Research on the RRB+ Tree for Resource Reservation

Libing Wu[1,2,*], Ping Dang[1], Lei Nei[1,2], Jianqun Cui[3], and Bingyi Liu[1]

[1] School of Computer Science, Wuhan University, Wuhan, China
[2] State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
[3] School of Computer Science, Central China Normal University, Wuhan, China
cswlb@126.com

Abstract. The performance of the data structure has a significant impact on the overall performance of the advance resource reservation in the distributed computing. Because the query and update operations of the B+ tree are of high efficiency, so this paper proposes a B+ tree structure suitable for resource reservation - the RRB+ tree. Also, we design and implement the corresponding algorithms of query, insertion and deletion. Different with the B+ tree that insert and delete one key word at a time, the RRB+ tree insert one reservation request and delete one tree node every time. The RRB+ tree is of a higher precision of expression. With the fixed reservation admission control algorithm and the same rate of acceptance, the experimental results show that the RRB+ tree is easier to operate for the complex and changing network environment, and have a higher utilization of storage space.

Keywords: Data structure, Advance resource reservation, RRB+ tree, Loop time slot array.

1 Introduction

In high-performance distributed computing environments, some applications require access to distributed heterogeneous resources. These resources are often located in different zones and subject to different management strategies, which make it difficult to co-allocate them. In order to solve this problem, a method of resource reservation is used. It can ensure that all required resources are available at the same time in a specific period of the future time, and it can guarantee the QOS of services [1]. According to the use time of reserved resources, the resource reservation can be divided into two types - the immediate resource reservation and the advance resource reservation. The latter is more widely used because of its flexibility, and this study focus on it.

The optimization of the data structure's performance plays a pivotal role in improving the overall performance of the advance resource reservation. Data structures are mainly used to store the real-time resource reservation information, involving operations of queries, insertions, and deletions. Among the processing time of the

* Corresponding author.

C.-H. Hsu et al. (Eds.): NPC 2013, LNCS 8147, pp. 83–93, 2013.

resource reservation, 60% is for the processing of the data structure, 8% is for the selection of suitable resources, and the remaining 32% is for the management of the resource reservation mechanism [2]. The key problem of building a data structure suitable for resource reservation is how to speed up the query rate and reduce the data redundancy. To this end, this paper studies the B+ tree for resource reservation - the RRB+ tree. By comparison with the loop time slot array, it is of better precision of expression, and higher utilization of storage space for the complex and changing network environment.

2 Related Works

The existing data structures can be divided into two categories: slot data structures, and non-slot data structures. The time slot array [3-4] is a typical example of slot data structures. Each array element represents a time slot, and the value of it represents the amount of reserved resources. The time slot array is simple and easy for the admission control of reservation requests. However, there are also many inadequacies. First, if the duration of a request is too long, a considerable number of array elements will need to store the information, which is a waste of memory. Second, its precision of expression is very low, which means that if the unit of the array is a second, you can not describe the precise time in milliseconds. Third, the size of the array is affected by the parameters of reservation requests. The slot-based segment tree [5] is another time slot data structure. With the flexible resource reservation, Mugurel et al. improves the segment tree so that it can be better applied to actual environments [6].

In the studies of non-slot data structures, Qing Xiong et al. propose a data structure based on the single linked list [7]. The experimental results show that its memory consumption is far less than the time slot array, and greatly superior to it in time consumption when the volume of requests is not very large. Libing Wu et al. propose an improved single linked list structure - the indexed list [8]. The experiments show that its memory consumption is lower than the single linked list, and its query time is shorter than the time slot array.

The tree structures are another widely studied non-slot data structures. Tao Wang et al. propose a bandwidth reservation resource tree [9]. All leaf nodes in the tree have the same depth. Each node represents a non-empty time interval. Each leaf node occupies a time interval, and the amount of remaining resources within this interval is the same. The interval that the parent node describes is the sum of the intervals occupied by all its children. However, the experimental results [7] show that the bandwidth reservation resource tree is not as good as the slot array in both the processing ability and the memory consumption. Other tree structures include the binary search tree [10] and the resource binary tree.

