BLAST: Off-the-shelf hardware for building an efficient hash-based cluster storage system

George Parissis, George Xylomenos and Dimitris Gritzalis
Athens University of Economics and Business, Department of Informatics
{*parissis,xgeorge,dgrit*}*@aueb.gr*

***Abstract*— During the past few years, large, reliable and efficient storage systems have become increasingly important in enterprise environments. Additional requirements for these environments include low installation, maintenance and administration costs. In this paper we propose a hash-based storage approach, combined with block-level operating system semantics. The experimental evaluation confirms that the proposed approach is viable and can offer a cost-effective storage solution.**

I. Introduction

Key requirements of an enterprise storage system include high I/O throughput rates, resilience to component failures, capabilities for on-the-fly storage expansion and low deployment, administrative and maintenance costs. As pointed out in [1], “the ideal storage system is globally accessible, always available, provides unlimited performance and capacity for a large number of clients and requires no management”. A plethora of software and hardware storage systems has been designed to fulfill those requirements. Notable innovations include hardware-based storage area networks, distributed block-based storage systems as well as remote and distributed file systems.

Traditional hardware-based storage systems utilize disk arrays, interconnected via centralized disk controllers. These systems are able to achieve very high I/O throughput rates, while supporting redundant hardware components and hot swappable disks. Nevertheless, their deployment cost is very high. Moreover, these systems are completely proprietary and are highly customized for specific storage environments with respect to the number of supported workstations and servers, as well as to the supported storage space. As a result, they do not scale well as the demand for storage space grows beyond the initial estimations and their proprietary nature precludes compatibility with commodity hardware.

Distributed file systems are implemented in the file level of the operating system (OS) I/O stack. These systems consist of a number of hosts that store the actual data and one or more metadata servers which are responsible for managing file metadata and the physical location of every file block. These servers need to be very powerful in terms of CPU and memory as they are accessed by the clients during every file transaction. They therefore increase system complexity and tend to become bottlenecks and points of failure. While distributed file systems are optimized for specific application characteristics, disk access patterns change over time as applications evolve, resulting in a continuous need for new file systems. Moreover, there are applications that bypass the file system layer so as to avoid the overhead associated with it.

Regarding distributed storage systems implemented in the block level, we can distinguish two categories: client-server block devices and distributed block-level storage systems. The former are used to build two-node clusters by mirroring all data to a secondary storage host. However, they cannot distribute data to multiple storage devices and, thus, tend to be vulnerable to multiple disk failures, while their performance is bounded by the maximum I/O throughput provided by a single storage server. The latter follow a design where all participants in the system run the same software modules. The location of each block is kept in some kind of global data structure, which must be kept consistent across all storage servers, a fact that increases system complexity and potentially bounds their performance.

In this paper we present the design and implementation of BLAST, a Block-Level hAsh Table based STorage system, which is completely decentralized. The storage devices that form the core layer are connected in a structured overlay network, on top of which a distributed hash table (DHT) is implemented. Our system fulfills the requirements mentioned above and overcomes the deficiencies of storage systems already proposed in the literature. BLAST is a software-based system that is able to run on off-the-shelf hardware, supporting systems that are heterogeneous in terms of network, CPU, memory and storage hardware. Thus, the deployment cost of the proposed system remains very low, while its performance, as presented in the evaluation section, is sustained in high levels. BLAST is able to support data redundancy via semi-synchronous replication and device snapshots. Moreover it is a lightweight storage system that can perform at least as well as other state-of-the-art distributed storage systems, while being simple and easy to administer. The DHT perspective allows us to design a system that keeps absolutely no metadata about the physical location of the stored blocks. By decoupling file system related semantics from the actual distribution of data and by providing a low-level OS storage interface, BLAST becomes a file system-neutral storage infrastructure.

The remainder of this paper is structured as follows: In section II we briefly describe related work and in section III we present an overview of our system. Sections IV and V include a thorough analysis of the clients and the storage core layer of our system, respectively. In section VI we present an experimental evaluation of the I/O performance of our system.

Finally, section VII outlines the main conclusions of this work as well as our future research work.

