Data Management for Distributed Scientific Collaborations Using a Rule Engine

Sara Alspaugh
Department of Computer Science
University of Virginia
alspaugh@virginia.edu

Ann Chervenak
Information Sciences Institute
University of Southern California
annc@isi.edu

ABSTRACT

A virtual organization (VO) consists of groups of many scientists and organizations from geographically distributed regions that pool their computing and storage resources together in order to achieve some common scientific goal via grid computing. Such collaborations often result in the generation of a vast amount of shared data from experimental apparatus or statistical simulation. In order to effectively use this data, VOs develop rules, procedures, and goals for data management, which can collectively be termed *policies*. In this work, we describe the development of a Policy-Driven Data Placement Service based on rule engine. We first provide a simple performance test of the rule engine alone. We then evaluate this service by implementing two practical data management policies, one for dissemination and one for replication, and measuring the performance of the system with these policies. Our work demonstrates that the use of a rule engine in such a manner is feasible and merits further research.

1. INTRODUCTION

Much of modern science research is done by collaborators in disbursed geographic regions who rely on high-performance distributed or grid computing applications to perform tasks such as analyzing raw data gathered from experimental apparatus and performing simulations of various phenomena. Examples of such applications are found in many fields, ranging from physics [8, 17] to biology [9], from astronomy [4] to seismology [25]. These applications tend to be highly data-intensive, processing and generating terabytes and even petabytes of data [5]. Thus, the applications require efficient access, not only to grid computing resources, but also to the data itself. Moreover, the individual scientists in the research collaboration, known as a virtual organization (VO), must also be able to easily and securely find, access, understand, and obtain the data they need in order to use it effectively [12].

To this end, VOs develop procedures to organize and disseminate the data, along with rules to govern maintenance and use, which collectively can be referred to as *policies* [5]. In order to manage their data effectively, VOs must be able to create and enforce their policies for data management. This policy enforcement is complex and difficult to achieve given the large-scale size and distributed nature of the data, as well as the diverse nature of VOs and policies. Thus, efficient, distributed, policy-driven data management is a challenging but critical issue that must be addressed in order to facilitate many scientific endeavors [5, 7].

In this paper, we demonstrate a Policy-Driven Placement Service that is based on an open source rule engine called Drools and that is integrated with two data management serv-

ices from the Globus Toolkit for grid computing applications: GridFTP for transferring data and the Replica Location Service for querying for information on the existence and location of data sets. We argue why this is a feasible and useful approach and provide results for two practical data management policies: one for maintaining a specified number of replicas of each data item in the distributed system and another for distributing data sets based on a tier-like dissemination model.

The paper is organized as follows: Section 2. explains what data management policies are in the context of this work and gives an overview of the operation and use of rule engines. Section 3. describes the architecture of our Policy-Driven Placement Service and details the workings of the two practical data placement policies we implement for our experiments. Section 4. examines our experimental setup, our methodology, and the results of our experiments, and provides a brief discussion. Section 5. reviews related work, and Section 6. concludes with a summary and describes potential ideas for future work to extend from that which is presented here.

2. AUTHORING AND ENFORING POLICIES WITH RULE ENGINES

2.1 Policies

In this work, when we refer to policies, we mean those goals, rules, and procedures developed by VOs for data management or placement. In general, such policies can touch upon a wide range of data-related issues, such as access, replication, processing, dissemination, and provenance maintenance, to name a few. In this paper we primarily focus on policies related to data dissemination, by which we mean the distribution of data to various sites and storage elements throughout the VO, and data replication, by which we mean the creation and maintenance of bitwise copies of data items at various sites in the VO for the purposes of availability, reliability, and integrity. As an example of such policies currently in use in the scientific community, gravitational wave physicists at the Laser Interferometer Gravitational Wave Observatory (LIGO) use a data management system known as the Lightweight Data Replicator (LDR) to both disseminate and replicate their data items according to the metadata queries of scientists at sites throughout the VO [1, 16, 17]. Another example is the high-energy physics Compact Muon Solenoid experiment's use of a data management system known as PheDEx that distributes, in a hierarchical or tiered fashion, subsets of the original data from the site of initial publication at CERN, to sites, organized into tiers, within the VO [3, 8, 22].

