LH*$_{RS}^{P2P}$: A SCALABLE DISTRIBUTED DATA STRUCTURE FOR THE P2P ENVIRONMENT

Extended Abstract

Hanafi Yakouben[1] , Witold Litwin[1] & Thomas Schwarz[2]

Abstract:

LH*$_{RS}^{P2P}$ is a Scalable Distributed Data Structure (SDDS) designed for P2P applications. It stores and processes data on SDDS peer nodes. Each node is both an SDDS client and, actually or potentially, an SDDS server with application or parity data or both. The scheme builds up on LH*$_{RS}$ scheme principles. The basic difference is that LH*$_{RS}^{P2P}$ key search requires at most one forwarding message, instead of two for LH*$_{RS}$. This property makes LH*$_{RS}^{P2P}$ the fastest P2P addressing scheme known at present. It is also likely to possess the fastest possible key search for an SDDS. The LH*$_{RS}^{P2P}$ scan search also has an upper limit of rounds of two. LH*$_{RS}^{P2P}$ parity management reuses the LH*$_{RS}$ Reed Salomon erasure correction scheme to deal efficiently with churn. The file transparently supports unavailability or withdrawal of up to any $k \geq 1$ peers, where k is a parameter that can scale dynamically with the total size of the data stored. We discuss the LH*$_{RS}^{P2P}$ design, some implementation issues and variants, as well as the related work.

1. INTRODUCTION

The concept of a Scalable Distributed Data Structure (SDDS) appeared in 1993 [4]. It was intended for multicomputers and more specifically for networks of interconnected workstations. Some SDDS nodes are *clients*, interfacing to applications. Others are *servers* storing the data in buckets and addressed by the clients only. The data are either application data or the parity data for a high-availability SDDS such as LH*$_{RS}$ [8]. Overloaded servers split, migrating data to new servers to make the file scalable. The first SDDS was the LH* schema that is now popular and exist in several variants and implementations [8, 6, 5]. The scheme has shown that a key search may need at most two forwarding messages (hops) to find the correct server, regardless of the growth of the file. This property makes the LH* scheme and its subsequent variants a very efficient tool for applications requiring fast growing and large files: distributed databases in general, warehousing, document repositories, e.g., for eGov, stream data repositories…

[1] Witold.litwin@dauphine.fr , hanafi.yakouben@dauphine.fr CERIA, Paris Dauphine University

[2] tjschwarz@scu.edu, Santa Clara University

In the early 2000s, Peer-to-Peer (P2P) systems made their appearance and became a popular topic. In a P2P system, each participant node functions as both a data client and a server. Earliest P2P systems implemented file sharing and used flooding of all nodes for every search. To avoid the resulting message storm, structured P2P systems appeared with additional data structures to make search more efficient, [10, 1]. Structured P2P are in fact specific SDDSs under a new brand name. *Dynamic Hash Table* (DHT) based structures are the most popular [3, 9]. The typical number of hops is O (log *N*) where *N* is the number of peers storing the file. Some peers may be *super-peers* performing additional coordination functions.

The coordinator in LH* is a super-peer in this terminology. However, a client is not a peer since it does not store any part of the SDDS file. It can be off-line or on-line without having an impact on the remaining system. In contrast, an SDDS server is designed to be available at all time. An LH* *peer* is a node that contains both a client and a server, at least potentially. In consequence, LH* peers strive to be online always. In this scenario, it becomes reasonable to assume that a server forwards from time to time some meta-data to selected clients. It may always send such data to its local client, but also to a few remote ones. To improve the already excellent LH* routing (typically, but not always one hop), pushing information on bucket splits and merges appears especially useful. Closer analysis that will appear in what follows shows the possibility to decrease the worst case forwarding to a *single* message only.

