Hydra: A Platform for Survivable and Secure Data Storage Systems

Lihao Xu
Dept. of Computer Science
Wayne State University
lihao@cs.wayne.edu

ABSTRACT

This paper introduces *Hydra*, a platform that we are developing for highly survivable and secure data storage systems that distribute information over networks and adapt timely to environment changes, enabling users to store and access critical data in a continuously available and highly trustable fashion. The Hydra platform uses MDS array codes that can be encoded and decoded efficiently for distributing and recovering user data. Novel uses of MDS array codes in Hydra are discussed, as well as Hydra's design goals, general structures and a set of basic operations on user data. We also explore Hydra's applications in survivable and secure data storage systems.

Categories and Subject Descriptors

H.3.2 [**Information Storage**]: File organization

General Terms

Design

Keywords

Distributed Storage, Survivable Storage, Secure Storage, MDS Array Codes

1. INTRODUCTION

The rise of ubiquitous computing have prompted the need of ubiquitous information storage and access. Supporting the availability, survivability, persistence, confidentiality and integrity of information is becoming more and more crucial. This calls for secure and reliable data storage systems that distribute information over networks, enabling users to store and access critical data in a continuously available and highly trustable fashion. Many research and development efforts have been made to address various issues in building such storage systems, including disk arrays, such as the RAID [8], clustered systems, such as the NOW [1], distributed file systems, such as the NFS (Network File System)[25], HA-NFS[3], xFS[2], AFS[22], Zebra[12], CODA [23], Sprite[18], Scotch[10] and BFS[7], storage systems, such as NASD [9], Petal[15], PASIS[27] and RAIN[6], and large scale data distribution and archival networks, such as Intermemory[11], Ocean Store[14] and Logistical Network[19].

As already indicated by these efforts, proper data redundancy is the key to provide high reliability, availability and survivability. Evolving from simple data replication or data striping in early clustered data storage systems, such as the RAID system [8], people have realized it is more economical, efficient and secure to use the so-called *threshold schemes* to distribute data over multiple nodes in distributed storage systems [27, 26, 11, 14]. The basic idea of threshold schemes is to map an original data item into n pieces, or *shares*, using certain mathematical transforms. Then all the n shares are distributed to n nodes in the system, with each node having one share. Upon accessing the data, a user needs to collect at least k shares to retrieve the original data, i.e., the original data can be *exactly* recovered from m different shares if $m \geq k$, but less than k shares will not recover the original data. Such threshold schemes are called (n, k)-threshold schemes. The threshold schemes can be realized by a few means, and can also be used for *secret sharing*[4, 24, 17, 20]. To maximize the usage of network and storage and eliminate bottlenecks in a distributed storage system, each data share should be of the same size. Otherwise the failure of a node storing a share with bigger size will have bigger impact on the system performance, thus creating a bottleneck in the system.

From *error control code* point of view, an (n, k)-threshold scheme with equal-size shares is equivalent to an (n, k) block code, and especially most (n, k)-threshold schemes are equivalent to (n, k) MDS (*Maximum Distance Separable*) codes [17, 16]. An (n, k) error control code uses mathematical means to transform a k-symbol message data block to an n-symbol codeword block such that any m symbols of the

StorageSS'05, November 11, 2005, Fairfax, Virginia, USA.

codeword block can recover all the k symbols of the original message data block, where $k \leq m \leq n$. All the data symbols are of the same size in bits. Obviously by the simple pigeon hole principle, $k \leq m$. When $m = k$, such an (n, k) code is called MDS code, or meets the Singleton Bound [16]. Hereafter we simply use (n, k) to refer to any data distribution scheme using an (n, k) MDS code. Using coding terminology, each share of (n, k) is called a data *symbol*. The process of creating n data symbols from the original data whose size is of k symbols is called *encoding*, and the corresponding process of retrieving the original data from at least arbitrary k data symbols stored in the system is called *decoding*.

It is not hard to see an (n, k) scheme can tolerate up to $(n - k)$ node failures at the same time, and thus achieve data reliability, or data *survivability* in case the system is under attack where some nodes can not function normally. The (n, k) scheme can also ensure the *integrity* of data distributed in the system, since an (n, k) code can be used to detect data modifications on up to $(n - k)$ nodes.