We note that in this paper, our use of policy closely corresponds with the usage in [5, 7]. However, in [5] in particular, the author considers data placement with respect to workflow execution, whereas in this work, we focus solely on data placement policies and strategies without considering workflow execution concerns. In addition, authors have described similar though not identical ideas to the policies described here. For example, researchers at the San Diego Supercomputer Center have also developed a rule-engine based data management system known as the integrated Rule-Oriented Data System (iRODS) that focuses on a wide range of data management policies such as those mentioned here, but also incorporates such policies as would be used in a digital repository or library [14, 18, 19, 24]. For instance, the authors mention policies for publication and curation, which would not necessarily be relevant for VOs in a scientific grid computing context [19]. As this work focuses on VO policies specifically in the context of grid computing for scientific applications, we do not consider some of the policies that those authors do. As another example, the authors in [26] describe a data management system for the European DataGrid. The focus of this work is on facilitating data replication, and while they do not explicitly use the term policy, their idea of what constitutes data replication and what is

necessary to implement it is very similar to our idea of a replication policy in this work.

2.2 Rule Engines

A rule engine is a type of program arising from artificial intelligence research in the area of expert systems. It encodes certain knowledge as concisely stated rules in first order logic, which have a precondition, or "if" statement, and a consequence, or "then" statement. In other words, the rules take the form "If x, then do y." State information, or facts, about a certain problem instance are input into the rules engine, and it then uses these rules to arrive at decisions by matching the facts against the rules and executing the consequence of those rules whose precondition is true given such facts. This matching is done by an inference engine, which may use any number of pattern matching algorithms to match the facts with the rules [15, 20].

The rule engine we chose to use is Drools, an open source Java-based rule engine. The inference engine of Drools implements the Rete algorithm [11, 20], which consists of a compilation and an execution phase. During the compilation phase, facts are matched with rules via a discrimination network, which means that the rules' preconditions are used to form nodes in a network, through which facts are propagated and filtered. When a terminal node is reached, the fact has been matched and the rule is executed [11]. In Drools, at runtime, the rules are placed into a repository known as a Rule Base. Facts are inserted into a Working Memory, which makes up a database of current state information. The inference engine matches facts inserted into Working Memory with rules in the Rule Base. The matching of rules and facts is done upon insertion and result in the creation of an Activation for each matching rule and corresponding set of facts. These Activations are then placed onto an Agenda to be executed. Rule execution may alter the current state, so there is a feedback mechanism that allows for the update and re-matching of those rules affected by such alteration [20].

We chose to implement policy-driven data management with a rule engine as opposed to other methods, such as an algorithmic approach, for several reasons. Rule engines are well suited to applications whose logic may change over time and as such require flexibility. As policies will vary from VO to VO, as well as over time as the organization, grid environment, and scientific goals change, this is appropriate. Moreover, given the declarative nature of logic in rule engines, data management policies can be readily characterized as rules that govern the maintenance and movement of data and that can be used to make decisions based on the current state of data and resources within the VO. Another possible advantage is the potential speed and scalability of the algorithms used for pattern matching. In addition, the rule engine can provide documentation regarding how data files were created and manipulated (often called provenance information), since the engine stores information regarding which rules executed and why. Such information is important for reliability and integrity in data management [15, 20].

3. DESIGN AND IMPLEMENTTION

3.1 System Architecture

Our Policy-Driven Data Placement Service consists of three key components: the Drools rule engine [20], the Globus GridFTP client and server [2], and the Globus Replica Location Service (RLS) [6]. Figure 1. illustrates the high-level architecture of our application. The application resides on a single host and directs data movement among remote nodes in order to enforce policies. During execution, the application first queries the RLS for information about the files stored on the grid system, such as the logical name, the number of copies, and the physical location of each copy. It then encapsulates this file information that was obtained from the RLS as facts, and passes them to the

Drools rule engine, along with facts encapsulating storage element information.