A typical P2P system suffers not only from temporary unavailability of some of its constituent nodes, but also from *churn*, the continuous leaving and entering of machines into the system. For example, a P2P system (a.k.a. a desktop grid) such as Farsite [2], currently in the process of being commercialized, uses the often underused, considerable desktop computing resources of a large organization to build a distributed, scalable file server. Even though participant nodes are under the control of the organization, the natural cycle of replacing old system by new ones will appear to the P2P storage system as random churn. However, an application, whether a file server or a database, needs the availability of all its data, regardless of the fate of some participant nodes. It is necessary then to have some redundancy of stored data. LH*$_{RS}$ responds to the database high-availability needs with a specific parity calculus. It is done over so-called *groups* of size $n = 2^i$ of its application data buckets. The value of n is arbitrary; thought in practice should be like 8 – 32. The scheme tolerates the unavailability of up to any $k \geq 1$ of servers per group. The value of k may scale dynamically. The calculus uses a new variant of Reed Salomon erasure correcting coding. The storage overhead is small, in the orders of k / n. These properties seem attractive to deal with reasonable amounts of churn as well.

All these observations are the rationale for the LH*$_{-RS}^{P2P}$ design. We now present it more in depth.

2. PEER ARCHITECTURE

We assume that the reader is familiar with LH*$_{RS}$ as presented in [8]. Every LH*$_{RS}^{P2P}$ *peer* node has a *client* component (the *client* in what follows), that interacts with applications. It also has the *server* component that carries, or is waiting to carry, a data or a parity bucket. Every peer joining the file is expected to store some data, as part of service for the peer community. As for LH*$_{RS}$, the server at a peer can be a *data server*, called simply a *server* in what follows, managing a data bucket. Likewise, it can be a *parity server* managing a parity bucket. A new peer finally can be a *candidate* peer. It acts as the client and serves as a hot spare for a bucket. The situation occurs when there is no pending storage need when the new peer appears.

The client acts as an intermediary between an application and the data servers as in LH*$_{RS}$. This includes the LH hash functions and the Image Adjustment Messaging (IAMs). It has also additional functions we detail below. The data servers are the only to interact with the parity servers, basically during the updates. The data and parity servers behave basically as for LH*$_{RS}$, with additional capabilities we present soon.

Every server peer informs its client component of each new split or merge. It transmits simply the new value of its bucket level *j*. It also sends this information to the parity peers of its reliability group. Finally, it may send it to selected candidate peers. We say that the peer is then a *tutor* to its *pupils*. All the client components that receive the info adjust their images accordingly, using the usual LH* IAM algorithm. Merges are rarely implemented; we do not deal more here with this case.

3. FILE EVOLUTION

A peer wishing to join the file contacts the LH*$_{RS}^{P2P}$ coordinator. The coordinator adds the peer to its location tables and checks whether there is a pending request for the bucket space. This is typically not the case. The coordinator declares then the peer to be a candidate and assigns it a *tutor* of which the candidate will become a *pupil*. To do this, it uses the IP address of the candidate or any seemingly random number determined by the candidate's identity as its key. It then executes the current LH hash function on this key using the current file state (*i,n*). The resulting bucket address is that of the tutor. The coordinator then sends the message to the tutor that in turn contacts the candidate. In particular, it sends its bucket level *j* and the locations of the servers it knows about. We recall that these are the locations of buckets 0… $a + 2^{j-1}$ for a bucket *a*. The pupil stores the addresses, adjusts its image, starts working as a client and acts as a spare, waiting for a bucket to be needed.

A candidate that gets a bucket may get either a data or a parity bucket. In both cases, the peer informs the tutor that it is no longer its pupil. The tutor takes note of it. On the other hand, when the bucket at the tutor splits, then the pupil might get assigned to a new tutor. Namely, to the one that carries the newly split-off bucket. This happens if the pupil's address gets hashed after the split to the new bucket. If this happens, then the new tutor informs all its pupils of the change of assignment. Each pupil updates its metadata to reflect

the change. The scheme flexibly distributes the load of tutoring to all the available servers. The load is as uniform as LH allows. Therefore, the file can handle efficiently many pupils. The number of pupils can grow dynamically with the file size, as one would expect for some applications.

