Using a rule engine for distributed systems management: An exploration using data replication

Quan Pham
1 INTRODUCTION

Dynamic changes in distributed systems are common, due to their many components and the fact that different components are frequently subject to different policies. These changes can make it difficult to construct applications that ensure functionality or performance properties required by users [1]. In order to run efficiently and get high performance, applications must adapt to those changes. (To use a popular terminology, they must incorporate autonomic [2] capabilities.)
However, the logic required to perform this autonomic adaptation can be complex and hard to implement and debug, especially when embedded deeply within application code. Thus, we ask: may it be possible to reduce the complexity of distributed applications by using higher-level, declarative approaches to specifying adaptation logic?

The following example illustrates some of the challenges. The Laser Interferometer Gravitational Wave Observatory (LIGO), a multi-site national research facility, has faced a data management challenge. They needed to replicate approximately 1 TB/day of data to multiple sites on two continents securely, efficiently, robustly, and automatically. They also needed to keep track of replica locations and to use the data in a multitude of independent analysis runs [3]. Yet while the high-level goal is simple (“ensure that data is replicated in a timely manner”), its implementation is difficult due to the fact that individual sites, network links, storage systems, and other components can all fail independently. To address this problem, LIGO had developed the Lightweight Data Replicator (LDR) [4], an integrated solution that combined several basic Grid components with other tools to provide an end-to-end system for managing data. Using LDR, over 50 terabytes of data have been replicated to sites in the U.S.A and Europe between 2002 and 2005 [3]. LDR makes use of Globus Toolkit components to transfer data using the GridFTP high-performance data transport protocol. LDR works well, but its replication logic is embedded within a substantial body of code.

In our project, we explore the feasibility of using a particular approach to declarative programming, namely rule engines, to implement a particular type of autonomic distributed system management, namely data replication. Rule engines are frameworks for organizing business logic that allow developers to concentrate on things that are known to be true, rather than the low-level mechanics of making decisions. We use the Drools rule engine with its declarative programming approach to build our data replication system management. We hope that by showing the applicability of a rule engine with its declarative expression approach to solve data replication challenge, we can prove that rule engines can be used efficiently and with high performance for distributed systems management.

2 BACKGROUND

We first provide some background on rule engines, autonomic computing, and data replication, and review prior related research.

2.1 Rule engines

A rule engine is a software system that executes rules according to some algorithm. It combines a set of facts that are fed in to the system with a rule set to reach a conclusion of triggering corresponding actions. The rules usually describe in a declarative manner the business logic that is to be implemented; in the environments that we consider here, they will normally change only rarely. The facts describe the conditions of the system that is to be operated on; they may change frequently.
A system with a large number of rules and facts may result in many rules being true for the same facts; these rules are said to be in conflict. Different rule engines can use different conflict resolver strategies to determine the order of execution of the conflict rules.

In a rule engine, there are two execution methods: Forward Chaining and Backward Chaining [5]. An engine can implement either method—or, in a hybrid engine Hybrid Rule System, both methods. Forward chaining is a "data-driven" method. Upon facts being inserted or updated, the rule engine uses available facts and inference rules to extract more facts until a goal is reached, where one or more matching rules will be concurrently true and scheduled for execution. Hence the rule engine starts with facts and ends with conclusion.

Backward chaining is a "goal-driven" or inference method, which is reversed with forward chaining. Backward chaining starts with a conclusion or a list of goals that the engine tries to satisfy. If it cannot satisfy these goals, then it searches for sub-goals that it can satisfy that will help satisfy some part of the current goals. The engine continues this process until either the initial conclusion is proven or there are no more sub-goals.

2.1.1 Autonomic computing

Research in autonomic computing [2] seek to incorporate self management functionality into computing systems, with the aim of decreasing human involvement. The term autonomic has been applied to self-configuration, self-optimization, self-healing, and self-protection features.

Self-configuration: For large computing system, the process of installing, configuring the system is error-prone and challenging. Autonomic system can configure itself automatically following some predefined, high-level policies. The policies should specify what the components in the system should accomplish, not how. For example, when a new component is introduced into the system, the new component should be aware of the system configuration and able to adjust itself to the whole system.

Self-healing: When errors or failure occur in a large computing system, it usually takes a long time and much effort for administrators and users to diagnose and trouble-shooting the problems. Sometimes, the problems might disappear without identifying the clear root of failure. An autonomic computing system should be able to detect, diagnose, and repair the system to some extent. If it cannot fully repair the system, it should alert the administrators / developers of the failure.

Self-optimization: An autonomic system can continually find ways to improve its operations. It should be able to tune itself towards more efficient in performance or cost. Through monitoring and self-learning, the system should get more and more efficient. This is a challenge for human tuning in large complex systems with hundreds of tuning parameters and configurations.

