Int J Parallel Prog (2016) 44:709–734
DOI 10.1007/s10766-015-0371-8

Fast LH∗

Juan Chabkinian[1] · **Thomas J. E. Schwarz SJ**[2]

Received: 5 January 2014 / Accepted: 6 July 2015 / Published online: 23 July 2015

Abstract Linear Hashing is an efficient and widely used version of extendible hashing. LH∗ is its distributed version that stores key-value pairs on up to hundreds of thousands of sites in a distributed system. LH∗ implements the dictionary data structure efficiently by not using a central component and allows the key-based operations of insertion, deletion, actualization, and retrieval as well as the scan operation. Because it does not use a central addressing component, clients or servers in LH∗ can commit an addressing error by sending a request to a wrong server. This server then forwards the message to the correct server either directly or in one but never more than one additional forward operation. We discuss here methods to avoid this double forward, which, while rare, still might breach quality of service guarantees. We compare our methods with $\mathrm{LH}*_{\mathrm{RS}^{\mathrm{P2P}}}$ that pushes information about changes in the file structure to clients, whether they are active or not. A second problem especially relevant in high churn environments such as modern data centers is that sites can suddenly become inaccessible. The various high and scalable reliability versions of LH∗ then reconstruct the data lost on this site elsewhere. We present a solution to the resulting "wandering bucket" problem that allows clients to find the data at their new location.

Keywords Scalable distributed data structures · LH∗ · Cloud computing

✉ Thomas J. E. Schwarz SJ
TSchwarz@uca.edu.sv; tschwarz@calprov.org

Juan Chabkinian
juanjose.chabkinian@gmail.com

1 Universidad Católica del Uruguay, Montevideo, Uruguay

2 Universidad Centroamericana, San Salvador, El Salvador

1 Introduction

Cloud computing has brought distributed computing to the masses. The popular Map-Reduce framework allows distributed and parallel scanning of a large file, followed by the aggregation of values to obtain an end result. In the popular implementations by Google and Yahoo (Hadoop), the file grows only by appending fixed-sized blocks. Scalable Distributed Data Structures (SDDS) such as LH*, which uses a linear hash partitioning [14], and RP*, which uses range partitioning [12], avoid this problem. In these SDDS, the file is stored in fixed-sized buckets and the file grows through bucket splits that add an additional bucket to it. One of the most attractive features of an SDDS is the absence of a central component that allows a client to find the address of a record. Instead, clients use a local *view* of the file to determine the bucket containing the record and the location of the bucket. An SDDS file adjusts to changes in the number of records by *splitting* and *merging* buckets. Consequentially, the views of clients become outdated. The client then can commit an *addressing error* by sending a request to the wrong server.

In LH$*$, clients that commit such an address error receive additional information about the file. Any request is resolved with at most two additional forwarding messages. Theoretical and empirical analysis shows that most addressing errors are resolved with one forward and only few with two forward messages. In this paper, we are investigating a method to avoid double forwards, since they might cause havoc with cloud Quality of Service (QoS) guarantees. Our method spreads file information among servers. In contrast, Yakouben [16,20] proposes to *push* file information to clients. This method guarantees that there are no double forwards, but has a high message overhead. We compare both methods.

Cloud environments see many node failures. Indeed, often node failures result from administrative shutting down of physical nodes. The applications are supposed to tolerate this [1]. The *scalable availability* version of LH$*$, called LH$*_{\mathrm{RS}}$, provides this tolerance [10]. It reconstructs unavailable buckets elsewhere using additional parity data. Each reconstruction leads to a second type of addressing error, where a client sends to the correct bucket, but at the wrong node address.

This paper proposes changes to deal with these two types of addressing errors. To deal with the first type of addressing errors, we avoid pushing information to inactive clients, since in a cloud environment clients can be very transitory and frequently fail abruptly. Our alternative strategy pushes information about changes to servers so that double forwards become rare. To deal with the second type, we use LH$*$ itself to store the association between bucket number and network addresses at the LH$*$ buckets and use the normal interaction between clients and servers to let clients update their localization information.

The rest of the article is organized as follows: We review shortly relevant work in Sect. 2. Section 3 briefly reviews the most important aspects of LH$*$ and LH$*_{\mathrm{RS}}$ without going into details that are not necessary for the understanding of our contributions here. We present an efficient address resolution protocol in Sect. 4 that solves the wandering bucket problem without the need of involving a central component. Section 5 presents fast LH$*$, our solution to avoiding double forwards and to limit single forwards. Section 6 presents an analytic and experimental evaluation of the methods proposed. We then draw our conclusions.

2 Related Work

Litwin introduced Linear Hashing in 1980 [7] and (with Neimat and Schneider) LH$*$ in 1993 [13,14]. This is not the only distributed data structure based on hashing; among the proposals DDH [4] and Extendible Hashing [5] stand out. Other scalable distributed data structures are RP$*$ [11] that uses range partitioning and predates this aspect of Bigtable [3], distributed k-d trees [18], and distributed search trees [6], to name a few. LH$*$ has seen been expanded to a variety of schemes providing availability [8]. LH$*_{\mathrm{RS}}$ is the most attractive of these possibilities since its failure tolerance increases with the likelihood of failures [10].

High availability creates the problem of the *wandering bucket* where after a site unavailability a bucket is recreated at an address unknown to clients who need to access it. Litwin et al. solve this problem by having the client request help from the coordinator, turning the coordinator into a potential bottleneck. Yakouben and colleagues propose to push file state information to clients [16,20]. Pushing file state information also avoids double forwards. The method of LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$ is preferable to ours if the set of active clients is stable. If however clients come and go, which would be a typical situation in cloud computation, then administering clients and sending the push messages create a big overhead in LH$*_{\mathrm{RS}^{\mathrm{P2P}}}$. The work on avoiding double forwards was presented earlier at SBAC-PAD 2013 [2].

