2013 25th International Symposium on Computer Architecture and High Performance Computing

Fast LH$*$

Juan Chabkinian
Universidad Católica del Uruguay
Montevideo, Uruguay
juanjose.chabkinian@gmail.com

Thomas Schwarz, S.J.
Universidad Católica del Uruguay
Montevideo, Uruguay
TSchwarz@ucu.edu.uy

***Abstract*— Linear Hashing is a widely used and efficient version of extensible hashing. A distributed version of Linear Hashing is LH$*$ that stores key-indexed records on up to hundreds of thousands of sites in a distributed system. LH$*$ implements the dictionary data structure efficiently since it does not use a central component for the key-based operations of insertion, deletion, actualization, and retrieval and for the scan operation. LH$*$ allows a client or a server to commit an addressing error by sending a request to the wrong server. In this case, the server forwards to the correct server directly or in one more forward operation. We discuss here methods to avoid the double forward, which is rare but might breach quality of service guarantees. We compare our methods with LH$*$ P2P that pushes information about changes in the file structure to clients, whether they are active or not.**

***Index Terms*—Scalable Distributed Data Structure; Linear Hashing; LH*; Cloud Computing**

I. Introduction

Cloud computing has brought distributed computing to the masses. It has also rekindled an interest in scalable distributed data structures and algorithms. Among scalable distributed data structures, LH$*$, the distributed version of linear hashing offers fast access to records distributed over potentially hundreds of thousands of (virtual) servers [7]. LH$*_{\mathrm{RS}}$ provides the same capability, but stores data in redundant form, in order to provide scalable high availability, where the number of down servers without loosing access to the data increases as the LH$*_{\mathrm{RS}}$-file becomes larger [6].

LH$*$ is a scalable, distributed data structure. It adjusts the number of servers to the number of records. One of its most compelling feature is the absence of a central component that allows a client to find the address of a record. Instead, clients store a *view* of the LH$*$ file, a table, which allows the client to find the current location of a record based on its record identifier. A file grows and shrinks by *splitting* and *merging* buckets. Such an operation outdates the view of all clients. Instead of pushing this information to all clients or even to other servers, LH$*$ allows clients and servers to have out-of-date views of the file state. In this case, clients can commit *addressing errors* and the server that receives an erroneous request might not be able to forward directly to the correct server. However, the structure of LH$*$ ensures that a request reaches the correct server in at most two forwards and guarantees that no client commits the same addressing error twice. It turns out that most requests go directly to the correct server, some need one additional forward and that double forwards are usually quite rare. Nevertheless, the additional transit time for a double forward might cause havoc with cloud Quality of Service (QoS) guarantees.

We present here ways to minimize the number of forwards and especially the number of double forwards. In contrast to the doctorate work by Yakouben [10], [13] where split / merge information is pushed to clients in groups, we investigate here methods that prioritize updating servers. As it turns out, relatively small changes to the LH$*$ algorithms are sufficient to make double forwards very rare whie not adding noticeably to the load of servers.

The rest of the article is organized as follows: We review shortly relevant work in Section 2. Section 3 briefly reviews the most important aspects of LH$*$ without going into details that are not necessary for the understanding of our contributions here. We then present our mechanism to minimize the need for message forwarding by avoiding clients requesting service from a server that does not have the relevant record in Section 5. The fifth section reports on our experimental results. We then conclude and lay out avenues for future work.

II. Related Work

Litwin introduced Linear Hashing in 1980 [4] and (with Neimat and Schneider) LH$*$ in 1993 [7], [9]. This is not the only distributed data structure based on hashing, among the proposals, DDH [1] and Extendible Hashing [2] stand out. Other scalable distributed data structures exists such as RP$*$ [8], distributed k-d trees [11], and distributed search trees [3], to name a few. LH$*$ has seen been expanded to a variety of schemes providing availability. An overview is given by [5].

LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ provides the same capability by pushing file state information to clients [10], [13]. This method is preferably to ours if the set of active clients is stable since it avoids all double forwards for clients of long duration. However, we argue that in the typical cloud structure, clients come and go and exist only for relatively short times, during which they *might* make lots of accesses. LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ occurs a big message overhead under these circumstances.

III. LH$*$

Linear Hashing (LH) is a widely adopted form of extensible hashing. It stores records in buckets, whose number automatically adjusts to the total number of records. The number of buckets determines the *file state*. A simple *address calculation* based on the file state determines the location of the record from its record identifier. The LH-file maintains a *load factor* defined to be the average number of records in a bucket. If the load factor becomes higher than a certain preset value or if a bucket overflows (by containing to many records), a new bucket is created through a *split* from another bucket. In contrast to other forms of dynamic hashing such as Fagin's extensible hashing [2] or Devine's DDH [1], the bucket to be split is not necessarily the one that caused the overflow. Buckets are numbered starting with 0 and split in a fixed order 0, 0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7, ... The design of LH results in slightly worse storage utilization than with extensible hashing, but also in considerably faster record lookups [12] because no central directory structure is needed. If there is an underflowing bucket or if the storage utilization is too small, then a *merge* operation undoes the last split operation.