Self-protection: An autonomic system should be able to identify and protect against malicious attacks or cascading failures that are not repaired by self-healing. The system should be able to avoid those attacks and failure if possible through log monitoring and other methods.
2.1.2 Drools Rule Engine

The Drools rule engine [6] that we use in this work implements an extended Rete algorithm[7]. The Rete algorithm is a pattern-matching algorithm for implementing production rule systems; it is more efficient than the basic naïve implementation of checking rules serially against the set of facts. The Rete-based algorithm creates a generalized trie of nodes, in which each node corresponds to one pattern in the conditional part of the rules. A path from the root of the trie to its leaf corresponds to one complete conditional part of a rule. When a fact is inserted or updated, it is propagated along the trie and nodes with a matching pattern are annotated. If all nodes on one path from the root to a leaf are annotated, the corresponding rule is satisfied and triggered. The Drools Rete implementation is called ReteOO, which stands for the optimized implementation of the Rete algorithm for object oriented systems.

Figure 1 shows the Drools architecture. Drools stores rules in its *Production Memory* and facts in its *Working Memory*. Facts are asserted into the Working Memory where they may then be modified or deleted. Drools uses its *Agenda* to manage the execution order of these conflicting rules. The default conflict resolution strategies employed by Drools are Salience (or priority, where the user assigns different priority numbers to each rule and the conflicting rule with the highest priority number is executed first) and LIFO (based on an internal assigned action counter value).

Currently, Drools uses forward-chaining method (as of version 5.0). There are plans for backward-chaining support in future releases.

![Figure 1 Drools Rule Engine overview](http://downloads.jboss.com/drools/docs/5.0.1.26597.FINAL/drools-expert/html_single/images/Chapter-Rule_Engine/rule-engine-inkscape.png)
2.2 Related work

Several programming frameworks have been developed that seek to implement autonomic management of parallel/distributed/grid applications, although in different ways. While Automate [8], K-Components [9], SAFFRAN [10], CoreGRID Component Model [11] all provide distributed system-based component frameworks with autonomic capability, each framework has been developed for a specific application. In our project, we want to use a commodity off-the-shelf rule engine to show the generality and applicability of rule engine on distributed systems management.

2.2.1 Autonomic computing

The term autonomic computing was first used by IBM in 2001, and is now known more generally as self-managed computing systems, with the aim of decreasing human involvement. There are many models for autonomic computing. IBM has suggested a reference model [12], the MAPE-K loop (Monitor, Analyze, Plan, Execute, Knowledge), which is being used more and more to communicate the architectural aspects of autonomic systems. In this model showed in Figure 2, the managed element may be any software or hardware resource that is to be given autonomic behavior by coupling it with an autonomic manager. Information about the managed element is monitored by the sensors, analyzed, and then the autonomic manager plans and executes actions on the managed element via the effectors. Goals are usually expressed using event-condition-action (ECA) policies. ECA policies take the form “when event occurs and condition holds, then execute action.”

Based on this MAPE-K model for autonomic computing, there are many implementations in both research and production areas. We review some of those implementations below.

2.2.2 Autonomic Toolkit

The Autonomic Toolkit “provides a practical framework and reference implementation for incorporating autonomic capabilities into software systems” [1]. It is an open set of Java class libraries, plug-ins, and tools created for the Eclipse development environment. It is implemented in Java, using XML messages to communicate with other application, for example, analyzing the
logs of a managed application. At the core, the Automated Management Engine (AME) hosts deployed resource models. Resource models define event types, polling intervals, thresholds, and actions to take when thresholds are crossed. The engine executes resource model scripts within a control loop. It also stores operational data in an embedded local database.

The developers of the Autonomic Toolkit describe an application development suite that provides software developers with a technology to develop autonomic applications, including dynamically self-configuring network services such as DHCP, DNS, LDAP, and other server platforms [1]. However, they do not present any performance measures.

2.2.3 ABLE toolkit

The ABLE toolkit [13] is a multi-agent architecture, implemented in Java. Figure 3 and Figure 4 show the design of the toolkit. The ABLE Rule Language (ARL) can be used to define a rich set of rule-based knowledge representation formats [13]. Using pluggable inference engines, the toolkit supports both forward-chaining and backward chaining algorithms. The implementation of these pluggable engines is not discussed.

The authors describe the set of functionality provided in the ABLE toolkit and demonstrate its utility via three application case studies: system administration, diagnostic application and auto-tune agent for Apache web servers.
2.2.4 Kinesthetics eXtreme (KX)

KX [14] is an implementation of an easily-integrable external monitoring infrastructure. The overview of the system is in Figure 5. KX can be used to add autonomic self-management and self-healing functionality to legacy systems that was not designed with autonomic properties in mind. Its developers describe three use cases in failure detection, load balancing, and email processing to demonstrate their solution. KX is implemented in Java, using the Little-JIL [15] formalism and the ACME ADL [16].

