Towards Autonomic Grid Data Management with Virtualized Distributed File Systems

Ming Zhao Jing Xu Renato J. Figueiredo
Advanced Computing and Information Systems Laboratory (ACIS)
Electrical and Computer Engineering
University of Florida, Gainesville, Florida
{ming, jxu, renato}@acis.ufl.edu

Abstract

This paper proposes an autonomic Grid data management architecture based on virtualized distributed file systems and WSRF-compliant management services. Autonomic functions are integrated into the services to provide self-managing control over the different entities of Grid-wide file system session, in accordance with high-level objectives, and operate together to automatically achieve the desired data provisioning behaviors. Important autonomic features are implemented in this system on cache configuration, data replication and session redirection. Experiments with the prototype demonstrate that this architecture can automatically and substantially improve the performance and reliability of Grid data access.

1. Introduction

Computational "Grids" aggregate computing and storage resources among multiple institutions to foster collaborations through shared access to large volumes of data and high-performance machines. Grid data management is a challenging task because of the heterogeneous, dynamic and large-scale nature of Grid environments. In this paper we present a framework that enables the implementation of autonomic techniques to deal with these challenges, while supporting well-known distributed file system interfaces for data access. Furthermore, the proposed approach requires no modifications to operating systems typically employed in Grid resources through the use of user-level virtualization techniques.

The Grid data management framework described in this paper is architected to address two important questions. First, *how to provide data with application-tailored optimizations?* Typically, operating systems are designed to support general-purpose applications, but it is often the case that "one size does not fit all." To achieve the best performance and reliability for a Grid application, data provisioning needs to be customized according to the application's requirements and characteristics.

Because an optimization tailored for one application (e.g. aggressive pre-fetching of file contents) may result in performance degradation for several others (e.g. sparse files, databases), application tailored features are typically not implemented in general-purpose O/S kernels. In addition, kernel-level modifications are difficult to port and deploy, notably in shared environments. Toolkits based solutions typically give users powerful APIs to program remote data access with desired behaviors, but few programmers are skilled to make effective use of such APIs. Instead, our approach provides applications and users with a familiar file system interface.

Second, *how to manage data provisioning in dynamically changing environments?* Customization often implies the consideration of various relevant factors and tuning of many parameters, in accordance with the desired behaviors and the surrounding environment. Dynamically changing availability of Grid resources further requires continuous monitoring of the data provisioning progress and timely reconfiguration.

In a large Grid system, these requirements are beyond the capability of end-users and even system administrators. Yet the goals of users or administrators are rather simple and explicit. From an application user's point of view, it is desired that the job execution is fast, responsive and reliable; from a resource provider's point of view, it is desired that the resource utilization is healthy and profitable.

This paper addresses these questions by proposing an autonomic Grid data management system based on virtualized Grid File Systems (GVFS), and autonomic data management services. GVFS employs user-level

virtualizations to provide user-transparent Grid data access with application-tailored enhancements. Autonomic services are designed as self-managing elements to control different entities of GVFS sessions and operate together to automatically achieve the desired data provisioning behaviors for the applications.

The contributions of this paper are as follows. First, we present a novel autonomic Grid data management framework upon which automatic monitoring, configuring, optimizing and healing of application-tailored Grid data provisioning can be achieved according to high-level objectives, and hence reduce management complexity and human intervention. Second, based on this framework, autonomic features of cache configuration, data replication and session redirection are designed and implemented in a prototype. Third, experimental evaluations of the prototype with I/O-intensive benchmark demonstrate that it can automatically and substantially improve the performance and reliability of Grid data access.

The rest of this paper is organized as follows. Section 2 introduces background of the proposed approach. Section 3 discusses the autonomic system architecture by explaining the autonomic data management services and their interactions. Section 4 presents experimental evaluations of the prototype and Section 5 concludes the paper.

2. Background

Grid Virtual File System (GVFS) is a virtualized distributed file system [10] for providing high-performance data access in Grid environments and seamless integration with unmodified applications. It leverages existing NFS (Network File System [6]) support in operating systems, and employs user-level proxies to authenticate and forward RPCs (Remote Procedure Calls) between the native NFS client and server and map user identities between different domains. GVFS helps users to access Grid data the same way as using typical LAN distributed file systems. It is also an important component of the In-VIGO virtualization middleware [1] for computational Grids.

Virtualized file system sessions based on GVFS are dynamically created to support remote data access for applications. A session is established between the client, where the application is running, and the server, where its data is stored, through the use of client-side and server-side proxies. User-level enhancements on performance, security and reliability are implemented between the proxies and can be customized for each session according to application requirements and characteristics [29].

GVFS sessions can be established, configured and terminated through the use of data management middleware implemented as WSRF-compliant [11] services [31]. These include the File System Service (FSS), which runs on every client and server and controls the local proxies to establish and configure specific GVFS sessions; the Data Scheduler Service (DSS), which provides central scheduling and customization of sessions through interactions with client-side and server-side FSSs; and the Data Replication Service (DRS), which manages application datasets and their replicas on servers to provide fault tolerance and load balancing for sessions.

This service-oriented data provisioning and management framework can serve end-users or a job scheduling middleware (e.g. the In-VIGO virtual application manager [1]) to prepare data sessions for application executions. The data services are key to supporting transparent resubmission of jobs for autonomic application management systems [27], which have been studied to automatically recover jobs from performance faults based on monitoring and predictions using application execution history and resource information stored in knowledgebase.

