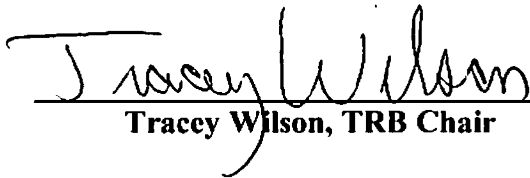
Framework for Loosely Coupled Wide Area File System Access

real testing | **real** data | **real** results

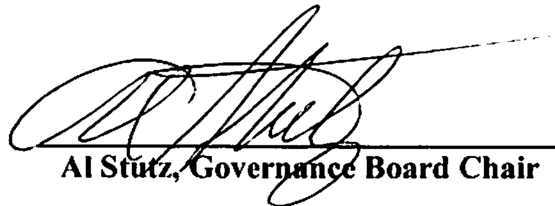
Signature Page, Release Acknowledgement

By signing below, the DICE collaborators certify that the contents presented in this document are accurate to the best of their knowledge.

Ohio Supercomputer Center



Tracey Wilson, TRB Chair



Al Stutz, Governance Board Chair

Project Title
Table of Contents

1 Project Executive Summary ... 4
2 Introduction ... 5
3 Project Overview/Description ... 6
4 Project Goals ... 7
5 Tests and Evaluations ... 9
6 Evaluation Results ... 17
7 Analysis of Deviations from Predictions ... 22
8 Conclusions ... 22
9 Proposed Next Steps ... 22
REFERENCES ... 23

1 Project Executive Summary

Taking advantage of wide-area networks to pool together multiple resources has become commonplace in recent years. Network technology has advanced to the point where geographically separated resources can be efficiently utilized. There has been much work in issues associated with running jobs across the wide area, both in inter-process communication and remote data access. Some of the most well known wide area success stories include the TeraGrid [1] and EUROGRID [2] projects.

All wide-area network applications suffer from the same fundamental constraint: the geographic distance between two resources. The propagation delay of information in transcontinental links adds in the best case a one-way latency of around 15 ms in North America, or up to 100 ms when switch and router overhead is added. When compared with latencies that are on the order of 50 µs inside the machine room, the difference of three orders of magnitude has a significant impact on how wide-area links can be used in a computation.

Despite distance being a fundamental limit, wide-area applications such as computational grids have seen success. In short the benefit of utilizing multiple resources outweighs the cost of the inter-site communication. The usage model is what enables this. It is unusual for message passing to be used to exchange information across the wide area, although it is the common mechanism used inside each cluster. The way that information is exchanged between sites is accomplished through a file system or database.

Building file systems to satisfy the wide area usage model is challenging. Due to both the large design space and persistent demand from applications, a large amount of effort has been spent in this area. Our approach to this problem is to address a limited but important class of data sharing through wide area file systems. The bulk of data, including scientific data, is “reference information,” that is, data that is generated once, never altered, and used multiple times. From this observation, we are led to a simplified design for a wide-area file system that not only performs better, but is also easier to implement than a general purpose file system.

We present here a design for a loosely coupled wide-area file system, focusing on the common use case of accessing read-only data at remote sites. Our system extends the existing Parallel Virtual File System [3] by adding external metadata mirroring and data caching components. Our implementation requires only minimal changes to PVFS, and does not alter its behavior as seen by clients, nor its performance in the local area. Client codes are completely unmodified, but gain performance improvements by accessing local caching servers.

This work contributes an extensible framework to link wide area parallel file systems together. It requires minimal changes to existing file systems by putting the details of implementation into separate server processes that use existing APIs to interact with the file

systems. The implementation presented here uses a combination of metadata and data caching to allow clients at a remote site to access data as if it were local. The framework is largely independent of the file system and is extensible to support multiple sites and multiple communication models such as peer-to-peer or encrypted.

2 Introduction

Description of the Company/Organization

OSC was established by the Ohio Board of Regents in 1987 as a statewide resource designated to place Ohio's research universities and private industry in the forefront of computational research.

Today, OSC is a fully scalable center with mid-range machines to match those found at the NSF centers and national labs.

Also in 1987, the OSC networking initiative provided the first network access to OSC's first Cray supercomputer. At the time, network access was available only within Columbus boundaries.

OSC Networking now provides Internet access to 88 Ohio higher education institutions and two million Ohioans. Since its founding, OSC Networking backbone usage has grown more than 100 percent per year.

During its first ten years, OSC focused primarily on providing high quality computing and networking services to its users. Recently, OSC has expanded its role to provide services to national high performance computing and networking groups with extensive research and education resources.

But OSC's real history -- and future -- is its clients.

As a leader in computing and networking, OSC is a resource for Ohio's scientists and engineers. The Statewide Users Group (SUG), composed of faculty members from colleges and universities across Ohio, guides OSC's user-related and technical operations. OSC Networking's Steering Committee, OSTEER, provides support from participating Ohio universities to ensure that its policies and financial structure are consistent with member needs. The Research Advisory Committee and Production Advisory committees were established in 2004, expanding the representation at OSC to include industrial and research organizations.

Past or present, computing or networking, OSC is an innovator and leader for our technology-based world. OSC's research and development initiatives put the Ohio in the position of education and technology state of the future.

3 Project Overview/Description

Project Background

The OSC DICE project can be broken up into three stages. Phase 1 was the initial work stage, Phase 2 was the planning and implementation stage, and Phase 3 was the experiment and analysis stage. Phases 1 and 3 were originally planned to be conducted in the DICE environment with Phase 2 being done in OSC's HPC research environment.

Initial work was conducted in the DICE environment utilizing two sites connected via Wide Area Network connections over the Defense Research and Engineering Network (DREN) and DONet, an ISP provider in the Springfield, OH area. These two sites were Linux based clusters maintained by DICE personnel and made available for use by OSC researchers. The original plan was to do initial testing to determine the nature of the problem. We then went off and implemented a solution, with the goal of returning to the DICE environment to do final testing. However due to resource and timing constraints we were unable to receive additional time on the systems after implementing our solution. In order to test software we modified the OSC Network Research Cluster to emulate a Wide Area Network, based on the latencies we experienced in the initial testing in the DICE environment. Our results were considered to be very similar if we utilized the DICE environment for Phase 3.