The Event Distiller performs sophisticated cross-stream temporal event pattern analysis and correlation to monitor desirable or undesirable behaviors by performing time-based pattern matching. Internally, according to the authors, the Event Distiller uses a collection of nondeterministic state engines for temporal complex event pattern matching.

2.2.5 Challenges

So far there has not been any comprehensive work on evaluation criteria or metrics for autonomic computing. The definition of how well an autonomic system performs depends on each system. Evaluation criteria can be challenging to define as the evaluation may not be based
on the increased performance of the system, but its ability to meet a certain SLA. An evaluation metric can be the convergence and time for the system to converge to some predefined stable states. Alternatively, there can be an establishment of a representative Grand Challenge Application (e.g., “keep this system running for a week without any human intervention”) that can allow differing techniques to be compared and rated.

### 2.3 Data replication

Data replication is a popular research topic. Its simple definition belies the potential for considerable complexity in an implementation, due to the many independent and interrelated failure conditions that can occur in a distributed system. We survey some relevant prior work here.

#### 2.3.1 The Replica Location Service (RLS)

The Globus Toolkit (GT), from version GT2, includes RLS [17], a simple registry to keep track of the physical storage location of one or more copies of files in a Grid environment. GT users can register files in RLS and later, query RLS for these file locations.

An RLS deployment consists of Local Replica Catalog service (LRC) and Replica Location Index service (RLI) as in Figure 6. LRC stores the mappings between logical and physical location of replicas, and is responsible for discovering corresponding replica of each logical file names. RLI stores information about the logical name mappings from several LRC(s). It is used in a distributed RLS, and can be used to answer user query on LRC. User can query the RLI to find which RLC contains mapping of a logical file name, and then query the RLCs to ask for physical location of those replicas.

![Figure 6 RLS Design](image)

To keep the RLIs updated with LRCs, the LRCs periodically sends information about its mappings to a set of RLIs using soft-state update protocols. Information in RLIs times out and
gets periodically refreshed by subsequent updates. To reduce the network and update delays, RLS implements Bloom bitmap filter [18] to compress the updates.

RLS performance has been measured to be millions of entries and one hundred requesting threads for a single RLS server or for a distributed RLS with multiple RLCs and RLIs [19]. The LRC achieves query rates of 1700 to 2100 per second, add rates of 600 to 900 per second and delete rates of 470 to 570 per second.

However, we need to note that the RLS does not check for correctness or consistency of RLS entries. The RLS is just a registry that allows users to register mappings. Hence it is the users / other application to determine what/how/where to make replica and register to the RLS. Also, if replicas are modified, the users must inform the RLS to update the mappings.

### 2.3.2 Lightweight Data Replicator (LDR)

The LIGO Lightweight Data Replicator (LDR) is a replication tool on multiple sites of a Virtual Organization or Data Grid. It is built on top of the Globus GridFTP for fast file transport, the Globus LRS for keeping track of file locations and a metadata service developed Globus Metadata Catalog Service (MCS) for organizing useful data file information. It provides a mechanism for keeping track of what/where data exists within the Data Grid, for determining what files need to replicated, for scheduling files to be replicated, for actually replicating files, and for storing replicated files.

In Figure 7, a typical LDR deployment, each site needs to run GridFTP server with a local storage, a Globus LRS service with one LRC and one RLI, a Metadata Catalog, a Scheduler Daemon and a Transfer Daemon for file transport.

![Figure 7 Typical LDR deployment on one site](image)

Using the site’s local metadata catalog, the Scheduler Daemon requests a set of files (a collection) with priority as one of its attributes. The Scheduler Daemon then queries the LRS for the collection existence, and if a file in the collection doesn’t exist on local storage yet, the Scheduler Daemon will add that file’s logical name to a priority-based scheduling queue.
The Transfer Daemon periodically checks the Priority Queue, uses the LRS to find location of the logical file name then choose randomly among the available remote sites to retrieve the file in a pull model.

Although LDR can be considered as a minimum collection of components necessary for fast, efficient, robust, and secure replication of data, it lacks the flexibility for users with more complicated scenarios.

2.3.3 Data Replication Service (DRS)

The Globus Data Replication Service [20] is a set of flexible, composable, general-purpose, higher-level data management services to support Grid applications. DRS was designed with the aim of generalizing LDR’s publication functionality to achieve independence from the LIGO infrastructure. DRS is based on GT4 Delegation Service, RFT, LRS and GridFTP services.

