Efficient Updates in Highly Available Distributed Random Access Memory

Damian Cieslicki Stefan Schäckeler Thomas Schwarz
Department of Computer Engineering, Santa Clara University, Santa Clara, CA 95053
damian.cieslicki@us.abb.com, sschaeck@engr.scu.edu, tjschwarz@scu.edu

Abstract

*With increased network speeds and throughputs, multicomputers (a system of computers connected by a high-speed network) have become an attractive alternative to store important data in their collective random access memory. Erasure codes provide space-optimal data redundancy to protect this type of storage from node unavailability. They have been used in LH*RS, the scalable high availability, distributed version of Linear Hashing. We present and evaluate a technique that uses the property of linear erasure correcting codes to make updates transactional and concurrent with recovery from one or more node availabilities without locks or two-phase commits. The technique significantly improves on previous work in update speed and also allows for serializable updates to a bucket that is in the process of being recovered.*

1. Introduction

As network speeds increase, multicomputers (systems of computers connected by a high-speed network) offer unbeatable price/performance ratios. In particular, we can use their collective RAM to store large amounts of data. Distributed RAM is about 60 – 70 times faster to access than disk drives (i.e. has access times around 100 msec) and can grow practically unlimited by adding nodes. Unfortunately, a large system is likely to suffer from unavailable nodes, so that we need to store data redundantly. The relative high cost of distributed RAM favors the much more space-efficient erasure correcting codes to generate this redundancy. Systems such as LH*$_{RS}$ [MLS04, LMS05] and HADRAM [CS04, CSS06] implement distributed RAM. The former is a Scalable Distributed Data Structure (SDDS) [SDDS] with scalable high availability; the latter provides a layer for implementing scalable high availability for any SDDS.

We present in this paper a general mechanism for fast updates and fast reconstruction to recover from node unavailability. We make progress over LH*$_{RS}$ [LMS05] by providing much faster small updates. We achieve this by using the fact that parity information only needs to cohere when it is used for reconstruction, though of course incoherence between any combination of data and parity data prevents successful recovery. Consequentially, we use lazy acknowledgments of parity updates during normal operations. We log updates and only purge the log when the update has percolated to all parity sites. Basically, we are looking for the "low watermark", therefore the name of our scheme. Before a reconstruction, we use the logs in order to bring all data and parity sites to the same state.

2. Related Work

Using erasure codes for redundancy is a standard method in disk based storage. RAID Level 5 [C+94] uses the *m*+1 simple parity code and RAID Level 6 a number of two-erasures correcting codes. Stonebraker [SS90] realized the benefits of physically separating the data in a reliability group. For larger, disk-based storage systems, erasure coding is also used. For example, Xin et al. [XMS04] show the importance of fast recovery for data availability in a very large storage system. Aguilera et al. [AJX05] propose a family of algorithms that allows concurrent updates and recovery operation. Basically, in their scheme, recovery falls to a client. Once a block has failed, a new block is created in its stead, with invalid data. A client that recovers that block locks it and then proceeds as normal. To insure that the other m blocks used for recovering are consistent, a list of past writes is kept and periodically garbage collected. Our scheme uses a similar list, but reconstruction is done at the server, consistent with the SDDS. Goodson et al. [G+04] use erasure-coding to store a single object redundantly over a number of sites, just as in IDA [Ra89]. While they are also concerned about good performance, their main contribution is using cryptographic hashes, a type of versioning and quorums to deal with Byzantine failures. Frølund et al. [F+04] investigate erasure coding for Federated Arrays of Bricks (FAB). They use a quorum protocol to achieve strict linearizability, something we do not do, though their method could easily be implemented in our setting. Using erasure coding becomes popular in other areas, such as grid-computing. Pitkanen et al. [P+06] report on an implementation of erasure-code protected storage for the grid. Similarly, Kubiatowicz et al. [K+00] propose erasure-coding for OceanStore. Myriad [C+02] realizes higher availability in a large disk-based system similar to us by using RS-coding and reliability groups. The paper shows that the hardware savings of erasure coding lower in effect the total cost of ownership. Myriad uses 2PC to keep updates consistent. Gallersdorfer and Nicola [GN95] improve performance in a replicated database by relaxing coherency constraints, a method that is not directly applicable to our problem.

Our basic design looks very similar to distributed, shared memory [Esk]. However, distributed shared memory operates at the operating system level whereas we describe an application designed to run over any operating system. In addition, distributed shared memory enables generic workloads, whereas our work is more interested in database applications.

3. Overview of Erasure Coding

For ease of reading, we briefly review linear erasure coding. A *m/n* systematic, linear, maximum distance separable erasure code [MS78] takes *m* data blocks and adds to them *k* parity blocks for a total of $n = m+k$ blocks such that all contents can be reconstructed from only *m* of the *n* blocks. To calculate the contents of the parity (a.k.a. redundant) blocks, we calculate with symbols – bit strings of length f with $f = 8$ or $f = 16$ being the more popular choices. The set of all possible symbols form a Galois field $GF(2^f)$. Its arithmetic follows the same rules as for the better known fields made up by the real or complex numbers. Indeed, it happens that addition and subtraction are both the bit-wise exclusive-or (XOR) operation. Assume that $X_1, X_2, \ldots, X_m$ are the contents of the data blocks written as equal-length column vectors with symbols as coordinates. The contents $X_{m+1}, X_{m+2}, \ldots, X_n$ of the parity blocks are then calculated as