3 LH$*$ and LH$*_{\mathrm{RS}}$

Linear Hashing (LH) is a widely adopted form of extendible hashing. It stores records in buckets, whose number automatically adjusts to the total number of records. The number of buckets (*the extent*) 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 too 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 extendible hashing [5] or Devine's DDH [4], the bucket to be split is not necessarily the one that caused the overflow. Buckets are numbered contiguously starting with 0 and split in a fixed order 0; 0, 1; 0, 1, 2, 3; 0, . . . 7; . . . 0, 1, . . . $2^k - 1$; . . ., where each round splits the buckets from 0 to $2^k - 1$ with increasing $k = 0, 1, \ldots$. The design of LH results in slightly worse storage utilization than with extendible hashing, but also in considerably faster record lookups because no central directory structure is needed [19]. If there is an underflowing bucket or if the storage utilization is too small, then a *merge* operation undoes the last split operation.

3.1 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. These buckets tend to be much larger (several hundreds MB vs. three or four records) and are often organized as a linear hash file. 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*, *update*, and *read* as well as the global operations of scanning, which for all records with a certain pattern in their contents, and function shipping, as long as the functions have only a single record as their argument. The latter capability was a precursor of the map-reduce scheme.

3.2 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:

$$(\mathbf{LH}) \qquad a = h_i(c) \qquad \mathbf{if}\, a < s: \quad a = h_{i+1}(c)$$

with hash functions h_j defined by $h_j(c) = c \bmod 2^j$ 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.

3.3 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).

```
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`. `self.j` is the bucket level

```
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 message with bucket level `j` and bucket address `a`. The algorithm updates the local file state `self.i` and `self.j`

If a bucket receives a request from a client for a record with key `key`, it executes the algorithm in Fig. 1 to determine whether the bucket itself should have the record or whether the request should be forwarded to another bucket [14]. In this algorithm, `self.j` is the level of the bucket and `h` is the hash function, which takes the level as its first parameter. The bucket address is `a`.

3.4 Client Image Adjustment

A client maintains a view 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 Fig. 2.

Knowing that a bucket exists does not imply that one knows the network address. We deal with this problem by maintaining a parallel, but much smaller LH$*$ structure using the same buckets where we store network addresses. In addition, we assume that IAM messages contain address information of all servers involved. Each client maintains a table of buckets and their (last-known) locations.

3.5 Splitting and Merging 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 $2^i + s$

Fig. 3 The global file state `self.split`, `self.level` is updated after a split to represent the additional bucket

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

Fig. 4 Development of a growing LH* file

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

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

and are therefore moved there. Afterwards the global file state is updated according to the algorithm presented in Fig. 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 (Fig. 4).

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 address 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 the algorithm given in Fig. 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.

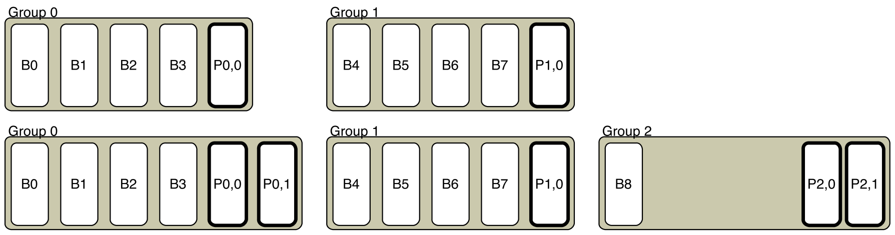

Fig. 6 LH$*_{\text{RS}}$ file incrementing its availability level. On top, the file encompasses 8 buckets, organized into two reliability groups each with a parity bucket. Below the result after splitting Bucket 0. The new Bucket 8 is for the moment the only data bucket in the new third group. The first and the third group now have two parity buckets, but the second one remains with one parity bucket

3.6 Example

We illustrate the development of an LH$*$ file from one to six buckets in Fig. 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 Fig. 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.

3.7 LH$*_{\text{RS}}$

Even in a normal distributed system, nodes fail and buckets become unavailable, and these unavailabilities are more pronounced in the harsh environment of the cloud. A variety of data structures were proposed to provide availability to LH$*$, but we concentrate here on LH$*_{\text{RS}}$ that provides *scalable* availability [9]. Scalable availability means that the protection against unavailable buckets increases with the total number of buckets. LH$*_{\text{RS}}$ groups buckets into bucket groups and adds to each bucket group a variable number of *parity* buckets. The parity buckets consists of parity records that are generated using erasure resistant coding from one record each of the *data* buckets in the group. The number of parity buckets per bucket group determines the *availability level* of the LH$*_{\text{RS}}$ file. The number of parity buckets per group is the same in all reliability groups unless the availability level is currently being increased. To increase the availability level and hence the resilience of the file against unavailabilities, we wait until the split mechanism reaches the first bucket in a bucket group. We then add a parity bucket to the current bucket group. The new bucket is for the time being the only bucket with data in the new bucket group which is created with a number of parity buckets equal to the increased availability level. Figure 6 gives an example, where the *availability level* is changing from one to two. Once the split pointer in this example reaches Bucket B4, Group 1 and the newly formed Group 3 will also encompass two parity buckets.

The availability of LH$*_{\mathrm{RS}}$ and similar data structures introduces a new type of addressing error, caused by a "wandering bucket". A wandering bucket is a bucket that becomes unavailable and then is reconstructed elsewhere changing its physical (IP) address. LH$*$ has a client who cannot find a bucket contact the coordinator, but this creates a hot spot if there are many clients and more than a few wandering buckets. A client using a wrong address might direct itself to a non-existing node or to a different service residing at the address. The client then has to use a time-out to decide that something is wrong, which increases the length of resolving the request. Since LH$*_{\mathrm{RS}}$ allows for out-of-band reconstruction for records demanded by a client and for out-of-band insertion and deletion, a client can in general expect rapid service once a bucket has been flagged as unavailable. Only if the client is so unlucky to try to interact with an unavailable bucket before it has been flagged as such, will the client experience bad service. We address the problem of not finding a wandering bucket here by using the distributed Address Resolution File and by a mechanism that informs clients of bucket locations.