3 The RRB+ Tree

The B+ tree is typically used in the database and the operating system's file system. Data in the B+ tree can be kept stable and in order. The insertion and update algorithms

of the B+ tree are of logarithmic time complexity, which means that the insertion and update operations can be done efficiently. Therefore, according to the characteristics of resource reservation, we improve the B+ tree's node structure and related algorithms to obtain better performance, and that is the RRB+ tree.

The reservation requests are defined as follows:

Request = (*bw*, *ts*, *td*);

The parameter *bw* indicates the reserving bandwidth within each unit time, the parameter *ts* indicates the beginning time of the reservation, and the parameter *td* represents the reserving duration. Different from the B+ tree that insert a keyword each time, the RRB+ tree insert a reservation request, which means that the RRB+ tree consider the time values as keywords and the bandwidth of that time as a record.

3.1 Tree Node Structure

The non-leaf node of an m order RRB+ tree contains only the largest keywords of its sub trees. There are two head pointers: the pointer *root* to the root node and the pointer *first* to the leftmost leaf node. The nodes of the tree are defined as follows:

```
struct Bnode
{
        int keynum;
        int key[m+1];
        int record[m+1];
        BNode *ptr[m+1];
        BNode *parent;
        int seq;
        struct BNode *next;
};
```

The parameter *keynum* represents the number of keywords in the node, and its range is [$\lceil m/2 \rceil$, m]. (*key*, *record*, *ptr*) describes a keyword: the parameter *key* represents the value of the keyword and it used to store the time in this article; the parameter *record* shows the resource reservation information of *key*, and its initial value is 0; and the parameter *ptr* is a pointer to the sub tree associated with that keyword *key*. The parameter *parent* points to the parent node of this node, the parameter *seq* indicates that it is the *seq*-th children of the parent node, and the parameter *next* is a pointer to the next node of the same level. Each node consumes the memory of 4 * (4 + 3 * (m +1)) bytes.

Figure 1 shows a third order RRB+ tree. From the figure, we can get the current bandwidth reservation information stored in this tree: the amounts of reserved resources is 10 during the 0-14 time period, 90 during the 15-26 time period, 50 during the 27-35 time period, and so on.

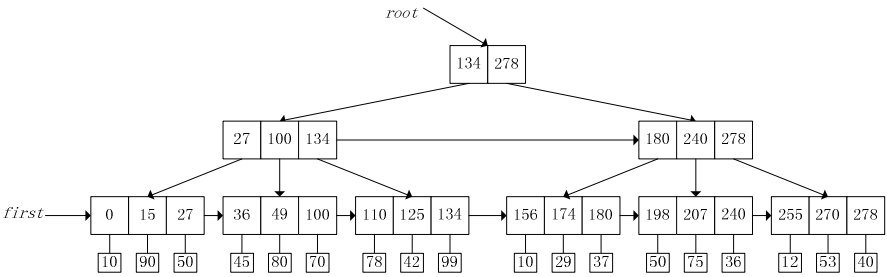

Fig. 1. Third Order RRB+ Tree

3.2 Algorithms

3.2.1 Query

After the server receives the request *Request* (*bw*, *ts*, *td*), it should find out whether the reserving start time (the keyword *ts*) is in the RRB+ tree. The query results are as follows:

(1) Empty tree;
(2) The tree is not empty, and the keyword *ts* is greater than all keywords in the current tree;
(3) The tree is not empty, *ts* is not greater than the largest keyword but it is not a keyword in the tree;
(4) The tree is not empty, and *ts* is one of the keywords.

The first two cases can be directly judged. For the first case, if the pointer *root* is null, the tree is empty. For the second case, the rightmost keyword of the node *root* is the biggest one in the tree. For the latter two cases, the query operation is similar to that of the binary sort tree. Traverse the tree from top to down since the node *root* – in every internal node, if *ts* is not bigger than a keyword of it from left to right, then enter the corresponding child node of that keyword until the traversal comes to leaf nodes. The structure of the query result is as follows:

```
struct Result
{
        BNode * ptr;
        int i;
        int tag;
};
```

The parameter *ptr* points to the leaf node where the insertion should begin for the latter two cases. For the first two cases, it will be NULL because it has to create a tree in the first case and the insertion place is clear in the second case (the rightmost leaf node). The parameter *i* indicates that the keyword *ts* should be the *i*-th keyword of the node *ptr*

for the third case, and is originally the *i*-th keyword of the node *ptr* for the fourth case. For the first two cases, it will be zero. The parameter *tag* describes which one of the four cases is true for that result.

