Exploring High Performance Distributed File Storage Using LDPC Codes

Benjamin Gaidioz [a] , Birger Koblitz [a] , Nuno Santos [a,*]

[a] *CERN, Geneva, Switzerland*

Abstract

We explore the feasibility of implementing a reliable, high performance, distributed storage system on a commodity computing cluster. Files are distributed across storage nodes using erasure coding with small Low-Density Parity-Check (LDPC) codes, which provide high-reliability with small storage and performance overhead. We present performance measurements done on a prototype system comprising 50 nodes, which are self organised using a peer-to-peer overlay.

Key words: Distributed File Storage, Low Density Parity Check (LDPC) Codes, High-availability, High-performance

1 Introduction

With the growing popularity of Grid and cluster computing, computer clusters built out of cheap commodity computers are becoming commonplace. While their CPU power is readily made available through batch or grid computing systems, the often substantial amount of disk space on the computer nodes is usually not made available for mid or long term storage. In this paper we investigate how to use this idle storage for high performance and reliable file storage, targeted mainly at workloads where files are written once and read often.

Erasure coding has been proposed [1,2] as an alternative to replication for reliable storage of files using unreliable nodes. It consists of splitting the original file in n data blocks plus m coding blocks of equal size, such that the original file can be

* Corresponding author. Tel.: +41 22 76 75710, E-mail address: nuno.santos@cern.ch
Email addresses: benjamin.gaidioz@cern.ch (Benjamin Gaidioz), birger.koblitz@cern.ch (Birger Koblitz), nuno.santos@cern.ch (Nuno Santos).

reconstructed from subsets of the blocks. Compared to replication, erasure coding has a smaller space overhead for similar levels of availability. However, in practice, most of the storage systems used on Local Area Networks rely on replication to ensure reliability. The reason is that traditional erasure coding like Reed-Solomon codes [3] demand a high computational effort, which grows quadratically with n and m, to reassemble the original data out of any n data or coding blocks.

Low-Density Parity-Check codes (LDPC) [4] provide a solution to this problem because they allow to reconstruct the original data using relatively few and cheap XOR operations. They do not, however, code the data optimally (in contrast to Reed-Solomon codes) but require fn blocks to reconstruct the stored file, where $f \geq 1$, while Reed-Solomon codes require only n blocks. The properties of LDPC codes are well understood in the asymptotics of $n \to \infty$ where $f \to 1$, but little is known about how to construct smaller codes ($n, m < 1000$). The discovery of efficient algorithms for creating large LDPC codes ($n > 10000$) with very fast encoding and decoding [5] has lead to a surge in the interest in these codes, in particular for the resilient storage of files on Grid and peer-to-peer networks. In this scenario a file is decomposed into many ($n + m$ large) blocks which are stored in a distributed manner on hosts connected by a Wide Area Network (WAN) [6].

The paper is organised as follows. In Section 3.2, we compare the availability provided by LDPC codes, Reed-Solomon codes and replication. We explain why small LDPC codes ($n, m \approx 10$) fit better the criteria needed for the implementation of a distributed storage system in a commodity computing cluster. These small codes cannot be constructed with standard techniques. We therefore present a way of constructing graphs with good guarantees on their redundancy using Monte Carlo techniques in Section 2. In Section 4, we describe the implementation of a file storage system based on small LDPC codes. Performance measurements are presented in Section 5. The remainder of the paper is a discussion of the implementation and the results obtained so far including references to related work (Section 6) and finally conclusions and a preview of our ongoing and future work (Section 7).

2 LDPC Codes

In the following we will give a brief overview of LDPC codes, for a full introduction see e.g. [7]. An example of a graph describing a simple LDPC code is shown in Fig. 1. From $n = 3$ data words d_1, d_2, d_3 (bits in the simplest case), $m = 2$ coding words c_1, c_2 are calculated by *xor*ing the data words. For example $c_1 = d_1 xor d_2$. Encoding the redundant information in the coding words is done in a time growing linearly with the number of edges in the graph. The data and coding words are then assembled into data and coding blocks, which can be stored in a distributed manner.

