International Journal of Computer Applications (0975 - 8887)
Volume 176 - No.3, October 2017

Deterministic Cluster-based Skip List Protocol for Dynamic Distributed Systems

Ahmed A. A. Gad-ElRab
Department of Mathematics
Faculty of Science
Al-Azhar University
Cairo, Egypt

T. A. A. Alzohairy
Department of Mathematics
Faculty of Science
Al-Azhar University
Cairo, Egypt

Khaled A. A. Khalf-Allah
Department of Mathematics
Faculty of Science
Al-Azhar University
Cairo, Egypt

ABSTRACT

Dynamic distributed system (DDS) is a continually running system with a large number of entities as processes or nodes that are connected with each other and each of them has only a partial view of the system as Peer-to-peer (P2P) system which can be decentralized, centralized or hybrid of both to make the system operations faster as possible. In P2P, any number of entities can join and leave the system at any moment which makes the topology of the system continuously changed. So, the system must deal with these changes to be stable as possible. So, data management algorithms are needed to build an overlay network which is a logical layer that used to store the information about these entities. Skip list structure is the most common and efficient overlay structure for data management in P2P systems. However, using this structure cannot minimize the time delay for query processes as searching, inserting, and deleting in case of there is a huge number of entities in the skip list. In addition, most of existing algorithms that use this structure have been developed based on a special structure of skip list and they did not be applicable for another structure of the skip list. In this paper, to overcome these drawbacks, a new skip List structure and query processing methods are proposed. The conducted simulation results show that the proposed structure and algorithm are much better than the existing algorithms in the time delay and the required number of steps to finish any query process.

Keywords

P2P, Distributed Systems, Dynamic Systems, Deterministic, Skip List

1. INTRODUCTION

The random nature of the dynamic distributed system is a continually running system with a huge number of entities connected with each other and each entity has a partial view of the system, and any moment any number of entities can be joining and leaving the system at any moment [2]. So, the topology of the system continuously changed, and the system must deal with these changes to be stable as possible. One of the most popular dynamic systems is peer-to-peer (P2P) system which can be unstructured/decentralized, structured /centralized networks. Of course, each model has an advantage and disadvantage. To make the system operations faster as possible hybrid model can be used. So, data management algorithms are needed to build an overlay network which is a logical or virtual layer that is built on the top of an existing physical network which is used to store the information about these entities [1].

Many data management algorithms are proposed and implemented to be used in such systems, and to reduce the time complexity of searching, updating and deleting processes. One of the most used data management algorithms is distributed hash table (DHT) for peer-to-peer systems. DHTs are used to lookup data in an unstructured environment based on $(key, value)$ pair, similar to hash tables. The big advantage of DHT is its scalability and fault tolerant architecture that makes it useful for several distributed search applications. The disadvantage of DHT its hash based look up which destroys the semantic locality of the keys, and because of the non-linear nature of the hash function that used to hash the keys and maps each one to its hash value. So, DHT can only provide service for a single query and cannot be used for systems which require range of query.

Distributed Segment Tree (DST) introduced to support of range query and cover query over DHT. DST is built on top of generic DHT based on the concept of a segment tree in maintaining the structure of ranges; DST is shown to be very efficient for supporting both range query and cover query in a uniform way. But it performs poorly for a large number of nodes because of the limitation on the number of keys at each node [17].

Skip List (SL) is a probabilistic alternative to balanced trees proposed by William Pugh. SL can be used instead of balanced trees [15]. In [15], the implementation of insertion, deletion and search algorithms are much simpler and faster than balanced trees based algorithms. SL is much faster compared to linked list. To search any element in a sorted linked list, we might have to traverse all the nodes of the list. Here the search complexity is $O(n)$. In a skip list can operate in the order of $\log n$ in average cases where n is the total number of elements [12]. As shown in Fig 1 and Fig 2 the SL is structure is constructed as levels. Each level is a sorted linked list. The lowest level contains n nodes, the next higher level is another linked list with subset number of nodes from previous level, and so on tell the top higher level reached with minimum number of nodes see Fig 2.

However, using this structure cannot minimize the time delay for query processes as searching, inserting, and deleting in case of there is a huge number of entities in the skip list. In addition, most

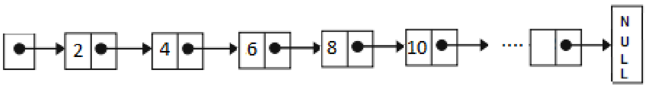

Fig. 1. Sorted Linked List

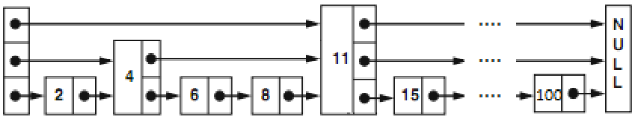

Fig. 2. Skip List

of existing algorithms that use the skip list structure have been developed based on a special structure of skip list and they did not be applicable for another structure of the skip list. In this paper, to overcome these drawbacks, a new cluster-based skip list structure and query processing methods are proposed. The rest of this paper is organized as follows: Section 2 gives an overview of the related works on distributed skip-list. Section 3 introduces ENTITY MANAGEMENT PROBLEM IN P2P and its formulation. Sections 4 introduces the proposed cluster-based skip list algorithm. Section 5 describes of the experimental and simulation results. Section 6 concludes this paper.

2. RELATED WORK

Skip lists are balanced by consulting a random number generator. Searching a linked list for a specific element may require that every element of the list be examined as shown in Fig. 3. If the list is stored in sorted order and every other element of the list also has a pointer to the element two ahead of it in the list, searching for an element requires that no more than $\lceil \frac{N}{2}+1 \rceil$ elements be examined (where N is the length of the list). By giving every $(2^i)^{th}$ th element a pointer 2^i elements ahead as shown in Fig. 4, the number of elements that must be examined can be reduced to $\lceil log_2 n \rceil$, while only doubling the number of pointers. Such that the levels of elements as shown in Fig. 4 are distributed in a simple pattern: 50% are level 1, 25% are level 2, 12.5% are level 3, and so on [15].

