Publication II

Mikko Pitkänen, Juho Karppinen, and Martin Swany. 2006. GridBlocks DISK - distributed inexpensive storage with K-availability. In: Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 2006) Workshop on Next-Generation Distributed Data Management. Paris, France. 20 June 2006. Pages 1-6.

GridBlocks DISK - Distributed Inexpensive Storage with K-availability

Mikko Pitkanen
Helsinki Institute of Physics
Technology Programme
mikko.pitkanen@hip.fi

Juho Karppinen
Helsinki Institute of Physics
Technology Programme
juho.karppinen@cern.ch

Martin Swany
University of Delaware
Dept. of Computer and Info. Sciences
swany@cis.udel.edu

Abstract

This paper describes an architecture for an archival data storage system. The design enables aggregation of storage resources in a scalable fashion to achieve highly reliable data storage. Reliability is implemented by using erasure coding which provides lower storage overhead than full replication. Scalability of the architecture is achieved through the ability to work with multiple different storage locations and a scalable metadata management system. The integrity of the data is ensured through use of cryptographic naming. Feasibility of the proposed design is assessed with a real world implementation named GB-DISK.

1 Introduction

It has been shown that erasure codes use orders of magnitude less bandwidth and storage than replication systems with similar reliability [20]. Moreover, erasure codes can be used to achieve high reliability in scalable distributed data structures with small theoretical storage overhead [11]. It has been suggested that erasure codes could be used in parallel with replication since they enable fast access to data by reducing network latency [16, 20]. Additionally, erasure codes help ensure the durability of storage over long periods of time.

We propose a design called GB-DISK which is a component of the GridBlocks architecture [7] for aggregating storage resources using erasure codes. GridBlocks is an application framework implemented in Java that is designed to provide building blocks for Grid application programmers. The commitment to use Java components in our design has guaranteed easy portability of the implementation to run on virtually any platform.

In addition to software portability we have aided extensibility by supporting multiple transport protocols. Different transport mechanisms enable operation under heterogeneous network policies and enable flexible resource aggregation. Moreover, the performance critical components, namely metadata and coding elements, are designed so as to enable scalable operation. Functionality of these central components can be flexibly replicated as the load on the system increases, as demonstrated by our performance tests for GB-DISK prototype.

The structure of this work is as follows. In Section 2 we will present the related work done both with distributed data storage and with erasure coding. Erasure coding is explained in Section 3. In Section 4 we explain the architecture of our GB-DISK archival storage. In Section 5, the feasibility of our design is demonstrated with real world use of the system. Finally, Section 6 concludes our work and discusses the future directions.

2 Related Work

Collaborative file sharing is a well known application in peer-to-peer networks [5, 16]. While many issues regarding indexing and routing have been solved the mechanisms developed for wide area environments are not always optimal for deployment in local area networks. In our earlier [13] work we have shown how Reed Solomon (RS) erasure code can be feasibly used to combine storage resources over wide area network. The striping and erasure coding were implemented as a client application and GridFTP was used for the transport. While the earlier work focused on performance issues in the wide area networks here we present an architecture suitable for metadata management, storage maintenance and flexible aggregation of resources in local area networks.

FreeLoader [19] is a recent approach to aggregate idle workstation resources to form a collaborative storage pool. Freeloader uses data striping to achieve load balancing between storage contributors. Also, the metadata management and the storage contributors are separated similar to our approach and no assumptions are made on the storage element availability. However, our use of erasure codes brings additional robustness to stored data and our virtual machine based implementation makes our architecture more flexible to deploy.

IBP [14] is a system for sharing storage in the network that can take advantage of coding. IBP exposes file metadata and client nodes are responsible for downloading and reassembling data if necessary.

DRALIC [8] is a proposed architecture to connect storage resources in local area network. The architecture implements reliable storage by using RAID techniques to calculate parity between disk space contributions. Moreover, each host node contributes a RAM partition to enable collaborative caching for fast access to data. However, DRALIC architecture requires access to low-level system resources being less flexible approach than the architecture we propose.

StarFish [6] is a block storage system that ensures high-availability of data through replication. StarFish is implemented as a low-level interface and requires operation system dependent device drivers. With StarFish high performance is achieved, however, reliability requires high overhead due to replication of data.

Litwin et al. [11] propose the use of RS codes to achieve high-availability in scalable and distributed data storage. Their design $LH*_{RS}$ uses linear hashing to address the data stored in the main memory of computers in a local area network. From their work we have adopted the idea to apply systematic RS codes to ensure high redundancy of data with minimum possible overhead. The code that we use is based on the work by Rizzo [17], which can be referred to for a good explanation of the workings of erasure coding.