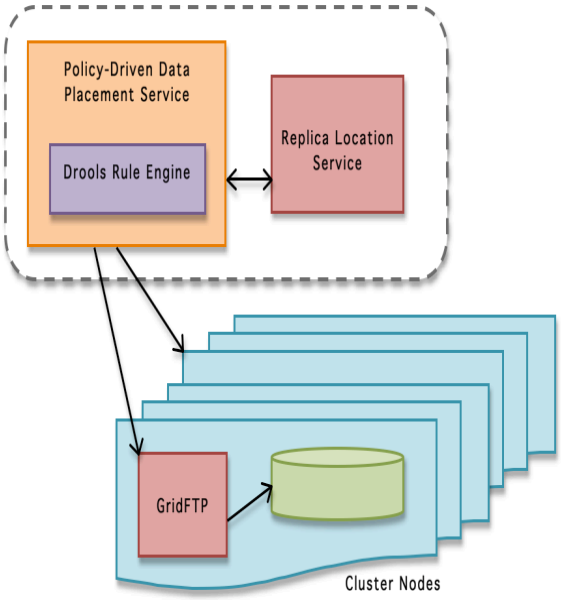


Figure 1 - High-level architecture of the Policy-Driven Data Placement Service

The rule engine contains the data placement policies encoded as rules in its production memory. When facts are inserted, the rule engine performs matching, as described in Section 2.2, and determines which rules to execute with which parameters. Rule execution involves moving data according to the policy encoded in the rule. To enforce the policy, data is transferred via GridFTP among nodes in a cluster. When a file is transferred, the facts about that file and the storage element to which it was transferred are updated and then reevaluated by the rule engine until the conditions set forth by the policies are satisfied.

3.2 Policy Design

To test our approach, we implemented two policies for data management. The first policy specifies a pattern for disseminating data files upon initial publication among nodes on our experimental grid based on a tiered-distribution model, where each successively lower tier is composed of a greater number of sites than the tier above it, but each of these sites also has less storage capacity than sites in the tier above it. An example of such a dissemination model is shown in Figure 2. This model is a simple approximation of that used by the CMS experiment at CERN [8], as described in Section 2.1. The dissemination policy assumes that all files have been initially published at the top tier site, Tier 0. Files are then disseminated to the lower tier sites such that if a file resides in a site at a given tier, it must also reside in some site in each tier above it. Also, the policy ensures that all files in a given tier are distributed equally among all sites in that tier.

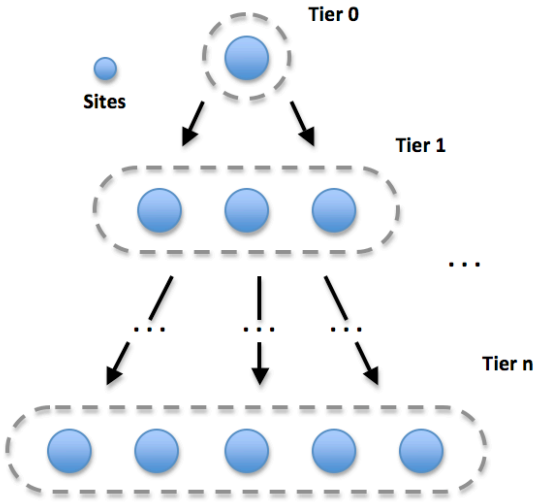


Figure 2 - Tiered dissemination model

The second policy stipulates that three copies of each data file on our experimental grid be maintained. The application obtains the list of data items and their locations from the RLS, and determines the number of copies of each item. The policy then determines whether the number of copies of that file is below three. If this is found to be the case, that file is replicated to other sites, subject to the constraints that no two copies of the same file should reside on the same storage element and the storage element has space for the new copy. This policy is based off of the goal of many VOs of data replication for the purposes of availability, ease of access, reliability, and data integrity. See, for example, [26]. Note that while we use files as the basic unit of data to be replicated, we could have just as easily used collections of files, or

some other construct, as the basic unit instead. Sample pseudo-code shows in Figure 3. demonstrates how this policy was encoded as a rule.

