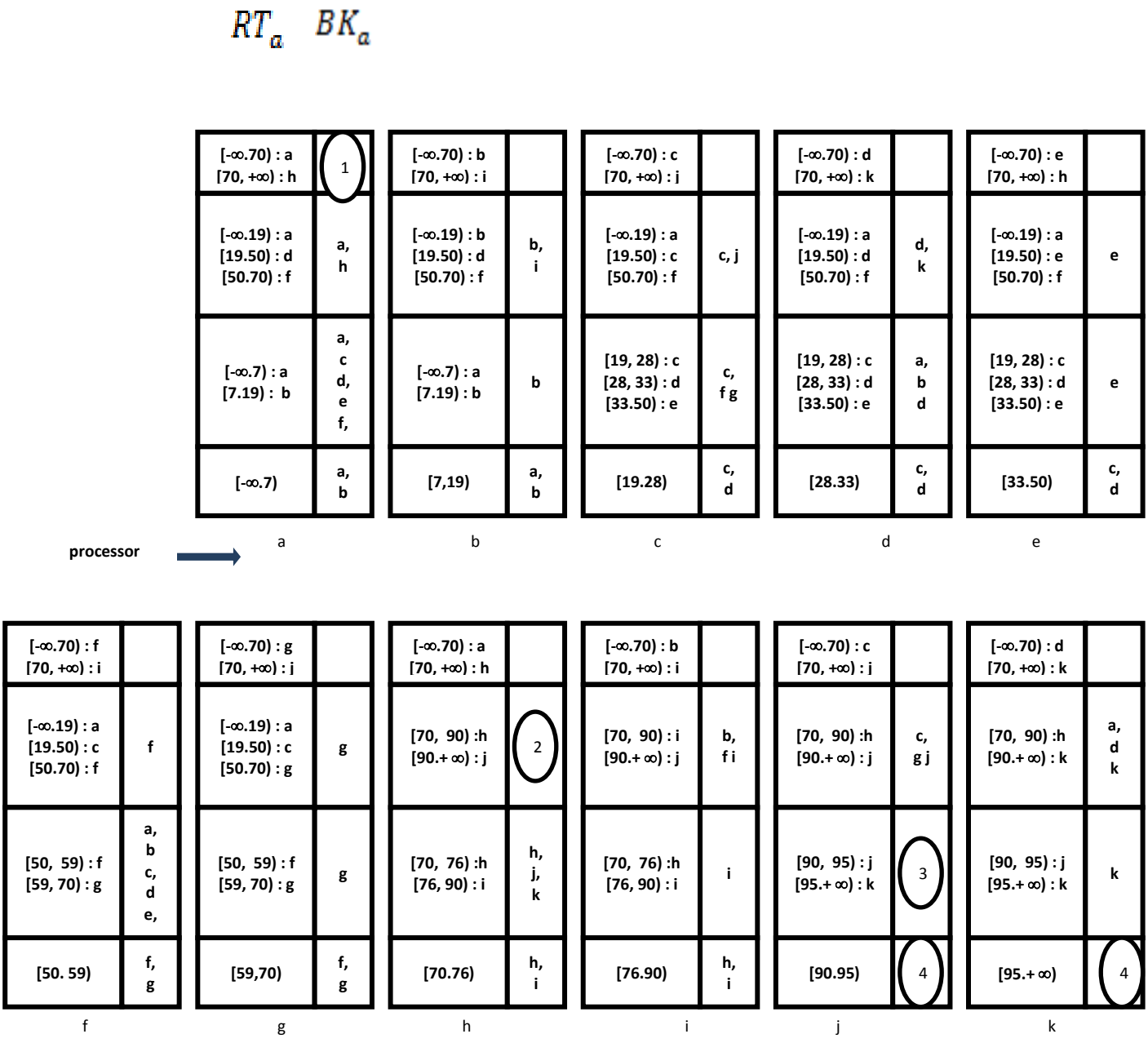


Fig. 3.5.2. Search for the range [90, 100) initiated by peer a

3.6 Updates in a consistent decentralized b-tree

In the decentralized distributed b+ tree introduced in Section 3.3, each peer maintains a branch of the tree, which results in duplication of the non-leaf nodes where each non-leaf node is expected to maintain a partial state of the distributed b+ tree. Since the local ranges comprised in the leaf nodes are not replicated among peers, Split/merge operations for leaf nodes remain simple. However, higher-level nodes require the updates to be coordinated by one master peer so that the state maintained by all involved nodes remain always consistent with one another. In the worst case, when the state of the root is changed, the update needs to be atomically propagated to all peers. Maintaining strong-consistency among the replicated states of the distributed b+ tree has been found problematic due to the huge overhead of such large-scale atomic updates [18]. We introduce the term transaction [18] which defines an atomic operation performed by a given peer in a P2P network. In fact, each transaction comprises a unit of work performed within the P2P network and treated by a given peer in a coherent and reliable way independent of other transactions. The peers use transactions to perform search and update operations atomically, without the assistance, interference or oversight of a central authority and each search or update operation is considered a single transaction.

4. Decentralized b+ trees with weak-consistency

4.1 Weak-consistency invariants

We consider three required constraints that are necessary for correct routing operations in the data structure where the primitive properties of the consistent distributed b+ tree may be violated when the peers are updated, even though, the search is still performed correctly in collaboration between the peers.

The weak-consistency constraints [1] are defined as follows:

- Invariant of universe A_U: Any peer should be able to initiate the search operation, so every peer *i* should maintain a description of the universe of key values *U* at the highest level *m* of its routing table.

  $\forall i : range(RT_i^m) = U$

- Invariant of navigability A_N: If any <*r,j*> is in RT_i^l, then the range *r* must be included in $range(RT_j^{l-1})$.

  $\forall i \forall l : < r, j > \in RT_i^l \Rightarrow r \sqsubseteq range(RT_j^{l-1})$

- Invariant of disjoint local range A_{LR}: The local ranges must be mutually exclusive among all peers.

  $\forall i \forall j : i \neq j \Rightarrow LR_i^0 \cap LR_j^0 = \emptyset$

We believe that there no need to maintain the dependency of the replicated states within one global state of the distributed b+ tree [1]. Indeed, each replicated state of a given non-leaf node can be updated atomically by means of separate transactions [18] independently of the other replicated states among other nodes. Since every non-leaf node RT_i^l maintains a back-pointer table describing its parent nodes RT_p^{l+1} pointing to node RT_i^l at the next lower level, each non-leaf node maintains a partial view of the tree and may update its sibling nodes surrounded by its partial view atomically until the updates are propagated to the higher levels nodes. In fact, the cascading split and cascading merge operations from the leaf nodes to the higher level nodes, i.e. leaf level split or merges triggering splits or mergers in successive higher levels, can be treated as separate transactions by the nodes at each level. The idea of using atomic transactions among nodes that are independent of the transactions performed by other nodes of the same peer or of different peers, is an alternative to the consistent decentralized b+ tree where the transactional updates of the tree-nodes are only performed by a peer itself, which results on a huge overhead due to the replicated state among different nodes at the same level of the distributed b+ tree. Such approach involves a perfect independence among the nodes and a replicated state update occurring in a one of the non-leaf nodes does not require the change of all the replicated states among other peers. This criterion allows different peers to have considerably different view of the b+ tree but still perform the search operations correctly. For instance, when the state of the root of a given peer is changed, the update does not need to be propagated to all peers but only the corresponding peer is locally updated. Hence, the different peers maintain different views of the root in the distributed decentralized b-tree.

4.2 Split and merge operations with weak-consistency

The following section describes how the update operations are collaboratively performed by nodes in the distributed b+ tree structure under the weak-consistency constraints, as introduced in *[1].* The split and merge algorithms are triggered independently by any leaf or non-leaf node and each update is treated as a separate transaction. Therefore, the updates may be executed in parallel if the corresponding transactions are independent. To ensure correctness in the presence of concurrent updates in real time systems, some concurrency control mechanism is required for each update. For instance, to allow a higher degree of parallelism, version number based optimistic transaction protocols *[9]* may be used. The method consists of maintaining a transaction number for the state of each updated entry. If any of the entries is being updated, the atomic operation is aborted and delayed to a later time. If not, the atomic operation is executed by the corresponding node and the version number is incremented once the atomic operation is achieved.

Here we define the atomic split and merge update operations for leaf level and non-leaf level separately. These algorithms assumes that the three weak-consistency invariants A_U, A_N and A_{LR} introduced in Section 4.1 are satisfied before and after each update operation and they assure that the invariants hold afterwards.

4.2.1. Split leaf node

Alg.4.2.1 is executed when a peer *i* wants to offload some data elements from its local range LR_i^0 held by its leaf node RT_i^0 to a new introduced peer *j*. First, LR_0 is split into two disjoint local ranges LR_1 and LR_2 . RT_i^0 preserves LR_1 while the set of keys and data in LR_2 are delegated to peer *j.* Second, all the parent nodes of RT_i^0 at the next higher level, and maintained by the back-pointer table BK_i^0 need to be updated as RT_i^0 looses part of LR_0 . Each node *p* back-pointed by BK_i^0, splits the entry comprising the offloaded range in RT_p^1 into two disjoint entries so that the additional entry points to the new peer *j.* Notice *that* BK_i^0 may include *i* if RT_i^1 has an entry that is pointing to RT_i^0 .Finally, the back-pointer table $BK_i^0$is copied to BK_j^0 and the top-most node RT_i^m is also copied to $\mathrm{RT_j^m}$. Mid-levels of RT_j may remain empty.

The leaf node split algorithm of [1] is presented below:

Algorithm 4.2.1 SplitLeafNode(i)

1: Initiator: processor i

2: Condition $\mathrm{RT_i^0}$ is overloaded in terms of storage or access load

3: Readset = $\{\mathrm{LR_i}, \mathrm{RT_i^0}, \mathrm{BK_i^0}, \forall\ \mathrm{P} \in \mathrm{BK_i^0}, \mathrm{RT_p^1}\}$

4: Writeset = $\{\mathrm{LR_i}, \mathrm{RT_i^0}, \forall\ \mathrm{P} \in \mathrm{BK_i^0}, \mathrm{RT_p^1}, \mathrm{LR_j}, \mathrm{RT_j^0}, \mathrm{BK_j^0}, \mathrm{RT_j^m}, \forall\ \mathrm{P} \in \mathrm{RT_j^m}, \mathrm{RT_p^{m-1}}\}$

5: Action:

6: Partition $\mathrm{RT_i^0}$ into 2 disjoint sets of keys $\mathrm{D_1}$ and $\mathrm{D_2}$ and $\mathrm{LR_i^0}$ into 2 disjoint ranges $\mathrm{LR_1}$ and$\mathrm{LR_2}$, accordingly

7: find 1 new peer j

8: $\mathrm{RT_i^0}$ <- $\mathrm{D_1}$, $\mathrm{LR_i^0}$<- $\mathrm{LR_1}$

9: $\mathrm{RT_j^0}$ <- $\mathrm{D_2}$, $\mathrm{LR_j^0}$<- $\mathrm{LR_2}$

10 : for $\forall\ \mathrm{P} \in \mathrm{BK_i^0}$ do

11: there must exist <x,i> $\in\ \mathrm{RT_p^1}$

12: if $\mathrm{x \setminus LR_1} \neq \emptyset$ then

13: $\mathrm{RT_p^1}$ <- $\mathrm{RT_p^1 \setminus <x,i> \cup \{<x \cap LR_1, i>, <x \cap LR_2, j>\}}$

14: BK_j^0 <- $BK_j^0 \cup \{p\}$

15: end if

16: if $x \cap LR_1 = \emptyset$ then BK_i^0 <- $BK_i^0 \setminus \{p\}$ end if

17: end for

18: RT_j^m <- RT_i^m, where m is the highest level of RT_i

19: , $\forall \mid P < r, p > \in RT_i^m$, BK_p^{m-1} <- $BK_p^{m-1} \cup \{j\}$

Alg. 4.2.1. Leaf node split algorithm as introduced in [1]

4.2.2 Split non-leaf node

Alg.4.2.2 is executed when a peer i wants to offload some entries from its non-leaf node RT_i^l, at level *l > 0*. Non-leaf node splits may be initiated by an over-loaded leaf node. The first step consists of finding an existing peer *j* with a routing table at the same level RT_j^l, that either contains some entries covering some common range with RT_i^l or has some space to take some entries from RT_i^l. Such operation is described as a transfer of responsibility and is further discussed in Chapter 6. Once an available peer *j* is found, the offloaded entries are transferred from RT_i^l to RT_j^l and each node RT_k^{l-1} of the peer *k* that is associated with the transferred ranges at a lower level is updated so that peer *k* is added to the back-pointer table BK_k^{l-1}. In addition, each node *p* back-pointed by BK_i^l, splits the entry comprising the off-loaded range in RT_p^{l+1} into two disjoint entries so that the additional entry points to the new peer *j.* The back-pointer table $BK_i^0$is also copied to BK_j^0 accordingly. Finally, if the top-most level of RT_i^m is split, one additional node RT_i^{m+1} is added to the routing table of peer *i.* in order to hold a pointer to the transferred ranges, such that the universe of key values *U* is maintained.

Algorithm 4.2.2 SplitNonLeafNode(i,l)

1: Initiator: processor i

2: Condition $\forall$ l\{0}. RT_i^l has too many entries

3: Readset = {RT_i^l,RT_j^l, BK_i^l, $\forall$ Pu $\in$ BK_i^l , RT_{pu}^{l+1} , $\forall$ Pd $\in$ RT_i^l , RT_{pd}^{l-1} }

4: Writeset = {RT_i^l,RT_j^l, BK_i^l, $\forall$ Pu $\in$ BK_i^l , RT_{pu}^{l+1} , $\forall$ Pd $\in$ RT_i^l , RT_{pd}^{l-1} }

5: Action:

6 : find j | <r,j> $\in$ RT_i^l $\vee$ j $\in$ BK_i^l s.t RT_j^l is empty or RT_j^l has some space for at least two entries or RT_j^l has some overlap E_e with RT_i^l, multiple such j (say, j_k may be chosen).

7 : Partition RT_i^l into two disjoint subsets E_s^l and E_e^l and partition range(RT_i^l) into disjoint ranges R_s^l and R_e^l accordingly. R_e^l may be partitioned into multiple sub-ranges R_{ek}^l

8: $\forall$ k do

9 : RT_{jk}^l <- RT_{jk}^l U E_{ek}^l

10: RT_i^l <- RT_i^l \ E_{ek}^l

11: $\forall$ P | $< r, p >$ $\in$ E_{ek}^l : BK_p^{l-1} <- BK_p^{l-1} \{ i } i U { j_k }

12: end for

13: $\forall$ P $\in$ BK_i^l do

14: There must exist <x,i> $\in$ RT_p^{l+1}

15: If if x \R_s^l $\neq \emptyset$ then

16: RT_p^{l+1} <- RT_p^{l+1} \ $< x, i >$ U $< x \cap R_s^l , i >$ U$_k$ $< R_{ek}^l , j_k >$

17: BK_j^l <- BK_j^l $\cup$ { P }

18: end if

19: if x $\cap R_s^l = \emptyset$ then BK_i^l <- BK_i^l \ { P } end if

20: end for

21 : if RT_i^l is the highest level in RT_i then

22 RT_i^{l+1} <- { $< R_s^l , i >$ U$_k$ $< R_{ek}^l , j_k >$ } , BK_i^l <- BK_j^l $\cup$ { i }

23 end if

Alg. 4.2.2. Non-leaf node split algorithm as introduced in [1]

Figure Fig. 4.2 identical to Fig.3 from Section 3.3 of [1] shows how a weakly-consistent b+ tree structure may grow with asynchronous leaf node and non-leaf node split operations.

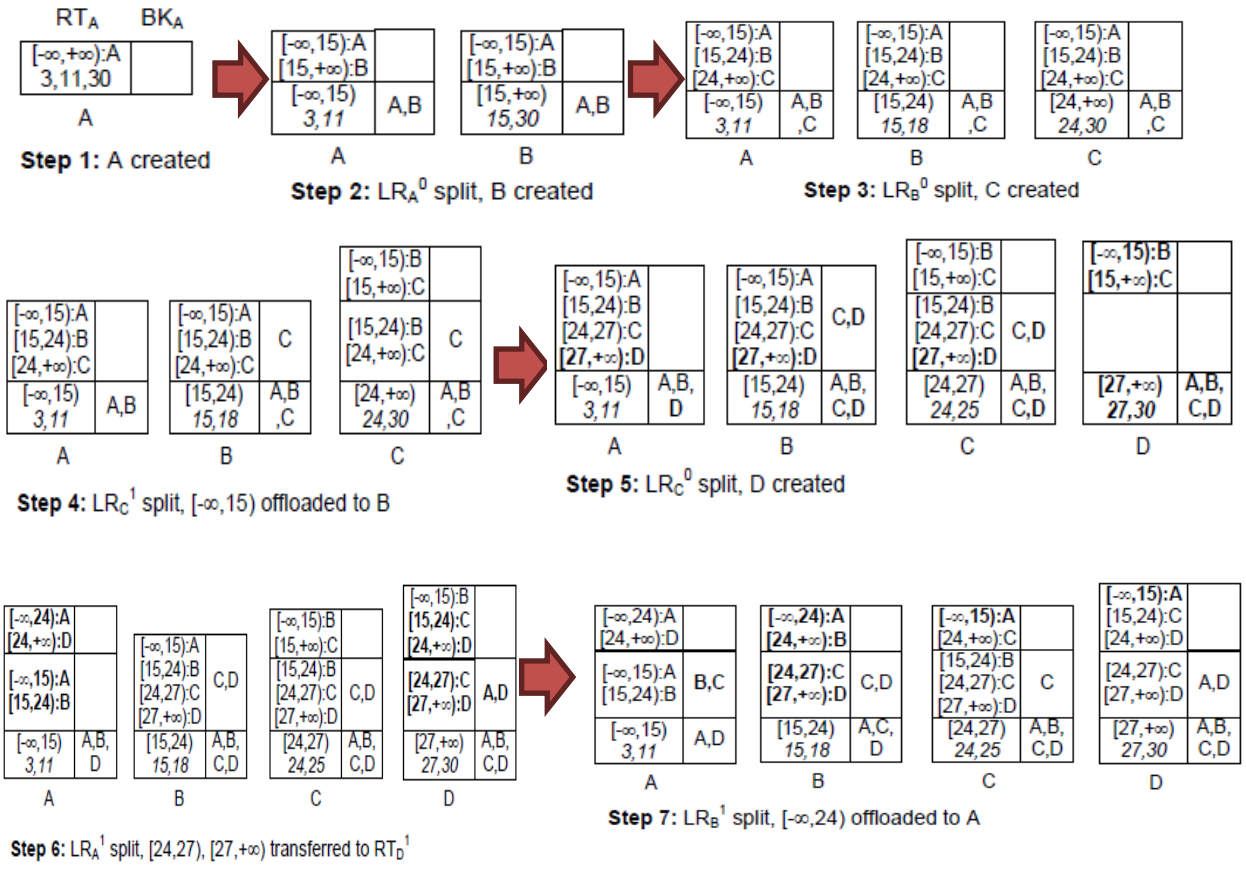


Fig. 4.2. Evolution of a weak-consistent b+ tree with asynchronous updates

Initially, Peer A is introduced and its routing table RT_A is created and comprises only a leaf node RT_A^0 *(step1).*This leaf node holds a local range LR_A described by the universe of key values *U = (-∞, +∞).* After some key data insertions, LR_A may become over-loaded and a leaf node split operation is performed in RT_A by off-loading a portion of LR_A to a new *peer B* . RT_A^1 is updated and then copied to RT_B^1 *(step2).* Similarly, after some data insertions in *peer B*, LR_B may become over-loaded and RT_B^0 is split into RT_B^0 andRT_C^0. The back-pointer nodes $RT_A^1$and RT_B^1 are updated by adding an entry that

contains the off-loaded range associated with a pointer to the new *peer C.* The back-pointer tablesBK_A^0, BK_B^0 and BK_C^0 are updated accordingly *(step3).* Suppose now that the non-leaf node RT_C^1 has too many entries and wants to offload a portion of its entries. A non lead node split operation is thus performed by RT_C^1 by choosing *peer B* to take responsibility of the transferred entry *<(-∞, 15], A>.* Since the top-most level of RT_C^2 is split, one additional node RT_C^2 is added to RT_C . This node is composed of two entries, the first one holds its self-range from RT_C^1 , while the second entry holds the transferred entry *(step4),* this way, the universe of key values *U* is maintained in the top-most level RT_C^m. Similarly to step1 and 2, step 5 consists of splitting the leaf node of peer C while steps 6 and 7 consist of non-leaf node split in RT_A^1 andRT_B^1.

We can see from the above example that the view of the b+ tree remains identical for all three peers A, B and C in steps 1, 2 and 3, while from step 4 onwards, different peers may have different views of the b+ tree. Indeed, the global view of the b+ tree structure under the weak-consistency constraints may no longer remain a single connected tree. Rather, the view may be distributed among several disconnected segments of the tree. Nevertheless, each peer maintains sufficient information for correct routing and the search operation is performed properly and can be initiated from any peer in the distributed b+ tree.

4.2.3 Merge of two leaf nodes

Algorithm 4.2.3 is executed when some data deletions cause a leaf node RT_i^0 of a peer *i* to become under-loaded. Therefore, peer *i* performs a leaf node merge operation in RT_i^0 and combines its local range $LR_i^0$with the local range of another peer LR_j^0. A suitable partner *j* for the merged can be found from the non-leaf node at the next higher level RT_p^1 , where *p* is held by the back-pointer table BK_i^0. If RT_p^1 is pointing to *j* for some other range *y*, then after the merged, the two entries <*x, i*> and <*y,j*> can be merged into one single entry <*x U y, j*> if the ranges are consecutive. In addition, all the higher level nodes of RT_i^l are merged with the corresponding nodes RT_j^l so that the information maintained by peer *i* for correct routing is not lost. Finally, peer *i* is released.

Algorithm 4.2.3 MergreLeafNode(i)

1: Initiator: processor i

2: Condition RT_i^0 is under-loaded in terms of storage or access load

3: Readset = $\{LR_i\ , RT_i^0, BK_i^0, \forall\ P\ \in BK_i^0\ , RT_p^1\}$

4: Writeset = $\{LR_i\ , RT_i^l, \forall\ P\ \in BK_i^0\ , RT_p^1, LR_j\ , RT_j^l, BK_j^0\}$

5: Action:

6: select j | $j\ \neq i\ \wedge < r,j > \in\ U_{\forall\ P\in BK_i^0}\ \ RT_p^1$

7: RT_j^0 <- $RT_j^0\ \cup RT_i^0\ ,\ \ LR_j^0$ <- $LR_j^0\ \cup LR_i^0$

8: $\forall\ l\ \ RT_j^l\ \ <-\ \ RT_j^l\ U\ \ RT_i^l$

9: $\forall\ l\ \forall\ P\ \in BK_i^l$ replace i by j in RT_p^{l+1} and merge node RT_p^{l+1} as necessary if it contains too few entries

10: release processor i

Alg. 4.2.3. Leaf node merge algorithm introduced in [1]

4.2.4 Merge of two non-leaf nodes

Algorithm *4.2.4* is executed when a peer i wants to merge some entries from its non-leaf node RT_i^l, at level *l > 0*, with a non-leaf node of another peer *j* at the same level RT_j^l . Such a merger may be initiated by an under-loaded leaf-node. First, peer *i* searches for an available peer *j* and combines its entries in $RT_i^l$with the other entries in RT_j^lat the same level. Similarly to the leaf node merge operation, a suitable partner *j* for the merged entries can be found from the non-leaf node at the next higher level RT_p^{l+1} , where *p* is contained in the back-pointer table BK_i^l. If RT_p^{l+1} is pointing to *j* for some other range *y*, then after the merged, the two entries <*x, i*> and <*y,j*> can be merged into one single entry <*x U y, j*> if the ranges are consecutive. The merged entry is thus removed from RT_i^l . Finally, if RT_p^{l+1} is the top-most level node and contains only one entry after the merger, that level may potentially be eliminated and the number of levels is reduced.