While the concept of (n, k) has been well understood and suggested in various data storage projects, virtually all practical systems use the Reed-Solomon (RS) code [21] as an MDS code. (The so-called *information dispersal algorithm* [20] used in some systems is indeed just a RS code.) The computation overhead of using the RS code, however, is large, as demonstrated in several projects, such as OceanStore [14]. Thus practical storage systems seldom use a general (n, k) MDS code, except for *full replication* or *mirroring* (which is an $(n, 1)$), *stripping* without redundancy (corresponding to (n, n)) or *single parity* (which is $(n, n - 1)$). The advantages of using (n, k) are hence very limited if not totally lost. We are thus developing a general storage platform, which we call *Hydra*, to provide flexible (n, k) schemes that can be computed efficiently, together with a set of basic operations on data files. As of our best knowledge, such a platform or interface does not exist in current commercial data storage systems. What makes Hydra *distinct* from other ones and thus novel and exciting is the use of *MDS array codes* (which we will have more discussions on in the following section) that we design and develop for this platform, and our novel use of an MDS code to achieve load balancing as well as fault tolerance.

2. ARRAY CODES AND HYDRA

2.1 Basics of Array Codes

MDS array codes are a class of MDS codes, whose encoding and decoding mainly are bitwise XORs (exclusive ORs). Known MDS array codes include the EVENODD code [5], the X-Code [28] and the B-Code [29]. These MDS array codes are very efficient to encode and decode.

Table 1 shows the encoding rule of a (4,2) B-Code. Here $+$ is the bitwise XOR operation (this notation will be used hereafter, unless otherwise stated). a_i's are user data symbols of the same size. A data symbol is a very flexible unit, which could be a bit, a byte or a block of arbitrary size. When applied to a distributed storage system, all the symbols in the same column are stored on the same node. For example, using this (4,2) B-Code for a system with 4 storage nodes, the 1st node stores symbols a_1 and $a_2 + a_3$ (note this symbol has the same size of a_i's), and the 2nd node has a_2 and $a_3 + a_4$, and so on. It is easy to verify the original 4 data symbols a_1 to a_4 can be recovered from all the symbols stored on *any* two nodes.

$node1$	$node2$	$node3$	$node4$
a_1	a_2	a_3	a_4
$a_2 + a_3$	$a_3 + a_4$	$a_4 + a_1$	$a_1 + a_2$

Table 1: Encoding of a (4,2) B-Code

2.2 Flexible Use of Array Codes in Hydra

2.2.1 Symbol Size

As already mentioned above, a data symbol in an array code is a very flexible data unit. It can be a bit, a byte, a block of any size, or even a file. An array code can be applied on a file, using a block whose size is optimal for the application. An array code can also be used for a *group* of files, with files being data symbols, as long as those files are of the same size. (We can use other simple techniques to apply an array code for files with different sizes. For example, set the data symbol size close to that of most of the files and use another symbol size for the remaining data.) This feature is very useful for certain applications, e.g., for an application which needs to achieve reliability of a set of files yet requires an individual file to be stored on a single node entirely for efficient computational processing.

2.2.2 Load Balancing

While an (n, k) MDS code ensures the exact recovery of original data from any k nodes, user data can certainly be recovered from more than k nodes. This feature enables dynamic load balancing among the storage nodes. For example, when the (4,2) B-Code in Table 1 is used, as long as all the 4 nodes are functioning, a user data can be easily recovered by collecting the 4 original data symbols in the first row from all the 4 nodes without any decoding computation. It is also possible to recover user data from 3 nodes, e.g., 1 symbol each from the first 2 nodes, and 2 symbols from the last node. In this case, decoding only needs to recover 1 original data symbol. Thus based on the available network bandwidth, processing load and response latency from each node, as well as processing power of the client node, different data recovery operations can be performed to achieve load balancing among the storage nodes, *dynamically* on block level. A client can also achieve trade-off between communication and computation. For example, if it is more expensive

to set up and maintain connections to storage nodes, due to latency and other factors, than local computations, a client may well choose to recover the original data symbols from only k nodes even if other nodes are accessible.