$$(X_1, X_2, \ldots X_m, X_{m+1}, X_{m+2}, \ldots X_n) = (X_1, X_2, \ldots X_m)\cdot\mathbf{G}$$

with a *generator matrix* $\mathbf{G} = (g_{ij})$ with coefficients g_{ij} in $GF(2^f)$ and the properties that the first *m* rows of **G** form an identity matrix and that any *m* by *m* submatrix formed by selecting any *m* rows from **G** is invertible. If we have *m* data or parity blocks, let's relabel them $(Y_1, Y_2 \ldots Y_m)$, and if we create a submatrix **H** of **G** from the corresponding rows, then

$$(Y_1, Y_2 \ldots Y_m) = (X_1, X_2 \ldots X_m)\cdot\mathbf{H}$$

and since **H** is invertible, we can recover the data buckets as $(X_1, X_2 \ldots X_m) = (Y_1, Y_2 \ldots Y_m)\cdot\mathbf{H}^{-1}$ and then recalculate the parity blocks. Calculating the parity buckets later makes sense in our context since we need to satisfy read requests in parallel with recovery. Otherwise, we can recover data and parity blocks simultaneously as

$$(X_1, X_2 \ldots X_m, X_{m+1}, X_{m+2} \ldots X_n)\cdot\mathbf{H}^{-1}\cdot\mathbf{G}.$$

For recovery, we need to access the blocks, multiply them coordinate-wise with an element of the *recovery matrix* $\mathbf{H}^{-1}$ and then XOR the products together.

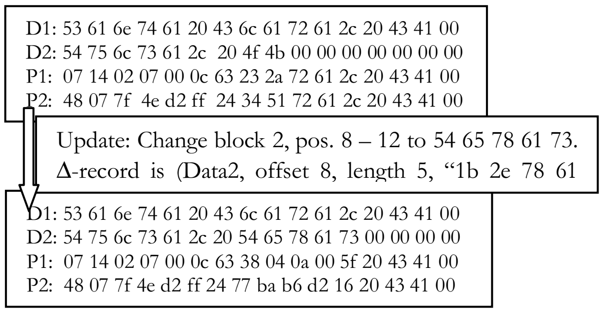
In our setting, we update a single data block at a time. Let that be block X_i that is changed to X'_i. Write $\Delta = X'_i - X_i$. A simple calculation then shows that parity block X_j changes to $X'_j = X_j + g_{ij}\Delta_i$. Thus, to update, we create the delta-value as the (XOR-) difference between the old and the new data value and send it to all sites storing parity blocks. There, we multiply the Δ block with a coefficient of **G** and add the result to the current parity block. Obviously, but quite important for us, parity updates commute. We only need parity blocks when we reconstruct data blocks, but then data and parity data needs to be consistent for a successful recovery.

4. LH*$_{RS}$ and HADRAM

LH* [LNS96], the distributed version of linear hashing, stores records consisting of key and a non-key field in buckets based on a hash of the key. The number of buckets and hence the hash function changes with the number of buckets, but clients do not have direct access to this number in order to avoid the bottleneck of a centralized addressing scheme. Clients might make an addressing mistake (because their view of the file state is not the accurate one) but they are guaranteed not to make the same mistake twice and they are guaranteed that their requests reach the intended bucket typically in a single hop, sometimes a double hop, and only rarely a triple hop. LH* thus offers access times to the records that are independent of the file size. However, as the number of buckets and hence the number of servers employed increases, an LH* file becomes more likely to suffer from node unavailability. For this reason, LH*$_{RS}$ places the LH* buckets (now called the *data buckets*) into Reliability Groups (RG) of size *m* and adds to each reliability group *k= n − m parity buckets*. The parity buckets contain parity records. A record in a bucket in an RG belongs to a *record group* made up of at most one record in each bucket. The records in the parity buckets belong to one record group and have an enumerator (*the rank*) as the key, a list of the keys of the data records as a second field, and a parity field calculated from a linear *m/n* erasure correcting code of all the non-key fields in the data buckets in the record group. When a data bucket fails, LH*$_{RS}$ first finds *m* available buckets in the RG. It then recovers each data record from the *m* available records in the record group. The key comes from the list of key of any parity record, whereas the non-key field is recovered with the *m/n* code. LH*$_{RS}$ reconstructs individual records for a waiting client before reconstructing a lost data bucket. LH*$_{RS}$ also increases *n* when the file becomes larger, thus guaranteeing constant minimum availability of the complete file. Details of LH*$_{RS}$ are in [LMS05].

Highly Available Distributed Random Access Memory (HADRAM) [CSS06] provides scalable high-availability to any SDDS. It stores user data in blocks (of size several MBs at least) in the distributed RAM of the multicomputer and just as LH*$_{RS}$ groups several of these data HADRAM blocks (H-blocks) into reliability groups and then adds parity blocks to the reliability group. All blocks in a RG are stored on different nodes of the multicomputer. In difference to LH*$_{RS}$, an H-block is flat, unstructured memory.