The original information of a file can be reconstructed directly from the data blocks

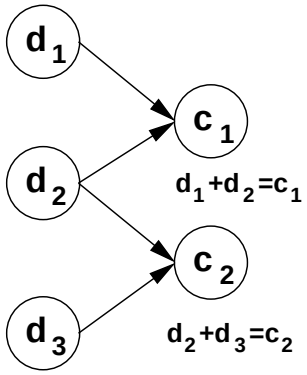

Fig. 1. An example of a systematic graph with $n = 3$ and $m = 2$.

by simply concatenating them. This will be possible in the majority of cases on LANs, which are normally relatively reliable. If data blocks are unavailable then they can be reconstructed from coding blocks using the following algorithm: if for a known coding block all but one of the data blocks from which it has been calculated are known, then the words in that unknown data block are the exclusive or of the corresponding words in the coding block and the known data blocks. By applying this algorithm recursively to the downloaded or reconstructed blocks, the original data can be reconstructed in linear time, if a sufficient number of data and coding blocks is available.

The amount of information encoded using an LDPC graph is the rate $R = n/(n + m)$. Since LDPC codes do not encode optimally, when randomly downloading blocks more than n blocks are needed to reconstruct the original file. The overhead factor f is defined by the average number fn of blocks which need to be downloaded to reconstruct the file, where $f > 1$. In the limit of very large ($n, m \to \infty$), LDPC codes become optimal ($f \to 1$), for small and medium sized codes ($n, m < 10000$) the overhead is typically in the order of 10%, depending on R [8].

2.1 Generating Efficient Codes

The construction of very small ($n \in \{2, 3\}$ and $m \in \{3, 4, 5\}$) optimal codes has been recently done [9] using exhaustive enumeration of all the possible graphs. However, such techniques are not feasible for generating larger codes, like the ones we are interested in using. The alternative is to use Monte Carlo techniques to construct and evaluate these graphs [8].

Using such a Monte Carlo technique we created graphs randomly for a fixed n and m and a probability p for a right-hand node to be connected to a given left-hand node. We used $0.4 < p < 0.6$ depending on n, m based on the findings in [9]. Instead of evaluating the average overhead factor by sampling the necessary overhead for many different downloading sequences of blocks, we compute $f_{\max}$

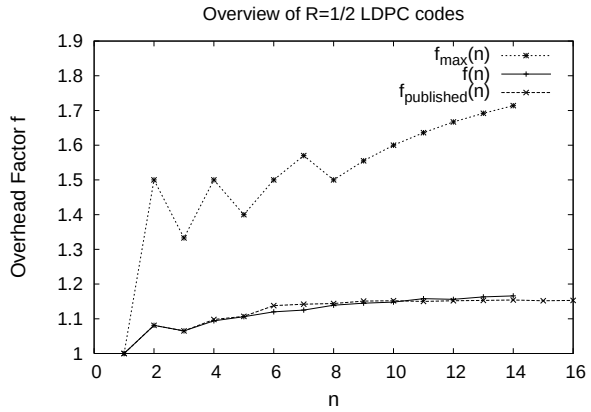

Fig. 2. Average overhead $f(n)$ of best generated codes compared to the overhead of published codes $f_{\mathrm{published}}(n)$ (taken from [8,9]) for $R = 1/2$. In addition the worst-case overhead factor $f_{\max}$ of the best generated graphs is shown.

for the given graph. Calculating the average overhead with s samples in average

$$E = s \prod_{i=n}^{fn} i \tag{1}$$

evaluations whether with the given set of blocks the original data can be reconstructed. The resulting error on f is $f/\sqrt{s}$. Computing $f_{\max}$ requires at most

$$E' = \prod_{i=f_{\max} n}^{n+m-1} i \tag{2}$$

reconstruction tries where $E' < E$ for small graphs ($n \ll s$). Since in practice most generated graphs will be unable to cope with even a very small number of missing coding blocks and $f_{\max}$ is found as soon as a single download sequence fails, the average number of tries is close to $n(n-1)$ (most graphs failing to compensate for 2 missing data blocks in all cases). Fig. 2 shows that the performance of the graphs generated and evaluated in this manner for $R = 1/2$ can compete with the best known graphs for $n < 15$.

3 LDPC Codes For Storage Applications

In this section we discuss the suitability of LDPC codes for storage applications.