2.2.3 Distribution of Symbols

When an (n, k) code is applied to a user data, in general one node stores one data symbol of a codeword. But these nodes can be grouped based on various application needs. For example, if a (8,3) code is applied on a user data, and we have storage nodes physically located at two different sites A and B. Then we can evenly split the 8 nodes between the two sites, with 4 each. In this case, the 4 nodes at site A can form a group, which can provide user data in normal situations. Should one node fail at site A, the remaining 3 nodes can still provide intact data to the user, without contacting the nodes at site B. Only when 2 or more nodes fail at one site, a user data needs to be recovered with the help of nodes from another site. Thus in normal situations, the two groups at the two sites can function independently for user data retrievals, while each group has certain degree of reliability. Combined, the two groups still maintain the designed overall reliability degree.

2.2.4 Tunable Parameters

In a practical system with N nodes, an (n, k) code (where usually $n < N$) can be applied to encode a user data. Then n symbols are stored on n of the N nodes in the system. However, both k and n are tunable based on the user's needs. Note here the reliability degree, i.e., the number of tolerable simultaneous node failures, depends on the difference of n and k, namely $(n - k)$; while the effective storage use depends on the ratio of n and k, i.e., k/n. Further more, even after both n and k are already determined, it is still possible to use an (m, k) code (where $m > n$) by storing only n of the m data symbols on n storage nodes. This gives Hydra another degree of flexibility in data placement. For example, to enable more random node *grouping*, some nodes other than the n designated notes may choose to store some of the codeword symbols as well. If just an (n, k) code is used, some data symbols are inevitably replicated in a naive fashion, thus the increase in reliability degree by those node is limited. In worst case, there is no reliability increase at all. But by using an (m, k) ($m > n$) code, extra nodes can store data symbols other than the n ones already used. This effectively increases reliability degree to the maximum possible.

3. DESIGN OF HYDRA

The *Hydra* platform is being developed for both traditional *file systems* and *database-like* structured data records. In the first phase, the development effort has been on file systems, upon which many applications are based. Thus in this section, our discussions are focused on file systems. Many important components to be discussed, however, can be readily shared by database-like data pools.

3.1 Design Goals

For file systems, the *Hydra* platform should meet the following objectives:

- The platform should not be restricted to either a particular operating system (OS) or a particular file system (FS). It should be easily portable to any OS and FS.
- The platform should not be dependent on any particular hardware.
- UNIX file system semantics should be maintained for UNIX clients.
- The platform's overhead on a host file system should be negligibly small, in terms of both performance and storage usage.
- The implementation should be transport-independent.
- It should be easily installable, configurable, scalable, flexible, maintainable and automatable.

3.2 Structure and Components

To meet the above objectives, the *Hydra* platform is built top of a host (distributed) file system. (We may also pursue to build Hydra under a file system to enable fast and painless migration of various applications to our Hydra platform; or we may choose to integrate this platform to a file system for further efficiency.) The advantage of this approach is to maximize the use of an existing file system to deal with common issues, such as name space, file consistency, read/write caching, stateful or stateless operations and semantics of file sharing, thus enables us to focus on new components that complement existing file systems.

The Hydra consists of the following basic components or building blocks:

1. Environment Monitor
2. Meta Data
3. Operations

We briefly discuss these components in the subsequent subsections.

3.3 Environment Monitor

In Hydra, each storage node has a monitor which monitors environment changes in a system. A monitor communicates with its peers on other nodes. The communications may use broadcast, unicast or both, depending on a system's architecture and properties, as well as applications' needs. What a monitor keeps track of includes information about

which storage nodes are accessible, the dynamic connectivity changes in the system, and various loads on each storage node. (An inaccessible node may be deemed to have infinitely high load on it.) Upon request, the monitor can direct an application to proper nodes for various operations.

The monitor may also perform functions that are necessary or desirable for various secure data operations, e.g., authentication among storage nodes.