LH*$_{RS}$ as an SDDS stores data in the manner of a relational database table, but typically with simple transactions. We can therefore expect the writes (inserts, updates) to be that of a single record. The only exception arises from a bucket split that moves about half of the data in a bucket to a new bucket. We are not concerned with the split operation in this paper, but assume that all writes are small. Our method still handles very large writes, but performance would suffer. HADRAM as a layer under an SDDS inherits the same performance characteristics.



```
D1: 53 61 6e 74 61 20 43 6c 61 72 61 2c 20 43 41 00
D2: 54 75 6c 73 61 2c  20 4f 4b 00 00 00 00 00 00 00
P1:  07 14 02 07 00 0c 63 23 2a 72 61 2c 20 43 41 00
P2:  48 07 7f  4e d2 ff  24 34 51 72 61 2c 20 43 41 00
```

Update: Change block 2, pos. 8 – 12 to 54 65 78 61 73.
Δ-record is (Data2, offset 8, length 5, “1b 2e 78 61

```
D1: 53 61 6e 74 61 20 43 6c 61 72 61 2c 20 43 41 00
D2: 54 75 6c 73 61 2c 20 54 65 78 61 73 00 00 00 00
P1:  07 14 02 07 00 0c 63 38 04 0a 00 5f 20 43 41 00
P2:  48 07 7f 4e d2 ff 24 77 ba b6 d2 16 20 43 41 00
```

Figure 1: Update of a HADRAM block.

Updates in both structures follow the pattern set by the “Small Writes” in RAID Level 5. To be more precise, a client sends an update to a data site. In the case of LH*$_{RS}$, the update is to a record, in the case of HADRAM, the update specifies an offset in the block and a replacement string. In either case, the data site generates a Δ-record, that consists of the XOR between the new and the old value and metadata such as the record key for LH*$_{RS}$ and the block number and offset for HADRAM. The data site sends the Δ-record to all parity sites. These calculate the new parity data by multiplying the Δ-record with a certain Galois field element (defined by the linear erasure correcting code used) and then XOR it to the old parity data. Figure 1 gives an example for an update in HADRAM.

Update operations at a parity bucket commute. In order to use the parity data for data reconstruction, we need to have applied every update exactly once, but it does not matter in what order. In addition, we can delay parity data update until we need the parity data for reconstruction.

Example

Assume that we have HADRAM blocks organized in a reliability group with two data blocks and two parity blocks. For the encoding, we choose the same code as in [LMS05]. Figure 1 presents the initial contents in hexadecimal notation before and after an update. It happens that the contents in the first parity block is the XOR of the two data blocks, however, the calculation of the second parity block is more involved and involves Galois field multiplication. The first data block contains the string “Santa Clara, CA” whereas the second one the string “Tulsa, OK”. An update now arrives at data site 2, requesting contra-factually to change the state of Tulsa to Texas. Data site 2 calculates the XOR of the old and new value and creates the Δ-record in a compressed version. It then sends the Δ-record to the parity sites, which use it to update their content. The first parity site simply XOR the difference to the substring specified by offset and length, whereas the second parity site has first to multiply each byte in the string “1b 2e 78 61 73” with a certain Galois field element determined by the code and the issuing data site and then XOR the result to its content starting at the offset.

5. Transaction Processing Messaging

A data update (i.e. a record insert, delete, and modification in LH*$_{RS}$; in HADRAM every modification of a block is an update) needs to be performed at the data bucket and at all the parity buckets for sure. A simple scheme is the well known One-Phase-Commit (1PC) [S79, BG83, CS84…], in which the parity site acknowledges all update requests from the data site and perform them immediately. After receiving all acks from the parity site, the data site can commit the update and ack to the client. This simple scheme is not sufficient for two or more parity sites in an RG as the following example shows. Assume that a data site sends a Δ-record to two parity sites, but fails during the send so that only one parity site receives the Δ-record. There is no information left in the system to decide which parity site has the correct value. Using both to recover the data site contents yields neither the new nor the old value, but gibberish. Logging all updates at parity sites (alone) does not help since there is no way to discard old update entries. The LH*$_{RS}$ prototype [LMS05] used 1PC because it assumed a completely reliable network.

An alternative is a variant of Two Phase Commit (2PC), in which the data site sends the delta record to all parity sites in a fixed order. The parity sites place the Δ-record in a queue of all such requests and send an acknowledgement to the data site. The data site waits for all $n-m$ acks from the parity sites to arrive and then sends out a "commit" message to all the parity sites in the same order. The order in this *serial 2PC* protocol allows parity sites with competing views of the state of a lost data site (for example, whether the data site is in phase 1 or has already moved to phase 2) to reconstruct the data site state up to messages sent to also failed parity sites. Its obvious drawback is the much longer time it takes to acknowledge to a client that the update has been performed *for sure*, that is, without the possibility that a subsequent data site failure and reconstruction recreates the contents before the update. This is the only protocol in [LMS05] that achieves transactional behavior of updates in the presence of failures.

Parallel 2PC (in which the data site sends out phase 1 and phase 2 messages to the parity buckets as a multicast) in contrast is much faster. However, if a data site fails in the middle of an update and if messages are lost, then we cannot always successfully restore parity sites to coherence. For an example, consider the following two scenarios: First, assume that the updating data site fails in phase 1, but that only parity site 1 has received the Δ-record. Second, assume that the updating data site fails in phase 2 and that only parity site 2 has received the "commit" message. There is not enough information at surviving parity sites 1 and 2 to decide between these scenarios.
The final protocol investigated (and the best according to our experimental evaluation) uses logging; we call it *low-watermarking* because participants exchange information about which site has received an update and incorporated it *for sure* in order to garbage-collect the log. When a failure is detected, all surviving sites exchange logs to bring each other to the same state. Acknowledgments in this protocol pose an interesting problem. If the data site acknowledges after sending out the Δ-records to all k parity sites, the update is k-available in the sense that it is retained if no more than a total of k messages and sites in the RG become unavailable. If it acknowledges after receiving the (delayed) acknowledgment of the parity sites, then the update is k-available in the sense that the data with the update survives up to k site unavailabilities in the RG.

