Peer-to-Peer Data Sharing for Scientific Workflows on Amazon EC2

Rohit Agarwal
Department of Computer Science and Engineering
Indian Institute of Technology, Ropar
ragarwal@iitrpr.ac.in

Gideon Juve, Ewa Deelman
Information Sciences Institute
University of Southern California
{gideon, deelman}@isi.edu

Abstract—**In this paper, we consider the problem of data sharing in scientific workflows running on the cloud. We present the design and evaluation of a peer-to-peer approach to help solve this problem. We compare the performance of our peer-to-peer file manager with that of two network file systems for storing data for a typical data-intensive workflow application. Our results show that while our peer-to-peer file manager performs significantly better than one of the network file systems tested, it does not perform as well as the other. Finally, we discuss the various issues that might have affected the performance of our peer-to-peer file manager.**

Keywords—Cloud computing; scientific workflows; data sharing; peer-to-peer (P2P); performance evaluation.

I. INTRODUCTION

Workflow applications are being used extensively by scientists to tackle complex simulations and data-intensive analyses [1,2,3,4]. Traditionally, large-scale workflow applications have been run on distributed cyberinfrastructure such as HPC clusters and grids. With the recent advent of cloud computing, many scientists are investigating the benefits of moving their workflow applications to the cloud. This is because clouds give workflow developers several advantages over traditional HPC systems, such as root access to the operating system, control over the entire software environment, reproducibility of results through the use of VM images to store computational environments, and on-demand provisioning capabilities.

One important question when evaluating the effectiveness of cloud platforms for workflows is: How can workflows share data in the cloud? Workflows are loosely-coupled parallel applications that consist of a set of computational tasks linked via data- and control-flow dependencies [5]. Unlike tightly-coupled parallel applications such as MPI codes in which processes communicate directly using network sockets, workflows typically communicate by writing data to files. Each task in a workflow produces one or more output files that become input files to other tasks. In order to function in a distributed environment, these files must be accessible by all compute nodes. This can be done by transferring the files between nodes, or it can be done by using a POSIX-compatible network file system. Most HPC clusters provide a network file system that is attached to all the compute nodes, but grids and clouds typically do not, either because of latency concerns due to wide-area networks between clusters in the case of grids, or because of the overhead and complexity of virtualization and the preference for non-POSIX storage services in clouds (such as Amazon S3 [6]). In order to function in grid and cloud environments, workflow systems usually operate by copying input and output files from a central storage location to the compute node and back to storage for each job. Needless to say, this adds substantial overhead because each file needs to be transferred multiple times (once when it is created, and once for each time it is used).

An alternative is to cache the files on the compute nodes, and transfer them directly between nodes rather than passing them through a centralized storage service. This avoids the extra transfer, increases the scalability of the system, and enables workflow management systems to consider locality when scheduling data-intensive workflows. In this paper we present the design and evaluation of a peer-to-peer file manager based on this idea. We compare its performance with that of two network file systems when storing data for a typical data-intensive workflow application. We test the scalability of different storage systems by running the experiments repeatedly on different number of nodes provisioned from the cloud.

Our results show that while our P2P file manager performs significantly better than NFS [7] (a centralized file system) on the benchmark application, it does not perform as well as GlusterFS [8] (a distributed file system).

The rest of this paper is organized as follows: Section II discusses related work. Section III describes the motivation for peer-to-peer data sharing and provides an overview of the architecture of our peer-to-peer file manager. Section IV gives an overview of the execution environment that was set up for the experiments on Amazon EC2 [9]. Section V provides the results of various benchmarks we ran to test our peer-to-peer system and the results of performance comparison between our system and various shared file systems. Section VI discusses the various issues that might have affected the performance of the P2P file manager. Section VII concludes the paper and outlines our future work.

II. RELATED WORK

Data-intensive workflows encounter various types of data management challenges. These challenges are examined in [10] and include data discovery, data provenance, data selection, data transfer, data storage etc. Our paper provides a peer-to-peer solution for the problems of data transfer and storage in data-intensive workflows.

This work is an extension of our previous work where we compared the cost and performance of several different storage systems that can be used to communicate data within a scientific workflow running in the cloud [11]. In this paper we have extended that work to include a peer-to-peer data sharing system and used a larger workflow to compare performance.