The use of a service-oriented framework to control the lifetime of sessions and virtualization to intercept, modify and forward file system calls provide a foundation for the implementation of several application-tailored enhancements. However, the management of GVFS sessions in a large-scale Grid system is still very challenging because changes to the workload and the utilization of shared processor, network and storage resources are very dynamic. It is desired that the sessions can self-manage to achieve user or job scheduler expected performance and reliability automatically. In the next section, the data management services are evolved into autonomic elements to work towards this goal (Figure 1). In this process, the above mentioned per-application and per-resource knowledgebase is leveraged to further improve the self-managing decisions in this autonomic system.

3. Autonomic Data Management Services

3.1 Autonomic Data Scheduler Service

As described earlier, the DSS schedules GVFS sessions for application executions. It interacts with DRS to request data replication and interacts with client-side and server-side FSSs to create, configure and terminate sessions. It is responsible for

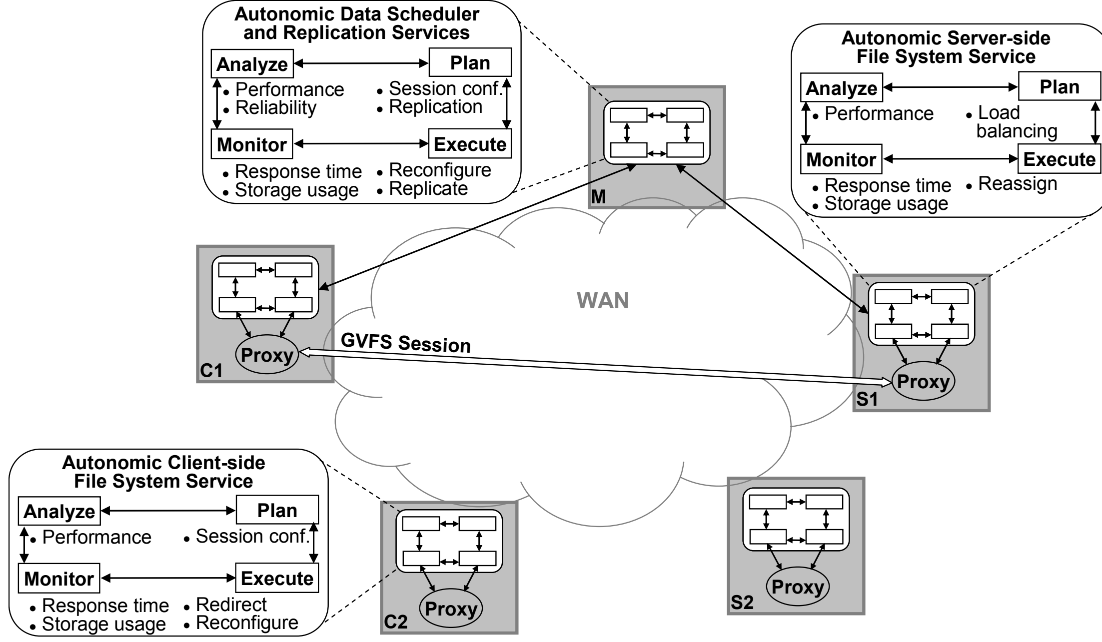

Figure 1: Autonomic data management system consists of autonomic data scheduler service, autonomic replication service, and autonomic client- and server-side file system services. They function as self-managing autonomic elements, which control the client, server and session of a GVFS data session according to the high-level objectives, and interact with each other to automatically achieve the desired data provisioning behaviors.

customizing and isolating different sessions with different configurations. One of the most important session parameters that can be customized is the size of the client-side disk cache maintained by GVFS.

Kernel NFS clients typically buffer data and metadata in memory, but the use of disk caching is rare. In wide-area, long-latency applications, the aggressive use of disk caching can be beneficial to several applications. Therefore GVFS implements a client-side proxy disk cache, which can leverage the large capacity of disks to further exploit data locality. However, as the dataset size of modern scientific and commercial applications grows rapidly, DSS needs to carefully manage the storage use for caching, which can have an important impact on performance of the sessions.

An application's remote I/O time can be estimated by,

$$T = N \times r_{mem} \times t_{mem} + N \times r_{disk} \times t_{disk} + N \times (1 - r_{mem} - r_{disk}) \times t_{network}$$

where N is the total number of remote data requests issued by the application; r_{mem} is the memory buffer cache hit rate, and r_{disk} is the proxy disk cache hit rate; t_{mem}, t_{disk}, and $t_{network}$ are the average service time of a request from memory, local disks and network storage, respectively.

For data intensive Grid applications, typically $r_{disk} >> r_{mem}$ and $t_{network} >> t_{disk} >> t_{mem}$, so the hit rate of proxy disk cache is crucial to delivering good application performance. A larger cache results in better hit rate, because capacity misses and conflict misses generally decrease as cache size grows [13]. However, the relationship between cache size and hit rate is a complex one, depending on the locality of data references and the associativity of the cache. Therefore, the DSS by default takes a conservative approach of configuring the proxy disk cache with a size larger than the application's dataset size.

There are also important scenarios where DSS needs to configure sessions with smaller disk caches. For example, when a node is more powerful or closer to the data server than the other nodes, it is chosen to execute the application because it can provide better performance even though its storage cannot hold the entire dataset. In another common case, multiple applications need to execute on the same node and the available disk space is not enough for their datasets. DSS can schedule their sessions to run sequentially with full-size disk caches. However, it may be necessary or more beneficial to configure each session with smaller disk caches and run them concurrently, in order to meet deadline requirements or achieve shorter total runtime.

