Highly Available Distributed RAM (HADRAM): Scalable Availability for Scalable Distributed Data Structures

Damian Cieslicki, Stefan Schäckeler, Thomas Schwarz, S.J.
Department of Computer Engineering
Santa Clara University
500 El Camino Real, Santa Clara, CA 95053
dcieslicki@engr.scu.edu, sschaeck@engr.scu.edu, TJSchwarz@scu.edu

Abstract

We propose that the challenges in the design and implementation of an SDDS can be significantly eased by separating the design of the scalable high-availability part (HADRAM) from the design of the SDDS proper. We have designed and partially implemented a HADRAM system that allows measurement to prove the validity of the concept. All other highly available SDDS provide failure tolerance at the record level, whereas HADRAM provides it at the memory level. This should speed up design of highly available SDDS.

1 Introduction

With the advent of high-speed, high bandwidth computer networks, clusters of computers, also known as multicomputers, have emerged as a technology with excellent performance / cost ratio. These systems need new data structures that have performance independent of the number of nodes in the multicomputer. A number of Scalable Distributed Data Structures (SDDS) was developed, most prominently LH* and RP* that use the collective RAM (or sometimes disk storage) of the nodes to provide seemingly unlimited storage at attractive speeds. As the number of nodes increases, availability and reliability of nodes emerges as a new problem. If for example a system uses 100 nodes each available at the five nines level (up 99.999% of time), then the combined system is only available at the three nines level (99.9%). In response, failure tolerant (or to be more precise: unavailability tolerant) SDDS were developed such as LH^*_g, LH^*_m, LH^*_{RS}. The latter even implements scalable availability, that is, the level of protection of data against node unavailability grows with the number of nodes to achieve a high system-wide availability of all the data. We propose here Highly Available Distributed Random Access Memory (HADRAM) as a layer that provides scalable available data storage in the combined RAM of the multicomputer, on which arbitrary SDDS structures can be build. In a manner of speaking, what Boxwood [M&al05] proposes for distributed disk storage, we propose for SDDS.

2 Scalable Distributed Data Structures

An SDDS stored data in the distributed RAM of the nodes of a multicomputer while offering access to these data in constant time independent of the number of nodes over which the data is spread. For this reason, they cannot have a central look-up scheme, which would turn into a bottleneck if the system grows too much. However, it is possible to have a central coordinator, as long as the coordinator is only rarely invoked. When the SDDS file (the collection of the data administered by the SDDS changes size) then data is moved to new nodes or data is evacuated from some nodes and moved to others. A good SDDS design will only move data sparingly so that data can be found quickly. For this reason, clients usually do not have an accurate picture of where data is, but the SDDS have a mechanism that allows updating the client's picture so that the same mistake is not made twice.

Most SDDS implement a simple interface to any application that uses them. They tend to store data in records, indexed by a key, and provide access (insert, delete, update, read) by key as well as parallel search. An SDDS is implemented as a middleware (Figure 1). Clients run an application which shares an

SDDS file. The clients access the files by sending their requests to the local SDDS client, which then uses the SDDS protocol to forward the request eventually (but usually directly) to the correct SDDS server running on one node of the multicomputer. The SDDS server itself stores the data in an SDDS bucket. It is possible that SDDS clients and SDDS servers reside on the same node, and that SDDS servers administer more than a single SDDS bucket. The designer / implementer of an SDDS thus has a complicated design that includes the design of the client interface, the design of the SDDS including the addressing algorithm, the movement of data when the SDDS file grows or shrinks, the update of the local image of the state of the SDDS at the client, etc, the design of the communication between clients and servers, and the design of the SDDS bucket. When adding the requirement of high availability, especially scalable availability, then the challenges are indeed great.