4. RECORD MANIPULATION

The record addressing, key search, scan, record insert, update and delete operations for LH*$_{RS}^{P2P}$ are these of LH*$_{RS}$. With the addition of the operation relating to tutoring, the split operation is also the same. We can now prove the following properties under the assumptions that all data buckets involved are currently available.

Property 1. The maximal number of forwarding messages for the key search is one.

Property 2. The maximal number of rounds for the scan search is two.

Property 3. The worst case addressing performance of LH*$_{RS}^{P2P}$ as defined by Property 1 is the fastest possible for any SDDS or a practical structured P2P addressing scheme.

Proof of Property 1.

Assume that we are addressing a peer a using the client image (i', n'). Assume further that a did not receive any IAM since its last split using h_{i+1}. Hence, we have $i' = j-1$, $n'=a+1$. At the time of the split this image was the file state (i, n). Any search for key coming to peer a before next split, i.e., of bucket $a + 1$, has no forwarding. Let us suppose now that the file evolved further. The first case in our case distinction is depicted in

1. **Figure 1a**. Some buckets with addresses beyond a has split but the file level i did not change, i..e. the split pointer n did not come back to $n = 0$. We thus have $n > n'$ and $i' = i$ still. The figure shows the corresponding addressing regions. Suppose now that peer a searches for key c. Let a' be the address receiving c from a. If a' is anywhere beyond [n', n], then the bucket level j must be the same as when a has created its image. Hence, there cannot be any forwarding of c. Otherwise, a' is the address of the bucket that split in the meantime. The split could move key c to new bucket beyond $2^{i'} - 1$. One forward is thus possible for the search. But this bucket could not yet split again. No other forwarding can occur.

2. Consider now the second case of addressing regions illustrated at Figure 1b. Here, the split pointer n came back to 0 and some splits occurred using $h_{i'+1}$. However, peer a did not yet split again, i.e., $n \leq a$. Peer a searches again for c and sends the search to peer a'. There are three cases:

   a. We have $n \leq a' \leq a$. A bucket in this interval did not split yet using $h_{i'+1}$. Hence the bucket level $j(a')$ is as it was when the image at peer a was created. There cannot be any forwarding, since $j(a') = i' + 1$ while $n' = a$.

   b. We have $a < a' < 2^{i'} - 1$. We are in case 1 above and there can be one forwarding message.

c. We have $a'<n$. Bucket a' split using $h_{i'+1}$ hence we may have a forwarding of c towards bucket $a' + 2^{i'}$ this split has created (one of blue buckets at the right side of the figure. That bucket could not split yet again. Bucket a would need to split first since $a < 2^{i'}$. Hence, there cannot be 2nd forwarding of c.

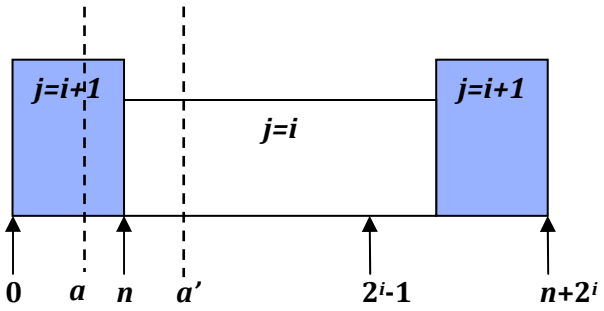

Figure 1a. Addressing regions in case 1

These are the only possible cases for LH*$_{RS}^{P2P}$. Hence, there cannot be more than one forwarding during any key search. If peer a got any IAM in the meantime, then its image could only become more accurate with respect to the actual file state. That is n' becomes closer to n or 'i' becomes perhaps new $i = i' + 1$ for **Figure 1b**. No additional forwarding is possible, only the single forwarding becomes less likely.

Proof Property 2.

Consider that peer a as above now issues a scan search. It sends it to every peer a' it has in its image. The file situation came be as at **Figure 1a** or **1b** above. In each case, peer a' could split once. It has to forward the scan to its descendant. These messages constitute the second round. No descendant could split in turn. They have to be beyond bucket a itself which would need to split first. Hence, two rounds is the worst case for the LH*$_{RS}^{P2P}$ scan.