Yan, et al. have conducted research on peer-to-peer workflow systems [12]. They developed SwinDeW, a decentralized, peer-to-peer workflow management system. In comparison, our work is more focused on optimizing data movement. Our system is only concerned with the problem of data sharing in workflow applications and not on the general workflow orchestration problem.

The problem with centralized servers and data-intensive workflows was previously investigated by Barker et al. [13]. They proposed an architecture for web services based workflow orchestration that uses a centralized orchestration engine with distributed data flow. This solution is similar to our approach, however, our approach is designed to work with task-oriented workflows and uses distributed caches to improve performance instead of planning all data movement and execution operations ahead of time.

There are many data management techniques for improving the performance and cost of data-intensive workflows. Çatalyürek et al. [14] describe a heuristic that models the workflow as a hypergraph and considers both data placement and task assignment. Using ratios of the storage capacity and computing power of execution sites, it generates plans that distribute the storage and computational loads among the sites in a manner that reduces the cost associated with file transfers. Chervenak et al. [15] describe a policy service for data management that can improve the performance of data-intensive workflows by giving advice to the workflow system about how and when to transfer data based on its knowledge of ongoing transfers, recent transfer performance, and the current allocation of resources.

III. PEER-TO-PEER DATA SHARING

A. *Motivation*

Storage system performance is a critical component in the development of data-intensive applications such as scientific workflows. When files reside on a central server, it is efficient to find the location of the file, but the server can be overwhelmed by a large number of requests for data. Thus, distributing the files across multiple servers may relieve that bottleneck, but it may also impose additional overheads in file discovery and retrieval.

Some of the overheads associated with parallel or distributed file systems occur because of the consistency constraints imposed on concurrent file access. These constraints are not required for workflow files. This is because workflow data files are *write-once*—once they are generated, they are not written to again.

In this paper, we examine a hybrid (P2P) approach, which does not have such a central bottleneck—all data transfers are done directly between nodes, the only centralized part of the system is the index server which can handle loads much greater than those encountered in workflow applications. (See Section V). Relying on the *write-once* property of the target workflow applications, our P2P approach does not employ these constraints and is thus much simpler and potentially faster.

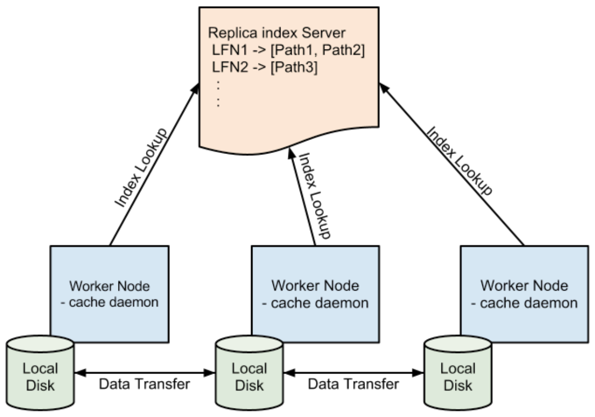

Figure 1. Architecture of Peer-to-peer file manager.

B. *Architecture*

Our peer-to-peer file manager [16] consists of 3 components: a *replica index service*, a *cache daemon*, and a *client*. The architecture is shown in Figure 1.

Each compute node stores in its local cache copies of all the files generated by jobs that run on that node, or are accessed by jobs that run on the node. In order to find input files for a job, the system uses a centralized replica index server that stores a list of mappings from each logical file name to a set of physical locations (URLs) from where a replica of the file can be retrieved. When a node needs a file that is not in its local cache, it looks it up in the replica index and retrieves it from one of the nodes where it is cached. If multiple nodes have copies of the requested file, then the node to retrieve the file from is chosen randomly. When a node generates a file, it saves it in the cache and registers a mapping for it in the replica index.

1) *Replica Index Service*

There is one replica index server (RIS) per cluster. The RIS stores mappings from globally unique logical file names (LFNs) to physical file names (PFNs) where the data can be retrieved.

The index server is an XML-RPC service written in Python and supports a few simple operations:

- `add(lfn, pfn)`: Adds an LFN to PFN mapping to the index.
- `lookup(lfn) => pfn[]`: Returns a list of PFNs for the given LFN.

- `delete(lfn, pfn)`: Removes an LFN to PFN mapping.