Algorithm 4.2.4 MergreNonLeafNode(i.l)

1: Initiator: processor i

2: Condition RT_i^l is under-loaded in terms of storage or access load

3: Readset = { $RT_i^l, l, \forall P \in BK_i^l, RT_p^{l+1}$}

4: Writeset = { $RT_i^l, \forall P \in BK_i^l, RT_p^{l+1}, RT_j^l, BK_j^l, \forall P \in BK_j^l, RT_p^{l+1}$

5: Action:

6: select j | $j \neq i \wedge < r, j > \in U_{\forall P \in BK_i^0} RT_p^{l+1}$

7: RT_j^l <- $RT_j^l \cup RT_i^l$ *and merge ranges in* RT_j^l as necessary

8: if l is not the highest level of RT_i then

9: RT_i^l <- ∅

10 : $\forall l \forall P \in BK_i^l$ replace i by j in RT_p^{l+1} and merge node RT_p^{l+1} as necessary

11: end if

12: If for any $p \in BK_i^l$, RT_p^{l+1} is the highest level of RT_p and contains only one entry pointing to p then

13: Delete RT_p^{l+1} and remove p from BK_p^l

14: end if

15 : BK_i^l <- ∅

Alg. 4.2.4. Non-leaf node merge algorithm as introduced in [1]

4.3 Challenges

The split and merge algorithms represented above and copied from *[1]* present some open problems. In fact, the major challenge of the split and merge algorithms is to find an existing peer *j,* whose routing table, at the same level, contains some entries covering some common range with peer *i* or have enough free space to hold the given range held by peer *i.* The algorithm presented in *[1]* does not provide any procedure for finding such peer *j*. We have developed such procedure *(see section 6.1)* called transfer of responsibility and have shown through our simulation studies that it is possible to find an existing peer that can be responsible for a given range at a given level of the distributed b+ tree. These procedures partially or fully explore some neighboring peers of the distributed b+ tree following a predefined convention.

We also found that we have to maintain a tradeoff regarding the transferred entries so that they remain balanced among the distributed b+ tree. Indeed, when a non-leaf node split operation is performed, the number of entries of the split node is reduced and part of the range is suppressed from the original node. If such reduction is performed randomly among the peers, we may face a situation where the common ranges are suppressed from many nodes and the probability of finding an existing peer *j*, that already contains some entries covering such common ranges with peer *i* becomes low. If no such peer is found, there is no peer that can be responsible for holding the requested ranges and the non-leaf split node becomes unfeasible. A solution to this problem is also described in *Section 6.3.*

5. Studying the validity of a the distributed b+ tree with weak-consistency

One goal of this work is to prove that the b+ tree structure allows for consistent search operations when the weak-consistency invariants A_U, A_N and A_{LR} of section 4.1 are maintained. In this chapter, we describe how we have established such a proof by using a specification language that can generate instances of models and simulate the execution of operations defined as part of the model. The model specification is described in the following sub-sections.

5.1 Purpose of Modeling the Decentralized B+ Tree with weak-consistency

The assumptions described by the weak-consistency invariants introduced in Section 4.1 let us think that if these assumptions are true, the search operations would properly work in practice. I chose to model the distributed b+ tree structure using the Alloy modeling language*[10]* and the goal was to explore the validity of weak-consistency using this modeling language. This goal may be achieved by defining the characteristics and properties of the distributed b+ tree and then checking the validity of some operations on the b-tree structure under these assumptions. Such validation is fairly straightforward to analyze using the Alloy analyzer. The Alloy analyzer was specifically developed to support so-called "lightweight formal methods". As such, it is intended to provide fully automated analysis, in contrast to the interactive theorem proving techniques commonly used with logic-based specification languages similar to Alloy.

In the following subsections, the Alloy language is introduced, together with the Alloy model of our b+ tree structure with weak-consistency. Then we discuss the results produced by the Alloy analyzer.

5.2 Introduction to the Alloy specification language

Alloy is a specification language that was developed by a team led by Daniel Jackson at MIT since 1997. Its aim is to check properties about a system model and to describe formally the components of a given system and the relationships between them. Influenced by the Z notation, Alloy is an object-based language that is based on set theory and first-order logic. By saying first order logic, we mean that it consists of a structural modeling language based on declarative statements that are unaffected by their order. For instance, two successive statements have no priority and the statements are completely independent from one another when Alloy checks for a given model specification.

Given the specification of a model in terms of object classes and relationships, the Alloy analyzer generates a several instance models consistent with the given model, and verifies for each instance model that certain given desirable assertions are satisfied. This is a kind of model checking. Alloy differs from many specification languages designed for model-checking in that it is declarative rather that imperative, and permits the definition of infinite models. The Alloy analyzer is designed to perform finite scope checks even on infinite models. More details on the Alloy specification language may be found in [10].

The structures in the Alloy models are basically built from atoms and relations, corresponding to the basic entities and the relations between them. The Alloy specification language is described by [10] as follows:

Atoms

An atom is a primitive entity that is:
Indivisible: It cannot be partitioned into smaller parts.
Immutable: its properties do not change over time.
Un-interpreted: It does not have any built-in properties.

In order to build entities that are divisible, mutable or interpreted, relations are introduced to capture these properties as an additional structure.

Relations

A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. A relation can be represented as a table, in which each entry is an atom. The order of the columns matters, but not the order of the rows. A relation can have any number of rows, called its size. The number of columns is called the arity of the relation, and must be one or more. Relations with arity one, two and three are said to be unary, binary and ternary, successively. A relation of arity of more than three is called a multi-relation.
For instance, a set of names and a set of addresses, each of size 3 may be represented by:

Name = {(N0), (N1), (N2)}
Address = {(D0), (D1), (D2)}

We can design a model in which there is only one address per name as a binary relation called location that relates names to addresses as follows:
location = {(N0, D0), (N1, D1), (N2, D2)}

Predicates

Predicates are parameterized constraints, i.e. conditions that may be true or false and can be used to analyze a model with constraints that may not always hold. The condition stated in a predicate describes the properties of a given system and are evaluated by the model checker according to some input parameters and for each parameter is associated a domain as an output.

For each of these parameters, if the input satisfies all the model properties described by the conditions that are listed in the body, then the predicate evaluates to true. Otherwise it evaluates to false.

For instance, a predicate having two parameters and 3 conditions is declared as follows:

pred name [parameter1: domain1, parameter2: domain2,…, parameterN: domainN]

{

condition1

condition2

condition3

}

Facts

Predicates that are assumed to always hold are called facts. A model can have any number of facts, each a paragraph of its own labeled by the keyword fact, and consisting of a collection of constraints that defines the properties of a given system.

Signatures

A signature introduces a set of atoms. For instance, the declaration *Sig A {}* introduces a set named *A.* Each set can also be introduced as a subset of another set. Thus, *Sig A1 {} extends A* introduces a set named *A1* that is a subset of *A.* Such signature is declared as independently of any other is a top-level signature.

The extensions of a signature are mutually disjoint. Additionally, an abstract signature has no elements except those belonging to its extensions. A signature declaration is expressed as
Sig {relations}{facts} where *relations* introduces the interactions of this set with other sets and *facts* introduces the facts related to this set of objects. The relations between objects are declared in Alloy as a field of signatures where each field is described by a domain as an input signature and a range as an output signature. For instance, we consider two signatures *Sig A {}* and *Sig B {}.* The binary relation *r: A -> B* introduces a constraint whose domain is A and whose range is given by *B*, declared as a field of signatures *Sig A {r: B}.*

Assertions

Assertions are predicates. The validity of each assertion is verified by the Alloy analyzer for each generated instance model, based on the given model specification and specifically, based on the facts that are included in this specification. An assertion is intended to follow from the stated facts of the model. If any assertion does not follow from these facts for a given model instance, then this model instance represents a counter-example which shows that the assertion is not necessarily satisfied. In the case that no counter-example is found, the corresponding assertion may be valid.

5.3 Modeling the decentralized b-tree

5.3.1 System model

In order to validate the properties of the b+ tree with weak-consistency, we modeled the structure by defining the routing tables, the nodes and the ranges as signatures that interact following some restrictions that are described by the facts of the model. Then, some assertions are introduced and used by the Alloy analyzer to check their validity. In particular the consistency invariants A_U, A_N *and* A_{LR}, introduced in Section 4.1, are introduced as assertions to be verified. In addition, a search predicate is defined in the end to illustrate the search operation for a given key value that is initiated by one of the existing peers in the distributed b+ tree. The distributed b+ tree system model that is built by the Alloy specification language is described in this section. Notice that each statement of the algorithm is supported by useful comment, described by the symbol "//" as in the most popular programming language such as in C and Java, to explain the meaning of its following statement.

Signatures

The following Alloy statements describe the different signatures, i.e. classes that are used to model the distributed b+ tree structure.

```
// main structure of the model
//each routing table, called RT, is composed of some nodes, called Node
some sig RT {nodes: some Node}
//some nodes have some entries called Entry and belongs to a level
some sig Node{entries: some Entry ,levels: one Level}
```

```
some sig Level {}

//some entries have some ranges called Range and points to one or zero node

some sig Entry{range: some Range, pointer: lone Node}

//some nodes are leaf nodes and called Leaf

some sig Leaf in Node{}

//Ranges are represented by a set of key values from A to F

abstract sig Range{}

//all key values are mutually disjoint

one sig A,B,C,D,E,F extends Range{}
```

Alg. 5.3.1.1. The signatures, i.e. classes used to represent the b+ tree structure

The above statements describe a set of routing tables, called *RT*. This set is related to a set of nodes called *Node* by the relation nodes. Each node itself is related to a set of entries called *Entry* by the relation entries and associated with one of the ordered levels. Each entry is related by the range relation to a set of key values denoted by *Range* and has a pointer that relates it to lone node, i.e. one or zero node. The set of leaf nodes, called *Leaf*, is represented as a partition of nodes, i.e. some nodes can be leafs while other nodes are not.

Ordering

The routing table, the levels and the search states are ordered. The model begins with open statements which impose an ordering on the set of atoms used to express the routing tables *(RT),* the levels *(Level)* and the search states *(State)* as follows:

```
// Routing tables, levels and search states are ordered
//ordering the routing tables
open util/ordering[RT] as ord1
//ordering the levels
open util/ordering[Level] as ord2

//ordering the search states
open util/ordering[Search_State] as ord3
```

Alg. 5.3.1.2. Statements for ordering the routing tables, the levels and the search states

In fact, we can use the ordering Alloy module *util/ordering* by adding the above lines to the beginning of the model to construct a linear ordering and refer to properties of that ordering. We parameterize a given set of atoms so that we can call order comparison operations on elements of that set. In the above example, the routing table, the levels and the states of the search operation (see next section), are ordered and denoted *ord1, or2 and ord3* consecutively. The *util/ordering* module provides few useful functions that we can use, including first, next and last: first returns the first atom in the linear ordering, last returns the last atom in the linear ordering while next maps each atom (except the last atom) to its succeeding atom.

Facts

Once the main structure of the distributed b+ tree is defined, we then need to organize the interactions between the different objects. Indeed, facts are used as an additional structure to define how these objects are related to each other and what kind of restrictions are needed to be able to correctly model our distributed b+ tree system. The complete list of restrictions is given below:

```
fact node_Transition
	{
	//every node is associated with a routing table by the relation nodes
	all n:Node|one r:RT | n in r.nodes
	}

fact leaf_nodes
	{
	//if two leaf nodes are different, their corresponding routing tables are also
	//different, contrary to the non leaf nodes since a routing table can have different
	instances of non-leaf nodes each one in a different level.
	all l1,l2:Leaf | l1 !=l2 => nodes.l1 != nodes.l2
	// the number of leaf levels is equal to the number of Routing tables
	// i.e each routing table has a unique leaf node
	#Leaf = #RT
	}

fact level_Ordering
	{
	// if two different nodes belongs to the same routing table cannot be at the same
	//level
	all r:RT,n1,n2:r.nodes | n1 != n2 => n1.levels != n2.levels
	//the lowest ordered node of each Routing table is in the leaf level
	all n:Node, l:Leaf | l in n => ord2/first = n.levels
	//more than one level must exist for each Routing Table
	all r:RT | #{r.nodes} > 1
	//no empty levels between the highest and the lowest level. Since levels are
	ordered, the system must generate instances of Level that may be used by the
	routing table. Otherwise, some internal levels may not exist.
	all r:RT, l:r.nodes.levels, l':^ord2/next[l], l'':^ord2/next[l']
		{
		l'' in r.nodes.levels => l' in r.nodes.levels
```

```
        }
    }

fact Range_distribution_for_Balancing
    {
    //all the key values must be in the local range of the leaf nodes
    all r:Range| one l:Leaf | r in l.entries.range
    // two different leaf nodes cannot have the same local range (Alr)
    all l1,l2:Leaf | l1!=l2 => l1.entries.range != l2.entries.range
    //the local ranges are balanced
    //(the size of the local range in each Routing table is equal else different of 1r
    all l1,l2:Leaf | l1 !=l2 =>
        {
        #{l1.entries.range} = #{l2.entries.range}
        or #{l1.entries.range} = #{l2.entries.range} -1
        or #{l1.entries.range} = #{l2.entries.range} +1
        }
    }

fact entries_structure
    {
    //the entry of the local range does not have a pointer
    no Leaf.entries.pointer
    //every node point to another node at a lower level
    all l:Level,l':ord2/next[l] | levels.l != none and levels.l' != none =>
    {levels.l'}.entries.pointer = levels.l
    //every pointer to another node at a lower level must include its range
    //corresponds to the invariant of navigability (An)
    all n:Node-Leaf,e:n.entries, r:e.range, p:e.pointer| r in p.entries.range
    //every entry is associated with a node by the relation entries
    all e:Entry | one n:Node | e in n.entries
    // two distinct nodes have distinct entries
    all n1,n2:Node | n1 != n2 => n1.entries != n2.entries
    //two different entries of the same node cannot share the same range
    all n:Node, e1,e2:n.entries | e1 != e2 => e1.range & e2.range = none
    //describing the Universal Range in the highest level of each Routing table (Au)
    //Au1 : describing part of the universal range invariant Au
    //This statement says that the universe of key values should be described by the
    range of each node at the highest level of each routing tables. This fact statement is
    not sufficient since our model should allow to have different peers that have different
    high levels. The following statement is only applied to the highest ordered level
    instance of all the level instances that are generated.

    all n: Node | n.levels = ord2/last =>Range = n.entries.range
    //every entry except the entries of Leaf nodes has to point to a given node
    all e:Entry - {Leaf.entries} | e.pointer != none
    // the number of entries for each node except the Leaf nodes is greater than 1
```

```
all n:Node-Leaf | #n.entries >1
//describing the universal range in the highest level of each Routing table
//corresponds to the invariant of universe (A_U)

//Au2 : describing part of the universal range invariant Au

//with each routing table is associated one node that comprises the set of all the
//key values in within its entries. This fact is not sufficient since this generated
one node can be leaf node and the search operation become inaccurate, we
should add another fact statement to tell which node can contains the universe of
key value and in which level (Au1 required)
all r: RT | one n: r.nodes | Range = n.entries.range
// for each routing table, there is always two succesives nodes
// where the higher one contains the set of all the key values
all r:RT, n2:r.nodes, l1,l2:r.nodes.levels { l2 = ord2/next[l1] and n2.levels = l2 =>
Range = n2.entries.range}

}
```

Alg. 5.3.2. The facts, i.e. restrictions, used to represent the b+ tree structure

The above facts are interpreted as follows: the first set of facts, called *node_transition* enforces that each node is associated with some routing tables and that one node instance cannot belong to more than one routing table at the same time. This fact is used to ensure that for each routing table, a number of nodes belonging to different levels are associated with it. The second set of facts is called *leaf_nodes* and states that if two leaf nodes are different, their corresponding routing tables are also different. It states also that the number of leaf levels is equal to the number of routing tables, i.e each routing table has a unique leaf node. The next set of facts is called *level_Ordering* and is used to order the levels of the routing table as following: two nodes belongings to the same routing table have to correspond to two different levels, and that the leaf node of each routing table corresponds to the node associated with the first ordered level. Also, more than one level must exist for each routing table for the routing to work properly. Another fact that is important to define is that for any two existing levels, all in-between levels between these two levels must also exist so that the level ordering principles used by the *util/ordering* module are not violated.

The last set of facts describes the entries structure as follows: The entry of the local range does not have a pointer, every node points to another node at the next lower level must comprise the latter range. Also, two different entries of the same node cannot have the same range, every entry except the entries of leaf nodes has to point to a given node and the number of entries for each node except the leaf nodes is greater than 1. The last two statements say that with each routing table is associated one node that comprises the universe of key values *U* within its entries and that for each routing table, there is always two succeeding nodes where the higher one contains the universe of key values *U*.

Key values

Since Alloy is not arithmetic and does not allow integer representation of key values, key values are represented as a set of ranges, i.e. instances of the signature *Range* where each elementary range describes a key value identified by a letter from *A to Z*. This results in a relaxation of the consecutiveness constraint of key values so that there is no ordering of key values i.e. Key values are randomly allocated among the peers. Notice also that the maximum number of instances for each signature in Alloy is 18. Thus, we may not be able to generate more than 18 letters, where each letter describes the local range of one peer. This implies that our b-tree model may not have more than 18 peers, since each peer should maintain a different letter describing its local range in its leaf node.

5.3.2 Model instance creation

The Alloy analyzer generates instances of the b+ tree model by processing different predicates and assertion. The Alloy analyzer first produces all possible instances of the model with the properties defined by *facts*. In addition, if an assertion is processed, the Alloy analyzer applies the additional constraints defined in the body of the corresponding assertion. If these constraints are consistent with the original facts of the model, the Alloy analyzer states that these constraints are consistent with the original facts. Otherwise, a model instance is generated as a counter-example to show that the assertion is not necessarily satisfied.

The run statement is used to instruct the Alloy analyzer to generate model instances. If refers to a predicate, which in many cases is a dummy predicate. Alloy must limit the number of instances to be generated in the *run* statement. The predicate *example* (below) uses 4 instance restrictions: *RT, Node, Level and Entry.* These instance restrictions are used to limit the number of instances the b+ tree model may contain. By giving the maximum number of instances that can be used by each of the signatures *RT, Node, Level and Entry*, the Alloy analyzer generates all the possible instances that correspond to this setting.

```
pred example {}

run example for 2 RT, 10 node, 10 level, 10 entry
```

Alg. 5.3.4. Running an example of the distributed b+ tree structure with two routing tables

The model instances generated by the Alloy analyzer can be automatically displayed in a graphical representation. Fig. 5.3.4 shows an example of the distributed b-tree system having 2 routing tables, 6 entries, 4 levels, 4 nodes and 4 pointers.

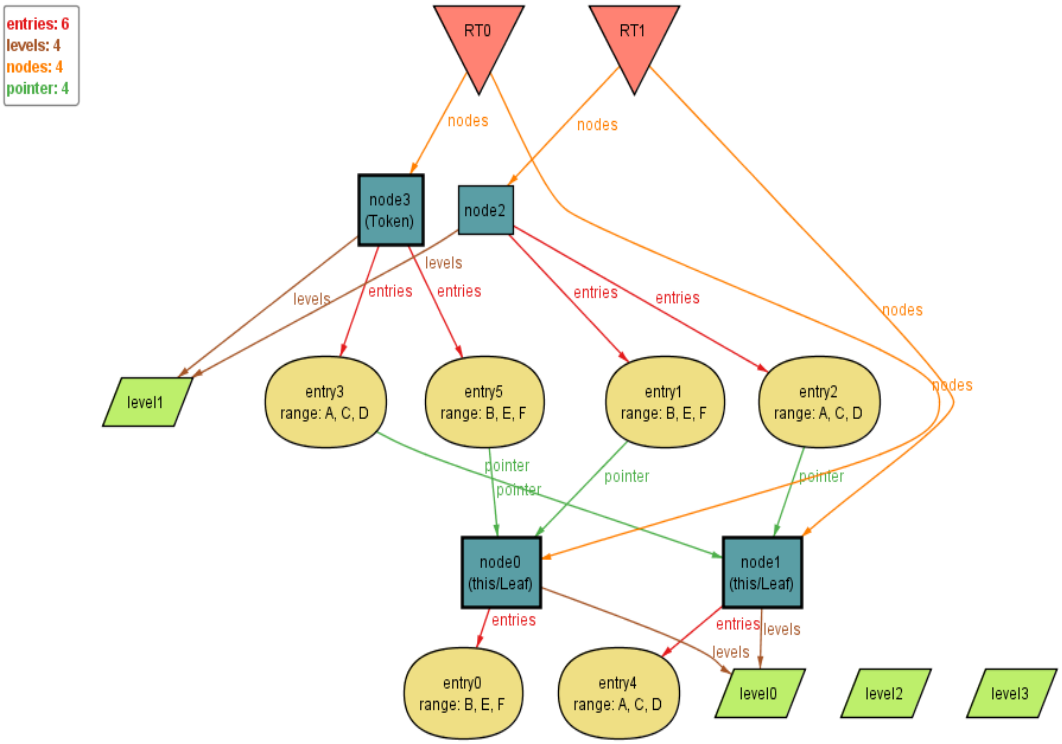


Fig. 5.3.2.1. Example of the distributed b-tree system with 2 routing tables, 6 entries, 4 levels, 4 nodes and 4 pointers (magic layout view).

We can clearly see from the above example that two routing tables denoted by RT_0 and RT_1 are used, each one of them has two different nodes corresponding to different levels and each node comprises some entries and points towards another peer. We considered here only 6 key values for the universe of key values *U = {A,B,C,D,E,F}* but one can add more key values if needed. The levels are ordered and the number of levels has to be determined in the predicate example by means of the parameter *Level,* but not all the levels are used here. For instance, we have set to 4 the maximum number of levels while only 2 levels were used by the above example. Notice that level0 corresponds to the first ordered level and is associated with leaf nodes.

The tab ‘next’ in the Alloy analyzer menu allows us to generate another instance permitted under the restrictions that were introduced as *facts*. For the sake of clarity, we used the magic layout view which illustrates the distributed b+ tree model with more simplicity and makes the structure easier to read. We can also see from *Fig 5.3.1* that the key values are balanced automatically with a difference of one key value between each two routing tables in the worst case, which is consistent with our expected results. Notice that in the case that the number of routing tables is higher than the number of key values, the predicate is inconsistent and does not generate any example since one of the main facts states that each routing table should have at least one key value.

Balancing

One of the design goals of the distributed b+ tree was keeping the search tree balanced while growing and shrinking. For this reason, it is desirable to have a similar number of stored key values in all leaf nodes. Thus, we introduce additional constraints, i.e. facts, to keep the distributed b+ tree structure balanced by having a similar number of key values within each peer.
The Alloy statements used for these facts are shown below:

```
// describes how the ranges are distributed for balancing
fact Range_distribution_for_Balancing{
//each local range contains some distinct Ranges
all r:Range| one L:Leaf | r in L.entries.range
all L1,L2:Leaf | L1!=L2 => L1.entries.range != L2.entries.range
//choose the best distribution
//(the number of ranges in each Routing table is equal else difference of 1}
 all L1,L2:Leaf | L1 !=L2 =>
        {
        #{L1.entries.range} = #{L2.entries.range}
```

```
        or #{L1.entries.range} = #{L2.entries.range} -1
        or #{L1.entries.range} = #{L2.entries.range} +1
        }
}
```

Alg. 5.3.2.4. Restrictions used to balance the distributed b-tree nodes

The above constraints state that the balancing should be done systematically by maintaining the number of elementary key values associated with each leaf node to be equal. If this is not possible, the program then distributes the key values in such a way that only a difference of one in the number of elementary key values is allowed. The above figure shows an instance of the distributed b+ tree model having 6 entries, 4 levels, 4 nodes and 4 pointers. The Alloy analyzer generates all the possible instances of the given model. The magic layout view can be used in the settings of the Alloy analyzer to clearly illustrate the system model of the distributed b+ tree structure, where key values are balanced and equally distributed among all the peers. Indeed, *Fig. 5.3.4* demonstrates an example where the local ranges are distributed equally among the two routing tables. The structure is composed of two local ranges at level 0, the first local range is associated with the routing table *RT0,* and corresponds to entry0 with 3 distinct ranges *B, E, F,* while the second local range is associated with the routing table *RT1*, and corresponds to entry4 with 3 distinct ranges *A,C* and *D.* Notice that the sum of all disjoint local ranges represents the universe *U = {A,B,C,D,E,F}.*

5.3.3 Search operation

The search operation is possible in Alloy by distinguishing between the key values *A, B,C* etc. When performing a search operation, the user should tell the program who is the initiator, looking for a given key value. This can be done by using a predicate denoted "*initiator[X]*", were *"X"* is the routing table that is initiating the search operation. For instance, I chose for the next example *X* as "*ord1/first*" which means that the first element of the ordered routing tables i.e. RT_0 is the initiator. In addition, one has to tell which key value the initiator is looking for. This can be done by calling a predicate "*target[Y]*", were *"Y"* is the target key value. For instance, I have chosen for the following example *Y* as *"B",* meaning that the initiator is performing a search operation for the key value *B.*

Performing a search operation for a target key value is done by means of search states by sending a token as a message, starting at search *state0,* i.e. the highest level node of the initiating routing table, and navigating from one routing table to the next lower level of the same or another routing table, until the query reaches the lowest level i.e level 0, in the last search state, when the target key value is found. Starting from the initiating routing table, the token is moved from one node to another by making a state transition at each hop of the navigation and should reach its final destination *Y* given by *target[Y]*". If the token reaches its final destination, the search operation is performed successfully. Otherwise, the search operation is not consistent with the stated properties in *facts* that were used to define the system model.