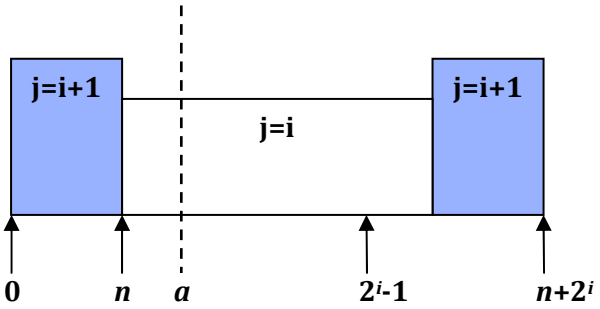

Figure 1b. Addressing regions in case 2

Proof Property 3.

The only better bound is zero messages for any key search. The peer *a* that issues a search can be any peer in the file. To reach zero forwards for any peer *a* would require to propagate synchronously the information on a split to **every** peer in the file. This would violate the goal of scalability, basic to any SDDS. The same restriction stands for structured P2P systems. Hence no SDDS, or practical structured P2P scheme can improve the worst case addressing performance than LH*$_{RS}^{P2P}$.

5. CHURN MANAGEMENT

LH*$_{RS}^{P2P}$ manages the churn through the LH*$_{RS}$ management of unavailable data and parity buckets. In more detail, it recovers up to *k* data or parity buckets found unavailable for any reason, in a single bucket group. Globally, if *K* is the file availability level, *K* = 1,2… then *k* = *K* or *K* = *k* +1. In addition, LH*$_{RS}^{P2P}$ offers to a server or parity peer the possibility of quitting with notice. The peer notifies then the coordinator and stops dealing with incoming record manipulation requests. The data or parity bucket at the peer is transferred elsewhere. Any bucket in the bucket (reliability) group gets the new address. The quitting peer is finally notified of the success of the operation. The whole operation should be faster than recovery, but implies additional waiting for the quitting peer.

A candidate peer may always leave without notice. When its tutor, or the coordinator, does not receive a reply to a message, it simply drops it from its related tables.

It may finally happen that a peer appears to be unavailable for a while, e.g., because of a communication failure or an unplanned system shutdown, and that its data is recovered elsewhere. The new address is not posted to other existing peers. When the peer becomes again available, another peer might not be aware of the recovery and send a search to the "former" peer. To prevent this somewhat unlikely possibility of an erroneous response, we consider two types of search. The usual one does not prevent the occurrence of this type of error. In contrast, the *sure search*, new to LH*$_{RS}^{P2P}$, prevents this scenario. A sure search query triggers a message to one of the parity servers of the bucket group. These have the up to date addresses of all the data buckets in the group. The parity server getting the sure search inquiry informs the outdated peer about its (outdated) status. It also provides it with the correct peer address. The outdated peer avoids the incorrect reply, resending the query to the correct peer. This one replies instead, with the IAM piggybacked.

6. IMPLEMENTATION

Our current implementation consists basically in adding tutoring functions to the LH*$_{RS}$ coordinator and data server components. Likewise, in extending the client component with the pupil's status related functions. The overall picture of new function seems simple.

A specific issue is how to provide *k*-availability to the tutoring data at each data server, i.e., to the addresses of the pupils. The approach at present is to create a dedicated LH*$_{RS}^{P2P}$ record within each data and parity bucket, called the *tutoring record*, with a (unique) key

and these addresses in the non-key part. The keys of the tutoring records form a subset in the key space that is forbidden to applications. For instance, for the integer key space $0...2^{32}$-1, the keys for a tutoring record at data bucket l could be $2^{32} - l$-1. A 1M node file would need for the tutoring records about 0.5/1000 of the key space. All the tutoring records have furthermore the same rank r that is $r = 0$ at present. We recall that r is the key of all the k related parity records under LH*$_{RS}$ scheme, with our version of Reed-Salomon based erasure correcting encoding [8]. The tutoring records are then recovered together with any data or parity records under LH*$_{RS}$.