II. Related Work

The Network and the Enhanced Network Block Devices (NBD, ENBD) [7], [8] are simple client server systems that support storing blocks of data at a remote storage host through a TCP connection. The Distributed Replicated Block Device (DRBD) [9] is a Linux kernel module for building a two-node high availability cluster by mirroring all data to a secondary storage host, following a RAID1 approach. Compared to BLAST, these systems are based on the client-server model and, therefore, they lack a lot of characteristics, such as storage decentralization, self-management and resilience to failures. The Global Network Block Device (GNBD) [10] provides block device access to GFS [2] over TCP/IP. Contrary to BLAST, GNBD servers access a centralized shared storage. The idea of distributed disk arrays was pioneered by TickerTAIP [11] and Petal [1]. The latter uses a master-slave replication protocol, which cannot tolerate network partitioning. Additionally, it has a period of unavailability during failover, which can cause clients to initiate unnecessary recovery actions. LeftHand Networks [12] and IBM [13] have proposed similar storage systems, but no details about them have been published. The FAB [20] design follows the same lines as Petal, but it can recover from network partitioning. All these systems require the maintenance of some global data, which involve the physical location of each block by all storage nodes, using algorithms such as [14]. This approach increases their complexity and may lead to performance deficiencies. Contrary to BLAST, they do not keep any kind of proximity state so their performance in multi-sited clusters may suffer due to interconnection latencies. A similar approach to our system has been presented in [18] but the experimental results have been very poor. Lustre [15], Ceph [16] and Panasas [17] are the most recent research contributions in the field of cluster file systems. These systems store file metadata in separate servers, which are contacted by all clients during every file transaction. Their file-level design increases system complexity and means that they cannot support applications that prefer direct access to the block-level storage interface. Note that the Global File System [2] is designed for use in a cluster environment and can be combined with the GNBD as well as with BLAST.

III. Overview

Figure 1a depicts an abstract view of the proposed system. A number of storage servers form a structured peer-to-peer network (shown as a ring), which is completely decentralized. Storage servers utilize their back-end storage devices to form a storage infrastructure, which is completely transparent to the clients. Clients are able to connect to one or more storage servers according to various connection strategies. For instance, client C maintains a primary connection with the storage server C, while a second, failover connection to server B exists. Clients may be simultaneously connected to a number of servers, which serve their I/O requests in a round-robin fashion (i.e. client A).

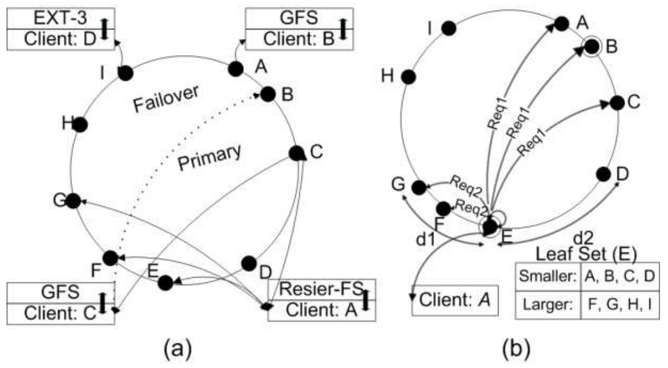

Fig. 1. System overview (a), chunk storage (b)

In order to support file system neutrality and direct integration of the proposed system, clients are implemented as virtual device drivers in the OS kernel. Using this approach, any file system can be mounted on top of a client. The type (e.g. single-access or multiple-access, read-write or read-only) and the size of the storage device, which is exported by each client, is a matter of administrative policy and, thus, irrelevant to the storage interface provided by the DHT. Whether a client is single-access or multiple-access (i.e. one or more physical hosts can access the device simultaneously) depends on the file system that is mounted on top of it. For example, mounting an EXT-3 file system on top of a virtual device precludes the simultaneous read/write access by multiple physical hosts. In contrast, mounting a file system that supports a distributed locking mechanism, such as GFS [2], on top of a virtual device, and flushes every write request to the DHT immediately (i.e. by switching off block level caching), leads to a storage environment that can be shared among several physical hosts.

IV. The client side

In this section, we provide a detailed description of the client architecture. Without loss of generality the following description is based on the Linux I/O system semantics. Clients are low-level kernel components that implement the block device driver interface exported by the Linux kernel. A block device driver provides access to storage devices (physical or virtual) that transfer randomly accessible data in fixed-size blocks. Their main data structure is a request queue into which the I/O subsystem places requests for data blocks. The description of the complete set of structures that the Linux kernel utilizes for managing I/O requests is widely discussed in [3].

We identify several advantages of implementing clients as kernel modules that lie below the file system level. First of all, the block device driver is completely agnostic to the I/O requests' contents, leading to file system neutrality, since the file system is the only responsible entity for organizing data and metadata into the flat storage space exported by the device driver. Placing the client into the kernel minimizes the context switching between kernel and user space when data are being stored, thus minimizing CPU overhead.