4 ARF: Address Resolution File

LH$*$ clients and nodes store information about bucket addresses in a table associating bucket numbers with node addresses. Since node addresses are between 4B and 16B, each client or node can easily maintain an association table for hundreds of thousands of buckets. Buckets only need to maintain a small table of their direct descendants in order to allow forwarding. By piggy-backing information on replies to forwarded messages, a single- or a double-forward adds to the client's association table.

4.1 The Address Resolution File

Unfortunately, the wandering bucket problem cannot be solved with just this simple device. Instead, we form an *Address Resolution File* (ARF) that is stored in buckets together with existing buckets of the LH$*$ file. It is a distributed version of the association table that each client maintains. It allows a client or a bucket to find a reconstructed bucket quickly and a new client to quickly generate its private association table or possibly even replace it. For its proper functioning, the ARF needs to have the following properties:

- The address of a bucket is not stored at the bucket itself.
- The ARF tolerates failures.
- A single bucket address is replicated at various buckets of the ARF and requests for this address are guided to different replicas to avoid hot spots.
- Only few buckets need to be updated if a bucket is reconstructed elsewhere.

Fig. 7 Pseudo-code for the calculation of the mixed vector from a category A address a, a category B address b, a mixing vector, and a size, all treated as an unsigned integer

```
def mixer(a, b, mixing, size):
    result = 0
    power = 1
    for i in range(size):
        cCur = mixing%2
        mixing = mixing//2
        if cCur == 0:  #choose from a
            cur = a%2
            a = a//2
        else:          #choose from b
            cur = b%2
            b = b//2
        result = cur*power+result
        power *= 2
    return result
```

– Information on a new bucket address needs to be sent to only a few buckets.

The load of an ARF bucket depends on the number of clients that want to access a wandering bucket and that do not have the new address. The LH$*$ structure however has only access to the total number of LH$*$ buckets and not to the number of clients in this situation, but it can react to the number of requests received by an ARF bucket to grow and thereby dilute the load. At the same time, each replica of a bucket address—IP address record needs to be updated when a bucket is reconstructed.

4.2 LH-Files Governed by Two Categories

A normal linear hash file grows based on a single factor, the number of records in it. We want to control growth of the AFR based on two independent parameters, namely the LH$*$ file extent—the number of buckets in it—and the likely number of clients that need information. We can assume that both parameters only grow. We now adapt the LH addressing mechanism to these two categories.

The key idea is that of *mixing* two different addresses. Let $\mathbf{c} = (c_n, \ldots, c_0)$ be a binary vector, the *mixing vector*, and let $\mathbf{a} = (a_r, \ldots, a_0)$ and $\mathbf{b} = (b_s, \ldots, b_0)$ be two addresses, also given as binary vectors. The *combined* address is made up of the bits in **a** and **b** in order. Let z_i denote the position of the i^{th} zero in **c** from the right and u_i the position of the i^{th} one in **c** taken from the right. For example, if $\mathbf{c} = (110110110110)$, then $z_0 = 0$, $z_1 = 3$ and $z_2 = 6$ and $u_0 = 1$, $u_1 = 2$, and $u_2 = 4$. We observe that both series together iterate through all positions 0, 1, … of **c** and that we can therefore invert them. The combined address **m** is defined by the following rule for its i^{th} coordinate

$$\mathbf{m}_i = \begin{cases} a_j \text{ if } z_j = i \\ b_j \text{ if } u_j = i \end{cases}$$

With words, starting from the left, we put coordinates from **a** whenever the mixing vector coordinate is a 0 and coordinates from **b** whenever the coordinate is 1, where we exhaust all coordinates from **a** and **b**. Perhaps, the pseudo-code in Fig. 7 is an easier definition.

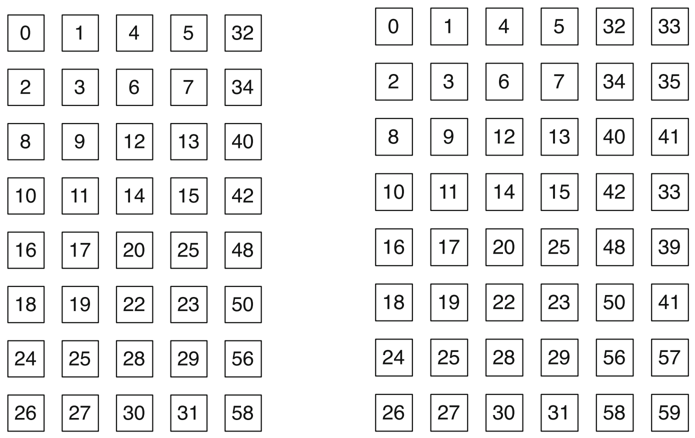

Fig. 8 *Left* Bucket allocation where category A has file state $j_A = 2$ and $s_A = 1$, category B has file state $j_B = 3$ and $s_B = 0$ and mixing vector (011010). *Right* Bucket allocation after changing the category A file state to $j_A = 2$ and $s_A = 2$

Example: Let $\mathbf{c} = (1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0)$. Then the combined address of $\mathbf{a} = (a_{12}, a_{11}, \ldots, a_0)$ and $\mathbf{b} = (b_{12}, b_{11}, \ldots, b_0)$ is

$$(b_{10}, b_9, b_8, b_7, b_6, a_3, b_5, b_4, b_3, a_2, b_2, b_1, a_1, b_0, a_0)$$

We can now come to the definition of the LH file determined by two growth categories A and B. In our case, the categories will be the number of bucket addresses stored, category A, and the number of replica of each address, category B. Thus, A controls the size of the information stored and B controls the load resulting from address request for a wandering bucket.