In more detail, low-watermarking numbers each update arriving at a data site consecutively. By appending this number to the unique identifier of the data site, all updates receive a unique label. (In LH*$_{RS}$, clients might send an update request to another site, but the scheme forwards the update request to the correct data site in at most two additional hops. The correct LH* bucket always knows that it is the correct one handling this update and only it will talk to the parity buckets. Similarly for migrating buckets due to reconstruction of unavailable buckets.)

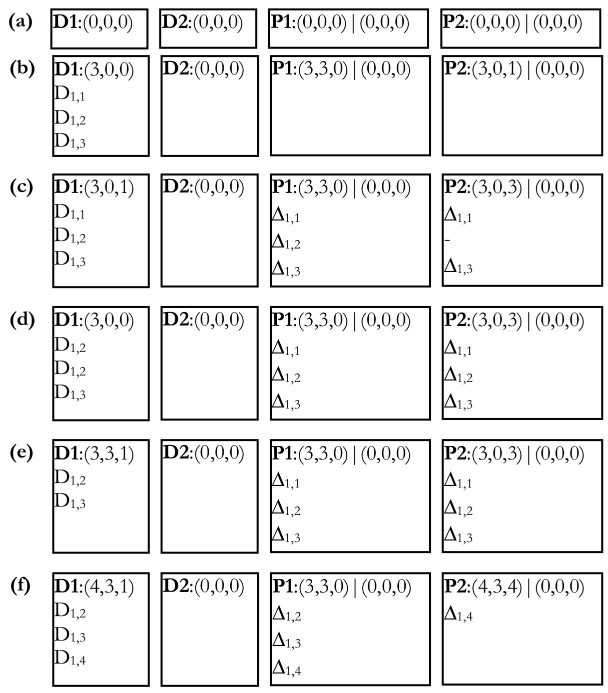

(a)
D1:(0,0,0)
D2:(0,0,0)
P1:(0,0,0) | (0,0,0)
P2:(0,0,0) | (0,0,0)
(b)
D1:(3,0,0) $D_{1,1}$ $D_{1,2}$ $D_{1,3}$
D2:(0,0,0)
P1:(3,3,0) | (0,0,0)
P2:(3,0,1) | (0,0,0)
(c)
D1:(3,0,1) $D_{1,1}$ $D_{1,2}$ $D_{1,3}$
D2:(0,0,0)
P1:(3,3,0) | (0,0,0) $\Delta_{1,1}$ $\Delta_{1,2}$ $\Delta_{1,3}$
P2:(3,0,3) | (0,0,0) $\Delta_{1,1}$ - $\Delta_{1,3}$
(d)
D1:(3,0,0) $D_{1,2}$ $D_{1,2}$ $D_{1,3}$
D2:(0,0,0)
P1:(3,3,0) | (0,0,0) $\Delta_{1,1}$ $\Delta_{1,2}$ $\Delta_{1,3}$
P2:(3,0,3) | (0,0,0) $\Delta_{1,1}$ $\Delta_{1,2}$ $\Delta_{1,3}$
(e)
D1:(3,3,1) $D_{1,2}$ $D_{1,3}$
D2:(0,0,0)
P1:(3,3,0) | (0,0,0) $\Delta_{1,1}$ $\Delta_{1,2}$ $\Delta_{1,3}$
P2:(3,0,3) | (0,0,0) $\Delta_{1,1}$ $\Delta_{1,2}$ $\Delta_{1,3}$
(f)
D1:(4,3,1) $D_{1,2}$ $D_{1,3}$ $D_{1,4}$
D2:(0,0,0)
P1:(3,3,0) | (0,0,0) $\Delta_{1,2}$ $\Delta_{1,3}$ $\Delta_{1,4}$
P2:(4,3,4) | (0,0,0) $\Delta_{1,4}$

Figure 2: Update example

Using watermarking, sites never explicitly acknowledge messages. Rather, they implicitly use bulk acknowledgments by maintaining and exchanging a state that encodes the messages that were received.

The state of data site $\mathbf{D}_i$ contains the last update number u_i that it has seen. For each parity site $\mathbf{P}_j$, it also contains the number of updates $u_i(\mathbf{P}_j)$ that are directed to $\mathbf{D}_i$ and for which $\mathbf{D}_i$ knows that the parity site $\mathbf{P}_j$ has received the corresponding Δ-update. The data state is hence made up of $k + 1$ numbers, where k is the number of parity sites in the RG.

The state of a parity site $\mathbf{P}_j$ consists of n components, one each for each data site. Each state component contains the state of the data site as the parity site knows about it.

