Annales UMCS Informatica AI X, 1 (2010) 29-39
DOI: 10.2478/v10065-010-0033-9

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

SDDSfL vs. local disk – a comparative study for Linux

Arkadiusz Chrobot*, Maciej Lasota†, Grzegorz Łukawski‡, Krzysztof Sapiecha§

Divison of Computer Science, Świętokrzyska Polytechnic, Al. Tysiąclecia Państwa Polskiego 7, 25-314 Kielce, Poland.

Abstract – Efficient data management and distribution in a multicomputer is a subject of much research. Distributed file systems are the most common solution of this problem, however, recent works are focused on more general data distribution protocols. Scalable, Distributed Data Structures (SDDS) are another promising approach to this issue. In this paper we discuss the efficiency of an implementation of SDDS in various applications. The results of experiments are presented.

1 Introduction

A multicomputer [1] is a distributed system that consists of a number of computers (called nodes) which cooperate by exchanging messages and sharing resources over a fast network. Special software is required to enable a collaboration of all nodes of such a system. A part of the software should be responsible for efficient distribution and management of the data stored on a multicomputer. There are many known solutions of this issue. One of them is a distributed file system like NFS or AFS [1, 2]. It allows nodes of multicomputer to share the files of data stored on their hard disks. More general solutions are data distribution protocols like the Chord or the Symphony [3, 4]. Such a protocol is responsible for locating and assigning data to nodes of a

*a.chrobot@tu.kielce.pl
†m.lasota@tu.kielce.pl
‡g.lukawski@tu.kielce.pl
§k.sapiecha@tu.kielce.pl

multicomputer regardless of where the data is actually stored. It is especially suitable for peer-to-peer networks. Scalable, Distributed Data Structures (SDDS) [5, 6] are another approach to the problem of sharing data across a multicomputer. SDDS use hashing or range partitioning to address the data located in RAMs of the nodes. The advantages of SDDS are scalability and fast access to data which is particularly valuable as far as transactional databases are concerned.

In our previous work SDDS for the Linux operating system (the SDDSfL) were presented which is partially implemented on the operational level [16]. The implementation was subjected to various tests in order to evaluate its efficiency for databases. The tests results were promising. However, the gap between the performances of SDDSfL and the local disk was surprisingly low.

The aim of the paper is to investigate whether or not SDDSfL could work considerably faster than local disks for the applications from beyond the scope of databases. To this end two problems computationally hard for local disk, namely pattern matching and sorting, are implemented on SDDSfL and on up to date disk systems and thoroughly compared. The next section contains a short overview of previous works. In section 3 the motivation for this work is given. Section 4 describes SDDS and SDDSfL briefly. Section 5 addresses the SDDSfL efficiency evaluation methods. The results of experiments are described is section 6. The paper ends with conclusions.

2 Related work

The problem of distributing data across multicomputer nodes may be solved in many different ways. A distributed file system is one of the examples. The other is a protocol for data distribution. There are many distributed file systems. Each of them is designed with different requirements in mind. The Network File System (NFS) [1, 2] allows many clients to access files stored on a local disk of only one server. It is designated for Local Area Network and a rather small number of clients. OceanStore [7], on the other hand, is a distributed file system which is used in peer-to-peer networks that are built over the Internet. Nodes in such networks may join and leave at any moment, and for that reason OceanStore distributes files over many servers. It is also fault tolerant and assures privacy. Distributed file systems are very common in multicomputer environments because files are one of the most flexible and useful abstract data structures.

Protocols for data distribution are a quite new approach. They are applied on lower level than the distributed file systems but they are also more universal. Such protocols are available for user applications through special libraries which create an abstraction layer. This abstraction layer simulates some data structure, usually a distributed hash table (DHT). It is possible to build a distributed file system on the top of such DHT (OceanStore is one of them). Data distribution protocols are usually applied to peer-to-peer (P2P) networks. The Chord protocol [3] organizes nodes of such a network into a ring. Every node in a ring is given a unique identifier, which is used for addressing and

assigning data to the node. Nodes may join or depart from the network at any time. The Symphony [4] is a similar protocol but it uses the knowledge about Small World Phenomenon to optimize a long range routing of data packages. Data distribution protocols do not specify where or how the data should be stored inside a node.