A. General File Structure

The distributed version of LH, LH$*$ stores records that consist of a record identifier and the contents. The latter can be structured according to the needs of an application. In contrast to LH, LH$*$ stores the buckets in different servers. They tend to be much larger (several hundreds of MB versus three or four records). As in LH, the LH$*$ file uses split and merge operations to adjust the number of buckets to the file size.

While LH$*$ can start with any number of initial buckets, we discuss here the simpler variant with only one initial bucket. In this case, the *file state* consists of two integer parameters, the *file level* i and the *split pointer* s. The total number N of buckets is always $N = 2^i + s$. Reversely, a number N of buckets determines the level as $i = \lfloor \log_2(N) \rfloor$ and the split pointer as $s = N - 2^i$.

LH* supports the record based operations of *insert*, *delete*, *read* as well as global operations of scanning – looking for all records with a certain pattern in their contents and function shipping, as long as the functions shipped use only a single record. The later capability was a precursor of the map-reduce scheme.

B. Addressing

Each record is identified by a unique Record IDentifier (RID). Given a key c treated as an integer, the bucket number a where the record resides is given by the LH-addressing algorithm:

$$\textbf{(LH)} \qquad a := h_i(c); \qquad \textbf{if } a < s \textbf{ then } \ a := h_{i+1}(c)$$

with hash functions h_j defined by $h_j(c) = c \mod 2^j$ and where s and i constitute the file state.

Each client uses its version of the file state, called its *image*. This file state can be identical to the actual state of the LH* file, but it can also be outmoded. In this case, many address calculations will still succeed, as we will see. However, the client can also send a request to the wrong bucket. Any bucket needs to check whether it has the record and if necessary calculate the bucket, where the record should be. This calculate could also be based on out-of-date information, but LH$*$ can be shown to find the correct address in at most two forwarding operations. If a client has made an addressing mistake and sent the request for a record to an incorrect bucket, the bucket that has the record will eventually send the answer together with an *Image Adjustment Message* (IAM) that updates the image of the client.

```
def check(self, key):
   aprime = h(self.j, key)
   if not aprime == a:
      atwoprime := h(self.j-1, key)
      if atwoprime > a and atwoprime < aprime:
         aprime = atwoprime
   return aprime
```

Fig. 1: Competency check executed when a server with address a receives a message with key `key`.

```
def updIm(self, j, a):
   self.i = j-1
   self.s = a+1
   if n >= 2**self.i:
      self.n = 0
      self.i = self.i+1
```

Fig. 2: Image update algorithm for the file state image at a client that receives an IAM messages with bucket level j and bucket address a.

C. Bucket File State

Each bucket maintains a single value, called its *level* j, in order to reroute misdirected requests from clients. The bucket level counts the number of times a bucket has been split, starting with the level of the bucket from which it has split. As we will see, this implies that $j = i$ or $j = i + 1$. (Recall that i is the file level).

If a bucket receives a request from a client for a record with key `key`, it executes the algorithm in Figure 1 to determine whether the bucket itself should have the record or whether the request should be forwarded to another bucket [7]. In this algoritm, self.j is the level of the bucket and h is the hash function taking its level as the first parameter. The bucket address is a.

D. Client Image Adjustment

A client maintains a vew of the file state (i', n'). If the client makes an address mistake and the requested record is found at bucket a, then bucket a sends an IAM with its level and address. The client executes the file state image adjustment algorithm given in Figure 2.

Knowing about the existence of a bucket does not imply knowing how to send a request to it. The original work on LH$*$ assumes that clients use a mechanism like DNS to resolve bucket numbers to IP addresses. Our variants of LH$*$ actually allow servers to include the information in their Image Adjustment Messages (IAM) to clients. Unfortunately, space reasons prevent us from presenting them. These mechanisms are also important to avoid a problem in LH$*_{\text{RS}}$ where a server failure and bucket reconstruction allows a bucket to reappear at a different location.

```
def stateUpdate(self):
   self.split = self.split + 1
   if self.split >= 2**self.level:
      self.split = 0
      self.level = self.level + 1
```

Fig. 3: The global file state update after a split represents the additional bucket.