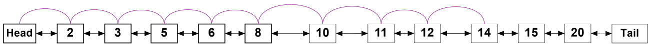

Fig. 3. Search inside Sorted Linked List to find entity number 14

There is a lot of algorithms which worked on skip lists [8], [9] and [10]. In [8], Interval skip list or IS-List was proposed to support interval indexing in a centralized environment. The IS-list is the first P2P system that achieves both content and routing path locality that allows stabbing queries and dynamic insertion and deletion of intervals. A stabbing query using an IS-list containing N intervals takes an expected time of $O(\log n)$. Inserting or deleting an interval in an IS-list takes an expected time of $O(\log 2n)$, if the interval endpoints are chosen from a continuous distribution [8]. In [3], and [6], [7] skip list was used to present an efficient and practical technique for dynamically maintaining an authenticated dictionary.

Harvey et al. have proposed a scalable overlay network called SkipNet [9]. SkipNet uses the randomized version of the skip list. So, it is difficult in the presence of an insertion and a deletion. In addition, due to the random nature of the data structure, it is difficult to determine the shape of the skip list which might create problem with the large number of nodes [11].

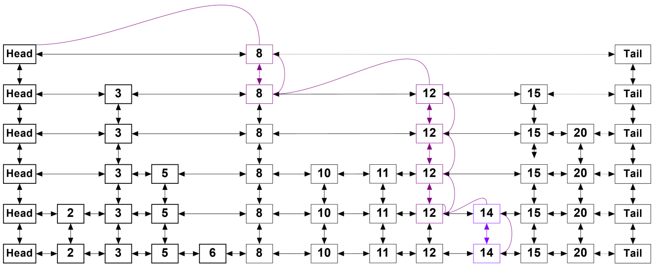

Fig. 4. Search inside Sorted Skip List to find entity number 14

In [16] the skip list is intoduced to overcome the challenge of on-demand streaming with asynchronous requests, demonstrate a practical skip list based streaming overlay with typical asynchronous streaming multicast VCR operations in the application-layer overlay networks [5], and show that the skip list based overlay is highly scalable, with smooth playback for diverse interactivities, and low overheads. So, the skip list can be applied to the management of a massive set of replying comments in web bulletins by exploiting the power-law distribution [10].

Many other algorithms introduced deterministic skip list to give an upper bound of worst case insert and delete complexity. In [13] several versions of deterministic skip lists have been introduced, simple data structures with guaranteed logarithmic search and update costs. In these algorithms, a skip list can be viewed as a multiway search tree in which the path length from the root to any leaf is the same; this path length corresponds to the height of the skip list, or to the number of vertical steps in the path for the search of any element in the skip list.

Corona [14] is a deterministic self-stabilizing algorithm for skip list construction in structured overlay networks. Corona operates in the low-atomicity message-passing asynchronous system model. A self stabilizing sparse 0–1 skip list on top of a self stabilizing sorted list call Tiara presented in [4].Here, 0-1 skip list means that any step between any two entities n_i and n_{i+1} at $level_j$ can only skip 0 (skip no entities) or 1 (only one entity skipped) in level $level_{j-1}$ of skip list. They have proposed a self-stabilizing algorithm for a sorted list first, and then have extended it to the sparse 0-1 skip lists which allows logarithmic searches and topology updates. They proved that the algorithm is correct in the shared register model.

In [12] a set of algorithms has been proposed to overcome the shortcomings of existing architectures, as proposed in Tiara [4] and Corona [14]. Beacuse the Tiara architecture cannot be extended for 1–2 skip lists. Deterministic 1–2 skip list works better for searching in peer-to-peer and overlay networks, as it resembles a balanced 2–3 tree, and thus supports range queries effectively.

3. ENTITY MANAGEMENT PROBLEM IN P2P

Entity management problem (EMP) is one of the key issues in the P2P systems. This problem deals with how to minimize the time cost for entity management in dynamic environments. In this section, the assumptions and models will be described then the formulation of EMP problem will be introduced.

3.1 Assumptions, and Models

We assume that there is a dynamic environment which contains a set of devices as nodes N, which share information and recourses. Each node can join or leave the system at any moment, and the topology of the system changes over the time and these changes may make the system fails. We assume that the P2P system model constructed from set of nodes $N = \{node_1, node_2, node_3, ..., node_i : i > 0\}$. Each $node_i$ contains $(key_i, value_i)$, these nodes are connected together and communicating with each other by sending messages to share information and resources like CPU and Memory.

3.2 Problem Formulation

The total average delay insert operation time is denoted as $T_{I_{total}}$. The total average delay delete operation time is denoted as $T_{D_{total}}$. The total average delay search operation time is denoted as $T_{S_{total}}$. The big challenge is to deal with the changes in the topology of the system by applying these changes on the entity management of data structure such insertion and deletion, to make an efficient search operation.
So, based on the proposed system model, the total average delay time $T_{I_{total}}$ and $T_{D_{total}}$ are calculated as following:

$$T_{I_{total}} = T_S + T_I \tag{1}$$

$$T_{D_{total}} = T_S + T_D \tag{2}$$

where T_I is the delay time of insert operation, T_D is the delay time of delete operation and T_S is the delay time of search operation.
Objective function: the objective function is to minimize the total avrage delay time cost for $T_{I_{total}}$, $T_{D_{total}}$ and $T_{S_{total}}$ operations by reducing T_I, T_D and T_S. This objecctive function is defined as follows.

$$\text{Minmize} \quad T_{I_{total}} + T_{D_{total}} + T_{S_{total}} \tag{3}$$

such that

$$\begin{aligned} &\exists\, path(node_i,\ node_{i+1});\\ &\forall\, node_i, node_{i+1} \in\ N\\ &\text{and} \quad node_i \neq node_{i+1} \end{aligned} \tag{4}$$

$$\begin{aligned} if\ value(node_i) &= value(node_{i+1})\\ \Rightarrow key(node_i) &= key(node_{i+1})\\ \Rightarrow node_i &= node_{i+1}\\ \forall\, node_i, &node_{i+1} \in N \end{aligned} \tag{5}$$

$$steps(search) \neq \infty \tag{6}$$

$$T_{stop}(node_i) \leq T_{stop}(node_j);\ \forall\, node_i, node_j \in N, j \leq i \tag{7}$$

where constraint (4) means that there is no circle or path between any node and itself to avoid infinite loops i.e. if there is a path from $node_i$ to $node_{i+1}$ then $node_i$ and $node_{i+1}$ are not equal. constraint (5) means that the value of any nodes is not the same i.e. if the values of two nodes $node_i$ and $node_j$ are equal then the nodes are the same node to avoid the duplication of nodes. constraint (6) means that the total number of required steps to reach any node must be finite to avoid infinite loop constraint (7) means that the required time to stop search operation of $node_i$ less than or equal the required time to stop search operation of $node_j$.

