DHTbd: A Reliable Block-based Storage System for High Performance Clusters

George Parisis,
Computer Lab
Cambridge University
Cambridge, UK
georgios.parisis@cl.cam.ac.uk

George Xylomenos and Theodore Apostolopoulos
Department of Informatics
Athens University of Economics and Business
Athens, Greece
{xgeorge, tca}@aueb.gr

***Abstract*—Large, reliable and efficient storage systems are becoming increasingly important in enterprise environments. Our research in storage system design is oriented towards the exploitation of commodity hardware for building a high performance, resilient and scalable storage system. We present the design and implementation of DHTbd, a general purpose decentralized storage system where storage nodes support a distributed hash table based interface and clients are implemented as in-kernel device drivers. DHTbd, unlike most storage systems proposed to date, is implemented at the block device level of the I/O stack, a simple yet efficient design. The experimental evaluation of the proposed system demonstrates its very good I/O performance, its ability to scale to large clusters, as well as its robustness, even when massive failures occur.**

***Index Terms*—distributed storage; distributed hash tables; resilient storage**

I. Introduction

Storage systems based on commodity hardware represent the state-of-the art in storage infrastructure, even for large enterprise storage clusters. The main reason for this shift away from proprietary hardware-based systems, is the drive for low deployment and maintenance costs, as well as the desire for scalability and decentralized operation, since proprietary systems are usually customized for specific storage environments, scaling badly as the required storage space and number of clients grow beyond the original deployment estimations.

Recent research work has mostly focused on storage systems implemented at the file level of an operating system. These systems can be distinguished in two categories: *distributed* and *shared* file systems. Distributed file systems, such as PVFS [1], Lustre [2], Ceph [3] and Panasas [4], usually export a single file namespace to multiple end-users. These systems tend to be very complex, as they must support all file system related semantics in a distributed environment. To control this complexity, they are built in a hierarchical manner: a small number of servers store all file and directory metadata and the physical location of every file block, while a potentially large number of storage servers store the actual data. Most such systems cannot take advantage of the processing and memory resources of storage servers, relying instead on very powerful metadata servers. As a result, they do not scale well, as the small number of metadata servers often becomes a bottleneck and, more importantly, a single point of failure. In addition, their hierarchical structure often dictates a static view of the computer cluster, which also results in potential scaling problems. Finally, distributed file systems are usually optimized for specific application characteristics. Disk access patterns change over time however, as applications evolve, resulting in a continuous need for new file systems. There even exist distributed applications, such as databases, that bypass the file system layer, so as to avoid the overhead associated with it, implementing their own shared-access file systems. Such applications cannot be supported by any current distributed file system.

Shared file systems, such as OCFS [5], GPFS [6] and GFS [7], assume underlying storage devices that can be physically shared across multiple storage nodes. In this manner they can provide multiple hosts with shared access to a single namespace using a distributed file locking mechanism. As described below, shared file systems are complementary to our proposed storage system.

Moving to the block level of an operating system, distributed block level storage systems can be distinguished in two categories: *client-server block devices* like NBD [8], GNBD [9] and DRBD [10] and *distributed block-level storage systems*, such as Petal [11], FAB [12], Sheepdog [13], Parallax [14] and BLAST [15]. Client-server block devices are used to build two-node clusters by mirroring data to a secondary storage host. 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 server.

In distributed block-level storage systems like Petal and FAB, all system participants run the same software; the location of each block is kept in a global data structure which must be consistent across all storage servers, using an algorithm such as [16]. This increases system complexity and potentially bounds their performance. Sheepdog uses consistent hashing to create storage volumes accessed by *virtual machines* (VMs), supporting volume management features such as snapshot, cloning, and thin provisioning. However, its architecture is tailored to VM storage volumes and its design details, as well as the supported replication mechanism, are not publicly available. Parallax operates at the block level, providing a block device interface to VMs, but it requires a shared global block store and it is not appropriate for use with well known

shared or single access file systems.

In this paper we present the design and implementation of DHTbd, a storage system built from commodity hardware that extends in many aspects our previous work, BLAST [15], an early design and prototype implementation based on similar principles. DHTbd is implemented at the block level of an operating system, thus separating file system semantics and policies from data distribution. This results in a simple yet efficient storage platform, which is fully decentralized and can scale well in terms of storage space, storage servers and end-users. Decentralization is achieved by building the core storage layer on top of a structured overlay network that uses consistent hashing to organize itself and to distribute client data across storage nodes. The interface with the storage system is integrated into the operating system as a virtual disk driver. Although this interface does not support shared-access semantics, it can be used as a virtually shared device upon which shared filesystems can be built, thus complementing the file systems mentioned above. Dealing only with data distribution simplifies the creation of mechanisms to support system expansion and storage resilience.

We also present a thorough performance evaluation that clearly indicates the potential of the proposed system. We show that DHTbd's performance is competitive compared to well known cluster filesystems in a 14 node cluster, a representative size for small and medium sized enterprises. We also show that its performance sustains high levels in a cluster of 140 virtual nodes (10 per physical host), despite the fact that 10 JVMs ran in each host. Pastry [17], the DHT used for DHTbd, scales logarithmically as the number of nodes in the overlay grows. Therefore, we expect that deploying DHTbd in a relatively larger cluster (e.g. 1000 nodes) would not severely affect its performance. As we examine in section II-A, increasing the number of storage nodes only affects the number of hops required to find the storage node that stores a specific chunk of data. Pastry's experimental evaluation [17] shows that the average number of required hops to deliver a message in an overlay network consisting of 10^3 and 10^5 nodes is about 2.5 and 4, respectively. Finally, a brief evaluation of DHTbd's behavior is presented when multiple nodes join the storage infrastructure or crash in close succession.

DHTbd exhibits two significant advances compared to BLAST. First, in BLAST all storage nodes must be aware of each other, leading to significant scalability limitations. In contrast, DHTbd does not make any assumptions regarding the size of the cluster, which can be arbitrarily large. A storage node needs to know about a small number of other nodes (the Pastry leafset whose size is usually 16 or 32), with which it exchanges hartbeat messages periodically. Second, in BLAST all blocks have to be sent first from a storage node to a proxy node and then from a proxy node to a client (and vice-versa when writing data), an approach very costly in terms of bandwidth. In contrast, in DHTbd storage clients are fully integrated in the operating system kernel, they are independent of the storage layer and there is no need for a proxy node per client in the storage overlay. As a result DHTbd's performance is doubled and the number of clients can be arbitrarily large, as it does not affect the performance of storage nodes.