States are updated whenever an update, either from a client in the case of a data site or a Δ-record from a data site in the case of a parity site, arrives. They are also updated when sites exchange states. In more detail, assume that a client update arrives at a data site. In this case, the data site increments its count u_i. If a Δ-record arrives from data site $\mathbf{D}_i$ at parity site $\mathbf{P}_j$, then only the component corresponding to $\mathbf{D}_i$ in $\mathbf{P}_j$'s state changes. If the Δ-record has sequence number u and $u \neq u_i+1$, then the parity site knows that it missed a Δ-record and requests are resend from $\mathbf{D}_i$. In any case the count u_i of updates that $\mathbf{D}_i$ has seen (according to $\mathbf{P}_j$'s knowledge) is incremented to the new value u. Similarly, the number $u_i(\mathbf{P}_j)$ of Δ-records from $\mathbf{D}_i$ that $\mathbf{P}_j$ has seen is incremented. All other parts of the state stay the same.

States also change when a site sends it state to another site. Data sites also piggy-back their state to Δ-messages to the parity sites. Parity sites will "spontaneously" send information to a data site, triggered either by counting the number of updates received, by expiration of a timer, or when a parity bucket requests a re-transmit. The latter occurs if the sequence number of a Δ-update more than one more of the parity site's value u_i. In this case the parity site "knows" that it missed a Δ-update and requests a retransmission. If a state is received, then the local state is changed to reflect the new knowledge embedded in the state by setting values to the maximum of the old and the new state.

All data sites log client updates. All parity sites log Δ-records. These entries are purged if – according to the local state – the data site and all parity sites have processed the update and the respective Δ-records.

Example

Assume that we have two data sites, labeled D1 and D2, and two parity sites, P1 and P2, see Figure 2. Data sites number updates they receive. A data site maintains a state (i,j,k), where i is the last update number, j is the last update that it "knows" that P1 has received, and k the last that P2 has received. The parity buckets maintain a state for updates from either data bucket. The original state at each site is (0,0,0) for data sites and (0,0,0)|(0,0,0) for parity sites (Figure 2a). Assume that D1 receives three inserts $D_{1,1}$, $D_{1,2}$, $D_{1,3}$, and creates and sends three Δ-records to P1 and P2. P2 does not receive $\Delta_{1,2}$. The states are updated accordingly (Figure 2b) and each site logs all updates. P2 requests a resend of $D_{1,2}$ and sends its state along. Correspondingly, D1 updates its state to (3,0,1) (Figure 2c) and resends $\Delta_{1,2}$ from its log to P2 (Figure 2d). "Spontaneously", e.g. because a timeout has past since the last state exchange, P1 sends its state to D1, effectively acknowledging the three updates. D1 updates its state, and since $min(3,3,1) = 1$, it can purge its log of update $D_{1,1}$. When update $D_{1,4}$ arrives, it sends Δ-records to the parity buckets together with its state. Accordingly, P1 and P2 update their states and purge their log. P1's state changes to (4,4,1) and it can purge $\Delta_{1,1}$. P2's state changes to (4,3,4) and it can purge $\Delta_{1,1}$, $\Delta_{1,2}$, and $\Delta_{1,3}$.

6. Failure Recovery

In our system, members of a reliability group monitor each other by periodically exchanging states. In addition, a client that does not receive a required acknowledgement from a data site report this unavailability to the coordinator in LH^*_{RS} or to the parity buckets in HADRAM. Once an unavailable site is reported, the members of the reliability group elect a *Recovery Coordinator* (RC) provided that there are at least m of them, otherwise, the unavailability is catastrophic. The RC determines a list of available sites in the RG, finds replacement sites for lost sites and updates the addressing scheme. (We do not address the problem of maintaining bucket addresses in a highly available SDDS here.) The RC then initiates the recovery process. We recover data sites first and then parity sites. While it is possible to recover all sites simultaneously, data sites receive priority since clients only interact with them. To recover the data sites, all participating sites need to be brought first to the same state. Since parallel and serial 2PC effectively serialize updates at the data sites, at most m updates – one for each data site – have to be performed at the parity sites. In the case of watermarking, the potential number of updates is possibly much larger and determined by the number of updates from a data site that do not have to be acknowledged. In parallel to these updates, the recovery coordinator determines the m sites that it is going to use to recalculate the lost data sites. Based on these m sites, the recovery coordinator (which should be located at one of the replacement data sites), pre-calculates the information necessary. In our case, this involves inverting an m-by-m matrix in the Galois field used by the erasure correcting code, and calculating m multiplication tables (of the size of the Galois field, if the Galois field is small ($GF(2^8)$, or several small tables otherwise). These multiplication tables are used to speed up the processing of the data slices coming in from the sites participating in the recovery operation. It then requests slices of the site's data from the *m* recovery sites, uses these to reconstruct slices of the unavailable data sites' contents, and send these slices to the replacement data sites.

After recovering the data sites, the missing parity sites are reconstructed from the *m* data sites. When this is finished, the data sites apply all the updates that have accumulated at them in the normal way and the system has regained its old functionality.

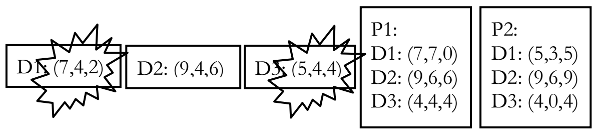


Figure 3: Reconciliation example: Before recovery

Figure 3 gives an example how we use the state information to make parity and data sites consistent. There we need to recover two out of three data sites in a RG using the remaining sites. After exchanging states, the states for D2 are (9,9,9). For D1, P2's state is going to be (7,7,5) showing that updates $D_{1,6}$ and $D_{1,7}$ need to be redone. These updates can be retrieved from either site D2 or site P1. The D3-state at both parity sites is (4,4,4) and no updates need to be retrieved. However, update $D_{3,5}$ has been lost. This loss occurs because we have one site failure and two lost messages, more than our availability level $k = 2$ can support.