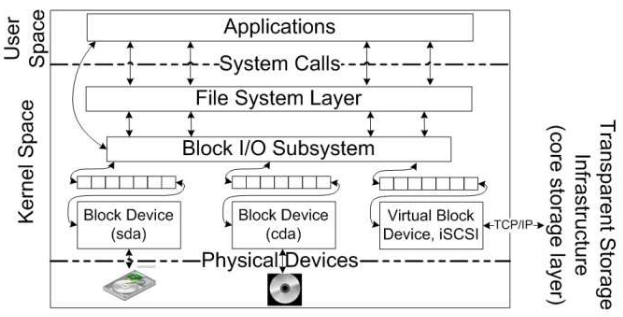

Fig. 2. An abstract view of the client

The basic functionality of BLAST clients is described below. Applications running in the user space of an OS cannot directly access the block device driver, as shown in Figure 2. Instead, they forward their I/O requests to the file system via the system calls exported by the kernel. The file system is mounted on top of the virtual device driver, receives these requests and transforms them to block I/O requests, taking into account the file metadata, which are also stored in the virtual disk. The final step is performed by the block I/O subsystem, which distributes the I/O requests for blocks to the responsible device drivers. The specific I/O scheduler for the device driver may rearrange the requests so as to contain contiguous blocks of data. In our case, the device driver pulls the requests from the request queue and forwards them to the DHT. Each client is able to forward several I/O requests to the DHT before it starts receiving responses, a fact that allows parallel I/O processing in BLAST. Regarding read requests, each client supports block prefetching, which can be configured at runtime. Finally, using the I/O control system call (*ioctl()*), several administrative tasks, such as taking device snapshots and real-time monitoring and reconfiguration can be accomplished without interfering with the device's operation.

V. The core storage layer

The core storage layer is a structured overlay network running on top of the physical network, in which storage servers are attached. Structured overlay networks, such as [4], [5], are mainly targeted for large scale networks. We argue that a similar approach, where routing takes place in hash-based fashion, is able to support several desirable characteristics in a cluster storage system, as well. More specifically, system decentralization, self-organization and transparent handling of node failures can be achieved by a structured overlay network that runs on top of a high performance network. The following description is based on the Pastry overlay network, as thoroughly described in [4], albeit it could be easily applied to any other structured overlay network. A Pastry node is assigned with a unique 128 bit nodeID, which is used to indicate its position in a circular nodeID space in the range from 0 to 2^{128-1}. BLAST servers are assigned with nodeIDs based on the computation of a cryptographic hash of their IP address. The basic assumption followed in the BLAST design is that the leafset size of each node is greater than the total number of storage nodes. In a cluster storage environment this assumption holds, since the number of storage servers cannot grow extremely large, while the leafset size is a configurable parameter of the overlay network. The above assumption translates into a structured overlay network where all nodes are one overlay hop "away" from any other node.

BLAST clients handle I/O requests for fixed-size blocks of data, whose size is statically configured during their initialization. The DHT layer processes I/O requests in larger data units, called data chunks. Data chunks are the unit of disk allocation for every storage node. For every data chunk that is to be stored to or retrieved from the DHT, a 160 bit cryptographic hash value of its number, concatenated with the device ID, is computed. A secure hash function, which is common to all storage nodes, is used. Grouping more blocks of data into a single chunk results in fewer computations and, thus, less CPU overhead. Assuming that for a time interval no failures and no new arrivals occur, a data chunk i that contains data belonging to a device d, can be stored by one or more specific storage servers, depending on the replication policy followed for this device. During a store operation, BLAST stores a chunk on the $r + 1$ BLAST nodes whose nodeIDs are numerically closer to the 128 most significant bits of the chunk ID. The replication factor r is a device-specific system parameter. This deterministic approach allows our system to be independent of any kind of metadata regarding the physical location of a data chunk. Any chunk can be located by any storage node by just computing the respective hash value. BLAST achieves a high level of load balancing without extra computational or network overhead, since both sets of existing nodeIDs and chunkIDs are quasi-uniformly distributed in their respective ID spaces.

A. Chunk storage and retrieval

A demonstration of BLAST operation, in the absence of any node failures and arrivals, is described below. In Table I, a number of sample I/O requests from a client, which is connected to a storage node, are depicted. Clients forward to the core layer storage requests of type: <*DeviceID, Type, First_Block, number_of_blocks, [data]*>.

