A Distributed Hashing Algorithm Based on Virtual Neighborhood Search and Retrieval

Prof. Dr. Ebada Sarhan* Dr. Mohamed Belal* Mohamed Khafagy**

* Computer Science Department, Faculty of Computers & Information, Helwan University

** Computer Science Department, Faculty of Information Systems & Computer Science, 6th October University

Abstract

Dealing with a huge amount of data nowadays increase the need to distribute this data among cooperated servers in order to increase its availability and the performance of accessing and retrieving data.

Rapidly growing networks implies that future files and database system are likely to be constructed as networked clusters of Distributed nodes and algorithms should be devised to work in this environment

In this paper we describe the design and implementation of an innovated distributed algorithm using arbitrary architecture. This algorithm spreads data across multiple nodes in network with an arbitrary and varying architecture. Using novel autonomous location discovery and searching algorithm that cooperates with the other nodes to uniformly distribute the data among the neighborhood instead of using a centralized algorithm.

Performance results show that the innovated algorithm is superior to the extendable hashing Algorithm EH*[5] in the distributed environment based on several performance measurements.

1. Introduction

Hashing, a technique that mathematically converts the key into a storage address, that corresponding record in the data file offers one of the best methods of finding and retrieving information [6]. Hashing algorithm can be classified as either static or dynamic. A static hashing algorithm uses a constant sized hash table, on the other hand dynamic hashing techniques allows the storage to expand with the number of data insertions and deletions. A dynamic hashing algorithm differs from static hashing algorithm because the table can grow and shrink from its initial size according to insertion and deletion operations [1].

A few distributed hashing algorithms have been introduced. Litwin [3] Introduced LH*, an efficient, extensible, distributed version of Linear Hashing (LH) which generalize Linear Hashing to parallel or distributed RAM and disk

files LH* like LH is a directory less algorithm. It allocates buckets through one or two algorithms based on the current split-level and the bucket number. Davine [2] introduced DDH (Dynamic Distributed Hashing), an extension of the dynamic Hashing method, which was a scalable distributed Data Structure. Kroll [4] introduced a distributed Search Tree (DRT) with good storage space utilization and high query efficiency, Hilford [5] proposed a distributed Extendible Hashing EH*, buckets are spread across multiple servers, and autonomous clients can access these buckets in parallel. The EH* algorithm is scalable in the sense that it grows gracefully, one bucket at a time to a large numbers of servers.

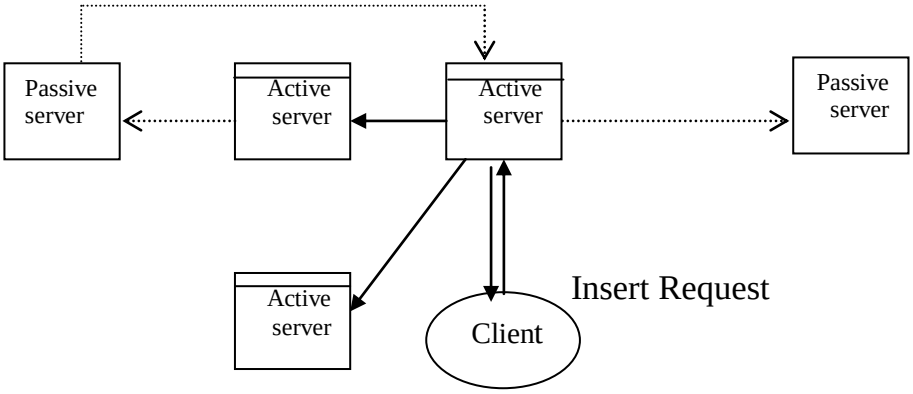
In this paper an outline of a new approach for distributed hashing technique is introduced, this technique is based on constructing a virtual distributed network of servers with virtual links to build a virtual topology of that servers that will be called VH* (**V**irtual Neighborhood Distributed **H**ashing Algorithm). Each bucket has its own number in the server and the record will be assigned to this bucket and if an overflow occurs, it is distributed among its neighborhood and if there exist no place for it, it assigns a passive server to insert the record. The same is followed for searching when we try to retrieve a bucket. This technique minimizes the splitting time to zero and bound the search time to a maximum of two searches with a slightly increase in communication complexity. The proposed algorithm will be compared with the EH* algorithm.

2. VH* Algorithm Description

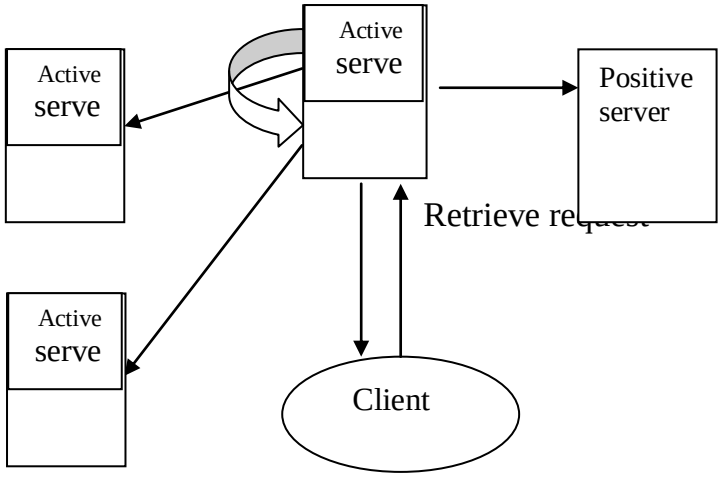
The following is the general description of the mechanism; the details of algorithm will be in the next subsection. The environment consists of clients and servers. Clients issue insert keys by INSERT REQUEST and retrieve keys by SEARCH REQUEST. A client's request is sent to server, based on dividing keys and assigns the number of server and bucket.