If the application's data access pattern is known from the knowledgebase, DSS can use existing methods [32] to estimate the session's miss rate given the configured cache size, and then estimate its remote I/O performance using the above equation ($t_{network}$ is monitored online; N is known from history and offset by the already transferred requests which is also monitored online [1]). This information can further facilitate DSS to allocate the available storage capacity among multiple sessions which are scheduled to the same node.

A session i's utility U_i represents the value of providing a given level of service to the application. It can be calculated by considering the deliverable session performance and the application priority,

$$U_i = Performance_i \times Priority_i ,$$

where shorter runtime and higher priority generate greater utility value. Since a session's remote I/O time is affected by its disk cache size, given the available storage space as the constraint, the optimal allocation is achieved when the total utility from the different sessions on the node is maximized. The complexity of the optimization algorithm is bounded by the limited number of concurrent sessions and possible cache sizes.

To perform the above analysis, DSS must monitor the storage usage and the data server response time on the client. This is realized by interacting with the client-side FSS, which operates a monitoring daemon. Due to the dynamic and non-dedicated nature of Grid resources, environment changes may trigger DSS to reconfigure the session parameters. For example, when the disk usage is reaching the limit because of other local activities, DSS will detect it and reduce the total space occupied by the caches to avoid overflowing the storage. On the other hand, when more space becomes available for Grid use, DSS can expand caches as necessary.

Note that the changing resource availability may falsify the prediction which has motivated the end-user or the application manager to submit the job on this resource. Or even worse, the client node may crash and fail the job execution. An autonomic application manager should subscribe to these changes and accordingly resubmit or resume the job to another resource with a new data session prepared by DSS. This is beyond the scope of this paper. However, the server-side fault-tolerance is provided by the autonomic DRS.

[1] Besides of data requests kernel NFS issues a considerate number of metadata requests for consistency checks. These are typically satisfied by proxy disk cache which overlays GVFS consistency models upon the kernel NFS's [30].

3.2 Autonomic Data Replication Service

Data replication has long been recognized as key to achieving high availability. In Grid environments replication not only needs to be performed across servers in order to provide fail-over on server failures, but also should be distributed to different sites to protect against network partitions. However, limited bandwidth and high delay make wide-area replication very expensive. Although DRS uses high-throughput transfer mechanisms (e.g. GridFTP [2]) to duplicate data, the overhead is still considerable for large datasets. And because an application's session cannot start until the necessary replicas are ready, this overhead needs to be considered into the cost associated with the session.

The choice of the replication degree, i.e. the number of replicas, for a given dataset, is a decision that needs to be made based on benefit-cost analysis. Typically at least two replicas are required for each dataset, so that an application can continue its execution in presence of infrequent failures. As the failure rate goes higher, more replicas are required to provide good reliability, but the cost from replica creation (and teardown) is also increased. Although it is generally difficult to predict a particular data server's failure rate or MTTF (Mean-Time-To-Failure), it can be estimated based on observation and analysis. Initially every server has a hypothetical failure rate stored in the knowledgebase, and it is adjusted and updated by DRS as failures happen over time. Gradually the value becomes more representative of the server's actual reliability.

The available storage capacity for replica placement is shared among the existing sessions. The replicas prepared for a past application execution may also occupy disk space because a lazy style of cleanup is used, where a replica is not removed immediately after its session finishes, in anticipation of future use of the same dataset. Therefore, the storage management takes into account the values of different datasets, in which higher priority applications' datasets have higher values, and live sessions' datasets always value more than those that are not currently in use.

Based on the above considerations, a utility function is also used to solve the replication degree and placement problem. The utility U_d^i of having the ith replica for dataset d is computed by the product of the dataset's value V_d, and the reliability R_d^i provided by the its replicas, $R_d^i = 1 - \prod_{j=1}^{i} r_d^j$, where r_d^j is the failure probability of dataset d's jth replica (it is decided by the failure rate of the data server where this

replica is stored). The cost of creating these replicas is $C_d^i = \sum_{j=1}^{i} c_d^j$, where c_d^j is the overhead from copying the dataset to the jth replica's data sever from the nearest existing replica.

When considering adding a replica for a dataset, its utility and cost and reliability constraints are used to decide whether to add it and where to place it, as follows:

$$U_d = V_d \times R_d^i,$$
$$R_d^i \geq R_{\min},$$
$$C_d^i \leq C_{\max}$$

where $R_{\min}$ is the desired minimum level of reliability for the dataset and $C_{\max}$ is the maximum tolerable replica creation overhead. This algorithm tends to place a replica to the more reliable servers when reliability is more important, and put it to the closer servers if cost is more concerned. When multiple allocations are available, DRS chooses the server that has the best fit available space for the dataset's size. If replacement is necessary, it is decided based on the replicas' utilities.

When a data server failure occurs, the running sessions' replicas on that server need to be promptly regenerated in order to minimize the time windows in which the necessary replication degrees are not satisfied for the datasets. In this process, human-intervention should be avoided because it tends to be slow and costly, which means autonomic replica regeneration also needs to be supported by DRS. Last but not least, the impact of failures on applications should also be reduced to the minimum. It is desirable that applications can continue their executions without interruptions even in face of failures. This is realized by the autonomic FSS and will be discussed shortly.