The search algorithm is defined below:

```
//describing the search states
sig Search_State {Token: one Node}

//describing the initiating routing table
pred initiator[Initiator:RT]
        {
        //in the initial state, only the Initiator has the token
        all n:Initiator.nodes, l:n.levels, l':ord2/next[l] | levels.l' =
        none and levels.l != none => ord3/first.Token = n
        }

//describing the target key value we are looking for
pred target[Target:Range]

        {
         //in the final state, the token must be in the leaf node having the corresponding
        //key value
        one n:Node | {ord3/last}.Token = n and Target in n.entries.range
        and n in levels.{ord2/first}
        // describes the state transtion i.e. how the search query navigates from a node
        to another one
        all s: Search_State, s': ord3/next[s], e:s.Token.entries
                {
                Target in e.range and {s.Token}.levels != ord2/first =>
                s'.Token in e.pointer
                else {}
                }
        }
```

```
// choosing search parameters (the search parameters may be changed)
fact choosing_the_parameters
        {
        //choosing who is the initiating routing table
        initiator[ord1/first]
        //choosing the target key value
        target[B]
        }
```

Alg. 5.3.2.5. Search operation algorithm for the key value B and initiated by RT0

We generate some example visualizations of the search operation by declaring the predicate called "*Search_for_B_Starting_at_RT0*"as follows:

```
pred Search_for_B_Starting_at_RT0{}
run Search_for_B_Starting_at_RT0 for 2 RT, 10 Node, 10 Level, 10
Entry, 2 Search_State expect 1
```

Alg. 5.3.2.6. The predicate executing the latter search operation

The predicate "*Search_for_B_Starting_at_RT0*" uses 4 parameters, one for the number of states, which must be equal or more than the number of used levels in order to guarantee that the token, starting at the highest level of the initiator will get to the lowest level of the routing table corresponding to the peer having the target key value at the last state. Using a small number of search states may be insufficient for the target to reach its final destination.

The other parameters used are the number of routing tables and the maximum number of nodes and entries that can be used by the distributed b+ tree structure. In this algorithm, key values are arbitrary allocated among the peers and each routing table has at least one key value stored in its leaf node. The parameter *"Expect 1"* means that the predicate is expected to find only one target range in its final state, i.e. the query stops as soon as the target key value is found. In order to have a good view of the state transitions during the search operation, it is suggested to make a projection over the states. This can be done by selecting *projection over states* in the tab menu of the Alloy analyzer. In fact, the Alloy analyzer allows projecting the models over one or more objects so that users can observe the instances of all related objects from the view of the projected object.

Below are shown the results of the predicate *"Search_for_B_Starting_at_RT0"* for 2 routing tables with a maximum of 8 levels.

Search State 0

The token is in the highest level node of the initiating Routing table RT0.

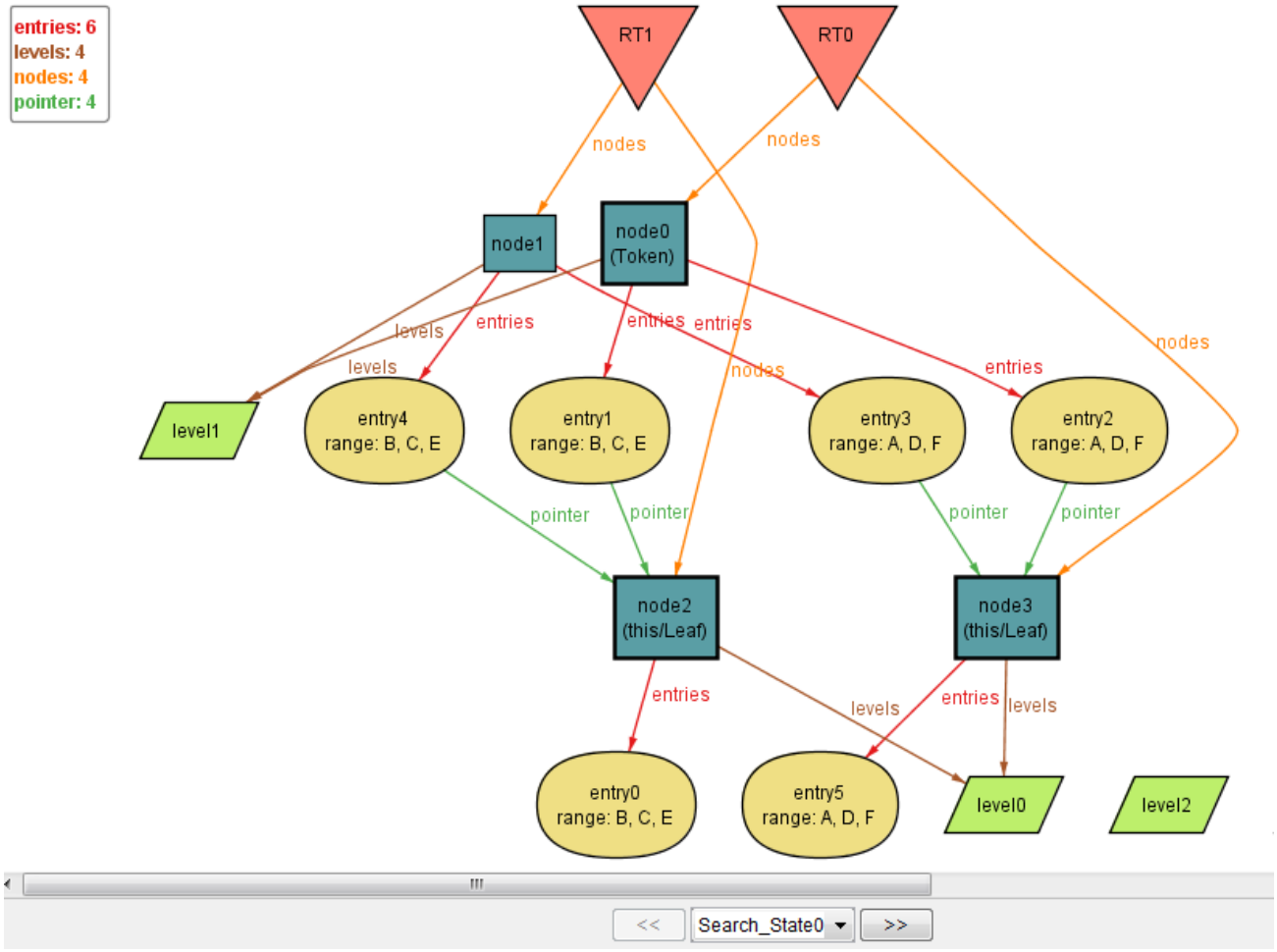


Fig. 5.3.2.2. State0 of the look up operation on the distributed b+ tree (magic layout view).

Search State 1

The token is moved from the highest level node, i.e. *level1* of the initiating routing table *RT0,* to *level0* of the routing table *RT1* by following the pointer of *entry0* comprising the key value *B.* The token is now in *node2*, which is leaf node, and has the key value *B* in its range. The target therefore has reached its final destination and the search operation is terminated.

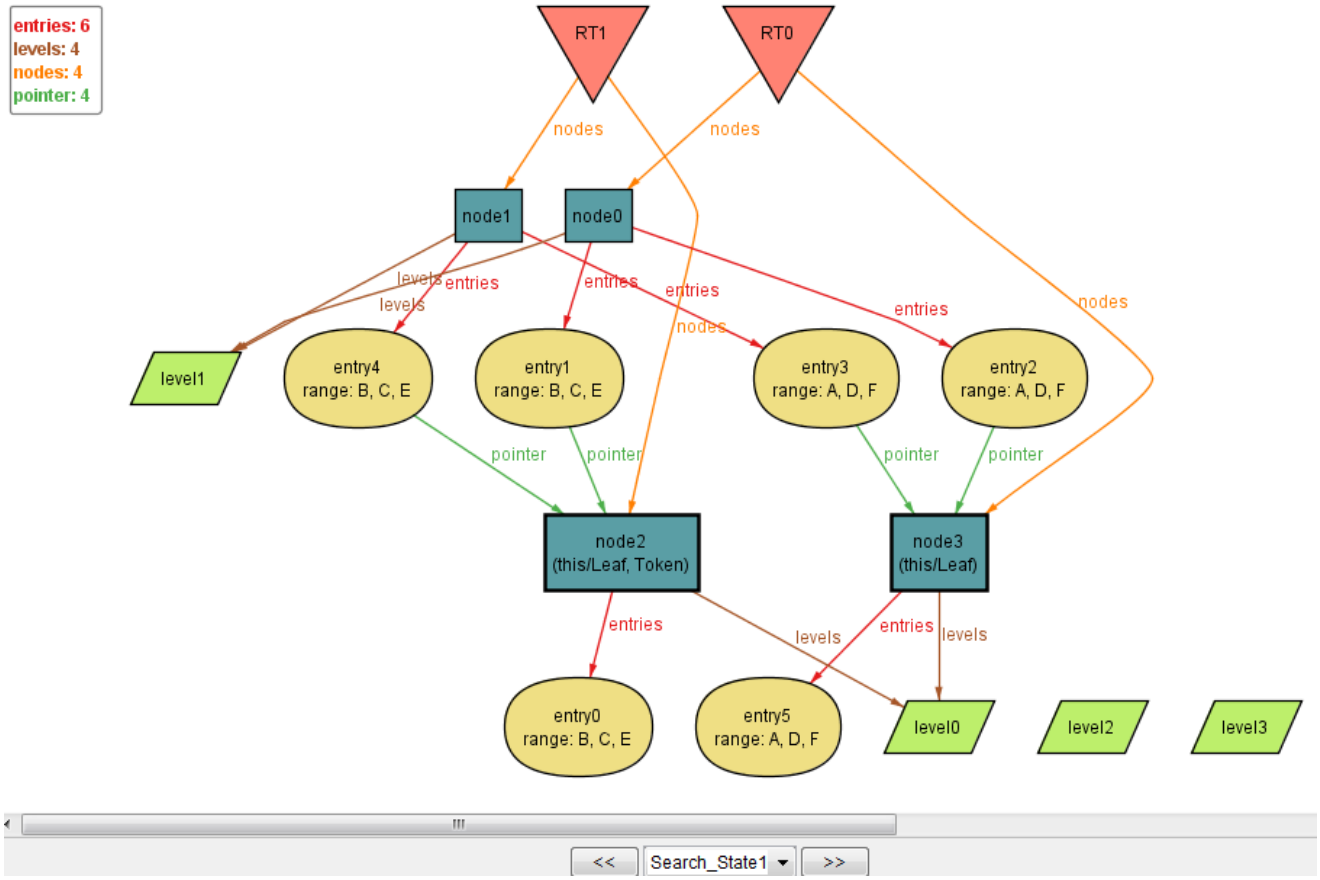


Fig. 5.3.2.3. State 1 of the look up operation on the distributed b+ tree (magic layout view).

Notice that one can change the maximum number of search states that can be used, but it is more preferable to maintain it equal to the number of levels since the query starts from the highest level of the initiating routing table and reaches its final state in the leaf node of the routing table having the target range. In the case where the number of chosen search states is less than the number of levels, the search operation fails and the token i.e. query can never reach the final state.

5.4 Weak-consistency invariants

5.4.1 Formalization

The goal behind using the Alloy modeling language is to verify that the search operation is performed properly under the weak-consistency invariants A_U, A_N *and* A_{LR} as defined in *Section 3.7.* By checking the validity of some assertions having constraints that define A_U, A_N *and* A_{LR} successively, it is possible to verify whether the search operation is consistent with the b+ tree model with weak-consistency.

We note that these assertions are already included as facts in the model specification presented in Section 5.3..

Invariant of universe A_U

Since any peer should be able to initiate the search operation, each peer should maintain a description of the universe of key values U at the highest level of its routing table. The assertion given below is used to verify the validity of this constraint.

```
//1) Invariant of universe (AU)

assert AU{

all r:RT | one n:r.nodes | Range in n.entries.range

}
```

```
check AU for 10Assert
```

5.4.1.1. Assertion to verify the universal coverage invariant

Invariant of navigability A_N

For correct navigation of the search queries, an assertion is declared to verify that any range in a given entry belonging to a given node must be included in the range of the node it points to at the next lower level i.e. any range in a given non-leaf node can find a destination node having that range within its entries at a next lower level. The assertion given below is used to verify the validity of this invariant.

```
//2) Invariant of navigability (AN)
assert AN{
all n:Node-Leaf, r : n.entries.range | r in {range.r}.pointer.entries.range
}
check AN for 10
```

Assert. 5.4.1.2. Assertion to check the navigability invariant

Invariant of disjoint local range A_{LR}

This assertion is used to verify that two different leaf nodes must hold two disjoint sets of key values. In other words, every routing table has a unique local range so that there is only one destination in the final state and that a target key value cannot be found at the leaf nodes of two distinct routing tables.

The following assertion is used to verify this assumption.

```
//3) Invariant of disjoint local range

// Every Routing Table has a uniqe Local Range
assert ALR{
all L1,L2:Leaf | L1!=L2 => L1.entries.range != L2.entries.range
}
check ALR for 10
```

Assert. 5.4.1.3. Assertion to check the Invariant of disjoint local range

5.4.2 Inconsistent b+ tree structures

The assertions described in *5.4.1* verified that for any number of routing tables, levels and nodes used, the search operation is performed properly and the query always reaches its final destination in the last state when the b+ tree model includes the weak-consistency invariants. But what happens if one or more of these invariants is removed? In other terms, what happens if these constraints become too weak? Can the search operation still perform correctly? For the sake of investigation, we remove one of the facts related to the weak-consistency invariants A_U and we generate the same example using the same empty predicate as in *Section 5.3* and see what happens.

The following statement is removed from the facts of the b+ tree structure.

// for each routing table, there is always two succesives nodes

// where the higher one contains the set of all the key values

all r:RT, n2:r.nodes, l1,l2:r.nodes.levels { l2 = ord2/next[l1] and n2.levels = l2 =>

Range = n2.entries.range}

Alg. 5.4.2. Fact removed from the constraints of the b+ tree structure

By generating some instances of the model, we obtain the following b+ tree structure:

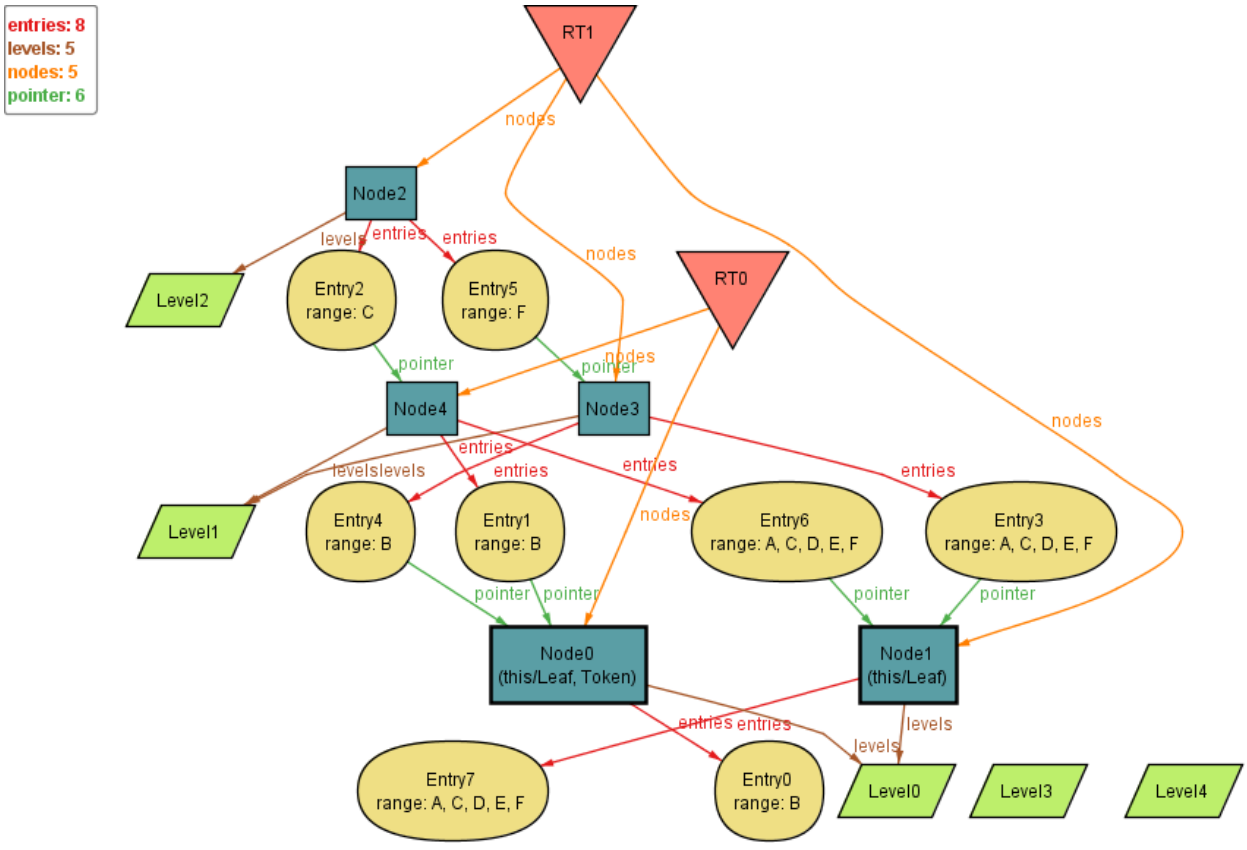


Fig. 5.4.2. Inconsistent b+ tree structure

We clearly see from the above example that the highest node at level2 of the routing table *RT1* does not comprise the universe of key values *U= {A,B.C,D,E,F}* within its entries. The reason is that we removed the constraint stating that for each routing table, there are always two successive nodes where the higher one contains the set of all the key values *U.* Therefore, the Alloy analyzer generated a model where *U* is in a node other that the highest level node, i.e. in an internal node at level 1 in the model above. Hence, the search operation would not be possible for the key values that are not described in the entries of the highest level node. The search operation in the b+ tree structure is thus incorrect. Similarly to A_U, if one of the facts describing the consistency invariants A_N *and* A_{LR} are removed, the model structure also becomes too weak and the distributed f b+ tree structure becomes inconsistent with search operations.

The next step would be to define a predicate of weak-consistency invariants, denoted *WCI,* and a predicate of minimally constrained architecture, denoted *MCA,* and finally a predicate describing consistent search, denoted *PCS*, in order to prove the following:

The weak consistency invariants are sufficient to ensure consistent search:
$MCA \wedge WCI \Rightarrow PCS$

The weak consistency invariants are necessary to ensure consistent search:
$PCS \Rightarrow MCA \wedge WCI$

If these two statements are true, we can conclude that the weak-consistency invariants A_U, A_N and A_{LR} are necessary and sufficient for the correctness of search operations on the distributed b+ tree structure. We did not have enough time to achieve this part and leave it for future work.

5.5 Conclusions and limitations

By using the Alloy modeling language, we were able to validate the weak-consistency properties that are sufficient for maintaining the b+ tree structure consistent with search operations. However, we did not have enough time to show that the weak consistency invariants are necessary to ensure consistent search and leave it for future work.
Since Alloy is declarative, it is difficult to express the insertion and deletion of some ranges since all the instances of each signature have to be declared in number before any assertion or operation can be made. For instance, for a b+ tree structure having two routing tables, if we would like to insert a new routing table, we have to fix the number of routing tables to three before the insertion operation is started. All the facts that maintain the routing table structure has to be applied to the first two routing tables only while the third added routing table structure must be kept initially empty. However, such restriction is complex to apply since Alloy is declarative and hence the declared facts are assumed to be true for all the routing tables. This is one of the main weakness of the Alloy modeling language that restraint its application to validate assumptions of static models only.
A second and important drawback of the Alloy modeling language is that the number of instances of any signature in a model instance cannot exceed 18 scopes. This is a serious limitation. For instance, the maximum number of instances that are allowed for Range signature describing the universe of key values U in the system is 18, which means that it is not possible to have more than 18 key values. As a consequence, we cannot draw a b+ tree structure with more than 18 routing tables since each routing table must have a unique key value in its leaf node. However, it is usually useful to check the validity of a system with a higher number of instances and therefore making the model more realistic.
In addition, representing the set of key values in Alloy as integer data-keys is not easy since Alloy does not support integers. The alternative was to describe the set of range as disjoint key values {A,B, C…} and deals with each key value as an elementary range that cannot be divided into sub-ranges. However, we were able to prove using the Alloy modeling language that it is possible to model the distributed b+ tree structure and validating the search operation under such restrictions.

6. Revised updates with weak-consistency

6.1 Transfer of responsibility mechanisms

The major challenge of the non-leaf node split and merge algorithms introduced in *Section 4.2* is to find an existing peer *j*, whose routing table at the same level RT_j^l, that is empty or already contains some entries covering some common range with RT_i^l . Such procedure is called transfer of responsibility mechanism and consists of an algorithm that partially or fully explores some neighboring peers in the decentralized b+ tree following a predefined convention to lead to a suitable peer *j*. We defined two algorithms to perform this operation. The first one is called Ping-Pong strategy while the second one is called Ping-Only strategy. Both of them were implemented in our simulation system described in Chapter 7.

6.1.1 Ping-Pong strategy

Starting from a non-leaf node RT_i^l, this strategy consists of going down to the peers responsible for the ranges within the entries of RT_i^l at the next lower level (ping), then going backwards to the peers responsible for nodes pointing to these entries at the next higher level, which leads us back to different node RT_j^l at the original level (pong). We can then check if such a peer has an empty node or holds the entries the initial peer i decides to transfer. The Ping-Pong method is composed of two queries: The ping query is performed by going down using the pointers of the ranges x_k in the transferring entries described by the tuples $<x_k,t_k>$ in RT_i^l. The query is thus directed to all routing tables RT_{tk}^{l-1} of the traversed peers t_k. Since the goal is to find an existing peer that can replace peer i at the same level l, the pong query, in contrast to the ping query, consists of going backwards using back-pointer tables, i.e. the list of all predecessor peers p maintaining a routing table RT_{pk}^l at the next higher level and identified by the backward pointer table BK_{tk}^{l-1}.

Therefore, the query reaches the explored peers j_k at the same level l and we can now check whether the node at that level is empty or contains entries with ranges to be transferred from RT_i^l. In the case that more than one suitable routing table RT_{jk}^l is available, only the first obtained peer j from all the j_k is chosen and the routing table RT_j^l of the peer j becomes responsible for the transferred entries $<x_k,t_k>$ from RT_i^l.

This operation may be repeated at larger depths, i.e. going down more than one, as long as the leaf level is not reached, and then going backwards the same number of levels so that we can reach peers at the original level. This increases in the number of explored peers and raises the chance of finding a peer j that is suitable. The number of traversed levels to perform the Ping-Pong strategy is called the order of traversal and denoted by s. Starting with order 1, the strategy consists of exploring the b+ tree and increasing the order if the last order did not reach any suitable peer j, until a suitable peer j is found. Notice that all the traversed nodes in the lower levels are intermediate nodes and their only role is to route the Ping-Pong query, however, the query does not check whether they share many information with the initiating node. Only the ending nodes at the same level are inspected when the Ping-Pong algorithm is executed. It may be noted also that the maximum possible order of traversal s is equal to the level l of the initiating node. The Ping-Pong algorithm is defined below.