A server receives the client request and checks if there is available, it sends INSERT ACK, if not sent to connected server to store in the same bucket number, or if it is not available, then check the extension to store and send INSERT ACK. If the neighborhood is full and the server does not have a passive server, then the server will assign a new passive server and store the data in the same bucket number, and move all data from extension to passive server, then send INSERT ACK. If the server already has passive, it will store, the data in the passive servers of the servers that connected with it and connect this passives to the server in the case of no available bucket in these passives then the server assigns new passive server as shown in **figurer (2)**.

When client sends RETRIEVE REQUEST, the server search for the same bucket in it and in all servers (active or passive) in neighborhood and in its extension, finally it sends RETRIEVE ACK, as shown in **figurer (1)**.



A client sends an Insert Request
Figure 1



A client sends a Retrieve Request
Figure 2

2.1 Client Algorithm:

We use the same notation used in the Extendible Hashing algorithm. Thus, given a record Ri with a key Ki, the pseudo key K is generated as the following:

1- The key is changed to its binary number representation.
2- The key is divided into three parts K1, K2, and K3.
3- The first part from the right K_1 is XORed with the third part from left K3 and the result represents the server that the record will be stored or retrieved.
4- The second part in the middle is XORed with the third part (from left) mod B (B number of bucket in the server) the result represents the Bucket number K_2 that the record will be stored or retrieved.

The following algorithm generates the server and bucket addresses:

Function generate_add (***K***)

K1← right (***K***)

K_2 ← middle (***K***)

K_3 ← left (***K***)

Server ← $(P_1 \oplus P_3)$ mod S

Bucket ← $P_2 \oplus P_3$ mod B

End

Function search (k)

Generate- addr (k)

Send- message (server, message ('search', k))

End

*Function insert (k, *rec)*

Generate- addr (k)

*Send- message (server, client (ID), message ('insert, R, *rec);*

End

Function ACK- Insert (k), status)

Send- To- user (k, status)

End

*Function ACK – Search (k, status, *rec)*

*Send- To- user (k, status, *rec)*

End

The same function for all the client's requests is invoked. These requests generate k1, and then the request is send to K2 bucket for K_1 server then the client waits for an acknowledgement. If there is no response the client will resend the request and if the client receives a stored ACK then the client proceeds with its next request. Several clients can operate on this distributed file at the same time and all requests are performed concurrently.

2.2 server Algorithm:

The server starts with empty passive server list, and it waits for requests Each request can come from client or another server .The server can receive requests to insert a key (INSERT REQUEST), or retrieve a key (RETRIEVE REQUEST). The server can initialize a passive server (new passive server INIT) and it receives an acknowledgement for forward insertion, retrieve message (INSERT ACK), (RETRIEVE ACK) available posits (AVA ACK), or found key (FOUND ACK). When a client wants to retrieve a record it generates k1, K2, then, it sends a message to K1 server with K2 bucket. If it is found, it would retrieve and send record to client and send RETRIEVE ACK, But if it is not found. The server would send Retrieve Request to the connected active and passive servers, then it send FOUND ACK to server, which in turn, retrieve the record to client and send FOUND ACK. If the server has not response, the server will search in the neighborhood of the connected server

and its extension, and if it is found, it would retrieve the record and send FOUND ACK to the client.

2.2.1 In case of insert:

When a server receives a message from client, it checks the bucket that the client determined whether it is available or not:

- If it is available, it will lock for the client and store the record, then, unlock the bucket and will send INSERT ACK to client.
- If it is not available, the server will send message to the servers connected with it. Every server will has a message to check the availability of Bucket, then, lock the Bucket and send AVA ACK to the initial server.

The server chooses the first server that Replies an AVA ACK, then, stores the record in it, and sends to client to INSERT ACK and unlock all bucket. If there is no response, The server will check the Extension, if the Extension is empty the server will lock the Extension and store the record, and unlock the Extension, then, send INSERT ACK to the client.

```
Function Insert (k, *rec)
     If  AVL- bucket (B)
        Lock (B)
        Store (*rec)
        Unlock (B)
        Send- ACK ()
   Else
      Send- req (server- list)
      ACK← false
      Wait for ack ()
      If ack
        Send- rec (*rec, ACK- Server)
       Send- ACK (Client ID, k)
      Else
     Send- message (‘ overflow ‘)
     Send- ACK (Client ID, k)
End
End
Function Search (k)
     Calc- addr ()
     If avail (B)
       Rec← get- rec (B)
      Send- rec (*rec, client)
  Else
     Send- Retrieve (Server-list)
```

```
    ACK← false
    Wait for ACK (Timer)
    If ACK
      Found ←true
      Send- ACK (client)
   Else
      Send- ACK (client ID)
  End if
End
```

- If the Extension is locked or there is no space and the server has no passive server, the server will assign A passive server, then, locks it and stores the data and sends INSERT ACK to client and checks the extension bucket, if it is suitable to passive, it will move data to the passive and unlock the passive Extension.
- If the server has passive, it will check all passive servers that is connected to another server that connected with it and sends message. if it has response, it will send a message to other servers to check their passive servers whether have available space or not, then lock that passive and send AVA ACK. the server will choose the first server Reply AVA ACK, then, stores the record in it and unlocks all and sends insert ACK to client and adds passive to list, if no response. Then, the server assign new passive server.