3.4 Meta Data

For a user file, its encoded shares are distributed through the host file system's *write* operations. Conversely, a user file's encoded shares are collected through the host file's *read* operations. Every encoded share of a user file has a *new* metadata (index) file associated with it, which is tentatively called the *hnode* (Hydra Node). The *hnode* stores the information necessary for distributing and recovering the associated file(s). It includes the following attributes:

- Locations of all associated shares and the original file, which may be specified by a path name with the server node ID if not already included in the path name.
- Error correcting code used for encoding and decoding the shares.
- Security flag: indicates if an encryption key is needed to recover the user file.
- Access rights: rights for read, append, write or delete
- Extension for other information

Each of the encoded shares of a user file, as well as the file itself (if the user chooses to store a local copy in addition to its encoded shares on server nodes) has the same *hnode* information stored with it for high redundancy. Since *hnode* is very important and its size is small (less than a hundred bytes), this high redundancy brings another level of reliability and security for easy detection of any modifications on it and easy recovery in case some *hnode* is destroyed together with its encoded share.

In addition to the *hnode* for each user file, *Hydra* also maintains system wide configuration information, mainly the mapping between the logical server ID and its location information, such as IP address or host name. This configuration file should be small and easy to manage.

The server nodes may but not necessary replicate directory trees on a client machine, which can be decided by users or system administrators.

3.5 Operations

3.5.1 Distribute (Write)

This operation uses a proper code, which may be specified by the user, to encode the desired shares. Then it creates a corresponding *hnode* file if not existed already, and writes the shares together with the *hnode* files to proper server nodes (which again may be specified by the user). The file *create* and *write* operations are just the ones supported by the host file system.

If the *hnode* file exists, it updates the related *hnode* files if necessary, for example a previous *Move* operation was incomplete (see the operation below) or a different code is used for this write operation.

3.5.2 Recover (Read)

This operation first reads share information from the *hnode* and updates all the *hnode* files if necessary (see the *Move* operation below), then reads proper shares from as many accessible server nodes as possible. The availability of a share file on a server node can be easily detected by opening the share file. After collecting enough shares, a decoding operation is performed to recover the original user data. The user data then is passed to proper applications or written to a local file specified by the user.

Note here the decoding operation may not need any computation at all if all the share files are accessible. For example, if the (4,2) B-code in Table 1 is used to distribute the original data, and all the 4 share files are accessible, this recover operation simply reads all the 4 original data symbols at the first row from the 4 server nodes, without any computation. In general, the more server nodes are accessible, the less computation the decoding operation needs.

Again the file *open* and *read* operations are from the host file system.

3.5.3 Detect

This operation is to detect whether the original file (local copy) and/or the share files have been modified (or *corrupted*) on some server nodes. The operation employs a proper *error detection* algorithm to detect possible modifications in the share files and/or original file. It issues proper report if modifications in the original or any share file are detected.

An (n, k) MDS code can detect up to $(n - k)$ symbol errors[16].

3.5.4 Repair

Upon detection of any modifications in the original or any share file, this operation may be used to correct the modifications if the number of the modifications is within the code's *error correcting* capability. This operation employs a proper *error correction* algorithm of the code to correct the errors (i.e., the modifications).

An (n, k) MDS code can correct up to $\lfloor \frac{n-k}{2} \rfloor$ symbol errors [16].

3.5.5 Restore

When some share files are known to be modified or corrupted or totally destroyed due to attacks or disasters, this operation can be used to restore those share files (and thus

the original file as well if needed) to a consistent state with the other intact share files. In coding theory, these share files are regarded as *erasures*, and this operation uses a proper *erasure decoding* algorithm to restore these erased symbols (share files). Note the difference between this operation and the *Repair* operation is whether the locations of corrupted share files are known or not. This operation can be used if the corrupted share files are already identified (e.g., a server node has been compromised and a share file on it has been destroyed), while the *Repair* operation only knows there are corrupted share files but their locations are *not* known, i.e., which share files are corrupted is unknown. Thus for a given code, the *Restore* operation is more efficient to compute and can recover more corrupted share files.

An (n, k) MDS code can correct (restore) up to $(n - k)$ symbol erasures [16].