3.2.2 Insertion

On the basis of the query results above, the admission control will be done first. If it is of the first two cases, the server can directly decide the current reservation request to be admitted. For the latter two cases, the sever needs to read the resource reservation information during the time period of [*ts*, *ts* + *td*), and thus judges whether the request can be accepted. If it is able to accept the request, the corresponding insertion operation will be done.

If the tree is empty, establish a tree and insert the request. If the tree is not empty, the insertion operation will be divided into two steps: the insertion of the starting keyword *ts* and the insertion of the end keyword (*ts*+*td*). The insertion is only done in leaf nodes. When the number of keywords in the node is greater than m, it needs to be split into two nodes and their parent node should contain both biggest keywords of them, which may lead to the splits of internal nodes layers up. If the split is at the node *root*, a new one should be created. The implementation process of the internal nodes' split algorithm is as follows:

```
void Split(BNode *tmp)
while(tmp->keynum > m)
    //create a new node t as a successor to the node tmp, and the latter
    //tmp->keynum/2 keywords are moved to t
    Create(t);
    if(tmp == root)
        //create a new node as root with the node tmp and t as its children
        CreateRoot(tmp, t);
    else
        //update the node tmp's parent node so that it contains both the
        //biggest keywords of the node tmp and t
        Update(tmp->parent);
        //continue upward, to see whether the internal nodes need to split
        tmp = tmp->parent;
    end if
 end while
```

For the insertion of the starting time *ts*, there will be no operation if the current tree contains the keyword *ts*(in case 4). If *ts* is greater than all the keywords in the current

tree (in case 2), update the biggest keywords of each internal layer with the value of *ts* from top to down. And insert *ts* into the rightmost leaf node, setting its record value as equal as that of its previous keyword. If the tree doesn't contain the keyword *ts* and it is not greater than the largest keyword of the current tree (in case 3), insert it into the leaf node that the pointer *ptr* points to and set its record value the same as the previous keyword. Keywords' insertion may lead to leaf node's splitting and it is slightly different with the splitting algorithm of internal nodes (shown in the above algorithm) - the keyword may fall into the node *tmp* or *t* (the variables *tmp* and *t* are in the above algorithm), so the query result needs to be updated according to the real situation. The parent nodes will contain one more keyword, which may lead to the split operations upward. The insertion algorithm of *ts* is shown as follows:

```
void DealWithTs(Result r)
if(r.tag == 2)
    //update the biggest keywords of internal layers with the value of ts
top-down
    ReplaceDown(ts);
    //r.ptr is the rightmost leaf node•insert ts to r.ptr
    Update(r.ptr);
else if(r.tag == 3)
    //insert ts to r.ptr as the i-th member
    Insert(r.ptr, i);
end if
if(r.ptr->keynum > m)
    //if the leaf node splits after the insertion, update r
    SplitUpdate(r.ptr);
    if(r.ptr->parent > m)
        Split(r.ptr->parent);
    end if
end if
```

Take the 3 order RRB+ tree shown in Figure 1 as an example, insert the *request* (30, 108, 20) into it, and that leads to the calling of the insertion algorithm of *ts* at first. Insert the keyword 108 to the third leaf node on the left and set its record value equal to that of the previous keyword 100 (that is 70). Then the number of keywords in that leaf node becomes 4, greater than 3, so it must be split. After the splitting, the number of keywords in its parent node becomes 4, so it needs to split, too. The new tree is shown in Figure 2.