TABLE I
Sample I/O requests

Request Type	First Block	Number of Blocks
Write	0	64
Write	8	2
Read	16	48

Assuming that the chunk size, the replication factor and the block size supported by the client are 32 kilobytes, 2 replicas and 1024 bytes, respectively, the storage node transforms the above requests to the chunk requests shown in Table II. Given a client's I/O request, the respective chunks that should be requested by the DHT are trivially calculated using the chunk size. For instance, the first request is transformed to two DHT storage requests, since it contains 64 blocks of 1024 bytes each and the chunk size is 32 Kilobytes. A chunk request contains the calculated chunkID, the chunk offset at which the requested data should be placed and the size of the request.

TABLE II
TRANSFORMED DHT REQUESTS

	Request Type	ChunkID	Offset	Size
1	Store	Hash(conc(deviceID,0)	0	32K
2	Store	Hash(conc(deviceID,1)	0	32K
3	Store	Hash(conc(deviceID,0)	8	2K
4	Retrieve	Hash(conc(deviceID,0)	16	16K
5	Retrieve	Hash(conc(deviceID,1)	0	32K

Figure 1b, illustrates how node E processes the transformed chunk requests. Regarding the first storage request, node E discovers the nodes which are numerically closer to the chunkID. The closest node is node B and, since the replication factor is 2, nodes A and C are also chosen. Then, the storage request is transmitted to the selected nodes. Regarding the second request, the numerically closest node to the calculated ChunkID is node E itself. The next two closer nodes are node G and F, which are virtually located on node's E "left side", since $d_1 < d_2$.

Figure 3 depicts the back-end storage of node B. Every BLAST node keeps a local map, into which every chunkID currently stored by the node is mapped to the physical offset of the device where the chunk is stored. BLAST nodes use Oracle BerkeleyDB [6] to maintain this mapping. The storage space for each chunk is allocated the first time a node receives a storage request for it. This space is allocated even if the request contains part of a chunk. In fact, BLAST nodes are not aware of what chunks they store (i.e. the chunk number and the responsible client), since they only keep a hashed value of this information. After allocating the storage space to hold the data, the nodes cannot know whether part of or the entire chunk is written. BLAST is independent of the back-end storage and supports heterogeneous storage ranging from local files to fast network storage devices.

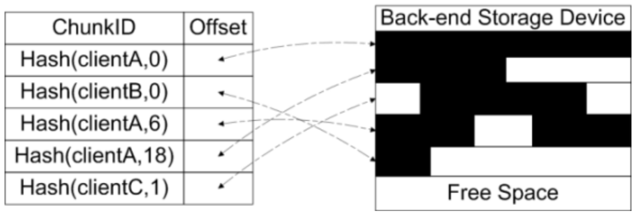

Fig. 3. Back-end storage system

Chunk retrieval is straightforward. Following the previous example, we observe that the third client request is a read request for 48 blocks of 1024 bytes each. Node E transforms this request to the retrieval requests shown in Table II and calculates the respective chunkIDs. Then, it discovers the $r + 1$ nodes that are numerically closer to them, where r is the replication factor assigned to the requesting client. If the cluster extends to multiple physical sites, the calculated nodes are compared to the nodeIDs residing in its neighborhood set. The neighborhood set involves the storage nodes that are physically closer to the present node and is maintained as analyzed in [4]. If the cluster does not extend to multiple sites, the neighborhood set is ignored and a random node is picked. The target node receives the request, locates the requested data using its local map and responds to the requesting node.

B. Chunk Replication

Our system uses semi-synchronous replication of data chunks. More specifically, storage nodes that receive a storage request mark it as pending and immediately respond to the requesting node. Assuming that the replication factor is larger than the number of nodes that can simultaneously fail, at least one storage node will respond to the requesting node, which, in turn, informs the client that its write request succeeded. If the request has been transformed to more than one storage requests, the requesting node would have waited for the responses from every node. All destination nodes keep a storage request as pending until they flush the chunk data in their back-end storage device. In this semi-synchronous manner, we minimize the response time visible to a client, while system consistency remains assured. The possible system failures that can take place while a write request is being processed are as follows: If the requesting node fails while a write request is being processed there is a high chance that some of the target storage nodes will end up holding the new version of one or more chunks, in contrast to some others that never received the request. In any case, system consistency is maintained because the client never receives a response for its I/O request and, thus, it resends this request to another BLAST node. The hash based storage approach ensures that the same storage nodes will receive the request and store the chunk data. The case where one or more destination nodes crash during this interval is analyzed in the next subsection.