4. CLUSTER-BASED SKIP LIST ALGORITHM

To solve the EMP problem, a new skip list algorithm called Cluster-Based Skip List Algorithm (CBSL) is proposed. This proposed algorithm is an improvement for the Standard Skip List (SL) algorithm by convert the standard Skip List to Clustered-Based Skip List.

4.1 Basic Idea

The proposed CBSL is based on (1) dividing each level in the skip list into a group of clusters which is considerd as new structure. (2) proposing two different methods for join, delete, and search operations. The first method called Sequential method (SM) which searches for any entity in a sequential order cluster by cluster. While the second method called Prediction method (PM) which searches for any entity by predicting its nearest cluster first, then jumps to this predicted cluster and searches for the required entity. In the rest of this section, formation of clusters is described, the proposed methods are introduced and finally the insert, delete, and search operations are described.

4.2 Clustering Formation Process

All nodes in SL are distributed on levels $\{level_1, level_2, level_3, ..., level_j : j > 0\}$. Each $node_i$ conatins $(key_i, value_i)$, and has foure links. Two for the connections $right_i$ and $left_i$ with its neighbors, and the other two links up_i and $down_i$ to connect the node with itself at different levels. In CBSL, these nodes will be distributed on clusters $M_{clusters} = \{C_1, C_2, C_3, ..., C_k : k > 0\}$. Each cluster has a number of entities C_{max} with the same levels in the SL as shown in Fig. 6. These clusters are another logical layer on SL to search about any entity by using two search methods to find which cluster contains on the target entity, which makes the search inside the cluster faster than the search in the whole skip list as shown in Fig. 5. Each cluster has maximum number of nodes C_{max}. C_{max} is fixed for all clusters and is unchanged through running time. Each cluster has three main elements $start_i, end_i$ and top_i elements, and has two connections to previous and next clusters $next_i$, $perev_i$. $start_i, end_i$ and top_i elements specified while building the skip list and changed after adding or deleting elements as shwon in Fig 6.

4.3 The proposed methods

Based on clustering design, CBSL uses two different searching methods: *Sequential method (SM)* and *Prediction method (PM)* These two methods are described as follows.
(1) **Sequential method (SM)**: which checks each cluster of the list in a sequential order cluster by cluster until it finds a cluster that has the target entity. If the algorithm reaches the end of the list, the search terminates with failure as shown in Fig (9) and and Algo (3)
.
(2) **Prediction method (PM)**: in this method, the proposed algorithm pridicates the posiotion of the cluster or the closest cluster position for the required entity as shown in Fig (10) and Algo (4) by using the following equation:

$$C_{ex} = \lceil \frac{X}{C_{max}} \rceil \tag{8}$$

Such that C_{ex} the expected position of the cluster, X is $NodeID$ and C_{max} maximum number of elements can be added for each cluster.

By using one of these two methods, CBSL checks if the target cluster was reached or not. If not, go to the next cluster or the previous one depending on the value of $nodeID$ and the $start_i$ and end_i of the current cluster. If the cluster does not exsit then a new one will created on the right postion and the connections added.

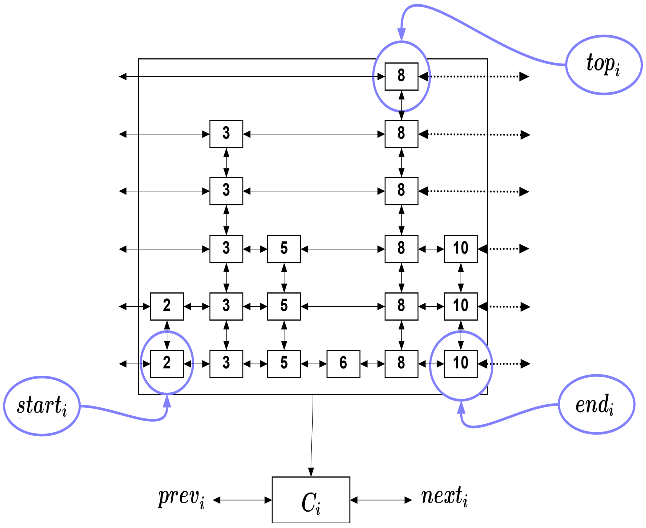

Fig. 6. Cluster Structure and it is Componets

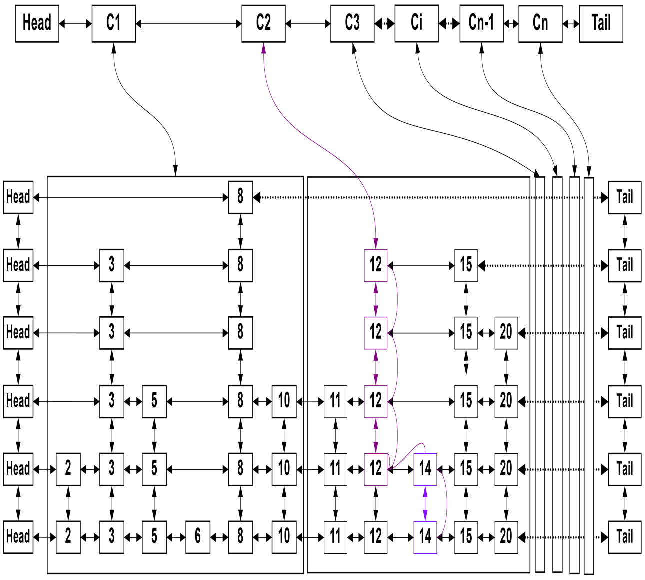

Fig. 7. Search inside CBSL

4.4 Search, Insert, and Delete Operations

4.4.1 Search operation. As shown in Fig 7 and Algo (1) , CBSL starts to find the position of cluster first by using SM or PM methods. Then starts to search inside the cluster to find the node position. The search inside the cluster from the top elemnet of this cluster. Then the search move to left or right or down based on the key of the node.