DHTbd can be used in various and, potentially, diverse application scenarios in both cluster and grid computing environments. As an example, a single access virtual block device exported by DHTbd would be ideal for dividing a large, distributed storage space to multiple users and applications. A participant to this system, depending on the appplication scenario, may or may not contribute its physical storage space. Moreover, a single virtual disk could be shared by multiple physical hosts that coordinate their access to the device using one of the shared filesystems mentioned above. For example an Oracle Real Application Cluster could be deployed on top of a shared DHTbd block device, on top of which OCFS [5] would provide the required file locking guarantees.

II. Storage System Design

DHTbd consists of two independent components, the *Virtual Disk Driver* (VDD) and the *Storage Node* (SN). SNs are organized in a structured overlay network, the *Core Storage Layer* (CSL), using a *Distributed Hash Table* (DHT), such as [18], [17] and [19], therefore decentralization is based on exploiting the basic properties of these networks. From the DHTs proposed to date, we chose Pastry for the following reasons. First, the state that each node is required to maintain to participate in a Pastry network matches the replication and recovery model presented in this paper. Second, Pastry supports proximity properties that can be useful when dealing with clusters spaning multiple physical sites. Third, FreePastry [20], a Java implementation of Pastry, supports an efficient asynchronous mechanism for handling network I/O and is well documented. VDDs are dynamically loadable kernel modules, presenting the end-user with an illusion of a local disk. Each physical host runs a VDD, i.e. an in-kernel module that acts as a driver for one or more virtual disks.

A. The Core Storage Layer

The Core Storage Layer consists of SNs that form a structured, decentralized, overlay on top of the physical network; all SNs run the exact same software. There is no global cluster configuration, a fact that eases system administration: each SN only needs to be aware of another SN, which acts as its bootstrap node. For clusters spaning several physical sites, a joining SN needs to be aware of an SN physically close to it. SNs utilize their back-end storage devices to form a storage infrastructure which is fully transparent to the clients.

Each SN exports a set of access methods to the clients. The *lookup()* method returns to a VDD the set of SNs responsible for a specified set of contiguous blocks. This is the only method related to the overlay network; the VDD uses its results to direct the remaining calls to SNs. The *store()* method is used by a VDD to store a number of blocks, which, as described in Section II-B, are grouped into larger data entities, the data *chunks*. VDDs store data directly to the SNs without any intervention from the overlay network. The *retrieve()* method is used to retrieve one or more contiguous blocks of data directly from an SN. The *getChunkVersion()* method returns to a VDD the version of a data chunk currently stored by an SN. Version numbers are associated with data chunks

in order to support storage resilience, as described in Section III. Finally, the *informUpdate()* method is called by a VDD to inform an SN that it has to synchronize a chunk (and its version number) with one of the other replicas.

Each SN is assigned a unique 160 bit node ID, used to indicate its position in a circular ID space. As the number of SNs in a cluster storage infrastructure is tiny compared to the ID space, there are no guarantees that IDs computed using the SHA-1 function over an SN's IP address, as in Pastry, will be uniformly distributed. For this reason, node IDs can be produced manually or semi-automatically by the system administrator, depending on the size of the cluster and its hardware heterogeneity. The diversity in terms of CPU and memory resources in a system using commodity hardware can be dealt with by running multiple virtual SNs in a physical host. Note however that the IDs of these virtual nodes must also be evenly distributed in the Pastry ring, otherwise a failure of a single node may result in the loss of all available replicas stored in the system for a specific ID, since replicas are stored in nodes with adjacent IDs.

The basic Pastry property that we exploit is that given an overlay network consisting of N SNs, a message with a given destination ID can be routed to the node with the numerically closest node ID in less than $\log_{2^b} N$ overlay hops, assuming a stable system (b is a configuration parameter, normally a small integer). A VDD sends a lookup request for one or more data chunks whenever it receives a request by the client's block I/O subsystem. Each SN utilizes Pastry to answer the lookup requests forwarded to it by the VDDs. For each lookup request, the SN creates a unique chunk ID, computed as the SHA-1 hash value of the concatenation of the VDD's name along with the requested chunk number, and passes a lookup message with that ID to the overlay network. A VDD name may be used by a single client accessing a private storage space or shared among many clients accessing the same storage space. The message is routed based on the Pastry algorithm and, depending on the number of replicas that the CSL supports, the response will contain one or more node handles to the SNs numerically closer to the requested chunk ID.

Each SN maintains some local Pastry data structures, namely the *routing table* and the *leafset* (FreePastry does not support the neighborhood set). The routing table [17] contains a number of node IDs along with their IP addresses, used to route messages to their next overlay hop. The leafset contains L nodes (L is a configuration parameter), whose node IDs are numerically closer to the current node. The routing algorithm guarantees that every message will be routed to a node whose leafset contains the destination node or to the destination node itself; this is the case even with concurrent SN failures, unless if L nodes with adjacent node IDs fail simultaneously.

DHTbd is independent of the back-end storage that each SN has access to; a local file or a local or shared block device may be used. The only requirement for each SN is to maintain a data structure for mapping the IDs of each chunk currently stored by the node to the physical location of the data. The current Java implementation of the CSL utilizes BerkeleyDB to store these mappings in a B-Tree. The range ID queries supported by B-Trees offer a performance boost when an SN must identify the (adjacent) IDs that will be transferred to new or recovering nodes, as explained in Section III.