Sites and Equipment Involved

The DICE sites that were utilized in the first part of our project included NASA Goddard in Greenbelt, MD and the Avetec site in Springfield, OH. The NASA Goddard system called “Marbles” consisted of an 8 node 64 AMD Core PANTA Linux cluster with 1 GB of memory per core, while the Avetec system called “Fusion” consisted of a 16 node 64 AMD Core Linux Networx cluster with 1 GB of memory per core. Only 8 nodes of the Avetec system were used in this project.

Software resources used included the PVFS file system and custom software written by OSC. No special DICE software was required.

A Local disk was needed for servers to have low latency access to disk so as to add as little overhead as possible and isolate the effects of the network.

Public interfaces were also required on the compute nodes between the two clusters in order to accommodate direct connections between clients and servers.

Phase 3 experiments were conducted on Ohio Supercomputer Center’s Network Research Cluster. The compute nodes of this cluster are equipped with dual Opteron 250 processors, and 4 GB of RAM. The disks are all 80 GB SATA drives.

4 Project Goals

The Wide-area file system access has been studied in many contexts. Here we examine three areas of relevant work.

A. Wide-area distributed file systems

A number of popular LAN file systems like NFS [4], and the Andrew File System (AFS) [5] have been extended to wide area. AFS has been deployed successfully in the wide area [6] utilizing a set of trusted servers, each being responsible for a sub-tree of the global namespace. Our approach is different in that clients talk to local servers, rather than remote servers. We also replicate metadata for near local performance.

Punch [7] builds a framework for enabling wide-area access between NFS servers and clients distributed over a grid. In Punch, proxies interpose themselves between distributed servers and clients to enable seamless access. Similarly in our system, communication between the data and the remote sites is done through the use of wide-area file system agents. Our approach of replicating metadata is similar to WireFS [8]; however we replicate complete metadata unlike WireFS which replicates only interested sub-trees.

In versions 2 and 3, NFS is stateless and implements client side caching via a timeout mechanism [9]. In contrast, AFS is stateful, and implements caching via server callbacks. This allows for more precise caching semantics. In PVFS [3] servers are stateless which restricts us to less sophisticated cache consistency schemes. Coda [10] utilizes disconnected operations where clients can independently modify cached data and conflicts from multiple modifications are resolved in an application-specific way or by using manual resolution. We adopt Coda's idea at the data site, which on remote site disconnection, maintains a log of updates to be transmitted once the connection is established again.

LBNFS (low bandwidth network file system) [11] has contributed ways of tackling redundant data transfers by exploiting commonalities between different versions of a file and even across different files and dispatching only changes to the servers. LBNFS' techniques could be integrated with NFS and AFS for transparent deployment over the wide area, especially over low bandwidth links. Unlike content-based chunking employed by LBNFS, our approach uses simpler offset-based chunking.

B. Peer-to-peer decentralized file systems

Many read-only wide-area file systems are found in the context of peer-to-peer systems. Such a decentralized system made up of untrusted nodes is suitable for read-only or read-mostly uses. CFS [12] is a read only file system built on Chord [13] distributed hash infrastructure. CFS focuses on performance, with replica location taking worst case O(logN) hops. Unlike such peer-to-peer networks, at present our system relies on direct connections between data and remote sites. CFS uses chunk-based caching in contrast to PAST [14] which does full file caching. Our system accommodates both approaches by enabling variable chunk sizes.

One of the issues with decentralized systems is the growth of replicas and consistency management among these replicas. Pangea [15] tackles the issue of runaway growth of replicas and keeping the replicas consistent. Unlike Pangea, which pushes data updates to its replicas, our system pushes update notifications and pulls data only as required.

Shark [16] is a hybrid system made up of centralized servers for metadata, and a peer-to-peer decentralized system for data access. It is targeted for read-mostly applications. It combines the advantages of peer-to-peer systems in high availability and near local data availability, with the ease of use of a standard local file system interface. Shark caches file chunks, which are computed using the Rabin fingerprint [17] technique similar to LBNFS [11]. Shark extends xFS' [18] co-operative cache to the wide area. Unlike Shark, remote sites in our system get both data and metadata from the data's host site. It is to be noted that nodes in our system are clusters rather than individual compute elements like Shark.

C. Grid based systems

The Legion [19], [20] wide-area computing environment is an operating and resource management system. Legion follows the object oriented paradigm for system design. There is no traditional file system entity since the objects are aware of persistent vaults and the naming service helps in locating the objects. Objects communicate via RPC. Storage Resource Broker (SRB) [21] is a middleware that provides uniform access to shared heterogeneous storage resources. Both Legion and SRB tackle higher-level design issues, whereas our focus is narrower towards enabling sharing in general use systems.

The use of parallel file systems like GPFS [22] and Lustre [23] in the wide area relies on a high bandwidth infrastructure like TeraGrid [1] to provide fast and direct access to remote clients. Such systems are suitable for streaming applications, but would incur latency penalties for metadata intensive or random-access workloads. Our system specifically tackles this drawback by replicating metadata, and caching frequently used data on servers at remote sites.

ARCHITECTURE OF WIDE-AREA FILE SYSTEM

A single producer, multiple consumer paradigms is very common in many scientific applications. For example, the National Virtual Observatory [24] and Large Hardron Collider [25] generate large amounts of data which is of interest to scientists all over the world. However, due to security and logistical reasons, access to the machines which host data can not be given to everyone that wants data. This necessitates a read-only or read-mostly infrastructure, the focus of our work. Widely replicated read-only data is one of primary ways in which large scientific data grids enable access to their geographically dispersed users.

There are several ways of enabling access to remote clients. Figure 1 shows three different models for remote access. In the figure, the data site hosts a parallel file system made up

of metadata and data servers, and the three models differ in how they enable remote access to file system metadata and data. The left figure shows "remote mount" configuration that is popular in TeraGrid [1] type system. In this configuration, clients on the remote site directly access both metadata and data servers across the WAN—there are no caching components involved. In the next configuration of "metadata mirror", metadata is replicated at remote sites for improved metadata performance however data is still accessed remotely.

The final configuration of "data cache" data servers are added on the remote site that hold replicas of some data; here, clients never make direct accesses across the WAN Our design falls in the "data cache" category. In the next section we describe our architecture for wide area file access.