C. Node failures and arrivals

Storage nodes are built from commodity hardware and so they are expected to fail quite often. However, we notice that failure frequency does not cause the well-known node churn problem that large scale overlay networks suffer from. The BLAST design follows an optimistic approach, based on the following assumptions: Storage servers are trusted nodes, which leave the network only if they fail or if they have to be reconfigured. They join the network only if they have previously crashed, reconfigured or if the storage infrastructure is being expanded or upgraded. BLAST nodes that hold the replicas of a specific chunk cannot fail simultaneously. If a node fails, it cannot rejoin the network before the overlay network gets stabilized (i.e. leaf and neighborhood sets are automatically reconfigured). A failed node always rejoins the network after a time period so that the number of live replicas for a specific chunkID does not fall under a certain critical level. A backup offline node may be used for that reason.

To handle node failures, BLAST storage nodes periodically exchange keep-alive messages. If a node is unresponsive for a period T, it is presumed failed. The node that discovered the failure notifies the nodes in its leafset (actually all nodes in the cluster) about the failure. Finally, the failed node is removed from the leafset of every participating node. The storage system must overcome the failure transparently to the connected clients. Figure 4 depicts the two possible cases that may take place when a storage node tries to store a chunk, and the responsible node for storing the data is down. In the first case (Figure 4a), node E has not discovered yet that node C has crashed. Node E calculates the $r + 1$ nodeIDs that are numerically closer to the chunkID and forwards them a storage request. Nodes A and B receive the request, mark this request as pending and respond to node E with an acknowledgment message. Node C is down and so it never receives the request. Node E waits for the last response until a timeout expires. After the timeout expiration, it sends a special message to nodes A and B indicating that they should record the specific chunkID in their local log, because a failure has potentially occurred. After receiving the response for this message, it informs the client that its request has been completed. The client suffers performance degradation for each write request it sends until node E realizes the failure. This degradation is not critical because, even in the worst case, where the client processes bulk writes, only part of the storage requests will be actually affected by the failure.

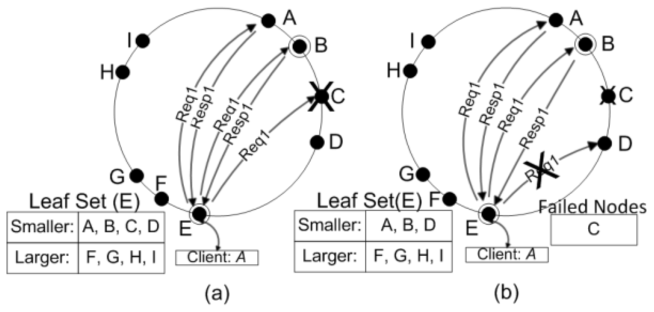

Fig. 4. Chunk storage in the middle of a failure

In the second case (Figure 4b), node E is already aware of node's C failure. It calculates the $r + 1$ nodeIDs that are numerically closer to the chunkID (i.e. nodes A, B and D). Then, it calculates the distance between the chunkID and the nodeID of A, B, D and C. Let d_i be the distance between the chunkID and node i. If $d_c > d_j$ or $d_c < d_j$, where j is A, B or D, the failure of node C does not affect this request and, thus, node E sends the three storage requests to the respective nodes. If the above formula does not hold, as in Figure 4b, node E does not send the request to node D, which temporarily "owns" node's C chunkIDs. Instead, it includes a flag in its message to nodes A and B indicating that they should record this chunkID in their log. Read request processing is as follows: if the node that processes a client request is not aware of the node failure, it creates a retrieval request and forwards it to the respective node. If the destination node was the one that has previously failed, the requesting node realizes it, because a timeout assigned to this request would expire. A second request to another replica holder is sent and the data are retrieved. In the opposite case, it uses the aforementioned formulas and forwards a retrieval request to a node that holds a replica of the chunk.