DRS achieves autonomic replica regeneration by means of automatic failure detection and replica reconfiguration. A failure is discovered by DRS through notification from the DSS which detects a session failure, or by the DRS's periodic contacts with the server-side FSS for the purpose of storage usage monitoring. Note that the client-side FSS may not realize a failure because of the use of caching. And a failure reported by it can be caused by network partitioning between the client and server. This case is confirmed by the DRS if it can still connect to that server, and then only the datasets that are used by this particular client need to be regenerated.

Once a failure is determined, DRS immediately reconfigures the storage allocation using the above algorithm and regenerates the lost replicas for the running sessions on the newly selected data servers. The information about these new replicas is also informed to DSS, so that it can reconfigure the concerned sessions to use them in need of fail-over.

3.3 Autonomic File System Service

The client-side FSS facilitates the task of the autonomic DSS by monitoring storage usage and server-response time, and executing the session configurations decided by DSS. As discussed in Section 3.1, FSS controls the proxy to shrink or expand a session's disk cache as instructed by DSS. A proxy disk cache is structured as file banks which contain data blocks hashed according to their file handles and offsets. Sophisticated algorithms can be conceived to reduce a cache's size by evicting least recently used blocks and rehashing the other blocks. However, this often incurs substantial overhead perceived by the application. Instead, the proxy removes the least-used and most-clean file banks from the cache till the required shrinkage is achieved. This needs only a simple remapping of file banks but not any rehashing of data blocks, and the new cache size can immediately take effect.

Client-side FSS is also the key to realizing non-interrupt fail-over in presence of server-side failures, including data server node failure, network partitioning between the client and the server, and server or network overloading. These are detected when a major timeout (e.g. 100 times of the average response time) occurs to a data request. In order to recover from the fault, the proxy immediately redirects the session to the backup data server. Transparent session redirection is accomplished by the proxy via mapping the file handles among the replica servers. The FSS also reports the detected fault to DSS so that it can ask DRS to take actions on replica regeneration.

To achieve non-interrupt fail-over, a session uses an active-style replication among the replicas. Every data modification request issued by the application is multicast to the session's sever-side proxies and performed on all the replicas, and it does not return until the client-side proxy has received successful response from everyone. Consequently each replica has the exact same copy of the dataset during the entire session, and if any of them crashes, it generally has no impact upon the performance of the application since the remaining replicas can continue to service as usual.

Although it is necessary to write-all, a session can choose to use read-all or read-one. In the former case read operations are also performed on all the replicas. This model is employed to further improve reliability against not only server crashes, but also Byzantine

failures, in which the client-side proxy collects and compares all the replies and then decides on the correct one. However, the disadvantage of this model is that it bounds the performance of the session to the slowest replica server all the time. On the other hand, it is often safe to assume that a successful data access operation is correct because there are other mechanisms from hardware to software that are in place and can promise that an error would not happen without being noticed. Read operations are the most common ones for typical applications, and thus the most valuable to optimize to achieve speedup according to Amdahl's Law. Therefore, GVFS sessions typically use the read-one/write-all replication model.

This model relies on client-side FSS' autonomic functions to choose the best replica server to perform read operations throughout a session. The chosen server is called the primary server for the session. It is decided from the session's replica servers based on the performance that can be delivered for the application's remote I/O operations. The primary server can change over time as network conditions and server loads vary. So the FSS monitors the performance of the replica servers periodically using a monitoring daemon. However, it is important to design an accurate and low overhead mechanism to measure the performance.

From a session client's point of view, the response time of a remote data request is determined by the network delay, the server CPU's processing delay and the server disk's data access delay. Simple network probing mechanisms, e.g. ping, can give information about the network's performance, but not the server's. Using null RPC requests to the server incurs little overhead and can measure both the first and the second delays, but it cannot reveal the server's I/O load and disk performance. Instead, the monitoring daemon issues very small writes on the GVFS partition and uses the response times to estimate the performance that can be delivered by the server for the session.

Writes are used in the measurement because it is difficult to prevent the effect of server-side caching (processor caching and memory buffering) with read operations. The monitoring daemon periodically performs a one-byte write at a different block of a hidden remote file, and requests the server to commit the write so as to avoid the effect of server-side write-delay. Even though the hidden file's size may reach a large value for a long session, it does not actually occupy much space on disk because the server's file system uses holes on this file. And to further save the overhead, the monitoring daemon automatically wraps around to write from the first block again after a few hundreds of probes have been made.

Because the monitoring is done outside of the proxy, who is responsible of processing application's data requests, the session's performance is intact. FSS can use well-known time series analysis algorithms to predict the replica servers' future performance based on the observed response times, and then make decisions on primary server selection. Complex algorithms are not suitable because they would intensively compete for CPU with the running applications and not necessarily give the best predictions. On the other hand, simpler algorithms have been proved efficient and effective in many cases [23]. In the prototype exponential smoothing is used. Once the primary server is planed to change, the FSS controls the proxy to switch transparently.

The autonomic server-side FSS monitors the server's storage usage, which helps the autonomic DRS to decide replica placement. It also monitors the response times of RPC requests forwarded by the server-side proxy to the data server. The server is not necessarily the proxy's local host, but can be a virtualized server which consolidates network attached devices [3]. There are well studied algorithms [20] that provide load balancing in these scenarios which can be leveraged by the FSS and hence are not the focus of this paper.