Algorithm 6.1.1 Ping-Pong (i,l,s, x_k)

1: Initiator: peer i,

2: Condition: $l > s$

3: Readset = $\{RT_i^l, \forall\, l\, \forall\, s' < s\ \forall\ tk \in RT_i^{l-s'+1}, RT_{tk}^{l-s'}, BK_{tk}^{l-s'}\ ,\ \forall\, l\, \forall\ s' < s\, \forall\ \forall\, tk' \in\ RT_{tk}^{l-s'},\ RT_{tk'}^{l-s'-1},\ BK_{tk'}^{l-s'-1}\}$

4: Action:

5: $s' = 1$

6: follow the pointer associated with the searched range x_k described by the tuples $<x_k,t_k>$ in $RT_i^{l-s'+1}$ and direct the query to the routing table $RT_{tk}^{l-s'}$

7: while s' < s do

8: $\forall\, tk \;\forall\, tk' \in RT_{tk}^{l-s'}$ do

9: execute breadth-first search ($RT_{tk}^{l-s'}$)

10: s' <- s'+1

11: end do

12: end while

13: while s' > 1 do

14: $\forall\, tk \;\forall\, tk' \in RT_{tk}^{l-s'}$ do

send the Search query backwards to the next higher level nodes $RT_{tk}^{l-s'+1'}$ using backward pointer table $BK_{tk\prime}^{l-s\prime}$: execute inverse breadth-first search ($BK_{tk\prime}^{l-s\prime}$)

15: s' <- s'-1

16: end do

17: end do

18: $\forall\, RT_{tk}^{l-s'+1}$do

$if\;\; RT_{tk}^{l-s'+1}\; is\; empty\; or\; xk \sqsubseteq range(RT_{tk}^{l-s'+1})$ then send OK to RT_i^l

19: end do

Alg. 6.1.1. Ping-Pong algorithm

Fig. 6.1.1 shows step by step the Ping-Pong method of order 1 for transferring the range [25, 40] from RT_A^1. Starting from the node RT_A^1 (1), RT_A^1 uses the pointer that correspond to the range [25, 40] , which is peer C and redirects the Ping-Pong query to a node of peer C at the next lower level RT_C^0. The query is then sent backwards to all the back-pointers A, B, C at level 1 using the back-pointer table BK_C^0(2). The Ping-Pong query is now at the nodes RT_A^1, RT_B^1 and RT_C^1 at the same level as the initiating node RT_A^1. Thus, we can check if theses nodes are empty or contain the transferring range [25, 40]. We see that all these nodes contain the corresponding range. Since peer A is looking for another peer j different of peer A to become responsible of the transferring range, the query in RT_A^1 is ignored and one of the two left nodes RT_B^1 or RT_C^1 is chosen to be responsible for the transferred range (3).

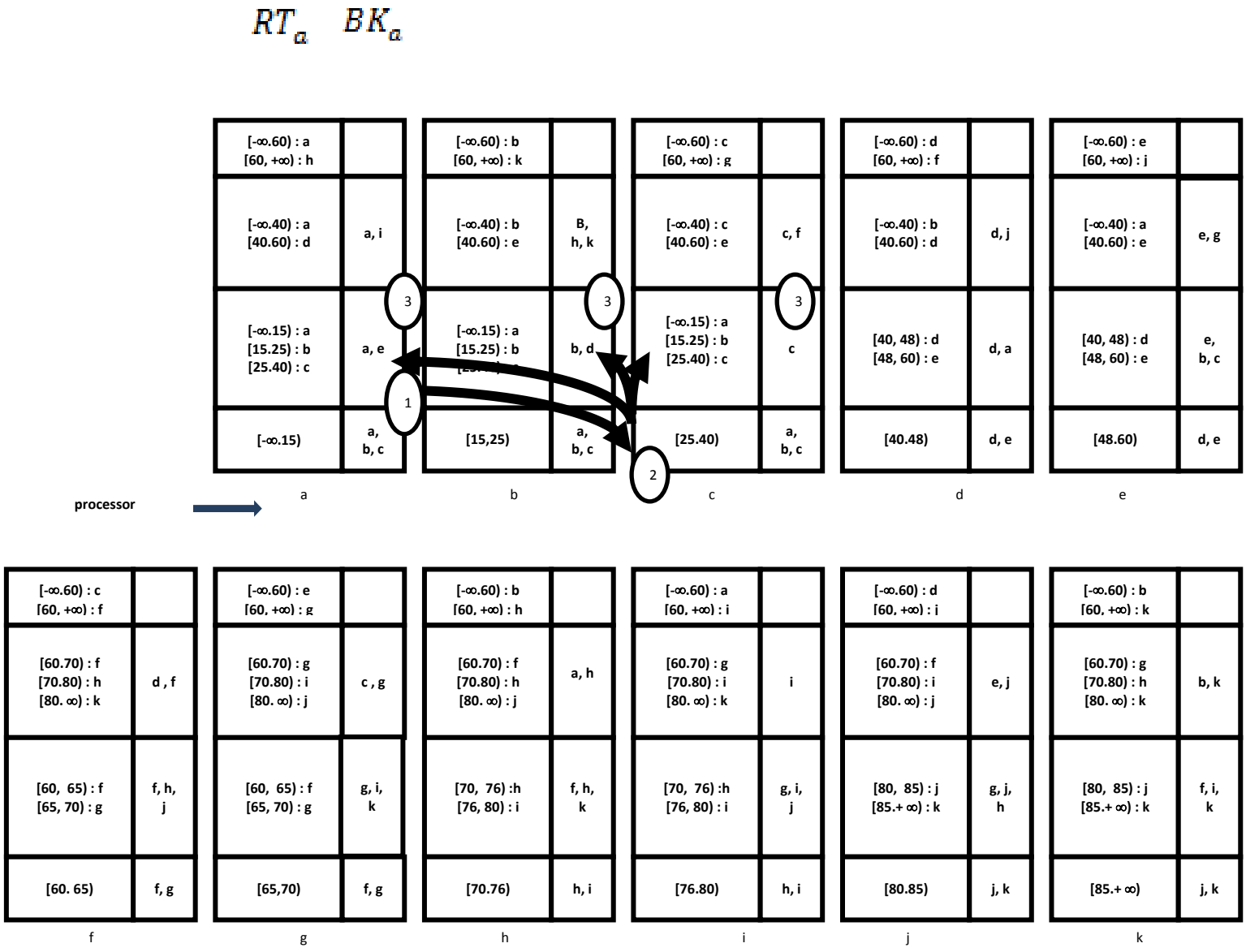


Fig. 6.1.1.a. Ping-Pong method of order 1 for transferring the range [25, 50] in RT_A^1.

Let us know take another example where the order of traversal is larger than 1.

Fig. 6.1.1.b illustrates an example of the Ping-Pong method of order 2. The transferring range is [40.60] and the initiating node is RT_A^2. Starting from the node RT_A^2 (1), peer A uses the pointer that correspond to the range [40, 60] , which is peer D and redirects the Ping-Pong query to a node of peer D at the next lower level and reaches RT_D^1. Similarly, peer D uses the pointer that correspond to the sub-ranges [40, 48] and [48,60] and redirects the Ping-Pong query to peer D and peer E at the next lower level and reaches RT_D^0 and RT_E^0 (2). The queries are then sent backwards to all the back-pointers at level 0 of both RT_D^0 and RT_E^0 using the back-pointer table

BK_D^0 and BK_E^0 (3). The Ping-Pong query is now in $RT_D^1$and RT_E^1. Finally, the queries are again sent backwards to all the back-pointers at level 1 of both RT_D^1 and RT_E^1 using the back-pointer tables BK_D^1 and BK_E^1 (4), and the pong-pong query reaches its final destination nodes RT_A^2, RT_B^2, RT_C^2, RT_D^2 and RT_E^2 at the same level as the initiating node RT_A^2.

Thus, we can check if these nodes are empty or they contain the transferring range [40, 60]. We see that they all contain the corresponding range. Since peer A is looking for another peer j different of peer A to become responsible of the transferring range, the query in RT_A^2 is ignored and one of the other nodes RT_B^2, RT_C^2, RT_D^2 and RT_E^2 is chosen to become responsible for the transferred range (5).

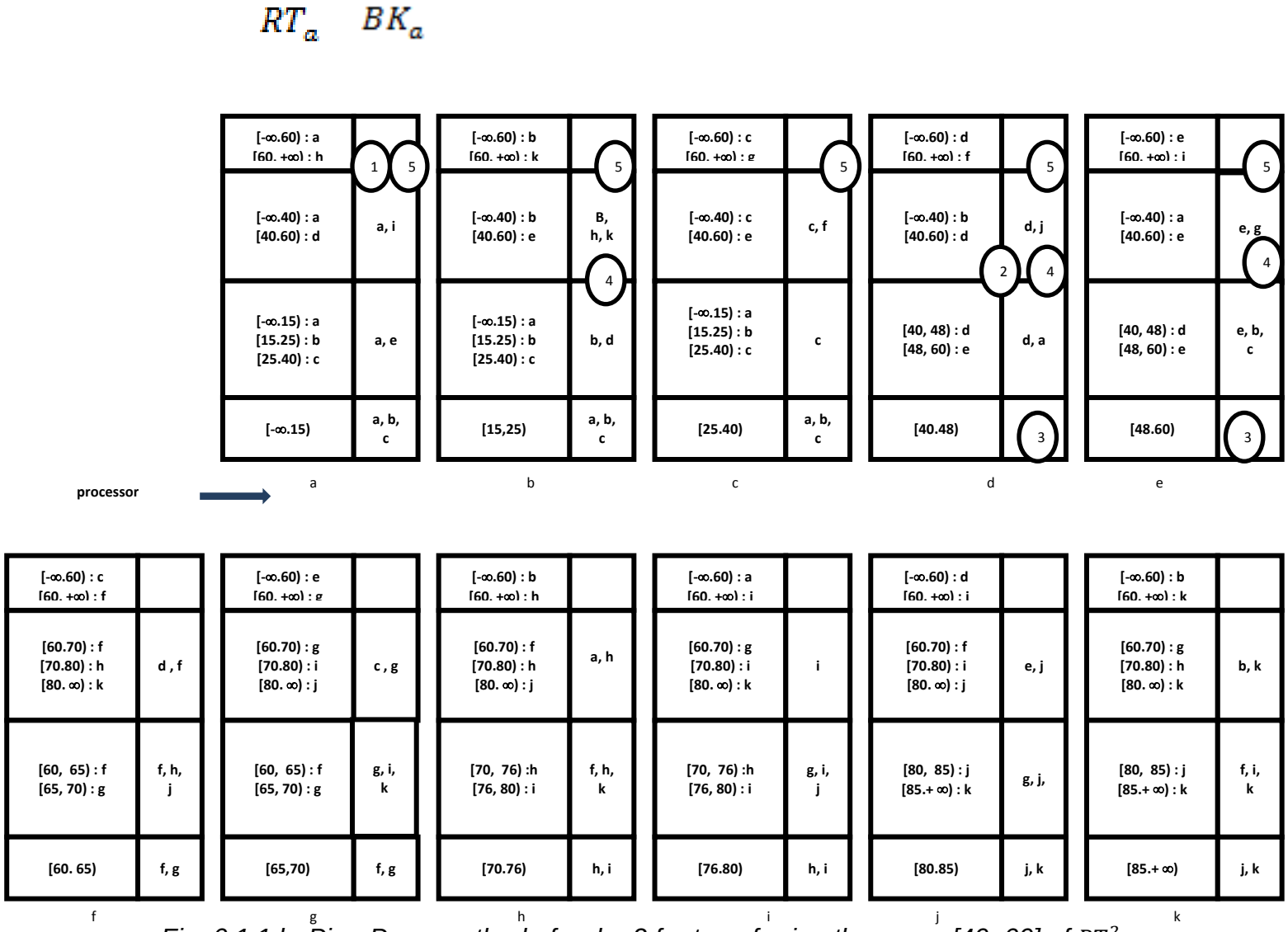


Fig. 6.1.1.b. Ping-Pong method of order 2 for transferring the range [40, 60] of RT_A^2.

6.1.2 Ping-Only strategy

A major drawback of the Ping-Pong algorithm introduced in *Section 6.1.1* is that all the intermediate nodes at the lower levels are only used for routing the query and are not checked for common information at level l with the initiating node RT_i^l. Only the nodes at the same level are inspected. In order to overcome these weaknesses, we introduce the Ping-Only algorithm that consists of only going down to the peers responsible of transferring the ranges at a the next lower-level, and for each traversed node $RT_k^{l'}$ with l' < l, the node at the same level of the initiating node RT_k^l is checked empty space or common information. This means that all intermediate peers inspects their nodes at the same level as the initial node RT_i^l which could be more efficient for finding a suitable peer j.

The Ping-Only algorithm is an improvement to the Ping-Pong algorithm in terms of both message and time complexity since the traversal is only performed in one way to the next lower levels.

The Ping-Only algorithm is introduced below.

Algorithm 6.1.2 *Ping-Only(i,l, x_k)*

1: Initiator: peer i

2: Condition: l > 0 , the transferring range $x_k \neq \emptyset$

3: Readset = {RT_i^l, $\forall\, l\, \forall\, s \;\; < l \;\; \forall \;\; tk \in RT_i^{l-s\;+1}$, RT_{tk}^{l-s} , $\forall\, l\, \forall\; s \;\; < l \;\; \forall\, tk' \in RT_{tk}^{l-s}$, $RT_{tk'}^{l-s^{-1}}$ *}*

4: Action:

5: s = 1

6: while s ≤ l and no OK received by RT_i^l *do*

7: $\forall\, l\, \forall\, s \;\; < l \;\; \forall\, tk' \in RT_{tk}^{l-s}$ *do*

8: execute breadth-first search *(*$RT_i^{l-s'+1}$*)*

9: $if\ \ RT_{tk}^{l}$ *is empty or* $x_k \sqsubseteq\ RT_{tk}^{l}$ *then send Ok to* RT_i^l

10: *else s <- s+ 1*

11: *end if*

12: *end do*

13: end do

Alg. 6.1.2. Ping-Only algorithm

Fig. 6.1.2 illustrates an example of the Ping-Only method. The transferring range is [40.60] and the initiating node is RT_A^2. Starting from the node RT_A^2 (1), peer A uses the pointer that correspond to the range [40, 60] which is peer D and redirects the query to peer D by going to the next lower level and reaches RT_D^1. Peer D checks locally its node at level 2 (at the same level as the initiating node RT_A^2) which corresponds to RT_D^2, and verifies if it is empty or it contains the transferring range [40, 60] . We can see from Fig. 6.1.2 that RT_D^2 contains the latter range. Therefore, Peer D is chosen for the transfer of responsibility from RT_A^2 for the range [40, 60] and the Ping-Only operation is terminated.

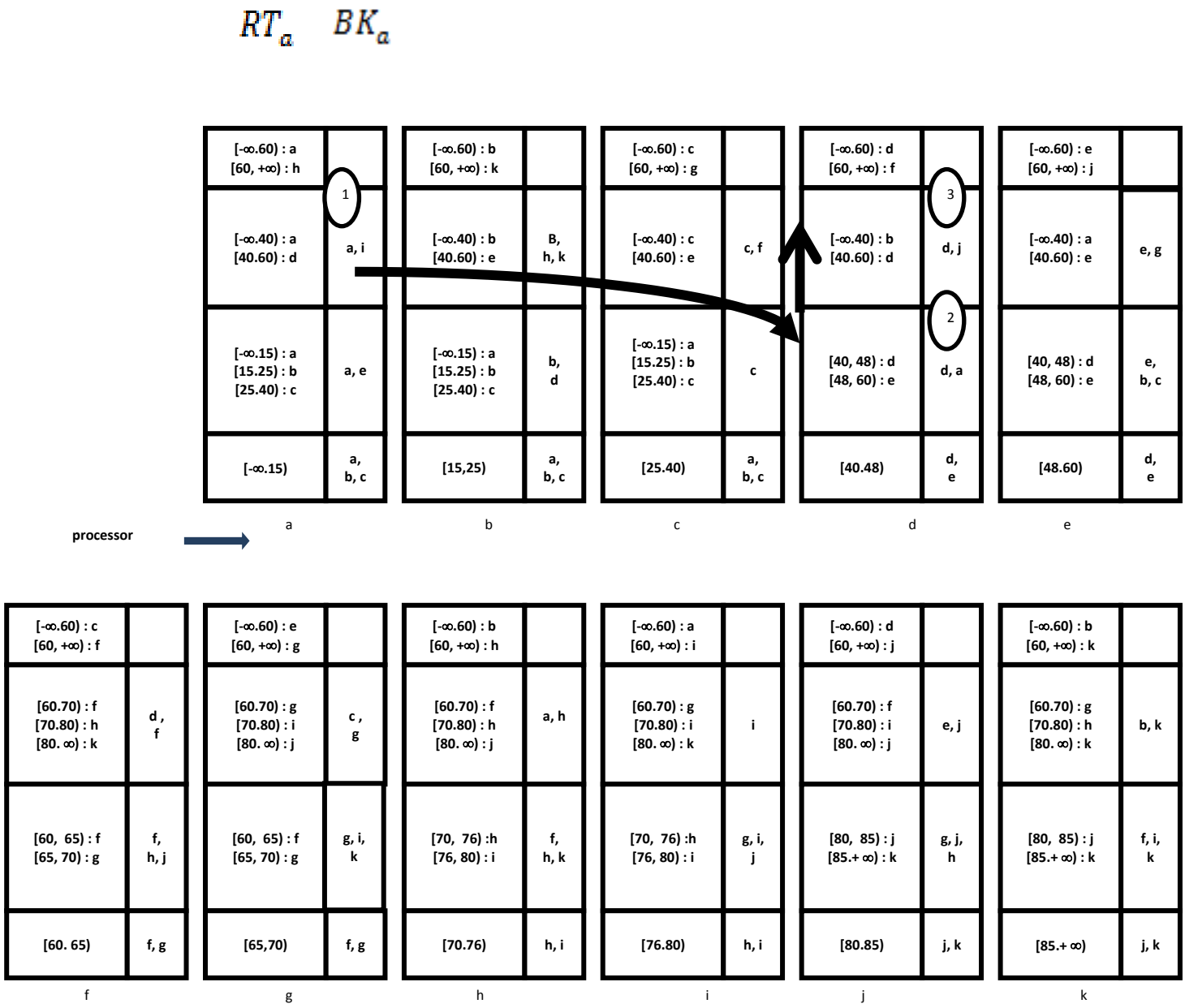


Fig. 6.1.2. Ping-Only method for transferring the range [40, 60] from RT_A^2.

6.2 Modified version of the split and merge algorithms

In addition to the transfer of responsibility mechanisms introduced in Section 6.1, the split and merge algorithms described in Section 4.2 need to be modified as we observed some inaccuracy when updating the distributed b+ tree structure. These modifications as well as the revised algorithms are described in the following subsections.

6.2.1 Split with weak-consistency

6.2.1.1 Leaf node split

In the leaf node split algorithm Alg. 4.2.1, we take the following statements under reconsideration. Line 13 updates all the back-pointers BK_i^0 that are pointing to the split leaf node RT_i^0. The update consists of splitting the entry corresponding to the tuple <x,i> of back-pointer RT_p^1, where p ∈ BK_i^0, into two disjoint entries $< x \cap LR_1 , i >$ and $< LR_1 \setminus x, j >$. In addition, line 14 adds the new introduced peer j to the back-pointer table BK_p^1. However, these two lines are restricted to the condition that x ⊑ LR_i^0. We think that this condition is irrelevant if we assume the consecutiveness constraint. This constraint assume that the split of a given range results in two disjoint consecutive ranges and the merge operation, in contrary to the leaf node split, can be performed only if the merger of local ranges between the initiating peer and the other peer are consecutive and can be combined into one single local range Therefore, all back-pointers in BK_i^0 must describe the whole range x associated with the pointer i to RT_i^0 thereby x = LR_i^0 and there is no case where x ⊏ LR_i^0. Thus, the corresponding condition in line 13 is removed. Therefore, lines 12, 15 and 16 can be removed from the leaf node split algorithm of Section 4.2.1. We note that the consecutiveness constraint was not assumed in [1] where the algorithm was first described.

6.2.1.2 Non-leaf node split

Line 6 of the algorithm Alg. 4.2.2 consists of off-loading some entries from the routing table. RT_i^l at level l > 0 by finding a suitable peer j, whose routing table at the same level, RT_j^l, either already contains some entries covering some common range R_x with RT_i^l or has some space to take a few entries from RT_i^l. If many peers sharing the common range R_x are available, R_x may be partitioned into multiple sub-ranges R_{xk} according to the multiple j_k peers chosen. However, we are not sure whether the b+ tree structure remains consistent with the search operation or if there are some other concerns to take into consideration due to sub-range multiplicity of R_{xk}. For this reason, we ignored the case when R_x can be partitioned into multiple sub-ranges R_{xk} and we consider only the case where R_x is not partitioned. We have defined two transfer of responsibility mechanisms in Section 6.1.1. Both of these algorithms consider only the case when that the transferring is not repartitioned. Again, the consecutiveness constraints is assumed, therefore the requested common range R_x must be partly covering the right side or the left side of the range in RT_i^l and can never represent a middle overlap. The related lines from line 6 to line 23 are also modified accordingly and some related notations are also changed accordingly: Line 6 executes the one of the transfer of responsibility algorithms. Line 7 partitions RT_i^linto two disjoint subsets E_s^l and E_e^l, and partition RT_i^l into disjoint ranges R_s^l and R_e^l, where E_s^l refers to the entries to be kept and E_e^l refers to the entries to be transferred. Notice that the decision of which side to be transferred is made by RT_i^l. The other lines are similar to the original version but consider the partition of the range to be transferred into only two ranges rather than a multiple number of ranges.

The revised parts from the non-leaf node split algorithm are presented below:

Algorithm 6.2.1.2 SplitNonLeafNode(i,l)

6 : execute Algorithm Alg. 6.1.1 or Alg. 6.1.2 to find j | s.t RT_j^l is empty or RT_j^l has some overlap denoted E_e with RT_i^l

7 : Partition RT_i^linto two disjoint subsets E_s^l and E_e^l, and partition RT_i^l into disjoint ranges R_s^l and R_e^l. accordingly

8 : There must exist $E_e^l \in RT_j^l$. If RT_j^l is empty, RT_j^l <- E_e^l

9 : RT_i^l <- E_s^l

11 : $\forall P \in RT_i^l, \forall P_e \in E_e^l, BK_p^{l-1}$ <- $BK_p^{l-1} \setminus \{P_e\}$

12: $\forall P \in BK_i^l$ do

13: There must exist <x,i> $\in RT_p^{l+1}$

14: If $if\ x \cap R_e^l \neq \emptyset$

15: RT_p^{l+1} <- $RT_p^{l+1} \setminus < x, i >$

16: RT_p^{l+1} <- $RT_p^{l+1} \cup < x \cap R_e^l, i > \cup < R_e^l, j >$

17: BK_j^l <- $BK_j^l \cup \{P\}$

18: end if

19 : If RT_i^l is the highest level in RT_i

20 RT_i^{l+1} <- $< R_s^l, i > \cup < R_e^l, j >$

21 end if

Alg. 6.2.1.2. Revised non-leaf node split algorithm

6.2.2 Merge operation

6.2.2.1 Leaf Node merge

In the merge leaf node algorithm described in Alg.4.2.3, line 6 is executed by the merged peer i and consists of finding a suitable partner j for the merged found in RT_p^1 , where p points to i for some range x ∈ LR_i^0, If RT_p^1 is pointing to j for some other range y, then after the merged, the two entries <x, i> and <y,j> can be merged into <x U y, j>. However, in order to merge the two local ranges x and y into one local range, the local ranges of the merging peers have to be successive so that we can combine them into one local range. Hence, line 6 is adjusted accordingly.

Additionally, the use of separate transactions involves all merge operations are atomic and only the next higher level node RT_i^1 has to be checked and merged as necessary after merging the leaf node of the peer I and not all the higher-level nodes. In fact, the operation of merging all the higher level entries from RT_i^l with RT_j^l regardless the overlap between them is no more accurate under the consecutiveness assumption since a range interruption may occur among the new merged ranges. This assumption involves that the other atomic operations in the internal nodes should be handled and initiated only by the preceding internal nodes after invoking a merge operation. Therefore, line 6 is modified accordingly.

As a consequence, line 10 releasing the peer i becomes invalid after a leaf node merge operation under the assumptions that every merge operation consists of a separate transaction and thus all the nodes of the peer i must be empty for the release. Such operation should be achieved in a separate transaction corresponding to the non-leaf node being merged. Additionally, merging the higher level nodes by peer i before being released is not a good approach since these nodes may not be under-loaded and thus performing a non-leaf node merge may results in off-loading the resulting node once the merger is completed. This operation will cost an additional non-leaf node split in the worst case after each non-leaf merge which makes it not efficient.

Therefore, line 10 is replaced by another statement that calls the Ping-Only algorithm on all the higher level nodes of peer i to search for another peer j that is empty or already contains the range of RT_i^l , i.e. $range(RT_i^l) \subseteq range(RT_j^l)$.

. In both cases, peer j will keep the same number of entries as before and thus will not be updated. The releases of peer i will also be realized by each merger if he finds out that all the nodes of its corresponding peer, including the leaf node, are empty. Notice that if a leaf node merge occurs, the highest level node becomes useless since the corresponding peer is being released. All the existing entries are therefore removed from RT_i^m(line 8).