If a data site needs to be recovered, the RC initiates a replacement site with the data (the HADRAM block or LH*$_{RS}$ bucket), an initially zero log of updates, and state data. At this point, the replacement site starts to fully function as a data site. In parallel, it recovers the data from the lost site in slices. As the performance measurements in [LMS05] shows, a small slice size leads to poor performance, but for larger slices the recovery times are good and not very dependent on the size of the slice. During recovery, the site satisfies read and write requests as soon as possible.

While HADRAM memory is flat, we implement it as a collection of pages, similar to Virtual Memory. During the recovery process of a data site, we maintain a small data structure for each page that tells us whether the page has been reconstructed, and whether there is a write to it pending. Recovery proceeds by reconstructing page per page, possibly in batches to use larger and hence fewer messages. When a write to one or more pages arrives, the site marks the corresponding pages as dirty. It then requests reconstruction data from the participating sites and blocks until these pages have been reconstructed. If a read arrives, the site checks whether the area to be read has already been reconstructed. If not, it recovers the pages needed out of order, by sending a request to the sites from which it receives the recovery data. When we reconstruct a dirty page, we unblock the write operation(s) that wrote to that page, which now calculates the Δ-record that are sent out to the parity buckets, so that the write proceeds as normal. Reconstruction of a page might also cause a read operation to be unblocked. This then proceeds as normal by sending the requested data to the client application. Basically, the HADRAM procedure is the same as for LH*$_{RS}$ which also recovers requested records with priority, with the one complication that HADRAM writes and reads might be to more than a single page.

The algorithm at a reconstructing parity site is simpler. We basically reconstruct the parity site's data slice by slice and then perform all the updates that we kept in the log. In the unlikely case of another site becoming unavailable during the reconstruction, we restart the process. Since reconstruction can proceed at the rate of a few MB/sec, it is quick.

Clients will see a short interruption of service when a data site becomes unavailable since they address their requests to a site that is down. In LH*$_{RS}$, they contact the coordinator (a central agent) that starts recovery and also handles the request. In HADRAM, a handle to HADRAM memory contains not only the site where the block is stored but also the addresses of the other members of the RG. The client then contacts one of these. Eventually, the client receives the address of the replacement site, which by then has been selected and is either still recovering data or has already entered normal mode.

Our scheme is not designed to handle Byzantine failure. In addition, a very slow data site can be replaced by a replacement site, but still serve read requests. If a client writes to such a site, then the parity sites will inform the slow data site that it has been replaced. However, the slow site can still answer reads until the periodic heart-beat monitoring with the parity sites fail. To increase robustness, we could run a protocol like Paxos [L01] in a RG to maintain a robust view of which sites are currently considered up or down. However, a system that suffers frequent loss of connectivity cannot run HADRAM, since clients will not be able to find their data directly and have to take recourse to broadcasting "where are you" messages.

7. Performance Results

We improve on the parity calculation in [LMS05] by constructing tables on the fly for multiplying a number of elements with a constant. This operation occurs when processing parity data or during recovery, when we multiply the contents at sites involved in recovery by multiplying them with a coefficient in the recovery matrix. We found that the overhead of first creating multiplication tables is amortized when we process more than about 1 MB of data. A typical value for multiplying a whole bucket by a single Galois field element is 1.6 millisecond per MB of data, obtained on a 3 GHz P4 machine. At this range, using tables provides a 2-4% gain over using the logs-antilog method in [LMS05].

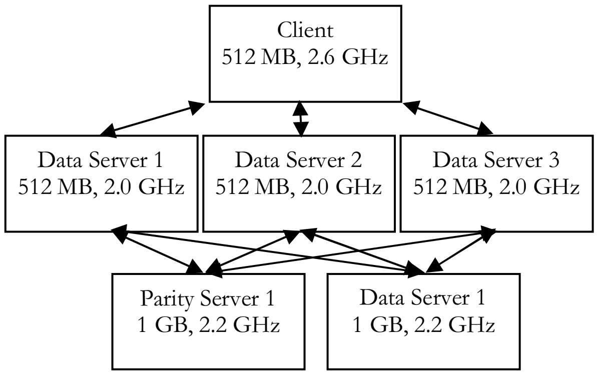


Figure 4: Reliability group implementation

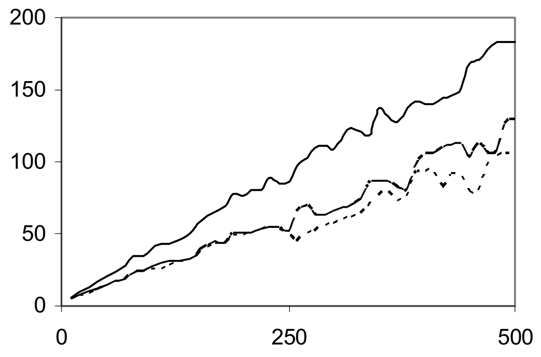


Figure 5: Comparison between response times (msec) for 1-500 messages using 1PC (top), watermark A and watermark B (bottom).