When a node joins the overlay network, it needs to initialize its state sets (i.e. leaf and neighborhood set) and to inform other nodes of its presence. The node bootstraps via another BLAST node. When servers are distributed in several physical locations, the bootstrap node should be one of the physical neighbors of the joining node so that a consistent view of the physical neighborhoods is kept. Then, the joining node asks its bootstrap node to route a join message with its nodeID as the key to the overlay network. Given that the leafset size is always larger than the maximum number of storage servers forming the overlay network, the bootstrap node can directly discover the destination node and forward this message to it. The destination node forwards its leafset to the joining node, which in turn, creates its own leafset. Finally, the joining node broadcasts its newly created leafset to all BLAST nodes. At this point all nodes have the same view of the overlay network, which is now stabilized again. If the new node just recovered from a previous failure all other nodes would have already known it because they track node failures. Therefore, the new node requests all chunk updates that took place while it has been off-line. Each BLAST node checks its local log and responds respectively. While updating the chunks in its local storage, the new node responds to all retrieval requests with a "not ready" message, whereas it processes all storage requests arriving to it as usual. Eventually, and with minimum network overhead, the new node gets updated and is ready to completely cooperate in the overlay network. On the other hand, if a new node joins the network because the storage layer is being expanded, its "overlay neighbors" are aware that they should migrate part of their data to it. In case some other node has previously failed, part of their logs should be potentially migrated. This procedure may last for hours depending on the size of the data that has to be transferred to the new node and, as in any similar bandwidth-intensive administrative task, should be scheduled for a non-peak period.

VI. Experimental Evaluation

In this section, we present an evaluation of BLAST, which includes a thorough performance benchmarking of several storage systems. The experiments took place in a gigabit Ethernet network, consisting of 8 identical desktop PCs (3.2GHz CPU, 1MB RAM, 1 SATA disk) running Ubuntu 8.10. We have installed the Lustre file system with 1 metadata server and 7 object storage devices and configured it to support three file striping policies -no striping, 256 KB and 2MB striping-. GNBD and DRBD have also been deployed with an EXT3 file system mounted on top of them. The set of experiments involved measurements from the local file system (EXT3). The BLAST core layer implementation has been based on FreePastry, a Java implementation of the Pastry routing algorithm, while system clients have been implemented as Linux kernel modules. The BLAST core storage layer has been deployed to all available PCs and a client was run in one of these machines. The BLAST deployment supported blocks and chunks of 4096 bytes and 128 kilobytes with an EXT3 file system being mounted on top of it. The IOzone [19] benchmarking tool has been used to support the experimental procedure.

The first set of experiments includes several file access patterns for various file sizes. All experiments have been repeated multiple times and the measurements were averaged. BLAST ran without support for chunk replication, so that measurements are comparable. Figure 5a depicts the I/O throughput measured during a bulk write of files whose size ranged from 100 up to 1024 MBs. DRBD and GNBD performance is getting poor as the block caching effect, provided by the OS kernel, fades out and file sizes grow. For files larger than 500 MBs, they can perform approximately at 30MB/sec. The throughput observed for local I/O is relatively higher since the local disk takes advantage from its dedicated cache memory and no network latencies occur. The Lustre file system, without support for file striping, has similar behavior, since a single storage device is used for each file copy. On the other hand, Lustre performance is significantly higher when files are striped to several storage devices. BLAST outperforms all client-server storage systems and reaches a write throughput of about 105 MB/sec even for file sizes that approximate total system RAM. We should notice that the Lustre deployment which supports file striping presumes an external mechanism that guarantees data redundancy, since, without such a mechanism, a single failure would result to the corruption of all stored files. Figure 5b depicts the I/O throughput of random writes in files of the same sizes. The block caching provided by the kernel smoothes the randomness of writes and so the results are similar to the previous case. We notice that BLAST handles more efficiently writes of small files when block caching occurs, since its memory requirements are very low (∼50MB per storage node).

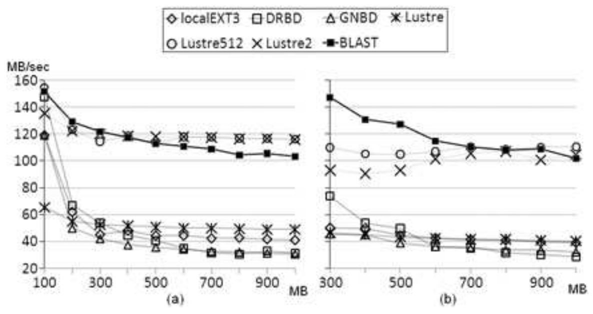

Fig. 5. Sequential (a) and random (b) write

Table III presents the read throughput measurements, in KB/sec, for file sizes ranging from 800 to 1024 MBs. IOzone performs file reading after having written the same file, therefore, the measurements for file sizes less than 800MB are of no interest, since almost the entire file is cached leading to extremely high I/O rates.