7. VARIANTS

Up to now we have sketched the basic version of LH*$_{RS}^{P2P}$. A first interesting variant is where a peer joins the file only when the coordinator creates the data or parity bucket. Then, every peer is from the beginning a data or parity server. There is no candidate peers hence no need for the tutoring function neither.

Another variant, steams from the observation that, ideally, each peer should uniformly provide the data access, data storage, and parity management services. In LH*$_{RS}^{P2P}$, like in LH*$_{RS}$, the parity buckets are in fact more loaded. For a bucket reliability group of size n, each parity bucket has to support on the average n time more updates than a data bucket in the group. Typically, this should not be a problem since searches dominate updates. But it could be a handicap in an update intensive file. Likewise, a parity bucket has typically more records than a data bucket. The number of parity record is indeed the maximal number of records in a data bucket of the group.

In [8], we sketched a variant of LH*$_{RS}$ free of these shortcomings. Basically, the records in the same parity bucket were spread instead over parity buckets on nodes of the next group. The algorithm was designed so that no parity records of the same record group could land in the same bucket, assuming however $k \leq n$. An additional important benefit was the recovery speed up to almost n times faster, since performed at n nodes in parallel. The scheme can be the basis for an interesting variant of LH*$_{RS}^{P2P}$ as well.

8. CONCLUSION

LH*$_{RS}^{P2P}$ is intended for peer nodes, where every node both uses data and serves its storage, or is at least willing to serve the storage when needed. Under this assumption it is potentially, with respect to messaging, the fastest SDDS & P2P addressing scheme known. It should also protect efficiently against churn. Current work consists in the implementation of the algorithm over the existing LH*$_{RS}$ prototype [7]. We add the above discussed local communication of the bucket level from the server towards the client at a peer during the splitting, the similar messaging towards the parity buckets of the group, and all the tutoring functions.

Acknowledgments: partial support for this work came from EGov-Bus IST project, number FP6-IST-4-026727-STP.

9. REFERENCES

[1] Adina Crainiceanu, Prakash Linga, Johannes Gehrke, and Jayavel Shanmugasundaram. *Querying Peer-to-Peer Networks Using P-Trees. In Proceedings of the Seventh International Workshop on the Web and Databases (WebDB 2004). Paris, France, June 2004.*

[2] Bolosky W. J, Douceur J. R, Howell J. *The Farsite Project: A Retrospective. Operating System Review, April 2007, p.17-26*

[3] Devine R. *Design and Implementation of DDH: A Distributed Dynamic Hashing Algorithm, Proc. Of the 4th Intl. Foundation of Data Organisation and Algorithms –FODO, 1993.*

[4] Litwin, W. Neimat, M-A., Schneider*, D. LH*: Linear Hashing for Distributed Files. ACM-SIGMOD Int. Conf. On Management of Data, 93.*

[5] Litwin, W., Neimat, M-A., Schneider, D. *LH*: A Scalable Distributed Data Structure. ACM-TODS, (Dec., 1996).*

[6] Litwin, W., Neimat, M-A. *High Availability LH* Schemes with Mirroring, Intl. Conf on Cooperating systems, Brussels, IEEE Press 1996.*

[7] Litwin, W. Moussa R, Schwarz T. *LH*rs- A Highly Available Distributed Data Storage. Proc of 30th VLDB Conference, Toronto, Canada, 2004.*

[8] Litwin, W. Moussa R, Schwarz T. *LH*rs- A Highly Available Scalable Distributed Data Structure. ACM-TODS, Sept 2005.*

[9] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. *Scalable, Distributed Data Structures for Internet Service Construction, Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*

[10] Stoica, Morris, Karger, Kaashoek, Balakrishma. *CHORD : A scalable Peer to Peer Lookup Service for Internet Application. SIGCOMM'O, August 27-31, 2001, San Diego, California USA*