2.2.2 In case of retrieve:

When a client wants to retrieve a record it generates k1, K2, then, it sends a message to k1 server K2 bucket, if it is found, it will be retrieved and it will send (RETRIEVE ACK) to client. If it is not found, the server would send (RETRIEVE REQUST) to the connected servers (active or passive) but if it is found, it would send (FOUND ACK) to the server, then, the server will send it to the client. If there is no response, the server will search in his extension, and if it is found, it will send (FOUND ACK) to the client. The operations that take place are executed in parallel. Multiple clients can insert and retrieve key in parallel with each other. Server executes all requests from clients or other server in a sequence. Servers define the consistency points as they send and receive message.

2.3 Passive Server Algorithm:

The passive server waits for request. Each request must come from active server The passive server can receive request by insert a key (INSERT REQUEST), or by retrieve key (RETRIEVE REQUEST) and receive an acknowledgement for insert or retrieve message (INSERT ACK) and (RETRIEVE ACK) and (AVA ACK) and (FOUND ACK). The passive server can be locked by Active server that connected with him

An Activity Diagram for Process: Insert Element is shown in figure (3)

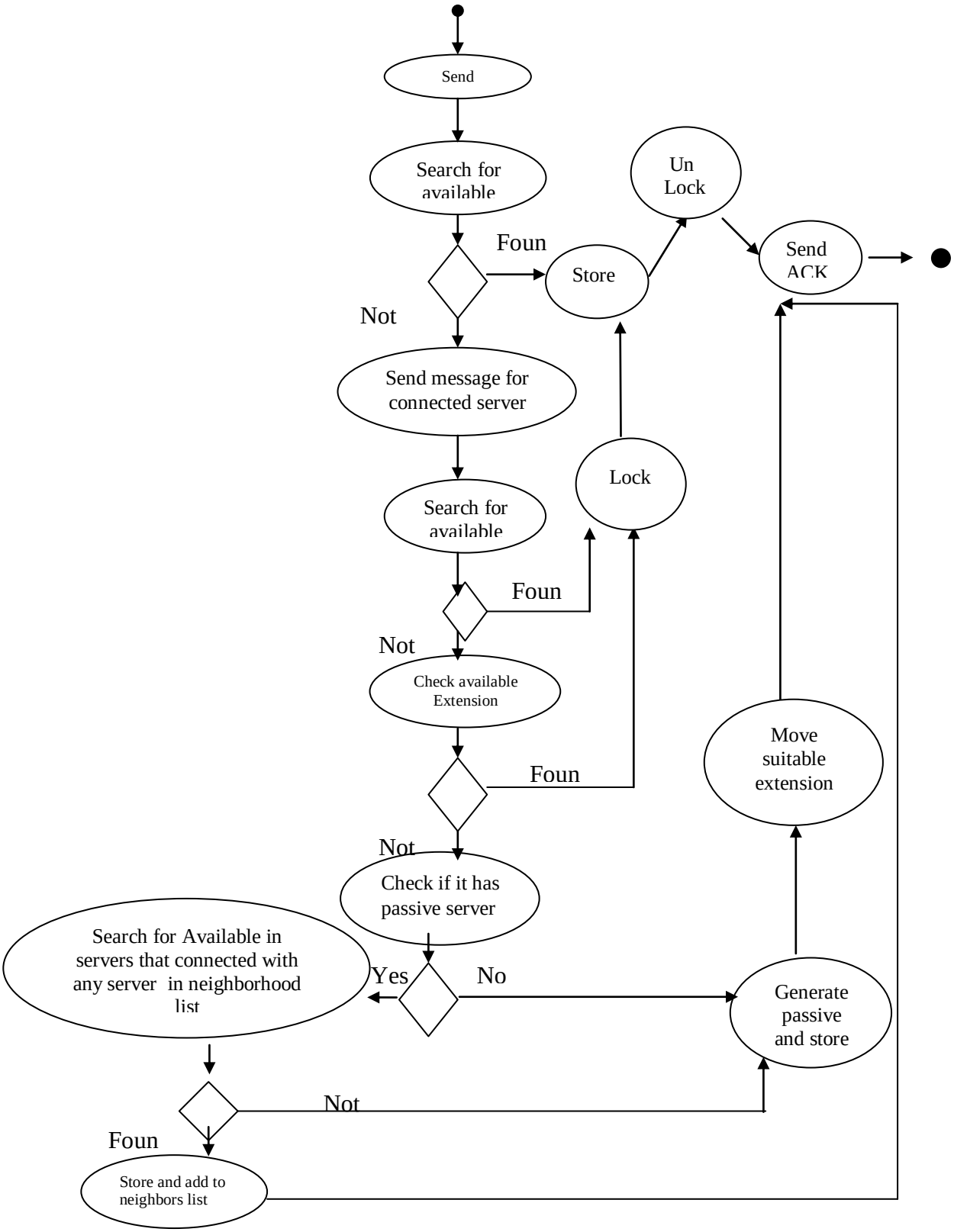


Figure (3) : Activity Diagram for Insert

An Activity Diagram for Process: retrieve element is shown in figure (4)