2) Cache daemon

The cache daemon runs on each compute node to manage a local cache of file replicas. It contacts the replica index server to locate files that are not in its cache, and to update the index server with mappings for files that are added to its cache. Cache daemons running on different nodes contact each other to directly retrieve files that are not in the local cache.

The cache daemon is an XML-RPC service written in Python that supports these operations:

- `get(lfn, path)`: Retrieve a file identified by the given LFN (from the local or a remote cache) and store the contents at a given path in the local file system.
- `put(path, lfn)`: Store the contents of the file at a given path in the cache and register a mapping with the index service from the given LFN to a PFN that identifies the location of the cached replica.
- `delete(lfn)`: Remove any cached replica of the file identified by the given LFN and remove any mappings held by the index server that map the given LFN to PFNs that point to the local cache.

3) Client

The client is a command-line interface to the index server and the cache daemon. The client performs `get`, `put` and `delete` commands on the local cache daemon to service requests made by the application.

IV. EXECUTION ENVIRONMENT

In this section we describe the setup that was used in our experiments. All the experiments were performed on Amazon's EC2 [9] infrastructure as a service (IaaS) cloud.

There are many ways to configure an execution environment for workflow applications in the cloud. The environment can be deployed entirely in the cloud, or parts of it can reside outside the cloud. For this paper we have chosen the former approach. In this configuration there is a *submit node* that orchestrates the workflows and manages the replica index server, and several *worker nodes* (compute nodes) that execute tasks, store data, and run the cache daemon. Both the submit node and the worker nodes run inside the cloud. The setup is shown in Figure 2.

A. Software

The configuration of the execution environment is based on the idea of a virtual cluster [17, 18]. A virtual cluster is a collection of virtual machines that have been configured to act like a traditional HPC system. Typically, this involves installing and configuring job management software, such as a batch scheduler, and a data sharing system, such as a network file system. The challenge in provisioning a virtual cluster in the cloud is collecting the information required to configure the

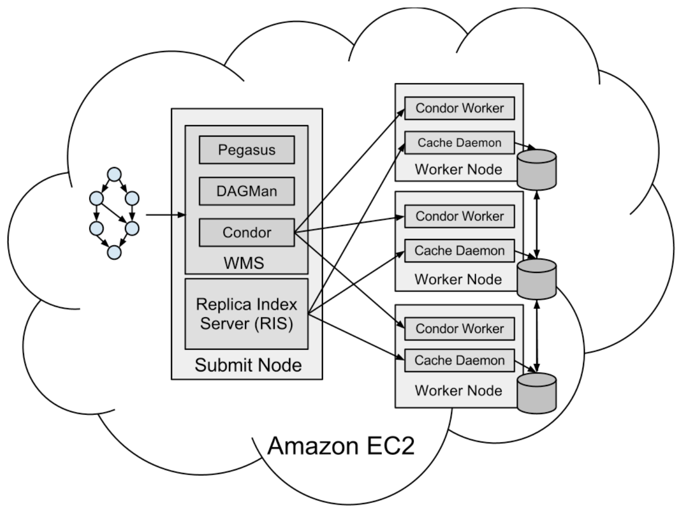

Figure 2. Execution Environment.

cluster software, and then generating configuration files and starting services. Instead of performing these tasks manually, which can be tedious and error-prone, we created VM images which have the required software already installed and wrote shell scripts that used the instance-metadata service provided by Amazon EC2 to provision and configure the virtual clusters for this paper.

All workflows were planned and executed using the Pegasus Workflow Management System [1], which includes the Pegasus mapper, DAGMan worfklow engine [19] and the Condor `schedd` task scheduler [20]. Pegasus is used to transform a resource-independent, abstract workflow description into a concrete plan, which is then executed using DAGMan. The latter manages dependencies between executable tasks, and Condor `schedd` manages individual task execution. Pegasus was modified to support the peer-to-peer file manager by adding hooks into the Pegasus file transfer module to support a custom p2p:// URL scheme. These p2p:// URLs are used as LFNs in the workflow to tell Pegasus when to use the cache daemon client to store and retrieve files.

A single CentOS 5 virtual machine image was used for both the submit host and the worker nodes, and Pegasus and Condor were pre-installed on the image. In addition, shell scripts were added to the image to generate configuration files and start the required services during the boot process.