The revised parts from the leaf node merge algorithm are presented below:

Algorithm 6.2.2.1 MergreLeafNode(i)

6: select j | $\forall P \in BK_i^0, < x, j > \in RT_p^1$ and LR_i *and* LR_j *are successive*

8 : RT_i^m <- ∅

9: $\forall l$ RT_i^l do

10: execute Algorithm Alg. 6.1.2 to find j | s.t RT_j^l is empty

or $range(RT_i^l) \subseteq range(RT_j^l)$

11: end do

Alg. 6.2.2.1. Revised leaf node Merge algorithm

6.2.2.2 Non-leaf node merge

Similar to the leaf node merger, only the next higher level node i.e. If RT_i^{l+1} , after merging a given node RT_i^l, has to be checked and merged as necessary and not all the higher nodes. The other atomic operations in the internal nodes should be handled and initiated by the preceding internal nodes after invoking a merge operation and follows the restriction of successiveness. Therefore, line 10 is modified so that only the node RT_i^{l+1} is merged if it has too few entries. In addition, if RT_i^l finds out that all the nodes in peer i including the leaf node are empty, the peer i is released. Therefore, line 16, 17 and 18 are added accordingly.

The revised parts from the non-leaf node merge algorithm are presented below:

Algorithm 6.2.2.2 MergreNonLeafNode(i.l)

6: select j | $\forall P \in BK_i^l, < x, j > \in RT_p^{l+1}$ and the ranges of $RT_i^l and\ RT_j^l are\ successive$

7: RT_j^l <- $RT_i^l \cup RT_j^l$

8: if l is not the top-most level of RT_i then

9: RT_i^l <- $\emptyset$

10 : merge node RT_i^{l+1} as necessary if it contains too few entries

11: end if

12: If for any $p \in BK_i^l$, RT_p^{l+1} is the top-most level of RT_p and contains only one entry pointing to p then

13: Delete RT_p^{l+1} and remove p from BK_p^l

14: end if

15 : BK_i^l <- $\emptyset$

16 : If for $\forall\, l$, $RT_i^l = \emptyset$

17: Release peer i

18: end if

Alg. 6.2.2.2. Revised non-leaf node Merge algorithm

7. Simulation of the b+ tree with weak-consistency

Modeling and analysis are important for studying distributed computing systems and can be classified as structural or behavioral modeling. Structural modeling focuses on the organization of the network and its components while behavioral modeling focuses on the network dynamics. We presented in Chapter 4 a structural model of the decentralized distributed b+ tree using the Alloy modeling language in order to explore the validity of static properties of the distributed b+ tree with weak-consistency, it also gave us a good amount of confidence that the b+ tree structure under the weak-consistency conditions maintains correct search operations. However, such a model is inadequate for the study the dynamics of the P2P network which changes over time since data insertions and deletions occur such that the structure of the b-tree may change. Such analysis can only be conducted effectively using simulation. The simulation can be represented as an abstract model of the distributed b+ tree structure that captures the network features, properties and characteristics to facilitate the analysis, and thus can be useful to predict the system performance in terms of both message complexity and time delays. We need to investigate the dynamic behavior when performing an insert, delete or update operation on the distributed b+ tree through simulation to observe how the configuration of the peers and their corresponding routing tables adapt to such changes.

7.1 Programming environment

The simulation software that I developed is written in the Java programming language using the Helios Service Release 1 of the Eclipse software development environment. The reason why I chose Eclipse is that it comprises an integrated development environment (IDE) and an extensible plug-in that makes it efficient for simulation-based methods measurements and analysis. For the purpose of simulation, we used the Java SSIM simulation package [14] available at
http://www.inf.usi.ch/carzaniga/ssim/index.html

SSIM is an object-oriented utility library written in java that provides discrete event process-based simulation. SSIM is available free to users since 2003. SSIM implements a simulator to run reactive discrete-time processes. These processes execute actions in terms of discrete execution steps performed at given times in response to an event where events describe a piece of information exchanged between two processes through the simulator. During the execution of an action, a process may schedule other future actions for itself, or it may signal events to other processes, which will respond by processing the signaled events at the given further time. The simulation terminates when no more actions are scheduled. These processes are defined by the interface Process and must be implemented by a simulated process. The SSIM library defines the basic interface of a process, and provides the main simulation scheduler, including methods for creating, starting, and stopping processes, and for scheduling events or signaling other events. For instance, a process can start an action immediately once being created or can stop an action being executed. It can also execute an action in response to an event signaled to its process by another process or execute an action in response to a timeout. Alg. 7.1.1 introduces the methods that may be used by the simulated processes to implement the interface Process.

```
//action executed when the process is created.
public void init() {
	}

//action executed in response to an event signaled to this process
public void process_event(Event event) {
	}

// action executed in response to a timeout.
public void process_timeout() {
	}

// action executed when the process is explicitly stopped.
public void stop() {
	}
```

Alg. 7.1.1. Methods used by the simulated processes to implement the interface Process.

The class that realizes the generic discrete-event sequential simulator is denoted by Sim. Sim maintains and executes a time-ordered schedule of discrete events at different virtual times. In case some events are scheduled at the same virtual time, the simulator is asked to queue these events as a priority queue, i.e. first in first out fashion. The Simulator is also responsible for starting and stopping the simulation. The class Sim compromises a set of methods that can be used for creating and releasing processes, scheduling and signaling events or running and stopping the simulation. For instance, the simulator can create, schedule, stop and delay a current process at a given time. It can also signal an event to be executed at a given time.

Alg. 7.1.2 introduces the methods used by the class Sim.

```
//Advance the execution time of the current process by a given delay
static void advance_delay(long delay)

// Resets the simulator making it available for a completely new simulation.
static void clear()

// Returns the current virtual time for the current process.
static long clock()

//Creates a new process creates a new process for a  given simulated process.
static long create_process(Process process)

//Run the simulation
static void run_simulation()

// Sets a timeout for the current process after a given amount of (virtual) time t.
static void set_timeout(long t)



// Signal an event to the given process that identified by pid.
static int signal_event(Event evt, long pid)

// signal an event to the given process that identified by pid after a given amount of (virtual) time delay.
static int signal_event(Event evt, long pid, long delay)

// Stops the execution of the current process.
static void stop_process()
```

```
//Stops the execution of a given process that is identified by pid.
static int stop process(long pid)

// Stops execution of the simulation
static void stop simulation()

// Returns the pid of the current process
static long this process()
```

Alg. 7.1.2. Methods used by the class Sim

In this project, we keep the study of the distributed b+ tree with weak-consistency in a real peer-to-peer system for future work and we focus on the simulation part of the program that uses the Simple Discrete-event Simulation Library SSIM.

This implementation is tested on an Intel(R) Core(TM) 2 Duo CPU T6500 Peer running at 2.10 GHZ. The system has a 32 bit processor with a RAM Memory of 4.00 GB.
For figures and Charts, I used the MATLAB numerical computing environment. Developed by MathWorks, MATLAB allows plotting of functions and data, creation of user interfaces and interfacing with programs written in other languages , including C, C++ , Java and Fortan.

For the simulation results, I limited the number of simulated peers to 1000. The reason I chose a maximum of 1000 peers is that it is easy and fast to test and verify the results of a network having less than 1000 peers. For a network having more than 1000 peers, the execution time takes somewhat longer but we presume that the measurements on a system with no more than 1000 peers are sufficient and consistent with the ones using large-scale network databases.

7.2 Design of the b+ tree simulation system

The simulation software is essentially composed of three main classes: the Peer, the User and the P2PSim, the main class that represents the simulation environment. The different classes interact with each other in the system by means of messages and interfaces as shown in the following diagram:

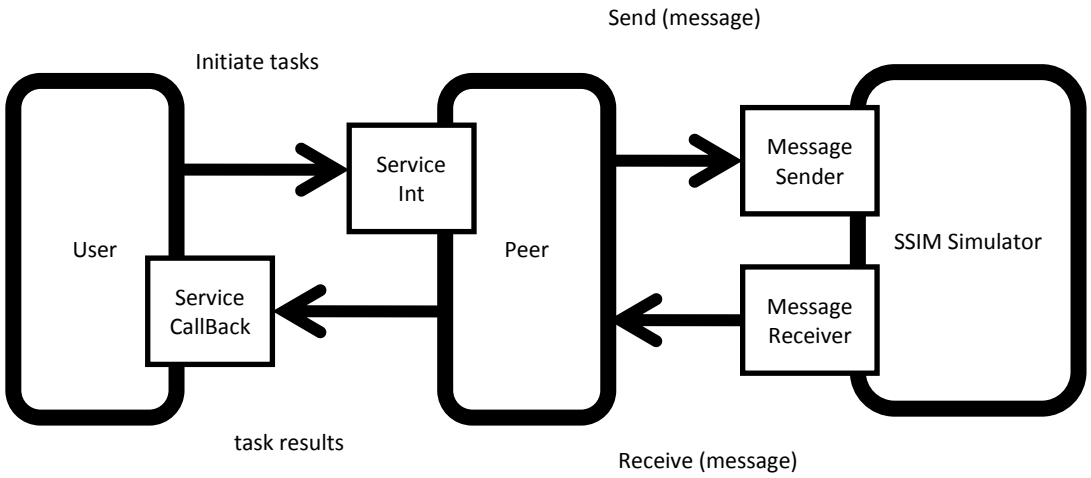


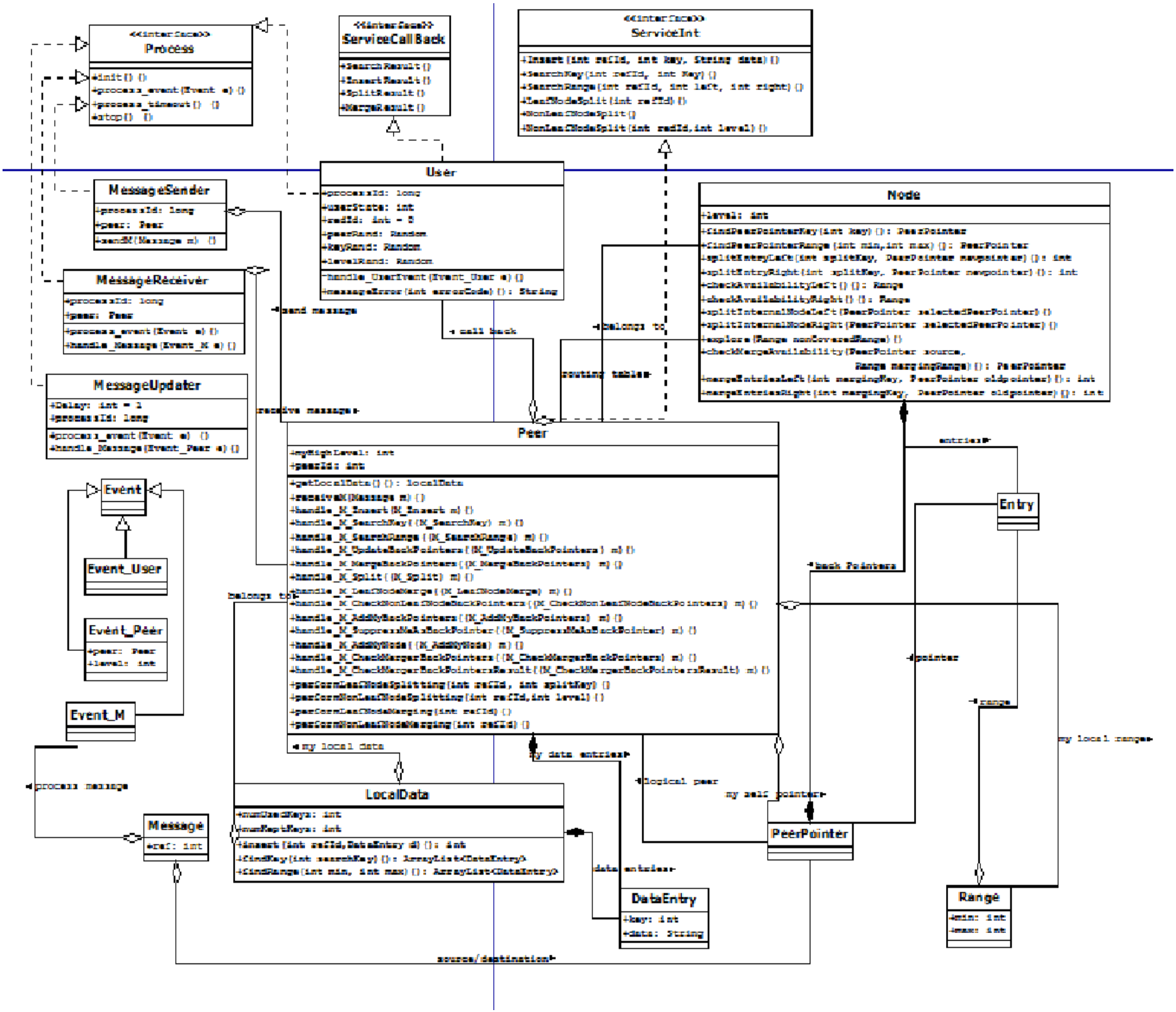
Fig.7.2.1. Interaction between the main classes in the Simulation environment

We define the architecture, the specifications and requirements as well as the behavior of the distributed b+ tree simulation system using the Unified Modeling Language UML [19]. Developed by the Object Management Group OMG [19]. UML is a standardized modeling language that defines the notation and semantics of object-oriented software systems. The OMG group defines the UML language as follows:

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

We use a UML Class diagram to model the distributed b+ tree structure. The Class diagram is used to represent the static structure of the system model using objects, attributes, operations and relationships.

The class diagram describes the architecture of the system by illustrating the system's classes, their attributes, and the relationships among these classes. Each class in the class diagram is represented by a rectangle that is divided into three compartments: the name of the class, the class attributes as well as the class methods. Additionally, the classes are linked together if necessary depending on the relationships between them. The complete distributed decentralized b+ tree Class diagram is shown in Fig. 7.2.2. .


«Interface»
Process
+init()
+process_event(Event e)
+process_timeout()
+stop()
«Interface»
ServiceCallBack
+SearchResult()
+InsertResult()
+SplitResult()
+MergeResult()
«Interface»
ServiceInt
+Insert(int refId, int key, String data)
+SearchKey(int refId, int Key)
+SearchRange(int refId, int left, int right)
+LeafNodeSplit(int refId)
+NonLeafNodeSplit()
+NonLeafNodeSplit(int refId,int level)
User
+processId: long
+userState: int
+refId: int = 0
+peerRand: Random
+keyRand: Random
+levelRand: Random
+handle_UserEvent(Event_User e)
+messageError(int errorCode): String
MessageSender
+processId: long
+peer: Peer
+sendM(Message m)
MessageReceiver
+processId: long
+peer: Peer
+process_event(Event e)
+handle_Message(Event_M e)
MessageUpdater
+Delay: int = 1
+processId: long
+process_event(Event e)
+handle_Message(Event_Peer e)
Node
+level: int
+findPeerPointerKey(int key): PeerPointer
+findPeerPointerRange(int min,int max): PeerPointer
+splitEntryLeft(int splitKey, PeerPointer newpointer): int
+splitEntryRight(int splitKey, PeerPointer newpointer): int
+checkAvailabilityLeft(): Range
+checkAvailabilityRight(): Range
+splitInternalNodeLeft(PeerPointer selectedPeerPointer)
+splitInternalNodeRight(PeerPointer selectedPeerPointer)
+explore(Range nonCoveredRange)
+checkMergeAvailability(PeerPointer source,
Range mergingRange): PeerPointer
+mergeEntriesLeft(int mergingKey, PeerPointer oldpointer): int
+mergeEntriesRight(int mergingKey, PeerPointer oldpointer): int
Peer
+myHighLevel: int
+peerId: int
+getLocalData(): localData
+receiveM(Message m)
+handle_M_Insert(M_Insert m)
+handle_M_SearchKey((M_SearchKey) m)
+handle_M_SearchRange((M_SearchRange) m)
+handle_M_UpdateBackPointers((M_UpdateBackPointers) m)
+handle_M_MergeBackPointers((M_MergeBackPointers) m)
+handle_M_Split((M_Split) m)
+handle_M_LeafNodeMerge((M_LeafNodeMerge) m)
+handle_M_CheckNonLeafNodeBackPointers((M_CheckNonLeafNodeBackPointers) m)
+handle_M_AddMyBackPointers((M_AddMyBackPointers) m)
+handle_M_SuppressMeAsBackPointer((M_SuppressMeAsBackPointer) m)
+handle_M_AddMyKeys((M_AddMyKeys) m)
+handle_M_CheckMergerBackPointers((M_CheckMergerBackPointers) m)
+handle_M_CheckMergerBackPointersResult((M_CheckMergerBackPointersResult) m)
+performLeafNodeSplitting(int refId, int splitKey)
+performNonLeafNodeSplitting(int refId,int level)
+performLeafNodeMerging(int refId)
+performNonLeafNodeMerging(int refId)
Event
Event_User
Event_Peer
+peer: Peer
+level: int
Event_M
Message
+ref: int
LocalData
+numUsedKeys: int
+numKeptKeys: int
+insert(int refId,DataEntry d): int
+findKey(int searchKey): ArrayList<DataEntry>
+findRange(int min, int max): ArrayList<DataEntry>
DataEntry
+key: int
+data: String
PeerPointer
Entry
Range
+min: int
+max: int
send message
call back
belongs to
routing tables
receive messages
entries
back Pointers
pointer
range
my local ranges
belongs to
my local data
my data entries
logical peer
my self pointer
data entries
process message
source/destination

The main classes shown in Fig. 7.2.2 and their relationships may be described as follows:

Peer Class and relationships

The peer class comprises the simulated peers executing given tasks that may be requested by users. In order to initiate a given task, the Peer class implements the ServiceInt interface containing the list of tasks that can be initiated by users. Alg.7.2.1 shows the methods used by the interface ServiceInt.

```
public interface ServiceInt {

            //this method inserts a data with a given key
            public void Insert(int refId, int key, String data);

            //this method deletes a data associated with a given key
            public void DeleteKey(int refId, int Key);

            //this method search for a given key
            public void SearchKey(int refId, int Key);

            //this method search for all the keys in a given range
            public void SearchRange(int refId, int left, int right);

            //this method splits the Local Range in the vicinity of the mean
            key
            public void LeafNodeSplit(int refId);

            //this method splits the Range of a node in a given level in the
            vicinity of the closest border to the mean key
            public void NonLeafNodeSplit(int redId,int level);

            //this method merge the local range of a leaf node
            public void LeafNodeMerge(int redId);
```

```
        //this method splits the Range of a node in a given level in the
        vicinity of the closest border to the mean key
        public void NonLeafNodeMerge(int redId,int level);


}
```

Alg.7.2.1. Methods used by peers to implement the interface ServiceInt

These tasks are performed collaboratively by several peers. In order for the peers to communicate with one another in the system. three classes, MessageSender, MessageReceiver and MessageUpdater implement the interface Process and play the role of a communicating actor for the peers in the simulation environment.

Indeed, each peer is associated with a message sender, an instance of the class MessageSender, used to send messages, and a message receiver, an instance of the class MessageReceiver, to receive messages. Additionally, an instance of the class MessageUpdater is used to schedule an update event to a self node at a given level. When a message is given to the message sender, a message event is created and scheduled by the simulator. The event is of the class Event_M, which corresponds to a message sent by a peer, and is inherited from the class Event. In fact, peers communicate with one another by means of messages and each message corresponds to an instance of the Message class. When a message is sent, a new event is scheduled at a further time. This event is then processed by the message receiver of the destination peer at the scheduled virtual time and delivered to the corresponding destination peer instance. Notice that for performance purposes, we have chosen to maintain the message propagation delay between peers to zero so that each message sent by a source peer is immediately received by the destination peer or by a given user with no delays. In addition to the messages sent by peers, two other classes inherited from Event are used: MessageUser and Message_Updater. MessageUser corresponds to an external query made by the user that used the ServiceInt interface in the simulation environment. Message_Updater is used by a peer each time its number of

entries changes at a given level of the routing table. In the case that this number is less than the minimum number of entries allowed, a merge operation is performed. In contrary, when a node of the peer is over-loaded in terms of entries, a split operation is performed on that node.

Alg.7.2.2. shows the attributes and the methods used by the message senders to implement the interface Process.

```
public class MessageSender implements Process{

	//describes the process identification of the message receiver in the
	//simulation environment
	public long processId;

	// describes the destination peer to receive the message sending
	//event
	public Peer peer;

	//constructor
	public MessageSender(Peer peer){
		this.peer = peer;
		processId = Sim.create_process(this);

	}

	//call the sendM on the appropriate Peer instance

	public void sendM(Message m) {

		//signal the message event on the appropriate MessageReveiver
		instance

		Event e = new Event_M(m);
		m.dest.peer.receiver = new MessageReceiver(m.dest.peer);
```

```
Sim.signal_event(e,m.dest.peer.receiver.processId,P2PSim.MEAN_MES
SAGE_PROPAGATION_DELAY);
```

Alg. 7.2.2. Attributes and methods used by message senders to implement the interface Process.

Alg.7.2.3. shows the attributes and the methods used by the message receivers to implement the interface Process.

```
public class MessageReceiver implements Process {

      //describes the process identification of the message receiver in the
      simulation environment
      public long processId;

      // describes the destination peer to receive the message reception
      //event
      public Peer peer;

      //constructor
      public MessageReceiver(Peer peer){
            this.peer = peer;
             processId = Sim.create_process(this);
      }


      //Process the message reception event e
      public void process_event(Event e) {

            //handle the message reception event if it is of type message
      event Event_M
```

```
        if(e instanceof Event_M) handle_Message((Event_M)e);
    }

    //call the reveiveM on the appropriate Peer instance
    private void handle_Message(Event_M e){
        Peer destination =e.message.dest.peer;
        destination.receiveM(e.message);
                }
```

Alg. 7.2.3. Attributes and methods used by message receivers to implement the interface Process.

Additionally to the MessageSender and MessageReceiver classes that implement the interface Process, a supplementary class called MessageUpdater implements also the Process interface and is used to make a self update operation in a peer on one of its nodes at a unique future time. The updating message consists of signaling a split operation when the number of entries is too high or a merge operation when the number of entries is too low. When an update message occurs, a peer event is created and scheduled by the simulator; such an event is of the class Event_Peer inherited from the class Event. This event is processed and executed by the same peer at given unique future time as a transaction (as explained in Section 3.6). In order to ensure the time uniqueness when processing the update operations, a static attribute denoted by Delay is used by the MessageUpdater class to maintain all the peer events distinct from one another in the event scheduler, in contrast to MessageSender and MessageReceiver instances that assume that a source peer is immediately received by the destination peer or by a given user with no delay.

Alg.7.2.4. shows the attributes and the methods used by the message updater to implement the interface Process.

```
public class MessageUpdater implements Process {


	// static attribute denoted by Delay is used to maintain all the update
	events distinct from one another in the event scheduler
	public static long Delay = 1;

	//describes the process identification of the message receiver in the
	simulation environment
	public long processId;

	//constructor
	public MessageUpdater(){
		processId = Sim.create_process(this);
	}


	//handle the update event if it is of type message Event_Peer
	public void process_event(Event e) {
		if(e instanceof Event_Peer) handle_Message((Event_Peer)e);
	}


	// call the NonLeafNodeSplit or the NonLeafNodeMerge on the appropriate
	Peer instance at the appropriate level.

	private void handle_Message(Event_Peer e){
		Peer concernedPeer =e.peer;
		int concernedLevel = e.level;
		//ref id -1 means it's an automatic update
```

```
        if((concernedLevel < concernedPeer.routingTable.size() &&
        concernedPeer.routingTable.get(concernedLevel).entries.size() <
        P2PSim.MIN_ELEM) || concernedPeer.flagMerge == true){
              concernedPeer.NonLeafNodeMerge(-1,concernedLevel);
        }

        elseif(concernedPeer.routingTable.get(concernedLevel).entries.siz
        e() > P2PSim.MAX_ELEM){
        //ref id -1 means it isan automatic update
        concernedPeer.NonLeafNodeSplit(-1,concernedLevel);
        }

}
```

Alg. 7.2.4. Attributes and methods used by the message updater to implement the interface Process

Moreover, each peer maintains a unique peer id, denoted by peer_id, which is used locally to balance its ranges when a range transfer occurs so that the probability of finding a peer j for a transfer of responsibility remains uniform (see Section 7.3). This attribute may also be used for debugging purposes. A User instance is also referred to by the Peer class in order to call back the corresponding user and deliver the results of an operation.