Algorithm 1: CBSL Seacrh Algorithm

```
Input  : NodeID
Output: Node position and data
$cluster \longleftarrow$ findClusterByPM(NodeID)
                                        ▷ or findClusterBySM(NodeID)
$Node \longleftarrow$ findNode(NodeID, cluster)
```

Algorithm 2: CBSL insert cluster algorithm ***insertCluster*** function

```
function insertCluster $(cluster, newCluster)$
Input  : cluster, newCluster
Output: New cluster position
    $newCluster_{next} \longleftarrow cluster_{next}$
    $newCluster_{prev} \longleftarrow cluster$
    $cluster_{next_{prev}} \longleftarrow newCluster$
    $cluster_{next} \longleftarrow newCluster$
    return $newCluster$
```

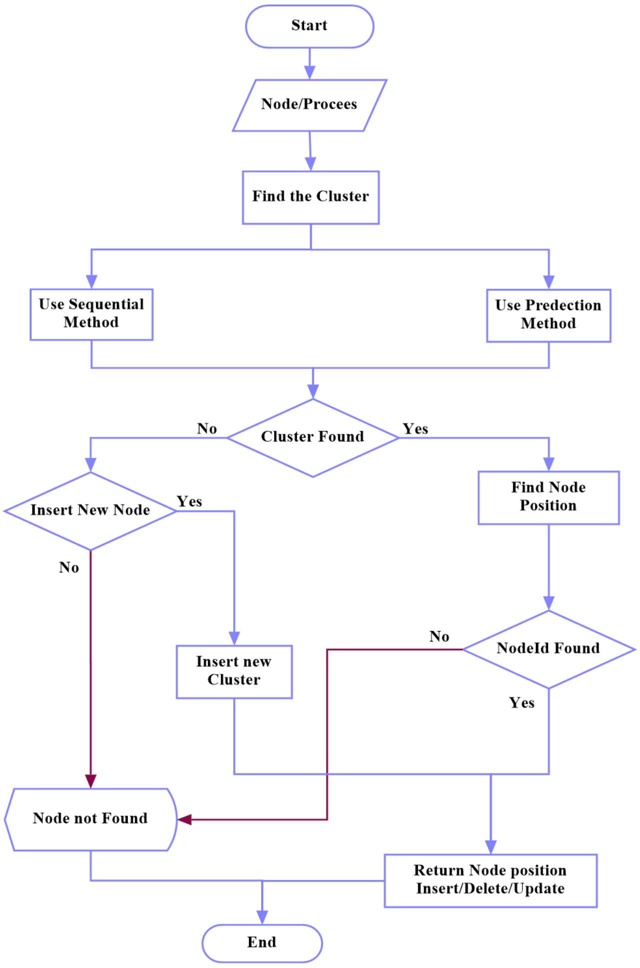

Fig. 8. Flow chart of the proposed CBSL

4.4.2 Insert Operation. To insert or join a new node in the list, CBSL follows the following steps as shown in Algo (9):

(1) Determine the level of the new node by using SM or PM methods as shown in Algo (1),(3), (4) .
(2) Insert a new node (NodeId) in its level.
(3) Add new edge between the left and a new node (NodeId).
(4) Add new edge between the right and a new node (NodeId).
(5) Delete the old edge between the left and the right nodes. (When describing a link we always state the smaller identifier first).

4.4.3 Delete Operation. To delete an existing node from the list, CBSL follows the following steps as shown in Algo (10):

(1) Search to find cluster which contains the node that will be deleted by using SM or PM methods.
(2) If the cluster exists then search inside it to find the node position, else do nothing.
(3) If the node exists go to next step, else do nothing.
(4) Add a new edge between the left and right nodes at all levels.
(5) Delete the old edge between the left and the right nodes. (When describing a link we always state the smaller identifier first).
(6) Update the other clusters if it is required.

Algorithm 3: Find Cluster using SM ***findClusterBySM*** function

```
function findClusterBySM ($NodeID$) ;
Input  : NodeID
Output: Cluster position
$cluster \longleftarrow cluster_0$
while $cluster \neq NULL$ do
    $start \longleftarrow cluster_{start}$
    $end \longleftarrow cluster_{end}$
    if $NodeID \geq start$ and $NodeID \leq end$ then
        return $cluster$
    if $cluster_{next} \neq NULL$ and $NodeID \geq cluster_{next_{start}}$ then
        $cluster \longleftarrow cluster_{next}$
    else
        return $failure$ ▷ or call insertCluster function for adding a new Node
```

Algorithm 4: Find Cluster using PM ***findClusterByPM*** function

```
function findClusterByPM ($NodeID$) ;
Input  : NodeID
Output: Cluster
$i \longleftarrow \left\lceil \frac{NodeID}{C_{max}} \right\rceil$
$cluster \longleftarrow cluster_i$
$start \longleftarrow cluster_{start}$
$end \longleftarrow cluster_{end}$
if $NodeID \geq start$ and $NodeID \leq end$ then
    return $cluster$

else if $NodeID \leq start$ and $cluster_{prev} \neq NULL$ and $NodeID \leq cluster_{prev_{end}}$ then
    return searchPrevClusters(NodeID, $cluster_{prev}$)

else if $NodeID \geq end$ and $cluster_{next} \neq NULL$ and $NodeID \geq cluster_{next_{start}}$ then
    return searchNextClusters(NodeID, $cluster_{next}$)
else
    return $failure$ ▷ or call insertCluster function for adding a new Node
```