B. Resources

Amazon EC2 offers several different resource configurations for virtual machine instances. Each instance type is configured with a specific amount of memory, CPUs, and local storage. Rather than experimenting with all the various instance types, for this paper only the m1.xlarge instance type is used. This type is equipped with 8 EC2 Compute Units (4 virtual cores with 2 EC2 Compute Units each), 15 GB RAM, and 1690 GB local disk storage. A different choice for worker nodes would result in different performance metrics. However, an exhaustive survey of all the possible combinations is beyond the scope of this paper.

V. Evaluation

A. Replica Index Server Throughput

The performance of the replica index server is important because it has the potential to be a significant bottleneck in the system. In order to determine if the index server is fast enough, we ran a benchmark to see how many lookup operations it can handle per second and compared it to the rate required by a typical workflow application.

To measure the throughput of the index server we deployed the server on a dedicated EC2 instance and started several clients in parallel on a separate EC2 instance. We measured the total time required to service 1000 requests from each client as we varied the number of parallel clients from 1 to 16. The results, shown in Figure 3, indicate that the index server was able to handle approximately 650 requests per second without any optimizations.

To determine the query rate required to support workflow applications we divided the number of entries in the replica index server (RIS) after completion of a Montage 10 degree square workflow (described in Section V.C) by the runtime of the workflow to determine the average number of requests per second that the RIS would be required to support to complete this workflow. The results are shown in Table I.

The results indicate that the average number of requests per second generated by the workflow application used in our experiments (~10-25 requests/second) is much less than what the un-optimized index server is capable of supporting (~650 requests/second). We expect that this will be the case for most workflow applications, however, if the index server becomes a bottleneck for larger workflows or larger clusters we can try storing the index entries in a Distributed Hash Table or a scalable cloud database service such as Amazon SimpleDB [21].

TABLE I. MEAN REQUESTS PER SECOND FOR THE REPLICA INDEX SERVER

Number of workers	*Number of entries in RIS*	*Workflow runtime*	*Average number of requests per second*
2	63558	6699	9.5
4	76688	4705	16.3
16	87073	3704	23.5

B. Cache Daemon Performance

To test the cache daemon performance we deployed three VMs on Amazon EC2: one for the replica index server, and two to run the cache daemon. The cache daemons were configured to store data on the local ephemeral disks. Using this configuration we performed several experiments to test the performance of the put and get operations of the cache daemon. Also, since the performance of the cache daemon depends on the I/O performance of the disks where the data is stored, and the network used to transfer data between the cache daemons, we also measured their performance to help explain our results.

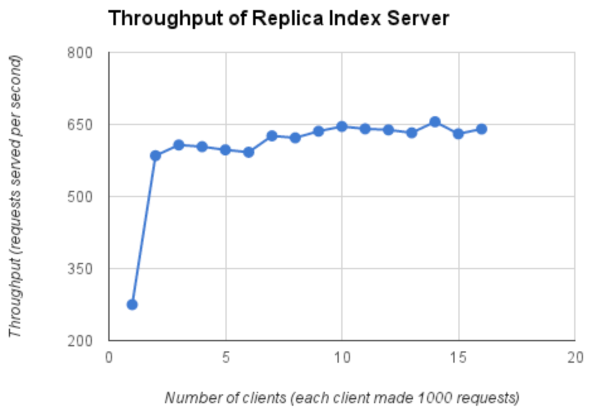

Figure 3. Throughput of Replica Index Server.

1) Disk and Network Performance

The performance of the ephemeral disks was measured by writing and reading 100MB files using the UNIX *dd* command. To ensure that the file system cache did not affect the measurements, the experiment called *sync* after writing the files and the disk cache was flushed between the write and read experiments. Each experiment was performed 100 times and the mean write performance was determined to be ~38 MB/s and the mean read performance to be ~109 MB/s. This asymmetry in write and read performance is likely due to the well-known "first-write penalty" on Amazon's ephemeral disks [11].

The network bandwidth between the two EC2 VMs was measured using the UNIX netcat (*nc*) utility. Netcat was used to send 100MB of data from one EC2 VM to another. The experiment was repeated 100 times and the average time was used to compute the bandwidth between the VMs, which was found to be ~89 MB/s. This value is slightly less than what we would expect on a gigabit network.