File State	**Bucket States** j
$i = 0, s = 0$:	B0: 0
$i = 1, s = 0$:	B0: 1, B1: 1
$i = 1, s = 1$:	B0: 2, B1: 1, B2: 2
$i = 2, s = 0$:	B0: 2, B1: 2, B2: 2, B3: 2
$i = 2, s = 1$:	B0: 3, B1: 2, B2: 2, B3: 2, B4: 3
$i = 2, s = 2$:	B0: 3, B1: 3, B2: 2, B3: 2, B4: 3, B5: 3
$i = 2, s = 3$:	B0: 3, B1: 3, B3: 2, B3: 2, B4: 3, B5: 3, B6: 3

Fig. 4: Development of a growing LH* file

```
def preDecessor(fileState):
   if fileState.split == 0:
      prevLevel = fileState.level-1
      prevSplit = 2**prevLevel-1
   else:
      prevSplit = fileState.split-1
   return prevSplit
```

Fig. 5: Algorithm to calculate the address of the predecessor, i.e. the bucket that was split in order to generate the current bucket.

E. Splitting Buckets

When a bucket overflows, it informs the split coordinator who causes the bucket pointed to by the split pointer to split. Let the current file state have level i and split pointer s. The coordinator creates a bucket $2^i + s$ with level $i+1$. It applies the hash function h_{j+1} (defined by $h_{j+1}(x) = x \mod 2^{j+1}$) to all the records in bucket s. Since the keys of these records have the same value modulo 2^j and are assumed to be uniformly distributed, about half of the records in bucket s now belong to bucket $2^i + s$ and are therefore moved there. Afterwards the global file state is updated according to the algorithm presented in Figure 3.

The resulting sequence of splits is 0, then 0, 1, then 0, 1, 2, 3, then 0, 1, 2, ... 7 and so on, running through all bucket numbers from 0 (included) until $2^i - 1$ where i is the current level.

F. Merging Buckets

When a buckets underflows and reports this to the split coordinator, then the coordinator will start a merge operation. A merge operation always merges the bucket with the largest bucket number with the bucket from which it was last split. To be more precise, assume that the current file state has level i and split pointer s and accordingly $N = 2^i + s$ buckets. The coordinator sends a message to bucket $N-1$ telling it to merge with its direct ancestor. We calculate the addres of the direct ancestor by “rolling back” the file state changes by a split operation moving the system from $N-1$ to N buckets with Algorithm 4 given in Figure 5.

The critical step in this algorithm treats the case where the change from $N-1$ to N has changed the level. In this case, the split pointer is now 0 and we need to reset the level and calculate the previous split pointer.

If buckets have vanished through merging, a client can wrongly send a request to the corresponding server. In LH$*$, the server just sends an error message to the client who decrements the split pointer and recalculates the address. Under many circumstances, an LH$*$ file does not shrink in the long run and the merge operation becomes unnecessary.

G. Example

We illustrate the development of an LH$*$ file from one to six buckets in Figure 4. Assume a client with an out-of-date file state of $i = 0$ and $s = 0$ that wants to retrieve record with ID $c = 101000101_b$ in the last state. The client uses the hash function h_0 which always returns 0 and accordingly sends the request to Bucket 0. There, the bucket checks the request using the algorithm in Figure 1. Since the bucket has $j = 3$, it applies h_3 to obtain $a' = 101_b = 5$. Accordingly, it calculates $a'' = h_2(c) = 1$ and returns 1. Therefore, the request is send to Bucket 1. That bucket has $j = 3$, sets $a' = 5$ and $a'' = 1$, but returns $a' = 5$. This is the correct bucket. The bucket sends an Image Adjustment Message to the client that changes its state to $s = 2$ and $i = 2$.

This example illustrates how LH$*$ uses local knowledge. The client only “knew” of Bucket 0. Bucket 0 knew when it was last split (in order to create Bucket 4, but did not know whether Bucket 1 was already split in order to generate Bucket 5. Therefore, it only forwarded to Bucket 1. Bucket 1 however knew about Bucket 5 because it was generated by splitting it.

IV. Surpressing Forwarding in LH$*$

Despite its mathematical elegance, LH$*$ can be improved to limit the number of forwarding messages. Since double forwards incur an additional network delay and are already quite rare, we are especially interested in avoiding them. The key towards this goal is to give more information on the file state to the buckets that as we recall only maintain the bucket file state level j. Instead, we use a complete image of the file state that corresponds to the current number of buckets in the system. We then let buckets update this information through messages and not only when buckets are split or merged.

A. LH* Forwarding

A client that only knows Bucket 0 will commit an addressing error with its first request for a record located in another bucket. We call this a *compulsory* forward. As the client interacts with the system, it receives an IAM with every addressing error. If the system is stationary (without split or merge operations), the clients will eventually learn the true file state. The number of IAM necessary depends on the luck of the client and the size of the system.