It may be noted that the same peer structure as described in the previous sections is used here, where each peer contains a multi-level routing table composed of a list of nodes, one at each level. The node at the lowest level of the routing table corresponds to the local range of the peer. This leaf node contains the local Data which holds the set of data entries and keys stored by users and described by tuples <key, data>. Each non-leaf node is composed of a list of entries of the form <range, pointer> where with each range is associated a pointer that points to the same peer or a different peer in the

next lower level. Each node maintains also a list of back-pointed peers to know which peers are pointing to it in the next upper level. Additionally, each Peer holds a variable called myHighLevel and is used to store the number of nodes i.e. the height of its routing table. Each node knows the Peer instance it belongs to and maintains an attribute denoted *level* to identify the level in the routing table it is associated with.

User Class and relationships

The user plays the role of an external entity to the system and can initiate an action or receive some feedback any time during the simulation process. The user communicates with the simulation environment by implementing two interfaces; the first is the Process interface and is responsible of signaling new queries when there is a new user initiative to be taken by the user. This initiative consists of executing a signaled event instance of the Event_User class. The action to be executed depends on an attribute in the User class called *userState* and corresponds to the user state. When userState is 1, the user is inserting a given data to the given key. userState 2 means that the user is searching for data that is associated with a given key. userState 3 indicates that the user is looking for data associated with a given range. In addition, userState 4, 5, 6 and 7 corresponds to local update operations that are performed on the distributed b+ tree structure such as splitting a given leaf node, splitting a non-leaf node, merging a leaf node or merging a non-leaf node, respectively. The software is basically designed in such a way that the peers are updated implicitly and only the peers can automatically initiate or signal a new update operation when needed. However, we implemented these user states from 4 to 7 for simulation purposes so that we can analyze and measure step by step the behavior of the b+ tree structure at any time when any update operations take place. Furthermore, a second user interface called ServiceCallBack interface is implemented by the User class and provides a call back service to the User instances. This ServiceCallBack interface is used by the corresponding peers to provide the results after performing a given task. It includes four methods, the SearchResult , the InsertResult, SplitResult and Mergedesult. These methods provide print-out, feedback gathering for statistics and recording information after the simulator finishes executing

certain tasks. They are useful for measurements purposes. Alg.7.2.5. shows the attributes and the methods used by the User class to implement the interface ServiceCallBack.

```
public interface ServiceCallBack {

	//returns the results of a search operation
	public void SearchResult(int refId , List<DataEntry> data,int status);

	//returns the results of an insert operation
	public void InsertResult(int refId , int status);

	//returns the results of a split operation
	public void SplitResult(int refId , int status);

	//returns the results of a merge operation
	public void MergeResult(int refId , int status);

	//returns the results of a delete operation
	public void DeleteResult(int refId , int status);


	//variable used to define the final state

	final int OK = 0;

}
```

Alg. 7.2.5. Attributes and methods used by the user to implement the interface ServiceCallBack

Description of the exchanged messages

The message class represents a message or a piece of information exchanged between two peers through the simulator. All the messages must contain as a parameter the source peer, the destination peer and a unique reference number of the message. In addition, each message uses some parameters that can be used by the other peers. A brief description of the query message types used in this simulation and their parameters is given below:

1.M_InsertKey (key, level, data)

Message navigating from one peer to another by going one level down until the destination peer is reached, and the given data and the corresponding key are inserted.

2.M_SearchKey (key, level)

Message navigating from one peer to another by going one level down until the destination peer is reached, and the data corresponding to the given key is searched

3.M_SearchRange (keymin, keymax, level, counter)

Message navigating from one peer to another by going one level down until it reaches its destination peer and searches for the data at its corresponding subrange bounded by keymin and keymax. The counter is used to count the number of keys in the sub-ranges to make sure that all the initial keys in the searched range are explored.

4.M_UpdateBack-pointers (level, splitkey, introducedPeer, side)

Message sent to the back-pointed peers at the next higher level to split its entry and one of these two entries will point to the hired peer.

5.M_MergeBack-pointers (level, mergekey, introducedPeer, side)

Message sent to the back-pointed peers at the next higher level to merge two of its entries to one single entry.

6.M_Split (myLocalRange, back-pointers, levelToCopy, entries, newData)

Message sent to a new introduced peer giving him responsibility of a part of its split local range at the lowest level.

7.M_LeafNodeMerge (myLocalRange, back-pointers, mergingData, mergingkey, side)

Message sent to another peer to merge its local range at the lowest level

8.M_CheckNonLeafNodeBack-pointers (level, nonCoveredRange)

Message sent to a given peer at a given level to check if its range can cover a part of a given range.

9. M_AddMyBack-pointers (back-pointers, level)

Message sent to a given peer at a given level to copy its back-pointers.

10. M_SuppressMeAsBack-pointer (level)

Message sent to a given peer at a given level to delete it from back-pointers.

11. M_AddMyNode (nodeToCopy, level)

Message sent to a given peer at a given level to copy its corresponding node.

12.M_CheckMergedBack-pointers (level, mergingRange)

Message sent to a given peer at a given level to check if a neighboring range can be merged up.

13. M_UpdateEntrySizeBack-pointers(level)

Message sent to a back-pointer at an upper level asking to check the size of the entries and update them accordingly.

In addition, to each query message corresponds a result message having the same name but ending with the word “Result”. For instance the result message for M_InsertKey is M_InsertKeyResult. Such message is sent back to the source and holds a binary parameter, 0 if the query was successful and 1 if the query encountered an error.

7.3 Implementation choices

Many actions could be taken to improve the distributed b+ tree system so that the system can reach its highest performance such as the appropriate maximum and minimum number of entries to be used by each node before making an update, Another major perfective optimization is to keep the workload equally distributed among the nodes before and after the update operations. In the subsections below, the range balancing criteria is described and the node/entry trade-off is then discussed.

7.3.1 Balancing the Ranges

Our first observation is that in the distributed b+ tree structure, we have to maintain a convention regarding the way the entries are transferred so that they remains equally distributed and balanced among peers. Indeed, when a non-leaf node splits or a non-leaf node merges, the number of entries of the split node is reduced and part of the range is suppressed from the corresponding node and transferred to another available peer j. If the decision of splitting the concerned range is randomly made, we may face a situation where the same common ranges are suppressed from many nodes and the probability of finding an available peer j becomes very low for some entries and very high for other ones. In that case, there is no peer that can be responsible for holding the transferred range, and the non-leaf node splits or merges become unfeasible. In order to maintain a reasonable balancing between the nodes of all the peers at a given level so that the probability of finding a given range in a given node is equally distributed among all nodes, a convention has to be made following some implementation choices. A simple way is to give a unique peer ID that is maintained by each peer in the distributed b+ tree. This static parameter is a non-decreasing integer that is incremented and allocated to each new peer introduced. This unique ID can be used when a non-leaf split is performed by choosing which side of the node has to be suppressed. For instance, a parity convention may be maintained so that the odd peers i.e. the peers whose unique ID is odd always perform a left non-leaf split so that only the left side

portion of the corresponding node is suppressed, while the peers having an even ID always perform a right non-leaf split so that only the right side portion of the corresponding nodes is suppressed. By using such an approach, we attempt to keep the split ranges suppressed fairly so that the probability of finding an existing peer that could perform the transfer of responsibility at the same level is uniformly distributed among all peers. This approach has some cost in terms of memory allocation since it requires a new parameter peer ID to be maintained by each peer. However, it guaranties a perfect symmetry of the distributed ranges among different nodes at the same level.

Another solution to range balancing has a lower cost and consists of delegating the responsibility of choosing which side range comprised in a given node to be transferred to the leaf nodes of the corresponding peer. Such operation may be performed locally by each peer and a convention can be made so that each non-leaf node communicates with the leaf node of the same peer: if the leaf node holds a local range comprised in the left side portion of the universe of key value U, the non-leaf node performs a left non-leaf split so that only the left side portion of the corresponding node is suppressed. In contrary, if the leaf node holds a local range comprised in the right side portion of the universe of key value, the non-leaf node performs a right non-leaf split so that only the right side portion of the corresponding node is suppressed. This method consists of a simple comparison with the universe of key values U that is already maintained by each peer and there is no need to add a new parameter here, in contrary to the previous approach that uses a unique peer ID. However, this approach does not guarantee a perfect symmetry of the distributed ranges among nodes at same level since the split and merge operations depends only on the data key insertions and deletions in some specific ranges and such action is beyond the control of peers and only performed by users. We may face a situation where some ranges in the left side portion of the universe of key values are over-loaded while the ranges in the right side portion of U remain under-loaded which results in more splits and mergers in the left side of U than in the right side. Therefore there would be more suppressed ranges from the left side than from the right side, and the symmetry cannot be maintained. For this reason, we have used the parity convention for our simulation.

7.3.2 Node/Entry trade-off

Typically, performing a split operation results in raising the number of entries maintained by each node and this will slightly increase the depth of the distributed b+ tree, and similarly the number of levels in each peer may decrease while performing a merge operation. The lower and upper bounds on the number entries per node are parameterized by the order of the b+ tree p which is maximum number of entries allowed per node. The minimum number of entries allowed per node is p/2.

The average fan-out, i.e. the mean of the number of entries per nodes relates the number of levels per peer to the number of peers in the system by the relationship:

$$num(peers) = mean(fanout)^{levels-1}$$

To ensure that the b+ tree is maintained optimally for the number of entries per node, a study of the characteristics and properties is conducted by simulation to determine which settings should be used to reach the highest performance of the distributed b+ tree with weak-consistency in terms of number of exchanged messages and execution delays.

7.3.3 Adapting the program to a real P2P system

The simulation program is designed in such a way that it is relatively easier to make some changes in order to obtain a real implementation of peer nodes that would execute the algorithms in an environment where each peer is communicating through the Internet with other peers running on different computers, or running in the same computer as a separate task. Therefore, the class “Peer” can be either used by the simulation environment by means of the discrete event simulator SSIM or within distributed real time applications.

8. Results of the simulation

In this chapter, we discuss the experimental results obtained from the distributed b+ tree simulation described in Chapter 7. In particular, the growth of the distributed b+ tree structure is first introduced and the Ping-Pong and the Ping-Only strategies are compared, the system parameters in the stationary state with insertion and deletion periods are illustrated, and a comparison of some particular system situations is also given. Notice that the maximum number of entries per node, i.e. the order of the b+ tree is fixed to p = 3 in this simulation. If we use a higher p for the system, the number of levels is reduced and may perhaps not exceed 4 levels for a network of 1000 peers. Therefore, we have chosen a maximum fan-out of p = 3 which allows us to investigate with clarity the distributed b+ tree properties for a network of 1000 peers and a number of levels per peer higher than 4. However, b+ trees usually have much bigger fanouts in practical systems. This parameter may be changed for the study of networks with a higher number of peers i.e. for b+ tree systems holding millions or billions of nodes.

8.1 Growth of the distributed decentralized b+ tree

We grow the distributed b+ tree network by performing a certain number of key insertions until we reach a network having 1000 peers, and we observe instantly the behavior of the increase in both depth of the b+ tree and the number of back-pointers per non-empty node. Notice that only the Ping-Only algorithm is used in the following subsections as it has been shown that the Ping-Only strategy is much more efficient than the Ping-Pong strategy (see Section 6.1 for details).

8.1.1 The depth of the b+ tree

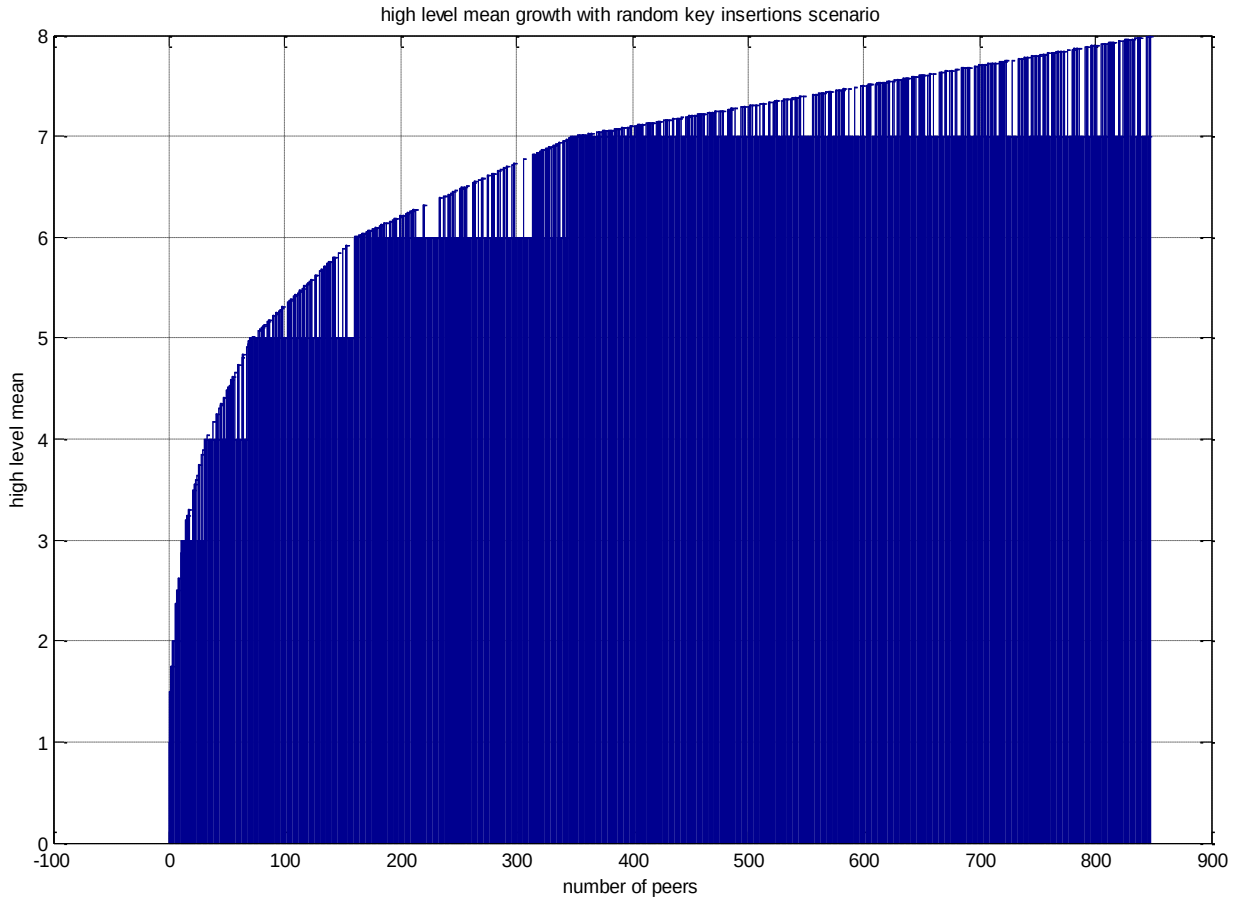


Fig.8.1.1. Growth of the distributed decentralized b+ tree structure

We observe from the above diagram that the mean of the highest level (averaged over all peers in the networks) constantly increases as we increase the number of peers. The system tries to maintain the same number of levels in each peer when the network is growing. For instance, the highest level mean is 6 when the b+ tree structure comprises 200 peers while the highest level mean is 7 when the b+ tree structure comprises 400 peers. In fact, performing leaf node split operations by key insertions results in an

increase of the number of entries maintained by each node when updating the back-pointed peers at the next higher level that have an entry pointing to the split leaf nodes. This leads to non-leaf split operations among these non-leaf nodes and will slightly increase the depth of the distributed b+ tree by increasing the number of levels in those peers. The system hence tries reasonably to maintain the same number of levels in order to keep the b+ tree structure balanced. This result comes from the assumption that the number of levels is only dependant of the number of entries per node as well as the number of peers in the system, as discussed in Section 7.3.

8.1.2 Mean number of back-pointers

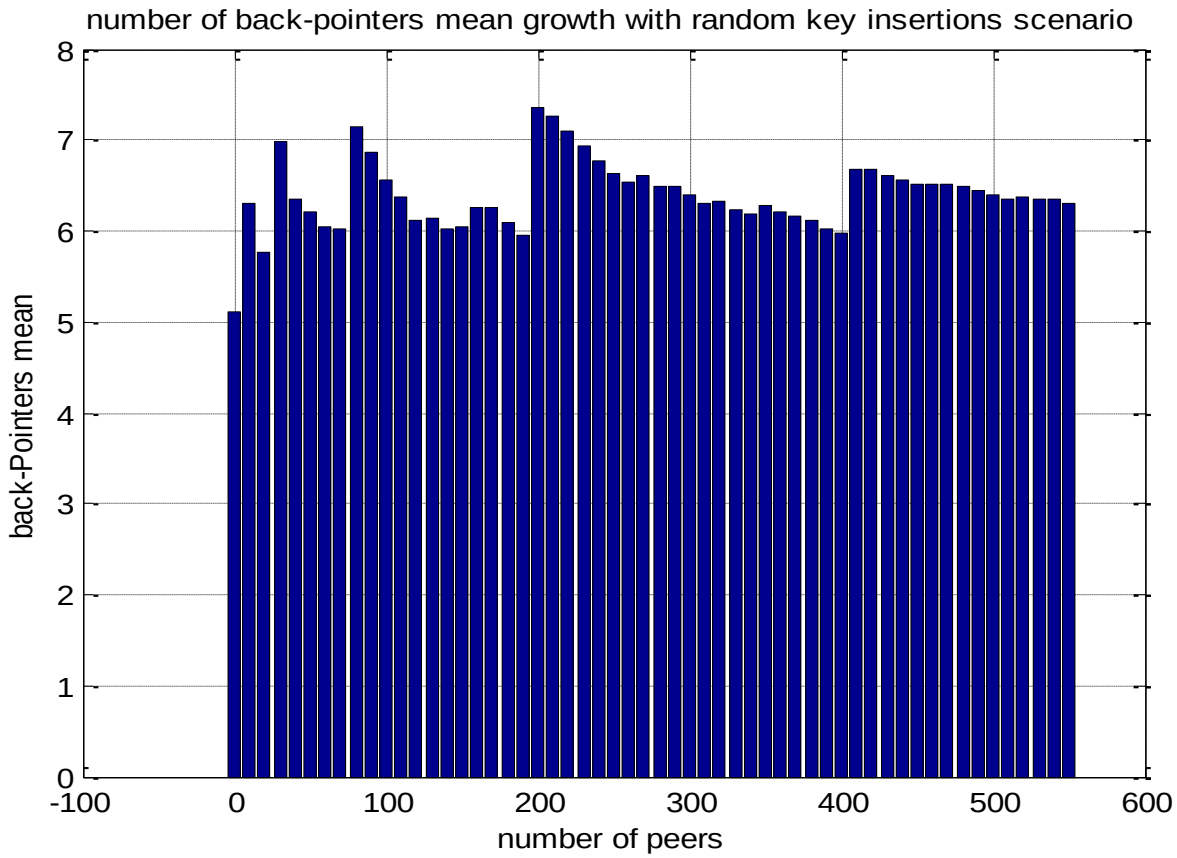


Fig.8.1.2. Effect of growing the distributed decentralized b+ tree on the mean number of back-pointers

When the b+ tree is growing by performing key insertions, leaf-node and non-leaf node updates occur which may slightly increase the number of back-pointers in the system. In fact, when a split operation is performed, a part of the range of the split node is transferred to another available node and all the back-pointers pointing to the split node will have an additional entry that is pointing to the new node at the next higher level. However, since the back-pointer table is maintained by the nodes and not by the entries, the back-pointers of the split node may be copied to the back-pointers of the new available node. But these back-pointers may keep pointing to the split node if they comprise in their entries a range that is described in the kept part of the split range. Hence, this will slightly raise the number of back-pointers in the system. Additionally, when the split occurs in the highest level node of a given peer, a new level is added to its routing table and this new node describes the universe of key values and is composed of two entries: one entry comprise a self-range that describes the kept part of the split range in the node of the same peer at the next lower level, while the second entry comprises a range pointing to the available node that became responsible of the transferred part of the range at the next lower level. This procedure will also slightly increase the number of back-pointers in the system. However, we can see from the above diagram that the mean number of back-pointers obtained after a long period of random insertions tends to be constant as we grow the b+ tree structure, which is considered a good sign of stability.

8.2 Comparing the exploration mechanisms

The experimental results of the Ping-Pong strategy showed that it is not always possible to find an existing peer j whose routing table, at the same level RT_j^l, is empty or already contains some entries covering some common range withRT_i^l. In contrary, the Ping-Only strategy always finds such a peer. The major reason for the Ping-Pong strategy drawback is that most of the explored tree nodes are only used to traverse the b+ tree. It consists of performing the Ping operations first by traversing the next-lower level nodes, but then returns the query to higher level nodes by using the back-pointers of the traversed nodes until it reaches some nodes at the same level as the initiating node. Therefore, the algorithm can only reach non-empty nodes by traversing the back-pointer nodes, and all the empty nodes, which represent a good prospect for a range transfer, are ignored. In order to overcome this weakness, we somewhat modified the Ping-Pong version by allowing it to behave as a Ping-Only algorithm for only one level of traversal. This procedure is expected to reach empty nodes with a reasonable probability by checking the ranges of the peers associated with their pointers at the same level and thus making the comparison of the two algorithms possible. We compare both the modified Ping-Pong strategy and the Ping-Only strategy by means of message complexity, the order of traversal as well as the level distribution of the empty-nodes and non-empty nodes found for a network of 1000 peers where each peer comprises a routing table composed of 9 levels. The experimental results obtained for both Ping-Pong and Ping-Only algorithms, while the b+ tree structure is grown up to 1000 peers, are illustrated in the following sub-sections. Notice that all the records are obtained during the system buid-up.

8.2.1 Comparing the message complexities

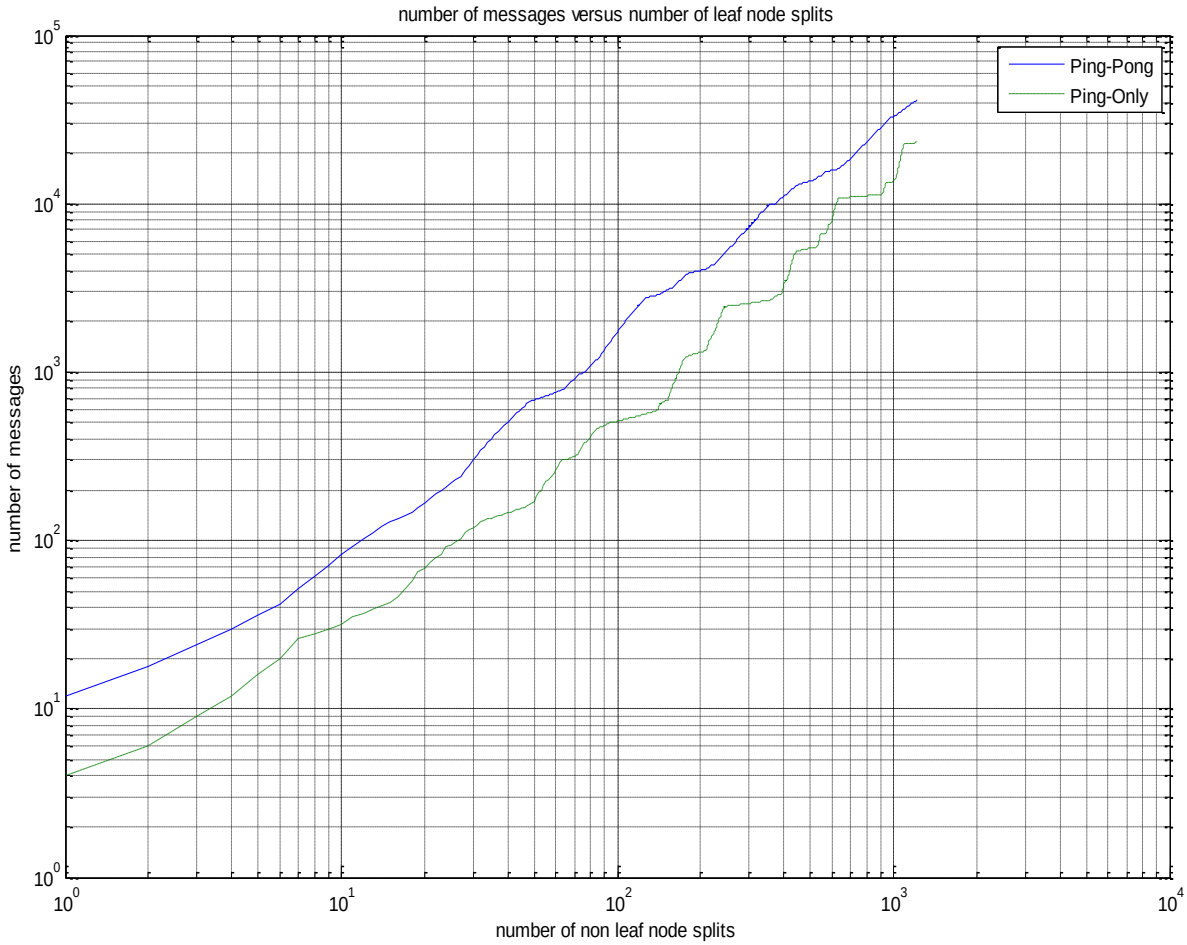


Fig 8.2.1. Comparing the message complexities when the b+ tree is growing

The figure above shows the increase in the number of messages for the Ping-Pong strategy (strong line) and the Ping-Only strategy (dashed line) with respect to the number of non-leaf node splits performed by inserting data elements with randomly chosen key values.