3.1 Optimal n and m Values for LAN Storage

Erasure code techniques can be used with very different values for n and m, providing different levels of availability and of system complexity. Larger number of blocks improve the chances of recovering a file in the presence of failures, but increase the overhead to reassemble the file, the storage overhead and the complexity of the system.

Most storage systems designed for WANs assume a large number of very unreliable nodes, like personal computers connected through the Internet. Under these conditions, it will be common for a significant fraction of the file chunks to be unavailable and, therefore, good file availability can only be achieved with high levels of redundancy. For LDPC or Reed-Solomon codes, this means splitting the file in many blocks, possibly hundreds. Another argument for having a large number of blocks is that it allows to better select blocks for downloading based on latency or available bandwidth, which can vary widely on a WAN. More choices equates to potentially better performance.

On a LAN the picture is very different. The node availability is typically much better than on a WAN, making it possible to achieve high-levels of file availability with lower redundancy levels. In addition, the latency and bandwidth are typically homogeneous, which removes the need for selecting "nearby" blocks for optimising file download, as all blocks are equally near. In fact, in the normal case clients will try to reconstruct the file by retrieving n data blocks, which need only to be concatenated, without requiring any expensive decoding. In this case, having a small n is advantageous as it simplifies the management of the system and minimizes the overhead to reconstruct the file, as fewer storage nodes need to be contacted.

For our purposes we chose to $n, m \approxeq 10$, as this provides a good balance between file availability and storage/performance overhead.

3.2 Availability Analysis - LDPC vs Reed-Solomon vs Replication

Replication, Reed-Solomon codes and LDPC codes are three techniques that can be used to achieve high-availability in storage systems. In this section we evaluate the availability provided by each of them and the corresponding cost in terms of storage overhead.

For each redundancy mechanisms considered here, the availability of a file can be expressed by a formula that takes as input the availability of the storage node μ plus some parameters specific to the coding mechanism. A common parameter to all these mechanisms is the stretch factor $S = (n + m)/n$, which indicates the storage overhead.

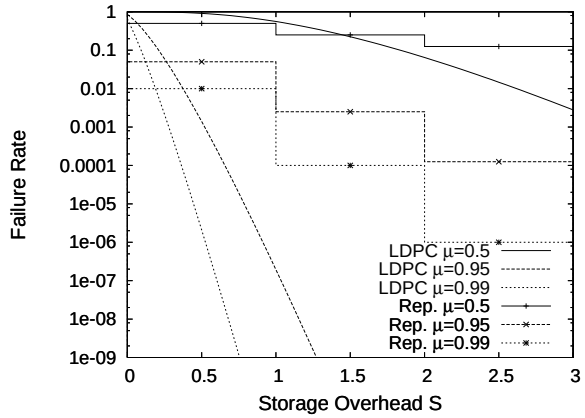

Fig. 3. The average failure rate of files stored redundantly using replication and using LDPC codes versus a given storage overhead. We choose $n = 8$ for the LDPC codes and assume an overhead factor of $f = 1.1$.

The availability of a replicated file is provided by the following equation:

$$A_r(\mu, S) = \sum_{i=1}^{S} \binom{S}{i} \mu^i (1-\mu)^{S-i} \tag{3}$$

For a file encoded using Reed-Solomon codes, the equation is:

$$A_e(\mu, n, m) = \sum_{i=n}^{n+m} \binom{n+m}{i} \mu^i (1-\mu)^{n+m-i} \quad , \tag{4}$$

The rate of the codes is

$$R = \frac{1}{S} = \frac{n}{n+m} \quad . \tag{5}$$

LDPC codes do not code optimally and therefore introduce an overhead factor f. This means they are only able to reconstruct the original file from in average fn chunks, where $f > 1$. Concerning the overhead, LDPC codes are comparable to normal erasure codes with

$$n' = fn \quad \text{and} \quad m' = (1-f)n + m \quad . \tag{6}$$

An upper bound for the availability of an LDPC encoded file can be given by (4) using $f_{\max}$, the maximum overhead of a graph (that is the original data can be reconstructed from *any* $f_{\max} n$ blocks).