Figure 6 gives the result of a simulation of a system with one thousand clients and various numbers of buckets. We simulated a total of one million messages sent by random

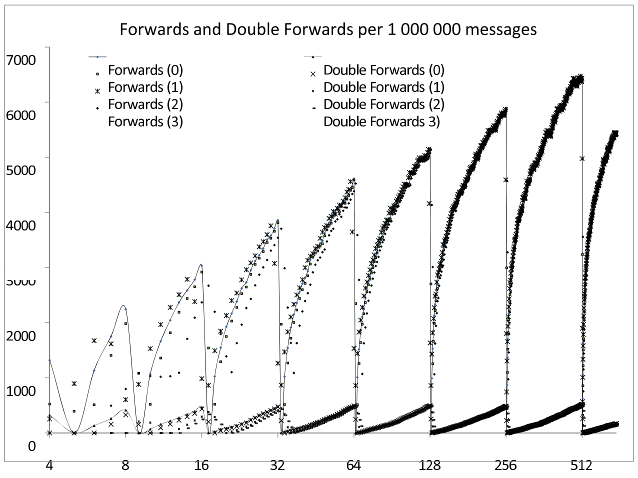


Fig. 6: Number of non-compulsory and double forwards with 1000 clients and a total of 1000000 messages in a stationary LH* system. The clients were preloaded with knowing only Bucket 0 (0), Buckets 0 and 1 (1), Buckets 0, 1, and 2 (2) and Buckets 0, 1, 2, and 3 (3).

clients with random RIDs. At the end of the simulation, the client image of most of the clients is exact, while some do not know about the last or the last and second-last bucket. We then give the number of non-compulsory forwards and the number of double forwards. We assumed that the clients start out with a file image of $i = 0$ and $s = 0$, corresponding to a system with only Bucket 0. As we can see, the number of forwards and double forwards depends very much on the number of buckets in the system, with a peak if a system bucket number is a power of two and a minimum right afterwards. If we have clients that have a different initial state corresponding to “knowing” the first two, three, or four initial buckets in the system, the behaviour stays roughly the same, but the peak moves slightly to the right. The behavior is explained by the power of the first IAM message, that gives much knowledge of the file state. As the number of buckets increases, so does the number of forwards, as it now takes more IAMs in order to learn the complete state of the system. As the clients are almost all up to date after a million messages, a second set of messages would add just a tiny bit to the number of total non-compulsary forwards, namely when a client request an operation on a record in the last (or more rarely next-to-last) bucket, but if these buckets are not included in the view of the file state that the client has.

We also see that double forwards, while more than ten times rarer than double forwards, follow roughly the same pattern, at least for systems with a larger number of buckets. If the number of buckets is small, there are situations where a double forward is never necessary.

B. Pushing File States to Servers

In contrast to LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$, we propose to push information about a changed file state to servers instead of clients. The rationale is simple: Servers are always active whereas many

```
def updIm2(self, nrBuckets)
   self.i = int(floor(log(2,nrBuckets))
   self.n = nrBuckets - 2**self.i
```

Fig. 7: Modified IAM algorithms for use with buckets that maintain their own file state image.

clients will become inactive. This is especially the case in a typical cloud applications. In order to be able to do so, we need to abandon the spartan amount of file information that LH$*$ stores at the servers (namely only the bucket file state j) and instead store a complete view that is – as we recall – equivalent to specifying the number of buckets in the system. This knowledge is initialized when the bucket is involved in a split or a merge. At this moment, the view of the file state that the bucket has is correct. With the first additional split or merge operation, the information becomes out-of-date and we need to update it, but in a lazy manner in order to not cause a message storm in the system when a split has happened. For the actual functioning in a cloud system, is it important that LH$*$ and our variants function perfectly correctly even if buckets rely on stale information for the file state.

Definition 1: Given a bucket b, we call N_b the number of buckets in the LH$*$ file when Bucket b was created, split from or merged with.

Proposition 1: For any Bucket b, $N_b \leq N$, where N is the total number of buckets in the file at any time.

Proof: The file state and the number of buckets only change with merge and split operations. After each change, the bucket numbers form a continuous range starting with 0 and ending with $N - 1$.

We prove the statement by induction on the steps in the file development. At the time when Bucket b was created, $N_b = N = b$. If Bucket b vanishes through a merge operation, then the Proposition is a vacuous statement until the bucket is created again. This lays the base case. Assume now that the proposition is true before a split or merge operation changes the state of the file. A split operation that does not involve the bucket, only increases N, but not N_b. A split operation that splits the bucket updates N_b to N. A merge operation that involves the bucket either removes the bucket, or has it merge with the last bucket split from it. In this case, $N_b = N$. This leaves us with a merge operation that does not touch the bucket. Since $N_b \leq N$ before the merge and since the merge operation does not touch Bucket b, $N_b < N$. Therefore, $N_b \leq N$ after N has decremented through the operation. ■

Based on Proposition 1, a server with Bucket b can safely update clients by sending N_b to the client. This results in the modified IAM algorithm presented from Figure 7. This algorithm calls N_b `nrBuckets` and uses it to calculate the file state corresponding to N_b buckets. The file state might not be exact (because of intervening split operations) but will never have the client send an request to a non-existing bucket.