Scalable, Distributed Data Structures [1] are a similar solution to the Chord and the Symphony in the way that they are visible to user applications as DHT. However, SDDS require that data should be usually kept in RAMs of nodes and stored on hard disk only when necessary. SDDS will be described in more detail in section four of this paper. We have developed our own implementation of SDDS for Linux-based multicomputers (SDDSfL), which is accessible to user applications as a block device. In that way, it brings together some features of distributed file systems and protocols for data distribution. We have also tested our implementation in order to evaluate its efficiency for different types of applications. Efficiency of other SDDS implementations was evaluated by other authors. Mokadem and Litwin designed and measured the efficiency of a special string matching algorithm for SDDS [8]. Gupta et al. [9] evaluated simple data operations (inserting, getting) for a variant of SDDS called LH*lh. Gribble et al. [6] proposed and assessed their own version of SDDS for building distributed Internet services.

3 Motivation

Most of the work on evaluation of SDDS was done around year 2000 or earlier. Then an access to single data distributed over RAMs of a multicomputer was much faster than an access to the same data stored in a local hard disk of a single server [5]. However, much has changed in the technology of hard disks and network devices since that time. Moreover, usually an access to a single data is not required, except for transactional databases. In most data manipulation algorithms packages of data are processed which may considerably diminish mean time of access to one item of the data.

The Scalable Distributed Data Structure for Linux (SDDSfL) is a variant of SDDS but also has some features of a block device. Actually, it is a hybrid solution engaging both operational and user levels of Linux [16]. As such it is very attractive for development of open source distributed applications. In [16] its usefulness for distributed transactional databases was investigated. An answer to the question of how it would work for another kind of applications particularly for data manipulation ones is of great importance. In the paper we evaluate the efficiency of such a hybrid solution for different classes of data manipulation algorithms. We also assess the impact of changes in technology on the efficiency of SDDSfL and block devices. Summarizing, the goal of our research is to discover what sort of applications may benefit form using the SDDSfL.

4 Implementation of SDDSfL

4.1 SDDS

Multicomputer systems are often used in applications that need huge data storage with a short access time. Local hard disks are not sufficient in such situations. Scalable, Distributed Data Structures [5] were created as a possible solution for this issue. All nodes of a multicomputer that use SDDS are divided into two nonexclusive groups: servers and clients. SDDS store records of data in buckets of the same size located in RAMs of servers. Data from a bucket is saved on a hard disk only when necessary e.g. when the server is shutting down. Scalable, Distributed Data Structure adjusts its capacity to current needs by splitting the buckets. When a bucket is overloaded it sends a message to a Split Coordinator (SC). The SC takes a decision which bucket should split. During a split operation a new bucket is created and it gets about half of the records stored in the bucket chosen by the SC. The client uses a special function to address records in buckets. There are two kinds of SDDS that use different addressing functions:

- RP^* – the client uses range partitioning to locate the data [10],
- LH^* – the client uses linear hashing to locate the data.

Every client has its private image of SDDS structure. After the split this image becomes outdated and the client may send request to an incorrect bucket. In such situations the message is forwarded to the correct bucket and the client receives Image Adjustment Message. The IAM contains new parameters for client's addressing function, which update its SDDS image. There is no single point of failure in SDDS structure except for the SC because no central directory is used for addressing. However, there are SDDS architectures (like RP) that do not require the SC. Other SDDS architectures are resistant to control and data failures [11, 12]. Original SDDS concept was inspiration for similar solutions, especially for Scalable, Distributed Data Structures for Internet Services [6].