We can clearly see that the number of messages in the Ping-Pong mechanism is much higher than the number of exchanged messages in the Ping-Only mechanism. This is due to the fact that in the Ping-Pong transfer of responsibility, the nodes traversed by the query at the lower levels have only a role of routing and they do not self-check if they have empty-nodes or if they share some common ranges at the same level as the initial peer. Instead, it makes use of the back-pointer tables to reach the nodes at the same level as the initiating node. Since each node may maintain a back-pointer table having more than one back-pointed peer, the number of traversed peers increases as we go backwards to reach the peers that are at the same level as the initiating peer, which exponentially increases the number of exchanged messages.

In contrary, the Ping-Only operation consists of only pointing down to the peers responsible for the transferring ranges at the next lower level, and for each traversed node, the node belonging to the same peer at the same level as the initiating node is checked instantly, instead of pointing backwards to the peers responsible for nodes containing these entries in the next higher level nodes. The number of exchanged messages is thus significantly reduced, which makes the Ping-Only algorithm much better to use than the Ping-Pong algorithm in terms of message complexity.

8.2.2 Comparing the order of traversals

The order of traversal in the Ping-Pong strategy (left diagram) describes the number of traversed levels to reach a break-point node i.e. the pausing node after the execution of Ping operations and just before the execution of the Pong operations. In contrast, the order of traversal in the Ping-Only strategy (right diagram) describes only the number of traversed lower levels, i.e. only the performed Ping operations that are required to reach an available node. The figure below illustrates the histogram of the order of traversal with both Ping-Pong and Ping-Only algorithms.

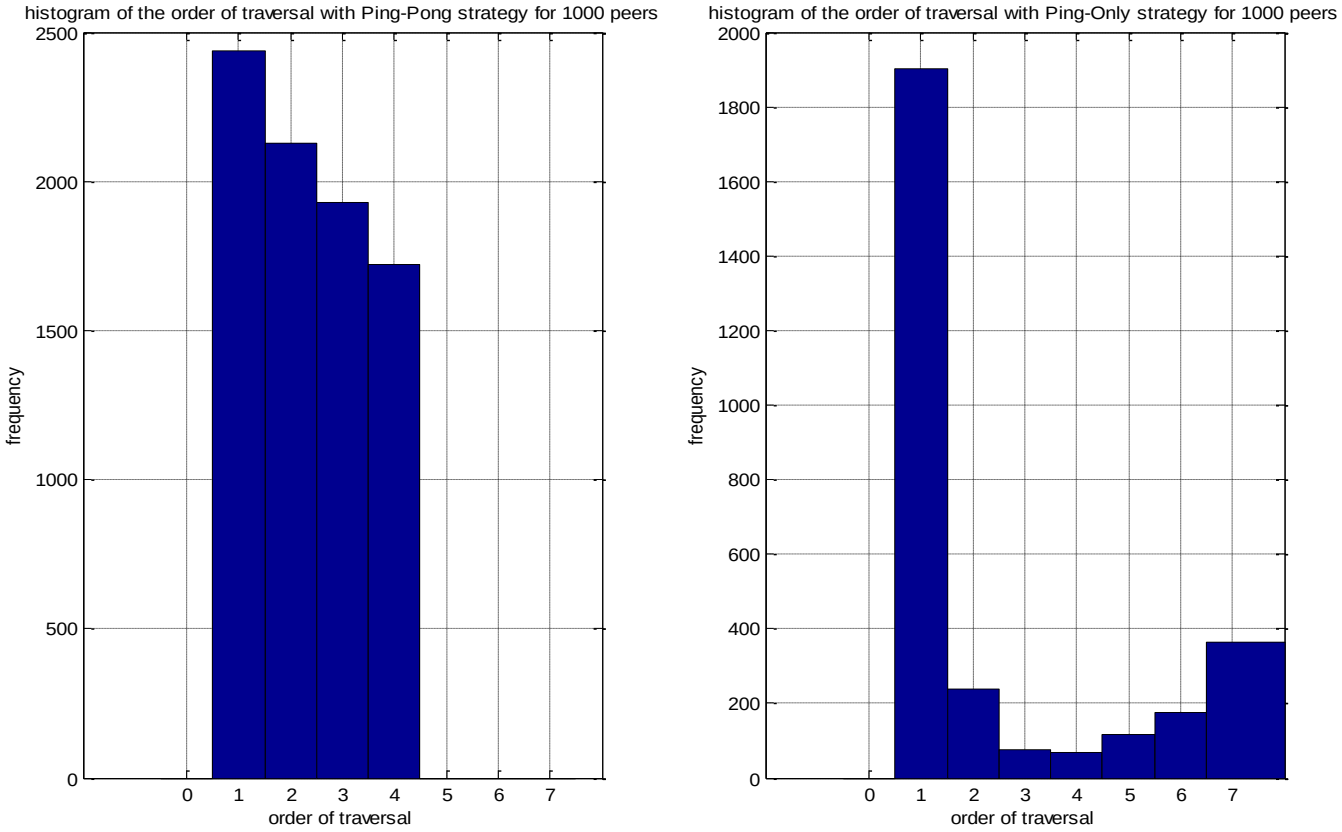


Fig 8.2.2. Comparing the order for traversal by Ping-Pong algorithm and Ping-Only algorithm

The distribution of the order of traversal in the Ping-Pong strategy consists of a decreasing Stair-step graph where most of the available nodes are found in proximity to the initiating node. This shape is due to the recursive behavior of the Ping-Pong strategy which consists of repetitively going-down a certain number of levels then repetitively going backwards the same number of levels until an available node is found. In each Pong operation in the Ping-Pong strategy, all the back-pointers of the traversed nodes receive a Pong query and this operation is repeated in a recursive way until an available peer is found. The above histogram of the Ping-Pong algorithm only shows four orders of traversal since the algorithm was implemented in a restricted manner allowing only up to 4 orders. However these implemented orders are sufficient to expect the stair-step shape of the order of traversal distribution.

In contrary, Ping-Only strategy starts from the highest node and is always oriented towards the next lower level nodes which results in an irregular traversal of the distributed b+ tree where the traversed nodes only check if the peer they belong to comprises a suitable node at the same level. If not, it redirects the query to all the pointers associated with its ranges, until an available peer is found. We also observe from the above histogram of the order of traversal with the Ping-Only algorithm that the frequency of reaching a leaf node by traversing the distributed b+ tree structure from a highest level node to the lowest level node is high.

8.2.3 Comparing the levels of the suitable nodes found

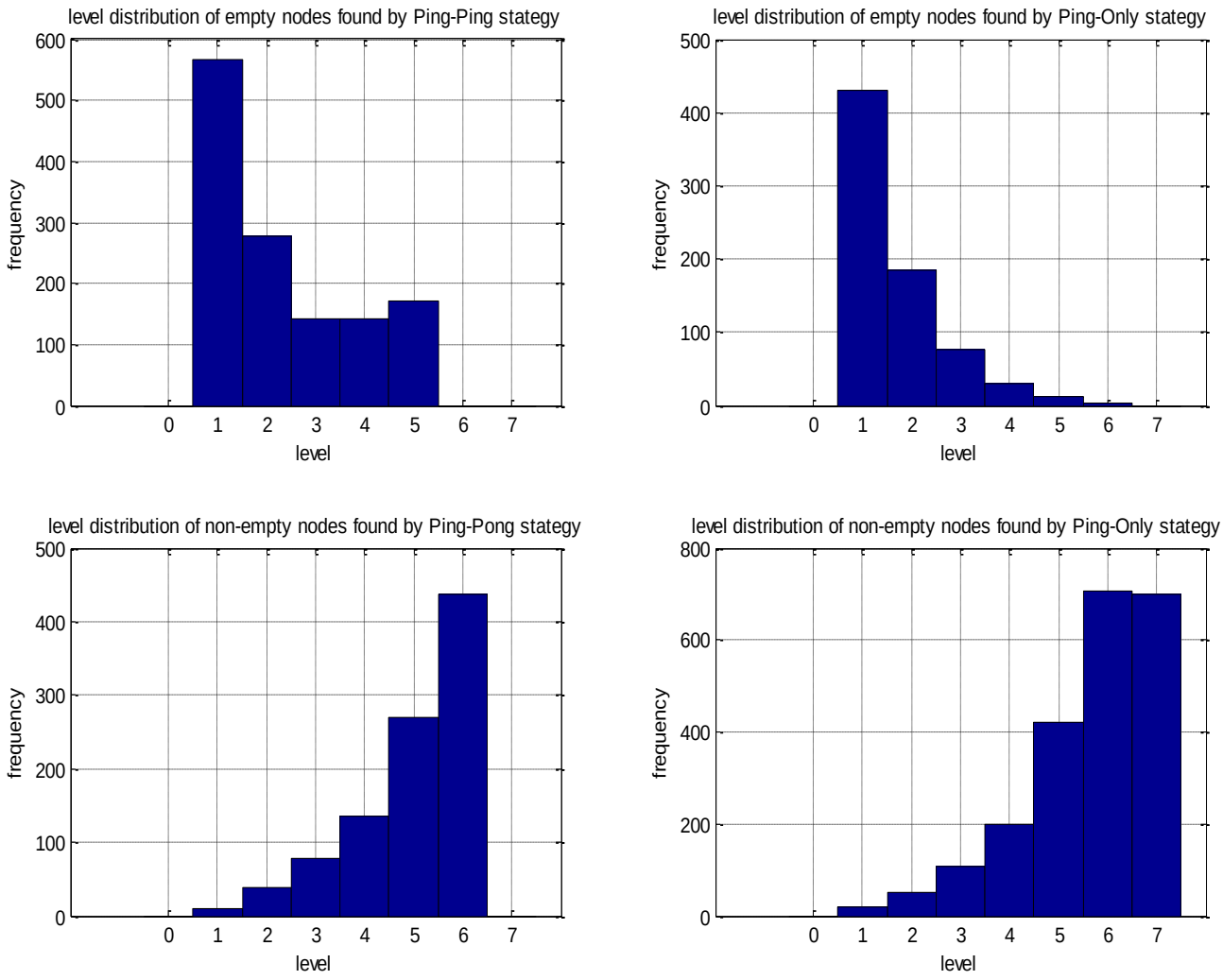


Fig 8.2.3. Comparing the level distribution of non-empty nodes found by Ping-Pong algorithm and Ping-Only algorithm

We observe from the above level distribution diagrams that the frequency of finding an empty node in both Ping-Pong and Ping-Only algorithms increases with the increase of the level index in which the empty node is found and the frequency of finding empty-nodes decreases as we reach the highest level nodes. In fact, when we perform split leaf node operations, a new peer is introduced and only the leaf node and the highest level node of the new peer allocate new entries that are copied from the split node while all the internal nodes, i.e. the nodes between them, remain empty. Since the highest level node of the new peer contains the universe key value *U*, there is a low probability of finding a node that contains this global range at the same level. Therefore, there is more chance to find empty nodes in the lower levels than in the higher level nodes when a transfer of responsibility operation is performed using both Ping-Pong or Ping-Only strategies. This frequency of finding empty nodes increases as we reach the highest level nodes. The same reasoning as in the level distribution of empty-nodes found is also applied here.

In contrary, the frequency of finding a non-empty node decreases with the increase of the level index in which the empty node is found in both Ping-Pong and Ping-Only algorithms.

8.3 Stationary system state with insertion and deletion periods

In this section, we intend to reach the steady state of the distributed b+ tree structure by performing several key insertion and deletion periods after the system is initially grown to 1000 peers, An insertion period consists of insertions of random keys until a total number of 1500 peers is reached. This is followed by a key deletion period which consists of randomly selected key deletions until the number of peers in the system reaches again 1000 peers. After several such key insertion and deletion periods, we expect the system to enter a steady state, meaning that the probability distribution of the number of empty nodes, of back-pointers and of entries in non-empty nodes do not change when another sequence of insertion and deletion periods is executed.

In this simulation, we perform three successive insertion and deletion periods starting from a network with 1000 peers where each peer maintain a routing table composed of 9 levels from 0 to 8.The following results are collected after each pair of insertion and deletion periods. We assume that if the resuklts of subsequent periods have have similar shape, then we can assume that the system has reached its steady state.

8.3.1 Distribution of empty nodes

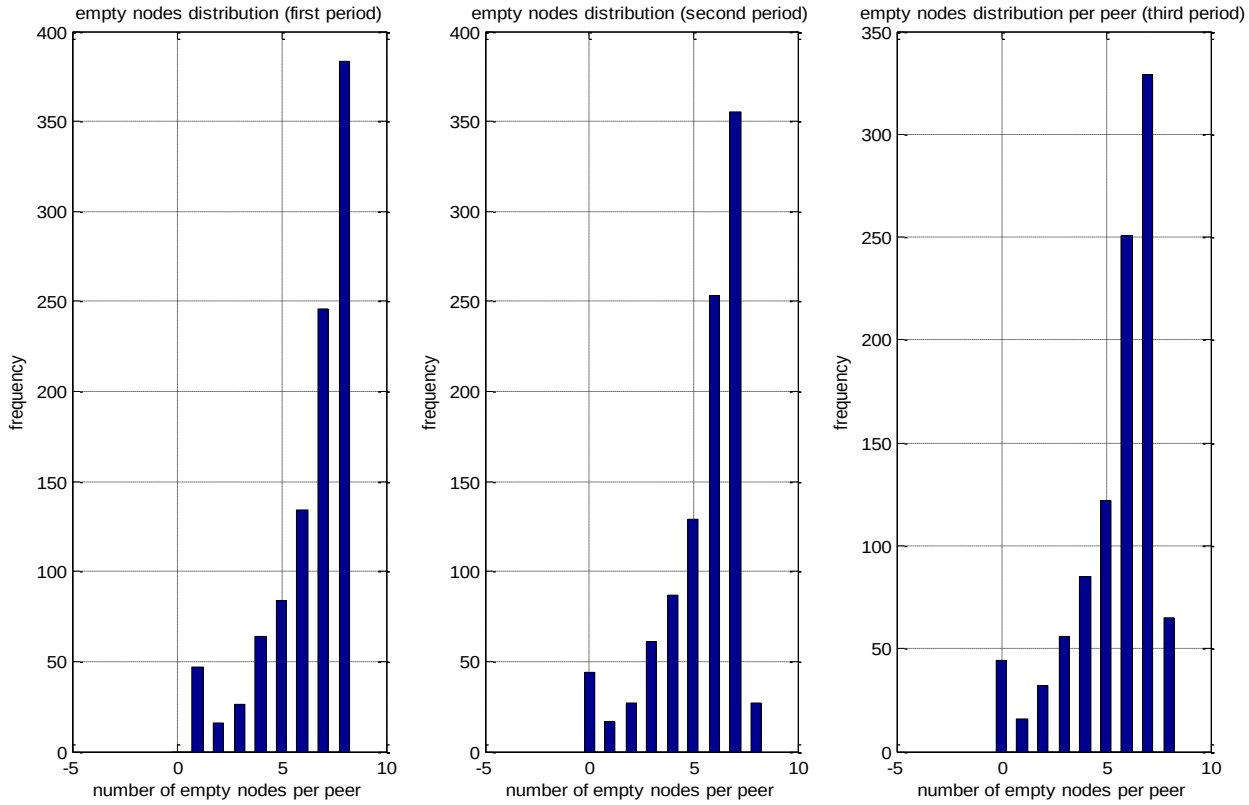


Fig.8.3.1 Distribution of the number of empty nodes per peer in the stationary system state

We can see from the above diagram that the empty nodes distribution in each peer is relatively similar for the three insertion and deletion periods. This observation allows us to assume that the steady state is reached for the three consecutive insertion and deletion periods operations starting from a network of 1000 peers.

We also observe that the highest frequency of the number of empty nodes per peer is observed in the peers having 7 empty nodes when the highest level in each node is maintained at 8. We may believe that this edge is due to the leaf node split updates that consists of transferring a part of the local range of the merging leaf node to the local range of a new introduced peer and copying the highest level node to the node at the same level of the latter peer. Thus, only two nodes are occupied while the new introduced peer is maintaining a routing table of 9 levels from 0 to 8. This leads to 7 empty nodes in this introduced peer. If such operation is performed a reasonable number of times, all the new introduced peers involved will maintain a constant number of 7 empty nodes. We may assume that the highest frequency of the empty nodes per peer occurs when the system reaches the steady state with a number of 7 empty nodes per peer. This observation is thus consistent with our expectations.

8.3.2 Distribution of the number of back-pointers

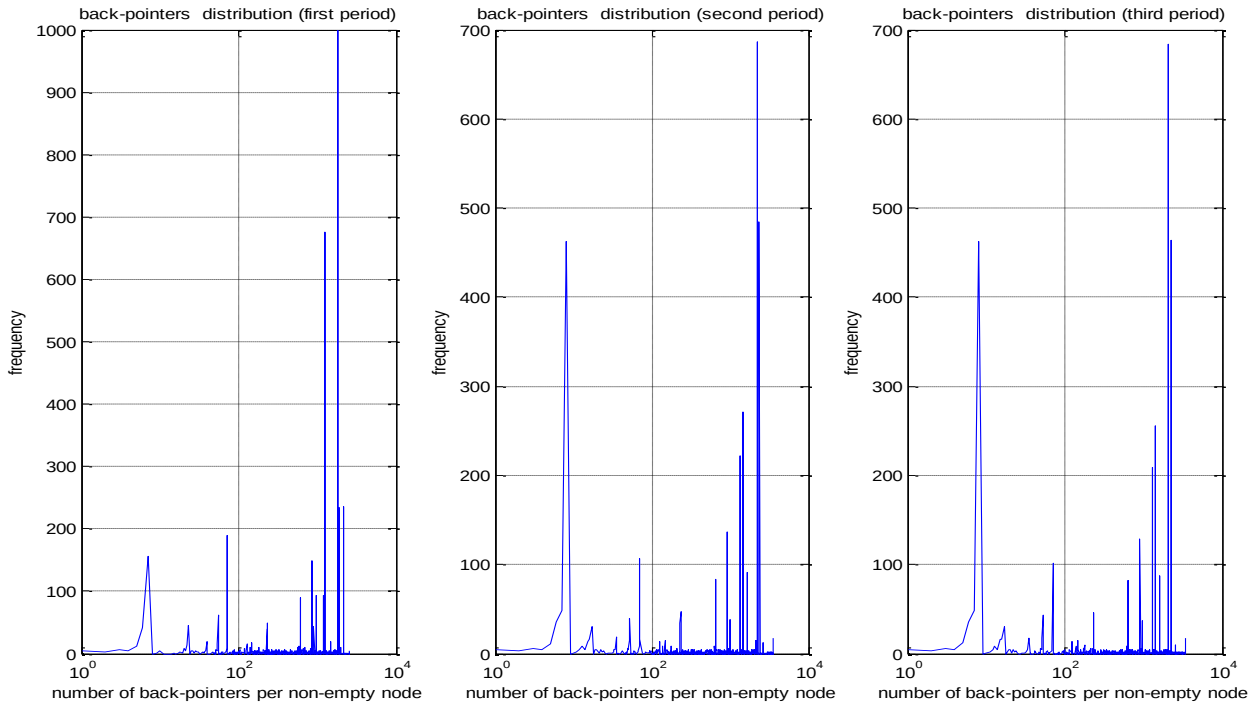


Fig.8.3.2 Distribution of the number of back-pointers per non-empty node in the stationary system state

We can see from the above diagram that the distribution of the number of back-pointers per non-empty node is relatively similar for the three insertion and deletion periods. This observation allows us to assume that the steady state is reached for the three consecutive insertion and deletion periods operations starting from a network of 1000 peers.

We also observe that the frequency of the number of back-pointers per non-empty node is noticeably high with nodes having approximately 300 and 400 back-pointers in each non-empty node. This means that some nodes maintain a high number of back-pointers and there are only few paths that are used by many peers to reach the range comprised in the entries of such nodes at a given level. We were not able to identify the reasons for this high number of back-pointers and this subject remains an open problem for future studies.

8.3.3 Distribution of the order of traversals

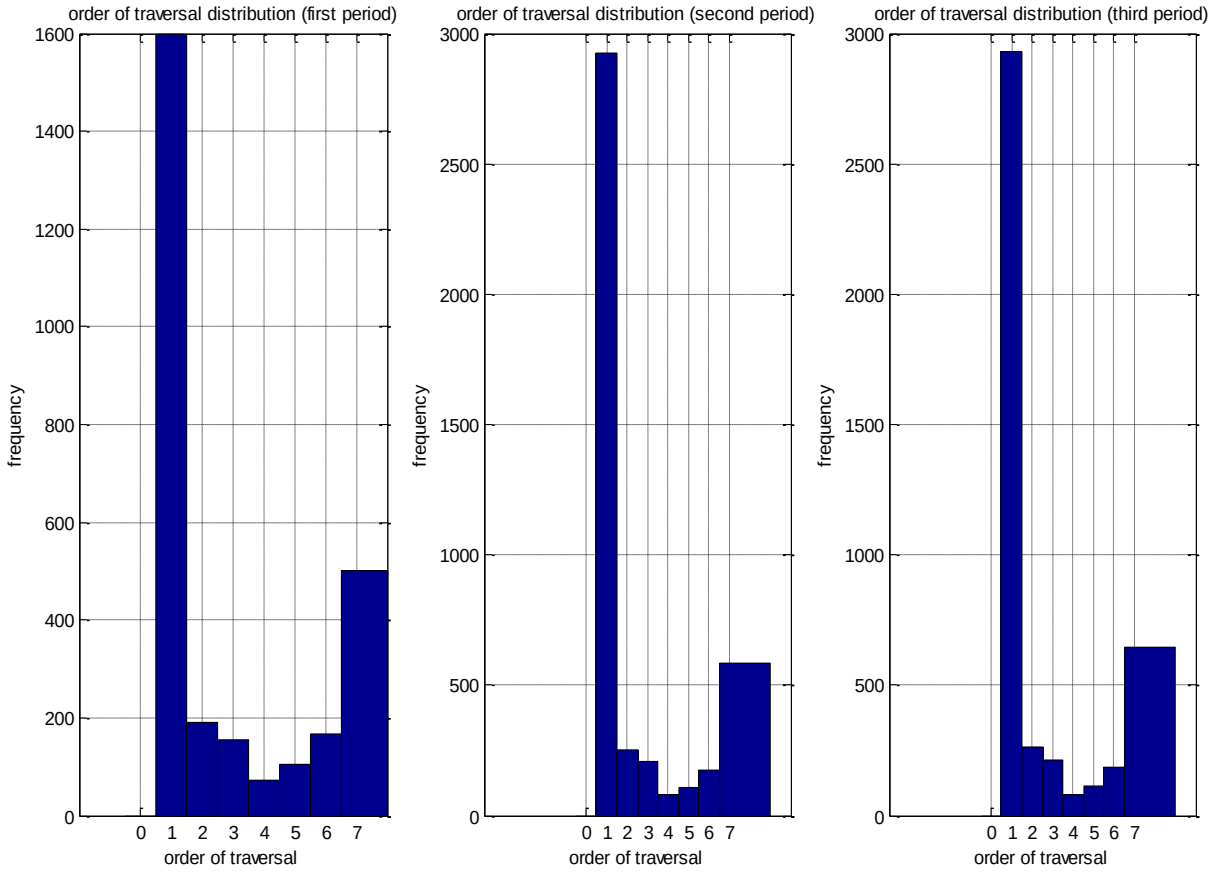


Fig.8.3.3 Distribution of the order of traversal in the stationary system state

The above diagram illustrates the order of traversals, i.e. the number of traversed lower levels performed by the Ping-Only operations. We observe from this diagram that the order of traversal distribution is similar in the second and third insertion and deletion periods in comparison to the order of traversal distribution in the first insertion and deletion period. Notice that only the Ping-Only strategy for the transfer of responsibility mechanism is considered here.