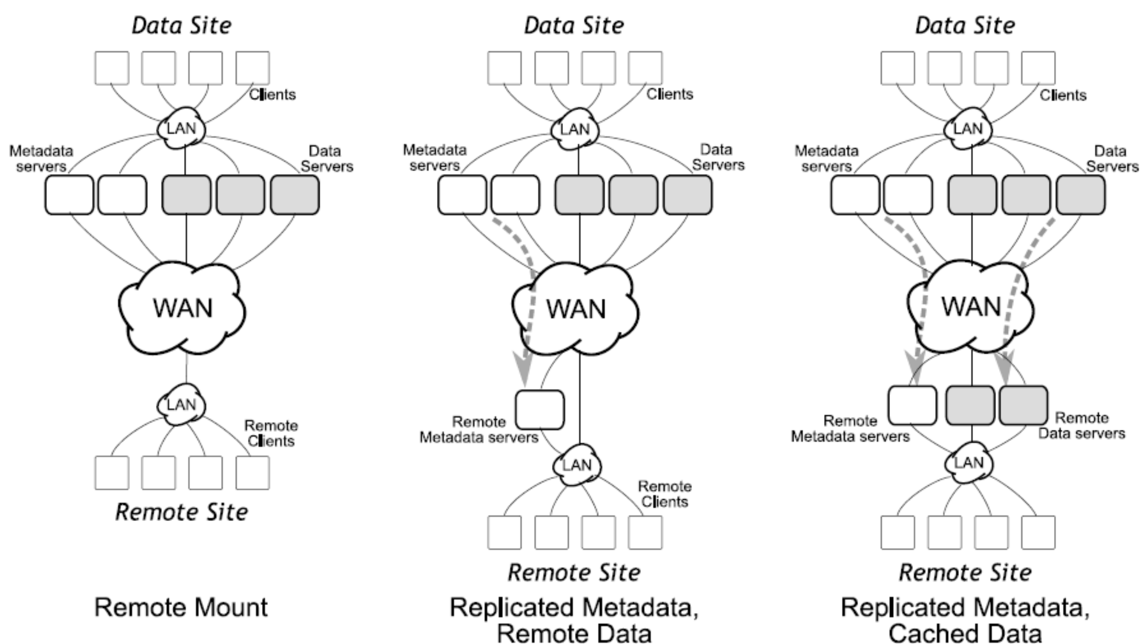


Fig. 1. Diagrams of three possible configurations for wide-area file system access. Left: remote mount; center: metadata mirror; right: data cache.

5 Tests and Evaluations

Initial testing was conducted in the DICE environment between two remote clusters connected via a Wide Area Network. The goal was to characterize the performance of parallel file systems in the Wide Area where servers and clients are on opposite sides of the WAN connection. Initial tests included throughput measurements of read and writes of various sizes. Latency is not a key measurement we were after as it is so large and will vary in the WAN. We were more concerned with the amount of throughput we can get. Comparing these throughputs to local network experiments we concluded some sort of caching mechanism is needed since even with streaming data operations full throughput can not be achieved.

A. Functional Overview

The key design decision is how to enable metadata replication and data access at the remote sites. One may configure the remote site servers to accomplish the task. However, that would involve intrusive modification to server software, as the servers need to be made aware of data dependency issues involved with the data they would be hosting. Another approach is to interpose an entity or a proxy between remote and data sites who could mediate between the two sides. This is the approach we took for our design.

Figure 2 shows a schematic of our system. In general, there are two sides involved in the operations, data and remote. At the data site, data is generated and stored. Remote sites are the systems which are interested in that data. Each site runs a wide-area network file system agent (WANFS Agent). Different sites communicate with each other through these agents.

The proxy-based architecture is quiet common in wide area and grid deployments. For example, in network weather service [26], the state communication between different sites is handled by the clique-leaders. Likewise, in our system the agent service is responsible for inter-site communication. However a key differentiator of our design is that rather than

treating the agent service as an addendum, we design our wide-area infrastructure around it. This means that most of the complexity of state maintenance is delegated to the agents. This ensures minimal modification to servers both at the data and remote sites. As far as the clients are concerned, they are unaware of the background activity.

With the agent-based architecture, the servers and clients at a particular site are oblivious to the remote sites that its agent service is connected to. It is possible for a site to act as data site for some file systems, and also to act as remote site for other file systems that it doesn't host. This functionality is handled transparently by the WANFS agents.

When a site wants to advertise data, it starts up the agent service. The agent then establishes connections with the file system servers and advertises its availability to the outside world. At this stage the data site agent has no state information other than connections to its servers. Any remote site interested in the data must also start its agent service and point to the data site. At present we don't support a global search like peer-to-peer systems [12], [16], [27] as it is orthogonal to this work. However, our architecture is flexible to adopt the search infrastructure developed by prior systems.

Once the agents establish connections with each other, they enter the bootstrap phase. During the bootstrap, the data site agent sends all file system metadata to the remote site. One thing to keep in mind is that data site is live; while metadata is being sent to the remote site it would have changed at the data site. One possible solution to this is maintaining periodic metadata snapshots of the file system and dispatching the latest snapshot. Since all communication to remote sites is routed through agents, all updates can be serialized there. The metadata changes happening concurrently can be logged at the agent and sent to the

remote site after the metadata snapshots. For the system to work, the rate of increase in size of metadata updates should be strictly less than the bandwidth between the sites, which is true for our target systems. Once the bootstrap phase completes, the proxies enter the steady state phase.