4. Experiments

4.1 Setup

A prototype for the proposed architecture has been implemented and its autonomic features are evaluated in this section. VMware-based virtual machines are used to setup the file system clients and servers, which are hosted on two physical servers. Each physical server has dual 2.4GHz hyper-threaded Xeon processors and 1.5GB memory. Each VM is configured with 64MB memory and runs SUSE LINUX 9.2. The emphasis of the experiments is in wide area environments, which are emulated using NIST Net. Unless otherwise noted, every link is configured with a typical wide area RTT of 40ms.

The experiments are conducted by using a typical file system benchmark, Iozone, to represent the I/O part of typical Grid applications. The benchmark is executed on the client nodes with input accessed from the data servers via GVFS. The GVFS sessions are virtualized upon NFSv3, using 32KB data block transfer, and the data servers export the file system without write delay and with synchronous access. Every experiment is started with cold kernel buffer and GVFS disk caches by unmouting the file system and flushing the disk cache.

4.2 Autonomic Session Redirection

In a Grid environment network latency and throughput are often affected by the existence of parallel TCP transfers [17], for example, the popular use of GridFTP [2]. Figure 2 shows the latency between two nodes under such influence in a real wide-area setup. Each node is located in a different 100Mbps LAN and there are parallel TCP transfers between another pair of nodes from these two LANs. The data demonstrate that the latency grows rapidly as the number of parallel TCP streams used by this third-party transfer increases.

Such a scenario is emulated in the experiment and used to study the effectiveness of autonomic session redirection in the presence of network performance fluctuations. The client node is connected to two data servers (Server 1 and Sever 2) via two independently emulated WAN links, where each link's latency varies randomly with the values in Figure 2 in existence of 0, 2, 4 and 8 parallel TCP streams, and with decreasing probabilities for these values.

The Iozone benchmark is executed on the client node, which reads and rereads a 256MB file accessed through GVFS with different data server configurations: using *Server 1* statically; using *Server 2* statically; and using *Autonomic session redirection* between these servers dynamically. The average server response times are collected every 10 seconds throughout the execution of the benchmark, as shown in Figure 3(a).

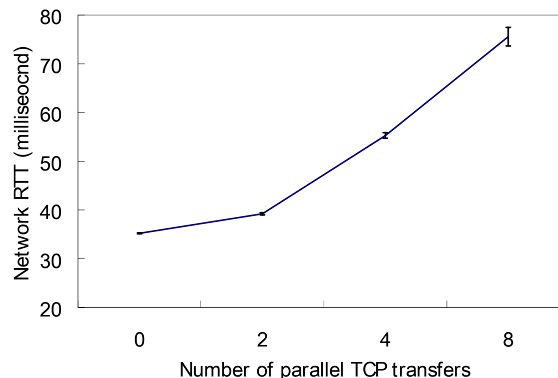

Figure 2. Network RTT between WAN-connected nodes influenced by third-party parallel TCP transfer. Each node is located in a different 100Mbps LAN; another pair of nodes from these LANs use IPerf to transfer data with multiple TCP streams.

The results show that with autonomic redirection the GVFS session can almost always choose the better link to support data access, and consequently the runtime of the benchmark is about 13% better than using Server 1 and 16% better than using Server 2, in the given network environment.

The second setup considers the influence of server load variations to the performance of a session. I/O

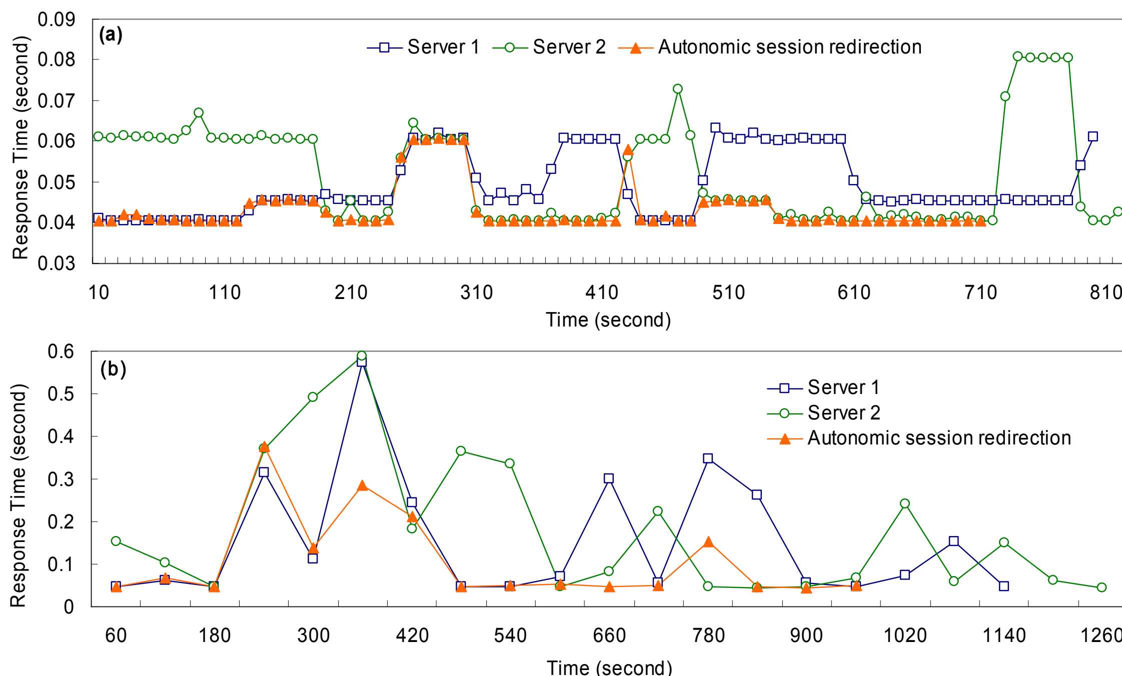

Figure 3. The average response times during the execution of Iozone (read/reread mode) on the client node with a 256MB input accessed through GVFS with different data server configurations: using Server 1 statically; using Server 2 statically; and using Autonomic session redirection between these servers dynamically