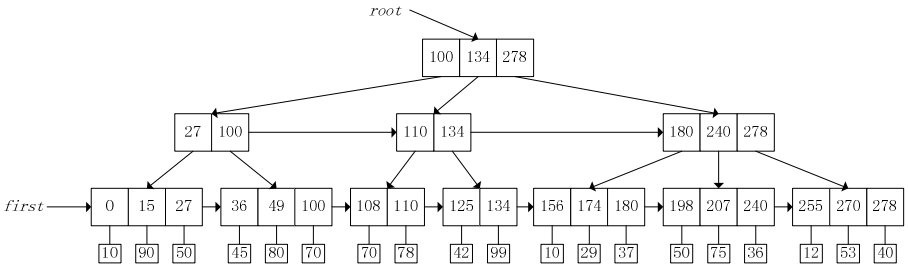

Fig. 2. After the Insertion of the *ts* (108)

To insert the *request* (30, 108, 20) completely, it needs to traverse 20 time units starting from the keyword 108. The next keyword of 108 is the keyword 110 and two units of time between them, which is less than 20, so the record value of keyword 108 updates to 100 (70 + 30 = 100). The next keyword of 110 is the keyword 125 and 15 time units between them, which is less than 18 (20 - 2 = 18), so the record value of the keyword 110 updates to 108 (78 + 30 = 108). The next keyword of 125 is the keyword 134 and 9 time units between them, which is greater than 3 (18 - 15 = 3), and then the keyword 128 (108 + 20 = 128) is inserted between the keywords 125 and 134, with its record value equal to the current record value of the keyword 125 (42), and the record value of keyword 125 becoming 72 (42 + 30 = 72). After these operations done to the tree in the Figure 2, it changes and its new state is illustrated in Figure 3.

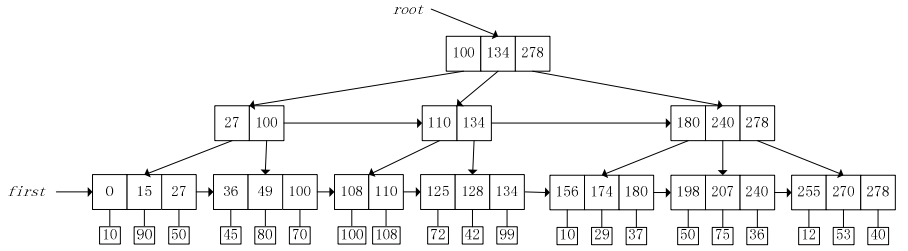

Fig. 3. After the *Request* (30, 108, 20) Inserted Completely

After the insertion of *ts*, the parameter *ptr* points to the node where the keyword *ts* has been inserted and the parameter *i* indicates the position of *ts* in the node *ptr* pointing to for all these four search results, which makes the subsequent insertion algorithm of (*ts* + *td*) transparent to the four different insertion cases and simplifies this algorithm implementation. In the (*ts* + *td*) insertion algorithm, firstly all record values of keywords during the time period [*ts*, *ts* + *td*) are updated, and then the keyword (*ts* + *td*) is inserted.

4 Comparative Experiments

4.1 The Loop Time Slot Array

As time goes on, resources that have been reserved will be used, and there are new resources available for reservation. The traditional time slot array consumes much memory, resulting in poor performance. So this paper chooses the loop time slot array as the data structure for the comparison with the RRB+ tree. As shown in Figure 4, the size of this array is represented by the parameter *MAX*, and the size of a slot is represented by the parameter *SLOT*. The initial value of the variable *time* is 0. It will point to the next slot once a slot of time has been gone, and reset the value of the prior slot zero. When it points to the slot *MAX*-1, after a slot of time it will point to the slot 0 again, i.e., the variable *time* indicates the value of the current time (Its exact value is *currentTime%MAX*). When the request *Request*(*bw*, *ts*, *td*) arrives, find out the starting position in the loop slot array with the values of the variables *time* and *ts*, and then the array cycle down *td* units to reserve resources.

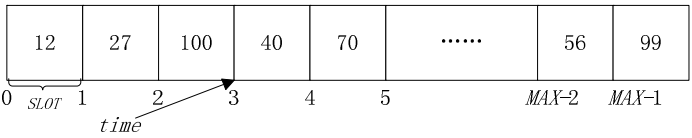

Fig. 4. The Loop Time Slot Array

4.2 Experimental Environment

Use Visual Studio 2010 as the development platform and the programming language is C++. Characterizations of the relevant parameters are as follows:

avgI, the average interval between the time requests arrive. The moments when requests arrive follow the Poisson distribution with the parameter λ, and $avgI = \lambda$. If $\lambda == SLOT$, the server receives a request each time slot on average.

bw, the reserved bandwidth. Its value follows the uniform distribution within the range of ($b1$, $b2$) and the mean value is $B = (b1 + b2)/2$.

td, the duration. It follows the exponential distribution and the variable *E* represents its mean value.