2) Put Operation Performance

Clients call the *put* operation to add files to the local cache and register them in the index server. The following actions take place when the *put* operation is invoked:

- The client sends a put request to the local cache daemon specifying the source path to a file to be cached and the LFN to use when indexing the file.
- The file is added to the local cache.
- The cache daemon sends a request to the index server to register the LFN.

There are two implementations of the code to add files to the cache. One implementation copies the file from the source path to the cache, leaving the source path untouched. The other implementation moves the file from the source path to the cache and replaces the source path with a symlink to the file in the cache.

To measure the performance of the put operation we called put for files varying in size from 0 MB to 100 MB. The experiment was repeated 100 times for each file size. The mean response time for the put operation using both the symlink implementation and the copy implementation is shown in Table II.

TABLE II. MEAN RESPONSE TIME OF PUT OPERATION IN SECONDS

Implementation	0 MB	1 MB	10 MB	100 MB
copy	0.007	0.009	0.350	4.360
symlink	0.008	0.007	0.008	0.008

As expected, the results show that the symlink implementation is significantly faster than the copy implementation and does not depend on the file size. In addition, this experiment indicates that each cache daemon can service approximately 125 put operations per second.

3) *Get Operation Performance*

Clients call the *get* operation to retrieve files from the local or remote cache and save them to a local destination path. The following actions take place when the *get* operation is invoked:

- The client queries the local cache daemon for the LFN.
- If the file is found in the cache, it is retrieved from the local cache and made available at the destination path.
- If file is not found in the local cache:
    - The local cache daemon queries the replica index server to find a remote cache daemon that has the file.
    - The local cache daemon requests the file from the remote cache daemon.
    - The remote cache daemon reads the file out of its cache and transfers it over the network to the local cache daemon.
    - The local cache daemon writes the file to the local cache and makes it available at the destination path.
- The local cache daemon registers the cached copy of the file with the index service

Like the *put* operation, there are two implementations of the get operation: one that copies the file from the cache to the destination path, and one that symlinks the file from the cache to the destination path.

In the case of a cache hit, the *get* operation is simply the inverse of the put operation and therefore has performance that is nearly identical to the put operation. In order to measure the worst-case performance of the get operation, the experiment was set up so that the file was never found in the local cache to force the local cache daemon to retrieve the file from the peer node.

The *get* operation was executed using files ranging in size from 0MB to 100 MB. Both the mean response time and the effective throughput (in MB/s) were computed. The effective throughput was computed by dividing the amount of data transferred by the response time. We expect that, without file system caching, the effective throughput will be limited by the bottleneck in the transfer path, which, base on our earlier disk and network benchmarks, is writes to the ephemeral disk (~21 MB/s). In order to isolate the effects of the file system cache we made sure that the cache on the remote node was cleared in all experiments. In addition, we created another implementation of the cache daemon that calls *fsync()* on all files written to the local cache before returning results to the client.

The results of this experiment for the copy implementation, the symlink implementation, and the implementation using symlinks and *fsync* are shown in Tables III and IV. All numbers are the mean of 100 experiments.

TABLE III. MEAN RESPONSE TIME IN SECONDS FOR THE GET OPERATION

Implementation	0 MB	1 MB	10 MB	100 MB
copy	0.016	0.031	0.178	3.951
symlink	0.017	0.033	0.146	1.841
symlink+fsync	0.017	0.073	0.373	3.182

TABLE IV. MEAN EFFECTIVE THROUGHPUT IN MB/S FOR THE GET OPERATION

Implementation	1 MB	10 MB	100 MB
copy	31.784	56.048	25.310
symlink	30.571	68.734	54.329
symlink+fsync	13.776	26.824	31.423

This data illustrates several important points. First, the response time for 0 MB files shows that the overhead of the get operation is ~17 ms (min response time was 15 ms for 0-byte files). This results in a maximum throughput of ~59 requests per second (max 67 req/s), which could be a limiting factor in system performance. This overhead is a result of the three network requests required to get a file: one to the cache daemon, one to the index server, and one to the remote cache daemon. Second, when the effects of the file system cache are accounted for, the performance for 1 MB files is significantly less than the average disk bandwidth. This is partially a result of the system overhead, which is a significant fraction of the total response time for small files. This is important because many workflows contain lots of small files.