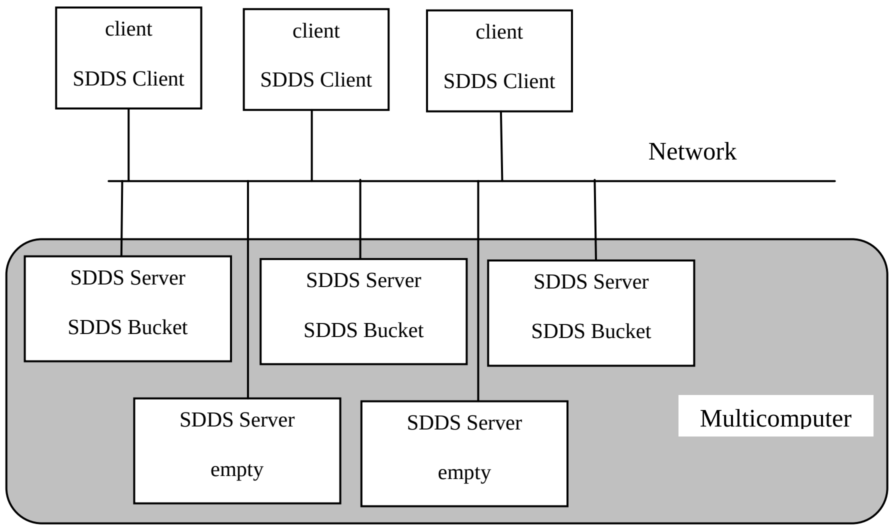

Figure 1: SDDS as Middleware

3 Related Work

High-speed, high-capacity networks (including the Internet) enable cluster computing. To be successful, multicomputer need to provide *service properties* that include the ability to scale to large, rapidly growing user populations, high availability in the face of partial failures, consistency maintenance of users' data, and operational manageability. In response to these needs, Distributed Scalable Data Structures (SDDS) were proposed in 1993, [LNS93], SDDSs gave rise to important research effort, materialized by several algorithms, papers and implementations. It was shown that SDDS files are able to scale to thousands of sites, and terabytes in distributed RAM, with constant access performance, and search times under a millisecond. Multi-key searches requiring an hour or so in a traditional file, e.g., a k-d file, may succeed in less than a second in an SDDS file [LN95], [L96]. All these properties are of prime importance especially in the DBMS design arena, [ASS94]. Some SDDS are:

SDDSs for hashed files. These extend the more traditional dynamic hash data structures, especially linear hashing, [L80], [SPW90] and dynamic hashing [L88], to multicomputers [D93], [KLR96], [LNS93], [LN96], [VBWY94]. The basic algorithm for such SDDSs is LH* has found recognition in the computer literature [K98], [R98].

SDDSs for ordered files. These extend the traditional ordered data structures, B-trees or binary trees, to multicomputers, [LNS94], [KW94].

SDDSs for multi-attribute (multi-key) files. These algorithms extend the traditional k-d data structures [S89] to multicomputers, [LN95], [N95], [L96]. Performance of multi-attribute search can improve by several orders of magnitude.

High-availability SDDSs. These structures are designed to transparently survive failures of server sites. They typically apply principles of mirroring, or of striping, or of grouping of records, revised to support the file scalability [LN95], [LN96], [LR97], [LMR98][LMS05].

There are several implementations of the SDDS structures. An interesting implementation of high-availability variant of LH* implemented in Java is presented in [L98]. Finally, [R98] discusses an application of another variant of LH* to telecommunication databases.

The Boxwood project [M&al05] at MS Research explored the feasibility and utility of providing high-level abstractions or data structures as the fundamental storage infrastructure fulfilling basic requirements such as redundancy and backup to tolerate failures, expansion mechanisms for load and capacity balancing, and consistency maintenance in the presence of failures. Boxwood targets a multicomputer with locally attached storage. Each node runs the Boxwood software components to implement abstractions and other services on the disks attached to the system. Failure tolerance results from using chained-declustered replication [HD90]. We make Boxwood's central thesis our own and apply it to primarily RAM based SDDS.

4 Highly Available Distributed RAM (HADRAM)

Storing data in RAM is fast, even over a network, but also expensive. As we need to store data redundantly in order to recover from node failure or node unavailability, we therefore need to use the less storage intensive redundancy offered by erasure correcting codes. There are a large number of erasure correcting codes available and their properties are well documented. In our scheme, we place *m* buckets on *m* different nodes in a *reliability group* and add to these *m data buckets k* additional buckets – the *parity buckets* – containing *parity data* calculated from the *m* data buckets. This is the same redundancy mechanism as in LH*$_{RS}$ (provided we use a generalized Reed-Solomon code), but in contrast, HADRAM treats buckets as a big, flat chunks of main memory without any internal organization.

