Policy-Based Autonomic Storage Allocation

Murthy Devarakonda, David Chess, Ian Whalley, Alla Segal, Pawan Goyal, Aamer Sachedina, Keri Romanufa, Ed Lassettre, William Tetzlaff, and Bill Arnold

IBM Corporation

Abstract. The goal of autonomic storage allocation is to achieve management of storage resources, including allocation, performance monitoring, and hotspot elimination, by specifying comparatively high-level goals, rather than by means of low-level manual steps. The process of automation should allow specification of policies as administrator specified constraints under which the resources are managed. This paper describes the system design and implementation experiences from a prototype autonomic storage manager being developed in IBM Research. The prototype is being developed for a storage network that includes a SAN switch, an IBM Enterprise Storage Subsystem, and AIX servers. Our early experience from this prototype implementation is that there are a large number of mundane manual steps in storage management and it is feasible to automate them such that the automation is driven by higher-level goals under policy control. However, to manage heterogeneous storage a standard ontology is needed for specification of goals and how to achieve them.

1 Introduction

Today storage allocation and management of allocated storage is an incredibly manual process. This is especially true in networked storage, where many choices exist and several elements need to be configured and managed. In allocating network storage, administrators have to determine resource availability, use a methodology for allocation of resources, and execute several commands or use graphical user interfaces to implement the allocation decisions. Once storage is allocated it is necessary to monitor for performance hotspots and capacity, determine how to alleviate these problems, and then carry out appropriate actions to alleviate the problems. The monitoring needs to be an ongoing process to assure acceptable levels of service.

This need for manual involvement in storage allocation and management significantly increases the real cost of storage, is a source of serious errors, often results in significant delays in deployment of applications, and is the root cause of general dissatisfaction with the state of the art of storage management. Therefore, storage vendors are working on automating these processes. At IBM Research, we have been exploring a management automation methodology based on autonomic computing principles [1] and policy-based management [2]. Specifically, the goal of policy-

M. Brunner and A. Keller (Eds.): DSOM 2003, LNCS 2867, pp. 143-154, 2003.

based, autonomic storage allocation is to achieve allocation of storage resources, their performance monitoring, and hotspot elimination by specifying comparatively high-level goals, rather than by means of low-level manual steps. This process of automation should allow specification of policies as constraints under which the resources are managed.

In this paper, we describe the design and implementation of a policy-based autonomic storage allocation manager prototype, called ALOMS-Tango, being built at IBM Research. ALOMS-Tango allows administrators to define classes of service for storage in terms of performance and space metrics, set up alerts to be generated if the actual performance of the allocated storage comes within a given fraction of violating the requirements of the class of service, and visualize the configuration of the storage system for the allocated storage and identify performance bottlenecks.

We modified IBM's database management system (DB2) to enable a user to request a database tablespace container in terms of the class of storage service it requires, forward such requests to ALOMS-Tango, and use the storage allocated as policy-managed autonomic storage.

In response to a request for the creation of policy-managed storage in a particular service class, ALOMS-Tango automatically performs computations and low-level configuration operations necessary to create required storage, monitors the performance of the allocated storage, and produces alerts if the trigger conditions (as specified through the user interface) are met.

The prototype implementation demonstrates the potential that autonomic systems have for reducing the workload on administrators by allowing them to specify their requirements in comparatively high-level terms and by automating routine low-level allocation and monitoring operations. One of the key lessons we learned from the present prototype is the need for a standard ontology to support different types of storage systems. Storage Management Initiative-Specification (SMI-S) [3] is a great start in this direction but our preliminary analysis of SMI-S indicates that further standardization is needed for autonomic storage allocation.

Previous work in the area of storage management dates back to IBM DFSMS on mainframes [4], which provided policy-based allocation of storage and life cycle management of data. More recently, several efforts from the HP Labs have developed tools for automated storage allocation [5][6]. Our work is the same spirit as DFSMS but it extends the ideas to open systems as well as incorporating a feedback loop into the management process (which is absent in DFSMS). Unlike the HP labs work, our allocation management system has been designed for handling dynamic environments that require on-demand storage allocation and management of changes in the storage system status or its usage.