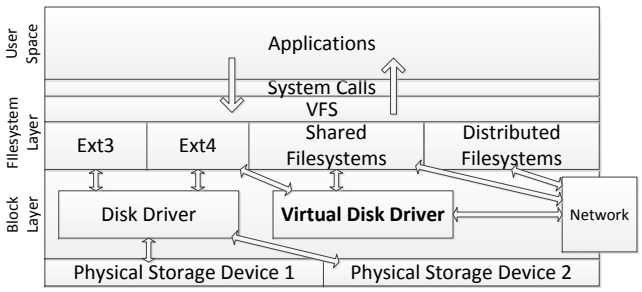


Fig. 1. Client architecture

B. Virtual Disk Drivers

Virtual Disk Drivers are implemented as in-kernel loadable modules. Figure 1 shows an abstract view of the I/O system of an operating system. As depicted in the figure, DHTbd's clients (VDDs) reside in the block layer of the I/O stack and send all I/O requests via the netwok to the CSL. Following such an approach, user applications are oblivious of the underlying storage system, therefore DHTbd can be integrated directly into a cluster storage environment without any change to the OS or applications. Moreover, VDDs are file system-neutral: they simply provide a mechanism to distribute data to multiple SNs; file system related aspects are policies implemented at a higher level. Recall that, as mentioned in Section I, VDDs do not support any shared-access semantics; these may be implemented by the file system mounted on top of them. As shown in the figure, an Ext4 file system would be adequate for single-access storage, while shared-access storage would be supported by a file system such as GFS or OCFS, using DHTbd as a virtually shared block device.

In this figure we also present how other categories of systems are placed in the I/O stack. A client of a distributed filesystem, like Lustre [2] or PVFS [1], bypasses the underlying I/O system and directly communicates with a set of data and metadata servers via the network interface. A shared filesystem, like OCFS [5], directly access a shared physical device via the network.

VDDs process I/O requests in terms of blocks, while the CSL processes them in terms of larger data units, the data *chunks*. The reason for grouping contiguous blocks into chunks is twofold. First, the CSL uses unique IDs in order to discover the nodes responsible for some data. A significant processing cost hides behind the calculation of each ID, so we chose to calculate IDs for groups of blocks. Second, by grouping blocks into chunks we improve I/O performance since for each request there is a network delay until the VDD discovers (by querying an SN) the nodes responsible for the requested data. Note also that in most I/O workloads, I/O requests contain multiple blocks of data.

In order to explain how VDDs process I/O requests, we present a simple example where the request queue of a VDD contains the following write requests: blocks 0 - 63, 128 - 129 and 1036 - 1071. Let us also assume that I/O requests contain

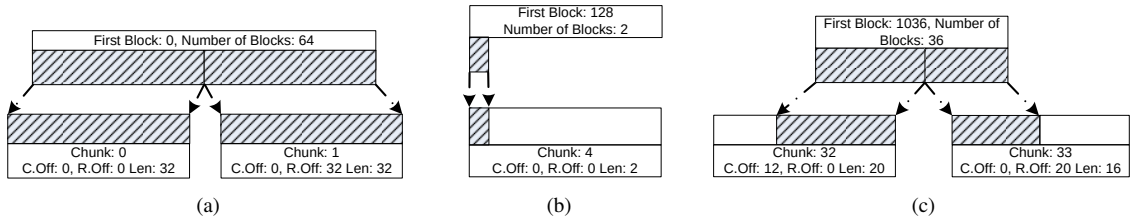


Fig. 2. Mapping blocks to chunks: 1^{st} request (a) , 2^{nd} request (b), 3^{rd} request (c)

1 KB data blocks and that the CSL processes requests for 32 KB chunks (or 32 blocks). Starting from the first I/O request, the VDD must discover the set of SNs responsible for storing the requested data. Assuming that no replication is supported, a single SN is responsible for each chunk. The VDD maps each I/O request to one or more requests for chunks using the formula $StartingBlock \div ChunkSize$.

As shown in Figure 2(a), the first I/O request is mapped to two chunks. The length of the first chunk (chunk 0) is 32 and its blocks are mapped to the first 32 blocks of the original I/O request (chunk offset=0 and request offset=0). The second chunk (chunk 1) contains 32 blocks that are mapped to the last 32 blocks of the original I/O request (chunk offset=0 and request offset=32). The second I/O request is mapped, as shown in Figure 2(b), to chunk 4; only the first 2 blocks of the chunk are mapped to the blocks of the I/O request. Note that only the mapped data of a chunk are actually stored to or retrieved from the responsible SNs. The third I/O request is mapped to two chunks as depicted in Figure 2(c). The first block of the original request is mapped to block 13 in chunk 32 (chunk offset=12 and request offset=0); starting from block 21, the remaining blocks are mapped to chunk 33 (chunk offset=0 and request offset=20).

After mapping an I/O request to the appropriate chunks, the VDD sends a lookup request for each chunk to an SN in the CSL. As described above, when an SN receives a lookup request from a VDD, it calculates a unique 160 bit ID for each chunk, using the chunk number and the VDD's name, and routes a message with this ID to the overlay network, in order to discover the set of nodes responsible for that chunk.

Depending on the leafset size, the number of SNs in the system and the replication factor supported, the number of hops required until an SN is able to respond to the lookup request may vary. To clarify this, assume that Figure 3 presents a part of the Pastry ring and that the leafset size supported by the system is 6, therefore the leafset of node G ranges from node D to node J. Now, consider a lookup message for the indicated chunkID arriving at node G. Although G is not the closest node to the requested chunkID, it may be capable of responding to the request, depending on the system's replication factor. If a single replica is stored, node G can respond that E is the responsible SN. With two replicas, node G responds that the closest SNs are E and D. However, if more than two replicas are supported then G should route the message to the destination node, E, since it is not aware about node C that lies outside its leafset.

After discovering the SNs responsible for the requested chunkID, the SN informs the VDD about them. The VDD

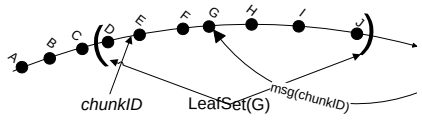


Fig. 3. Responding to a lookup request

then stores the requested data, now mapped to chunks, to these SNs (*store()*). Each of these SNs stores the data and increments the value of the chunk version it currently holds. A VDD declares the completion of the I/O request to the upper levels of the block I/O subsystem only when it has received acknowledgements from all SNs involved in the request.