3.5.6 Copy

When the original file or a share file is copied, it creates a *hnode* file for the new file, and copies the associated *hnode* to the new *hnode* file. The *copy* operation is the host file system's file copy operation.

3.5.7 Move

When the original file or any share file is moved, this operation updates its *hnode* with the new pathname of the moved file and moves (using the host file system's *move* operation) the updated *hnode* to the new location as well. The *Move* operation then updates *all the other* related *hnode* files with the new locations of the moved file and its *hnode*. If some *hnode* files are inaccessible, this operation marks a flag in the extension field of the moved *hnode* to indicate this move operation is incomplete and related *hnode* files need to be updated when accessed later.

3.5.8 Check

This operation checks and reports the consistency of all the related *hnode* files and performs proper updates if necessary. It can be used before the *Detect*, *Repair* and *Restore* operations.

3.5.9 Delete

First any share file and/or its *hnode* is *not* expected to be deleted separately. If such a deletion is detected through the *Check* operation, it indicates possible compromise on the file and further actions may be needed.

This operation is used to delete (using the host file system's *delete* operation) the original file (if exists) and deletes all the associated share files and *hnode* files. If some share files and/or *hnode* files are inaccessible, a *deletion incomplete* flag is marked in the original file's *hnode*. This *hnode* is not deleted until all the other related files are deleted.

3.5.10 Others

Besides the above basic operations, other utility operations are needed to access, display, modify and update *hnode* files. More operations may be needed if necessary.

4. APPLICATIONS OF HYDRA

In addition to prototyping the *Hydra* platform, we will also develop a few applications.

Application of the *Hydra* platform for survivable storage is straightforward. Using this platform, important data files can be distributed to multiple storage nodes that are possibly placed at different physical locations.

The *Hydra* platform can also be used to provide a *secure* distributed storage for various applications and facilities. In most attacks on storage systems, one of first things an (educated) attacker does after successfully compromising a node is to change, corrupt and/or destroy related log files. Such actions certainly increase the difficulty of timely detecting, defending and repairing a corrupted system. While itself alone is not enough for secure storage, together with other tools, such as system activity loggers (e.g., *syslogd* in UNIX type systems), network activity monitors (e.g., *netstat* in UNIX), file integrity checker (e.g., *tripwire* [13]) and other intrusion detection and prevention tools, *Hydra* provides a secure storage platform for various important system log information. Generally one can assume it is much harder for attacks to be successful on multiple server nodes at the same or relatively short time than on on a single node. Thus by distributing important system files, e.g., sensitive configuration files, log files and important binaries, to multiple server nodes using the *Hydra* platform and integrating various *Hydra* operations, such as *Check, Detect, Repair* and *Restore*, it is much easier to detect, prevent and restore against various attacks.

More details on applications will be reported as they are developed.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we briefly describe *Hydra*, a platform that is being developed for survivable and secure data storage systems and applications. Using flexible (n, k) MDS array codes that can be encoded and decoded efficiently for practical systems and applications, the Hydra platform provides a set of basic data operations, as well as necessary auxiliary tools. These operations and tools can be readily integrated, adapted and expanded to various systems and applications.

In the first phase, we have designed the general structure of Hydra, and we have implemented a library of encoding and decoding operations for certain (n, k) MDS array codes. In the following phases, implementation of more codes will be added to the library. The basic operations described in Section 3.5 will be implemented, including their interfaces to UNIX file systems. A few applications will also be developed to demonstrate Hydra's potential for data security.

Progresses on Hydra will be timely reported. Upon its completion, we hope the Hydra platform will become a useful building block and be adopted in various related data storage systems and applications.

6. ACKNOWLEDGMENT

This work is being support by the National Science Foundation through grant IIS-0430224.

7. REFERENCES

[1] T.E. Anderson, D.E. Culler and D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, 15(1), 54–64, 1995.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang, "Serverless Network File Systems", *ACM Trans. on Computer Systems*, 41-79, Feb. 1996.