Many recent proposals to use erasure codes for highly available data storage have suggested the use of low density parity check (LDPC) codes. In LDPC very fast encoding and decoding processes can be achieved with a slightly larger storage overhead. However, the RS codes can have a better performance when used with relatively small (< 20) values for n [15]. The codes of the LDPC family have the property that not all combinations of file fragments are sufficient for decoding the original data. This makes the management of metadata and stripe download selection a more complicated process [3]. However, the work on LDPC codes is currently active and with different stripe placement approaches the novel codes may prove more efficient to be used with file storages. These novel coding techniques include e.g. Tornado [1], Raptor [18], LT[12], and LEC [4] codes.

3 Erasure Codes and File Striping

Reed Solomon erasure codes can be used to provide K-availability. This property states that k can be chosen freely and any random k data segments can be lost without losing any original information. The RS code can be used to encode m data items into $n > m$ encoded data items. Later, it is sufficient to have any m out of these n items to decode the original information. We use the erasure code with striping of files to provide data redundancy in the storage. Thus the original data can be recovered even if $n - m$ of the stored file stripes are unavailable. Moreover, the code we use is a systematic code meaning that the original m stripes form a verbatim copy of the input file. With systematic code the decoding is unnecessary when there are no lost stripes. This enables partial file access and very fast reconstruction in the event that no data is lost.

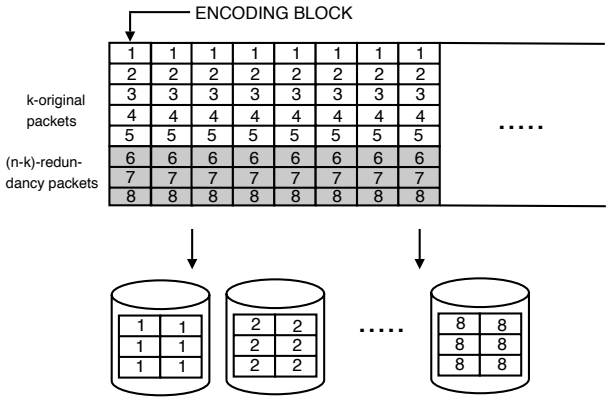

Figure 1. Using striping and erasure codes to provide redundancy of arbitrary data files.

Figure 1 illustrates how we use file striping and erasure codes to provide redundancy of files. In the process of encoding, the file is read block-by-block. Afterward, each block is encoded from m packets to n packets and these packets are each written to a separate file. Thus, we have implemented a striping approach where one file is read as input and n files are produced as output. The stripes can be then handled as traditional files enabling easy operation with different transport protocols and storage resources.

The block size defines the balance between time used to do the I/O operations and calculating the encoding, and vice-versa in the reconstruction phase. Thus the choice of block size has an effect on the performance of the coding procedure. While the optimal block size is dependent on the host system, we have observed sizes between $10 - 40$

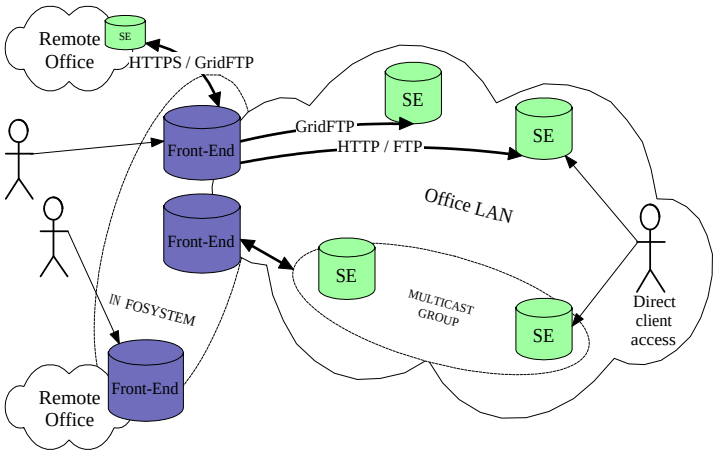

Figure 2. Overall architecture of GB-DISK.

KB to show good performance on Linux systems; see [13] for details on the evaluation of different coding parameters. With the chosen approach only one block needs to be maintained in memory during coding. This leads to a design that is general and works independent of file size.

4 GB-DISK Architecture

The GB-DISK architecture is illustrated in Figure 2. We have grouped the functionality of the GB-DISK into the following core entities:

- **Front-end** (FE) hosts the *Coordinator* that orchestrates the storage work-flow. The Coordinator controls the file striping and is responsible for keeping the storage information up-to-date in the metadata service. The Coordinator manages the leaving and joining of storage elements. Additionally, the Coordinator provides a simple API for third party applications and user interfaces.