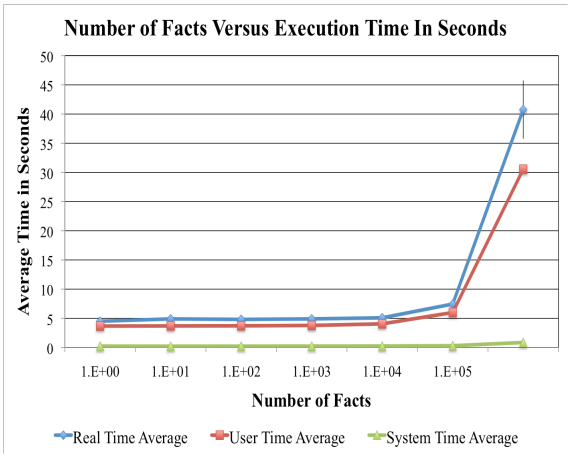
```
rule "replication policy"

when
      exists {f : f is a file and f.copies < 3}
      and exists {s : s is a storage element such that
                  s.availSpace >= f.size and not
                  s.elementOf(f.locations)}
then
      source = f.locations[random(0, f.locations.length)]
      destination = s
      transfer(f, source, destination)
      f.copies++
      f.locations.add(destination)
      RLS.addEntry(f, destination)
      s.availSpace = s.availSpace - f.size
      update(f);
      update(s);
end
```

Figure 3 - Replication policy pseudo-code

4. EVALUATION

4.1 Experiments and Results

In addition to testing these policies, we also conducted a simple test to measure the scalability of the rule engine, which is a necessary requirement for data-intensive high-performance scientific applications. The scalability test was conducted on a single core i686 GNU/Linux machine with 1 GB of memory. In conducting the test we kept a constant number of one thousand rules. We increased the number of facts by powers of ten, ranging from one to one billion. The facts were randomly chosen in such a way that in each case, approximately ten per cent of the facts matched the rules, so in each time, the number of rule firings was equal to one-tenth the number of facts. The consequence of each rule execution consisted of a simple increment operation. Taking timing measurements, we inserted the facts into the working memory, upon which the rule engine performed the matching and executed those resulting activations. We repeated this test ten times for each data point and took the average. To obtain the timing measurements, we used the Linux `time` command line utility. Before executing the test, we first precompiled the rules in order to exclude this from the measurements. Thus, only fact insertion, matching, and rule execution were measured.



Graph 4 - Drools rule engine simple performance test results

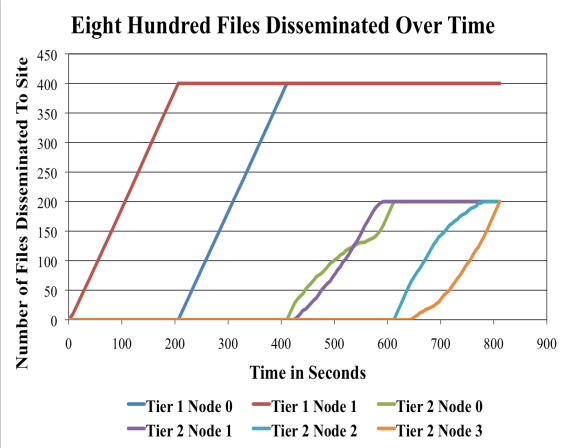
Graph 1. shows the result of these performance tests. Execution with ten facts took 4.9 seconds, increasing up to approximately 7.4 seconds with one hundred thousand facts. At one million facts, the execution time increased dramatically to approximately 40.8 seconds. Finally, although the java heap size was set to the maximum possible value, at ten million facts and beyond, Drools caused an out-of-memory error and did not complete execution. The graph shows the average execution times out of ten runs for each data points along with the standard deviations. In all cases but one, the standard deviation was very small. The exception to this was in the case with one million facts, which had a relatively large standard deviation.