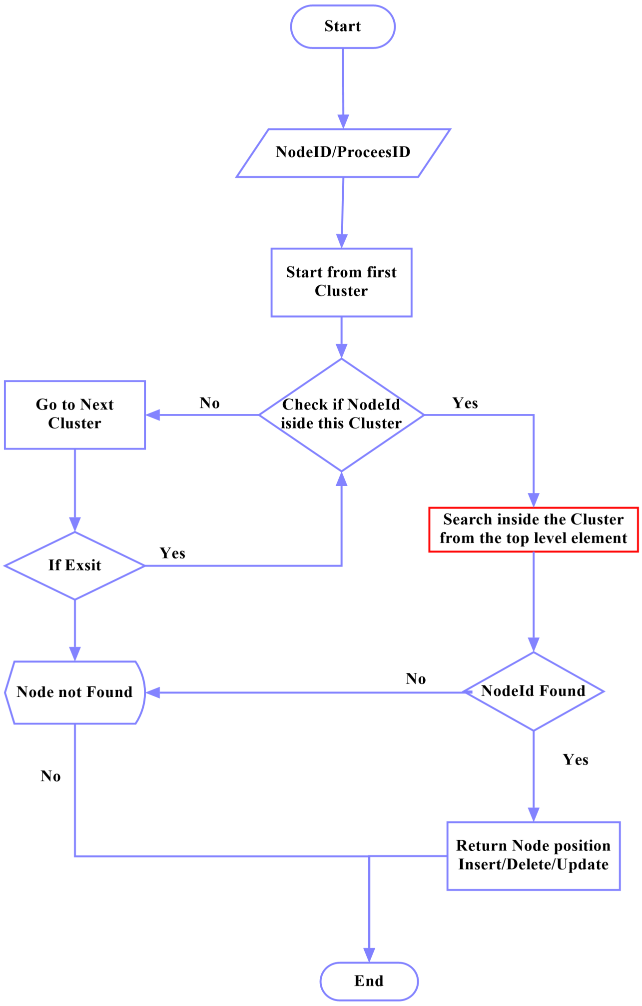

Fig. 9. SM Algorithm Flow Chart

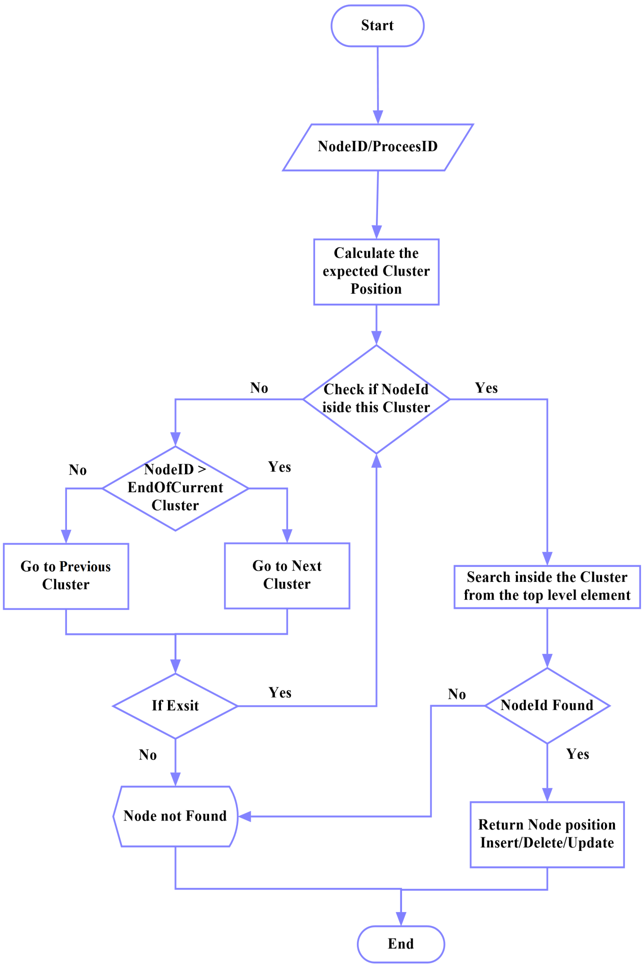

Fig. 10. PM Algorithm Flow Chart

Algorithm 5: Search next clusters ***searchNextClusters*** function

```
function searchNextClusters ($NodeID, cluster$) ;
Input  : NodeID, cluster
Output: Cluster position
$start \longleftarrow cluster_{start}$
$end \longleftarrow cluster_{end}$
if $NodeID \geq start$ and $NodeID \leq end$ then
    return $cluster$
while $NodeID \geq end$ and $cluster_{next} \neq NULL$ and $NodeID \geq cluster_{next_{start}}$ do
    $cluster \longleftarrow cluster_{next}$
    $start \longleftarrow cluster_{start}$
    $end \longleftarrow cluster_{end}$
    if $NodeID \geq start$ and $NodeID \leq end$ then
        return $cluster$
```

Algorithm 6: Search previous clusters ***searchPrevClusters*** function

```
function searchPrevClusters ($NodeID, cluster$) ;
Input  : NodeID, cluster
Output: Cluster position
$start \longleftarrow cluster_{start}$
$end \longleftarrow cluster_{end}$
if $NodeID \geq start$ and $NodeID \leq end$ then
    return $cluster$
while $NodeID \leq start$ and $cluster_{prev} \neq NULL$ and $NodeID \leq cluster_{prev_{end}}$ do
    $cluster \longleftarrow cluster_{prev}$
    $start \longleftarrow cluster_{start}$
    $end \longleftarrow cluster_{end}$
    if $NodeID \geq start$ and $NodeID \leq end$ then
        return $cluster$
return $failure$
```