- **Metadata service** stores information about files including erasure coding parameters, stripe integrity and respective storage locations. The service is implemented using caching and replication of data objects over a reliable multicast protocol [10]. The resulting design leads to a scalable, highly-available and fault tolerant metadata service.

- **Storage elements** (SE) contribute storage space for storing the stripes. No intelligence resides in storage element side and the performance impact to hosts is limited to network and disk I/O. The SEs can be easily deployed as they only need to support a simple set of operations and various transports can be chosen.

The minimal installation of GB-DISK requires a single information server and one storage element. These services can be run in the same host. In more realistic deployments, the information system runs on multiple front-end servers to provide load balancing and high availability of the metadata. After striping and coding the file fragments are stored into n separate storage elements.

A user can connect to the storage system using command line utilities from his or her workstation or using a web interface which runs in the front-end. The web interface can be used through HTTP browser or WebDAV folder utility. The simple interface and automatized working of the coordinator enables easy integration as a back-end of different storage services. The project is released as open source and the source code and technical documentation can be found at Sourceforge [7].

4.1 Management of Metadata

The GB-DISK metadata system uses JBossCache [9] to automatically distribute and synchronize the metadata objects among all hosts participating to metadata service. The cache uses an Aspect Oriented Programming (AOP) architecture, which enables transparent and fine-grained control over the replication of simple Java data objects.

A single metadata object contains the required information to fetch and decode a file from our system. Our current design stores files in a flat name space. However, the implementation is not limited to this as the cache is internally organized as a tree hierarchy. This can be easily used to support hierarchical file naming.

Figure 3 illustrates an example metadata object, which we store into our info-system. Each metadata object contains the coding parameters m and n. We also store the

```
fileName=jboss-jmx.jar
m=3
n=5
packetSize=10000
fileLength=591108
0=d3472
1=81d71
2=37398
3=914e6
4=e6de7
d3472=gsiftp://pchip69/
81d71=gsiftp://pcolrent20/
37398=http://pchip69:8080/
914e6=http://pcolrent20:8080/gb-disk
e6de7=gsiftp://pchip69/
```

Figure 3. An example metadata instance.

packet size used during the striping (as shown in Figure 1) as well as the length of the original file in bytes. In addition the naming and location information is stored for each file. The metadata description is the only place where the original file name is stored. For each stripe the hosting storage element is stored with the name of the stripe, which also serves as an integrity check-sum.

After encoding the file, the stripes are named by calculating a checksum over their contents. This ensures integrity and enables the use of a flat name space within the architecture. Checksums are stored only for the stripes so that their integrity can be checked before using them in the decoding process. We assume that a reliable decoder produces valid output whenever the input stripes are intact. The use of the flat name-space enables integration of our design with various DHT storage architectures [2].

A new metadata object is instantiated and stored in the cache for every new file. The fine-grained control of metadata creation, together with synchronization based on reliable multicast, increases the scalability and performance of the cache. In addition the cache supports transactions to be sure that no conflicts occur in the metadata service.

In our prototype the content of the cache can be listed or searched by the name of the file. The result of the query is an instance of the metadata object. Since the metadata is replicated over all hosts participating in the metadata service, the searches can be processed locally without imposing network delay or using network resources. Moreover, the cache contents are automatically check-pointed to disk to ensure persistency of the information.

4.2 Transport and Storage Aggregation

For easy aggregation of resources we have implemented support for multiple transport protocols between coordinator and storage elements. Our simple transport API supports the following operations:

```
put()        get()
exists()     delete()
```

The design makes minimal assumptions about the functionality of the storage element, and access to this functionality can be provided through the most suitable transport channel. The support for multiple transport protocols enables a host to contribute storage even in the presence of firewalls or other required extended security features. Transport over the following protocols is supported:

- **HTTP(S)** protocol offers flexible operation in various environments as only one network port needs to be used for data traffic. We have packaged a minimal server setup that requires only Java to be installed on the host computer. The server can be accessed also through WebDAV protocol. HTTP communication can be secured with SSL when storage elements are placed outside firewalls.

- **(Grid)FTP** is used to enable resource aggregation in multi-site operation. The certificate based security enables resource aggregation over untrusted networks. Moreover, we benefit from single-sign-on when stripes are gathered from multiple sites. We support also plain FTP to enable operation with existing FTP servers. The use of multiple ports for data channels, however, increases complexity of the configuration process.