4.2 SDDSfL

The Scalable Distributed Data Structure for Linux is based upon LH* version of SDDS. Each bucket uses open addressing for managing records therefore this version of LH* SDDS may be called LH*OA. A SDDSfL client is implemented on the operating system level as a kernel module. Most of the devices in Linux are accessible to user space processes through special files [13]. There are two kinds of such files: one for character devices and the other for block devices. A block device is a device that transmits data using large units called blocks. Such a device also enables random access to information stored in it. A single block consists of smaller data units called sectors. The typical size of one sector is 512 bytes. The largest blocks in Linux may have the size of eight sectors, which gives 4096 bytes (4KiB). The SDDSfL client creates an artificial block device which allows the Scalable Distributed Data Structure to look

like a hard disk. It means that any user application that works with files may use it at once. No changes in application source code are needed. The blocks of data are treated as records for SDDS. Almost all data transmissions in SDDSfL are done with the use of UDP/IP protocol. The only exception is communication between servers during a split operation, which is done using the TCP/IP protocol. The use of SDDSfL is limited to clusters.

5 Evaluation method

We used two kinds of applications for tests. The first one was a pattern matching program which implements the Boyer-Moore algorithm. It is one of the most frequently used pattern matching algorithms and requires sequential data access. The program was tested with the files of the sizes of 1MiB, 2MiB, 4MiB, 8MiB, 16MiB, 32MiB, 128MiB, 256MiB, 512MiB, 768MiB and 1024MiB. In every file the program searched for several patterns of different lengths (characters number). The second kind of tests was done with the use of file sorting program. The program used the Quicksort algorithm which requires random access to data. The files for sorting consisted of records of 2KiB size or 4KiB byte size. We used several such files which were of different sizes: 1MiB, 8MiB, 16MiB, 32MiB, 64MiB, 128MiB, 256MiB, 512MiB and 1024MiB. Every test was repeated several times and we took the average of the results of all series. In our experiments we used a multicomputer build of PC computers, each with 1.5GB random access memory and AMD Athlon 64 3200+ processor, as a test environment. We tested SDDSfL with the Gigabit (1Gb) local area network (LAN). Every node in the network runs under control of the Linux 2.6.20 (Fedora 6) operating system and is equipped with the Intel PRO/1000 GT Gigabit network adapter. The SDDSfL results were compared with those of other block devices, such as USB flash drive (Kingston Data Traveler 8GB), SATA hard disk (Seagate SATA 2, 80GB), PATA disk (Western Digital Caviar 80GB) and Ultra320 SCSI disk (Seagate 73GB 15000 rpm). The maximum size of SDDSfL block device was 4GiB (four servers were used). The Ext3 file system was used with the all data medium.

6 Experimental results

Fig. 1 - 4 show the results of pattern matching applying tests with the use of Boyer-Moore algorithm. For the four character pattern SDDSL is slower than the SATA disk and the USB flash (pendrive), but with the increasing length of the pattern the efficiency of SDDSfL improves. For longer patterns it is at least one of the most efficient block devices. In the case of the longest pattern (1024 characters), it is only second to the SCSI disk. Summarizing, the longer is the pattern the better is the performance of SDDSfL.

The pattern matching is an application which requires mostly sequential access to data. For the test which requires random access we have chosen sorting with the use