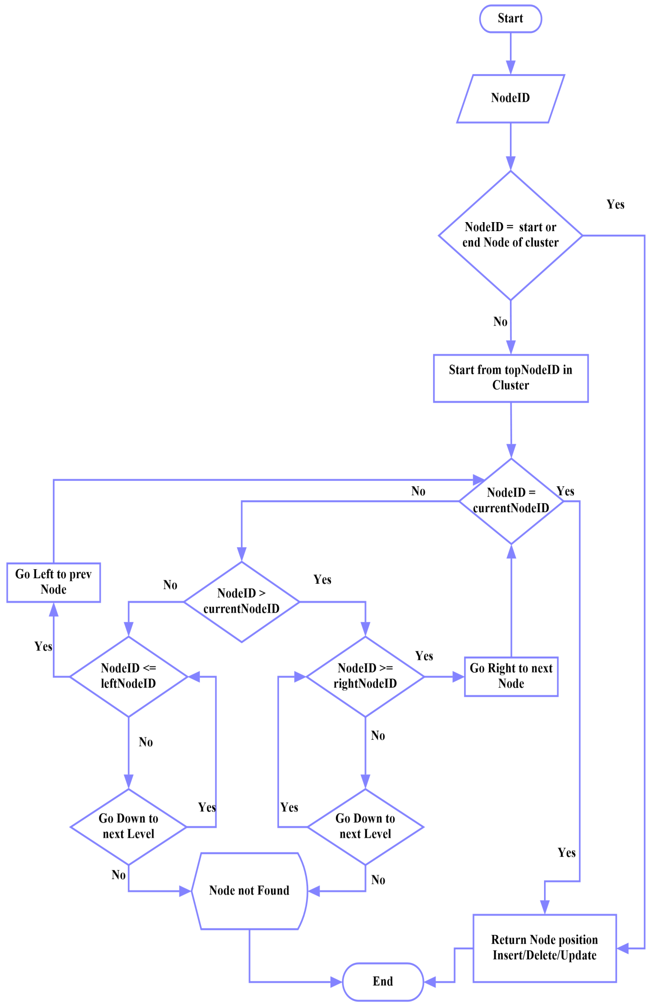

Fig. 11. The search inside the cluster Flow Chart

Algorithm 7: Find Node ***findNode*** function

```
function findNode ($NodeID, cluster$) ;
Input  : NodeID, cluster
Output: Node Data
$start \longleftarrow cluster_{start}$
$end \longleftarrow cluster_{end}$
$top \longleftarrow cluster_{top}$
if $NodeID = start|end|top$ then
    return $start|end|top$
else
    return findNodeInCluster(NodeID, top, start, end)
return $failure$   ▷ or call insertCluster function for adding a new Node
```

Algorithm 8: Find Node inside cluster ***findNodeInCluster*** function

```
function findNodeInCluster ($NodeID, top, start, end$) ;
Input  : NodeID, top, start, end
Output: Node and Node Data
$node = top$
while TRUE do
    if $NodeID < node_{key}$ then
        while $node_{down} \neq NULL$ and $node_{left_{key}}$ ¿ $node_{key}$ or $node_{left_{key}}$ ¡ $start_{key}$ do
            $node = node_{down}$;
        while $node_{left} \neq NULL$ and $node_{left_{key}} \geq start_{key}$ and $node_{left_{key}} \leq end_{key}$ do
            if $node_{key} \leq NodeID$ then
                break
            $node = node_{left}$
        if $node_{down} \neq NULL$ then
            $node = node_{down}$
        else
            break
    else
        while $node_{right} \neq NULL$ and $node_{right_{key}} \leq end_{key}$ and $node_{right_{key}} < NodeID$ do
            $node = node_{right}$
        if $node_{down} \neq NULL$ then
            $node = node_{down}$
        else
            break
return $node$;
```

Algorithm 9: CBSL insert algorithm ***InsertNewNode*** function

```
function InsertNewNode ($node$)
Input  : node
Output: Insert success or failure
$NodeID \longleftarrow node_{id}$
$cluster \longleftarrow$ insertNode(NodeID)
                                        ▷ or findClusterBySM(NodeID)
if cluster not found then
    $cluster \longleftarrow$ insertCluster($node, cluster$)
    The node will be added at start of the cluster
else
    $node \longleftarrow$ findNode(NodeID, cluster)   ▷ Find the Node or smallest node before it if not found
    The node will be added to the first level of the cluster
    The links $left$ and $right$ will be created between the node and itś neighbrs
    $up$ and $down$ links will created between the node and itself at each level
return $node$
```

Algorithm 10: CBSL Delete Algorithm ***deleteNode*** function

```
function deleteNode ($NodeID$)
Input  : NodeID
Output: Delete success or failure
$cluster \longleftarrow$ findClusterByPM(NodeID)
                                        ▷ or findClusterBySM(NodeID)
if cluster not found then
    return $failure$
else
    $node \longleftarrow$ findNode(NodeID, cluster) ▷ Find the Node or the
    smallest node before it if not found
    The links between the left and right neighbors will be created
    at each level first before the nodeś links are removed
    $up$ and $down$ links will created between the node and itself at
    each level
    The $node$ will be deleted to the first level of the $cluster$
return $success$
```

4.5 Complexity

The number of steps to find cluster by PM and SM can be represented as following:

—$SM_{cost} : O(M_{clusters})$.

—PM_{cost} : 1.

Such that SM_{cost} is the complexity of SM to find the cluster, and PM_{cost} is the complexity of SM to find the cluster.
The number of elements that must be examined can be reduced to:

—Search Operation Complexity by using SM is $O(log\ Node_{MAX}) + O(M_{clusters})$.

—Search Operation Complexity by using PM is $O(log\ Node_{MAX}) + 1$.

Clusters layer can be extended as skip list of clusters, so the complexity of finding the cluster can be computed as following:
$SLM_{cost} : O(log\ M_{clusters})$
Then the the number of elements that must be examined can be reduced to: Search Operation Complexity by using SLM is $O(log\ Node_{MAX}) + O(log\ M_{clusters})$.

5. SIMULATION RESULTS AND ANALYSIS

In this section, to evaluate the proposed CBSL algorithm, CBSL is compared with the standard skip list protocol for query delay time. The simulation parameters are shown in Table 1.

Table 1. SIMULATION PARAMETERS

Parameter	Value
Total Number of Nodes	5000, 10000
maximum Number of Nodes Per Cluster	5, 10, ..., 500

Fig 12 shows the average delay time against the the target NodeID for searching, inserting, and deleting when the number of nodes was 5000 and the maximum number of nodes per cluster was 5 nodes. As shown in Fig 6, the average delay time increases as the target NodeID increases, this is because, the existence of more nodes in the list needs more time to manage the list for inserting, searching, and deleting operations. Also, the average delay time of CBSL-SM and CBSL-PM is much better than the standard SL method. This is beacuse, CBSL uses clustering design for managment the skiplist while SL does not. In addition, PM achieves lower delay time than SM method, this is because, PM can reach to the required cluster in less number of steps by skipping some clusters.