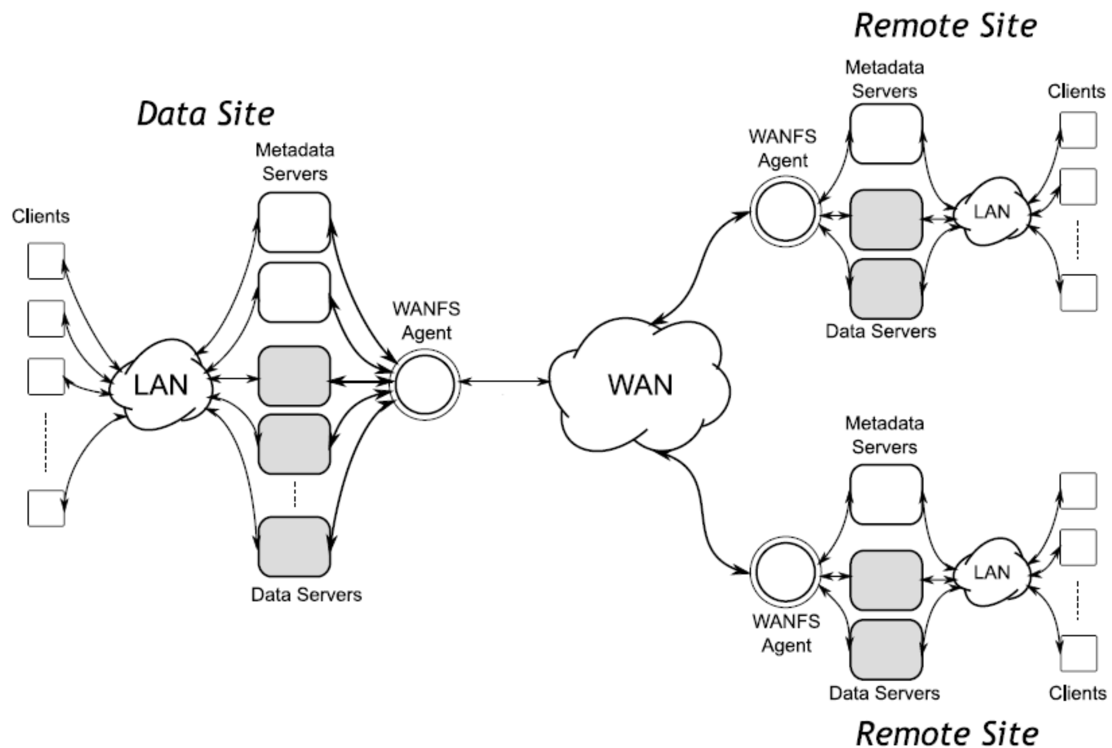


Fig. 2. Architecture of wide-area file system

The main purpose of the steady state phase is to keep the metadata between data and remote sites synchronized. At the data site, whenever a new state-modifying metadata operation is successfully completed at one of the servers, the metadata update is forwarded to the file system agent. The agent post processes and optimizes the request and schedules it for the delivery to the remote site. Upon receipt of the metadata update, the remote site de-marshals it and runs it against its servers to replicate the effect of the update. In PVFS the median size of a metadata update is around 1 kB. The data site agent has many options for optimization as we explore later. The key benefit to keeping metadata synchronized like this is that metadata operations on the remote site are able to proceed as if they were local metadata operations.

Just like the metadata update, any state-modifying data updates are also forwarded to the remote agent. The data updates only inform about the nature of the update rather than actual data involved in the update. Upon receipt of the data update, the remote proxy marks all the cached data elements that intersect with the data update as dirty. For some high demand data elements, this dirty bit forces the remote agent to pre-fetch. Since the data updates only modify the state of the cache, there is no communication necessary with the remote data servers, as the cache is maintained by the remote agent. The size of the data

update is proportional to the number of non-contiguous regions affected by the data operation. Typical size in PVFS is around 2 kB.

While metadata is pushed by the data site, data is pulled by the remote site. On a read request from the client on the remote site, the servers need to check if they have the requested data. Since the servers are unaware of the cache, they query their agent about the state. If the agent determines that cached data is up to date, the server proceeds with the request. However, if the data is uncached or stale, the remote agent requests the data from the data site agent. While data is being transferred, the client blocks until the remote server has all the data to serve it. This behavior is similar to NFS when the back end is a tape device.

In implementing the metadata replication and data caching, our goal is to create as little overhead on the data site servers as possible. We achieve this by leaving the bulk of the work up to the WAN FS agents at each side. Leaving state management to the agents is not only important, but necessary as PVFS [3] is stateless.

A key observation is that the WAN FS agent is not central to the operation of the file system at a site. If the data site agent is unavailable, its file system is completely unaffected. However, if the data site agent or the remote site agent does not function properly the remote site file system will not receive updates, but the file system will continue to function uninterrupted for files already cached. To account for fault tolerance, the agent service could be implemented by multiple cooperating processes. But that will complicate update traffic serialization.

B. Metadata Management

The primary issue with metadata management is how to synchronize the metadata between data and remote sites. Metadata can either be actively pushed out to remote sites like our system does, or the remote sites could selectively pull the metadata they are interested in. Naturally the pull model has the advantage of only requiring the remote site to store a subset of all the metadata. Thus, rather than keeping the entire metadata tree, only interested sub-trees might be kept. However, even for large numbers of files, metadata is relatively small, and our target systems can easily accommodate it.

Another issue with the pull model is lack of knowledge of current state at the remote site. Therefore, every metadata request would in-turn have to be forwarded to the data site to check for latest version. This means the pull model does not accrue any advantage unless metadata activity at the remote site is far slower than at the data site.

Pushing metadata out to the remote site does not induce any undue overhead on the servers at the data site as it is the agents that take care of ensuring that data is sent. The data site servers need not be concerned with the delivery of metadata updates. Another advantage of the push model is that it is relatively asynchronous in that not every push requires an acknowledgment. However in the pull model every request needs to be followed by a reply. The push model also only needs to be concerned with metadata that has changed. Where

as the pull model must make requests even if data is unchanged (as it can not know ahead of time).

Another approach is a hybrid of the other two. The data site regularly pushes notifications about the elements that have changed and the remote site marks those elements dirty. Whenever a dirty metadata element is requested, its state is fetched from the data site. The hybrid approach is not advantageous over the push model. The size of each metadata update is in the order of 1 kB and the bandwidth delay product of the wide area could easily accommodate a large number of updates. Moreover, one could also combine multiple metadata requests and use sophisticated compression techniques before they are dispatched to the remote site.

In parallel file systems, a single metadata operation is often broken into multiple sub-operations. For example in PVFS [3], creating a file involves creating data files, creating metadata files, setting attributes, and finally inserting a directory entry. Since operations are not atomic, failure in any step results in unwinding of the state modification affected by previous steps. The agent must take this information into account rather than blindly forwarding the updates to the remote site. Not only would it result in replicating failed operations, it would also waste network resources. The metadata updates must be forwarded only after all the sub-operations are successfully completed.

The agent must parse and understand the metadata requests. The need for parsing becomes more acute when the agent service is implemented by multiple processes. In this case the sub-operations might end up at different processes in different order. This would require for the sub-operations to be tagged and rearranged. They can be only dispatched to the remote site once the whole operation successfully completes. Metadata parsing also allows the proxy an opportunity to use its semantic knowledge to reduce multiple metadata updates into a single update.

C. Data Caching

While we replicate metadata locally for performance, the data is managed differently. High bandwidth links have shown sustainable throughput over long periods of time [22]. Therefore if workloads are dominated by long streaming applications like repeated check pointing, then the “metadata mirror” (Figure 1) configuration would be suitable. However, if data is going to be read repeatedly, shared by many applications or involves a random walk over the file, remote data access would always incur the long latency penalty. Such scenarios motivate the “data cache” approach.