TABLE III
SEQUENTIAL READ

	800MB	900MB	1024MB
localEXT3	42544	42678	42721
DRBD	49415	49071	46508
GNBD	46038	46312	46362
Lustre	46241	46272	46304
Lustre512	95727	91428	96496
Lustre2	88166	89409	89322
BLAST	84774	85580	85144

The second experimental set involves a comparison of the same storage systems in the absence of block caching. We have performed a number of benchmarks with IOzone, with the O_DIRECT file system flag being set. Database servers usually access the device using the O_DIRECT flag, because they prefer to implement the caching level by themselves. When setting the O_DIRECT flag, the size of the file involved in the I/O does not affect the I/O throughput rate, since no caching occur and data are directly stored in the storage device. On the other hand, a very important factor that greatly influences I/O is the record size that the application passes to the file system for read or write. Figure 6a depicts the I/O throughput rate for sequential writes with record sizes ranging from 8KB up to 16MB passed to the file system. As we can observe, BLAST outperforms all other systems for record sizes between 128 KB and 4MB. It is worth noticing that the Lustre file system performs very poorly under these circumstances, for every file striping strategy. The reason for this is that Lustre runs on a customized Linux kernel, for which preemption must be disabled. As a result storage nodes are much less responsive, especially during direct I/O of small record sizes, where the observed I/O rate was about 100-400KB/sec. Note that the respective I/O rate for BLAST ranged between 15-50MB/sec.

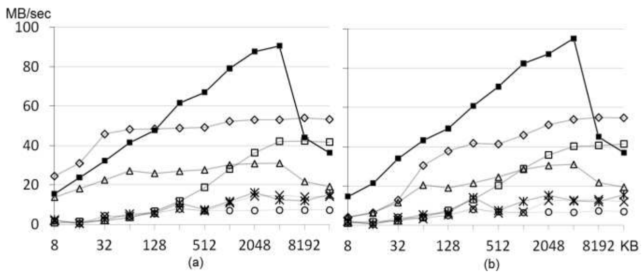

Fig. 6. Sequential (a) and random (b) write (O_DIRECT)

Figure 6b illustrates I/O performance when random writes take place. The BLAST design involves multiple storage nodes and, therefore, randomness in write I/O does not make any difference compared to sequential writes. Actually, BLAST faces every I/O as random, because, for every chunk that has to be stored in the DHT, a hash value is calculated and one or more storage nodes are chosen for storing the data. These nodes take advantage of the block level caching provided by their OS for their back-end storage and, thus, they smooth their local I/O. The reason why BLAST performance, both in sequential and random write I/O, is getting degraded for very large record sizes is related to the Linux kernel rather than to the storage core layer. An indication that supports this statement is that in both figures we see that GNBD measurements follow the same pattern as BLAST, whose clients' implementation is based on GNBD. Figure 7a and 7b illustrate the I/O throughput rates for direct sequential and random read I/O.

In these cases, BLAST performs exceptionally reaching throughput rates of about 100MB/sec, while Lustre barely reaches 30MB/sec. Compared to the measurements presented in Table III, we observe that BLAST is able to reach slightly larger I/O rates when direct I/O is used, because direct I/O involves much less CPU overhead than asynchronous I/O and, so, our system is able to process I/O requests faster. Moreover, we observe that GNBD performs much better than the other

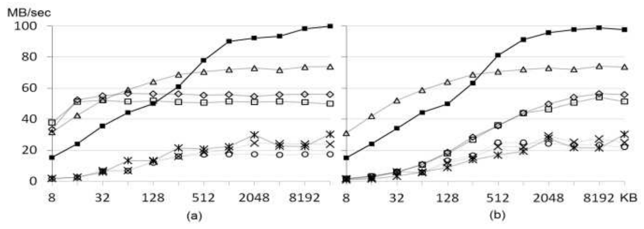

Fig. 7. Sequential (a) and random (b) read (O_DIRECT)

systems, because on the server side most of the data are locally cached by the OS. However, if the server crashes all cached data will be lost and a *fsck*-like application must run in order to make the device operable again.

Figure 8 shows a performance comparison of BLAST for three different replication factors. When chunks are not replicated, the measurements, both in asynchronous and direct I/O, are as described before. We notice that a write throughput of about 105 MB/sec is slightly less than the maximum bandwidth provided by the gigabit connection of the storage node. Actually, when no replication occurs, the storage node is bounded by the CPU. On the other hand, the network uplink almost gets saturated when chunks are replicated to one or two more storage nodes, leading to throughput rates of about 60 and 40 MB/sec, respectively. The throughput rate of read I/O is independent of the replication factor supported by the system and, therefore, read I/O sustains the same levels as before.