C. Algorithm B0: Fast Client Initiation

A new client has only one bucket in the range of its file state image, Bucket 0. Our first change to LH$*$ is $B0$, in which

```
def check(self, key):
   a = h(self.level,key)
   if a < self.split:
      a = h(self.level+1,key)
   if self.id == a:
      ACCEPT AND PROCESS REQUEST
   else:
      forward(a,
              key,
              clientRequest,
              self.id,
              forward)
```

Fig. 8: Check in the Update on Double Forward Protocol. The server receives a message with key `key`.

every change to the file state is sent directly to Bucket 0. If Bucket 0 receives an erroneously addressed request, it updates the client's file state with the correct image. A new client or a client lucky enough to make an error involving Bucket 0 will immediately be updated to the correct file state. However, the file state can change afterwards, so that the client can still commit further errors. If the client is quite inactive, the file state can change so dramatically that a double forward is still possible.

The reason for treating Bucket 0 differently follows directly from the design of LH$*$. Bucket 0 is the only bucket that is guaranteed to exist. It is usually colocated with the coordinator on the same system, so that updating of the bucket state only involves local messaging. Additionally, as we just observed, Bucket 0 is the entry point for any new client, and also, if merges are implemented, for clients that have reached an inaccessible bucket.

D. Algorithm UDF: Update on Double Forward

A design principle underlying LH$*$ and other scalable, distributed data structures is the avoidance of hot spots and bursts of activity. The elegance of LH$*$ stems from the avoidance of update messages among servers. However, while updating servers at each split is against design principles, LH$*$ still has forwarding messages between servers. We propose now an algorithm that uses server updates in the case of double forwards to avoid further double forwards. As before, this algorithm updates client state file images based on the number of buckets. It also changes the procedure for receiving client requests at a sender.

A server now stores three pieces of metadata: its identity (`self.id`), and its file state image consisting of level (`self.level`) and split pointer (`self.split`).

When a server receives a request from a bucket, it can directly use the file state to check whether the request needs to be handled by the bucket or needs to be forwarded. If the message is forwarded, then the server includes its bucket number.

When a server receives a forwarded message, it checks whether the sender of the message is the one that sent the forwarding message. If this is not the case, then it is a double forward and the server sends an image adjustment message to the original server. This image adjustment message just

```
def receiveClientRequest(self, client, key):
   ...
   self.gossipCount -= 1
   if self.gossipCount == 0:
      self.gossip()
      self.gossipCount = SERVERGOSSIPNUMBER
   ...

def gossip(self):
   if(self.nextBucket < self.bucketNr):
      adjust(self.nextBucket, self.nrBuKnown)
   nextBucket += 1
```

Fig. 9: Gossip algorithm to update servers.

sets the file state of the original server to the state of the receiving sender. Thus, the original server cannot repeat the same mistake, but has all the information to determine that the receiving sender is the server where the record in question should reside. The complete algorithm is given in Figure 8.

E. Gossiping algorithms

Any system that wants to limit forwards to only compulsive forwards needs to maintain file state at all servers and push information to clients, whether they commit an addressing mistake or not. At the same time, the principles of scalable distributed data structures need to be maintained, which precludes maintaining a global state. A compromise between these two incompatible design decisions lies in updating server and client information in a *lazy* manner.

We introduce two simple gossiping mechanism, one for updating bucket servers, the other one for updating clients. In contrast to the usual sense of the word, in our scheme gossip messages are not randomly triggered. Each bucket maintains a *gossip counter* called `gossipCount`, Figure 9, and a value `nextBucket` that is the next bucket to be updated. Whenever the bucket is created or whenever the bucket is split, and thereby "knows" that the total number of buckets in the scheme has increased, it sets these numbers to their initial values. The value of `nextBucket` becomes 0 and the `gossipCount` is set to a system parameter representing the eagerness with which information is shared. At each processing of a client request, the bucket decrements the value of `gossipCount` until it reaches 0. In this case, the value is restored to the initial value and a message is sent to the bucket indicated by `nextBucket` with the information of the number of buckets in the system. The bucket that receives the message (i.e. Bucket `nextBucket` updates its image of the file state to reflect the new information, but only if it does not have better one. In the case of a file that only grows, the decision of which information is better is simple: The one about more buckets is newer. In the case of a file that allows bucket merges, the file state image needs to also contain a time stamp by the split coordinator.

Our second gossiping mechanism is for client updates. A client maintains a similar counter. The counter is decremented whenever the client sends a request to a bucket. If the counter reaches 0, then the client sets a flag in the message requesting a file image adjustment from the responding bucket. In contrast to normal IAMs, there is no guarantee that the client receives new information.

V. Experimental Evaluation

For the experimental evaluation, we focus only on growing LH$*$ files. Shrinking files are not only rare, but also ideosyncratic and modelling them requires making assumptions that are hard to justify.