- **JGroups** multicasting components are embedded to our HTTP storage elements for providing resource discovery and keep-alive information in LAN environment. The same protocol stack can be used for file transfer with ability to operate behind firewalls.

4.3 Coordinating and Logic

In our design, storage elements have a role of simple data storage without any embedded logic. The metadata service is an independent look-up service and does not know how to use the stored information. The Coordinator running on a front-end, or a user's desktop, is the intelligent component which orchestrates the storing and fetching processes.

The metadata system can be used to store operational information such as performance statistics from storage elements. Based on this information, and applying the storage policies, the coordinator can decide the best locations to upload new files into. The Coordinator can be used also to deduce the fastest storage elements for downloading the stored stripes. As an initial strategy we have used simple round-robin scheduling for uploading the stripes. In the downloading process n transfers are started. After m stripes have arrived and been verified to be intact, the remaining transfers are stopped.

5 Evaluating Performance

In this section we provide performance measurements to justify the applicability of our design. The measurements utilize workstations in a classroom. Each workstation runs the Linux operating system and features a 2.4 GHz processor and 1 GB of memory. The 100Mbps LAN set-up is typical for a university or enterprise network. Data transfers are performed by using plain HTTP operations. For testing we use five storage elements with variable number of FEs and clients. Performance logging is done in the FE to get detailed information as execution proceeds.

With our performance evaluation we illustrate two desirable properties of the design: i) a single FE can handle multiple simultaneous users, and ii) adding FE elements increases overall system performance.

5.1 Using Single Front-end

In the measurements a single FE is used to serve multiple clients. For each measurement every client transfers 50 files through the FE, each having a random size from the set [20,40,60,80,100] MB. Using random file sizes in the clients avoids the situation where simultaneous file transfers would synchronize and create artificial load peaks. For each measure max, min, median, lower quartile and upper quartiles are plotted.

The figures plot aggregate bandwidth that is file size divided by time it takes as the file travels through the components of the system. This includes transfer between client and FE, coding, checksum, metadata operations and transfer between FE and SEs. This describes the capacity with which the FE is able to process files. For all measurements we have used $m = 3$ and $n = 5$. This provides a configuration where any two SEs can be simultaneously unavailable.

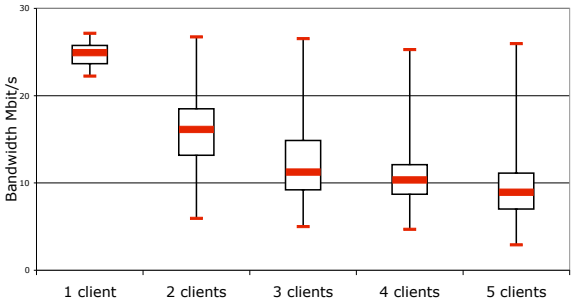

Figure 4. Download throughput from client perspective, single FE and multiple clients.

Figure 4 illustrates download performance with a single FE. The figure shows bandwidth of the whole transfer and decoding procedure as seen from the client perspective. A single FE can serve one client with half of the cases falling between $24 - 26$ Mbit/s. With greater number of simultaneous clients the median performance decreases moderately, achieving close to a third of the performance with 5 clients compared to the single client case. As the number of the clients increases the performance becomes less predictable. While the performance decreases from the clients' perspective, the overall throughput increases when more clients are served simultaneously. This is seen because file transfers, decoding and integrity tests have different system requirements and do not interfere with each other continuously.

The upload measurements are very similar to those of the downloads thus, the figures are omitted for brevity. However, the clients upload files to FE like to a traditional server, after which the FE takes over processing. Thus, the client sees faster upload excluding time to code and to transfer to SEs. However, the presented measures describe the overall performance of the system in the upload case as well.

5.2 Using Multiple Front-ends

Next, we show how multiple front-end nodes help to increase the throughput of our storage. Clients use simple round-robin scheduling to distribute "put" and "get" requests between two identical FEs. The infosystem keeps the file distribution metadata replicated in every FE. Thus, the metadata of files can be accessed locally in any available FE independent of the FE through which the file was originally stored. In addition to load balancing, the metadata replication helps to preserve data when FEs are lost.

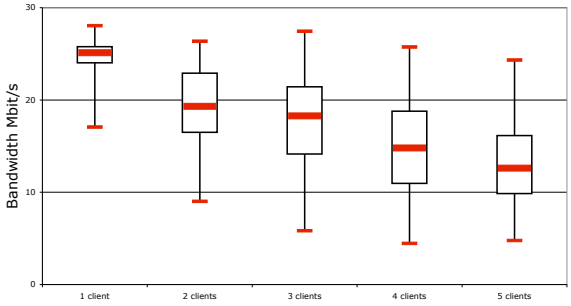

Figure 5. Download throughput from client perspective, two FEs and multiple clients.