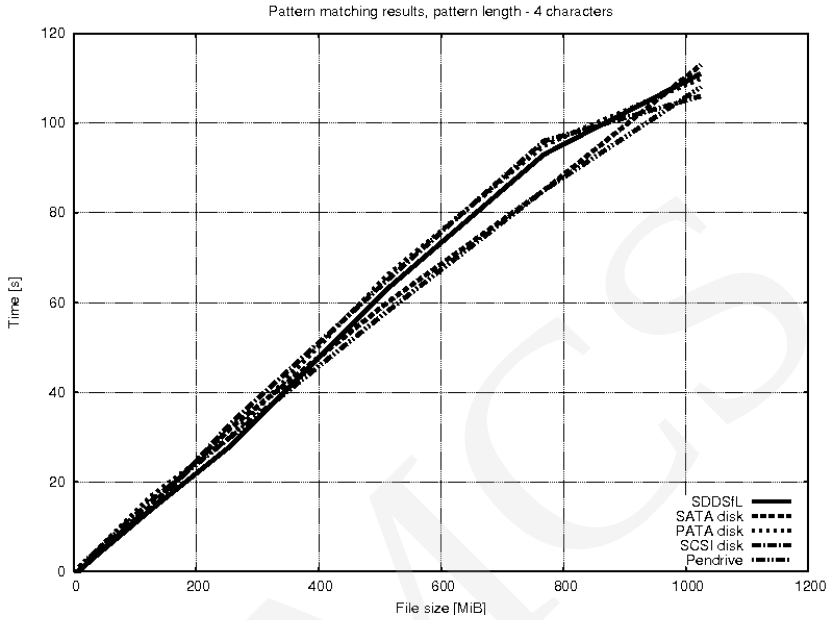

Fig. 1. Pattern matching results, pattern size - 4 characters.

Fig. 2. Pattern matching results, pattern size - 128 characters.

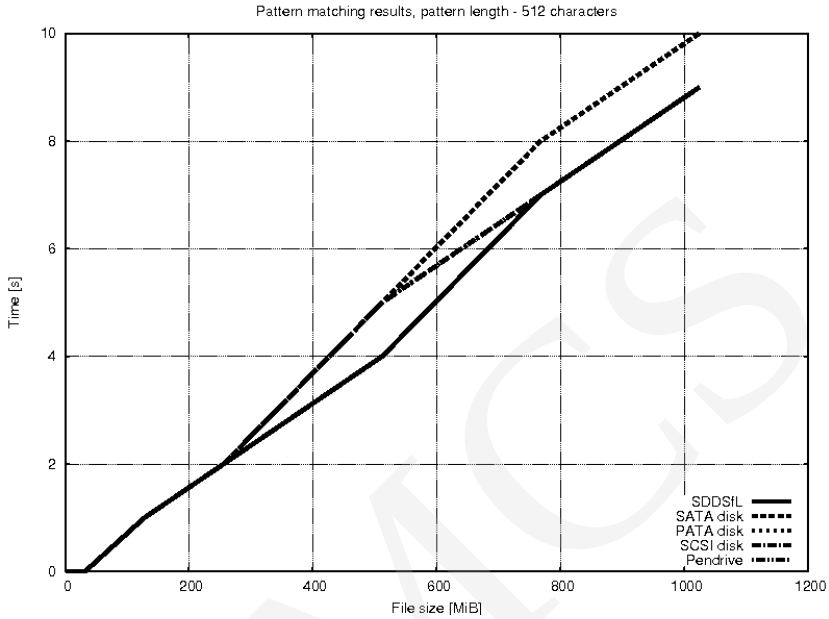

Fig. 3. Pattern matching results, pattern size - 512 characters.

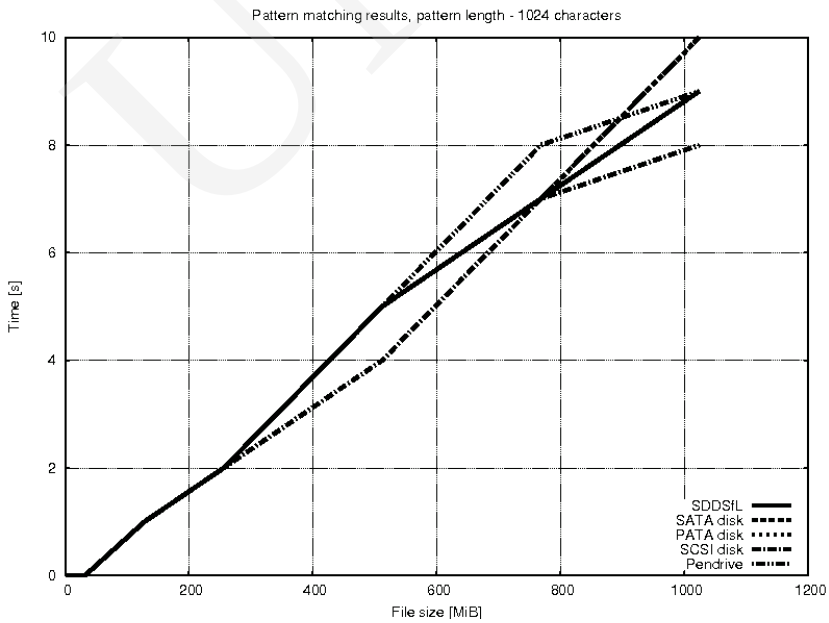

Fig. 4. Pattern matching results, pattern size - 1024 characters.

of Quicksort algorithm. The USB flash drive was excluded from the test because of its limited number of write operations. The results are shown in Fig. 5-7.