In our design, metadata is replicated and data is cached. In many wide-area systems [15], [16], [12], [28] caching is frequently used to hide large latencies involved in locating and fetching data. These systems are usually peer-to-peer which exhibit high churn rate. There, aggressive replication is used not only for performance reasons but also for high availability to tolerate faults induced by peer departures and network partitioning. Our target systems do not exhibit such high churn rates and networks are relatively stable. This simplifies data replication management.

The primary issue with cache management is to determine an entity responsible for it. Usually the cache manager is co-located with the cache. This would imply that in our case, the remote site servers should be responsible for cache management. However, PVFS [3] is a stateless file system and maintaining cache at the servers would violate its core design principle. Hence we designated the responsibility of cache management to the WANFS agents. The servers are still responsible for storing the data and serving it to clients; however, they are oblivious to the state of the data they host. Making agents responsible for cache management simplifies integration with PVFS and the design is flexible enough to be used in other parallel file systems [29].

Unlike the push model of metadata, data caching uses pull model since caching is an on-demand activity and is controlled by the party interested in data rather than the producer of data. On the remote site, whenever the server gets a request for a data element, it queries its agent for the state of the element. If the data is cached and fresh, the server is allowed to service the client. Otherwise the remote agent requests the data site agent for the data. Once the data is available at the remote site, the server is signaled about the availability and it proceeds to serve the clients.

Due to the delays involved in transferring data, semantics exposed to the client applications become important. One approach is non-blocking semantics, where the server returns immediately to any client request, but possibly delivers an error message for data chunks that were unavailable. The client can retry later, after the server has had a chance to request and cache the missing data. Non-blocking semantics overlaps local data transfers with remote data transfers and tries to mask wide-area latency. However, PVFS servers were designed by assuming that either they have data or not and they employ blocking semantics. Though compatible with a stateless design, non-blocking semantics require effort in applications or libraries to handle the uncertainty of data delivery. At present we expose blocking semantics that are more natural from an application point of view.

The element that is cached may vary from chunks of a file [12], [16] to full files [14], [15]. Full file caching is advantageous when interest is shown for the whole file, the file is relatively unchanging and the file size is modest enough to amortize its transmission cost over a large number of reads. Our target applications deal with large files, which have low modification rates and the applications might be interested only in portions of the file, for example file headers to determine if the file meets their interest. This means rather than full file caching, we do fixed size chunk level caching. We use simple offset-based chunking, although more complex indexing schemes exist [11], [16]. This simplifies cache management at the cost of possibly more data transfers. In the wide area, to amortize for transmission cost, the chunk sizes are usually very large. The best chunk size is application and file dependent and is the subject of investigation in Section IV.

Our cache management uses an extent-list structure to keep information about the cached blocks. The advantage of using such a structure is that caching policy could be tailored per file. Based on the size and access pattern, either whole file or fixed chunks could be fetched. The remote site is in complete control of the caching policy and the requests for data need

to only incorporate chunk size and extents to describe data to be fetched. Another advantage is that caching policy for a file could be changed on the fly, by modifying chunk size and recalculating extents.

In order to keep replicas consistent, the data site occasionally pushes data updates to its remote sites. These update messages only inform the remote site about the chunks that have been modified. On receipt of the data update, the remote agent marks the chunk dirty and combines the new update with any previous updates targeted for the chunk. An update is usually an array of (offset, size) pairs representing contiguous file regions. When interest is shown in a dirty chunk, the remote site looks up the combined history of updates to the chunk and requests the data site for modified regions.

Even though the remote sites are clusters, they have finite resources which necessitate them to periodically purge elements from cache to make room for relevant data items. We use the common “least recently used” (LRU) approach to determine the candidates to be purged. Besides the LRU list, we also keep a “high interest list” of the most frequently used elements. For these high priority data elements, any data update notification from the data site forces the remote site to schedule an immediate data pull request for the modified regions. This ensures minimal latency for frequently accessed elements. Additionally, we are working towards incorporating automatic predictive pre-fetching [30] techniques to augment the existing system.

D. Inter-agent communication

In high-bandwidth high-latency links, it is usually better to send a small number of large messages rather than a large number of small messages. The metadata and data updates sent by the data site are very small in the range of 1– 2 kB. Factoring in the urgency of the request, the agents usually wait until they have accumulated enough updates before dispatching them. However, there are some exceptions. On the remote site, cache misses are considered urgent and request for missing data are dispatched on priority.

Though the networks for our target systems are relatively stable, they are not immune to network partitioning and rerouting. Under such circumstances, the data site keeps a log of metadata and data updates which can be played later at the remote site after the connection establishes. If the data site agent runs out of space, it discards the log and marks the remote site for bootstrap phase.

Unlike peer-to-peer systems, the agents in our system directly connect to the data source. When the number of sites involved is small and is geographically concentrated, this arrangement works fine. However, for the system to scale to large number of sites and regions, a peer-to-peer like overlay would be required. In our architecture the remote site is agnostic to the source of the data as long as it is consistent. Hence the design is extensible to a peer-to-peer system if desired. However two essential problems have to be solved for it to work: determination of replica location and maintaining replica consistency. Both these problems have been tackled by prior works in peer-to-peer file systems [12], [14], [15], [16], and these could be adapted for our system.

E. Communication optimizations

Even though large messages should be sent over the network for optimal utilization, compression is useful to contain heavy update traffic, especially during the bootstrap phase. Our design accommodates the following compression techniques:

• Content compression: Apply loss-less data compressors such as LZW [31].
How is this different than above? Ignored.
• Semantic compression: Upon queuing events that affect each other, an agent could modify them to minimize data transmission. For instance, when a file create event is followed by a removal event of the same file, both events can be deleted from the data stream as long as the create event has not reached the remote side.

• Update aggregation: Multiple updates are aggregated to amortizing latency and communication costs.

• Operation coalescing: Frequent patterns of operations can be grouped together into pre-defined aggregates. For instance, a PVFS create operation from a client results in four fundamental events to the file system. The data-side agent can recognize this pattern and replace the four with a single meta-operation that is in turn decomposed on the remote side into its constituents.

• Read combining: Data pull requests targeted for different file could be combined.

• Read coalescing: Data pull requests for different byte ranges of the same file can be merged and overlapped areas consolidated.