The file state now consists of two traditional file states, one for category A and one for category B, and the mixing vector. The mixing vector has a length equal to the sum of the levels of the file states in each category. A client addresses a bucket using both file states to calculate a category address and then mixing them using the mixing vector to obtain a combined address. Addressing errors are dealt with in the usual manner and lead to updates of the client's file state.

The big difference between normal LH$*$ and two-category LH$*$ is the costs of extending the address range in one of the categories. We define a *super-bucket* in category A by taking a category A address a and defining $S_A(a)$ to be the sets of buckets with a combined address of a and any other current category B address. We define $S_B(b)$ for a category B address b to be the set of buckets obtained by combining any current valid category B address with b. The number of buckets in a set S_X, $X \in \{A, B\}$, is given by the extent of the other file in the other category.

Originally, the file is stored in a single bucket 0. A bucket calculates the load for each category and sends an overflow message to the split coordinator whenever it perceives a overload. This is simple for category A. If a bucket stores too many addresses, it suffers an overload. The observed number of requests for address information on buckets depends on the number of bucket addresses stored (which is also controlled by category A), the total number of clients, the proportion of clients that obtain address information through other mechanisms, and the activity level of clients, which might also be variable. Thus, controlling in category B is more difficult, leading potentially to over-provisioning.

When the split coordinator receives an overload message in one category, let it be X, the coordinator increases the extent of the file in that category by splitting the address c pointed to by the split pointer of the category. As a consequence, the coordinator sends split messages to all buckets in $S_X(c)$. If the split pointer is pointing to Bucket 0, then the coordinator also needs to increment the length of the mixing vector and set its new coordinate to reflect the current category.

Example: Figure 8, left, shows the buckets used by a double category LH$*$ files. Category A has file state with level $j_A = 2$ and split pointer $s_A = 1$, category B has file state with level $j_B = 3$ and $s_B = 0$, and mixing vector (011010). Assume that a client accesses a record with address $3 = 11_2$ for category A and $5 = 101_2$ for category B. According to the mixing vector, the combined address is $15 = 001111_2$. Assume now that the coordinator wants to split according to category A. This changes the file state to $j_A = 2$ and $s_A = 2$ and splits $S_A(1)$. $S_A(1)$ consists of all buckets that are mixed from A-address 1, i.e., all those with binary representation $**0*1_b$ where the asterisk stands for any binary digit. Therefore $S_A(1) = \{1, 3, 9, 11, 17, 19, 25, 27\}$ or the second column in Fig. 8. The right side gives the result of the split. According to LH$*$ addressing and the combination rules, the new buckets used have bucket numbers added by 32. Finally, assume that the coordinator now decides on a category B split, changing the file state to $j_B = 4$ and $s_B = 1$. First, the coordinator needs to extend the mixing vector to (1011010). The split of category B address 0 creates a new address 8. $S_B(0)$ consists of all buckets whose bucket number has a binary representation $*00*0*_2$, which is the first row in Fig. 8, right, and consists of Buckets 0, 1, 4, 5, 32 and 33. It produces the new Buckets 64, 65, 68, 69, 96 and 97.

When we apply this scheme to our two categories of replication and bucket addresses stored, we restrict splits so that we do not create buckets for the ARF where no bucket of the original LH$*$ file exist. Finally, we need to guarantee that the address of a bucket never gets stored at the bucket itself. To do so, we assume that the category of bucket addresses stored has at least extent 2. Therefore, a valid bucket address has at least one binary digit in this category. We now calculate the ARF category address by inverting the last digit, i.e. flipping it from 0 to 1 or from 1 to 0. We also assume that the mixing vector assigns the last digit of the address to this category. It then follows, that a record about the address of Bucket b never gets assigned to Bucket b itself. For the replication category, we assign each client a unique identifier or just let the client assign a random identifier to itself.

Example continued: Category A gives the address of buckets and category B gives the replication. Assume that a client with identifier 432 needs to find the new address of Bucket 101 in the situation of Fig. 8. The category A address is based on the bucket number $101 = 1100101_2$. We flip the last digit to obtain 1100100_2 and obtain a category A address of $8 = 100_2$. The category B address is based on the client identifier of $432 = 110110000_2$. It is 0. After mixing, the combined address is $100000_2 = 32$. Should Bucket 32 also fail to respond, then the client adds one to its client identifier, so that the category B address changes to $1 = 001_2$. Therefore, the client now contacts the bucket with combined address 100010_2 or Bucket 34 for information on the location of Bucket 101.

4.3 Pushing New Address Information to Clients

We propose and evaluate below a simple gossiping strategy that is based on the capability to quickly recognize that information is not outdated. We use algebraic signatures [15] for this comparison.

Each client maintains its local copy of the ARF split into pages based on the category A addressing used above. Whenever the client interacts with a server that has part of the ARF, it uses algebraic signatures to ascertain that the information that the client has is the same as the information that the server has. We use version numbers to distinguish between the unlikely case that the server has not the actual information (because the coordinator did not succeed to push changes to all copies) and the much more likely case that the client has out-of-date information. The reason to use algebraic signatures in addition to version numbers is that they depend directly on the contents while they can be quickly updated when information changes. For example, calculating the algebraic signature of a combination of two pages can be done without rescanning both pages so that managing splits is easier.

If the client has an out-of-date page, it receives a new version of the ARF file. The methods for remote file comparisons [17] could be used to cut down on the amount of information sent, but this seems hardly necessary.

Finally, in order to tune the amount of interaction between client and server, we can have the client request a comparison only with a given probability p.