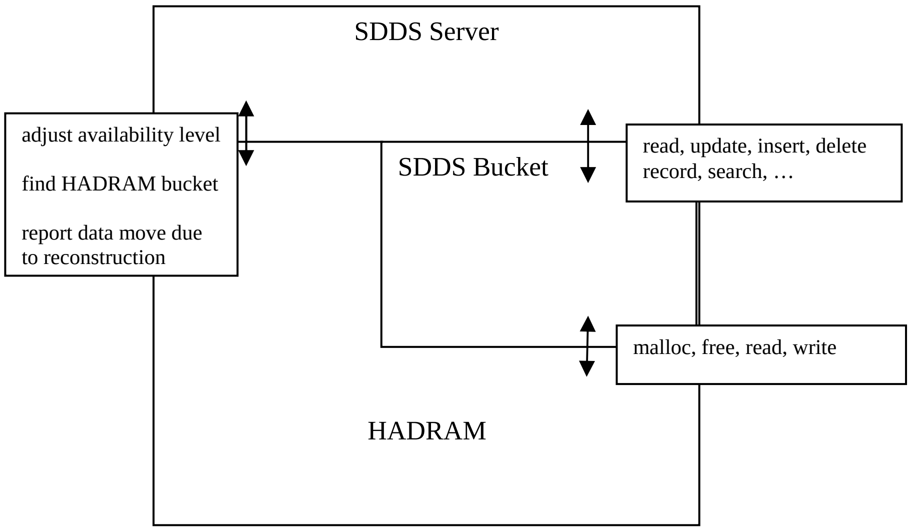
SDDS Server
adjust availability level
find HADRAM bucket
report data move due to reconstruction
SDDS Bucket
read, update, insert, delete record, search, …
malloc, free, read, write
HADRAM

Figure 2: HADRAM – SDDS Server Layers at a Node.

HADRAM is designed to be the highly available memory layer of an SDDS. It provides memory allocation and de-allocation as well as failure tolerance and simultaneous atomic writes. The SDDS server stores its data in a bucket on HADRAM provided memory. The bucket can be organized in any way the SDDS designer chooses, a Linear Hash file, a B+-tree or a link tree, etc. (In the spirit of modularization, the design of the SDDS bucket can be "off-the-shelf".) When the bucket operation demands more memory, it requests it from the HADRAM layer through a malloc-like call, when the memory is no longer needed, then it frees it by a call to the HADRAM layer that operates like the "free"-operation. The SDDS server can request an adjustment of the availability provided by the HADRAM layer.

To detect unavailable buckets, the $n = m + k$ buckets in a HADRAM reliability group monitor each other. If they detect node unavailability, they collectively react to it. The central part of the reaction is to rebuild the lost data on another node in the SDDS. If a parity bucket is lost, then rebuilding it is all that needs to be done and the occurrence is transparent to the SDDS layer. In order to find a place for the replacement bucket, HADRAM can contact a HADRAM coordinator or can use a broadcast message. If a data bucket is lost, then the SDDS layer needs to be involved since requests need to be redirected from the unavailable to the new data bucket. We summarize the interaction between the SDDS and the HADRAM layer in Figure 2. The bulk of the SDDS Server – HADRAM interaction are memory interactions such as allocate HADRAM memory, free HADRAM memory, read HADRAM memory, and write HADRAM memory. These need very little explanation besides the HADRAM design that supports them (Section 5). Note however that an operation such as a record insert changes many different memory locations. For example, if we insert in to a LH bucket implemented according to Ellis [E87] and the insert results in an LH bucket split, then we reset quite a number of pointers and possibly deallocate and allocate memory. For performance reasons, these changes should be processed by the HADRAM system in a single, atomic step; since changes in a HADRAM bucket need to be processed at all *k* parity buckets and involve at least one network round-trip. Because the SDDS server stores data in a local HADRAM data bucket, all reads become local and only writes incur network delays.