• Multi-client consolidation: For configurations with multiple remote proxies, a message queuing architecture can reduce bandwidth requirements. By using the "publish/ subscribe" paradigm, intermediate message queues can gather metadata changes and transmit them once across the wide area to a subscription agent that disperses the changes to interested remote agents. The use of multicast network operations may also achieve the same goal.

F. Open architecture

The choice of delegating the caching and mirroring activity to separate agents, rather than implementing it in the file system servers, brings some interesting consequences. This componentized design can be extended for other purposes beyond what has been presented here. For instance, the fundamental change notification service of the source data file system can be used to drive backups, auditing, logging or performance monitoring. It is also possible to do name-based or content-based filtering, for example, just to cache certain file types remotely; or to schedule transfers based on network conditions. These extensions are somewhat more complex to implement and to manage if integrated into the core file system.

6 Evaluation Results

Using what was discovered in the DICE environment our implementation seeks to mitigate the effects of the large latencies in the Wide Area which we believe will lead to an increase in throughput. The following experiments seek justify this.

EXPERIMENTS

The following experiments were conducted on Ohio Supercomputer Center's Network Research Cluster. The compute nodes of this cluster are equipped with dual Opteron 250 processors, and 4 GB of RAM. The disks are all 80 GB SATA drives. The cluster is segregated into 3 zones. Zone A connects to Zone B via a 1 Gbps uplink between switches. Zone B is also connected to Zone C via a 1Gbps uplink. Thus to travel from Zone A to Zone C requires crossing two single 1 Gbps links. In the experiments that follow the data site is always in Zone A and the remote site is always in Zone C. Both data and remote sites run PVFS version 2.6.3 with wide area extensions. The data site is configured to have 1 metadata and 8 data servers while remote site is made up of 1 metadata and 4 data servers.

In order to simulate a wide-area environment, we take advantage of the Linux kernel module NetEm which among other things allows one to specify a delay on packets. Figure 3 shows how NetEm fits in the kernel architecture in relation to the TCP stack and the network device. By using IP filtering, we can direct certain packets to NetEm to be delayed and allow the rest of the traffic to proceed unimpeded. This lets us emulate WAN traffic for connections to the nodes in the remote zone of the cluster. By setting the delay to a certain value we can approximate what would happen in a true wide-area network.

We use a delay of 0.5 ms on both ends, for a total round-trip time of 1 ms, to model a server room environment, or perhaps a campus-wide network at most. A round-trip delay of 6 ms reflects the network delay on a state-wide network, perhaps between two branches of a state university. We use a delay of 20 ms to model inter-state level communication. This delay is equivalent to the in-fiber flight times across a 2000 mile separation, or closer, accounting for delays in intervening switches.

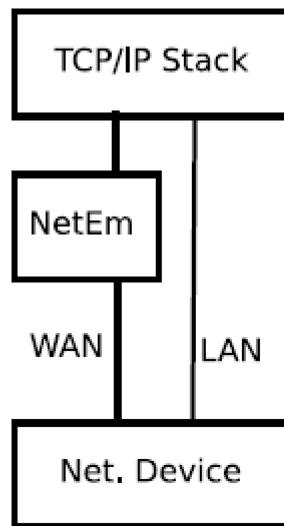


Fig. 3. NetEm TCP Kernel Architecture.

TCP tuning is a significant factor in achieving good throughput across links with high and width-delay products. Recent Linux kernels (including the 2.6.25-rc1 used here) auto-tune the socket buffers and can achieve most of the available 1 Gbps link bandwidth for all three delay settings, using the standard BIC congestion control algorithm. In order to test various file system commands, we rely on the kernel module interface for PVFS. This kernel module exposes the PVFS file system through a standard POSIX interface, allowing the use of standard Unix commands.

To analyze various aspects of the wide-area file system coupling, we consider the three different configurations, corresponding to the three diagrams in Figure 1. All three of these configurations can be tested using the three different delay values for the wide-area network.

A. Data-side overheads

The first evaluation is to determine the impact of the mirroring and caching code on the site that has the data. Ideally the overheads involved for the data-side servers to send metadata updates and data invalidations should be small. We timed two operations, only on the data side: extract the entire contents of version 2.6.24.4 of the Linux kernel archive [32], and recursively remove all those files and directories. These tests are extreme use cases that stress the file system.

	1 ms	6 ms	20 ms
None	138.6 ± 5.5	$\leftarrow$	$\leftarrow$
Metadata	137.6 ± 3.9	140.4 ± 5.4	136.8 ± 4.1
Cache	143.8 ± 5.4	142.4 ± 6.3	140.2 ± 3.6

TABLE I
OPERATION TIMES, IN SECONDS, FOR DATA-SIDE UNTAR TEST.

Table I shows the operation times, in seconds, to unpack the archive by a single client accessing the data-side servers locally. The first row shows the time when using an unmodified PVFS file system. The next two rows show the performance of "metadata mirror" and "data cache" configurations (Figure 1) respectively. For these two rows, the performance depends on the delay between the data and cache sites. There is a slight, albeit statistically insignificant, increase in the time to completion when using data caching, due to the overhead of sending messages for each new file across the wide-area link.

	1 ms	6 ms	20 ms
None	59.2 ± 0.9	$\leftarrow$	$\leftarrow$
Metadata	59.6 ± 0.8	59.8 ± 0.4	60.2 ± 0.4
Cache	79.1 ± 0.2	79.3 ± 0.2	78.8 ± 0.2

TABLE II
OPERATION TIMES, IN SECONDS, FOR DATA-SIDE REMOVE TEST.

Table II shows the operation times to remove all the files that were created in the previous test. Here, the metadata mirroring overhead is insignificant, but the performance with data caching is noticeably worse. Even though the number and size of the messages that must traverse the wide-area link is of same order in both tests, the overhead is masked in the untar test due to its longer overall runtime. In the remove test, queuing of messages to traverse the wide-area link throttles the operation of the data site agent, as it can only buffer a finite amount of update messages. The agent in turn influences the servers whose effects are finally seen by the client. The roundtrip delay does not affect the results as the update mechanism is designed so that the data site servers need not wait for a response.

In the course of the application tests to follow, no overhead could be measured at the data site. The tests presented in this section were particularly chosen to strain the link.

B. Metadata throughput

This test is performed on the remote site, unlike in the previous section. The predominant use for metadata is to look up files and to "stat" them for ownership, modification time,

size, etc. Using the same kernel tree, we perform an “ls -Ra” for each of the three remote file system configurations and for the three wide-area delays.