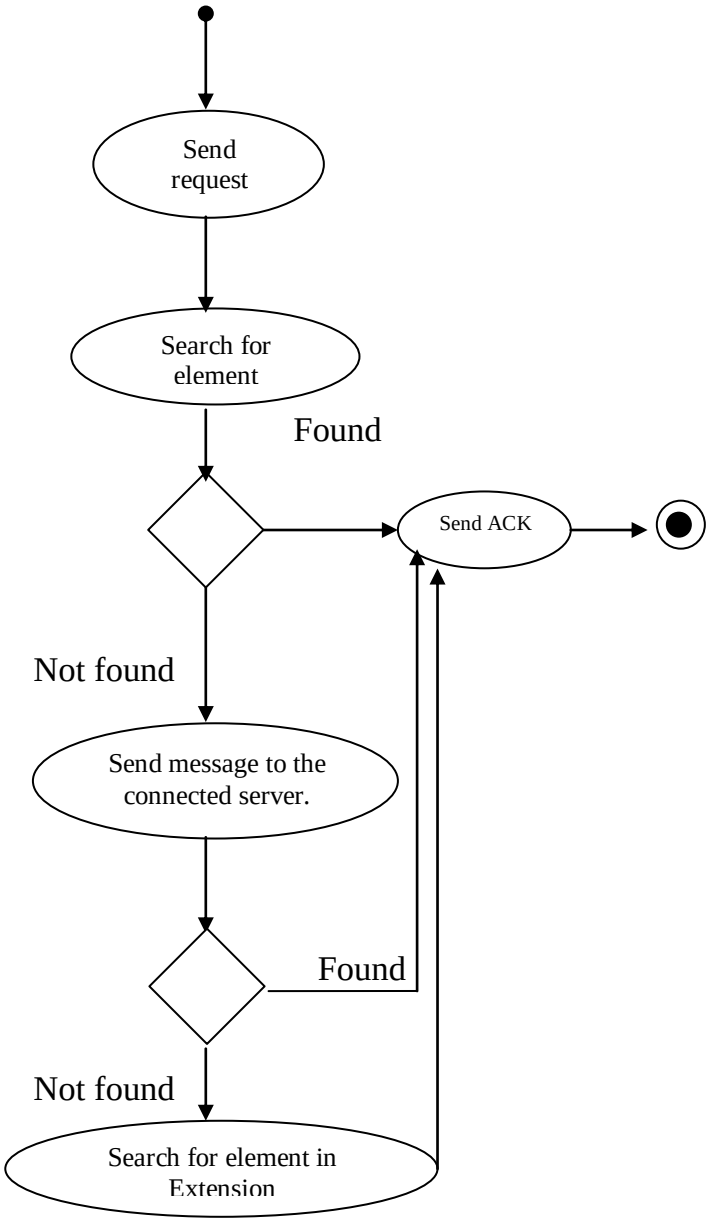


Figure (4) Activity Diagram for Search

3. Simulation and Results

In this simulation, we suppose that we have eight servers and they constitute a virtual topological network as shown in figure (5).

It should be note that this topology is not fixed and it can be generated randomly at the initial time. Knowing the basic server at first, servers can randomly determine its neighbors .