Read I/O requests are processed similarly. After retrieving the responsible SNs for a specific chunk, the VDD asks them for the version of the chunk that they currently hold (*getChunkVersion()*) using the chunkID calculated during the first step. After receiving all version numbers, the VDD gets the blocks from a randomly selected SN that holds the newest replica. Different versions of a chunk may exist in the CSL due to previous failures of one or more SNs or VDDs. In such cases, the VDD does not update the SN by itself because it may not currently hold the entire chunk; recall that VDDs map I/O requests to chunks but they only store or retrieve the requested blocks. Therefore, the VDD just informs the outdated nodes that they should update their chunk from an SN known to the VDD (*informUpdate()*).

III. Handling Node Failures and New Nodes

SNs are built from commodity hardware, therefore they are expected to fail quite often. Moreover, in a cluster storage infrastructure multiple SNs may stop responding simultaneously due to a network failure. Our system follows an optimistic approach to handle node failures, based on a set of assumptions. First, SNs only leave the CSL due to failures or system reconfiguration, and they only join the CSL due to preceding failures, system reconfigurations, expansions or upgrades. Second, the replication factor is such that the probability of always locating at least one up-to-date replica per data chunk is very high. Third, if an SN fails, it cannot rejoin the network before the CSL stabilizes. Finally, a failed SN always rejoins the network after a time period that ensures that the number of live replicas does not a fall below a certain critical level; a backup standby SN may be used to ensure that this is always the case.

A. Handling node failures

To handle node failures, SNs exchange keep-alive messages. Note that, even in a large deployment, nodes exchange such messages only with their logical neighbors in their leafset, a fact that does not limit system scalability as the number of SNs grows. If an SN is unresponsive for a predefined time period, it is presumed failed. When all nodes whose leafset contained the failed node realize the failure, the overlay network is stable again. Routing entries are lazily updated, as analyzed in [17]. During the overlay reconfiguration process, some routing entries may point to failed nodes, therefore special attention has to be paid to ensure data consistency. On the other hand, while at certain time periods one or more SNs may hold outdated versions of chunks, as long as an updated version exists VDDs, will never retrieve outdated data.

The inherent decentralization of DHTbd means that SN failures do not affect all other SNs; only the L neighbors of a failed node must take action in order to stabilize their state by updating their leafsets. A node failure affects system operation when an SN is the responsible node for a specific chunk ID or is part of the path towards the responsible node. In both cases, such a failure will cause a timer to expire in the source SN and the message to be rerouted. FreePastry supports automatic message rerouting using a randomly selected SN from the present SN's leafset as the first hop. As a result, the probability that the lookup message will end up in a node able to respond to the request is significantly increased. In the worst case the source SN will keep rerouting the message until the overlay is stabilized. Note that when the replying SN is not aware of the responsible SN's failure, the response sent to the VDD after a lookup request may contain the failed SN; we explain below how VDDs handle this.

When SNs are aware of a node failure, DHTbd behaves differently. When the failure has been noted by the overlay neighbors, a lookup request will end up either in the node with the closest ID to the message ID or in a node able to respond to the lookup request. In DHTbd, chunk responsibility never changes due to node failures; it only changes when new nodes join the CSL (i.e. during system startup or system expansion). The SNs implement this policy by keeping a second leafset, apart from that required by Pastry, which we call the *shadow leafset*. When no failed nodes exist in the overlay network, the shadow leafset is identical to the regular leafset. When nodes fail however, each node that realizes the failure records the failed nodes in its shadow leafset. Therefore, the shadow leafset is a superset of the Pastry leafset. Shadow leafsets are never used for routing; an SN that receives a lookup request only uses its shadow leafset to check if it is capable of responding to it, even when no failures have occurred.

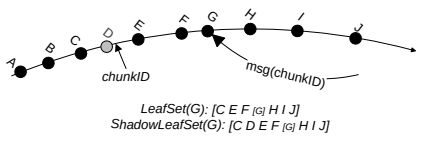


Fig. 4. Shadow leafset usage

In order to explain how shadow leafsets are used, we present a simple scenario where the indicated lookup request arrives at node G in Figure 4. We assume that node D has failed, node G is aware of its failure and the supported leafset size is 6. Since node G knows about D's failure, the shadow leafset contains D, unlike the regular leafset. The closest node to the requested chunkID is the failed node D. Since chunk responsibility never changes when nodes fail, G can respond to the lookup request if the replication factor is up to 3. If the replication factor is 1, it is obvious that all chunks stored in node D are now offline and, therefore, an I/O error will occur. If the replication factor is 2, node E is holding a replica of the requested chunk and so G's response will contain E. If the replication factor is 3, G's response will contain nodes E and C. On the other hand, if the supported replication factor is 4, node G is not capable of responding because it cannot know whether node B is closer to the chunk ID than F, since it is not aware of B at all. In this case the message has to be routed to the destination node, which, according to the leafset (not the shadow leafset), is node E. The SN replying to a lookup request will never include SNs that are known to have failed.

After describing how SN failures are managed by the CSL, we now turn to the VDDs. In general, when an SN fails while processing a request, the VDD that sent the request will realize the failure after a timeout. If the request was a lookup, the VDD will simply direct the lookup to another SN. If the request is a read or a write, it will lead to an I/O error if a single replica is kept for each chunk. However, if multiple replicas are kept, an SN failure during a write can be simply ignored, hoping that the writes to the other replicas will succeed, while an SN failure during a read causes the VDD to direct the request to another replica of the data.

Apart from the SNs, VDDs may also fail, potentially leaving the storage system in an inconsistent state. A VDD may fail during a write request, therefore there is a chance that different versions of a chunk will reside in the system. To handle such inconsistencies, each chunk stored in the system is accompanied by a version number so that VDDs can always retrieve its newest replica. Chunk versions are also used to protect VDDs from outdated replicas when previously failed nodes rejoin the overlay network. In general, using versions to maintain system consistency can become tricky. However, in our system a chunk cannot be simultaneously accessed by multiple VDDs. If shared access to a virtually shared block device is required, then a shared filesystem must be used over DHTbd. As a result, chunk versions can guarantee system consistency when SNs or VDDs crash or join the system, since a VDD will always retrieve chunks from an up-to-date SN even when node failures have previously occurred.

An extreme scenario where system consistency is still maintained is as follows. A VDD writes a chunk, but one replica is currently not available due to a temporarily broken connection. Therefore the version of this replica will not be increased. If the connection is quickly repaired, no failure will be detected. Assume now that when the next write occurs that replica is available, therefore it performs the write and increases its version. Its version is now out of sync with the others and its chunk may be out of date due to the missed update (recall that writes may affect part of a chunk).