The high-availability interactions impose changes in the design of the SDDS server, which we now describe. If an SDDS file grows or shrinks, it needs to adjust the degree of availability of individual buckets. HADRAM can increase the availability by adding a parity bucket and decrease it by releasing a bucket. It can also change it by combining two HADRAM reliability groups $((m + k) + (m + k) \rightarrow (2m + k))$ or splitting them. However, this also changes the performance of writes, which need to percolate to all parity buckets, which in turn see an increase or a decrease in write requests. In addition, it involves the SDDS layer in locating a group to merge with. The SDDS can communicate its need for a change in reliability by giving a command to HADRAM.

The HADRAM layer needs to allocate replacement buckets in case of unavailability. In addition, if the SDDS creates a new data bucket, it needs to either find a slot in an existing HADRAM reliability group or create a new one. Depending on the needs of the SDDS, we might have to provide these services through a central HADRAM coordinator.

5 HADRAM Design

As we have seen, HADRAM implements redundant, distributed storage in the main memory of a cluster of commodity computers. We implemented its basic interchange with the SDDS bucket and in particular investigated a messaging architecture that provides transactional updates even in the presence of (limited) message loss.

5.1 HADRAM Reliability

Internally, HADRAM data buckets are placed in reliability groups to which parity buckets are added. The size of the data buckets is quite large, so that reconstruction of unavailable buckets is done in bulk. In order to provide transactional updates to the local SDDS bucket, HADRAM allows many writes to be grouped in a single transaction, even in the presence of failures. Bundling not only speeds up communications, but also simplifies the memory interface design. A write request to the HADRAM layer is simply a list of offsets within the HADRAM bucket and a list of modifications. HADRAM guarantees that all the modifications are performed simultaneously and atomically. It provides two types of acknowledgements, a first one that the request was received by the HADRAM layer and a second one that the request was performed and is permanent in the sense that it survives failures and unavailabilities (within the limits of our recovery capacity, i.e. not more than *k* unavailabilities within a reliability group).

HADRAM itself is responsible for failure/unavailability detection and repair. Operationally, it detects the failure of parity buckets when they do not respond to an update request. In addition, parity buckets monitor the data buckets through "heartbeat monitoring". In case an unavailable bucket is detected, the HADRAM buckets in a reliability group act in conjunction. We can use a distributed consensus protocol such as Paxos [GM02] or simply a leader election algorithm, though the latter can run into difficulties if messages can be severely delayed.   The group (or the leader) first decides which buckets are available and which are unavailable. It then determines which buckets need to be reconstructed and finds locations for the new buckets. It also determines which (typically) *m* are implied in the reconstruction. The reconstruction itself depends on the erasure correcting code. We use a generalized Reed-Solomon code as in [LMS05]. Accordingly, the group leader first inverts an *m* by *m* matrix formed from the code's generator matrix. For each bucket to be reconstructed, the process needs to calculate the XOR of the Galois field product of a symbol from each of the reconstructing buckets as a symbol of the reconstructed bucket. As an improvement over LH^*_{RS}, we use extensive, and if necessary staggered table look-up to construct the products. In consequence, the leader first calculates the tables and ships them to the sites about to store the reconstructed bucket. All buckets involved in reconstruction then send their contents (in large slices, as in [LMS05]) to the sites which recalculate the bucket.

5.2 *HADRAM Messaging Architecture: Transactions Even in the Presence of Failures*

In HADRAM every modification of a block is an update and needs to be performed at the data bucket and at all the parity buckets for sure. In [LMS05], the well-known 1-Phase-Commit (1PC) [S79, BG83, CS84] was implemented, but unfortunately, if messages can get lost then 1PC does not guarantee that a once acknowledged update cannot be inadvertently rolled by a bucket failure and following reconstruction. Litwin et al. propose two variants of 2-Phase-Commit that trade off transactional behavior even in the presence of failures for speed. We found that another protocol, called *high-watermark*, maintains the transactional quality while offering much better performance. In this protocol, we maintain all updates in a log. We purge entries from the log by applying the update to the contents of the bucket. This happens only if the site knows that all other sites have the update in their log. They gain their knowledge through a series of *status update messages* that in part are piggy-backed on other messages. In more detail, all updates are identified by the data bucket from which they come and a message number. A data site maintains the status of all parity buckets and a parity bucket maintains the status of all buckets. They exchange status exchange messages periodically (i.e. every *l* updates), but a data bucket also sends its status and what *it* knows about the status of all parity buckets whenever there is an update request. When an unavailability is detected, then all surviving buckets send each others status and exchange update requests that were not delivered. It turns out that an update message can only be not processed if a total of *k* messages or sites are lost.