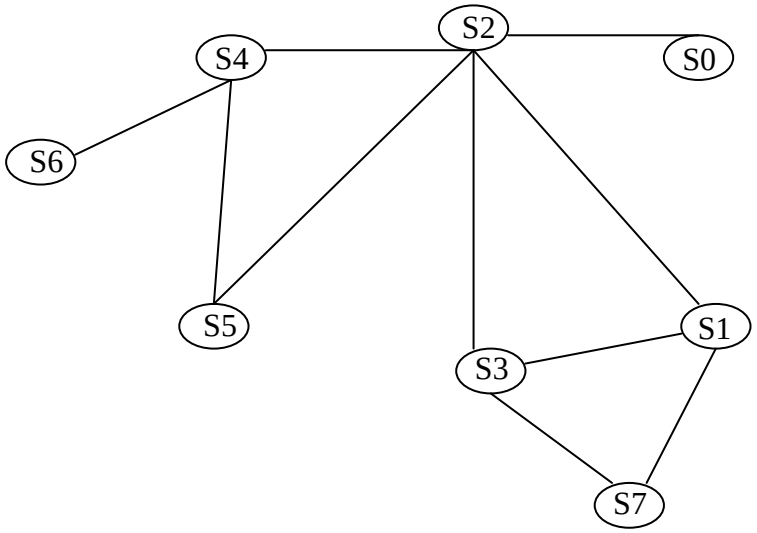


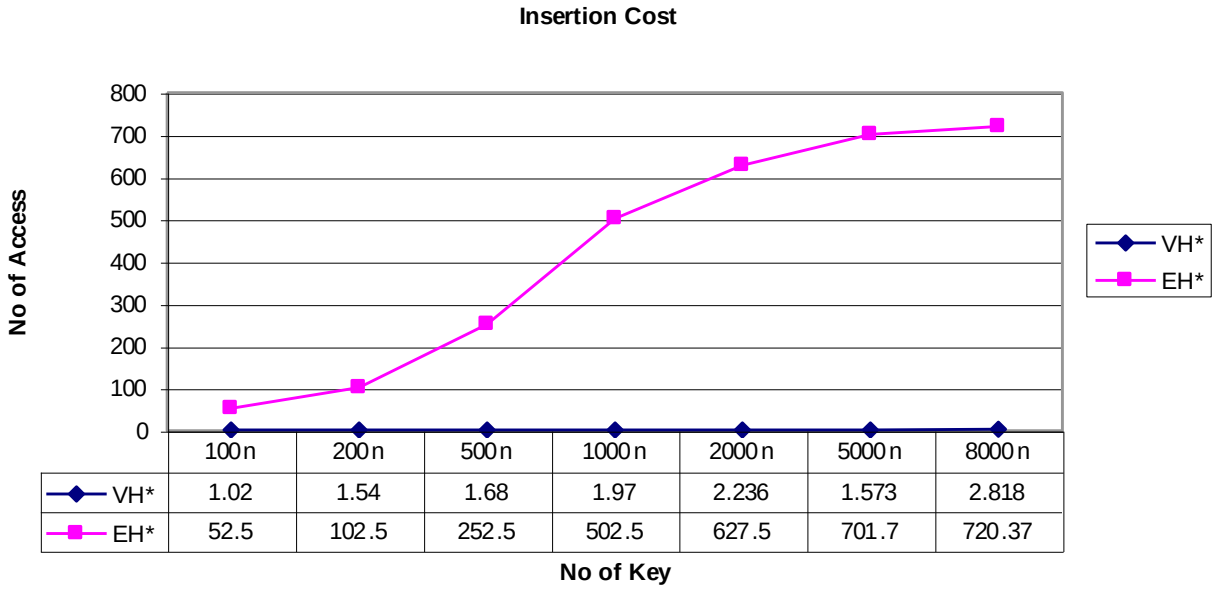
Figure 5
Model topology

In this simulation we assume the following

- No. of clients requesting insertion and retrieval is two
- No. of keys to be inserted is 7000.
- The clients can request at the same time.
- Each server has 1000 addressable buckets.
- Each server has 100 extension buckets
- A passive server could be assigned to any server whenever needs.
- The keys were generated randomly .

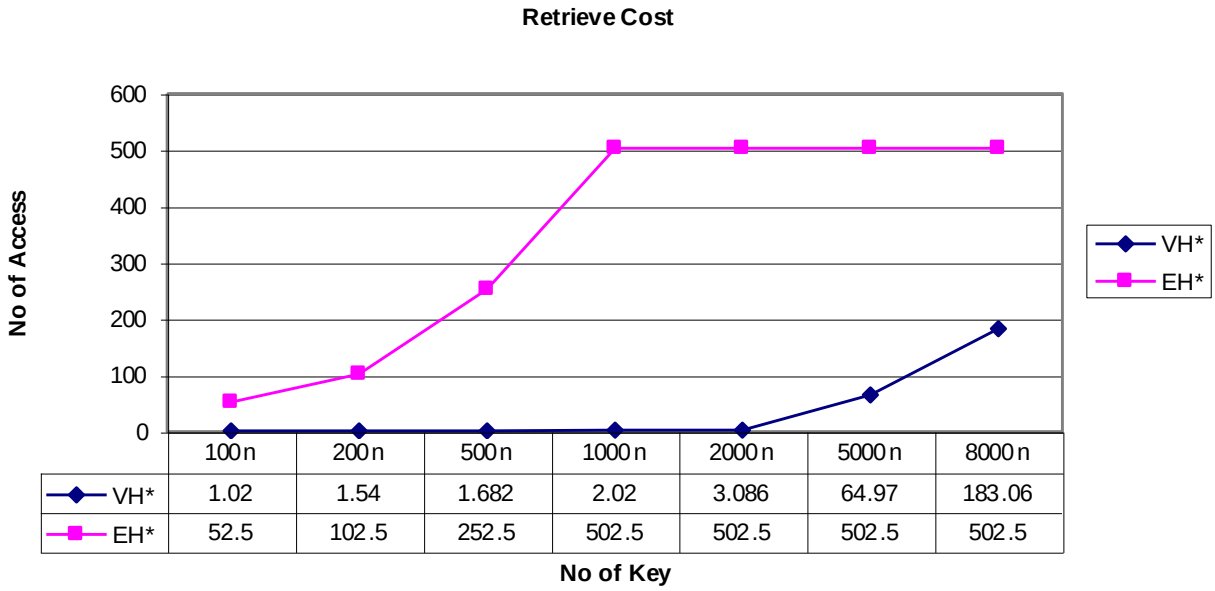
It should be note that in VH* each server doesn't need cache table.
. The performance of the proposed algorithm is compared with EH* the comparison was based on the following factors:

1. Insertion cost;
2. Retrieve cost
3. Average insertion message
4. Average retrieve message
5. Assign server cost
6. Communication cost
7. Storage utilization;
8. Splitting Cost



	100n	200n	500n	1000n	2000n	5000n	8000n
VH*	1.02	1.54	1.68	1.97	2.236	1.573	2.818
EH*	52.5	102.5	252.5	502.5	627.5	701.7	720.37

Figure 6



	100n	200n	500n	1000n	2000n	5000n	8000n
VH*	1.02	1.54	1.682	2.02	3.086	64.97	183.06
EH*	52.5	102.5	252.5	502.5	502.5	502.5	502.5