The SN will synchronize this chunk upon a VDD's request (*informUpdate()*) when part of or the whole chunk must be read, since VDDs always check the version numbers received by the SNs. When this chunk is rarely read, the SN will continue to hold an out-of-sync replica. To address this issue, SNs periodically exchange synchronization messages with their neighbors in the ID space for the chunk versions they hold and update them if required. Again this message exchanging does not affect system scalability, since, regardless of the number of SNs in the system, a node exchanges messages with the nodes in its leafset only.

B. Handling new nodes

When an SN joins the overlay network, it must initialize its routing table and leafset and also inform other SNs of its presence [17]. The initialization procedure refers to the stabilization of the overlay network to account for the newly joined node. Depending on whether the new node had previously crashed or not, further actions must be taken by a number of SNs to bring the system to a consistent state. The SNs that updated their leafsets due to the newly arrived node can distinguish between previously failed nodes and newly added nodes by checking their shadow leafsets.

When an SN joins the overlay network for the first time, the responsibility for a set of chunks must be transferred from its neighbors to it. The new SN calculates the ID range for which it is responsible, based on the replication factor supported by the system. A simple example of the aforementioned calculation is presented in Figure 5. When a single replica for each chunk is supported, the new node N is responsible for the chunkIDs numerically closer to it. When the replication factor is 2, this range is extended and contains the IDs for which the new node is numerically closer compared to nodes A and D. After calculating the ID range, the new node requests from the $r+1$ numerically closer nodes to inform it of the set of chunks included in its ID range that they currently hold, as well as their version numbers. As mentioned in Section II-A, each SN maintains a local B-Tree that maps chunkIDs to their physical location in the local storage device. This index eases the discovery of the set of adjacent chunks in the ID space, since B-Trees support efficient processing of range queries. After retrieving this information, the new SN lazily requests the latest versions of the appropriate chunks from its neighbors. We chose to incorporate such a configurable approach to be able to adjust the degree of performance degradation that VDDs realize when changes to the storage system take place. During this process the new SN processes store and retrieve requests sent by VDDs normally. Recall that a retrieve request follows a version request and, therefore, a new node that receives a retrieve request must have already acquired the latest chunk version. On the other hand, recovering nodes only need to retrieve from their neighbors the chunks for which they currently hold an outdated version.

When a new node joins, some SNs are no longer responsible for some chunks due to the new SN. Therefore, they delete them from their local storage device, after being notified by the new SN that it is fully updated. The deletion of a set of chunks is quick since only a mapping is deleted. The storage space in the local storage device will be reused when new chunks have to be stored.

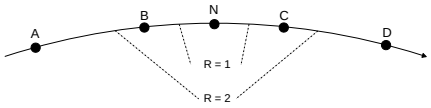


Fig. 5. ID Range Calculation

IV. Experimental Evaluation

The current implementation status of our system is briefly described below. SNs are implemented in Java, incorporating the FreePastry code to support Pastry operations. The back-end storage is implemented as a regular local file in each SN, since Java does not support direct access to raw devices without JNI. ChunkIDs are mapped to the file offset where the actual data are stored using BerkeleyDB. VDDs are implemented as Linux kernel modules for kernel version 2.6.27 and higher. They implement the block device interface provided by the kernel and directly communicate with the SNs using TCP socket connections. Network I/O is implemented by registering callbacks to each socket connection so that software interrupts are fired whenever socket buffers have space for writing or data for reading. Software interrupts are processed immediately and the actual network I/O is deferred for later processing using the workqueue interface provided by the Linux kernel.

The experimental evaluation of our system was conducted in a cluster of 14 identical PCs with dual core CPUs running at 1.2 GHz, 1 GB RAM, 1 HDD spinning at 7200 rpm and 1 Gbps Ethernet cards. All PCs were directly connected to a Gigabit Ethernet switch (HP ProCurve 1810g). The bandwidth measurement benchmarks conducted in the evaluation environment using `netperf` indicate a maximum data rate between two PCs (full-duplex) of about 800 Mbps. For all systems tested, except Lustre, we used the Ubuntu 10.04 distribution (2.6.32 Linux kernel). For Lustre we used Ubuntu 8.10 with the patched kernel provided in the Lustre web site (2.6.22).

A. I/O Throughput Comparison

In this subsection we evaluate the I/O performance of DHTbd under basic I/O workloads against the performance of three popular distributed file systems, Lustre [2], PVFS [1] and Ceph [3]. In order to keep the measurements as comparable as possible we used simple I/O workloads (i.e. sequential and random access patterns). Our system is implemented in the block level of the I/O stack, whilst creating complex I/O workloads (e.g. metadata heavy I/O patterns) mostly affects the way filesystems behave. The goal of this comparison is therefore not to find the "best" system, since DHTbd is implemented at the block level, while the other systems are implemented at the file level. Moreover, DHTbd clients mount a single access filesystem (Ext3 with journaling) on top of the VDD, therefore a direct comparison with the distributed file systems that support shared namespaces is not possible. Nevertheless, the comparison gives us clear indications regarding the potential

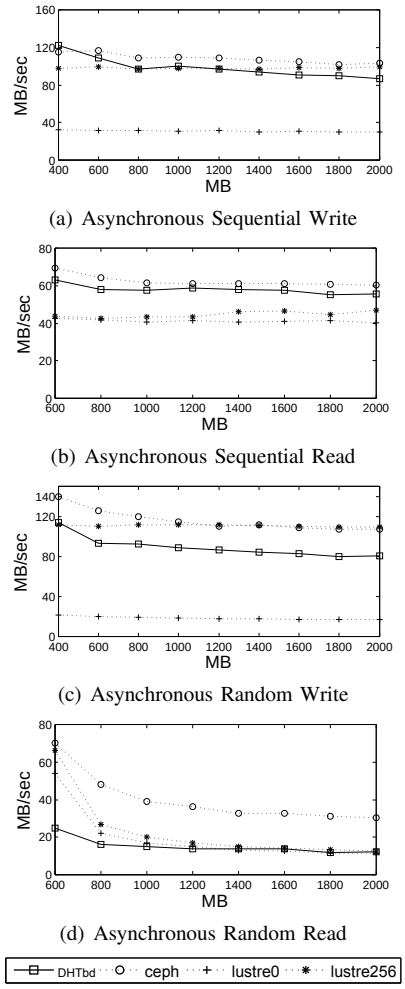


Fig. 6. I/O Throughput Comparison (AIO)

of our system, whose major characteristic is the complete decentralization and the absence of any kind of metadata. Note that the implementation of other systems implemented at the block level, such as Petal [11] and FAB [12], is not publicly available, therefore we could not evaluate them.