5 Fast LH∗

LH∗ allows an active client to access buckets in general in a single hop without maintaining a list of active clients to which address information is pushed. A new or a dormant client that reactivates will quickly obtain a complete or almost complete view of the file through IAM messages. However, the client achieves this by making addressing mistakes that involve one or two additional hops. Fast LH∗ limits the number of addressing mistakes by pushing information on file views to servers (not to clients) in order to make forwarding more efficient by allowing a server to forward directly to the server with the information that the client needs to access. The costs of changing the algorithm are some increase in the communication between servers. Since a typical bucket contains hundreds of megabytes or even gigabytes of records, the number of servers is in general not very high and the operational costs of our changes slight. We concentrate on avoiding double forwards in order to allow tighter service time guarantees.

Our strategy is to spread information about changes in the file state to servers and not to clients. Instead of just maintaining the bucket file state level j, we use a complete image of the file state (equivalent to the supposed number of buckets in the file). We update this image at a bucket through piggy-backing on messages between servers and by additional messages.

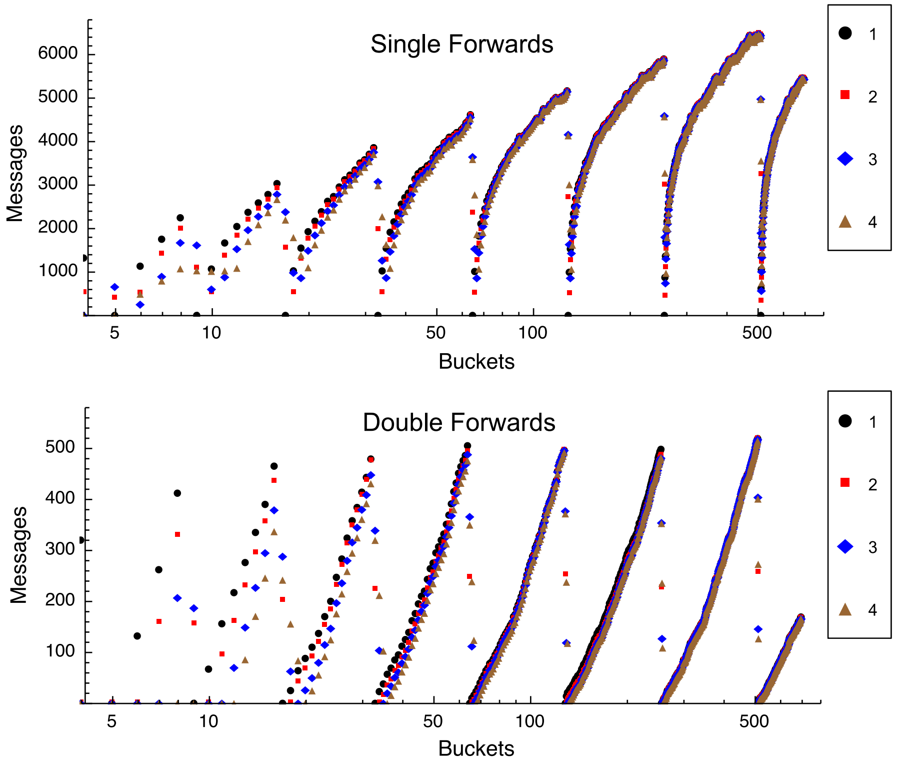

Fig. 9 Number of non-compulsory and double forwards with 1000 clients and a total of 1,000,000 messages in a stationary LH* system. The clients were preloaded with knowing only Bucket 0 (1), Buckets 0 and 1 (2), Buckets 0, 1 and 2 (3) and Buckets 0, 1, 2, and 3 (4)

5.1 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 9 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 clients with random RIDs. At the end of the simulation, the client image of most of the clients is exact, while only 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, represented by the scatterplots in Fig. 9, 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 requests 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 single 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.

5.2 Pushing File States to Servers

In contrast to $\mathrm{LH}*_{\mathrm{RS}^{\mathrm{P2P}}}$, we propose to push information about a changed file state to servers instead of clients. The rationale is simple: Buckets have to be always active whereas many clients will become inactive. In order to be able to do so, we need to increase the 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. The following proposition implies this.

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

Proposition *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,

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

Fig. 10 Modified IAM algorithms for use with buckets that maintain their own file state image consisting of file level and split pointer

```
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=True)
```

Fig. 11 Check in the update on double forward protocol. The server receives a message with key `key`

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. `q.e.d` □

According to the proposition, a server with Bucket b can safely update clients by sending N_b to the client. This results in the modified IAM algorithm presented in Fig. 10. 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 a request to a non-existing bucket.