Figure 8 shows the deployment and operation of DRS. In a discovery phase (6,7), the Replicator queries the RLI (6) to find the LRCs that contain the mapping of the requested files, filters the LRCs list by user-defined filter, then queries the remaining LRCs to get the physical filename of
the requested files. Next, in the transfer phase (8, 9, 10, 11, 12), the Replicator passes the control to the RFT resource and wait for the GridFTP transfer to complete. Then in the registration phase (13, 14), it adds the new mapping to the LRS services.

DRS performance can vary considerably on operations such as discovery (from 307 to 5371 milliseconds) and registration (from 295 to 4305 milliseconds) [20]. Note that any user replication request must specify the desired files, identified by their logical file names, and the desired destination locations, identified by URLs. Hence there is no automation in the selection of remote replication sites or any consistency check for replicas.

### 2.3.4 Autonomous systems with data replication

One recent work on autonomous data replication is from [21]. This replication system is designed to provide a suitable replica location to minimize file access time according to a user-specified Round Trip Time (RTT) requirement. From Figure 9:

- The Location Information Component (DKS, Node Location Service, AliveInfo) provides information on replica location. It is built on top of the Distributed K-ary System DKS (based on Chord and is a typical DHT)[22], a structured P2P middleware.
- Replica Selection uses the Autonomous Ant algorithm [23]. The ant algorithm reassembles the ant colony. A large number of relatively simple autonomous computing units (ants) are combined together to form the system, following three rules:
  - walk around randomly, until it encounters an object
  - if it was carrying an object, it drops the object and continues to walk randomly
  - if it was not carrying an object, it picks the object up and continues to walk

The ant algorithm is self-organized, adaptive, and distributed. The system uses the ant algorithm to explore participating node without any prior configuration of the environment, initial conditions and topology. The ants walk along the DKS ring to collect information of each place they pass by and record the best position (according to the RTT) in their statuses. The destinations of these ants are the nodes in the first level (level 0) of the DKS routing table for the node where the ants are sent out. At each step, the default next destination for the ant is the successor of the current node; hence it will eventually cover the entire DKS ring.

This paper has given some thought into using autonomous system with data replication. However, the use of the ant algorithm makes it permanent to some specific application, which is not flexible enough for a general framework of using autonomous system with data management.
3 SYSTEM DESIGN & IMPLEMENTATION

The purpose of the data replication system is to maintain a user-specified degree of data replication for user-supplied data objects, according to user rules, using a provided transfer protocol. The replication process should include managing replication sites, replication directories, monitoring directories to update changes to replication sites, and detecting and handling failures. To this end, the system implements the following operations:

- add / remove new replication sites
- add / remove replication directories
- monitoring replication directories for changes
- update changes in replication directories to replication sites

It also supports the following queries:

- File status
  - file replication status
  - number of replications
• location of replications
• Replication site status
  o site availability
  o number of files replicated on that site

### 3.1 System design

Application rules are specified to tell what the target of the replication process is. The Drools rule engine processes the set of rules and performs appropriate actions using provided tools. This whole application is wrapped inside a web service to provide external interface for any users. Figure 10 provides the design of the core system.

#### 3.1.1 Control module

The Control module consists of a rule engine and classes of object as facts in the rule sets. The facts in a set of rule can be

- DataCatalog: name the files that need replicated.
- DataDirectory: name the directories that need replicated. DataDirectory is not added directly to the rule engine fact database, but is crawled to get all the corresponding DataCatalog added.
- ReplicationSite: names a remote site used for replication.
- DataTransfer: provides information about the replication of a file on a remote site. This can be used to count number of replication, storage capability of a site, etc.
- RoundRobin: specifies the mechanism used to choose the next remote site for destination.

#### 3.1.2 Tool module

The Tool module creates the interface to all file transfer protocols. Currently this module only supports the GridFTP protocol; however, since this module is separated from the control module, other file transfer protocols can easily be added.

- FileOperation: interface to GridFTP for remote file/directory operation, such as create/remove remote directories.
- FileTransfer: interface to GridFTP for transferring file.
3.1.3 Web service module

To enable secure remote access to the data replication system, we encapsulate it in a web service running inside a Globus container as in Figure 11. On startup, the container initiates the web service, at which time the replication system is ready to accept replication client requests. The web service client then communicates with the replication system. Via command line interface, users can:

- add replication sites
- add directories
- query the status of replication sites or replicated files
3.2 Implementation

We describe the control, tool, and web service modules, and the rules that implement the data replication logic. We note that our implementation requires the following components:

- JBoss Drools 4.0.7 or later
- Globus Toolkit 4.0.7 Java Web Service Core
- Globus Toolkit GridFTP service
- Cog Toolkit 4.1.5 or later
- JBoss Drools 4.0.7 or later

3.2.1 Control module

The core rule engine is Drools, a business rule management system library with a forward chaining inference based rule engine, using an enhanced implementation of Charles Forgy's Rete algorithm. We use JBoss Drools 4.0.7 in our work. Replication rules are written in MVFLEX Expression Language (MVEL) and Java syntax within one file. To change the configuration of the replication system, users can make changes in a system configuration file and in the rules file. Other classes in the Control module are implemented in Java with reference to the Tool module.