The ALOMS-Tango prototype fits into the larger vision of Autonomic Computing first articulated by Paul Horn [15] and later elaborated upon by Kephart and Chess [1]. Kephart and Chess see new autonomic managers of traditional computing elements such as data base systems and storage systems. These autonomic managers take on tasks that were previously done by administrators. Policy is key to this process as a way to codify the actions and goals of the administrator so that autonomic managers can independently carry them out. The ALOMS-Tango prototype has been careful to

create functions that are well defined with respect to whether they would be in the autonomic data base manager or the autonomic storage manager.

The rest of the paper is organized as follows. Section 2 gives a functional description of our prototype. Section 3 describes design and operation of ALOMS-Tango. Section 4 provides prototype implementation specifics. Section 5 concludes the paper with an overview of what we have learned from the prototype development.

2 Functional Description

With the current storage management tools, when a storage-critical application such as a database requires storage, the process that administrators go through can be time consuming, tedious, and error prone, and can thus result in significant deployment delay. A methodology for allocating and managing storage for database applications is described in this tutorial from IBM [7]. First, administrators determine the performance and availability needs of the application, based on previous experience, a prototype installation, or even using mere guesswork. Next, they determine capabilities of the storage systems they have at their disposal, how the storage systems are connected to application host systems, present storage resource commitments, and the available capacity. Tools available for these purposes are at best inconsistent and often non-uniform across multiple vendor products. Next, administrators use a resource allocation strategy to decide on how to configure the storage systems for the application. The strategy can range from *ad hoc* to the best current practice depending on the experience of the administrative staff and the time available for this purpose. Next, the administrators have to configure the storage infrastructure according to the decisions made in the previous step. Usually, different graphical user interfaces are provided for various elements of the infrastructure as well as for products from different vendors. Therefore, this step is prone to serious human errors. Once these steps are successfully carried out, the administrators are ready to use the allocated storage in an application, such as the database. For example, a database administrator may type a command such as,

```
create tablespace fast1 managed by database using (de-
vice '/dev/rlv23' 20000),
```

to make use of storage that has been created as a logical volume named */dev/rlv23* in an AIX server. Note that many installations use 100s and even 1000s of tablespace containers. From this description the tedious, error prone, and time-consuming aspects of storage allocation should be clear.

Furthermore, regular monitoring of storage performance is critical to assure that the allocated storage meets application requirements. If the initial assumptions about the application requirements are incorrect or the requirements change, the process of reconfiguration is as daunting as or even worse than the initial configuration.

The ALOMS-Tango prototype supports performance goal specification as a named service class that can be later used in a storage allocation request. When a storage consumer, such as the database application, requires storage, it can obtain that storage simply by making a request to ALOMS-Tango, specifying an initial size and a service

class name. ALOMS-Tango also allows setting monitoring policies such that it can automatically detect if a certain storage allocation is missing its service goals by a margin observed over a certain time period. As an example, the following service class can be defined in ALOMS-Tango:

```
Service Class "Gold"
   Maximum size = 100Gbytes
   Throughput = 20 Mbytes/Sec
   Response Time = 5 ms/4K block
   Seq/rand access ratio = 100%
```

The attributes used in the service class definition represent requirements from an application point of view rather than the capabilities of storage hardware. This distinction is quite important and has been discussed in the context of a conceptual framework for policy-based storage management, in earlier publications [4][8]. DB2 can request a storage container of class "Gold" using the following SQL statement:

```
create tablespace fast1 managed by database using
(device '/policy/Gold' 20000)
```

In general, we have considered two interfaces between the database and the storage system; one used when a table space is created and the other when all of the storage backing a table space has been filled. Today, both the Data Base Administrator (DBA) and the Storage Administrator (SA) do part of the storage administration in both of these cases. In order to do automatic storage allocation and administration, the system must be primed with policies that will cause the system to act as the SA and the DBA would have. To accomplish this we have the storage administrator create a number of named storage service classes. The Storage Administrator conveys the available storage service classes to the DBA. When the DBA creates a table space, the policy name is associated with the table space. This allows a DBA to create a table space by only specifying the name of the policy that is to be used when storage space is needed.