5.3 Algorithm *B*0: 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 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.

```
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. 12 Gossip algorithm to update servers

5.4 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 its 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. The algorithm to check whether a request is a double forward is given in Fig. 11.

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 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.

5.5 Gossiping Algorithms

Any system that wants to limit forwards to only compulsory 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 an exact global state at all clients. A compromise between these two incompatible design decisions lies in updating server and client information in a *lazy* manner.

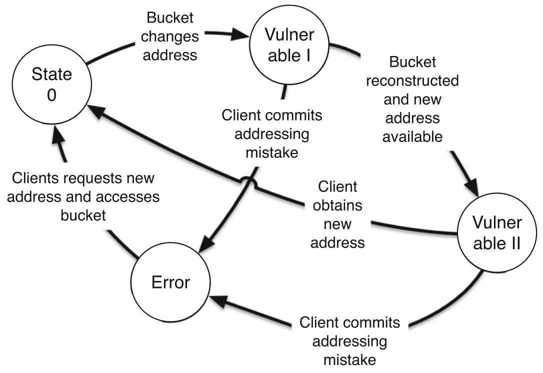
State 0
Bucket changes address
Vulner able I
Bucket reconstructed and new address available
Client commits addressing mistake
Client obtains new address
Clients requests new address and accesses bucket
Vulner able II
Error
Client commits addressing mistake

Fig. 13 Markov state model for determining addressing mistakes based on a wandering bucket

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`, Fig. 12, 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 about which information is better is simple: The state with more buckets is newer. In the case of a file that allows bucket merges, the file state image needs to contain a time stamp by the split coordinator.

If client accesses are very regular, then a deterministic gossiping protocol creates floods of gossiping messages. We believe that most LH$*$ applications would not show this regularity, but an implementation might do well to switch to a random scheme and send gossip messages with probability 1/`gossipCount` in order to be more robust.

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.

6 Evaluation

We first evaluate the probability of addressing mistakes and the efficiency of the Address Resolution File (ARF).

6.1 Probability of Addressing Mistakes

We determine analytically the probability that a client commits an addressing mistake triggered by a wandering bucket. We assume a client that interacts regularly with buckets. We assume that these interactions can be described by a Poisson process with parameter λ. We describe the position of the client with regards to a single bucket in a state model given in Fig. 13. The client is normally in State 0, where the client has the correct address of the bucket. If the bucket fails, then the client enters the "Vulnerable I" state. If the client makes a request to the Bucket in this state, then we have an addressing mistake: the client sends the request to the wrong address, the request times out (or is maybe answered with a "no such bucket is present here" message, depending on the type of unavailability), and the client uses the backup mechanism in order to find the new address of the bucket. It is even possible that the client is the first one to discover the unavailability of the bucket and that the bucket with the addresses informs the coordinator to undertake a reconstruction. When another bucket has triggered the reconstruction, the new bucket is available, and the address has been distributed to the sites with the ARF, then the client enters the "Vulnerability II" state, where the correct address information exists, but the client is unaware. Now a race conditions exists. If the client interacts with another bucket with the correct address information, then its state returns to State 0, otherwise, if the client tries to access the relocated bucket, it commits an addressing error.

We assume that the transition from State 0 to State "Vulnerable I" happens according to a Poisson process with rate ρ. The transition from State "Vulnerable II" to State 0 is $np\lambda$ where n is the number of buckets capable of pushing the new address information, p is the probability that an interaction with the client triggers the address update, and λ is the rate at which the client accesses a given bucket. (Here, we make a simplifying assumption, since in LH$*$, a recently split bucket has about half the number of records than a bucket yet to be split and hence has half the rate of client accesses. While we are only interested in the averages, the probabilities of being already split are higher for the buckets with the ARF.) The transition from State "Vulnerable I" to "Vulnerable II" is more difficult, since it models a Poisson process for the detection by the first client to make a request and the reconstruction of the unavailable bucket elsewhere. The remaining two transitions are the error transitions. They happen with a rate of λ.

In LH$*_{\mathrm{RS}}$, a relocated bucket can commence serving requests even before all its data are reconstructed. If a record is needed to satisfy such a request, but is not yet available at the new bucket, then the record is reconstructed outside of the normal reconstruction procedure. Unfortunately, these details are implementation dependent. We can assume that there is only a small, fixed time lag T_r needed between a client observing an unavailable bucket and a replacement bucket becoming available.