Our first scenario is one where the file is stationary and clients make a large number of request. We already applied this scenario in Figure 6. If we apply algorithm $B0$ in this scenario and the clients all start out fresh, then there will be only compulsory forwards (one per client) and no double forwards at all. This is of course different if clients have different initial states, since their file state image only gets updated to the correct state if they erroneously address Bucket 0 or if they obtain the correct state by addressing records in the last bucket created.

Our second scenario has an LH* file that is growing at different speeds. We simulate 1000 clients and calculate forwarding and double forwarding operations. In the *low growth* scenario, a client makes on the average one request before a split occurs. In the *moderate growth* scenario, a client makes on average 0.05 requests before the file grows. Finally, in the *fast growth* scenario, a client makes on average 0.005 operations before the file increases by an additional bucket.

As a comparison point, we evaluate first LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$. LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ uses a client ID and then uses LH$*$ addressing to assign each client to a *tutor*, which is a server that pushes its bucket file state to its assigned clients whenever it changes. A client can get a new tutor when its previous tutor splits or if the current tutor is merged with another bucket. That server than becomes the new tutor. It can be shown that LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ has no double forwards because each client is updated by its tutor once as the file grows from $2^k - 1$ to 2^{k+1} buckets (for any $k \in \mathbb{N}$).

We then evaluated the baseline behavior and our less involved adaptations of B0 (fast new client initiation by always keeping Bucket 0 actualized), and UDF (update servers on double forwards). We then evaluated two variants of gossiping, one where servers update another after 100 requests (Gossip 100) and one where servers update another server after 10 service requests and where clients request an update with every fifth operation that they make (Gossip 10 5).

Our evaluation uses simulation (written in Python). In order to avoid influences of the starting configuration, we only report the averages of simulations where we start with k buckets, $k \in \{20, 21, \ldots 500\}$ and 1000 clients. For the gossip protocols, the clients have a file state that correctly reflects the initial configuration, otherwise the client's view only has Bucket 0. The total number of buckets created depends on the growth rate. We observe the effects of 500,000 requests, each originating from a random client (among the 1000) and with a random RID.

Our results in Table I show first that single forwards are a reasonably frequent occurrence in these scenarios, but that double forwards are rare. Avoiding double forwards is thus more important for maintaining QoS agreements than for keeping average access costs down. In our scenarios, the costs of pushing information to clients in LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ pays off, even in the fast growth scenario, where there is a considerable amount of unsolicited update traffic between tutors and their assigned clients. We discuss this more below, Table III.

Secondly, they show that B0 – fast new client initiation – has the biggest effect in lowering the rate of single and double forwards. Only when there is fast growth do active clients experience a sufficiently dearth of activity that their image of the file state is so far away from the actual file state that double forwards appear at all. Indeed, in the moderate growth scenario, B0 performs (surprisingly to us) better than the gossipping protocol *gossip* 1000 that takes time to update the file state image maintained at B0.

Third, they show that the differences between B0 and UDF are minute. In fact, even while not identical, their graphs in Figure 10 overlap. Only the aggressive gossiping algorithms can do better.

While our numbers for LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ make it attractive, they do not take into account the overhead caused by pushing update information to the clients. All clients, whether active or not, receive one update message from their *tutor*, the bucket to which they are assigned. In our scenarios, these gives a message overhead that makes our other protocols at least competitive, if not better. In practice, the results will depend on the difference between activity levels between clients, which in turn depends on the nature of the application and does not lend itself to general rules. These numbers can be greatly higher if we have a substantial amount of churn, since each new client needs to find its tutor to get updated. We give the numbers for our simplified scenario in Table III. For example, in the fast growth scenario, we have 8.87 times that a tutor updates its pupil, which amounts to a tutor-to-pupil message overhead of 1.78% per client request. Since we assume no churn, true numbers should be higher.

We then investigated various parameter choices for the fast-growth environment. Our results are shown in Table II. The gossiping protocols all update Bucket 0 immediately with every split. As we can see clearly, the rate at which client update their file image has a much higher impact than the rate at which servers update each other. The explanation lies in the resilience of the original LH$*$ protocol. The process of bucket splits maintains servers reasonably well informed about the number of buckets in the file. Just based on the bucket level, the server knows the powers of two limiting the total number of buckets in the file. While the speed of server updates has a small, but predictable effect on the total number of messages that need to be forwarded, the number of double forwards sometimes ends in a statistically dead head or can even ever so slightly increase with higher growth rates. This is a perverse effect of the efficiency of the system, since less forwarding mistakes mean less load at lower-numbered buckets and therefore fewer updates from there. We observed it when studying the raw data, not the cumulative data given in Table II. It only happens when we start the simulation with

TABLE I: Percentage of client requests forwarded once and twice

.

	LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$	Baseline	B0	UDF	Gossip 100	Gossip 10 5
			Low Growth			
Single Forwards	4.810%	5.308%	4.872%	4.872%	4.872%	4.413%
Double Forwards	0.0000%	0.0493%	0.0000%	0.0000%	0.0000%	0.0000%
			Moderate Growth			
Single Forwards	6.930 %	8.257%	8.045%	8.044%	8.044%	7.323 %
Double Forwards	0.0000%	0.051226%	0.001169%	0.00095%	0.00095%	0.000605%
			Fast Growth			
Single Forwards	7.305%	8.918%	8.805%	8.802 %	8.802 %	8.047%
Double Forwards	0.0000%	0.064443%	0.015172%	0.014428%	0.014428%	0.011058%

TABLE II: Percentage of client requests forwarded once and twice for gossiping protocols

Protocol	**Single Forwards**	**Double Forwards**
gossip 2 (server), 2 (client)	7.195755%	0.007473%
gossip 5 (server), 2 (client)	7.200378%	0.007582%
gossip 10 (server), 2 (client)	7.203034%	0.007687%
gossip 100 (server), 2 (client)	7.205120%	0.007976%
gossip 1000 (server), 2 (client)	7.205322%	0.007986%
gossip 2 (server), 5 (client)	8.043643%	0.010941%
gossip 5 (server), 5 (client)	8.044891%	0.011090%
gossip 10 (server), 5 (client)	8.046876%	0.011058%
gossip 100 (server), 5 (client)	8.048471%	0.011275%
gossip 1000 (server), 5 (client)	8.048615%	0.011275%
gossip 2 (server), 10 (client)	8.404569%	0.012958%
gossip 5 (server), 10 (client)	8.404756%	0.012913%
gossip 10 (server), 10 (client)	8.406508%	0.012866%
gossip 100 (server), 10 (client)	8.408157%	0.013090%
gossip 1000 (server), 10 (client)	8.408270%	0.013079%

TABLE III: Client update messages in LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$.

Scenario	Updates per Client	Overhead per client request
Low Growth	1.830420	0.366084%
Moderate Growth	5.599402	1.119880%
Fast Growth	8.876860	1.775372%

a system with already more than 300 servers and clients that have file state views that are exact.

As we can see from Figure 10, the number of forwards becomes slightly lower as the LH$*$ files become larger. This is because a client with slightly inaccurate image will not be as likely to make an addressing mistake. We can also see how statistically stable the ranking of the various protocols are, but the small but negligible contribution of UDF over B0 is not visible.

As opposed to spreading information among servers, pulling information from servers to clients has a very noticeable effect both on lowering the number of single forwards (an improvement by 24%) and especially of double forwards, with an improvement of over 800%.

In conclusion, we observe that for our scenarios, pulling information via piggy-backing is a mechanism that is successful in limiting the number of double forwards and single forwards messages. Depending on the growth of the file, it can slightly outperform LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$.

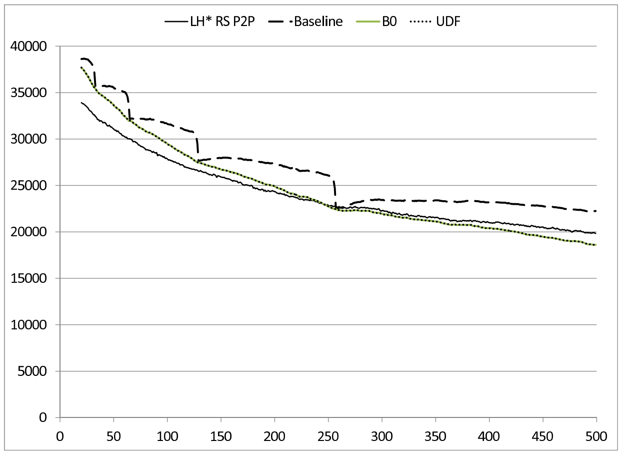
VI. Conclusions

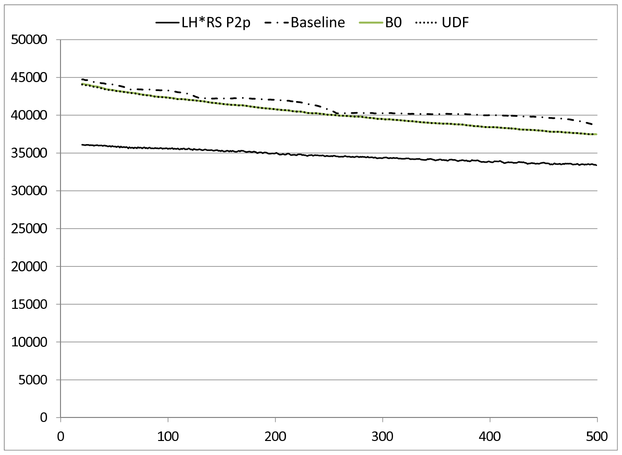
We have here presented solutions in preventing double forwards in LH$*$, a fundamental problem for allowing tight quality of service degrees. Our solutions show that a single change (represented in our algorithm B0) that operates on the way clients are initialized, suffices to remove all or almost all double forward operations. The QoS promise can then be set to reflect the latency of three messages among participants.