We implemented a test-bed as shown in Figure 4 for our experiments. In our first set of experiments, we measured the speed of updates. We used two different acknowledgment schemes for watermarking, watermarking A where we wait with the acknowledgment to the client until we receive an acknowledgment from all parity sites and watermarking B, where we do not acknowledge to the client but for the very last message. In Figure 5, we present the results of our experiment. The x-axis gives the number of updates and the y-axis the time in milliseconds to complete the task. In 1-PC, the client issues a new request after receiving an acknowledgment. The 1-PC numbers reflect the roundtrip from client to data to all parity and back. For watermarking, the client sends out bulk messages and receive bulk acknowledgments every 10 messages in A. The speed advantage of watermarking B shows the costs of client acknowledgments. We also observe that even though we used a dedicated network, measured times vary by about 5%.

In our next set of experiments, we use watermarking with status exchange every 10 messages and acknowledgment to the client after receipt from all parity buckets. All messages from the data server are subject to random message loss. Basically, whenever we send out a message, the server uses a random number generator to decide whether the message should be sent. As messages are lost, our scheme demands resending. Accordingly, the timing goes up. However, according to our results in Figure 7, even 70% message loss barely doubles the time it takes to process the updates. For a more realistic percentage of losses of 5% and 10%, the differences between the update times are barely discernible.

Regarding recovery of data, we measured the components of the recovery process. We recall that this process consists in leadership election (two rounds of broadcast messages), matrix inversion, exchange of logs to achieve consistency, and the actual recovery process. We use a simple Gaussian algorithm for matrix inversion. The relevant numbers are 9 μsec and 12.8 μsec for inverting an 8 by 8 matrix, needed when there are 8 data servers, and 3 μsec and 4.7 μsec for inverting a 4 by 4 matrix, measured on a laptop (1.5 GHz P4) and one of our 2GHz desktop machines (See Figure 7). The time to exchange the logs is equally short, encompassing a round of messages going back and forth to the participants, but the exact number depends on the size of the log. According to our experience, a typical number is in the range of 50 – 100 μsec. The bulk of the reconstruction work is taken up by transferring the data. As Figure 8 shows, the overall recovery time for a single data bucket is very linear, with an average of 9.11 MB/sec reconstruction speed without background reads. We also tested the system under load by reading (~ 470 reads per second, half of which blocked because the page to be read yet had to be reconstructed). Recall that the Galois field operations function at a much higher throughput, at least 200 MB/sec so that our recovery performance is entirely network bound, as was to be expected. When such a read blocks, then we have to wait an average of

53 µsec, a value that can also be calculated from our recovery data.

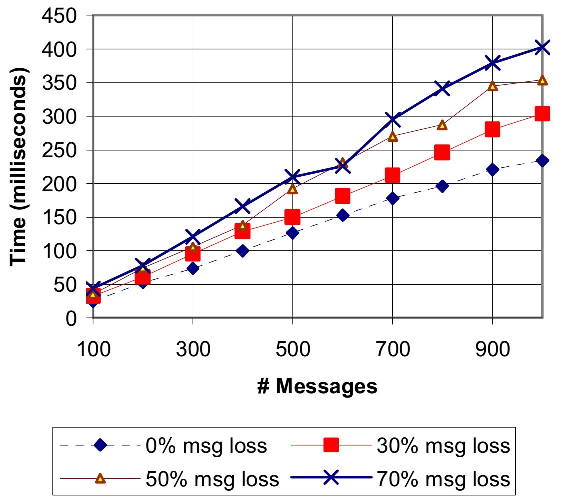


Figure 6: Update speed with message losses.

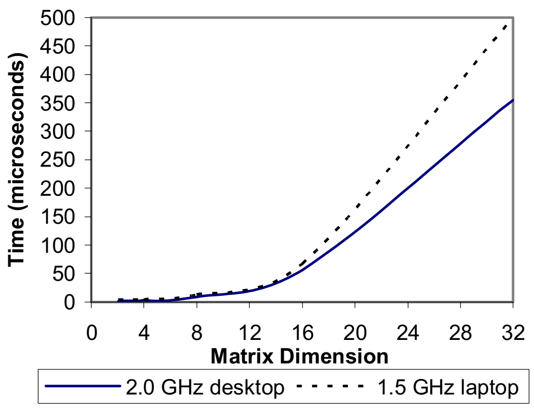


Figure 7: Matrix inversion times.

Read performance during recovery depends on the aggressiveness of reconstruction because giving the recovery thread equal priority with the threads dealing with client requests can lead to the majority of the bandwidth being allocated to recovery. In order to obtain a read value independent of the bandwidth allocated to recovery, we ran an experiment where we only recovered (512B) pages on demand. We obtained an average of 1138 buckets recovered per second, i.e. an average bandwidth of ~.56 MB/sec. The lower number is caused by the additional messaging that is needed including additionally hops between the client and the server. In particular, this means that under normal circumstances, a read to an unreconstructed block takes about 1 msec.

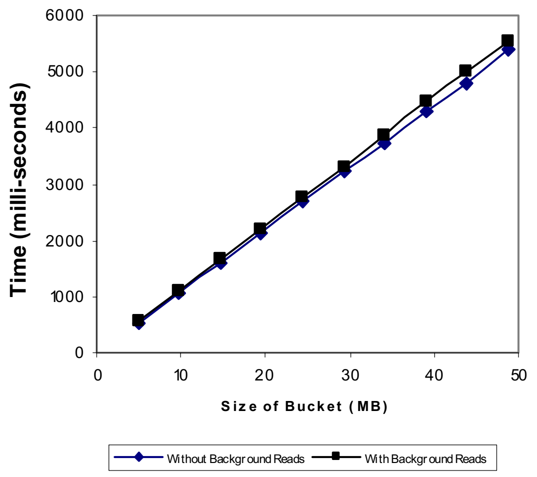


Figure 8: Recovery times with and without background reads.