The other key moment is when allocated storage has been completely filled. Further data insertion normally results in an out of space error condition, which would impact applications using the database (causing them to roll back). We have defined an interface that the database can use to request more storage. The data base system keeps certain metadata about the storage objects that are being used to back table spaces. This metadata mainly consists of the names of policy-managed logical volumes (in the terminology used in AIX, for example) or files that back a table space. If this association between table spaces and storage objects is kept in both the database and storage there would be potential problems of consistency between them, which would have to be resolved with two-phase commits and logs. In order to avoid this, the interface chosen calls for the data base to hold the association, and present the necessary information across the interface when storage is needed. Thus when storage is needed the database presents the storage policy name and the storage objects currently in use. It is left up to the storage manager to provide more storage either in the form of new storage objects or by extending the size of existing storage objects. This is done in a way that is consistent with both the policy and past allocations.

Note that ALOMS-Tango storage allocation requests can be made without DB2, by using its graphical user interface, command line interface, or the C/XML based pro-

gramming interface. In other words, while we cite DB2 as an important user of this prototype, ALOMS-Tango is neither dependent on DB2 nor limited to it.

An alert policy can be set to monitor throughput on allocated storage, as in:

```
generate an alert, if [throughput] for [a container]
falls below [95%] of the value specified in its service
class definition, in a 10-minute period.
```

In the future, ALOMS-Tango will be extended to support hotspot detection, remediation of the hotspot problems through re-allocation of storage, and a richer set of policies that will automatically select service classes based on requester (i.e. customer, applications, and workload) and usage patterns.

One can clearly see the difference between the present manual process of storage allocation and the automation provided in ALOMS-Tango. The administrators are freed from the tedious and error-prone tasks of determining resource capabilities, bookkeeping of the present usage, resource allocation strategies, and execution of configuration operations. Instead the administrators are given the tools to specify high-level goals in the form of a comparatively small number of service classes, request storage by specifying service classes, and monitor deviations from service delivery through policies. As mentioned earlier, the future plan is to extend this framework with richer policies that include automatic assignment of service classes to storage requests based on the usage patterns and/or requester characteristics.

3 ALOMS-Tango System Operation

Figure 1 shows the system view of the ALOMS-Tango prototype. An application or an administrative user can invoke ALOMS-Tango functions. In our prototype, as stated earlier, we are using a modified version of IBM DB2 as the user of storage. The modified DB2 makes storage allocation requests via a shared library developed for this purpose, and the shared library sends these requests to the ALOMS-Tango Management Unit (ATMU), the box in the right, lower quadrant of Figure 1. The current prototype supplies allocated storage as raw logical volumes, for use as DMS (database-managed storage) tablespace containers in DB2; we also have a variation of the prototype that provides files or file systems for use as SMS (system-managed storage).

While the control flows via the shared library to the ATMU for storage allocation, actual I/O requests go directly from the application using storage to the relevant component of the storage infrastructure, just as they do in the absence of the autonomic storage allocation manager. DB2 has also been modified to send a signal to an extender agent when a tablespace becomes full, which in turn sends a request to the ATMU to enlarge the corresponding container.

The ATMU is responsible for resource provisioning and re-provisioning, collection of configuration information and performance metrics, policy management and enforcement, and user interface support. The ATMU interfaces with the storage infrastructure via sensors and effectors. Sensors help the ATMU in collecting configuration information about the storage infrastructure. They also help in obtaining performance metrics.