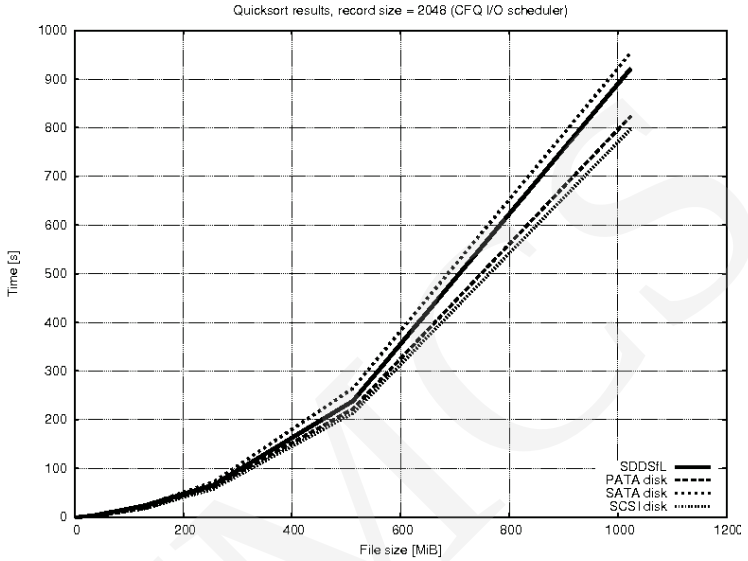

Fig. 5. Results of the Quicksort sorting (record size = 2048B, I/O scheduler = cfq).

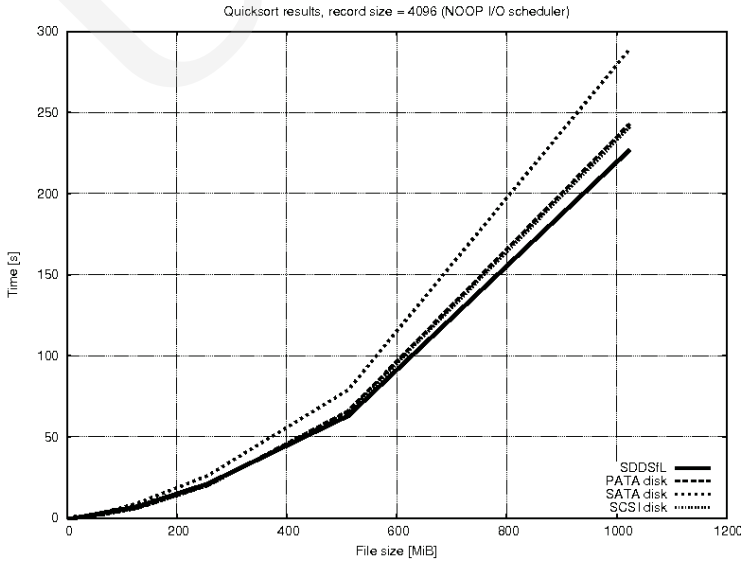

Fig. 6. Results of the Quicksort sorting (record size = 4096B, I/O scheduler = noop).