Figure 7

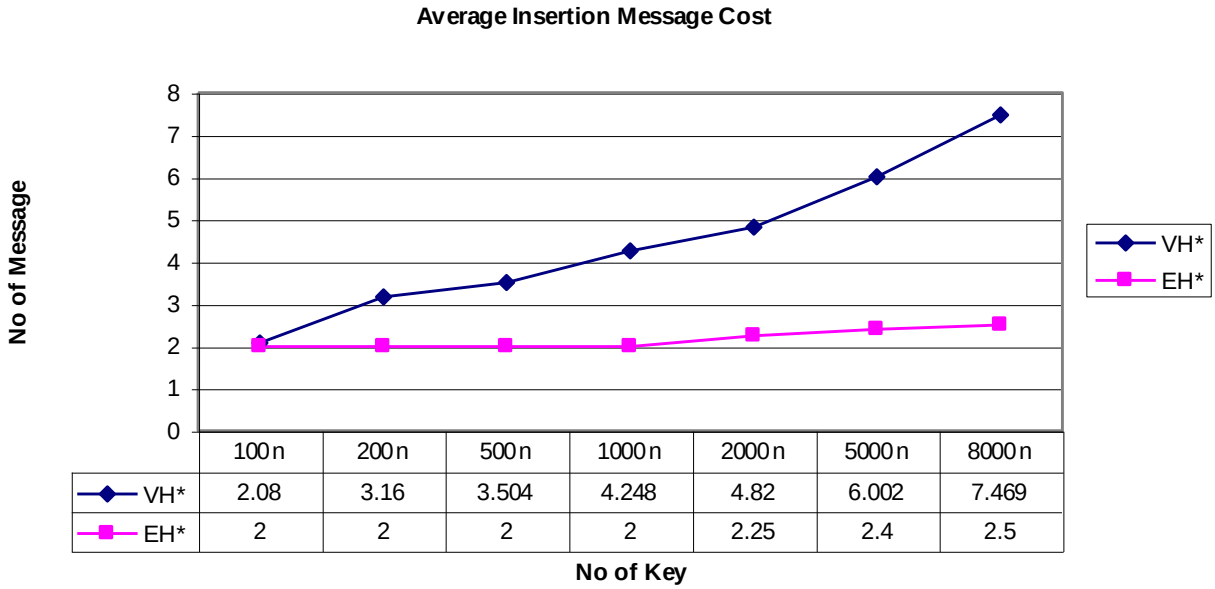


Figure 8

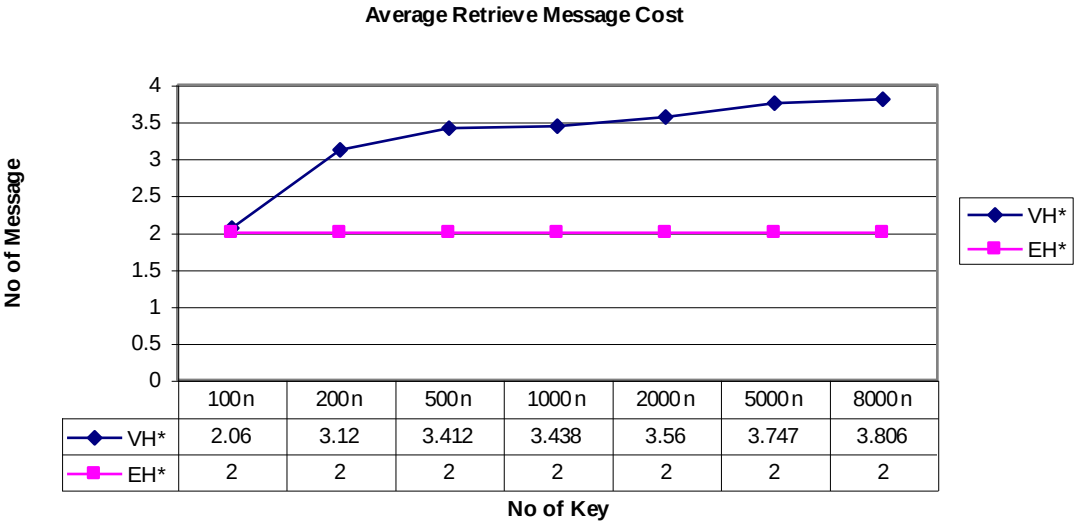


Figure 9

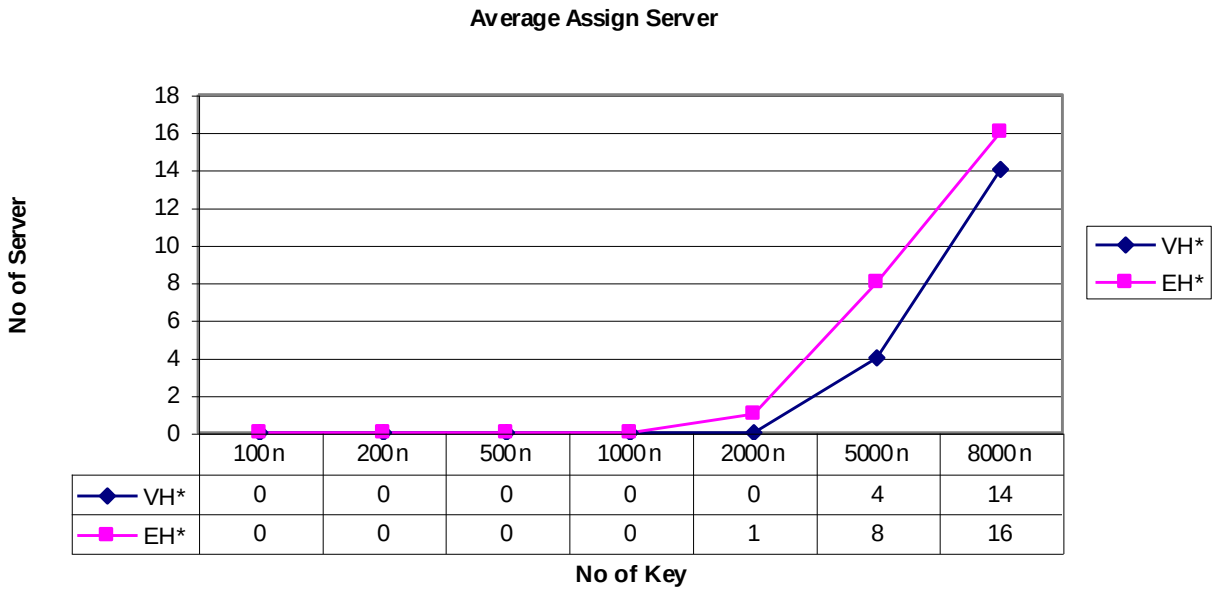


Figure 10

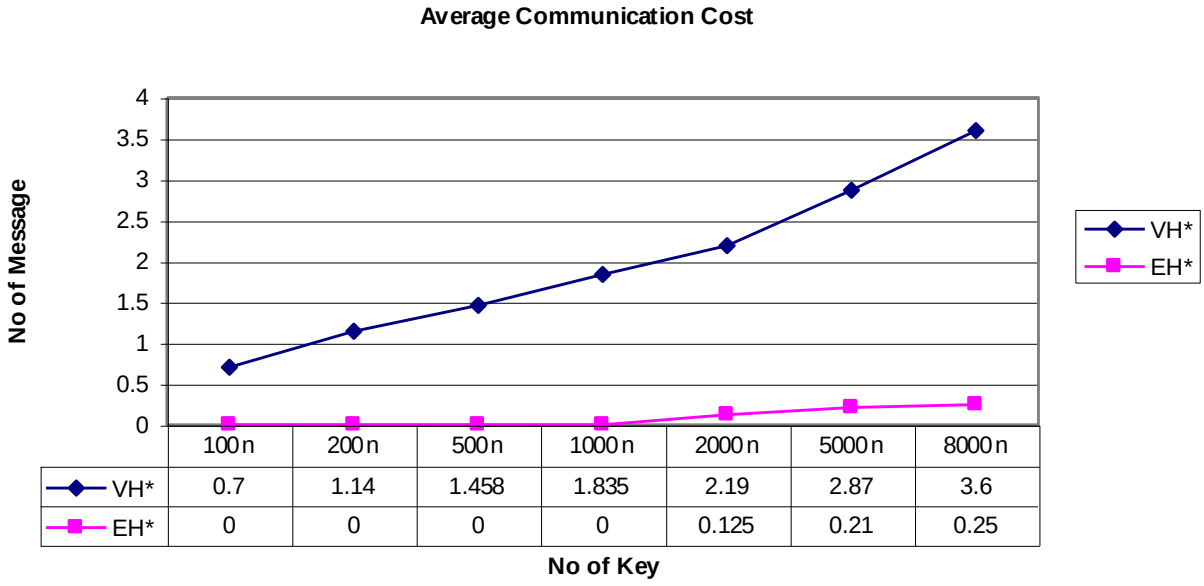


Figure 11

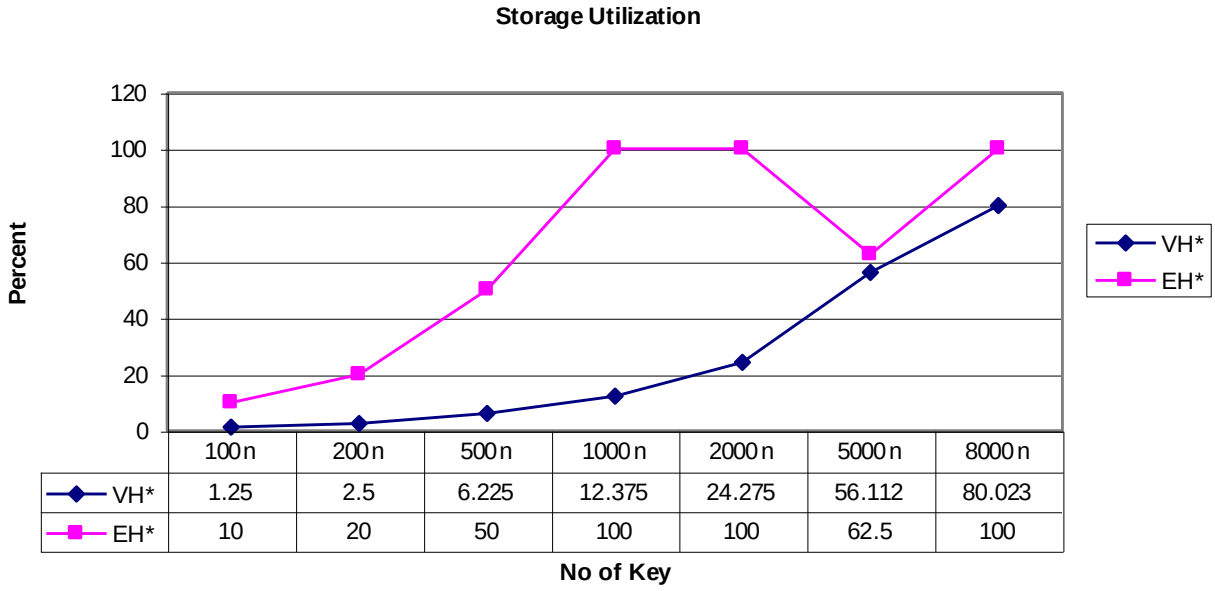


Figure 12

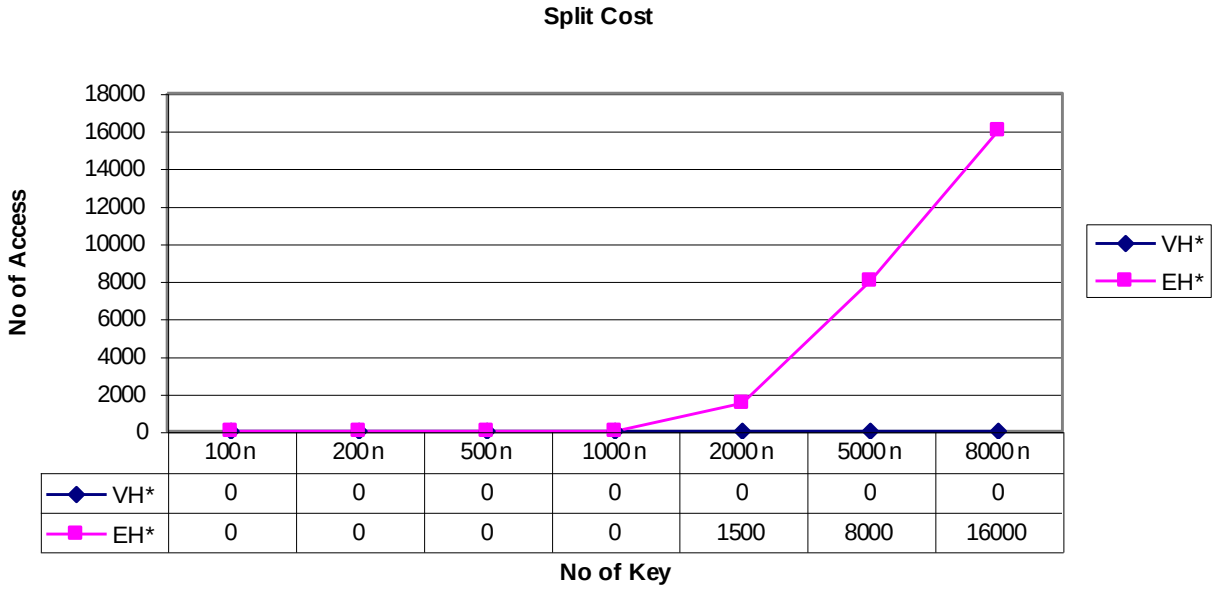


Figure 13

Figure 6 compares the insertion cost of both two algorithms, it was found that the cost of insertion for the proposed algorithm is less than that of

EH algorithm and increased gradually while the number of record increased to the value of 1.9.

On the other hand, the insertion cost for EH* algorithm increases to a value 502 at number of keys equals 1000. Comparing the retrieval cost of the two algorithms, as shown in figure 7, before insertion of 2000 records the cost was less than 3 for the proposed algorithm while the cost reaches a value of 500 for EH* algorithm. The cost of the proposed algorithm was less than that of EH* algorithm.. The cost of the proposed algorithm is almost constant because of insertion, and