We also observe from the above histogram that the frequency of reaching a leaf node by traversing the distributed b+ tree structure from a highest level node to the lowest level node is high, which is of order of traversal 7 in the above right diagram. This means that there is a reasonable probability that the worst case occurs in the Ping-Only algorithm with a complexity of $L = \log_p(N)$, where N is the number of peers in the system and p is the maximum number of entries in each node. However, the highest order of frequency is observed with only one order of traversal which represents a benefit in terms of the number of exchanged messages to perform the transfer of responsibility mechanism.

8.3.4 Distribution of the number of entries in nodes

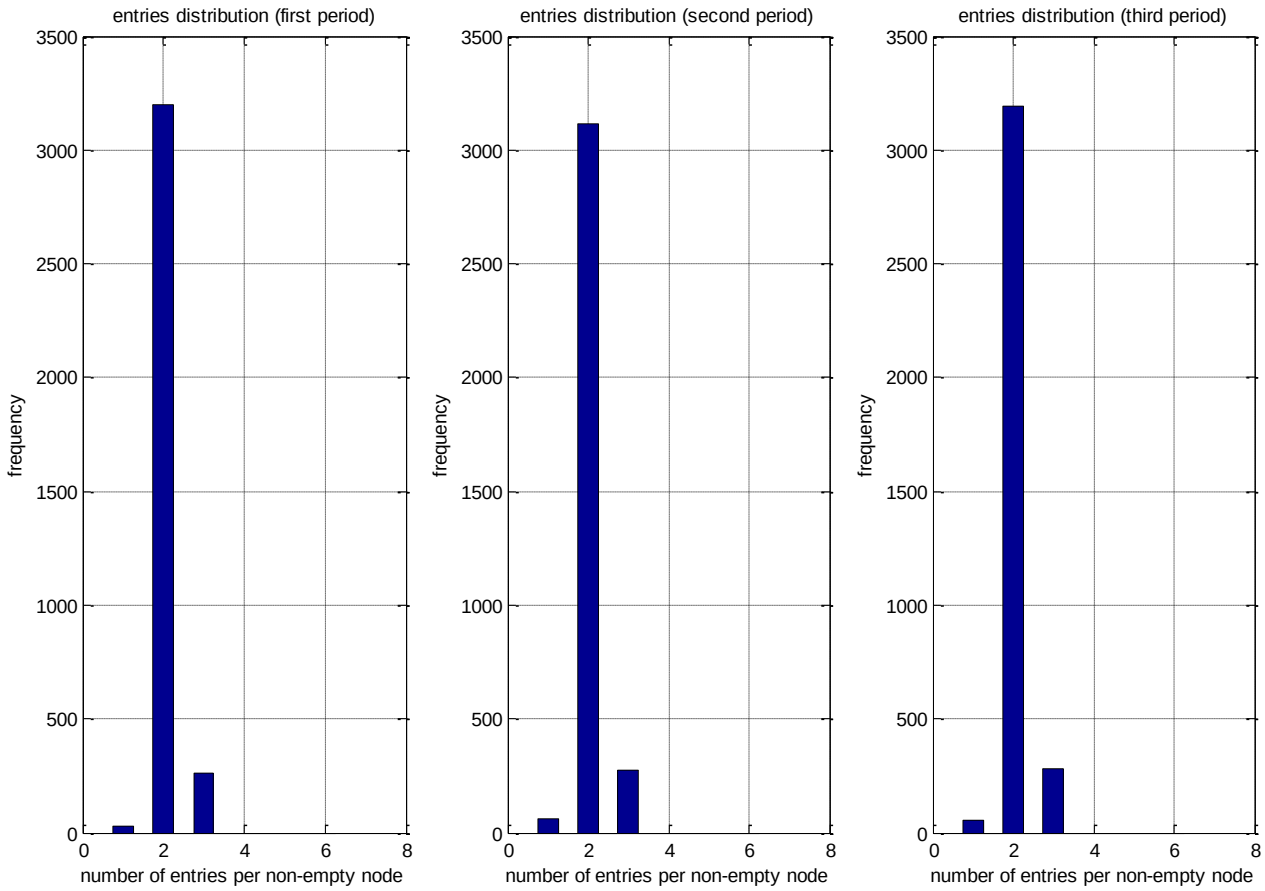


Fig.8.3.4 Distribution of entries per non-empty node in the stationary system state

Again, we observe from the above diagram that the number of entries per non-empty node distribution is relatively similar in all insertion and deletion periods phase. From this illustration, we may think that the steady state is reached for all the three consecutive insertion and deletion periods operations starting from a network of 1000 peers.

We also see from the above diagram that most of the non-empty nodes have 2 entries which are lower than the maximum number of allowed entries p (p = 3 in the system configuration). This means that the ranges remain reasonably balanced among b+ tree-nodes and the structure if the b+ tree reaches its highest level of performance when the mean of the b+ tree fan-out is less than p. This experimental result is thus consistent with our expectations.

Finally, we observe that some nodes have only one entry. Performing several key deletions results in a leaf-node merger that consists of merging the local range of the merged leaf node with another local range in the neighbouring peers by checking the ranges of the back-pointed peers so that the assumption of consecutiveness is maintained. The back-pointed peers, pointing to the merger at the next higher level, may become under-loaded after merging two of its entries and have to merge them as necessary. This procedure is cascaded to the higher levels and may reach the highest level node. In the case when a back-pointed peer at the highest level has only two entries, his one single resulting entry may be associated with a pointer to another peer and thus cannot be removed if there is no internal node from its peer that contains the universe of key values in its entries, so that the search operation remain correct. This is why we may obtain a few nodes that contain only one entry.

8.4 Uneven key insertions

Starting from the steady state with 1000 peers, we perform around 250 key insertions in a specific range until we reach 1200 peers. In this simulation, the specific range is defined by consecutive key values starting from the middle of the universe of key values U. If the corresponding inserting key is not empty, we move to the next available incremental key value until an empty key is found. .Once we reach 1200 peers, we compare the b+ tree structure with the one obtained after performing random key insertions. Our simulation program checks during an insertion operation whether the requested key exists already. If this is the case, the key is shifted to the right a until an available key is found. Therefore this uneven key insertion will involve a sequence of key values in the middle of the key universe. The results of these simulations are shown in the following subsections.

8.4.1 Distribution of empty nodes

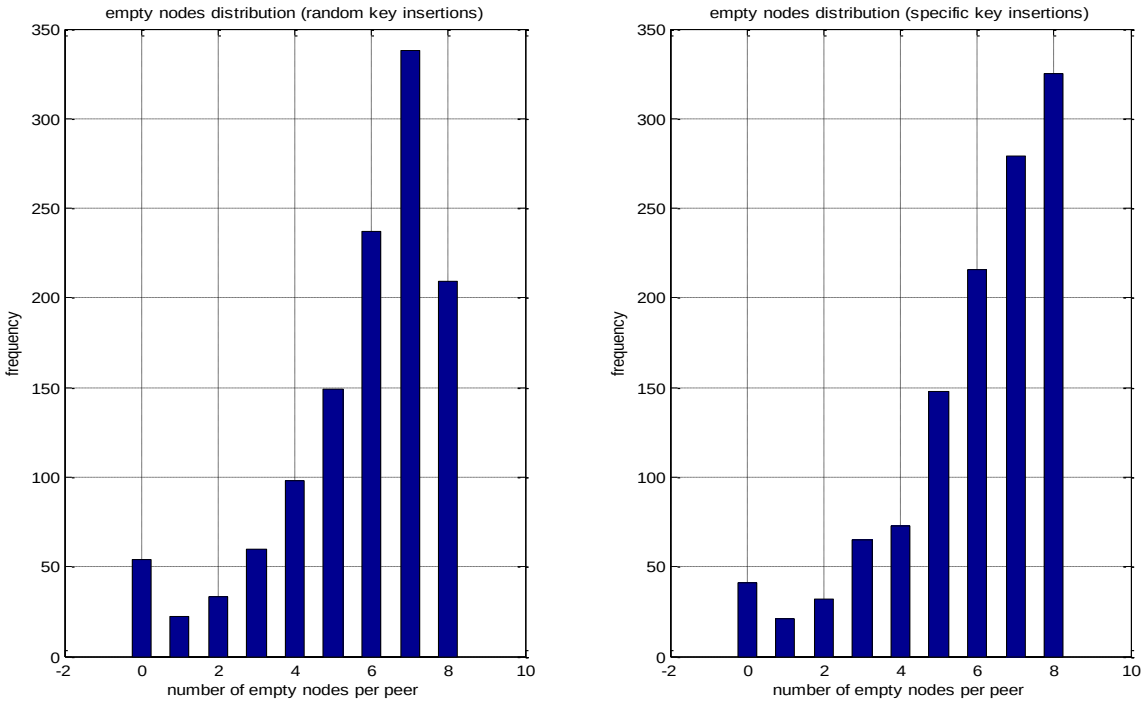


Fig.8.4.1 Distribution of the number of empty nodes per peer for random insertions scenario and specific key insertions scenario

We observe in the above diagram an important decrease of the number of peers having 7 empty nodes. The reason for this decrease is that when performing several key insertions in the specific range, the leaf node holding this key is split as its local range becomes over-loaded and the corresponding leaf node will then update its back-pointers. Thus, all the corresponding back-pointers at level 1 of the latter leaf node will split their entries and may become over-loaded, which will invoke new non-leaf node split by transferring a part of their ranges to an available node at level 1. Therefore, many empty nodes may be chosen to become responsible for the transferred ranges and the procedure of occupying empty nodes at level 1 will increase as we keep performing the insertion operations in the same specific range.

This will result in an increase of the number of non-empty nodes and decreases the number of empty nodes, which explains the decrease of number of nodes having 7 empty nodes to approximately 275 while the number of nodes having 7 empty nodes in the random key insertions scenario is maintained at 340. Thus, this observation is consistent with our theoretical expectations.

8.4.2 Distribution of the number of back-pointers

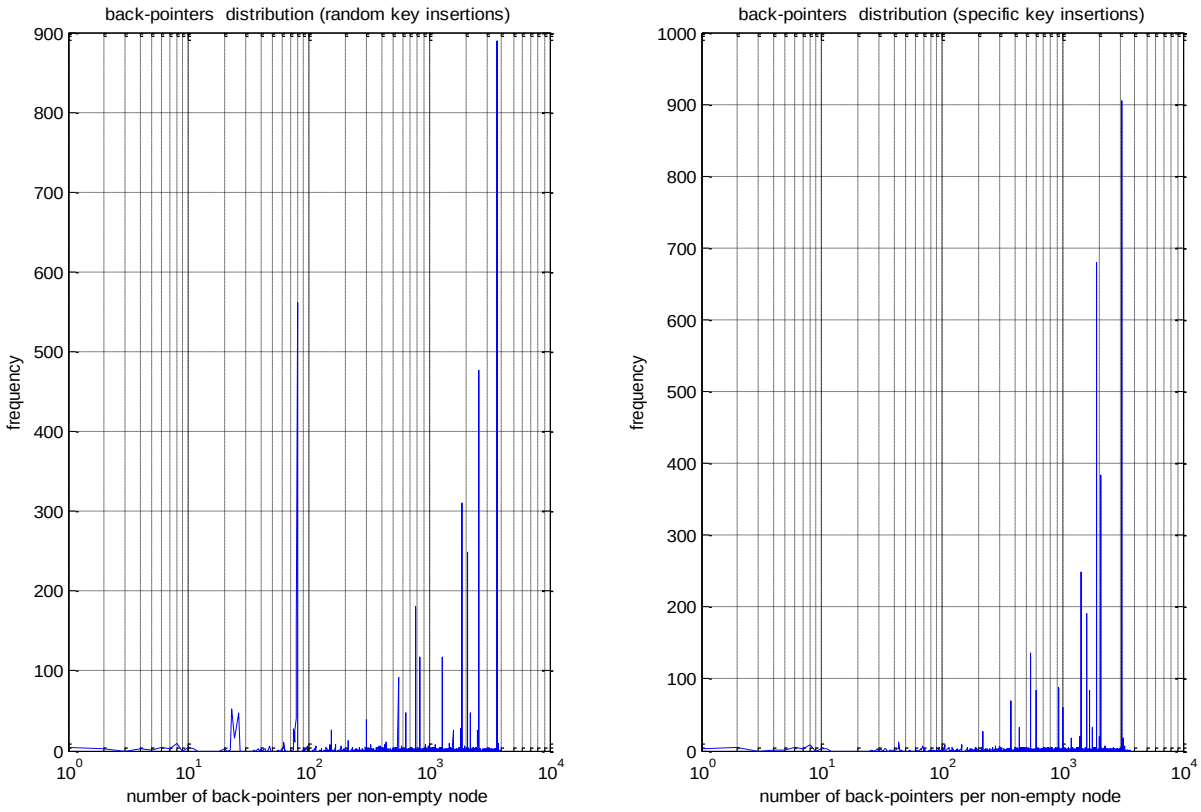


Fig.8.4.2 Distribution of the number of back-pointers per non-empty node for random insertions scenario and specific key insertions scenario

We observe in the above diagram an increase of the number of back-pointers per non-empty node when we perform specific key insertions as compared with the random key insertions scenario. For instance, the number of nodes having approximately 1000 back-pointers increases from 300 in the case of the random key insertions scenario to 700 in the case of specific key insertions. The reason for this increase is similar to the increase of the number of empty nodes discussed in Section 8.4.1.

Performing multiple leaf node splits will over-load the entries of the nodes back-pointed by the split node which results an increase in the number of non-leaf node splits at level 1 as we perform more specific key insertions. The back-pointer table of the split non-leaf nodes is thus copied to the available nodes at the same level. Performing more specific key insertions will therefore significantly increase the number of back-pointers used by the nodes at level 1. We conclude that this observation is consistent with our theoretical expectations. Notice that the spike around 10 back-pointers is not visible in this figure in contrast to Fig. 8.3.2. In fact, Fig. 8.3.2 is obtained when we have a steady state with a b+ tree having 1000 peers. Then, we perform random insertion and specific insertion operations until we reach a b+ tree with 1200 peers (Fig. 8.4.2). Therefore, we do not have the same number of peers. It is perhaps the reason why one does not see the spike around 10 back-pointers.

8.4.3 Distribution of the order of traversals

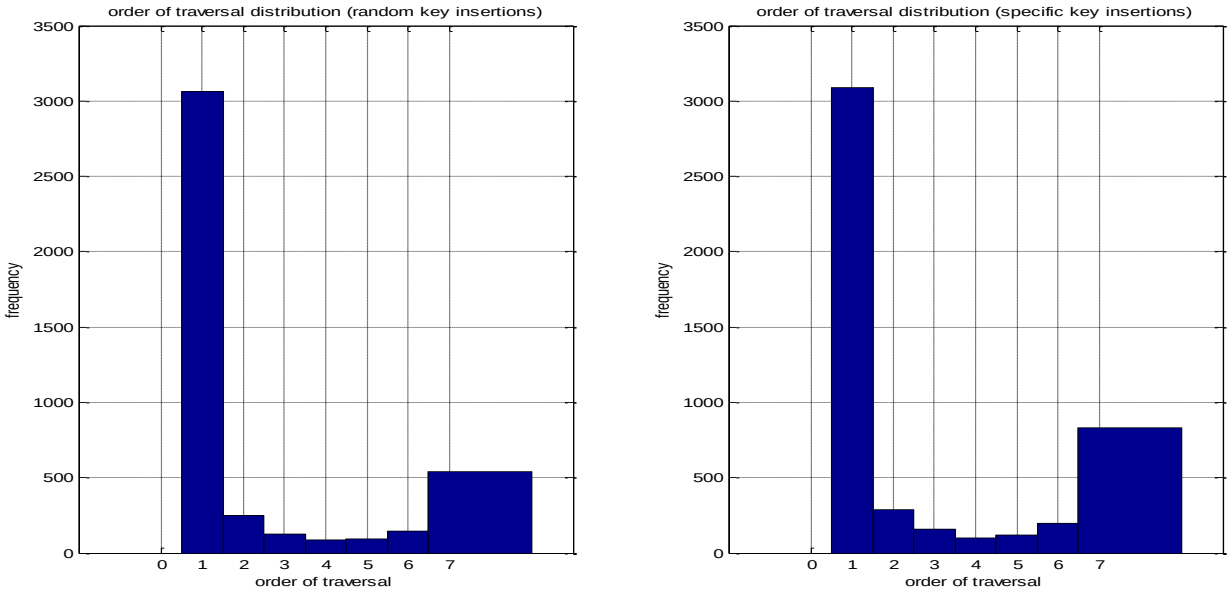


Fig.8.4.3 Distribution of the order of traversal for random insertions scenario and specific key insertions scenario

We observe in the above diagram that the order of traversal performed by the Ping-Only operation is insignificantly affected by the specific key insertions scenario. The reason is that the Ping-Only query is initiated from the highest level node of the initiating peer independently of the level in which the non-leaf node occurs. Since the specific key insertion operations only affects the non-leaf nodes at level 1 when each leaf node split involves an update in its corresponding back-pointers at the next higher level, all the higher level nodes are not affected by such a scenario and the order of traversal distribution with the Ping-Only strategy remains unchanged.

8.5 Conclusion

The simulation results showed that the Ping-Only algorithm can always find a suitable peer that is empty or has some common ranges with the initiating peer at the same level. Also, our comparative results of Section 8.2 showed that the Ping-Only algorithm is much more efficient than the Ping-Pong algorithm in terms of message complexity and order of traversal, with a message complexity of $\log_p(N)$, where N is the number of peers and p is the maximum number of entries allowed per node in the b+ tree. The major drawback of the Ping-Pong algorithm is that it does not check if the explored peers are empty or if they share some common ranges at the same level with the initiating peer. However, only the end nodes at the same level are inspected. In addition, another disadvantage of the Ping-Pong algorithm is that only back-pointers are explored and it is not possible to reach empty nodes at the same level as the initiating peer, while the search for an empty node in the b+ tree is crucial for the split and merges update operations to be performed properly under the weak-consistency invariants.

Our simulation results showed also that the mean number of back-pointers maintained by each peer tends to be constant as we grow the b+ tree. This observation, if true, is considered advantageous if the size of the back-pointer tables can be maintained to a certain limit. In contrary, the number of back-pointers per node is high for some non-empty nodes. This means that there are only few paths that are used by a huge number of peers to reach certain target ranges. This concern remains a open problem for further study.

The simulation results showed also that the number of entries in most non-empty nodes of the distributed b+ tree is less than the order of the tree p, where p is the maximum number of entries allowed per node, which lets us presume that most of the loads are maintained balanced and the distributed b+ tree has an average fan-out less than p.

9. Conclusion of the thesis

This thesis is about studying the properties of a distributed tree data-structure that allows searches, insertions and deletions of data items. The study consisted of distributing a b+ tree structure among several peers in a decentralized approach by assigning the responsibility of one branch of the tree, i.e. the path from root to a leaf node, to one peer instead of assigning only one tree-node to one peer. Such approach assumed weak-consistency conditions to be always true among its nodes but provides strong-consistency in terms of search semantics. This decentralization involves three weak-consistency invariants that are found to maintain the search operation correct, when each update transaction of the data structure maintains these invariants

9. 1 Contributions of the thesis

The contributions to this thesis are the following:

(1) We conducted a validation study of the distributed decentralized b+ tree with weak-consistency using the Alloy specification language and we proved that these weak-consistency invariants are consistent with the search operation.

(2) We also defined and tested several algorithms for the transfer of responsibility, i.e. the Ping-Pong and the Ping-Only algorithms, and we showed that it is always possible to find another peer that can be responsible for the transferred ranges of a given node at the same level when performing node split and merge operations.

(3) We also added the assumption of successiveness to the split and merge update algorithms described in [1] and revised parts of these algorithms, related to the

new assumptions in order to guarantee an optimal level of cost-efficiency for the search and update operations under the weak consistency.

(4) We have also implemented a simulation system for the distributed decentralized b+ tree under weak consistency. We have shown by our simulation results that it is possible to distribute a b+ tree structure among many peers by defining weak-consistency invariants that maintain the correctness of search operations. We also proved through simulation that it is possible to maintain relatively equal workload for each leaf-node while each non-leaf node maintains a partial replication of the state of the distributed b+ tree. The simulation program is also designed in such a way that it is relatively easy to make some changes in order to obtain a real implementation of peer nodes that would run of different computers interconnected by the Internet.

9. 2 Future work

We proved using the Alloy program that the weak-consistency invariants introduced in Section 4.1 are consistent with the search operations. The next step would be to prove that these weak-consistency invariants are necessary and sufficient for the search operation to be correct. In other words, we would like to know how weak the distributed b+ tree structure could be and remain consistent with the search operation. Also, it would be interesting to check that these weak consistency invariants are maintained by the node split and merge operations.

We also plan to find solutions to overcome the high number of back-pointers and to identify the reasons for this phenomenon. One intuitive way to solve this problem is to duplicate the entries of the nodes having many back-pointers into some other empty nodes and partitioning the back-pointers among these nodes so that it allows different nodes to access the same entries through different paths, which may decrease the number of back-pointers maintained by the original node.

It would be also interesting to study the fan-out property of b+ trees and find out which settings should be used so that the system reaches its highest level of performance in terms of load balancing and search message complexity. In particular, what is the most suitable number p that may be used to obtain a reasonable mean of the fan-out for the b+ tree so that one can keep the loads of the entries balanced among all nodes and obtain a relatively low depth that may optimize the cost of the search operation.

Finally, we plan to compare the complexity of the update algorithms of the distributed decentralized b+ tree structure under weak-consistency with the fully consistent distributed version of the b+ tree.

References

[1] S. Asaduzzaman and G. V Bochmann, {asad,bochmann}@site.uottawa.ca*, Distributed B-tree with Weak-consistency*, University of Ottawa, Ottawa, unpublished paper, 2010.

[2] K. Birman, G. Chockler, and R. v Renesse. *Toward a cloud computing research Agenda*, ACM SIGACT News, 40(2), pp:68-80, 2009.

[3] M. K. Aguilera, W. Golab, and M. S. Shah. *A practical scalable distributed B-tree. Proc. VLDB Endow*, 1(1), pp:598-609, 2008.

[4] A. Bar-Noy and D. Dolev. *A partial equivalence between shared-memory and message-passing in an asyncrhonous fail-stop distributed environment. Mathematical Systems Theory*, 26, pp:21-39, 1993.

[5] M. Kurant, A. Markopoulou and P. Thiran, *On the bias of BFS (Breadth First Search)*, International Teletraffic Congress, 22, 2010.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. *Bigtable: a distributed storage system for structured data, ACM Transactions on Computing Systems*, 26(2), pp:1-26, 2008.

[7] R. A. Finkel and J. L. Bentley. *Quad trees a data structure for retrieval on composite keys*, Acta Informatica, 4(1), pp:1-9, 1974.

[8] A. Guttman, R-Trees: *A Dynamic Index Structure for Spatial Searching*. IGMOD, 84, pp:47-57, 1984.

[9] H. T. Kung and J. T. Robinson. *On optimistic methods for concurrency control, ACM Trans, Database Syst*., 6(2), pp:213-226, 1981.

[10] D. Jackson, *Software Abstractions, logic Language and analysis*, 1st Edition, MIT press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2006.

[11] M.T. Ozsu and P. Valduriez, *Distributed and Parallel Database Systems, In Handbook of Computer Science and Engineering*, A. Tucker (ed.), CRC Press, pp:1093-1111, 1997.

[12] R. Schollmeier, *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications*, Proceedings of the First International Conference on Peer-to-Peer Computing (P2P.01), IEEE, 7(2), 2002.

[13] Cormer, Douglas*, The Ubiquitous B- tree, Computing Surveys* 11(2), pp: 123-137, 1979.

[14] A. Carzanig and M. Rutheford, *SSim, a simple Discrete-event Simulation Library,* University of Colorado,{ Antonio.Carzaniga, **Matt.Rutherford**}@usi.ch, http://www.inf.usi.ch/carzaniga/ssim/index.html, , 2003.

[15] T. H Cormen, C. E Leiserson, R. L Rivest*, Introduction to Algorithms*, The College of Information Sciences and Technology at Penn State, MIT Press, 3rd Edition, Chapter 5, 2009.

[16] D. A Heger, *A Disquisition on The Performance Behavior of Binary Search Tree Data Structures*, European Journal for the Informatics Professional 5 (5), pp: 67–75, 2004.

[17] G. Reese, *Database Programming with JDBC and Java*, 2nd Edition, Chapter 7, Distributed Application Architecture, pp:126-145, 2000.

[18] P A. Bernstein and E. Newcomer, *Principles of Transaction Processing*, 2nd Edition, Morgan Kaufmann (Elsevier), pp: 245-278, 2009.

[19] G. Booch, I. Jacobson and J. Rumbaugh, *OMG Unified Modeling Language Specification*, Version 1.3 1st Edition, pp:10-27, 2000.