	1 ms	6 ms	20 ms
Remote	248.6 ± 0.3	1318.12± 0.8	4304.5 ± 0.3
Metadata	44.9 ± 0.5	169.1 ± 0.3	493.2 ± 0.2
Cache	22.6 ± 0.3	22.6 ± 0.3	22.6 ± 0.3

TABLE III
OPERATION TIMES, IN SECONDS, FOR CACHE-SIDE RECURSIVE LISTING TEST.

The results in Table III show the total time taken to perform the recursive listing of the kernel tree, which contains 24000 files and 1400 directories. The “ls” operation is implemented by the C library and kernel in a series of steps. The “readdir” file system operation is used to obtain a list of names in a directory, and may be called more than once for big directories. For each file and directory, the “getattr” file system operation is called to obtain the metadata information that is shown in the output. Further for each directory, the process starts again at “readdir”, recursively.

For the remote mount case, shown in the first row of Table III, the time to complete the recursive listing increases proportionally with increasing wide-area link latency. This is due to the inherently synchronous natures of the POSIX interfaces described above that are used for file listing: each “getattr” processes one file at a time, with no pipelining. The second row shows the case where all the metadata has been mirrored to a local server, and at first glance, one would expect the times to be independent of the wide-area link latency. However, file sizes in PVFS are calculated by the client by querying each data server in turn and summing up the partial sizes they report, because files are striped across the servers. This process generates traffic across the wide area to perform that query in the metadata mirroring case. The final row shows the times for the case of fully-cached metadata and data. These times are independent of the wide-area link characteristics, and identical to the operation time when performed on the site that hosts the data.

Data throughput is not shown, as the results are not very interesting. In the data caching case, access to cached files proceeds at LAN speeds. When data is not cached, all accesses must flow over the wide area link, limiting the aggregate throughput to all cache-side clients as dictated by the WAN link speed.

C. Database application

Several applications skim the headers of the file to find interesting data set before reading. Others like databases, execute a random walk on the file in the processes of searching and executing a query. If such applications were run over wide area they would incur latency penalty for every random access. Data cache mode addresses these drawbacks by

caching interested data locally. However, the optimal granularity of cache blocks is function of application behavior and system characteristics and hence will vary from application to application. In spite of the complexity, one can get insights by looking at the access patterns, which might help WANFS agents determine the appropriate caching policy for the file or pre-fetching rule for the elements of the file.

Databases are typically implemented using pointer chasing for operation efficiency, which results in random walk over the file. To scope out the optimal cache block granularity for database, we created a 1.1 GB database file, with 32 tables, each table with 262144 rows. Every table has two columns, key and value, with key being used as index into the table. We use SQLite [33], an embedded database as our test application. Our test program runs a fixed set of queries against the database. Each query results in a random walk:

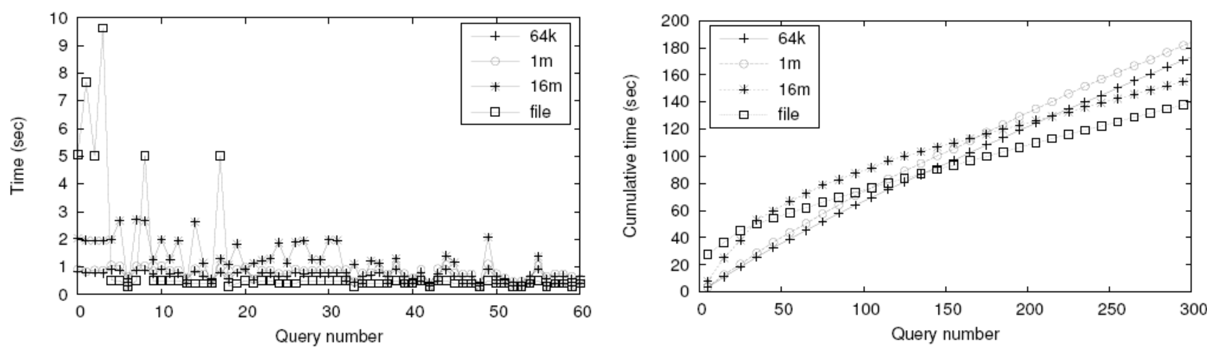


Fig. 4. Database application. Left: time for each query; right: cumulative sum. Wide-area round-trip time 20 ms.

a table is selected at random, then a row is selected and read. This behavior mimics web-query processing. Time for a random walk is calculated and reported. For our experiments we seed the pseudo-random number generator with a fixed value to repeat the sequence across different configurations. In this experiment 4 chunk (block) sizes were tested: 64 kB, 1 MB, 16 MB, and full file. The WAN latency was set to 10 ms.

In Figure 4, the left plot shows the time for a sequence of queries. The number of queries shown is restricted to the initial section of the sequence depicting the most interesting events. In the plot, the full file caching has about 6–7 peaks for the initial queries when the cache is getting populated. In PVFS, a file is split across different IO servers for increased throughput, in this case a file is made up of 8 elements and every peak represents cost of fetching 1 or 2 files. All curves converge together as we move towards later queries when most of the elements are cached. This is clearly seen as the density of spikes reduces towards later queries. One important point is high latencies even for cached blocks. This is due to PVFS kernel module, which does not cache data blocks locally, and each data access involves a fetch from the server. The cost is further increased by SQLite's pager module which reads 1 kB at a time, resulting in heavy network traffic.

Deciding on optimal caching scheme becomes clearer if we look at Figure 4 right plot, which shows the cumulative latencies of different queries. From the plot, full file and 16 MB

start with heavy penalties, however as they get data elements into cache they increase at a slower pace than 64 kB and 1 MB. Between 64 kB and 1 MB, 64 kB has lesser slope indicating for this application to appropriately amortize transmission cost over multiple reads, it is necessary to go over two orders of magnitude higher block sizes.

If the number of accesses is going to be less than 100, it is better to cache at smaller blocks. However, if the file is expected to be used more often and access pattern is random, then it pays to fetch and cache the whole file. Such insights could be built into pre-fetching rules for the file. Our architecture which has support for per-file caching scheme can exploit this information for better performance.

7 Analysis of Deviations from Predictions

The results obtained tracked well with the expected results. During Phase 1 of the project it was determined that the large latency was the limiting factor and mitigating that would achieve near local file system performance. By utilizing the file system agents to cache metadata we are able to realize near local file system performance for metadata operations on the remote file systems. Caching of file data also enables performance increases and allows for higher throughput. By virtue of our system design and the architecture of PVFS these caching mechanisms add very little overhead to the data side.