Fig. 3 shows a comparison of the failure rate $(1-A)$ of files stored using replication and stored using LDPC for three different node availabilities of $\mu = 0.5$, 0.95 and 0.99. For LDPC, we use $n = 8$ and an assumed overhead factor of $f = 1.1$. For low

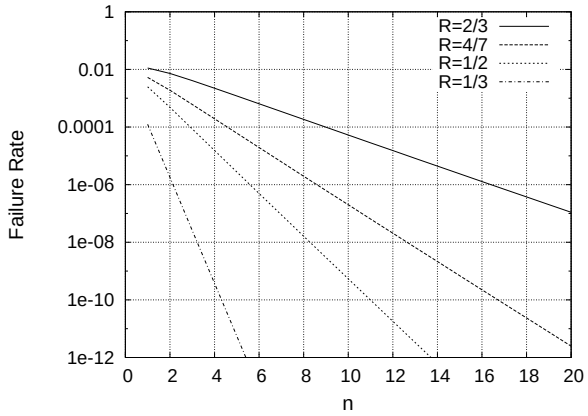

Fig. 4. Failure Rate as a function of the number n of data chunks for different code rates. We assume a node availability of $\mu = 0.95$.

node availability ($\mu = 0.5$), LDPC codes with such small number of coding blocks perform worse than file replication, at least for small storage overhead factors of $S < 1.5$. However for a good availability $\mu = 0.95$ (an estimate for the availability of nodes in a cluster of custom hardware) or even 0.99 small LDPC codes provide a better file availability for smaller overheads. This means that small LDPC codes have the potential to provide small storage overhead and excellent file availability on LANs while introducing only a small networking overhead due to the relatively small number of n parallel downloads.

3.3 Performance of LDPC Codes

In this section, we evaluate the performance of LDPC codes of different rates in order to select good values for m and n. We then use the solution presented in Section 2.1 to generate a graph with good properties. We implement this graph in our storage system prototype and evaluate the overhead of decoding with a varying number of missing data chunks.

Fig. 4 shows the system failure rate provided by four different code rates as a function of the number of data chunks n. Rates like $1/2$ and $1/3$ provide a low failure rate at the price of a high storage overhead. Rate $R = 2/3$ has a low overhead but a high failure rate. For example, with $R = 2/3$, we need $n = 14$ and $m = 7$ for reaching a failure rate of 10^{-6}, while the same availability can be obtained with a $R = 4/7$ (which has almost the same storage overhead) with $n = 8$ and $m = 6$. For our availability goals, rate $R = 4/7$ provides a good trade off in storage overhead and availability. We use $n = 8$ and $m = 6$.

To generate a good graph for the parameters $n = 8$ and $m = 6$, we ran the graph generation algorithm presented in Section 2.1. The resulting graph is shown on Fig. 5. It has the property of tolerating the loss of any three data chunks, that is, the

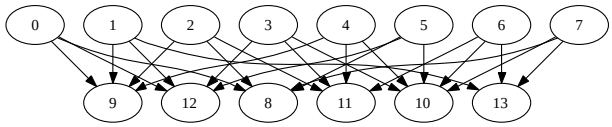

Fig. 5. An example of a graph with $n = 8$ and $m = 6$ which can be reconstructed with any 11 nodes. The average overhead factor is $f = 1.108$.

file can always be reconstructed even if any three data chunks are missing.

Reconstructing a file out of both control and data chunks has some overhead because it requires computations instead of simple concatenation of chunks. We evaluated this overhead by decoding files with variable number of missing chunks. Table 1 shows the data rates obtained when downloading chunks from several servers, as a function of the number of failing data chunks.

Table 1
Performance of reading a 500 MB file as a function of missing data chunks.

missing chunks	time (s)	rate (MB/s)
0	4.79	109.5
1	5.23	100.25
2	5.73	91.5
3	6.47	81.0

The file size is 500 MB. It was encoded using the graph shown on Fig. 5. Since this graph allows to reconstruct a file with up to three missing chunks out of fourteen, we vary the number of missing data chunks from zero to three. The client ran in the same LAN as the servers. We show here the average of 50 downloads.

There is an obvious cost in decoding missing data chunks out of control chunks. However, the performance is still acceptable, even in the worse case of three missing chunks. Considering that the worse case will not occur often in a LAN, the overhead in general will be much smaller.