[3] A. Bhide, E. Elnozahy and S. Morgan, "A Highly Available Network File Server", *Proc. of the Winter 1991 USENIX Technical Conf.*, 199-205, Jan. 1991.

[4] G. R. Blakley, "Safeguarding cryptographic keys", *Proc. AFIPS 1979 Nat. Computer Conf.*, 313-317, June 1979.

[5] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.

[6] V. Bohossian, C. Fan, P. LeMahieu, M. Riedel, L. Xu and J. Bruck, "Computing in the RAIN: A Reliable Array of Independent Node", *IEEE Trans. on Parallel and Distributed Systems*, Special Issue on Dependable Network Computing, 12(2), 99-114, Feb. 2001.

[7] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *Operating Systems Review*, ACM Press, NY, 173-186, 1999.

[8] P.M. Chen, E.K. Lee, G.A. Gibson, R.H. Katz, D.A. Patterson, "Raid – High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, 26(2), 145–185, 1994.

[9] G.A. Gibson and R. van Meter, "Network Attached Storage Architecture", *Communications of the ACM*, 43(11), 37-45, Nov. 2000.

[10] G.A. Gibson, D. Stodolsky, F.W. Chang, W.V. Courtright II, C.G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R.H. Patterson, J. Su, R. Youssef and J. Zelenka, "The Scotch Parallel Storage Systems," *Proceedings of the IEEE CompCon Conference*, 1995.

[11] A. V. Goldberg and P. N. Yianilos, "Towards an Archival Intermemory", *Proc. of IEEE Advances in Digital Libraries*, Apr. 1998.

[12] J.H. Hartman and J.K. Ousterhout, "The Zebra Striped Network File System," *ACM Transactions on Computer Systems*, 13(3), 274–310, 1995.

[13] G. H. Kim and E. H. Spafford, "The Design and Implementation of Tripwire: A File System Integrity Checker", *Proc. of 2nd ACM Conf. on Computer and Communication Security*, 1994.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage", *Proc. of the Ninth international*

Conference on Architectural Support for Programming Languages and Operating Systems, Nov. 2000.

[15] E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks", *Proc. ACM ASPLOS*, 84-92, Oct. 1996.

[16] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*, Amsterdam: North-Holland, 1977.

[17] R. J. McEliece, D. Sarwate, "On sharing secrets and Reed-Solomon codes", *Comm. ACM*, 24(9), 583-584, 1981.

[18] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer*, 21(2): 23-26, Feb. 1988.

[19] J. S. Plank, M. and T. Moore, "Logistical Networking Research and the Network Storage Stack," *USENIX FAST 2002, Conference on File and Storage Technologies*, work in progress report, January, 2002.

[20] M. Rabin, "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance", *J. ACM*, 32(4), 335-348, Apr. 1989.

[21] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. SIAM*, 8(10), 300-304, 1960.

[22] M. Satyanarayanan, "Scalable, Secure and Highly Available Distributed File Access", *IEEE Computer*, 9-21, May 1990.

[23] M. Satyanarayanan, J.J. Kistler, P. Kumar, M.E. Okasaki, E.H. Siegel and D.C. Steere, "CODA - A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, 39(4), 447–459, 1990.

[24] A. Shamir, "How to Share a Secret", *Comm. ACM*, 612-613, Nov. 1979.

[25] SUN Microsystems, Inc. *NFS: Network File System version 3 Protocol Specification*, Feb. 1994.

[26] M. Waldman, A. D. Rubin and L. F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant, web publishing system", *Proc. 9th USENIX Security Symposium*, 59-72, Aug. 2000. Online at: *http://www.cs.nyu.edu/ waldman/publius/publius.pdf*

[27] J. J. Wylie, M. W. Bigrigg, J. D. Strunk. G. R. Ganger, H. Kiliccote and P. K. Khosla, "Survivable Information Storage Systems", *IEEE Computer*, 33(8), 61-68, Aug. 2000.

[28] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. on Information Theory*, 45(1), 272-276, Jan., 1999.

[29] L. Xu, V. Bohossian, J. Bruck and D. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," *IEEE Trans. on Information Theory*, 45(1), 1817-1826, Nov. 1999.