7. Conclusions and Future Work

We have presented improvements to the update and reconstruction performance of LH*RS, a scalable distributed data structure with scalable availability and to HADRAM, the high-availability layer for generic distributed memory SDDS. These data structures are implemented at the application level and are OS independent. While providing scalable high availability to distributed memory as an OS paradigm might be interesting, we are not pursuing this route. We have shown that two simple principles: "parity updates can be performed out of order", and "parity updates only need to be done before reconstruction" leads to a significant better update and recovery implementation. Our method has the advantage of providing serializability even in the presence of recovery operations. Obviously, our log cannot grow to multiple sizes of the stored data without great loss of performance because of paging; hence we need to add an operation that overwrites complete bucket contents. These operations will only occur in an SDDS during a rare split operation when two new buckets are created to replace an old bucket.

Acknowledgment

We would like to express our appreciation to Jim Gray and Microsoft Research (BARC) for generous financial support and him and Witold Litwin for advice.

References

[AJX05] M. Aguileras, R. Janakiraman, L. Xu: Using erasure codes efficiently for storage in a distributed system. Proceedings 2005 International Conference on Dependable Systems and Networks (DSN05).

[BG83] P. Bernstein, N. Goodman: The failure and recovery problem for replicated databases. ACM Symp. 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.

[C+02] F. Chang, M. Ji, S. Leung, J. MacCormick, S. Perl, L. Zhang: Myriad: Cost-effective disaster tolerance. Proceedings 1st USENIX Conference on File and Storage Technologies (FAST02), 2002.

[C+94] P. Chen, E. Lee, G. Gibson, R. Katz, D. Patterson: RAID: high-performance, reliable secondary storage. ACM Computing Surveys, Vol. 26(2), June 1994, p. 145 – 185.

[CS04] D. Cieslicki, T. Schwarz.: Power control center applications using highly available distributed RAM (HADRAM), Proc., Carnegie Mellon Transmissions Conference, Dec. 15-16 2004, Pittsburgh, USA.

[CSS06] D. Cieslicki, S. Schäckeler, T. Schwarz: Highly Available Distributed RAM (HADRAM): Scalable availability for scalable distributed data structures. In Proc. 7th Workshop on Distributed Algorithms and Structures (WDAS06).

[Esk] R. Eskicioglu: A comprehensive bibliography of distributed shared memory. www.cs.umd.edu/~keleher/bib/dsmbiblio/dsmbiblio.html

[F+04] S. Frølund, A. Merchant, Y. Saito, S. Spence, A. Veitch: A decentralized algorithm for erasure-coded virtual disks. Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN04).

[GN95] R. Gallersdorfer, M. Nicola: Improving performance in replicated databases through relaxed coherency, Proc. 21st VLDB Conference, Zurich, Switzerland 1995.

[G+04] G. Goodson, J. Wylie, G. Ganger, M. Reiter: Efficient Byzantine-tolerant erasure-coded storage. Proceedings 2004 International Conference on Dependable Systems and Networks (DSN04).

[G+00] S. Gribble, E. Brewer, J. Hellerstein, D. Culler: Scalable distributed data structures for internet service construction, Proceedings 4th USENIX Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, CA, October 2000.

[K+00] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao. OceanStore: An architecture for global-scale persistent storage. Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Nov. 2000.

[L01] L. Lamport: Paxos made simple. ACM Sigacts News 32(4):18-25, 2001.

[LNS96] W. Litwin, M. A. Neimat, D. Schneider: LH*˜ A scalable, distributed data structure. ACM Transactions on Database Systems, Dec. 1996.

[LMS05] W. Litwin, R. Moussa, T. Schwarz: LH*RS – A highly-available scalable distributed data structure, ACM Transactions on Database Systems (TODS), Vol. 30(3). 2005.

[MS78] F. MacWilliams, N. Sloane: The theory of error correcting codes. Amsterdam; New York: New York, North-Holland Pub. 1978.

[MLS04] R. Moussa, W. Litwin, T. Schwarz: LH*RS, A highly available distributed data storage system. (Demonstration). Proceedings 30th VLDB Conference, Toronto, Canada, 2004.

[Ra89] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance Journal of the ACM (JACM), Volume 36(2), p. 335-348, April 1989.

[P+06] M. Pitkanen, R. Moussa, M. Swany, and T. Niemi: Erasure codes for increasing the availability of grid data storage. Proc. International Conference on Internet and Web Applications and Services (ICIW'06), February 2006.

[SDDS] SDDS bibliography: ceria.dauphine.fr /SDDS-bibliography.html

[S79] M. Stonebraker. Concurrency control and consistency of multiple copies in distributed INGRES. IEEE Transactions on Software Engineering, SE-5:188-194, 1979.

[SS90] M. Stonebraker and G. Schloss: Distributed RAID - A New Multiple Copy Algorithm. Proc. 6th International Conference on Data Engineering, 430 – 437, 1990.

[XMS04] Q. Xin, E. Miller, T. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. Proc. 13th IEEE Inter. Symposium on High Performance Distributed Computing, Honolulu (HPDC-13), HI, June 4-6, 2004.