4 A Prototype Implementation

A prototype for the system presented in the article has been implemented. We give in this section a description of the implementation.

There are two main components in the system: storage servers and a client applications.

Servers The server is run on the storage nodes and is responsible for hosting file

chunks, distributing them to clients (and also to receive them when a client stores a file in the system). We use HTTP both for file transfers and for control messages.

Clients Client applications link against a C++ client library that provides access to the storage system. The library currently provides calls to store and retrieve files. For the file names, we use a flat name space, as we do not intend to implement a file system interface to the system.

For better fault-tolerance we designed the system to be decentralized, using peer-to-peer techniques for organising the storage nodes. There are several freely available Distributed Hash Tables (DHT) implementations, but we decided against using one of these since they are all targeted to WANs. In a LAN the number of hosts and the churn rate are typically much lower, which allows different and more efficient strategies than the ones used in WANs. For instance, DHTs for WANs cannot keep full routing tables at each node, since the cost of maintaining them on a system with a large number of very volatile nodes would be excessive. Instead, they keep tables with a size around O(log N), which allows routing to be done in an average of O(log N) steps. But on a LAN it is feasible to keep full routing tables, thereby achieving one-hop routing. We considered that an efficient routing mechanism is important enough to justify developing a peer-to-peer overlay tailored to a LAN.

In our system, each server is identified by an id obtained by computing a hash on the name of the host it is running on. This implicitly defines an order on servers and provides a Distributed Hash Table that can be used to store file chunks.

When a client is started, it needs to update its local list of hosts in the system so that it will be able to get file chunks from them. For bootstrapping, clients are configured manually with a list of well-known hosts that they contact at startup to obtain an updated list. Clients store their list persistently, so that when they are restarted they can quickly reestablish contact with existing hosts. The list of hosts is kept consistent using a peer-to-peer overlay network.

To store a file, the client first splits it into $n + m$ chunks and assigns to each a name consisting of the original file name plus the index of the chunk. It then computes the hash of each chunk's name and stores the chunk on the server whose identifier is closer to the hash. Using a hash function enables chunks to be randomly distributed in the system. If one of the nodes selected to store the file fails before receiving the chunk assigned to it, our current implementation discards its chunk, but proceeds with the write using the other nodes. If the missing chunks are less than three, clients will still be able to reconstruct the file from the other chunks. This is of course not the ideal solution, since it reduces the redundancy of a file and, therefore, its availability. This is a limitation of assigning chunks to nodes in a static way, which does not reassigning a chunk easily. As future work, we would like to improve our implementation to assign chunks dynamically and to perform reconstruction of lost chunks when nodes fail.

To download a file, the client computes the hash of each chunk's name to determine the hosts storing each chunk. Initially, it tries to download only the n data chunks. Unless some nodes are unreachable or some chunks are not found, these n fragments are sufficient to reassemble the file very efficiently, as they only have to be concatenated. In case of failure, the client selects another chunk for download. This is done by analysing the graph to find one chunk that minimizes the number of extra downloads. It is possible that some control chunks fail, in which case the graph may permit to rebuild them with other chunks. If the available chunks do not permit to reconstruct the file, that is, if there are more than three missing chunks, the call fails.

There are many other details about the storage system, like the reconstruction of lost chunks, load balancing and security, but as this is not the main focus of the article we will not discuss them here.

5 Performance

In this section, we show performance measurements obtained with our prototype implementation. The main goal of these measurements is to have an idea of the overall data rate one can achieve with a system like the one presented here. Deeper studies regarding availability are ongoing work which we intend to implement through simulations instead. This would permit to evaluate the robustness of the system with various failure patterns.

We ran the prototype on two different clusters.

CERN "lxplus" computing farm is a cluster of about 100 dual Xeon 2.8 GHz with 2 GB of RAM with a Fast-Ethernet access to the network. We used 40 of them.

Münster cluster is a set of 50 dual Opteron 2 GHz with 2 GB of RAM. There are interconnected with a Gigabit network.