C. *Workflow Performance Comparison*

We used an astronomy workflow application, Montage [23], to compare performance of the peer-to-peer data sharing approach with the shared file system approach. We picked Montage, because it was the most affected by the choice of storage systems in our prior experiments [11].

We compared our P2P system with two different shared file systems: NFS [7] and GlusterFS [8]. We measured the makespan of the workflow as we varied the number of worker nodes from 2 to 16. Makespan is defined as the total amount of wall clock time from the moment the first workflow task is submitted until the last task completes. The makespan times reported in Section V.C.3 do not include the time required to boot and configure the VM as this is assumed to be independent of the approach used for data sharing.

1) Benchmark Workflow

Montage [23] is a data-intensive workflow application that creates science-grade astronomical image mosaics using data collected from telescopes. The size of a Montage workflow depends upon the area of the sky (in square degrees) covered by the output mosaic. A small workflow that illustrates the structure of Montage is shown in Figure 4. Montage can be considered to be a data-intensive workflow because it spends more than 95% of its time waiting on I/O operations and only 5% on computation.

For the workflow performance experiment we created a Montage workflow that generates a 10-degree square mosaic. The workflow contains 19,320 tasks, reads 13 GB of input data, and produces 88 GB of output data (including temporary data).

2) Network File Systems

NFS [7] is perhaps the most commonly used network file system. NFS is a centralized system with one node that acts as the file server for a group of machines. For the workflow experiments we used the submit node in EC2 to host the NFS file system. We configured the NFS clients to use the `async` option, which allows the NFS server to reply to requests before any changes made by the request have been committed to disk.

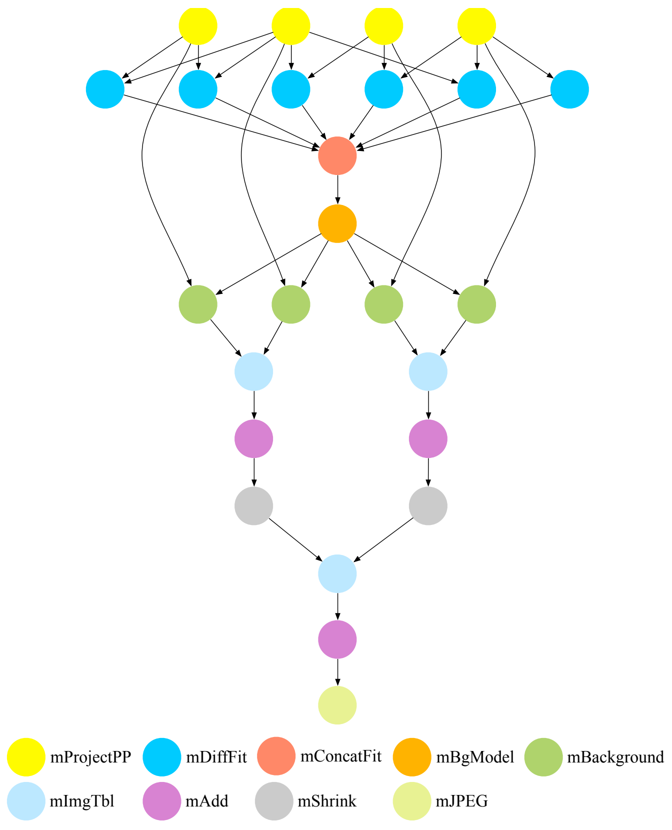

Figure 4. Task Dependencies in the Montage Workflow.

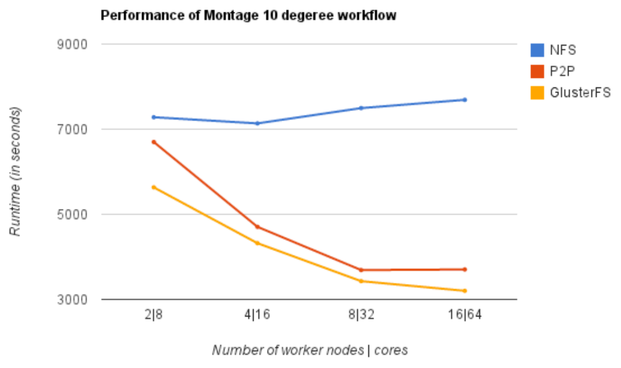

Figure 5. Performance of Montage using different storage systems.