The measurements were captured using the `IOzone` tool. In order to make the results as comparable as possible, each system was configured as follows. Our system used 13 SNs and a single VDD, on top of which we mounted an Ext3 file system. The leafset size of each SN was 16, therefore all SNs were aware of each other and no overlay routing was required; in the next subsection we present measurements for a larger deployment (140 SNs) where routing is required. The deployment did not support chunk replication. For all distributed filesystems, we used 13 storage devices, 1 metadata server running on a host that also acts as a storage device, and a single client. In all cases, clients ran in a dedicated PC and no other network traffic was produced simultaneously.

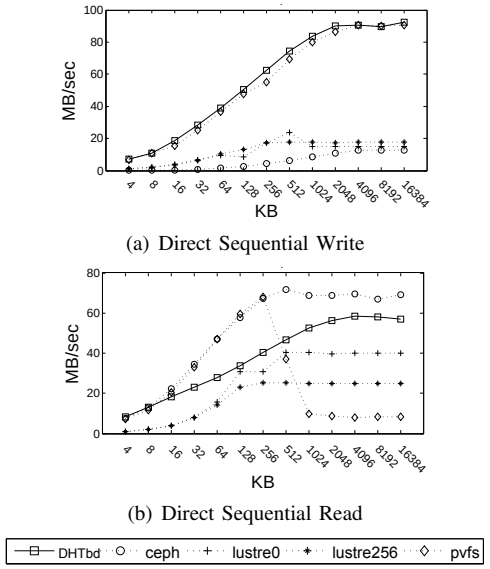
The chunk size supported by our system was configured to be 256 KB, the same as the file stripe size for Lustre and PVFS. We also measured Lustre performance with no file striping at all. Ceph's current implementation does not support stripe configuration, therefore we used the default of 1 MB. For that reason, as observed below, in some cases Ceph outperforms all other systems. Ceph's present deployment was configured to store a single copy of each file stripe. Finally, we point out that experiments were conducted multiple times and the average values are presented in all figures and tables.

Figure 6 presents performance with various asynchronous I/O workloads. Lustre256 and Lustre0 represent Lustre with and without striping, respectively. The X-axis represents the size of the file used in each experiment, ranging from 400 MB to 2 GB, while the Y-axis depicts the I/O throughput in MB/sec. Note that in asynchronous I/O mode all systems take advantage of the caching provided by the operating system, except for PVFS which bypasses the caching subsystem, and was therefore not included in these measurements.

In order to produce these measurements, we configured `IOzone` to first sequentially write an entire file to the file system exported by the storage system (in our case, a local Ext3 file system mounted on top of the VDD) and then sequentially read the entire file back. Afterwards, `IOzone` writes random parts of the file (128 KB each) in the storage system and, finally, reads random parts of the same size. The request size that `IOzone` passes to the filesystem for the sequential part of the experiment was also 128 KB; this size did not dramatically affect the results due to the asynchronous nature of the I/O. For all systems and workload types, we observe that as the size of the file grows, measurements tend to reach a stable throughput value. This is due to the caching effect fading out as the size of the data involved in the experiment grows. We do not present measurements for file sizes less than 400 MB for write I/O and 600 MB for read I/O; for these sizes the throughput rates are extremely large due to the fact that I/O is done almost entirely in RAM. For all workload types we notice that DHTbd performs similarly or even better than the distributed filesystems, especially when large files that exceed RAM size are involved in the I/O. It is worth noticing that Lustre write performance without striping is bounded by the maximum I/O capacity of a single hard disk (Figures 6(a) and 6(c)). Ceph performs best overall since, as mentioned above, the supported file stripe is much larger compared to all other systems.

In Figure 7 we present the performance of each system considered when `IOzone` uses direct I/O. Direct I/O is preferred by applications such as database systems that require control over the way data are written to the back-end storage. For direct I/O the size of the file does not affect performance, as no caching is provided by the operating system. What is critical for the performance in this case is the request size that `IOzone` passes to the file system for reading or writing. Therefore, our measurements were produced using the same procedure as above, albeit we kept the file size constant (200 MB) and measured performance for several I/O request sizes (depicted in the X-axis). Note that PVFS is included in these measurements, even though it does not support direct I/O, as PVFS bypasses the caching mechanism provided by the operating system, effectively emulating direct I/O operation.

As observed in Figure 7(a), DHTbd performs equally well as PVFS and much better than Ceph and Lustre when dealing with sequential write workloads. The maximum throughput achieved for very large request sizes is bounded by the maxi-



(a) Direct Sequential Write

(b) Direct Sequential Read

Fig. 7. I/O Throughput Comparison (DIO)

mum network throughput (about 95 MB/s). The performance of Lustre and Ceph is poor (about 18 MB/s); performing file operations directly for each filesystem request seems to be costly for these systems, unlike DHTbd on top of which we mounted a single access file system. DHTbd's performance for the sequential read workload is comparable to Ceph and much better than Lustre. It is also noticeable that PVFS performance collapses for request sizes larger than the supported file striping size. This is more likely due to TCP incast [21], a well studied problem in high performance data centers. Finally, note that even though in some cases PVFS performs better than our system, it does not actually support direct I/O and, therefore, a direct comparison between these two systems is not possible.