Figure 5 illustrates download performance from the client perspective when two FEs are used to balance the load. With the addition of another front-end, performance increases of about 50% are seen when three or more clients are accessing the storage. Again the upload performance is very similar with respect to all measures and the plot is omitted.

During the download case the average access time from the infosystem drastically decreases with two FEs. While the infosystem response takes 19 ms on average with one FE in two FE case the response is received in 5 ms on average. In both cases the infosystem look-up is processed locally but when two FEs are used the FE load stays lower. In contrast, during the upload, the synchronous replication of the metadata takes some time. However, the observed increase was small with two FEs. With larger FE clusters we could use asynchronous replication offered by the metadata system to increase its performance.

6 Conclusions and Future Directions

In this paper we have presented an architecture for flexible storage resource aggregation. The architecture uses software erasure codes to provide high reliability against data losses. We have implemented scalable and transparent infosystem for LAN environments. The infosystem is used to replicate metadata of file distribution within the system. We have demonstrated with empirical measurements that the proposed architecture is feasible to deploy on today's workstation environments.

We have focused on presenting an architecture that can efficiently distribute files within LAN environment providing high reliability. The design includes methods to work with plain files and to store the needed metadata. We have used our storage as a back-end of HTTP-service and as a WebDAV folder provider. In the future we plan to extend the front end functionality to support the storage interfaces used in the scientific Grid data management infrastructures. Moreover, we continue to study which parts of the presented architecture can be feasibly distributed over wide area networks.

Acknowledgments

This work was partly supported by the Finnish Academy of Science grant no 205961 and by the University of Delaware.

References

[1] J. W. Byers, M. Luby, and M. Mitzenmacher. Accessing multiple mirror sites in parallel: Using tornado codes to speed up downloads. In *INFOCOM*, pages 275–283, New York, NY, Mar. 1999. IEEE.

[2] M. Cai, A. Chervenak, and M. Frank. A peer-to-peer replica location service based on a distributed hash table. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 56, Washington, DC, USA, 2004. IEEE Computer Society.

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

[4] J. A. Cooley, J. L. Mineweaser, L. D. Servi, and E. T. Tsung. Software-based erasure codes for scalable distributed storage. In *IEEE Symposium on Mass Storage Systems*, 2003.

[5] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of HotOS VIII*, 2001.

[6] E. Gabber, J. Fellin, M. Flaster, F. Gu, B. Hillyer, W. T. Ng, B. Özden, and E. A. M. Shriver. Starfish: highly-available block storage. In *USENIX Annual Technical Conference, FREENIX Track*, pages 151–163, 2003.

[7] GridBlocks Framework. `http://gridblocks.sourceforge.net`, referenced July 17, 2007.

[8] X. He, M. Zhang, and Q. Yang. DRALIC: A peer-to-peer storage architecture. In *Proceedings of the PDPTA 2001*, 2001.

[9] JBossCache. `http://www.jboss.org/products/jbosscache`, referenced July 17, 2007.

[10] JGroups Project. `http://www.jgroups.org`, referenced July 17, 2007.

[11] W. Litwin, R. Moussa, and T. J. Schwarz. LH*$_{RS}$ - a highly-available scalable distributed data structure. *Journal of ACM TODS*, 35, 2005.

[12] M. Luby. Lt codes. In *FOCS '02: Proceedings of the 43rd Symposium on Foundations of Computer Science*, page 271, Washington, DC, USA, 2002. IEEE Computer Society.

[13] M. Pitkanen, R. Moussa, M. Swany, and T. Niemi. Erasure Codes for Increasing Availability of Grid Data Storages, To appear in Proceedings of ICIW06.

[14] J. S. Plank, A. Bassi, M. Beck, T. Moore, D. M. Swany, and R. Wolski. Managing data storage in the network. *IEEE Internet Computing*, 5(5):50–58, September/October 2001.

[15] J. S. Plank and 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, June 2004.

[16] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage, 2001.

[17] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.*, 27(2):24–36, 1997.

[18] A. Shokrollahi. Raptor codes. Technical report, Laboratoire d'algorithmique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, 2003. Available from `http://algo.epfl.ch/`.

[19] S. Vazhkudai, M. Xiaosong, F. Vincent, S. Jonathan, T. Nandan, and S. Stephen. Freeloader: Scavenging desktop storage resources for scientific data. In *SuperComputing'05*, 2005.

[20] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of IPTPS*, 2002.