intensive jobs (executions of Iozone with read/re-read of different 256MB input files) are loaded to the server following a Poisson process, and the intensity is varied by randomly choosing the number of parallel jobs between 0 and 6. Then the benchmark is executed with the same configurations as above. The average server response times are collected every 60 seconds throughout the execution and shown in Figure 3(b).

The results also demonstrate that autonomic redirection can achieve the best server response time, and helps the benchmark to run 16% faster than using Server 2, and 29% faster than using Server 1.

4.3 Autonomic Cache Configuration

The second experiment studies the use of autonomic cache configuration by the DSS while scheduling different data sessions. In the setup two tasks are about to run on the same client node, where each task executes Iozone with random reading of a different 256MB file accessed from the data server via GVFS. The DSS needs to prepare two data sessions for these tasks but the available storage on the client can only hold 256MB of disk cache. So it has three different options for the configurations of these sessions: *A*, starts the first session with full disk cache and the second one without caching, concurrently; *B*, starts the two sessions sequentially with full disk cache for each session; *Auto*, starts the sessions concurrently and autonomically splits up the available storage between the sessions' caches based on their utilities (in this setup each session gets half of the full cache because they are equally important).

Figure 4 shows the runtimes of the jobs as well as their total runtime and total number of requests received by the server during their executions. Compared to *A*, the autonomic configuration provides better fairness between the jobs and also greatly reduces the server load (the server received requests is reduced by 18%); compared to *B*, the autonomic configuration's total runtime is much shorter (reduced by 40%).

4.4 Autonomic Data Replication

This experiment investigates the autonomic data replication in presence of server-side node or network failures. A series of tasks are launched on the client node sequentially, where each one runs Iozone in random reading mode with 512MB input accessed from the data servers through GVFS. The data servers fail randomly, where the failures are modeled as a Poisson process with an average interarrival time of half an hour. The experiment uses a replication degree of 2 for the datasets, and failures are injected on the servers by randomly choosing one of them to stop its network connection. Two different situations are considered for the tasks: *independent*, each task has an independent dataset (the input file) and hence is scheduled with a different data session; *dependent*, the tasks have the same dataset and share the same data session.

Figure 5 shows the timelines of the events happened during the experiment. In the independent tasks case, totally four server-side failures have happened. Each failure has caused a new replica to be generated by copying the data from the remaining server to a new server. Two of the failures have occurred at the primary data servers and also triggered the client to redirect the connection to the backup server. These all

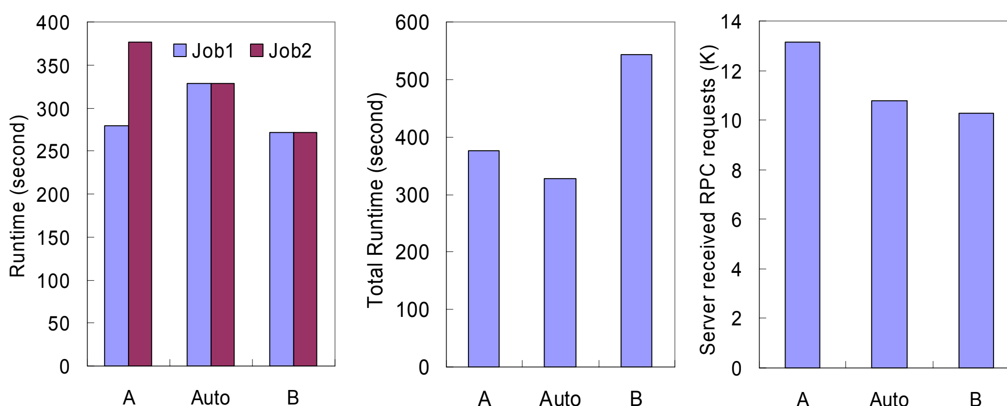

Figure 4: The runtimes of two executions of Iozone (Job 1, Job 2) with randomly reading a 256MB input accessed through GVFS, and their total runtime and total number of requests received by the server during their executions. Three different cache configurations are used for the sessions: *A*, starts the first session with full disk cache and the second one without caching, concurrently; *B*, starts the two sessions sequentially with full disk cache for each session; *Auto*, starts the sessions concurrently and autonomically splits up the available storage between the sessions' caches.

cause delays in the benchmark's executions, e.g. the second run takes the longest time to finish because two failures have occurred during that run. Nonetheless, every run has successfully completed regardless of the failures.

In the dependent tasks case, the warm disk cache not only substantially reduces the runtime of the benchmark, but also makes the client completely unaware of server-side failures and delays.