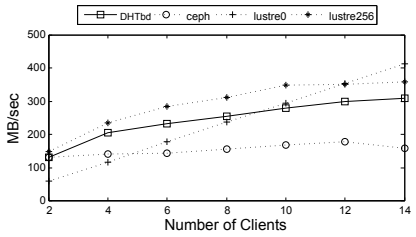
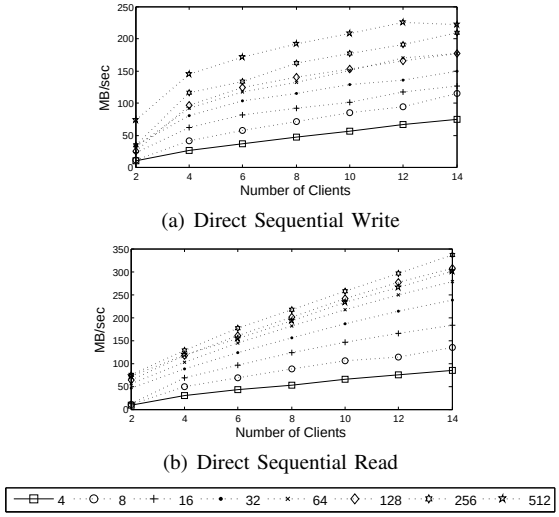


Fig. 8. Multiple clients - Sequential Write (AIO)

B. System Scalability

In this subsection we evaluate the scalability of our system. In particular, we measure the total system throughput when multiple clients access the storage system simultaneously as well as the I/O throughput of a single client when accessing a much larger deployment of the CSL. In order to measure total system throughput we used the `xdd` tool which supports distributed I/O benchmarking when multiple synchronized clients access the storage system simultaneously. We configured `xdd` so that in each step of the experiment a number of clients, increased by 2 in each step, sequentially writes and, then, reads a file for 30 s. All clients are synchronized to a timeserver in the cluster. For each step we calculate the sum of the I/O throughput rates for each client. In the presented figures, measurements represent the average values of multiple runs per experiment. The deployment details for each system are the same as in the previous subsection, except for the fact that all 14 hosts are used as storage devices. The maximum number of clients is 14, representing the extreme case where all physical hosts run a storage server as well as a client.

Figure 8 presents total system throughput when multiple clients simultaneously write a file using asynchronous I/O. DHTbd behaves very well, sustaining a write throughput of about 300 MB/s, while Ceph barely reaches the value of 180 MB/s. The most scalable deployment is that of Lustre when no file striping is supported, since for each client an idle storage device is assigned for file storage and, therefore, system performance increases linearly. Again, we did not include PVFS in these experiments. We do not show read performance, since local caching makes the results incomparable.



(a) Direct Sequential Write

(b) Direct Sequential Read

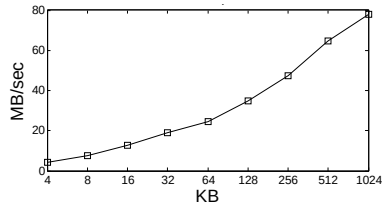
Fig. 9. DHTbd - Multiple clients (DIO)

Figure 9 depicts the total throughput of DHTbd when performing direct sequential writes (9(a)) and reads (9(b)) for request sizes ranging from 4 to 512 KB. We observe that in all cases the total throughput scales well, reaching the value of 220 MB/s for writes and 330 MB/s for reads when the request size is 512 KB. To keep the diagram readable, we omit the performance for larger request sizes but note that the measured performance was slightly better.

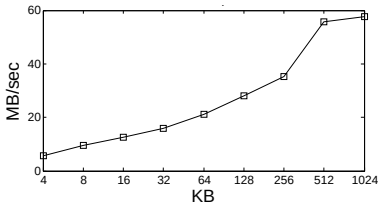
In all these measurements a single SN runs in each physical host and, therefore, the maximum number of SNs in the CSL is limited to 14. With a leafset size of 16, each SN is aware of all other nodes and no routing is required. However, in larger deployments the CSL may consist of hundreds or even thousands of nodes. DHTbd is fully decentralized and, therefore, each node does not have to be aware of all others. The cost for this decentralization is the need to route the lookups sent by VDDs. In order to evaluate this cost, for the next set of measurements we created a topology consisting of 140 nodes, by running 10 virtual nodes in each host in separate Java virtual machines. The leafset size remained 16, therefore

many lookups required overlay routing in order to reach a node capable of responding. The measurements below were captured using the `IOzone` tool and the presented workloads include direct sequential writes and reads for a single client and request sizes ranging from 4 to 1024 KB.

Comparing the measurements with 140 SNs (Figure 10) with the measurements for 13 SNs (Figure 7) we conclude that the performance of a single client does not change significantly when accessing a system almost 11 times larger. We omit the results for asynchronous I/O as they support the same conclusion. It is not clear whether the slight performance decrease is due to the overlay routing in the larger system or to the extra overhead, in terms of CPU and memory, required to run 10 Java virtual machines per physical host. Note also that this is an extreme case: the system is 11 times larger but only a single client accesses it. Increasing system size translates to more storage space and processing resources and, potentially, more aggregate bandwidth (compared to Figures 8 and 9).



(a) Direct Sequential Write



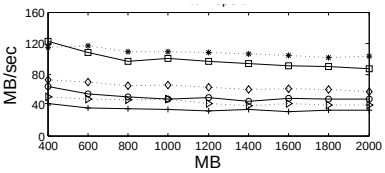
(b) Direct Sequential Read

Fig. 10. I/O Throughput in a large deployment

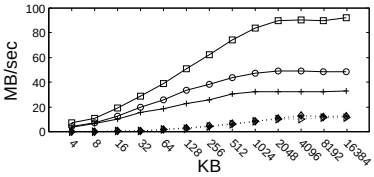
C. Replication and Self-Organization

In this subsection we discuss the performance of our system when replication is supported as well as during node failures and joins. Starting with replication, out of all the distributed file systems discussed above, only Ceph supports replication, therefore it is the only one that can be compared to our system. Apart from the replication level, the system deployments are identical to the ones presented in the beginning of this section. In the diagrams below, *systemX* refers to each system when *X* instances of each piece of data are stored.