BWMAX, the maximum bandwidth that the system can provide per unit time.

Interval, the interval between two deletions of the RRB+ tree.

4.3 Results and Analysis

The fixed reservation admission control algorithm is used to decide whether to accept a reservation request or not. In the simulation experiments, the admission control rate will be about 98% by adjusting the value of the variable *BWMAX* after the values of other parameters are set. So the performance evaluation of the two data structures will be more objective. The settings of the relevant parameters are as shown in Table 1.

Table 1. The Experimental Parameters

	m	*BWMAX* (*KB*)	*SLOT* (*ms*)	λ (*ms*)	*b1* (*KB/ms*)	*b2* (*KB/ms*)
Set 1	3	8000	1	100	100	1000
Set 2	5	8000	1	100	100	1000
Set 3	3	8000	1	100	100	1000
Set 4	3	8000	1	1000	100	1000

	B (*KB/ms*)	*b3* (*ms*)	*b4* (*ms*)	*T* (*ms*)	*E* (*ms*)	*Interval* (*s*)
Set 1	550	10	100	55	1000	1
Set 2	550	10	100	55	1000	1
Set 3	550	10	100	55	1000	60
Set 4	550	10	100	55	1000	60

The memory consumption of the RRB+ tree and the loop time slot array is shown in figure 5 below. "BD" represents the memory consumption of the RRB+ tree before deletions, "AD" represents the memory consumption of the RRB+ tree after deletions, and "Array" represents the memory consumption of the loop time slot array. In order to ensure the accuracy of experimental results, reservation requests of these four settings are the same (that means the random seed to generate the parameters of reservation requests is unchanged), and the value of the parameter *BWMAX* is also the same, so they get the same resource reservation results.

In Figure 5, the memory consumption of the loop time slot array in set 1, 2 and 3 is the same, and that in set 4 is different. That is because it takes the difference between the maximum reserved time and the current time as the actual memory consumption. And the values of λ in set 1, 2 and 3 are the same, so their resource reservation situation is the same and thus obtain the same memory consumption at the same output time. But the value of λ in set 4 is different, resulting in the different memory consumption at the same output time.

The only difference between set 1 and 2 is the different orders of trees. The orders of RRB+ trees are 3 in set 1 and 5 in set 2. And their performances of the memory consumption are almost the same. The only difference between set 1 and 3 is the frequency of deletions. The frequency is one second in set 1 and one minute in set 3, which means that the deletions will be done probably every 10 requests received in set 1 and 600 in set 3. So the memory consumption before the deletion of RRB+ tree in set 3 is much larger than that in set 1.