Each node is running a storage server. We then store files in the system by sending chunks to servers. When the files have been stored, we start a client on each node. Clients download all the files one after the other but following a random order to avoid a situation where all clients start downloading the same file simultaneously, which would skew the tests in an unrealistic way. Each client downloads all the required chunks in parallel by opening TCP connections to each server at the same time.

Both clusters were being used by other users during the tests, with many of the nodes being busy with other computations. This is not a drawback because our system is intended to be ran under such conditions in practice.

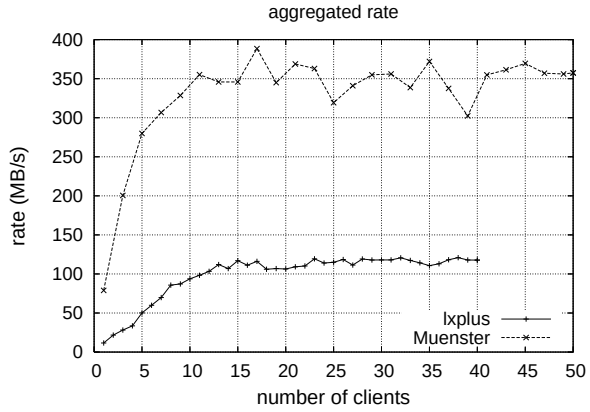

Fig. 6. Aggregated rate obtained on both clusters as a function of the number of client nodes.

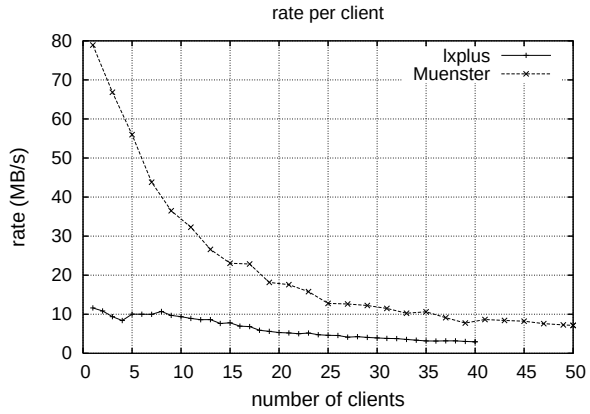

Fig. 7. Rate per node obtained on both clusters as a function of the number of client nodes.

5.1 Measurements

Fig. 6 shows the overall rate measured on both clusters and Fig. 7 the rate per client.

For a large number of hosts, the system hits a limit of 110MB/s on lxplus and 350MB/s in the Münster cluster. This corresponds very likely to the bandwidth of the switch backplane and shows that the use of LDPC coding by itself is not the bottleneck.

For a low number of hosts, although the rate per node decreases when increasing the number of clients, the loss in performance is relatively low compared to the number of nodes. On lxplus, the gain in performance per node is about 9.78 MB/s. On Münster's cluster, the increase in performance per node is about 50 MB/s.

While performing these measurements, we noticed that as the number of clients increases, the increase in the data rate is lower than what should be expected considering the speed of the network.

We believe this is a consequence of different load levels in the storage nodes, with the more loaded nodes slowing down the others. This happens because the TCP connections open by each client are kept synchronized so that the file can be reassembled in memory, thus avoiding the overhead of writing the individual chunks to the local hard drive. If one connection is substantially slower than the others, the others will stall waiting for the slow connection to receive the data needed to reassemble the current part of the file. When the connections finally restart, they have to go through TCP slow start again, resulting in degraded performance. If the slowest connection was able to continue operating at its top speed, this wouldn't be a problem as it is this connection that determines the whole file download speed. But when the fast connections resume the transfer, they saturate the network, forcing the slow connection to drop its speed and go through TCP slow start again. We are investigating the problem but we consider the numbers shown in this section promising. In fact, since we intend to implement this storage system on commodity computing clusters, we expect data rate to be affected more strongly by other factors like other tasks running on the nodes, the disk usage, and network usage. The most critical goal is to ensure high availability of the service.