Further improvement requires either reasonably aggressive gossiping or pushing information to client, which is done in LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$. The latter causes an important overhead.

A compromise solution would be to have an initialization service for new clients and use a self-adjusting time-out for clients that have made no access in order to revisit the initialization server. This solution splits the double functions of Bucket 0 into being a normal bucket and being the preferred bucket for important updates to a client's file state.

Future work will seek to overcome the problem of *wandering* buckets in LH$*_{\mathrm{RS}}$, the high availability version of LH$*$. These buckets are caused by server failures where the buckets on the failed server(s) are reconstructed elsewhere. How to bring the information about the new locations to the clients is the problem to be solved. This work will also try to make the distribution of bucket locations to clients in LH$*$ more efficient. In this case, which should prove the standard one for LH$*$ deployment, gossiping protocols become much more attractive. In LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$, a client only receives an update when its tutor has split. Thus, the client will discover that a bucket has wandered usually through error. The standard failure mechanism in LH$*_{\mathrm{RS}}$ handles this case well, but it efficiency could be improved.





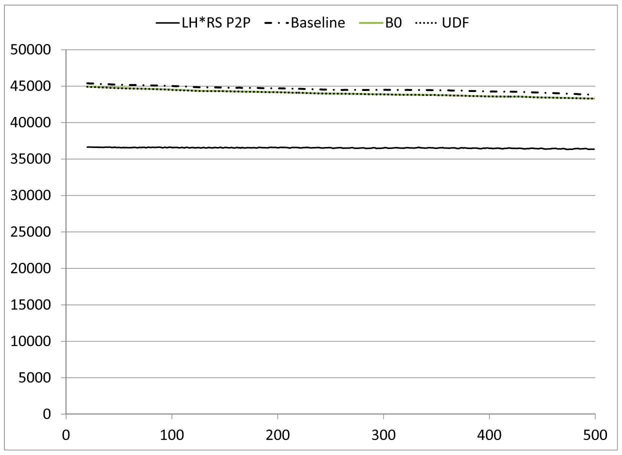


Fig. 10: Non-compulsory single forwarding messages in the low, moderate, and fast growth scenario depending on the starting size (in buckets) of the LH∗ file.

References

[1] R. Devine, "Design and implementation of DDH: A distributed dynamic hashing algorithm," *Foundations of Data Organization and Algorithms*, pp. 101–114, 1993.

[2] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing - a fast access method for dynamic files," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 3, pp. 315–344, 1979.

[3] B. Kröll and P. Widmayer, "Distributing a search tree among a growing number of processors," in *ACM SIGMOD Record*, vol. 23, no. 2. ACM, 1994, pp. 265–276.

[4] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Proceedings of the sixth international conference on Very Large Data Bases*, vol. 6, 1980, pp. 212–223.

[5] W. Litwin, J. Menon, and T. Risch, "LH* schemes with scalable availability," IBM Almaden, Tech. Rep., 1998, RJ10121 (91937).

[6] W. Litwin, R. Moussa, and T. Schwarz, "LH* RS—a highly-available scalable distributed data structure," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 3, pp. 769–811, 2005.

[7] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH* - a scalable, distributed data structure," *ACM Transactions on Database Systems (TODS)*, vol. 21, no. 4, pp. 480–525, 1996.

[8] W. Litwin, M.-A. Neimat, and D. Schneider, "RP*: A family of order preserving scalable distributed data structures," in *Proceedings of the International Conference on Very Large Data Bases*. INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS (IEEE), 1994, pp. 342–342.

[9] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH∗: Linear hashing for distributed files," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data (SIGMOD '93)*. ACM, 1993, pp. 327–336.

[10] W. Litwin, H. Yakouben, and T. Schwarz, "LH* RS P2P: a scalable distributed data structure for P2P environment," in *Proceedings of the 8th international conference on New technologies in distributed systems*. ACM, 2008, p. 1.

[11] E. Nardelli, "Distributed k-d trees," in *Proceedings 16th Conference of Chilean Computer Science Society (SCCC96)*. Citeseer, 1996, pp. 142–154.

[12] A. Rathi, H. Lu, and G. E. Hedrick, "Performance comparison of extendible hashing and linear hashing techniques," in *Proceedings of the 1990 ACM SIGSMALL/PC symposium on Small systems*, ser. SIGSMALL '90, 1990, pp. 178–185.

[13] H. Yakouben and S. Soror, "LH* RS P2P: a fast and high churn resistant scalable distributed data structure for P2P systems," *International Journal of Internet Technology and Secured Transactions*, vol. 2, no. 1, pp. 5–31, 2010.