GlusterFS [8] is a distributed file system that supports many different types of volumes (Distributed, Replicated, Striped and their combinations.) GlusterFS volumes are logical collections of one or more bricks where each brick is an export directory on a node. To create a new volume, one specifies the bricks that comprise the volumes and the way files would be distributed on those bricks. We used the distribute configuration in which files are spread randomly across bricks in the volume. We exported one directory from each of the worker nodes and one directory from the submit node to make up the volume.

3) Results

The performance results for Montage are shown in Figure 5. The results show that both GlusterFS and our P2P system significantly outperform NFS as the shared file system. This is a result of the fact that NFS does not scale well as the number of worker nodes increase because the NFS server becomes a bottleneck. In comparison, both the P2P system and GlusterFS improve in performance as the number of nodes increase. However, neither system has a linear performance increase because the structure of the Montage workflow contains several sequential bottlenecks that prevent perfect speedup. Finally, contrary to our expectations, GlusterFS performs even better than our peer-to-peer system. The reasons for this will be discussed in the following section.

VI. Discussion

We had hoped that, by avoiding the shared file system bottleneck and by making the simplifying assumption of *write-once* files, we could improve the performance of data-intensive workflows. However, using GlusterFS as the shared file system outperformed our peer-to-peer solution. There are a number of reasons for this.

Part of the problem is with our implementation. Our benchmarks show that the data transfer throughput of our peer-to-peer file manager is not very high (Section V.B), especially for small files. Because Montage is a very data-intensive workflow, storage system performance is the limiting factor in its performance. In addition, the average file in Montage is ~2 MB in size, which is small enough that the overhead of the peer-to-peer system has a large impact on the effective throughput that can be achieved. In comparison, the GlusterFS distribute configuration that was used has very low overhead to locate a file (essentially, it just needs to hash the filename). It may be possible to reduce overhead in our peer-to-peer system by optimizing the code or by rewriting it in a more efficient language.

Another problem may be the lack of locality in file accesses. Although the tasks are evenly distributed across the compute nodes according to First-Come, First-Served (FCFS) scheduling, the data may not be. The critical feature of our peer-to-peer file manager is that it enables aggressive caching and replication of files on worker nodes. However, if workflow tasks are not able to take advantage of that caching because of data imbalance and the lack of locality-awareness in task scheduling, then performance will not be as good as it could be. In the future we plan to investigate data-aware scheduling to reduce this problem.

Another fundamental difficulty with the peer-to-peer approach is the inability to do partial file transfers. Whenever a task needs to read a file that is not present in the local cache, the whole file is transferred from a remote cache even if the task only needs to read part of the file. In comparison, traditional file systems such as NFS and GlusterFS allow processes to seek and read parts of files without reading all of the data. In the Montage workflow there are several tasks that only require header information from the files. In the NFS and GlusterFS configurations these tasks are able to read only the portion of the file required while in the peer-to-peer configuration the entire file is retrieved from the remote cache. Further work is needed to determine what impact this has on the total amount of data transferred.

Another point to note is that GlusterFS is a fully distributed file system. It does not have a central data bottleneck as files are distributed uniformly across the nodes and transferred directly between nodes. GlusterFS also avoids the metadata bottleneck that many file systems have by distributing metadata storage. For all files and directories, instead of storing associated metadata in a set of static data structures, it generates the equivalent information on-the-fly algorithmic-ally. Because of this, there is never any contention for any single instance of metadata stored at only one location as each node is independent in its algorithmic handling of its own metadata. [24]

Finally, we compared the performance of these storage systems using only one data-intensive workflow as the benchmark application. It may be the case that our peer-to-peer system would perform better relative to GlusterFS on other applications. In the future we plan to run experiments with a variety of workflow applications.

VII. Conclusion

In this paper, we examined a peer-to-peer approach for data sharing for workflow applications running on the cloud. We described the design and implementation of a peer-to-peer file manager that uses caching to try and improve the performance of write-once workflow applications in distributed environments. We expected that the distributed nature of our solution and the aggressive use of caching would make the peer-to-peer approach more efficient than using a shared file system for workflow applications. Although our approach performed better than NFS, contrary to our expectations it did not perform better than GlusterFS on our benchmark workflow application. We discussed the various issues that may have caused this.