retrieval the average message cost for the proposed was greater than that of EH* as shown in figure 8,9. Both EH* and the proposed algorithm need to assign a server this need is comparative to each other as shown in figure 10, the average communication cost of the proposed algorithm is larger than that of EH* algorithm as shown in figure 11.

The storage utilization of the proposed system is increased linearly with the insertion of records but it changes for EH* as shown in figure 12.

It is noticed that the splitting cost equals zero for the proposed algorithm, on the other hand, the EH* is affected by the splitting cost as shown in figure 13.

4. Conclusion

In this paper, an outline of a new distributed hashing algorithm is introduced. This algorithm is based on building a virtual topology for the servers in order to create a local neighborhood for each server to search and to insert data.. This locality minimizes the insertion and retrieval cost and eliminates the need to split data. Comparison results show that the cost of insertion and retrieval of the proposed algorithm is less than that of the EH* algorithm , with a small increase in communication complexity .

VH* is an efficient, scalable, and distributed algorithm. It provides a new method to be used in applications such as next generation database

References

[1] Thomas R.Harbon, ”File Systems Structures and Algorithms”,1988.

[2] R.Devine, Design and Implementation of DDH, A Distributed Dynamic Hashing Algorithm, In conf. on foundations of Data Organization and Algorithms,1993, pp.101-114

[3] W.Litwin , M-A Neihnat, and D.Schneider, LH*; Linear Hashing for distributed files , In ACM-SIGMOSD InTL.conf.on management of data, pp327-336. 1993.

[4] Brigitte Kroll, Peter Widmoayar, “ Distributing a search tree among a growing number of process” In ACM- Sigmod Intl. Conf. on management of data pp 265-276,1994..

[5] Victoria Hilford , Farokh B.Bastani and Bosan Cukis, “EH*- Extendible Hashing in Distributed Environment “ IEEE 1997.

[6] Ramez El Masri and Shamkant B. Navathe “ Fundamentals of Data Base Systems “, 2000 p 217-222.