3.2.2 Tool module

The file operation and transfer tools use the Globus Java Cog Toolkit library to interface with the GridFTP service. All file operation and transfer exceptions are handled within the Tool module to provide an abstraction to the Control module. Other file transfer protocols can be added later within this Tool module.
3.2.3 Web service module

The web service wrappers (server and client) of the system use the Globus Toolkit Java Web Service Core. This module provides the communication protocol between users' client and the replication service on a server. However, in the experiment of the system, the interaction between client and server will not be counted towards the system performance.

3.2.4 Data replication system rules

Table 1 lists the primary rules used to implement our data replication system’s business logic. (The implementation also use some query rules to retrieve system information once the replica count reaches requirement. However, since these rules are not fired until the experiment has finished, we do not include them here.)

Table 1: Rules used in our data replication system

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Consequences</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rule &quot;New DataCatalog&quot;</strong></td>
<td></td>
</tr>
<tr>
<td>• $data with STATUS_AVAILABLE</td>
<td>• create a DataTransfer ($data, $site) to start transferring</td>
</tr>
<tr>
<td>• $site with STATUS_AVAILABLE</td>
<td>• update $data and $site</td>
</tr>
<tr>
<td>• no DataTransfer for this $site and $data</td>
<td>• increase connection counter</td>
</tr>
<tr>
<td>• number of replica is less than required</td>
<td></td>
</tr>
<tr>
<td>• connection counter is less than settings</td>
<td></td>
</tr>
<tr>
<td><strong>Rule &quot;Site Became Error&quot;</strong></td>
<td></td>
</tr>
<tr>
<td>• site has STATUS_ERROR</td>
<td>• remove $DataTransfer object (and stop any ongoing transfer if they still exist)</td>
</tr>
<tr>
<td>• there is a $DataTransfer object to this site (finished or not)</td>
<td>• decrease $data replica count</td>
</tr>
<tr>
<td><strong>Rule &quot;Data Transfer Finished Successfully&quot;</strong></td>
<td></td>
</tr>
<tr>
<td>• $DataTransfer has STATUS_FINISHED</td>
<td>• update $DataTransfer to STATUS_DONE</td>
</tr>
<tr>
<td>• update $data</td>
<td></td>
</tr>
<tr>
<td>• decrease connection counter</td>
<td></td>
</tr>
<tr>
<td><strong>Rule &quot;Data Transfer Failed&quot;</strong></td>
<td></td>
</tr>
<tr>
<td>• $DataTransfer has STATUS_ERROR</td>
<td>• remove $DataTransfer object</td>
</tr>
<tr>
<td>• decrease $data replica count</td>
<td></td>
</tr>
<tr>
<td>• decrease connection counter</td>
<td></td>
</tr>
</tbody>
</table>

We present one rule in detail to give a flavor of our what our rules look like. The following rule implementation specifies a name “New DataCatalog,” and indicates that rule will be specified in the Java programming language. The rule contains two parts, the **conditional** part, defined in the “when” clause, and the **consequence** part, defined in the “then” clause.
rule "New DataCatalog"
  dialect "java"
when
  # total number of replicas and in-progress replicas does not meet requirement
  $data : DataCatalog(
    status == DataCatalog.STATUS_AVAILABLE,
    requiredReplicaCount > replicaCount )
  # site still has free resource
  $site : ReplicationSite ( available == ReplicationSite.STATUS_AVAILABLE )
  # site does not have this replica yet
  not DataTransfer( data == $data && site == $site )
  # number of on-going transfers
  $transferCounter : MyCounter( value < 20 )
then
  Config.appendLog("INFO: JOB START: start replicate " + $data + " to " + $site);
  insert ( new DataTransfer ( $data, $site, $session ) );
  modify ( $data ) { addReplicationSite ($site) };
  modify ( $site ) { addDataCatalog ($data) };
  modify ( $transferCounter ) { inc() };
end

The conditional part evaluates whenever:

- A new DataCatalog object $data is inserted or updated in the WorkingMemory of the rule engine.
- A ReplicationSite object $site is inserted or updated.
- A DataTransfer object is updated or removed
- A transferCounter is updated.

The conditional part is designed to evaluate to true if a new replica should and can be created at a particular site. More specifically, it will evaluate to be true if all of the following conditions are true:

- The DataCatalog has available status and has less than the required number of replicas.
- The ReplicationSite is available.
- There is no DataTransfer object that represents the replica of this $data on the $site.
- The number of parallel transfers is less than some setting (here, the value 20).

