**LH\* Sever Side Addressing Algorithm—Class Notes**

**Basic Terminology and Setting:**

* The file is using a family of hash functions
* Here i = 0, 1, 2, 3… And M can be any integer. In this document (and the slides presented in class) M is set to 1 for sake of simplicity.
* Each bucket is located in a different server.
* Globally, the file is currently using the certain values for N, which is the next bucket to be split and, I, which is the value of i used to pick the hash functions from the family
* Each bucket (a server) holds its correct value of level denoted as J, which could either be I or I+1. Note that J is not allowed to hold any other value. This comes from the fact that all the splits are coordinated by a splitting coordinator which executes all the splits in a serial fashion.
* Each client also has its view of the file through N’ and I’. Note that I’ <= I (the client image readjustment algorithm ensures that)

**Addressing at the Server Side**

A server with an address A at level J receives a message for a data manipulation operation (search, insert or delete) with a key k. This request could have either come from a client or from another server. Nevertheless, the following script is executed at the server end.

A’ =

If A’ != A Then // Key should not have come here

A’’ =

If A’’ > A and A’’ < A’ then A’ = A’’

Forward the message to server with address A’

End IF

**Figure 1: Server Side Algorithm**

**Top level ideas used by the Server:**

Whenever a server receives a request for key k it is not supposed to, i.e., A’ != A, then it chooses between the following two options:

**Option 1:** Forward it to server with address

**Option 2:** Forward it to server with address

**Why not somewhere else? Think from the perspective of value of J and the global view of the file.**

As mentioned previously J can be either I or I+1. It cannot take any other value. So ideally, we could just use, which would be like rehashing using the correct hash function irrespective of what the client image looked like. And in this case we would just make one more forward and land up in the correct server.

However, this is not always a reliable idea. Consider a case when a message wrongly lands on a bucket at level I+1 instead of level I, in this case we rehash using , then we would land on a server which does not exist. Thus, in this case we use because we are certain that this server address exists (Why??). This scenario is illustrated on slide 38 in w1p2.pptx

**So how does the server know when to use and when to use ?**

A server is actually not very sure about it because it does not the value of N (the next bucket to split). Recall that in basic Linear Hashing, the value of N used to guide us choosing the correct hash function. But we don’t have that facility over here. So, the server basically makes a conservative estimate, i.e., it does not use unless it is absolutely very certain that the bucket corresponding to exists. At the slightest doubt, the message is sent to **.**

**So what if the server always keep making the conservative choice and never end up on the correct server address?**

The second if condition ( If A’’ > A and A’’ < A’ then A’ = A’’ ) of the server algorithm shown in Figure 1 prevents that from happening more twice. This scenario is illustrated on slides 39 – 41 in w1p2.pptx.

**Focus on the If condition (If A’’ > A and A’’ < A’). Lets study the cases when it can break. In all such cases the message is sent to**

**Scenario 1 (A’’ == A):** Consider A’’ which uses **.** If A’’ = A then it means the client was at level J-1 or less. The if condition (If A’’ > A and A’’ < A’) breaks and following two scenarios can take place.

**Case (a) if the server was at J=I+1, then client was at I or less.**

**Case (b) if the server was at J=I then the client was at I-1 or less.**

**Case (a):** In this case we need to careful so as to not to send to a bucket which does not exist. This can happen as all the buckets at level I may not have been split yet. So all the buckets corresponding to **k Mod 2I+1 M** would not exist. Now consider the relation between A’ (which is the result of)and A’’(which is the result of ). From the properties of the family of hash functions used in LH\* (or LH for that matter), we know that A’ >= A’’. If A’ happens to be same as A’’, it implicitly means that  **and** would hash the key k to the same address, i.e., the bigger hash function has no different effect on the key under consideration. And given that we already have all the buckets (servers) corresponding to the values of the function **k Mod 2I M,** it is safe to send it to result of as it is guaranteed to exist.

Now consider the case of A’ > A’’, this is implying that (A’)wants the key to be sent to a farther place than (A’’). There is good chance that this A’’ has not be split yet. However, note that from A’’ being equal to A, we know that client (or for that matter the server that forwarded the request) was mapping this key to level I, which now turns out to be using I+1. But from the properties of family of hash functions used to LH\*, we know that when a bucket at level I is split and rehashed using **k Mod 2I+1M,** it would either go to the old bucket or the new which is created after split and nowhere else. Now given that current bucket is already at I+1, we must be having the bucket created after its split as well. Therefore, it is safe to send it to the result of **.**

**Case (b):** This is an easy case. We know that the system would have all the buckets corresponding to the hash values **k Mod 2IM.** Thus it is safe to send it to the result of **.**

**Scenario 2 (A’’ < A):** In case A’’ < A, then it implies that client is at level I. Note that client cannot reach the level I+1 due to the conservative nature of the client image re-adjustment nature of the algorithm. The if condition (If A’’ > A and A’’ < A’) breaks and following two scenarios can take place.

**Case (a): It has the correct value of N (next bucket to split).**

**Case (b): It does not have the correct value of N (next to bucket to split).**

**Case (a):** Had this been the case, client would have automatically the sent to request to the correct address in the first attempt and there would have no need for re-addressing.

**Case (b):** If client has an incorrect value of N’ (its view of global N), it can send it to a wrong bucket. It is important to remember that a client’s N’ <= N if its I’ == I.

Thus the client can send it to a server with level I+1 thinking that it is still at level I. However, the server can simply forward it using **k Mod 2I+1M (its ) .** This server is guaranteed to exist as all buckets which are at level I+1 have been split already. Note that when a bucket at level I is split and rehashed using **k Mod 2I+1M,** it would either go to the old bucket or the new which is created after split and nowhere else.

**Scenario 3 (A’ == A’’):** A’’ uses and A’ uses **.** From the property of family of hash functions used in LH\*, we know that A’’ <= A’. In case A’ == A’’ we have two scenarios irrespective of where the client (or the server that forwarded the request).

**Case (a) if the server was at J=I**

**Case (b) if the server was at J=I+1**

**Case (a):** This is an easy case. We know that any stage of the development of LH\*, we would always have all the buckets (servers) corresponding to the values of the function **k Mod 2I M (its ).** Therefore we can be certain that result of exists in the pool of servers.

**Case (b):**  If A’ happens to be same as A’’, it implicitly means that  **and** would hash the key k to the same address, i.e., the bigger hash function has no different effect on the key under consideration. And given that we already have all the buckets (servers) corresponding to the values of the function **k Mod 2I M,** it is safe to send it to result of as it is guaranteed to exist.