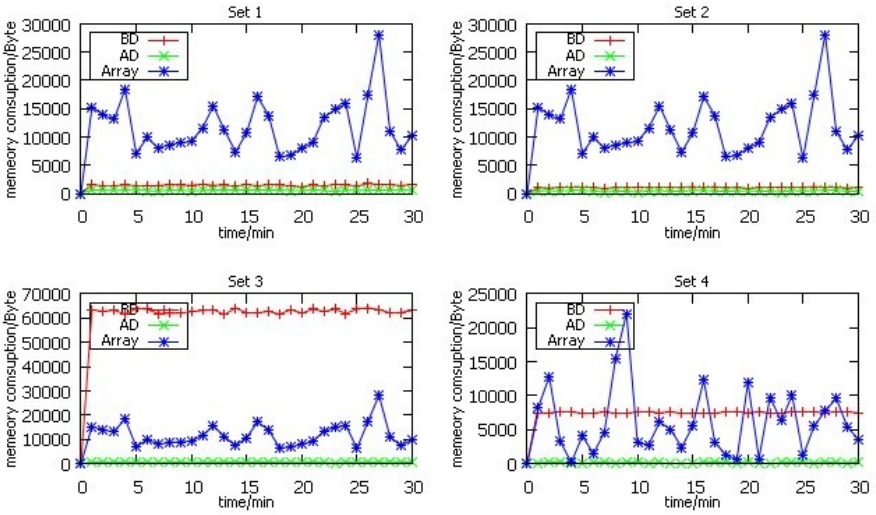

Fig. 5. Comparison of Memory Consumption

The only difference between set 3 and 4 is that the value of λ in set 4 is ten times as large as that in set 3. So it probably receives 10 reservation requests in set 3 and one in set 4 per second. And the memory consumption of the RRB+ tree in set 4 has been significantly improved compared to that in set 3.

It can be seen that the memory consumption of the RRB+ tree is far less than that of the loop time slot array, when its deletion interval is appropriate. For different kinds of reservation requests, the RRB+ tree can achieve better performance through appropriately adjusting its parameters. The memory consumption of the loop slot array only associates with the largest reserved time and the current time. And if requests reserve resources far earlier than the time they use them or the durations are too long, its memory consumption will greatly increase. Also it is hard to set the size of the loop slot array. Therefore, the storage space utilization of the RRB+ tree will be much higher than that of the loop time slot array for the complex and changing network environment.

5 Conclusions

The performance optimization of the data structure plays a pivotal role in improving the overall performance of resource reservation. Data structure is mainly used to store the real-time resource reservation information, involving the operations of query, insertion and deletion. The query and update operations of the B+ tree are of high efficiency, so this paper propose a B+ tree structure suitable for resource reservation - the RRB+ tree, and design the corresponding algorithms for it. The results of experiments compared with the loop time slot array show that the storage space utilization of the RRB+ tree

will be much higher than that of the loop time slot array for the complex and changing network environments.

Acknowledgment. This work is supported by National Science Foundation of China (No. 61070010, 61170017, 61272212) and Science & Technology Plan of Wuhan city.

References

1. Zhan, G., Siwei, L.: Dynamic grid resource reservation mechanism based on resource-reservation graph. Journal of Software 22(10), 2497–2508 (2011) (in Chinese)
2. Burchard, L.-O.: Analysis of data structures for admission control of advance reservation requests. IEEE Transactions on Knowledge and Data Engineering 17(3), 413–424 (2005)
3. Burchard, L.-O., Heiss, H.-U.: Performance evaluation of data structures for admission control in bandwidth brokers. Technical Report TR-KBS-01-02, Communications and Operating Systems Group, Technical University of Berlin (May 2002)
4. Andreicaf, M.I., Ţăpuş, N.: Time slot groups - a data structure for QoS-constrained advance bandwidth reservation and admission control. In: 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Timisoara, pp. 354–357 (September 2008)
5. Brodnik, A., Nilsson, A.: An efficient data structure for advance bandwidth reservations on the Internet. In: Proc. of the 3rd Conference on Computer Science and Electrical Engineering, pp. 1–5 (2002)
6. Andreicaf, M.I., Ţăpuş, N.: Efficient data structures for online QoS-constrained data transfer scheduling. In: International Symposium on Parallel and Distributed Computing, Timisoara, pp. 285–292 (July 2008)
7. Xiong, Q., Wu, C., Xing, J., Wu, L., Zhang, H.: A linked-list data structure for advance reservation admission control. In: Lu, X., Zhao, W. (eds.) ICCNMC 2005. LNCS, vol. 3619, pp. 901–910. Springer, Heidelberg (2005)
8. Wu, L., Yu, T., He, Y., Li, F.: A index linked list suited for resource reservation. Journal of WUT (Information & Management Engineering) 33(6), 904–908 (2011) (in Chinese)
9. Wang, T., Chen, J.: Bandwidth tree - a data structure for routing in networks with advance reservation. In: 21st IEEE International Performance, Computing, and Communications Conference, Phoenix, AZ, pp. 37–44 (April 2002)
10. SchelBn, O., Nilsson, A., Norrgkd, J., Pink, S.: Performance of QoS agents for provisioning network resources. In: Seventh International Workshop on Quality of Service(IWQoS 1999), London, pp. 17–26 (1999)