If the conditional part evaluates to be true, then the consequence ("then") part is executed. In this rule, the consequence part will:

- Insert a new DataTransfer object to perform the transfer. In the implementation of the DataTransfer class, upon construction, the DataTransfer object will start a new thread to transfer the give file to the given remote replication site. Once the transfer is finished, the
DataTransfer object will update its status (success/failure) in the WorkingMemory of the engine.
- Modify DataCatalog and ReplicationSite objects to update the new replica information.
- Update the counter of current parallel transfer.

4 EXPERIMENT AND DISCUSSION

We have sought to evaluate our rules-engine-based data replication system from the perspectives of both implementation cost and execution performance.

4.1 Implementation complexity

To measure our system complexity, we took the Java implementation, filtered it to remove commented and blank lines (see Appendix), and counted the number of lines in different components.

<table>
<thead>
<tr>
<th>Module</th>
<th>Line count</th>
<th>Byte count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control &amp; Tool</td>
<td>1454</td>
<td>49 KBytes</td>
</tr>
<tr>
<td>Web Service</td>
<td>1727</td>
<td>72 KBytes</td>
</tr>
<tr>
<td>Rules</td>
<td>167</td>
<td>5 KBytes</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3348</strong></td>
<td><strong>126 KBytes</strong></td>
</tr>
</tbody>
</table>

Most of the application code is contained within the modules that implement the client interface and perform the file transfer operations. The business logic proper is expressed concisely using Drool’s declarative language. All system logic is maintained in one file that is cleanly separated from the rest of the application code.

Ideally, we would have compared the code size of our implementation with that of other systems. In practice, this was not easy to do. Nevertheless, our review of technical descriptions of other systems makes us believe that that our implementation is significantly less complex and much easier to extend to incorporate new functionality.

4.2 Execution performance

We are also concerned with evaluating the performance of our system. A significant concern is that the use of a rule engine may introduce unacceptable overheads, particularly when the number of facts and matched rules in its agenda grows large. We would like our system to demonstrate the low latency even with high number of files, replicas or rules. Thus, we conduct experiments in which we run the system repeatedly while varying:

- number of files
- file size
• replication ratio
• network failure rate (intermediate failure of with some failure in transfer)
• replication site failure rate (replication site gone down completely)
The runtime experiments mainly aim to assess the overhead due to management of the rule engine.

We run our experiments on Teraport, an IBM e1350 eServer cluster based upon the AMD Operton architecture. We run the replication service on one IBM e325 node with two 2.2 GHz AMD64 processors, 4 GB RAM, and 80 GB local disk. The data that is to be replicated is located on the local disk of this node. Four GridFTP servers are located on four other nodes with the same hardware configuration. These nodes are connected via a switch with available bandwidth of 1 Gb/s per node. For simple comparison and configuration, GridFTP transfers are not striped, and the application can only have a pre-defined maximum number of concurrent connections.

We list experiment parameters in Tables 2 and 3.

<table>
<thead>
<tr>
<th>Table 2 Hardware configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control node</td>
</tr>
<tr>
<td>IMB e325</td>
</tr>
<tr>
<td>Two 2.2GHz AMD64 processors</td>
</tr>
<tr>
<td>4 GB RAM</td>
</tr>
<tr>
<td>80 GB local hard disk</td>
</tr>
<tr>
<td>Replication nodes</td>
</tr>
<tr>
<td>Same configuration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 3 Replication Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of files to be replicated</td>
</tr>
<tr>
<td>File size</td>
</tr>
<tr>
<td>Replica Rate</td>
</tr>
<tr>
<td>Replication Site</td>
</tr>
<tr>
<td>Total rules</td>
</tr>
<tr>
<td>Maximal concurrent connections</td>
</tr>
</tbody>
</table>

Each experiment proceeds in two phases. First, the files that are to be replicated must be identified. At the start of the experiment, the system is given a set of directories for replication. The system crawls those directories to add all files in those directories to the WorkingMemory of the rule engine. This part of the experiment is the Add-File period in the result graphs. During this period, no rule is fired, only the WorkingMemory of the engine is filled with file information.

The second part of the experiment is the Transfer-File period. In this period, the system keeps firing all the rules in the engine. If there is no satisfied rule due to current ongoing transfers, the system waits five seconds before trying to trigger all rules again. In our experiments, the system will stop once there is scheduled transfer after each time all rules are fired. Each experiment is run five times; we record the average run time in the graphs.
4.2.1 No failure during transfer, no replication site failure

We first present the results of experiments in which we measure the performance of the system when replicating large numbers of files via a variety of methods. These experiments are designed to measure the overheads associated with the rule engine.