5. Related Work

Autonomic computing addresses the complexity of managing large-scale, heterogeneous computing systems by endowing systems and their components with the capability of self-managing according to high-level objectives [15]. The building blocks of an autonomic system are autonomic elements, which manage their own resources/services guided by policies, and interact with other autonomic elements to realize the desired system-level behaviors [24]. This paper follows this approach to construct autonomic Grid data management system by building the services as self-managing and interacting autonomic elements.

Related Grid data management solutions such as GridFTP [2] and GASS [5] provide APIs upon which applications can be programmed to access data on the Grid; Legion [25] employs a modified NFS server to provide access to a remote file system; The Condor system [16] uses system call interception to support remote I/O by re-linking applications. This paper differentiates from these efforts in that GVFS-based data sessions allow unmodified application binaries to access Grid data using existing operating system clients/servers, and support application-tailored per-session customizations.

BAD-FS [4] also leverages middleware control to enable application-specific file system optimizations, however, it is batch-job oriented and does not address the reliability issues of the data servers; [12] is another framework which improves performance and fault-tolerance of bulk data transfers. In contrast, the approach of this paper can support the important interactive type of applications, and provide non-interrupt recovery in face of server-side failures.

Data replication has long been recognized as key to achieving high availability [18]. Wide-area replications have also been studied by recent research on Web Content Distribution Network (CDN) [19] and wide-area file system [21]. [7] develops wide-area data replication based on Globus RFT service and GridFTP. In [14] a utility-based algorithm is used to decide the replication degree for resource managers. IBM autonomic storage manager implements policy-based

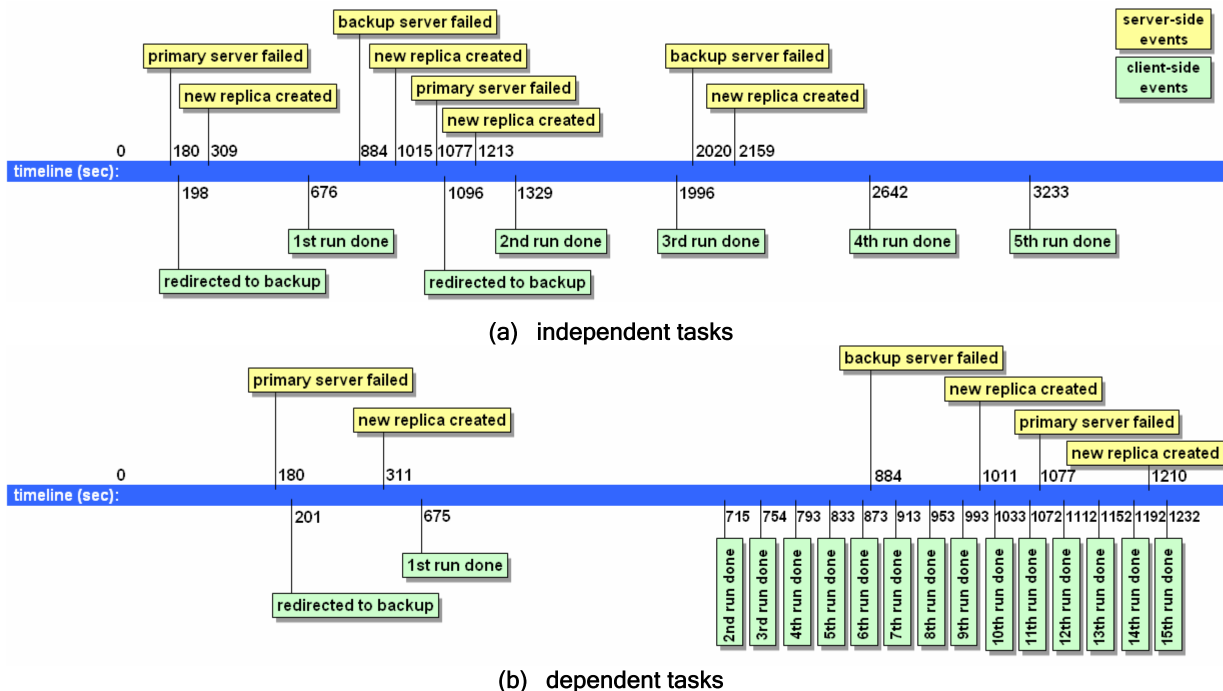

Figure 5. The timelines of the events happened during a series of executions of Iozone. Each run randomly reads a 512MB input accessed from the data servers through GVFS. The data servers fail randomly, and replica regenerations are triggered accordingly. In (a), the executions are independent and are supported by different GVFS sessions; in (b), the executions use the same dataset and share the same session.

storage allocation [9]. Automatic replica generation and distribution are developed in CDN [22] and peer-to-peer storage system [28]. Compared to these systems, this paper considers autonomic storage and replica management in the context of supporting dynamic Grid-wide file systems which provides user-transparent and application-tailored Grid data access.

6. Conclusion and Future Work

This paper addresses the challenge of Grid data management by constructing autonomic data management system based on application-tailored GVFS sessions and autonomic management services. Important autonomic features are implemented in the system and demonstrated by the prototype through experiments.

Many intriguing research subjects can be studied based on this framework. Among others, our future work is focused on: optimize proxy disk cache by making intelligent decisions on prefetching, block replacement and write-back etc., based on learning of application's data access pattern; interact with autonomic application manager to achieve utility-driven automatic checkpointing and resuming.