8 Conclusions

In this paper we demonstrated an extensible framework for realizing a wide-area read-only parallel file system. Our system delivers near local metadata performance to remote systems, while imposing minimal overhead at the data site. Interested data is transparently cached and managed at the remote site without the server involvement. Our architecture is flexible, involves minimal modification to the file system software and could be adopted for other parallel file systems.

9 Proposed Next Steps

In the immediate future we plan to expose non-blocking semantics to the clients at the remote site to overlap local data transfers with wide-area data transfers. We will also incorporate per-file caching policy and on-the-fly policy adaptation based on file access patterns. In conjunction to that, we are augmenting WANFS agents with automatic pre-fetching capabilities.

In the long run, we plan to augment agent connectivity using a peer-to-peer like overlay to extend our design to larger number of dispersed systems. Though our focus would still be read-mostly applications, we are trying to incorporate ideas from earlier wide-area systems with read/write capabilities. Expanding to larger number of systems would also require designing the WANFS agents to be file system agnostic to enable communication between heterogeneous systems.

REFERENCES

[1] C. Catlett, "The TeraGrid: a primer," http://www.teragrid.org/.
[2] D. Mallmann, "Eurogrid," http://www.eurogrid.org/, 2004.
[3] 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 4th Ann. Linux Showcase and Conf., 2000, pp. 317–327.
[4] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network file system (NFS) version 4 protocol," IETF RFC 3530, Tech. Rep., Apr. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3530.txt
[5] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: A distributed personal computing environment," Comm. of the ACM, vol. 29, no. 3, pp. 184–201, 1986.
[6] M. Spasojevic and M. Satyanarayanan, "An empirical study of a widearea distributed file system," ACM Trans. Comput. Syst., vol. 14, no. 2, pp. 200–222, 1996.
[7] R. J. Figueiredo, N. H. Kapadia, and J. A. B. Fortes, "The PUNCH virtual file system: Seamless access to decentralized storage services in a computational grid," in HPDC '01: Proc. of the 10th IEEE Int. Symp. on High Perf. Dist. Comp., 2001, pp. 334–344.
[8] J. Liang, A. Bohra, H. Zhang, S. Ganguly, and R. Izmailov, "Minimizing metadata access latency in wide area networked file systems," in HiPC '06, 2006, pp. 301–312.
[9] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," SIGOPS Oper. Syst. Rev., vol. 23, no. 5, pp. 202–210, 1989.
[10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A highly available file system for a distributed workstation environment," IEEE Trans. Comput., vol. 39, no. 4, pp. 447– 459, 1990.
[11] A. Muthitacharoen, B. Chen, and D. Mazi`eres, "A low-bandwidth network file system," in SOSP '01: ACM Symp. on Op. sys. pricp., 2001, pp. 174–187.
[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Widearea cooperative storage with CFS," SIGOPS Oper. Syst. Rev., vol. 35, no. 5, pp. 202–215, 2001.
[13] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," SIGCOMM Comput. Commun. Rev., vol. 31, no. 4, pp. 149–160, 2001.
[14] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," SIGOPS Oper. Syst. Rev., vol. 35, no. 5, pp. 188–201, 2001.
[15] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming aggressive replication in the Pangaea wide-area file system," SIGOPS Oper. Syst. Rev., vol. 36, pp. 15–30, 2002.
[16] S. Annapureddy, M. J. Freedman, and D. Mazi`eres, "Shark: scaling file servers via cooperative caching," in NSDI'05: Symp. on Net. Syst. Des. & Impl, 2005, pp. 129–142.
[17] M. O. Rabin, "Fingerprinting by random polynomials," Technical Report TR-15-81, Center for Research in Comp. Tech., Harvard University, Tech. Rep., 1981.
[18] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," SIGOPS Oper. Syst. Rev., vol. 29, no. 5, pp. 109–126, 1995.
[19] B. S. White, M. Walker, M. Humphrey, and A. S. Grimshaw, "LegionFS: a secure and scalable file system supporting cross-domain highperformance applications," in Supercomputing '01, 2001, pp. 59–59.

[20] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey, “Wide-area computing: Resource sharing on a large scale,” IEEE Computer, vol. 32, no. 5, pp. 29–37, 1999.
[21] C. Baru, R. Moore, A. Rajasekar, and M. Wan, “The SDSC storage resource broker,” in CASCON ’98:, 1998.
[22] P. Andrews, P. Kovatch, and C. Jordan, “Massive high-performance global file systems for grid computing,” in Supercomputing ’05, 2005, p. 53.
[23] S. C. Simms, G. G. Pike, S. Teige, B. Hammond, Y. Ma, L. L. Simms, C. Westneat, and D. A. Balog, “Empowering distributed workflow with the data capacitor: maximizing Lustre performance across the wide area network,” in SOCP ’07: Serv. oriented comp. perf., 2007, pp. 53–58.
[24] A. S. Sazlay, “The national virtual observatory,” in ASP Conf. Ser., ADASS X, vol. 238, 2001, p. 3.
[25] CERN, “Large hadron collider,” http://lhc.web.cern.ch/lhc/.
[26] R. Wolski, N. Spring, and J. Hayes, “The network weather service: A distributed resource performance forecasting service for metacomputing,” J. of Fut. Gen. Comput. Sys., vol. 15, no. 5, pp. 757–768, 1999.
[27] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, “Ivy: a read/write peer-to-peer file system,” SIGOPS Oper. Syst. Rev., vol. 36, pp. 31–44, 2002.
[28] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski et al., “Oceanstore: an architecture for global-scale persistent storage,” in Proc. of Arch. Support for Prog. Lang. and Op. Sys. ASPLOS), 2000, pp. 190–201.
[29] Cluster File Systems, Inc., “Lustre: a scalable high-performance file system,” Cluster File Systems, Tech. Rep., Nov. 2002, http://www.lustre.org/docs/whitepaper.pdf.
[30] J. Griffioen and R. Appleton, “Reducing file system latency using a predictive approach,” in USTC’94: USENIX Summer Tech. Conf., vol. 1, 1994, pp. 13–13.
[31] J. Ziv and A. Lempel, “Compression of individual sequences via variable-rate coding,” IEEE Trans. Inf. Theory, Sep. 1978.
[32] L. Torvalds et al., “Linux kernel,” http://www.kernel.org/, 2008.
[33] D. R. Hipp et al., “SQLite,” http://www.sqlite.org/, 2008.