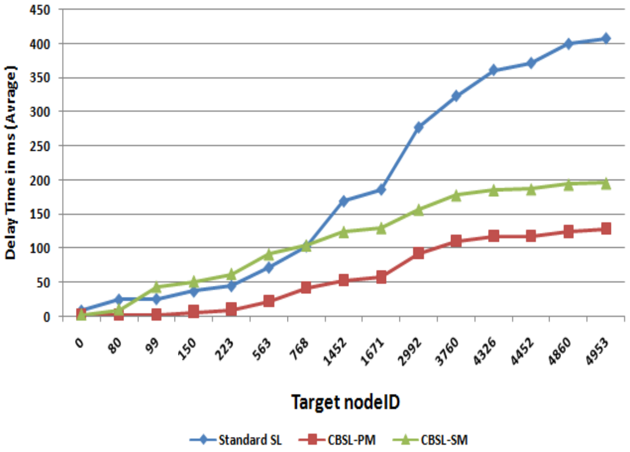

Fig. 12. Average delay time in search operation for 5000 node and 5 maximum number of nodes per cluster

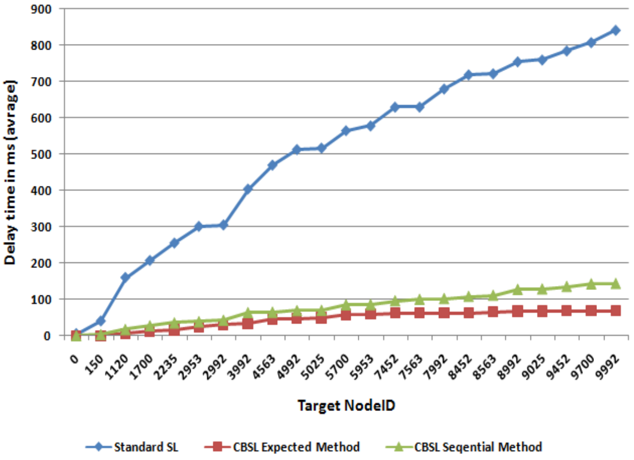

Fig. 13. Average delay time in search operation for 10000 node and 5 maximum number of nodes per cluster

Fig 13 and Fig 14 show the average delay time against the target NodeID for searching, inserting, and deleting when the number of nodes was 10000 and the maximum number of nodes per cluster was 5 and 10 nodes.
Of course the algorithms are tested with different maximum number of nodes for the clusters as shown in Fig 15 and Fig 16. as shown in Fig 15 and Fig 16, when the number of nodes for each cluster increased to be $C_{max} = 100$. The search inside the cluster by using the expected method becomes similar to the slandered SL search. Thus the expected method gives us a good result with cluster of small number of nodes. And the sequential method average

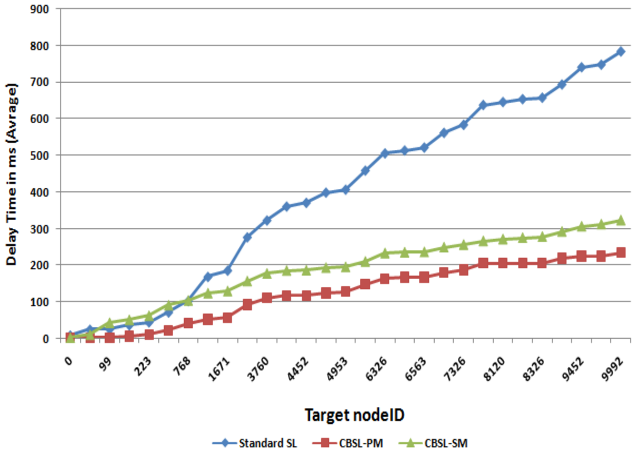

Fig. 14. Average delay time in search operation for 10000 node and 10 maximum number of nodes per cluster

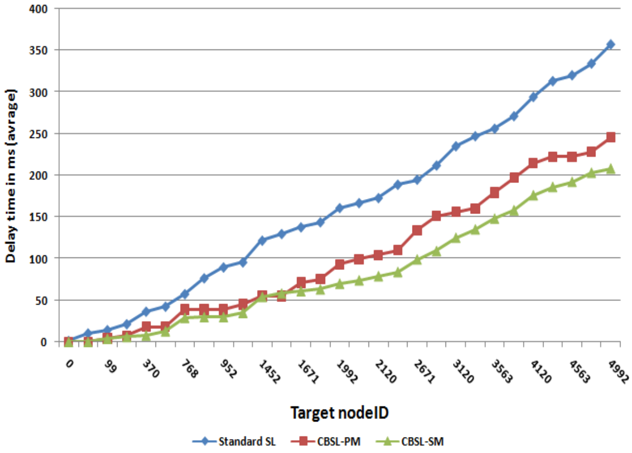

Fig. 15. Average delay time in search operation for 5000 node and 100 maximum number of nodes per cluster

delay time results become faster than the expected method average delay time, because the number of clusters was reduced.

Fig 17 show the relation between the maximum number of nodes of cluster and the total average delay time. As shown in Fig 17 the CBSL methods give us a good result for clusters of maximum length from 5 to 250. Thus when the maximum length of cluster increased after 250 the standard SL becomes better, because the search inside cluster of bigger length take more time than the clusters of small length. And this is make sense, because we loss the advantage of clustering.

6. CONCLUSION

In this paper, the most relevant data structure proto- cols types in dynamic distributed environments were introduced . In addition, a new cluster based skip list (CBSL) algorithm in dynamic distributed environment was proposed. CBSL algorithm divides the standard skip list into a group of clusters and uses two methods to

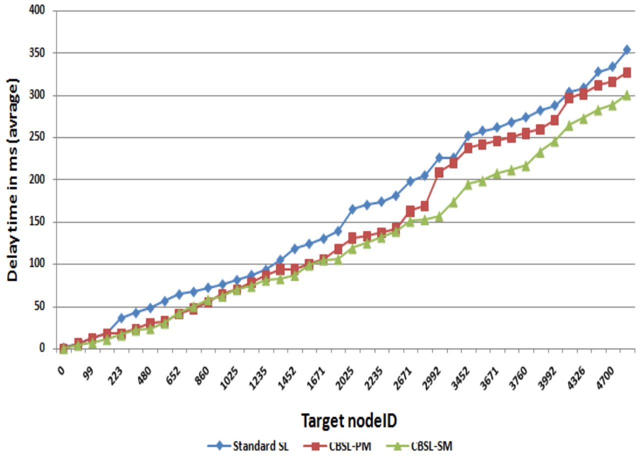

Fig. 16. Average delay time in search operation for 5000 node and 220 maximum number of nodes per cluster