6 Experimental Results

Implementing HADRAM as an abstraction for SDDS-memory is a difficult task and we have only implemented it partially. In particular, we implement the messaging structure and have shown that watermarking because of delayed acknowledgements is preferable to 1PC and 2PC, unless the system needs to acknowledge early on to the client. Figure 3 left shows the result of such an experiment where we pitch 1PC, 0PC (no acknowledgements) and watermarking with status exchange messages every 10 messages against each other. Table 1 gives these times in more explicit form.

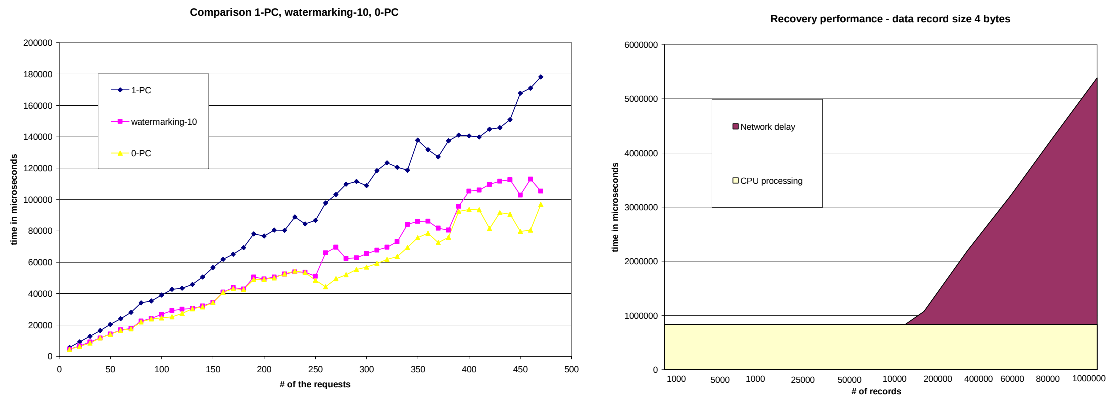

Figure 3: HADRAM performance. Messaging (left), Recovery (right)

We also tested the speed of operations, in particular updates and recovery (Figure 3 right). Our experimental platform is quite flexible and better measurements and a greater depth of explanation will be included in the final version of this paper.

Model type	**Time [μseconds]**
1PC	260
2PC	1050
Watermark (100)	100

Watermark (40)	233
Watermark (20)	251
Watermark (10)	274

Table 1: Update Times

References

[ASS94] Amin, M., Schneider, D., and Singh, V., An Adaptive, Load Balancing Parallel Join Algorithm. 6th International Conference on Management of Data, Bangalore, India, December, 1994.

[BG83] P. Bernstein, N. Goodman: The failure and recovery problem for replicated databases. ACM Symposium on Principles of Distributed Computing, Montreal, Canada, 114-122, 1983.

[CS84] M. Carey, M. Stonebraker: The performance of Concurrency Control Algorithms for Database Management Systems, VLDB, 1984.

[D93] Devine, R. Design and Implementation of DDH: Distributed Dynamic Hashing. Int. Conf. On Foundations of Data Organizations, FODO-93. Lecture Notes in Comp. Sc., Springer-Verlag (publ.), Oct. 1993.

[E87] Ellis, C. S.: Concurrency in linear hashing, ACM Transactions on Database Systems, 1987.

[GM02] Chockler, G. and Malhki, D.: Active disk paxos with infinitely many processes. In Proceedings of the 21st ACM Symp. on Principles of Distributed Computing (PODC-21), 2002.

[G96] Gray, J. Super-Servers: Commodity Computer Clusters Pose a Software Challemge. Microsoft, 1996. http://www.research.microsoft.com/

[HD90] H. Hsiao and D.J. DeWitt. Chained declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Conference on Data Engineering*, pages 456-465, February 1990.