The policy experiments were run on a single core i686 GNU/Linux machine with 1 GB of memory, in conjunction with a cluster of eight nodes, each of which were i686 GNU/Linux machines with 2 GB of main memory. The first machine ran the Globus RLS server and our Drools-based Policy-Driven Data Placement Service, in addition to a GridFTP client. The data files, each 1 MB in size, were replicated or distributed among the eight nodes, which also each ran a GridFTP

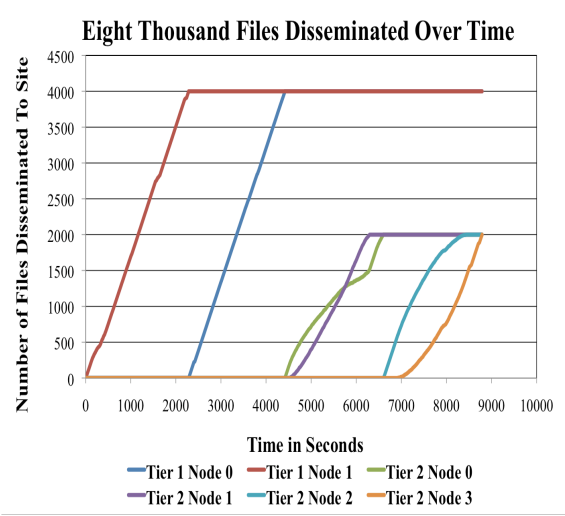
server. This setup is shown in Figure 1 in Section 3.1.

The first policy tested specified a tier-based distribution model for disseminating the data files as described in Section 3.2. In this particular experiment, we used seven cluster nodes, which we classified into three hierarchical tiers of one, two, and four nodes, respectively. We tested this policy first using eight hundred files and then again with eight thousand files, with each file being 1 MB in size. Each time, we assumed that the eight hundred or eight thousand files had been published at the top tier node and registered in the RLS prior to the start of the experiment. We ran our Policy-Driven Placement Service and recorded timing information to measure how long it took for the dissemination policy to be enforced and the files completely disseminated. Timing measurements were taken using the Java library's `System.currentTimeMillis()` method, which returns the time in milliseconds since January 1, 1970 UTC.

At the beginning of the experiment, our Policy-Driven Data Placement Service queries the RLS to acquire the names and locations of files to be distributed, then encapsulates this information as facts to insert into the Drools rule engine, which determines which files to move and where to move them according the policy, which has been encoded in the rule engine as rules. Next, the placement service initiates third-party GridFTP transfers to disseminate the data. In doing this, the service first copies half (four hundred or four thousand) of the total files to each of the nodes in the second tier level and then copies one quarter of the total files (two hundred or two thousand) to each node in the third tier level to achieve complete dissemination of the data. Graph 2. and Graph 3. show the execution times of this policy with eight hundred and eight thousand files, respectively. With eight hundred files, it took approximately .35 seconds to query the RLS and approximately 811.36 seconds to completely disseminate the data. In the larger case, with eight thousand files, it took approximately 1.2 seconds to query the RLS and approximately 8789.64 seconds to completely disseminate the data.



Graph 2 - Dissemination policy execution time using eight hundred files

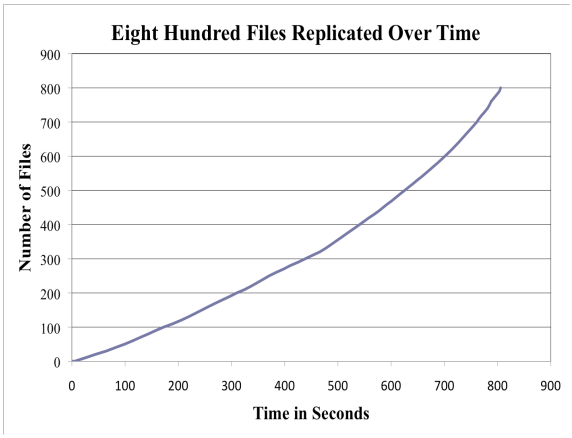


Graph 3 - Dissemination policy execution time using eight thousand files

The second policy tested maintains three replicas of each data file on the experimental grid. We assumed that prior to the start of the experiment there are two copies of each data file distributed on eight cluster nodes. Further, these 1 MB files exist in eight sets of one hundred files each and each node stores two different sets, for a total of two hundred files per node. Again, we took timing measurements to record how long it took for the policy to be completely enforced and all files replicated. For

this, as before, we also used the Java library's `System.currentTimeMillis()` method, which returns the time in milliseconds since January 1, 1970 UTC.