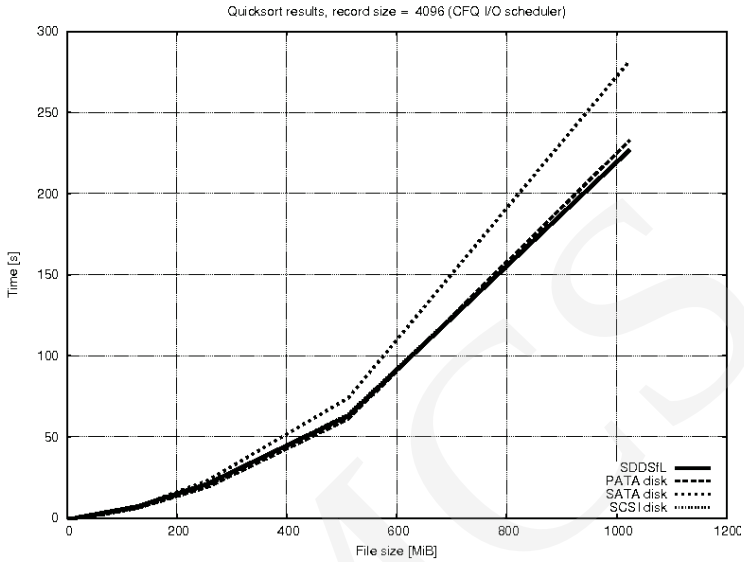

Fig. 7. Results of the Quicksort sorting (record size = 4096B, I/O scheduler = cfq).

We have tested the sorting with two kinds of I/O scheduler. The first one - CFQ is a default I/O scheduler for the Linux system. It performs an advanced optimization of requests before they are sent to hard drive [15]. The second one - NOOP is a simple I/O scheduler which only merges adjective requests [15]. The results show that hard drives benefit form the use of an advanced I/O scheduler. The SDDSfL does not use any such system and has no benefit from hardware level disk cache memory. Summarizing, the bigger are records in the sorted file, the better is the performance of SDDSfL. This is like in the case of pattern matching.

7 Conclusions

The results of the experiments show that SDDSfL could outperform the SATA and PATA disks which are commonly used in the desktop systems. However, this is only true for the applications which operate on large files consisting of relatively big records. Those applications include also transactional databases [16]. SDDSfL may also compete with more expensive server solutions like the SCSI drivers. The hard disks performance decreases with aging of head positioning mechanism. SDDSfL is free from such an issue. Implementation of original SDDS was tested for pattern matching by Mokadem et al. [8]. However, in their work the problem was defined differently and the implementation achieved good results because of parallel queries which are not implemented in SDDSfL. There is also another constraint that SDDSfL must respect. The maximum size of data transmitted between the clients and the servers is limited to the size of memory page

(usually 4KiB). This limitation is imposed by the construction of the Linux kernel block layer. SDDSfL tends to be more efficient for the applications which require random access to data. The Boyer-Moore algorithm uses heuristic methods for pattern matching. In the case of short patterns the heuristics is ineffective and the algorithm searches the file sequentially. For long patterns the search is also sequential but it is done more efficiently. The heuristics allows the algorithm to skip large parts of file in search for a pattern occurrence. This is more suitable for SDDSfL than for hard disks as Fig. 1-4 show.

From the viewpoints of scalable distributed application developers, it is worth noticing that the difference between the performances of all tested devices (including SDDSfL) is relatively small. According to the specifications the maximal bandwidth of PATA-100, Ultra-320SCSI, SATA II and Gigabit Ethernet interfaces is 800 Mbit/s, 2.6 Gb/s, 3.0 Gb/s and 1.0 Gb/s respectively. Those values are of insignificant practical meaning. The performance of hard disks is determined by their mechanical parts. The Gigabit Ethernet efficiency depends on the current network traffic and quality of overall network structure. The transfer of 4KB of data takes 98 s (seek-time + rotation delay + transfer time) in the case of 5400 RPM disks (our SATA 2 and PATA-100 disks), 65 s in the case of 7200 RPM disks and 43 s in the case of 15000RPM disks (our Ultra-320SCSI disk), which explains good experimental results of Ultra-320SCSI disk. A simple test with the use of ping command showed that the transfer time of 4KB packet in our network takes on average 224 s. The experiments also show that the PATA-100 disk is surprisingly better than the SATA II disk. This could be explained by the fact that we used the three - year old SATA II disk which is used as a local disk in a frequently used desktop computer, while the PATA-100 disk is quite new and it is only used for tests. The efficiency of SDDSfL is also dependent on network interfaces and their software drivers. During the tests we have switched from the integrated network interfaces to the Intel PRO 1000/GT networks adapters which resulted in significant improvement of SDDSfL performance. The lack of hardware level cache memory may also be the cause of the worse SDDSfL results than expected. Further experiments are required to prove it. The SDDSfL efficiency may be hampered by the traffic in local network. The tests have shown that the I/O scheduler has small influence on the performance of hard disk in the case of application that requires random access to data, but in other applications it may be significant. Eventually, solutions for the critical section problem used in the SDDSfL kernel module may play a significant part in efficiency of the whole system. That also should be tested in the future.