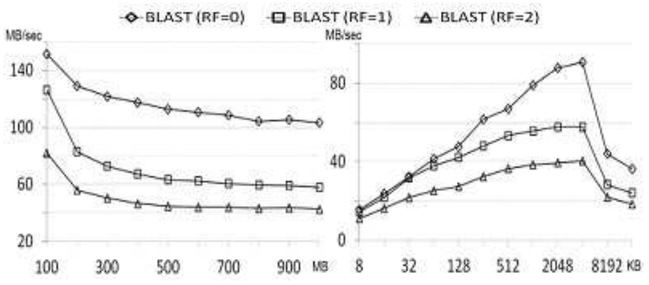

Fig. 8. Replication cost in sequential writes without (a) and with (b) direct I/O

Finally, it is worth mentioning that we have also tested the Ceph (version 0.7.1) system but this version did not support direct I/O and was not stable enough to complete the benchmarks produced by the IOzone application. On the other hand, FAB and Petal systems are not publicly available so we could not include them in the evaluation of BLAST.

VII. Conclusions

In this paper we have presented BLAST, a completely decentralized storage system that takes advantage of the special properties provided by a structured overlay network. Its block level approach makes its design relatively simple, while providing file-system neutrality. An extensive experimental evaluation confirms the performance benefits of our system in a cluster of off-the-shelf desktop PCs. BLAST proves to be competitive with respect to performance with other publicly available state-of-the-art storage systems. Part of our current research work involves BLAST evaluation in larger computer clusters that expand in several physical locations. Moreover, we are planning to test a C implementation of our system that is built on top of the Chord DHT, in order to identify potential performance improvements compared to the current Java implementation.

References

[1] E. K. Lee, C. A. Thekkath: Petal: Distributed Virtual Disks. *ASPLOS 1996*: 84-92

[2] Red Hat Inc., Red Hat GFS, http://www.redhat.com/gfs/.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman. Linux Device Drivers, pages 464-496, Third Edition. *OReilly Media Inc*, 2005.

[4] A. I. T. Rowstron, P. Druschel: Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Middleware 2001*: 329-350

[5] I. Stoica, R. Morris, D. R. Karger, et al.: Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM 2001*: 149-160.

[6] Oracle, inc. Oracle Berkeley DB Java Edition. http://www.oracle.com/.

[7] P. T. Breuer, A. M. Lopez, and A. G. Ares. The network block device. *Linux Journal*, Issue 73, March, 2000.

[8] P. T. Breuer. The enhanced network block device, http://www.it.uc3m.es/ptb/nbd/, August 2008.

[9] L. Ellenberg. Drbd 9 and device-mapper: Linux block level storage replication. In *Proceedings of the 15th International Linux System Technology Conference*, 2008.

[10] Red hat gfs 6.1: Administrator's guide, using the global network block device, May 2008.

[11] P. Cao, S. B. Lin, S. Venkataraman, and J. Wilkes. The TickerTAIP parallel RAID architecture. *ACM Trans. on Comp. Sys. (TOCS)*, 12(3):236–269, 1994.

[12] LeftHand Networks. IP-based storage area networks. *http://h18006.www1.hp.com/storage/highlights/lefthandsans.html.*

[13] IBM, inc. Intelligent Bricks - Hardware. *http://www.almaden.ibm.com/StorageSystems/projects/cibh*, April 2005.

[14] L. Lamport. The part-time parliament. *ACM Trans. On Comp. Sys. (TOCS)*, 16(2):133–169, 1998.

[15] Sun Microsystems, Inc. "Lustre File System. High-Performance Storage Architecture and Scalable Cluster File System," *http://wiki.lustre.org/*, October 2008.

[16] S. A. Weil, S. A. Brandt, E. L. Miller, et al.: Ceph: A Scalable, High-Performance Distributed File System. *OSDI 2006*: 307-320.

[17] B. Welch, G. A. Gibson: Managing Scalability in Object Storage Systems for HPC Linux Clusters. *MSST 2004*.

[18] G. Parissis and T. Apostolopoulos, A distributed hash table-based approach for providing a file system neutral, robust and resilient storage infrastructure, *NGI 2009*.

[19] IOzone Filesystem Benchmark, http://www.iozone.org/, October 2006.

[20] Y. Saito, S. Frølund, A. C. Veitch, et al: FAB: building distributed enterprise disk arrays from commodity components. *ASPLOS 2004*: 48-58.