As with the previous experiment, it begins with the Policy-Driven Data Placement Service querying the RLS to obtain the names and of the files and the number and locations of the replicas of each file. It again encapsulates this information as facts and inserts them into the Drools rule engine, from returns the names of the files to replicate via transfer, as well as the source and destination of the transfer. As specified in the rule-encoded policy, the destination node of these files is chosen such that this node does not already have another copy of the file in question, for the purpose of reliability, and the node also has no more than three hundred files already stored on it, to model real constraints on actual storage systems. The end result is that each file has three replicas, with each of the replicas of any given file is stored on a different node, and each node has three hundred files. Graph 4. shows the results of executing the Policy-Driven Data Placement Service with this policy. It took approximately .28 seconds to query the RLS and approximately 805.78 seconds to completely replicate all eight hundred files.



Graph 4 - Replication policy execution time using eight hundred files

4.2 Discussion

For the policy experiments, the timing results include the times for RLS to complete the query response and GridFTP to transfer files, as well as the time for the rule engine to initiate those transfers. Given that querying RLS took at most a little over one second, this component is not a bottleneck for performance. While the simple performance tests seem to be consistent with the policy tests, these results require further explanation to be augmented with further experimentation. This is because the performance of the rule engine depends on a number of things aside from the number of facts. Other factors that likely played a role in the performance of the rule engine include the number of rules, the complexity of the rules, and the machine on which the rule engine ran. For instance, given that using more than one million facts resulted in an out of memory error, it is likely that some performance loss is attributable to the over-taxation of the memory system. It is important to note here that the Rete algorithm, which is the pattern-matching algorithm used by Drools to match rules and facts, is known to sacrifice memory for speed. In addition, the performance tests were run on a machine with relatively small memory. Thus, it is unsurprising that high memory usage was a factor, as demonstrated by the fact that Drools caused an out-of-memory error in the performance tests and showed poor performance once the number of facts neared one million, as described in Section 4.1. One possible solution to this problem stems from the fact that Drools can be integrated with a database, which would likely reduce some of the potential performance loss due to memory issues.

Our Policy-Driven Data Management Service did not make use of the optimized features of the Drools nor GridFTP. This will also have influenced the performance time. Also, our application was written primarily in Java due to the fact that Drools is a Java-based rule engine. It is not clear to what extent this may have influenced performance. Overall, these results do not suggest that the particular rule engine used in this application would be appropriate for

petascale datasets. However, given the possible factors suggested above, and the fact that another rule engine is the basis for a large-scale data preservation system prototype known as iRODS at the San Diego Supercomputer Center with apparent success [14, 18, 19, 21], these results also do not warrant the conclusion that policy-driven data management cannot be based on a rule engine in an efficient and scalable manner.

5. RELATED WORK

In [5], the authors evaluate the use of a Data Replication Service to stage in data to workflows as opposed to having a workflow manager do this work. This choice can be considered a data placement policy. In [6, 7] the authors describe the implementation of one layer of an architecture for policy-driven data placement. [14, 18, 19, 21] describe the development of the integrated Rule-Oriented Data System (iRODS), which, like this work, implements data management policies as rules to govern the manipulation and movement of data. In [26], the authors detail the workings of a data management system used by the European DataGrid for data replication. This system does not make use of a rule engine, but rather, takes what can be considered a traditional algorithmic approach.

6. CONCLUSIONS AND FUTURE WORK

In this work, we have demonstrated the use of a rule engine as the basis for a Policy-Driven Data Placement Service. We have shown that this approach is feasible by implementing two practical policies for data management. One policy disseminates newly published data files to various sites in a VO in a tier-based distribution pattern. The other policy replicates data files in order to maintain a specified number of copies of each file according to certain constraints. We also provided some simple performance data. All of these results indicate that a rule engine-based approach to policy-driven data management warrants further research in the future.

In future work, we intend to expand upon the work presented in this paper in a number of ways. One such way is by testing our policy implementations in a larger and more diverse Grid environment. Another is by implementing a wider variety of other data management policies of increased complexity. We would also like to address many of the questions raised by the results of our experiments. To this end, we will conduct further performance testing of Drools. For instance, we may test it using standard rule engine benchmarks, or test the its integration with a database in order to attempt to quantify the performance effect of Drools' high memory usage. Another interesting exercise would be to profile our policy application code and look for bottlenecks. We would also like Lastly, we would also like to implement several algorithmic approaches to policy-driven data management to examine the trade-offs between the various approaches.