Figure 11 shows that with asynchronous sequential writes Ceph saturates the uplink of the client, with our system performing slightly worse. Since the Ceph stripe size is 4 times larger than the chunk size supported by our system, we expect that performance would be similar if both systems grouped date in pieces of the same size. On the other hand, the performance of our system with asynchronous direct writes is exceptional compared to Ceph. We omit measurements for read workloads since replication does not significantly affect the performance results already presented in subsection IV-A:



(a) Asynchronous Sequential Write



(b) Direct Sequential Write

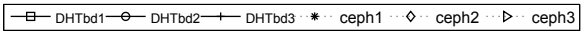


Fig. 11. I/O Throughput for various replication levels

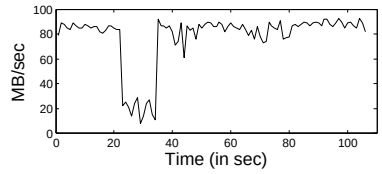
the client's VDD only needs to read a single replica of each chunk and the measured cost for reading the versions first was negligible. Obviously, DHTbd's performance drops when more replicas are stored since more bandwidth is required to support the I/O operations.

Finally, we turn to the behavior of our system when nodes join or fail. To capture the actual performance of our system, we used `Xdd` for direct sequential I/O operations with a request size of 16MB, thus avoiding the local caching of asynchronous I/O. Figure 12(a) shows the I/O performance of a single client captured every 1s. Initially, the CSL contains 7 SNs storing 4 GB of data without replication. After approximately 20 s the remaining 7 SNs join the overlay network. The figure shows that client I/O performance is degraded until all data are migrated from the existing SNs to the new ones, as described in section III-B. The performance degradation depends on the rate that existing SNs transfer chunks to the new ones (in this case, chunks are sent every 1 ms).

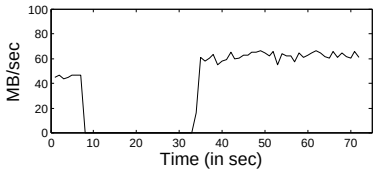
In Figure 12(b) we repeat the experiment starting with 14 SNs in the CSL, storing 2 instances of each chunk of data. At approximately 8 s we stop 7 SNs (one after the other). The SNs stopped are selected so as to leave one instance of each chunk in the CSL. In this case system operation seizes for about 20 s; this time is determined by the timeout period of the kernel client. For demonstration purposes only, we set this value to 15 s, with multiple failures increasing the downtime to 20 s. Apart from that timer, nothing prevents the system from continuing its operation. Note also that since after the failure the CSL contains half of the initial SNs, only 1 instance of each chunk is stored and, therefore, we observe an increase in client I/O performance.

V. Conclusion

We have presented the design and implementation of DHTbd, a completely decentralized distributed storage system that exploits the basic properties of the Pastry overlay network. Storage clients are implemented as Linux kernel modules that can be dynamically loaded into the operating system, providing applications with transparent access to a storage



(a) System Expansion



(b) Massive Failure

Fig. 12. Self-Organization

infrastructure that is resilient to multiple node failures and self-organized. The unique combination of a block level storage interface with a decentralized storage infrastructure makes our system self-organized as well as independent of any file system and, therefore, suitable for storage environments where multiple file systems, local or shared, must be supported.

The evaluation of our system proves its feasibility. Moreover, we have shown that the I/O performance of a single client is comparable or in many cases better than the one measured for other popular distributed file systems. The presented evaluation demonstrates that the performance of our system sustains high levels even when multiple clients access the storage system simultaneously or for a deployment of 140 storage nodes. The latter is very important, since the number of required hops in the CSL scales logarithmically with the number of SNs, meaning that we would expect similar performance even for much larger deployments. Finally, we presented experiments that show the feasibility of the proposed self-organization mechanism when new nodes join the system or fail. A significant observation was the fact that our system continued operating even after half of the SNs crashed.

ACKNOWLEDGMENTS

We would like to thank the Department of Informatics at the Athens University of Economics and Business for providing us with exclusive access to a computer lab for the experiments.

REFERENCES

[1] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "PVFS: a parallel file system for Linux clusters," in *Proc. of the USENIX ALS Conference*, 2000, pp. 28–28.

[2] SunMicrosystems, "Lustre: High-Performance Storage Architecture and Scalable Cluster File System," http://wiki.lustre.org/.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Proc. of the USENIX SOSP Conference*, 2006, pp. 22–22.

[4] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *Proc. of the USENIX FAST Conference*, 2008, pp. 1–17.

[5] Oracle, "The Oracle Clustered File System," http://oss.oracle.com/projects/ocfs/.

[6] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of the USENIX FAST Conference*, 2002, pp. 231–244.

[7] S. R. Soltis, T. M. Ruwart, and M. T. Okeefe, "The Global File System," in *Proc. of the NASA MSS Conference*, 1996, pp. 319–342.

[8] P. Breuer, A. Lopez, and A. Ares, "The Network Block Device," *Linux Journal*, no. 73, March 2000.

[9] RedHat, "Global Network Block Device," http://sourceware.org/cluster/gnbd/.

[10] L. Ellenberg, "DRBD 9 and device-mapper: Linux block level storage replication," in *Proc. of the Linux System Technology Conference*, October 2009.

[11] E. K. Lee and C. A. Thekkath, "Petal: distributed virtual disks," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 84–92, 1996.

[12] Y. Saito, S. Frolund, A. C. Veitch, A. Merchant, and S. Spence, "FAB: building distributed enterprise disk arrays from commodity components," in *Proc. of the ASPLOS Conference*, 2004, pp. 48–58.

[13] Sheepdog, "Sheepdog project," http://www.osrg.net/sheepdog/.

[14] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, "Parallax: virtual disks for virtual machines," in *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008, pp. 41–54.

[15] G. Parissis, G. Xylomenos, and D. Gritzalis, "BLAST: Off-the-shelf hardware for building an efficient hash-based cluster storage system," in *Proc. of the IFIP NPC Conference*, 2009, pp. 148–155.

[16] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[17] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proc. of the IFIP/ACM Middleware Conference*, 2001, pp. 329–350.

[18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. of the ACM SIGCOMM Conference*, 2001, pp. 161–172.

[19] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. of the ACM SIGCOMM Conference*, 2001, pp. 149–160.

[20] FreePastry, "The FreePastry project," https://trac.freepastry.org/.

[21] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan, "Measurement and analysis of tcp throughput collapse in cluster-based storage systems," in *Proceedings of FAST'08*, pp. 12:1–12:14.