References

[1] Tanenbaum A. S., Steen M., Distributed Systems – Principles and Paradigms (Prentice Hall, New Jersey, 2002): 21–23.

[2] Coulouris G., Dollimore J., Kindberg T., Distributed Systems Concepts and Design (Addison Wesley Longman Limited, London, 1996): 301–338.

[3] Stoica I., Morris R., Liben-Nowell D. et al., Chord: A scalable peer-to-peer lookup protocol for internet applications, Proc. of the 2001 ACM SIGCOMM Conference (2001): 149–160.

[4] Manku G. S., Bawa M., Raghavan P., Symphony: distributed hashing in a small world, Proc. of the 4th USENIX Symposium on Internet Technologies and Systems (USENIX Association Berkeley, CA, 2003): 127–140.

[5] Litwin W., Neimat M-A., Schneider D. A., LH* – a scalable, distributed data structure ACM transactions on database systems 21(4), Kim W. (ACM, New York, 1996): 480–525.

[6] Gribble S., Brewer E., Hellerstein J., Culler D., Scalable, distributed data structure for internet service construction, Proc. of OSDI 4 (USENIX Association Berkeley, CA, 2000): 319–332.

[7] Kubiatowicz J., Bindel D., Chen Y. et al., OceanStore: an architecture for global-scale persistent storage, ACM SIGPLAN Notices 35(11), Norris C., Fenwick J.B. Jr, (ACM New York, NY, 2000): 190–201.

[8] Mokadem R., Litwin W., String-matching and update through algebraic signatures in scalable distributed data structures, Proc. of the 17th International Conference on Database and Expert Systems Applications (IEEE Computer Society Washington, DC, 2006): 708–711.

[9] Gupta V., Modi M., Pimentel A. D., Performance evaluation of the LH*lh scalable, distributed data structure for a cluster of workstations, Proc. of the 2001 ACM Symposium on Applied Computing (ACM New York, NY, USA, 2001): 544–548.

[10] Litwin W., Niemat M.-A., Schneider D., RP*: A family of order preserving scalable distributed data structures, Proc. of the 20th International Conference on Very Large Databases, Bocca J. B., Jarke M., Zaniolo C., (Morgan Kaufman Publishers Inc. San Francisco, CA, 1994): 342–353.

[11] Sapiecha K., Łukawski G., Fault-tolerant protocols for scalable distributed data structures, Lecture Notes in Computer Science 3911 (2005): 1018–1025.

[12] Litwin W., Schwarz T., LH*RS: A high-availability scalable distributed data structure using Reed Solomon codes, Proc. of the 200 SIGMOD Conference on Management of data (ACM New York, NY, 2000): 237–248.

[13] Love R., Linux Kernel Development (Novell Press, USA, 2005): 235–276.

[14] Knuth D. E., The Art of Computer Programming Vol. 3 Sorting and Searching (Addison Wesley Longman, 1998): 566–572.

[15] Shakshober D. J., Choosing an I/O Scheduler for Red Hat Enterprise Linux 4 and the 2.6 Kernel (web article), Red Hat Magazine (unpublished, 2005), http://www.redhat.com/magazine/008jun05/features/schedulers

[16] Chrobot A., Łukawski G., Sapiecha K., Scalable Distributed Data Structures for Linux-based multicomputer, 7th International Symposium on Parallel and Distributed Computing, Tudruj M., (IEEE C. S., DC, 2008): 424–428.