6 Related Work

Reed-Solomon codes have been used by several storage systems, both for WAN environments (OceanStore [10]) and for LAN (RepStore [11] and FAB [12]). All these systems use erasure coding only for archival storage, since Reed-Solomon codes have a significant performance overhead. For frequently accessed files and for supporting updates, these systems rely on replication. In FAB, the storage space is statically partitioned into pools that are set at creation time to use either replication or erasure coding as redundancy mechanism. On the other hand, RepStore exposes a single storage pool that is divided internally into an hot and cold space; the hot space uses replication and is intended for frequently-accessed data or for performing updates, while the cold space uses erasure coding and is intended for rarely-accessed data. The two spaces are managed dynamically by RepStore, taking in consideration the access patterns. Many other distributed storage systems rely solely on replication, including Gnutella [13], CFS [14] and PAST [15] for WANs, and Petal [16] and the Google File System (GFS) [17] for LANs. In contrast to these systems, we rely solely on erasure coding by using LDPC codes. This is possible due to the LDPC codes' near real-time decoding speed, which allows us to have space efficiency without sacrificing performance.

LDPC codes have not been explored as much for storage applications. The most significant example of their use is the Digital Fountain system [18], where LDPC codes are used for the dissemination of bulk data to a large number of receivers over Wide Area Networks. There is also some recent work [6] that studies the suitability of LDPC codes for Wide Area Network storage. But to our knowledge, there is

no previous work on the use of LDPC codes for storage on a Local Area Network environment, where the focus is on performance and a small storage overhead.

The granularity of storage used by most of the systems mentioned here are files, just like in our system. The only exception is FAB which exposes blocks of storage to provide the abstraction of a virtual hard-drive.

Our system is based on a Peer-to-Peer topology due to its fault-tolerance and scalability properties. For the same reasons, many other storage systems use Peer-to-Peer or decentralised topologies. On the Wide-Area Network some examples include Gnutella, CFS, PAST and OceanStore. Gnutella is based on an unstructured topology, while the other three systems are structured using a distributed hash table. For Local Area Networks, xFS [19], Petal and FAB are all decentralised systems that rely on voting and consensus algorithms for organising their topology. RepStore takes an alternative approach by using a distributed hash table optimized to Local Area Networks. In traditional DHTs each node maintains only partial routing tables with size typically of the order of $O(logN)$, which allows for routing in an average of $O(logN)$ hops. RepStore, on the other hand, keeps full routing tables in every node, thereby achieving one-hop routing. This is more suitable to LANs, where the number of nodes rarely exceeds a few thousands and churn rates are low compared to WANs. In contrast to these systems, our gossiping protocol is more light weight and scalable, having as a drawback the possibility of temporary inconsistencies.

7 Conclusion and Future Work

We have presented a novel architecture for a reliable, high performance, distributed storage system on a commodity computing cluster. Storage of files is based on erasure coding with small Low-Density Parity-Check (LDPC) codes. These codes provide high reliability with a low storage and performance overhead. The main contributions of this paper are:

- an analytic evaluation of the availability provided by LDPC codes versus replication and Reed-Solomon codes,
- a way of constructing small LDPC codes with good guarantees on their redundancy,
- the description of an implementation of a file storage system based on LDPC encoding and performance measurements obtained with it on two different computing clusters of both the overall rate it provides and evaluation of the overhead of decoding.

Compared to other fault-tolerant storage systems that use replication or Reed-Solomon codes for redundancy, our prototype based on LDPC codes has a smaller storage overhead for similar levels of fault-tolerance. In addition, the performance is bet-

ter than in system using Reed-Solomon codes, since LDPC codes are substantially faster.

Availability provided by LDPC encoding techniques makes it a satisfying redundancy schema for the implementation of a storage system on a computing cluster. Our work on generation small graphs allows us to obtain a good availability of the service against possible failures of nodes. The initial performance results are promising.

The work presented here is ongoing work and many interesting details are under study.

- Techniques regarding LDPC codes are still being investigated. We continue our activity of generating good graphs with more sophisticated ways of controlling the probability distribution of the edges in the graphs as proposed in [8].
- So far we presented an analytic evaluation of the availability provided by the use of LDPC codes. In the future we intend to use simulations of the entire system instead, so that we can study various failure patterns, e.g. introduced due to failures in the peer-to-peer overlay.
- The implementation itself is still at an early stage. Using the peer-to-peer overlay it would be also possible for the system to actively recover missing blocks. In fact, given the nature of the coding graphs this would involve only a small number of hosts which have blocks that are related to the missing one.
- There is currently no load-balancing done by the implementation apart from the trivial case that node becomes unavailable due to their load such that blocks are taken from elsewhere. However, the servers could also distribute information about their load through the P2P network and actively reroute clients or initiate further replication.
- Since we intend to use this system in production, in particular on grid computing sites, part of our activity will be dedicated to its integration into grid file catalogues which will also allow to implement access controls.