4.2.1.1 Simulation Transfer

This first set of experiments is designed to evaluate rule engine performance. Each time the rule “New DataCatalog” is fired, the construction of a DataTransfer object will result in a new connection to a remote replication site and a start of the transfer. However, to exclude the influence of network instability and other environment variables, we do not make the real connection to replication sites. Thus, the actual transfer takes zero seconds to finish, allowing us to observe the performance of the application without any external influence.

<table>
<thead>
<tr>
<th>Number of files to be replicated</th>
<th>(10^4 \text{ -- } 10^8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size</td>
<td>Less than 5 KBytes</td>
</tr>
<tr>
<td>Replica Rate</td>
<td>3</td>
</tr>
<tr>
<td>Replication Site</td>
<td>4</td>
</tr>
<tr>
<td>Total rules</td>
<td>8</td>
</tr>
<tr>
<td>Maximum concurrent connections</td>
<td>20</td>
</tr>
<tr>
<td>Transfer failure rate</td>
<td>0%</td>
</tr>
<tr>
<td>Transfer time</td>
<td>0 second</td>
</tr>
</tbody>
</table>

Figure 12 Add-file times (right-hand axis) and transfer-file times (left-hand axis) for simulated transfers, as a function of the number of files replicated

Our results, in Figure 12, show that even when performing no actual transfers, the add-file period is less than 2% of the transfer-file period. As in addition the add-file time appears to grow roughly linearly with the number of files, we conclude that the add-file operation will likely not be a significant contributor to overall execution time, even for a large number of files. Thus, in all of the graphs and discussion that follow, we present only total runtimes: we no longer separate transfer-file and add-file times.
4.2.1.2 Small file transfer using GridFTP

In this second set of experiments, we transfer files via the Globus Toolkit implementation of the GridFTP protocol. Each file is transferred over a distinct connection and thus incurs the authentication and startup, teardown costs of the GridFTP protocol. The file size ranges from 1 KBytes to 5 KBytes, and the average total transfer time of each file is less than one second (including directory creation time if needed). Future development of this system should reuse GridFTP connections to remove the overhead of authentication and connection startup.

![Figure 13 Data replication system performance when using GridFTP](image1)

![Figure 14 Comparison of replication times when using GridFTP and simulated transfers](image2)

We see in Figure 13 and Figure 14 that runtime displays a roughly linear trend when replicating between $10^3$ and $10^4$ files. The $\sim$250 second difference between the simulated transfer case and the GridFTP-transfer case is surprisingly larger and seems likely to involve more than the delay of the GridFTP transfer. Perhaps we are seeing an increase in memory usage and stress on the system when calling the GridFTP library. This difference should be investigated further.
Since the experiments using simulation and experiment using GridFTP show great similarity, all the following experiments are based on simulation with zero transfer time.

4.2.2 Some failure during transfer, no replication site failure

Our next set of experiments are designed to evaluate how well the system performs when data transfers fail. We use the following data replication configuration.

<table>
<thead>
<tr>
<th>Number of files to be replicated</th>
<th>6000</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size</td>
<td>Less than 5 KBytes (same as above)</td>
</tr>
<tr>
<td>Replica Rate</td>
<td>3 (same as above)</td>
</tr>
<tr>
<td>Replication Site</td>
<td>4 (same as above)</td>
</tr>
<tr>
<td>Total rules</td>
<td>8 (same as above)</td>
</tr>
<tr>
<td>Maximal concurrent connections</td>
<td>20 (same as above)</td>
</tr>
<tr>
<td>Transfer failure rate</td>
<td>0% - 20%</td>
</tr>
</tbody>
</table>

We simulate failures by altering the returned result from the FileTransfer class in the Tool package. No rules are changed and replication sites are assumed to be always available.

![Figure 15 Comparison of transferring with no failure and 10% failure rate](image)

![Figure 16 Increase in replication time as a function of transfer failure rate (6000 files)](image)
Figures 15 and 16 show our results. We see that runtime increases roughly linearly as the failure transfer rate changes from 0% to 20%. At 18% transfer failure rate, the runtime increases by 30%. Due to the 18% failure rate, the percentage of re-transfer is:

\[
1 + 0.18 + 0.18^2 + 0.18^3 + \ldots = 1 / (1 - 0.18) \approx 122\%
\]

The actual runtime increase of 30% is presumably due to extra processing performed by the replication system: for example, facts being retracted (to delete failed transfer) and inserted (to add new transfers), and rule matching/triggering. More investigation may be needed to explain the difference.

### 4.2.3 Some transfer failures, one replication site failure

In this final set of experiments, we examine the impact of replication site failure. When a replication site fails, the replication system will determine that it needs to create a new replica at some other site. We use the following data replication configuration.