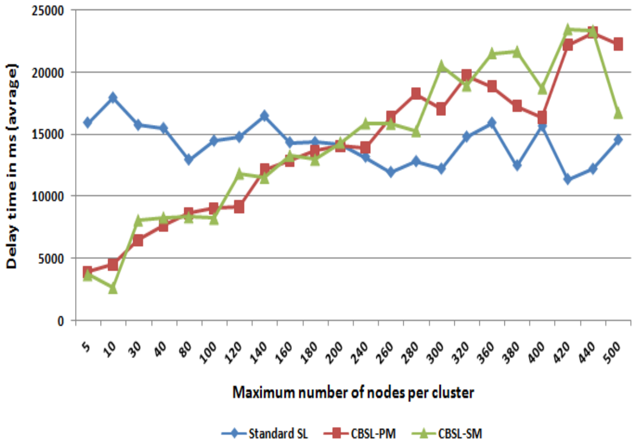

Fig. 17. Total average delay time report of diffrent cluster sizes

nd cluster position called sequential and predection methods. The performance of the new protocol has been simulated with different sizes of nodes for CBSL and a maximum number of nodes for each cluster. The simulation results show CBSL achieves average delay time for different number of nodes is better than standard SL protocol. In the future work the using of CBSL protocol is considered to develop a new adaptive stabilization protocol in dynamic distributed environments, based on m-n cluster based skip list, for data management in the P2P system and to make these environments more stable and share the information between other devices with the minimum of losing data and resources.

7. REFERENCES

[1] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, December 2004.

[2] Roberto ”Baldoni, Marin Bertier, Michel Raynal, and Sara” Tucci-Piergiovanni. ”looking for a definition of dynamic dis-

tributed systems". *"Springer Berlin Heidelberg"*, pages "1–14", "2007".

[3] Paolo Boldi and Sebastiano Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. *Springer*, pages 25–28, 2005.

[4] Thomas Clouser, Mikhail Nesterenko, and Christian Scheideler. Tiara: A self-stabilizing deterministic skip list. *Springer*, pages 124–140, 2008.

[5] Yi Cui, Baochun Li, and K. Nahrstedt. ostream: asynchronous streaming multicast in application-layer overlay networks. *IEEE Journal on Selected Areas in Communications*, 22(1):91–106, Jan 2004.

[6] Tingjian Ge and Stan Zdonik. A skip-list approach for efficiently processing forecasting queries. *Proc. VLDB Endow.*, 1(1):984–995, August 2008.

[7] Michael T Goodrich and Roberto Tamassia. Efficient authenticated dictionaries with skip lists and commutative hashing. *Google Patents*, 2000. US Patent 7,257,711.

[8] Eric N. Hanson and Theodore Johnson. The interval skip list: A data structure for finding all intervals that overlap a point. *Springer*, pages 153–164, 1992.

[9] Nicholas J.A. Harvey, John Dunagan, Mike Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. Skipnet: A scalable overlay network with practical locality properties. *microsoft*, page 38, December 2002.

[10] Y. J. Lee, S. M. Yoon, J. H. Ji, H. G. Cho, and G. Woo. How to apply the skip list to the management of a massive set of replying comments in web bulletins by exploiting the power-law distribution. *2010 10th IEEE International Conference on Computer and Information Technology*, pages 674–681, June 2010.

[11] Subhrangsu Mandal, Sandip Chakraborty, and Sushanta Karmakar. Deterministic 1–2 skip list in distributed system. *IEEE*, pages 296–301, 2012.

[12] Subhrangsu Mandal, Sandip Chakraborty, and Sushanta Karmakar. Distributed deterministic 1–2 skip list for peer-to-peer system. *Peer-to-Peer Networking and Applications*, 8(1):63–86, 2015.

[13] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic skip lists. *Society for Industrial and Applied Mathematics*, pages 367–375, 1992.

[14] Rizal Mohd Nor, Mikhail Nesterenko, and Christian Scheideler. Corona: A stabilizing deterministic message-passing skip list. *Theoretical Computer Science*, 512:119 – 129, 2013.

[15] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[16] D. Wang and J. Liu. Peer-to-peer asynchronous video streaming using skip list. *2006 IEEE International Conference on Multimedia and Expo*, pages 1397–1400, July 2006.

[17] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over dht. *The 5th International Workshop on Peer-to-Peer Systems*, 2006.

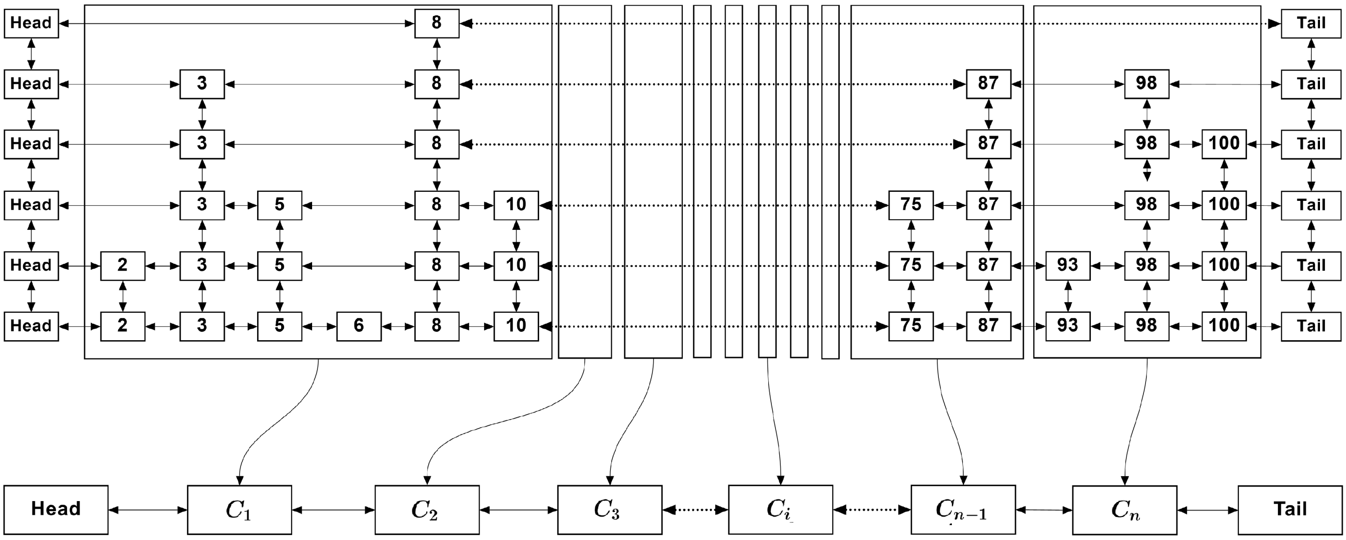

Fig. 5. CBSL Structure