Our investigation suggests several possible routes for future work. We plan to investigate locality-aware workflow scheduling techniques that may improve our system's cache hit rate. We will also look at scheduling techniques that consider load balancing (in terms of both I/O and CPU). We also plan to investigate code optimizations that may reduce the overhead of our approach. Finally, we plan to evaluate our solution on a variety of workflow applications to get a broader understanding of the benefits and drawbacks of our solution.

VIII. Acknowledgements

This research was supported by the National Science Foundation under award OCI-0943725 (STCI). Rohit Agarwal was supported by the Viterbi-India summer research experience program, which is funded jointly by the Indo-US Science and Technology Forum (IUSSTF) and the USC Viterbi School of Engineering. The use of Amazon EC2 resources was supported by an AWS in Education research grant.

References

[1] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz, "Pegasus: A framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming*, vol. 13, no. 3, pp. 219–237, 2005.

[2] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[3] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[4] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang, "Programming scientific and distributed workflow with Triana services," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1021–1037, 2006.

[5] E. Deelman, D. Gannon, M. Shields, I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities", *Future Generation Computer Systems*, vol. 25, no. 5, pp. 528-540, 2009.

[6] "Amazon Simple Storage Service (S3)", http://aws.amazon.com/s3.

[7] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem", USENIX Conference, 1985.

[8] "GlusterFS", http://www.gluster.org.

[9] "Amazon Elastic Compute Cloud (EC2)", http://aws.amazon.com/ec2.

[10] E. Deelman and A. Chervenak , "Data Management Challenges of Data-Intensive Scientific Workflows", 3rd International Workshop on Workflow Systems in e-Science (WSES 08), 2008.

[11] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berman, B. Berriman, P. Maechling, "Data Sharing Options for Scientific Workflows on Amazon EC2", 22nd IEEE/ACM Conference on Supercomputing (SC10), 2010.

[12] J. Yan, Y. Yang, and G. K. Raikundalia, "SwinDeW—A p2p-Based Decentralized Workflow Management System", *IEEE Transactions on*

systems, man and cybernetics—Part A: Systems and Humans, vol. 36, no. 5, pp. 922-935, 2006.

[13] A. Barker, J. Weissman, and J. van Hemert, “Eliminating the middleman: peer-to-peer dataflow”, 17th IEEE International Symposium on High Performance Distributed Computing (HPDC), 2008.

[14] Ü. V. Çatalyürek, K. Kaya, and B. Uçar, "Integrated data placement and task assignment for scientific workflows in clouds", 4th international workshop on Data-intensive distributed computing (DIDC11), 2011.

[15] A. Chervenak, D. Smith, W. Chen, E. Deelman, “Integrating Policy with Scientific Workflow Management for Data-Intensive Applications”, 7th Workshop on Workflows in Support of Large-Scale Science (WORKS'12), 2012.

[16] “mule p2p storage system”, https://github.com/juve/mule.

[17] J. Chase, D. Irwin, L. Grit, J. Moore, and S. Sprenkle, "Dynamic virtual clusters in a grid site manager", 12th IEEE International Symposium on High Performance Distributed Computing (HPDC), 2003.

[18] I. Foster, T. Freeman, K. Keahey, D. Scheftner, B. Sotomayer, and X. Zhang, "Virtual Clusters for Grid Communities", 6th International Symposium on Cluster Computing and Grid (CCGRID), 2006.

[19] "DAGMan", http://cs.wisc.edu/condor/dagman.

[20] M. Litzkow, M. Livny, and M. Mutka, "Condor: A Hunter of Idle Workstations", 8th International Conference on Distributed Computing Systems, 1988.

[21] “Amazon SimpleDB”, http://aws.amazon.com/simpledb.

[22] “Amazon Elastic Block Store (EBS)”, http://aws.amazon.com/ebs.

[23] D. Katz, J. Jacob, E. Deelman, C. Kesselman, S. Gurmeet, S. Mei-Hui, G. Berriman, J. Good, A. Laity, and T. Prince, "A comparison of two methods for building astronomical image mosaics on a grid", International Conference on Parallel Processing Workshops (ICPPW), 2005.

[24] Gluster, Inc., “Gluster File System Architecture”, white paper, October 2010.