[K98] Knuth D. The Art of Computer Programming. 3rd Ed. Addison Wesley,1998.

[KLR96] Karlsson, J. Litwin, W., Risch, T. LH*lh: A Scalable High Performance Data Structure for Switched Multicomputers. Intl. Conf. on Extending Database Technology, EDBT-96, Avignon, March 1996.

[KW94] Kroll, B., Widmayer, P. Distributing a Search Tree Among a Growing Number of Processors. ACM-SIGMOD Int. Conf. On Management of Data, 1994.

[L80] Linear Hashing : a new tool for file and tables addressing. Reprinted from VLDB-80 in READINGS IN DATABASES. 2-nd ed. Morgan Kaufmann Publishers, Inc., 1994. Stonebraker , M.(Ed.).

[L88] Larson, P. Dynamic Hash Tables, CACM, 31 (4), 1988.

[L98] Lindberg., R. A Java Implementation of a Highly Available Scalable and Distributed Data Structure LH*g. Master Th. LiTH-IDA-Ex-97/65. U. Linkoping, 1997, 62.

[LNS93] Litwin, W. Neimat, M-A., Schneider, D. LH* : Linear Hashing for Distributed Files. ACM-SIGMOD Intl. Conf. On Management of Data, 1993.

[LNS94] Litwin, W., Neimat, M-A., Schneider, D. RP* : A Family of Order-Preserving Scalable Distributed Data Structures. 20th Intl. Conf on Very Large Data Bases (VLDB), 1994.

[LN95] Litwin, W., Neimat. k-RP* : a Family of High Performance Multi-attribute Scalable Distributed Data Structure. IEEE Intl. Conf. on Par. & Distr. Systems, PDIS-96, (Dec. 1996).

[LN95] W. Litwin, M-A Neimat. LH*s : a high-availability and high-security Scalable Distributed Data Structure. IEEE Workshop on Research Issues in Data Engineering. IEEE Press, 1997.

[LN96] Litwin, W., Neimat M-A. High-Availability LH* Schemes with Mirroring. With M-A, Neimat. Intl. Conf. on Coope. Inf. Syst. COOPIS-96, Brussels, 1996. [LNS96] Litwin, W., Neimat, M-A., Schneider, D. LH*: A Scalable Distributed Data Structure. ACM Transactions on Database Systems ACM-TODS, (Dec. 1996).

[LR97] Litwin, W. Risch, T. LH*g : a high-availability Scalable Distributed Data Structure through record grouping. U-Paris 9 Tech. Rep. (May, 1997). Submitted for publ.

[LMR98] Litwin, W. Menon, J., Risch, T. LH* Schemes with Scalable Availability. IBM Almaden Research Rep. (May, 1998).

[LMS05] Litwin, W., Moussa, R., Schwarz, T.: LH*$_{RS}$ – A Highly-Available Scalable Distributed Data Structure, Transactions on Database Systems (TODS), September 2005.

[L96] Lomet, D. Replicated Indexes for Distributed Data to app. in IEEE Intl. Conf. on Par. & Distr. Systems, PDIS-96, (Dec. 1996).

[M&al05] MacCormick, J., Murphy, N., Narjok, M., Thekkath, C. and Zhou, L.: Boxwood: Abstractions as the Foundation for Storage Infrastructure,

[N95] Nardelli, E. Distributed Searching of Multi-dimensional Data: A Performance Evaluation Study. Journal of Par. & Distr. Computing 49, 11-134, 1998.

[R98] Ronstrom, M. Design and Modelling of a Parallel Data Server for Telecom Applications. Ph.D, Thesis U. Linkoping, 1998, 250.

[S89] Samet, H. The Design and Analysis of Spatial Data Structures.

[SPW90] Severance, C., Pramanik, S. Wolberg, P. Distributed linear hashing and parallel projection in main memory databases. VLDB-90.

[VBWY94] Vingralek, R., Breitbart, Y., Weikum, G. Distributed File Organization with Scalable Cost/Performance. ACM-SIGMOD Int. Conf. On Management of Data, 1994.