<table>
<thead>
<tr>
<th>Number of files to be replicated</th>
<th>1000 - 8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>File size</td>
<td>Less than 5 KB (same as above)</td>
</tr>
<tr>
<td>Replica Rate</td>
<td>3 (same as above)</td>
</tr>
<tr>
<td>Replication Site</td>
<td>4 (same as above)</td>
</tr>
<tr>
<td>Total rules</td>
<td>8 (same as above)</td>
</tr>
<tr>
<td>Maximal concurrent connections</td>
<td>20 (same as above)</td>
</tr>
<tr>
<td>Transfer failure rate</td>
<td>10%</td>
</tr>
<tr>
<td>Number of replication site</td>
<td>4</td>
</tr>
</tbody>
</table>

In this experiment, replication site failure is detected by an external script and reported to the rule engine via the rule engine web service interface. The unavailability of the replication site is reflected in the rule engine working memory as a change in the availability attribute of the corresponding object.

We evaluate three different scenarios. In each case, we perform some action after the replication process stabilized and measure the time that the system takes to respond that action.

In the first experiment, we delete a single file from a monitored directory. We observe that the replica management system takes 0.01 seconds to respond to this deletion by deleting the three replicas that have been created for that file.

In the second experiment, we add a single file to a monitored directory. We observe that the replica management system takes 0.01 seconds to respond to this addition by creating three replicas for the new file. Note that in these experiments, we are simulating transfers, so that 0.01 seconds does not include the time required to transfer the replicated file.

In the third experiment, we take down an entire replication site. Figure 17 shows the time required to recover from the loss of a site as a function of the number of files that are being replicated. As noted above, we have four replica sites in this experiment, and each file is to be replicated three times. Thus, if we are replicating N files, we will have 3N replicas after
stabilization, and will lose $3N/4$ replicas when a single site is removed. Therefore recovery requires the creation of $3N/4$ new replicas. This activity should have a cost similar to $\sim 3/4$ of a replication process with a replication rate of 1, and thus we also give that data in Figure 17. Surprisingly, the times for the two activities are quite different. One possible explanation is the different states of the system in the recovery vs. in the warm up processes. In the replication process (warming up before stabilizing), each new object is inserted into the Working Memory and must be pattern-matched with a data set that is still being constructed; whereas in the recovery process, each replacement object is pattern-matched against a data set that has already been organized. These behaviors demand more investigation.

These results show that our rule-engine-based replica management application can respond efficiently to the addition/deletion of files and the failure of a replication site.

5 CONCLUSIONS

We have explored the feasibility of using a rule engine to implement distributed systems management functionality by using a specific rule engine (Drools) to implement a particular distributed systems management function (replica management). Our Drools-based replica management system allows the user to specify, in a declarative fashion, high-level objectives (e.g., that a specified number of replicas should be maintained for each file) and associated business logic (e.g., if too few replicas exist for a file, a new replica should be created; if too many replicas exist, one should be deleted). The Drools-based system then evaluates these rules against a database of facts representing the current state of the overall system, and executes appropriate actions (e.g., create or delete replicas) as required. We have evaluated our solution from the perspectives of both complexity and performance, with satisfactory results.

We conclude from this experiment that it is indeed feasible to use a rule engine - and Drools in particular - to implement distributed system management logic. We have not engaged in any usability studies, but the compact and readable nature of the rules that underpin our implementation make us feel that this approach should be highly attractive to developers. In
future work, we should both implement yet more sophisticated behaviors and measure the effectiveness of developers as they add new capabilities.

We have also evaluated the performance of the Drools rule engine from the perspective of our application. We note that runtime increases linearly, as we might expect, with the number of files that must be replicated. In some settings (e.g., if many files are to be replicated and sites frequently change availability) then we may want to explore alternative implementation approaches: e.g., the replication of collections rather than individual files.

Our results also suggest a range of other topics for future work. From the perspective of performance, we would like to investigate the upper limit in number of files / replication sites a rule engine can handle, and the stability and performance of the replication system after the warm-up process. It would also be interesting to explore other rule engine implementation (not in Java?), and to explore opportunities for distributed rule engine implementations.

From a semantic perspective, we would like to investigate more complex replication policies, such as policies that seek to maximize replication performance by taking into account network topology or that vary replication rates based on recent loss rates. We are also interested in exploring situations in which multiple stakeholders impose policies that must be satisfied simultaneously, as for example when individual sites impose constraints on the maximum space that can be used for different purposes.

6 APPENDIX: RULE COMPLEXITY

RLS code counting:
- wget "http://www.globus.org/ftppub/gt5/5.0/5.0.2/installers/src/gt5.0.2-all-source-installer.tar.bz2"
- tar xjf gt5.0.2-all-source-installer.tar.bz2
- cd gt5.0.2-all-source-installer/source-trees/replica/rls
- find . -name "*.java" | while read filename; do cat $filename; done | grep -v -e "^ *\*" | grep -v "^$" | wc
- same for C code

7 REFERENCES