The probability of not committing an address error is the combination of a number of independent events: *(a)* The client is not the first one to discover the failed bucket. *(b)* The client does not try to access during time T_r. *(c)* The client updates the bucket address before accessing the bucket. The probability for the first step is given by $(N_{\mathrm{cl}} - 1)/N_{\mathrm{cl}}$ where N_{cl} denotes the total number of clients, for the second step by $\exp(-\lambda T_r)$, and for the third step by $np\lambda/(np + 1)\lambda$. Therefore, the probability of *committing* an address error is

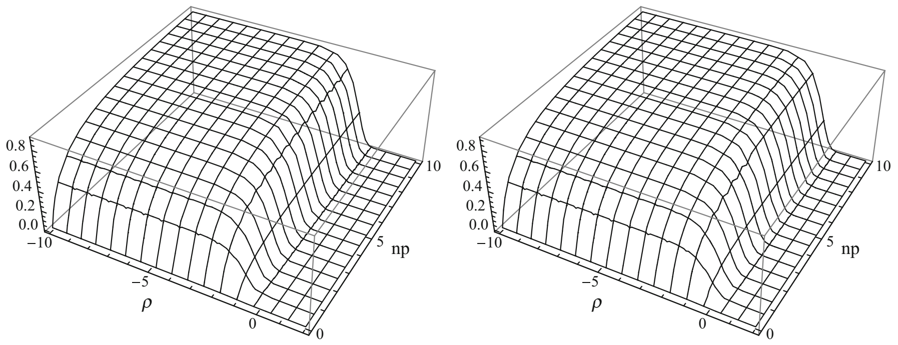

Fig. 14 Probability for not committing an address error for bucket access rate $\lambda = 10^{\rho}$, $-10 \leq \rho \leq 2$ per second, $T_r = 10\,\text{sec}$ (*left*) and $T_r = 2\,\text{sec}$ (*right*), and values of np between 1/100 and 10

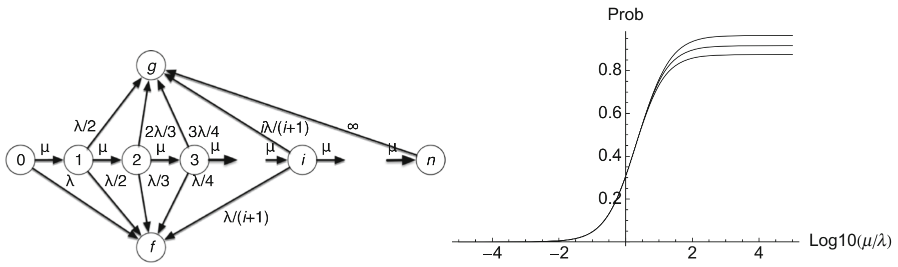

Fig. 15 *Left* Markov model for the evaluation of gossiping. *Right* Probability of *not* committing an address error (with 7, 11 and 27 buckets *from bottom to top*)

$$1 - \frac{N_{\text{cl}} - 1}{N_{\text{cl}}} \frac{np}{(np+1)} \exp(-\lambda T_r)$$

We give two examples in Fig. 14 assuming a million clients. As we can see, the probability of *not* committing an address error is high for reasonably small values of λ and reasonably high values of np. With other words, pushing ARF information to clients is quite successful. Obviously, if the number of (active) clients is very small, then the scheme looses effectiveness.

6.2 Analytical Evaluation of Gossiping Protocols

We now evaluate the efficiency of gossiping algorithms. We restrict ourselves to the typical case where LH$*$ files grow relatively slowly compared to the total number of buckets. We can then use the Markov model depicted in Fig. 15. This model describes the situation for a single client and a single new bucket. The client will make an address mistake if the client accesses the new bucket before receiving information on the existence of the bucket by interacting with another bucket. The information about the split is diffused to other buckets at a constant rate μ, which depends on the traffic by clients and the eagerness with which the information is spread. The gossiping rule has a Bucket i push this information to Bucket $i-1$ depending on the number of client requests that Bucket i receives. The central row of states in Fig. 15, States $0, 1, \ldots, n$ describes this behavior, where i is the number of servers informed about the split and n is the number of buckets before the split. For each state, there is a race condition between the client trying to access a record in the newly generated bucket and the client accessing a bucket with information on the split. The client accesses a record into the new bucket according to a Poisson process with parameter λ. If the client tries to access a record in the new bucket in State 0, the client will commit an address error, modeled by a transition to the terminal failure state, State f. If the system is in State 1, the client will access the single bucket with information about the split with probability 1/2 before trying to access the new bucket and therefore not commit an addressing error; this is modeled by a transition to the terminal success state, State g. However, with the opposite probability 1/2, the client will still commit an addressing mistake and the system will transition to State f. In general, if the system is in State i, the transition from State i to one of the two terminal states is taken with probability $i/(i+1)$ to State g and with probability $1/(i+1)$ to State f. Since the client accesses the new bucket at a constant rate λ, the transition rates are $\lambda i/(i+1)$ from State i to State g and $\lambda/(i+1)$ from State i to State f.

The Markov model is linear in the number of buckets, which limits our capability for a numerical evaluation of the model. We solved the resulting set of Chapman–Kolmogorov differential equations for $n = 7$ (bottom, Fig. 15), $n = 11$ and $n = 27$ and calculated the probability of avoiding an address error. This probability depends only on the ratio μ/λ. Figure 15 (right) gives the resulting graph with a logarithmic x-axis. As a rule of thumb, the probability is reasonably good if $\log_{10}(\mu/\lambda) > 1$ or $\mu > 10\lambda$. For example, if we have 1000 active clients making a request to a bucket at rate λ and if a site gossips after approximately every tenth access, then $\mu = 100\lambda$ and the probability of avoiding an addressing mistake is at least 80 % for a small LH∗ file and much better for a larger file.

6.3 Experimental Evaluation of Fast LH∗

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 Fig. 9. 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 though the total number of clients could be much higher. We determine the number of forwarding and double forwarding operations. In the *low growth* scenario, one of our sample clients 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. The moderate and fast growth scenarios occur if there are millions of active clients who insert records frequently. The more typical scenario should be the low growth one, where our algorithms succeed in eliminating almost all double forwards.

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 then 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 1 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 3.

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 Fig. 16 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

Table 1 Percentage of client requests forwarded once and twice

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