7. REFERENCES

[1] Abramovici, A., et al. "LIGO: The Laser Interferometer Gravitational-Wave Observatory." Science 256.5055 (1992): 325-33.

[2] Allcock, W., et al. "The Globus Striped GridFTP Framework and Server." Proceedings of Super Computing 2005 (2005).

[3] Barrass, T. A., et al. "Software Agents in Data and Workflow Management." Proceedings of CHEP, Interlaken Switzerland (2005).

[4] Berriman, G. B., et al. "Montage: A Grid Enabled Image Mosaic Service for the National Virtual Observatory." ASTRONOMICAL SOCIETY OF THE PACIFIC CONFERENCE SERIES 314 (2004): 593-6.

[5] Chervenak, A., et al. "Data Placement for Scientific Applications in Distributed Environments." Grid Computing, 2007 8th IEEE/ACM International Conference on (2007): 267-74.

[6] Chervenak, A., et al. "The Globus Replica Location Service: Design and Experience " to appear in IEEE Transactions on Parallel and Distributed Systems.

[7] Chervenak, A. L., and R. Schuler. "A Data Placement Service for Petascale Applications." Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07 (2007): 63-8.

[8] "The Compact Muon Solenoid, an Experiment for the Large Hadron Collider at CERN." CMS Project. 2005. <http://cms.cern.ch/>.

[9] Deelman, E., et al. "Pegasus: Mapping Scientific Workflows Onto the Grid." Across Grids Conference 2004 (2004).

[10] Forgy, C. L. "On the Efficient Implementation of Production Systems." (1979).

[11] Forgy, C. L. "Rete: A Fast Algorithm for the Many pattern/many Object Pattern Match Problem." Ieee Computer Society Reprint Collection (1991): 324-41.

[12] Foster, I., C. Kesselman, and S. Tuecke. "The Anatomy of the Grid: Enabling Scalable Virtual Organizations." International Journal of High Performance Computing Applications 15.3 (2001): 200.

[13] Foster, I. "Globus Toolkit Version 4: Software for Service-Oriented Systems." Journal of Computer Science and Technology 21.4 (2006): 513-20.

[14] Hedges, M., A. Hasan, and T. Blanke. "Management and Preservation of Research Data with iRODS." Proceedings of the ACM first workshop on CyberInfrastructure: information management in eScience (2007): 17-22.

[15] Jackson, P. Introduction to Expert Systems., 1986.

[16] "Lightweight Data Replicator." LIGO Project. 2004. <http://www.lsc-group.phys.uwm.edu/LDR/">.

[17] "LIGO - Laser Interferometer Gravitational Wave Observatory." LIGO Project. 2004. <http://www.ligo.caltech.edu/>.

[18] Moore, R., R. Arcot, and R. Marciano. "Implementing Trusted Digital Repositories." Retrieved December 4 (2007): 2007.

[19] Moore, R., et al. "Constraint-Based Knowledge Systems for Grids, Digital Libraries, and Persistent Archives." SDSC Technical Report 2005-9 (2006).

[20] Proctor, Mark, et al. "Drools Documentation." JBoss. 05/05/2008 2008. <http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/index.html>.

[21] Rajasekar, A., et al. "A Prototype Rule-Based Distributed Data Management System." High Performance Distributed Computing Workshop on "Next Generation Distributed Data Management", Paris, France, May (2006).

[22] Rehn, J., et al. "PhEDEx High-Throughput Data Transfer Management System." Computing in High Energy and Nuclear Physics (CHEP) 2006 (2006).

[23] Schopf, J. M., et al. "End-to-End Data Solutions for Distributed Petascale Science."

[24] Singh, G., et al. "A Metadata Catalog Service for Data Intensive Applications." Proceedings of the 2003 ACM/IEEE conference on Supercomputing (2003).

[25] Southern California Earthquake Center (SCEC), 2004. <http://www.scec.org/>.

[26] Stockinger, H., et al. "Grid Data Management in Action: Experience in Running and Supporting Data Management Services in the EU DataGrid Project." Arxiv preprint cs.DC/0306011 (2003).