8 Acknowledgements

This work was performed within the LCG-ARDA project and the authors would like to thank in particular Massimo Lamanna (CERN) for many fruitful discussions on this paper's subject. For the performance measurements a computer cluster at the University of Munster, Germany was used and we are very grateful for the excellent operator support we got. This work was partially funded by grant SFRH/BD/17276/2004 of the Portuguese Foundation for Science and Technology (FCT) and by Bundesministerium fur Bildung und Forschung, Berlin, Germany.

References

[1] W. K. Lin, D. M. Chiu, Y. B. Lee, Erasure Code Replication Revisited, in: 4th International Conference on Peer-to-Peer Computing (P2P 2004), IEEE, Zurich, Switzerland, 2004, pp. 90–97.

[2] H. Weatherspoon, J. Kubiatowicz, Erasure coding vs. replication: A quantitative comparison, in: International Workshop on Peer-to-Peer Systems (IPTPS), Vol. 1, 2002.

[3] J. S. Plank, A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems, Software – Practice & Experience 27 (9) (1997) 995–1012.

[4] R. Gallager, Low-Density Parity-Check Codes, MIT Press, Cambridge, MA, 1963.

[5] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, V. Stemann, Practical loss-resilient codes, in: 29th Ann. ACM Symp. on Th. of Comp., 1997, pp. 150–159.

[6] R. L. Collins, J. S. Plank, Assessing the performance of erasure codes in the wide-area, in: DSN-05: International Conference on Dependable Systems and Networks, IEEE, Yokohama, Japan, 2005.

[7] S. B. Wicker, S. Kim, Fundamentals of Codes, Graphs, and Iterative Decoding, Kluwer Acad. Publ., Norwell, MA, 2003.

[8] J. S. Plank, M. G. Thomason, A practical analysis of low-density parity-check erasure codes for wide-area storage applications, in: DSN-2004: The International Conference on Dependable Systems and Networks, IEEE, 2004.

[9] J. S. Plank, A. L. Buchsbaum, R. L. Collins, M. G. Thomason, Small parity-check erasure codes - exploration and observations, in: DSN-05: International Conference on Dependable Systems and Networks, IEEE, Yokohama, Japan, 2005.

[10] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, J. Kubiatowicz, Pond: The oceanstore prototype, in: Proceedings of the Conference on File and Storage Technologies (FAST'03), 2003.

[11] S. Lin, C. Jin, Repstore: A self-managing and self-tuning storage backend with smart bricks, in: ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04), IEEE Computer Society, Washington, DC, USA, 2004, pp. 122–129.

[12] S. Frølund, A. Merchant, Y. Saito, S. Spence, A. C. Veitch, FAB: Enterprise storage systems on a shoestring, in: 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003, pp. 169–174.

[13] M. Ripeanu, I. Foster, A. Iamnitchi, Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, IEEE Internet Computing Journal 6 (1) (2002) 50–57.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, Wide-area cooperative storage with cfs, SIGOPS Oper. Syst. Rev. 35 (5) (2001) 202–215.

[15] A. Rowstron, P. Druschel, Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility, in: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles, ACM Press, New York, NY, USA, 2001, pp. 188–201.

[16] E. K. Lee, C. A. Thekkath, Petal: distributed virtual disks, SIGOPS Oper. Syst. Rev. 30 (5) (1996) 84–92.

[17] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google File System, in: Symposium on Operating Systems Principles (SOSP), 2003, pp. 29–43.

[18] J. W. Byers, M. Luby, M. Mitzenmacher, A. Rege, A digital fountain approach to reliable distribution of bulk data, SIGCOMM Comput. Commun. Rev. 28 (4) (1998) 56–67.

[19] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, R. Y. Wang, Serverless network file systems, ACM Trans. Comput. Syst. 14 (1) (1996) 41–79.