Table 2 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

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 3. 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 2. The gossiping protocols all update Bucket 0 immediately with every split. As we can see clearly, the rate at which clients 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 2. It only happens when we start the simulation with a system with already more than 300 servers and clients that have file state views that are exact.

As we can see from Fig. 16, the number of forwards becomes slightly lower as the LH$*$ files become larger. This is because a client with slightly inaccurate image will

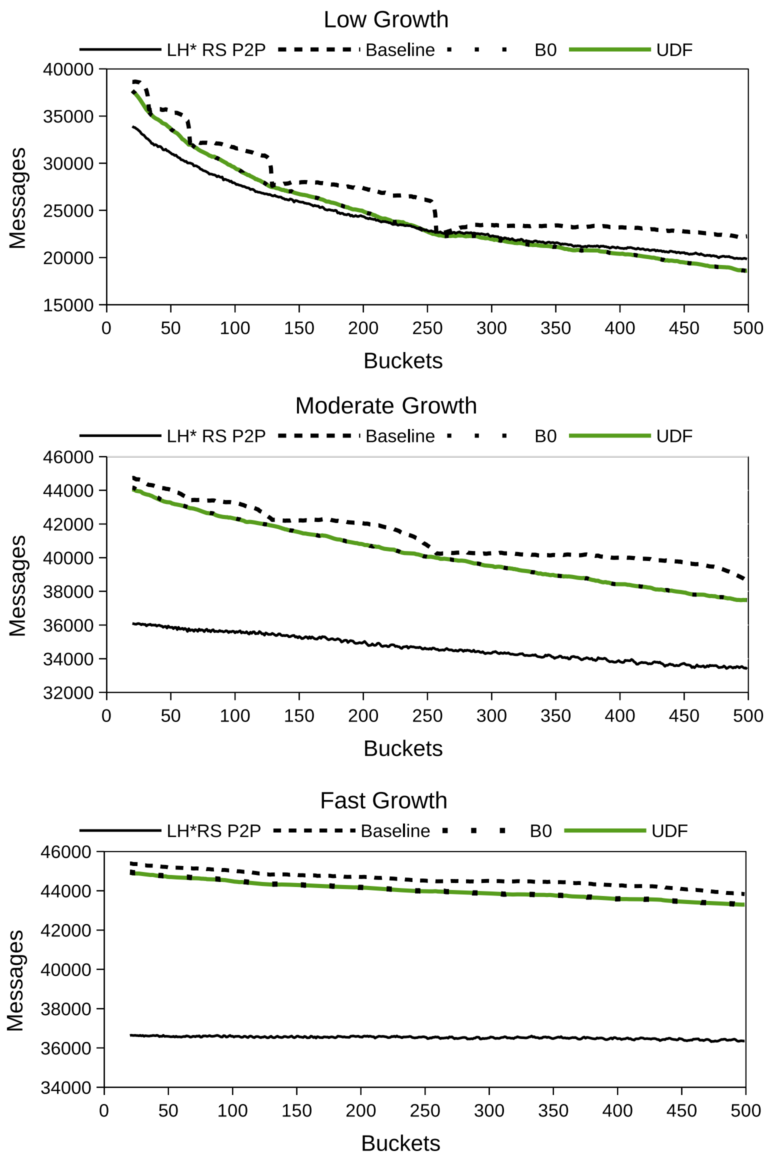

Fig. 16 Non-compulsory single forwarding messages in the low, moderate, and fast growth scenario depending on the starting size (in buckets) of the LH∗ file

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 sin-

Table 3 Client update messages in $LH*_{RS^{P2P}}$

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

gle 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 piggybacking 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 $LH*_{RS^{P2P}}$ (Table 3).

7 Conclusions

Large scalable distributed data structures such as Bigtable [3] are already used for big computations in cloud environments. We presented here ways of making one particular data structure, LH∗, which was proposed two decades ago, fit for the harsh environment of large data centers where level of service guarantees are important, but failures frequent, and where clients of services can be short-lived.

First, we presented alternatives to the approach of $LH*_{RS^{P2P}}$ that work by pushing information on the file state only to active clients. Relatively small changes in the scheme were shown to have reduce the possibility of interactions taking too long by double forwards. In this part, we update clients at the LH∗ level. Our second contribution shows how to solve the problem of changes in bucket addresses, i.e. at the TCP/IP level, through pushing information only to active clients. Both contributions show that LH∗ is a cloud-ready data structure that is of light weight and presents a strict key-value pair architecture.

References

1. Birman, K.P.: Reliable Distributed Systems: Technologies, Web Services, and Applications. Springer, Berlin (2005)
2. Chabkinian, J., Schwarz, T.: Fast LH*. In: 2013 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 57–64. IEEE (2013)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
4. Devine, R.: Design and implementation of DDH: a distributed dynamic hashing algorithm. In: Foundations of Data Organization and Algorithms, pp. 101–114 (1993)
5. Fagin, R., Nievergelt, J., Pippenger, N., Strong, H.R.: Extendible hashing—a fast access method for dynamic files. ACM Trans. Database Syst. **4**(3), 315–344 (1979)
6. Kröll, B., Widmayer, P.: Distributing a search tree among a growing number of processors. In: ACM SIGMOD Record, vol. 23(2), pp. 265–276. ACM (1994)

7. Litwin, W.: Linear hashing: a new tool for file and table addressing. In: Proceedings of the Sixth International Conference on Very Large Data Bases (VLDB), vol. 6, pp. 212–223 (1980)
8. Litwin, W., Menon, J., Risch, T.: LH∗ Schemes with Scalable Availability. Tech. rep., IBM Almaden (1998). RJ10121 (91937)
9. Litwin, W., Menon, J., Risch, T., Schwarz, T.J.: Design issues for scalable availability LH* schemes with record grouping. In: Second Workshop on Distributed Data and Structures, pp. 38–55 (1999)
10. Litwin, W., Moussa, R., Schwarz, T.: LH∗RS—a highly-available scalable distributed data structure. ACM Trans. Database Syst. (TODS) **30**(3), 769–811 (2005)
11. Litwin, W., Neimat, M.A., Schneider, D.: RP*: A family of order preserving scalable distributed data structures. In: Proceedings of the International Conference on Very Large Data Bases (VLDB), pp. 342–342 (1994)
12. Litwin, W., Neimat, M.A., Schneider, D.: RP∗: a family of order preserving scalable distributed data structures. In: Proceedings of the International Conference on Very Large Databases (VLDB), vol. 94, pp. 12–15 (1994)
13. Litwin, W., Neimat, M.A., Schneider, D.A.: LH∗: linear hashing for distributed files. In: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93), pp. 327–336. ACM (1993)
14. Litwin, W., Neimat, M.A., Schneider, D.A.: LH∗—a scalable, distributed data structure. ACM Trans. Database Syst. **21**(4), 480–525 (1996)
15. Litwin, W., Schwarz, T.: Algebraic signatures for scalable distributed data structures. In: Proceedings of the 20th International Conference on Data Engineering, 2004, pp. 412–423. IEEE (2004)
16. Litwin, W., Yakouben, H., Schwarz, T.: LH∗ RS P2P: a scalable distributed data structure for P2P environment. In: Proceedings of the 8th International Conference on New Technologies in Distributed Systems, p. 1. ACM (2008)
17. Metzner, J.J.: Efficient replicated remote file comparison. IEEE Trans. Comput. **40**(5), 651–660 (1991)
18. Nardelli, E.: Distributed k–d trees. In: Proceedings of the 16th Conference of Chilean Computer Science Society (SCCC96), pp. 142–154 (1996)
19. Rathi, A., Lu, H., Hedrick, G.E.: Performance comparison of extendible hashing and linear hashing techniques. In: Proceedings of the 1990 ACM SIGSMALL/PC Symposium on Small Systems, SIGSMALL '90, pp. 178–185 (1990)
20. Yakouben, H., Soror, S.: LH* RS P2P: a fast and high churn resistant scalable distributed data structure for P2P systems. Int. J. Internet Technol. Secur. Trans. **2**(1), 5–31 (2010)