References

[1] S. Adabala et al, "From Virtualized Resources to Virtual Computing Grids: The In-VIGO System", *Future Generation Computing Systems*, Vol 21 No. 6, Apr 05.

[2] B. Allcock, et al, "Data Management and Transfer in High Performance Computational Grid Environments", *Parallel Computing Journal*, Vol. 28 (5), May 2002.

[3] D. Anderson, et al., "Interposed request routing for scalable network storage", ACM Transactions on Computer Systems, 20(1):25–48, February 2002.

[4] J. Bent, et al., "Explicit Control in a Batch-Aware Distributed File System", In *Proc. 1st NSDI.*, 2004.

[5] J. Bester, et al., "GASS: A Data Movement and Access Service for Wide Area Computing Systems", In *Proc. of the 6th IOPADS*, May 1999.

[6] B. Callaghan, *NFS Illustrated*, Addison-Wesley, 2002.

[7] A. Chervenak, et al., "Wide Area Data Replication for Scientific Collaborations", In *Proc. 6th IEEE/ACM International Workshop on Grid Computing*, Nov 2005.

[8] G. Coulouris, et al., *Distributed Systems: Concepts and Design*, 3rd edition, Addison-Wesley, 2001.

[9] M. Devarakonda1, et al., "Policy-Based Autonomic Storage Allocation", In *Proc. 14th DSOM*, Oct 2003.

[10] R. J. Figueiredo, et al., "Seamless Access to Decentralized Storage Services in Computational Grids via a Virtual File System", In *Cluster Computing*, 2004.

[11] I. Foster (ed) et al., "Modeling Stateful Resources using Web Services", *White paper*, March 5, 2004.

[12] T. Kosar, et al., "A Framework for Self-optimizing, Fault-tolerant, High Performance Bulk Data Transfers in a Heterogeneous Grid Environment", In *Proc. of 2nd Int. Symp. on Parallel and Distributed Computing*, Oct 2003.

[13] J. Hennessy and D. Patterson, *Computer Architecture: a Quantitative Approach*, 3rd edition, Morgan Kaufmann.

[14] V. Kalogeraki, "Decentralized Resource Management for Real-Time Object-Oriented Dependable Systems", *Technical Reports*, HPL-2001-93, 2001.

[15] J. O. Kephart, D. M. Chess, "The Vision of Autonomic Computing", *IEEE Computer*, 36(1): 41-50, 2003.

[16] M. Litzkow, M. Livny and M. W. Mutka, "Condor: a Hunter of Idle Workstations", In *Proc. the 8th Int. Conf. on Distributed Computing Systems*, June 1988.

[17] Dong Lu, et al., "Modeling and Taming Parallel TCP on the Wide Area Network", In *Proc. IPDPS*, April, 2005.

[18] K. Marzullo, F. Schmuck, "Supplying High Availability with a Standard Network File System", In *Proc. the 8th Int. Conf. on Distributed Computing Systems*, 1988.

[19] L. Qiu, V. N. Padmanabhan, G. M. Voelker, "On the Placement of Web Server Replicas", *INFOCOM*, 2001.

[20] L. W. Russell, S. P. Morgan, and E. G. Chron, "Clockwork: A New Movement in Autonomic Systems," *IBM Systems Journal* 42, No. 1, 2003.

[21] Y. Saito, et al., "Taming aggressive replication in the Pangaea wide-area file system", In *Proc. 5th Symp. on Op. Sys. Design and Impl.*, Dec 2002.

[22] S. Sivasubramanian, et al., "GlobeDB: Autonomic Data Replication for Web Applications", In *Proc. the 14th Int. Conference on World Wide Web*, Chiba, Japan, 2005.

[23] R. Wolski, "Dynamically forecasting network performance using the Network Weather Service", *Cluster Computing*, Volume 1 Issue 1, January 1998.

[24] S. R. White, et al., "An architectural approach to autonomic computing", In *Proc. 1st ICAC*, 2004.

[25] B. White, A. et al., "Grid-based File Access: the Legion I/O Model", In *Proc. the 9th HPDC*, Aug 2000.

[26] R. Wolski, N. Spring, C. Peterson, "Implementing a performance forecasting system for metacomputing: the Network Weather Service", In *Proc. of the 1997 ACM/IEEE Conference on Supercomputing*, Nov. 1997.

[27] J. Xu, S. Adabala, J. A.B. Fortes, "Towards Autonomic Virtual Applications in the In-VIGO System", In *Proc. of 2nd Int. Conf. on Autonomic Computing*, 2005.

[28] Haifeng Yu, Amin Vahdat , "Consistent and automatic replica regeneration", In *TOS*, Vol 1, Issue 1, Feb 2005.

[29] M. Zhao, et al., "Supporting Application-Tailored Grid File System Sessions with WSRF-Based Services", In *Proc. of 14th HPDC*, pages: 24- 33, July, 2005.

[30] M. Zhao and R. J. Figueiredo, "Application-Tailored Cache Consistency for Wide-Area File Systems", *Technical Report*, October 2005.

[31] M. Zhao, J. Zhang and R. J. Figueiredo, "Distributed File System Virtualization Techniques Supporting On-Demand Virtual Machine Environments for Grid Computing", *Cluster Computing*, Vol. 9, Jan 2006.

[32] Y. Zhong, S. G. Dropsho, C. Ding, "Miss rate prediction across all program inputs", In *Proc. 12th PACT*, Oct 03.