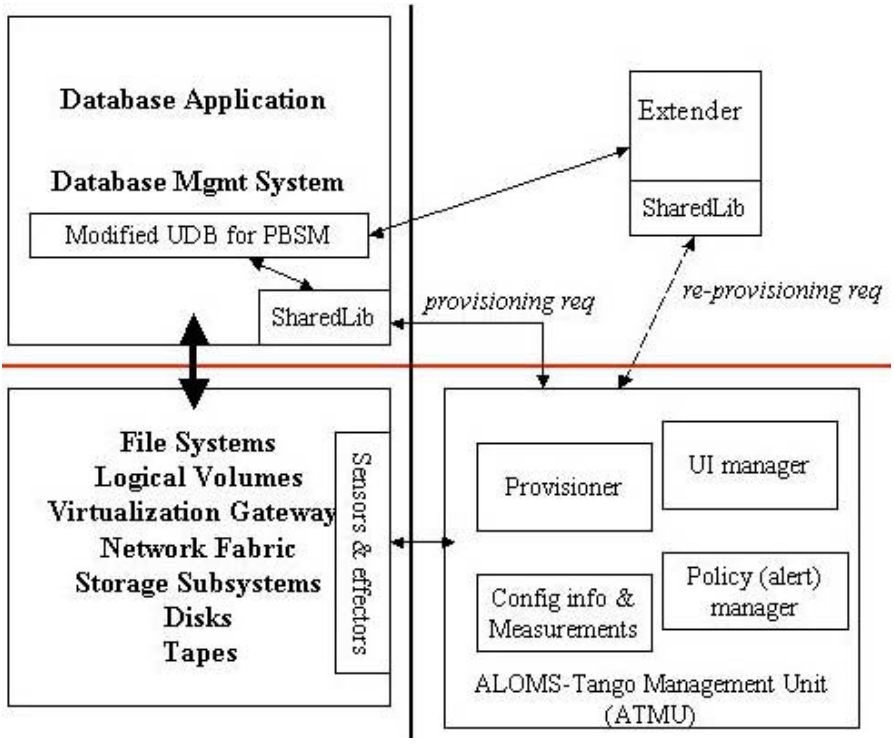

Fig. 1. A system view of the ALOMS-Tango design

The effectors, on the hand, carry out configuration commands as instructed by the provisioning logic in the ATMU. The storage infrastructure includes all elements that enable storing and retrieving of data, such as the operating system, file systems, volume managers, storage subsystems, disks, and tape. Specific infrastructure supported in the prototype is described later.

The UI (user interface) manager component of the ATMU allows the administrator to define the service classes that were introduced in Section 2. A storage administrator can define several service classes, each with a descriptive name. In most environments, service class definitions will be done infrequently relative to the frequency of storage allocation. While these service classes are currently defined in terms of maximum size, response time, throughput, and the ratio of sequential and random accesses in the current prototype, it will be possible in the future to specify additional attributes, such as an availability metric. The UI manager enables alerts to be created, such that they will be generated if the actual performance of a storage container comes within a given fraction of violating the requirements of the class of service. Lastly, the UI manager also helps to visualize the configuration of the storage system and identify performance bottlenecks.

The provisioner component of the ATMU embodies a large portion of the management intelligence and often orchestrates the overall functioning of the management software. At the system initialization time, it builds a logical model of the storage

infrastructure based on the configuration information collected via sensors. This logical model includes information such as the storage space available, whether disks are configured in a RAID format, how many physical paths exist from storage subsystem to the host, and how the physical storage is mapped into the operating system supported data access abstractions (i.e. logical volumes).

The provisioner accepts requests for creation and enlarging of policy-based storage containers in particular service classes, and automatically performs computations and low-level configuration operations necessary to create and enlarge the required containers. The logical model of the storage infrastructure coupled with the knowledge of how to configure each element of the model enables the provisioner to automatically perform the low-level configuration operations. The logical model, capabilities of the elements in the model, and capacity management algorithms enable the provisioner to determine the "best" allocation strategy to meet a given creation or enlarging request.

The policy manager makes use of both a policy repository and a policy execution environment. The policy repository validates and stores policies and policy schemas. The policy execution environment has been carefully designed as an extensible system so that when new policies are introduced into this prototype in the future they can be supported with relative ease. In the current prototype, the policy manager monitors the performance of the storage containers based on the measurements available in the configuration information and measurements component, and produces alerts if the trigger conditions (as specified through the user interface) are met. The policy execution environment is such a key element for extensibility of our system in order to support richer policies; the next section will describe it greater detail.

The configuration information and measurements component is a local repository of the static and dynamic information collected from the storage infrastructure using the sensors. The provisioner as described uses the configuration information earlier. The configuration information also includes information about storage allocations that have already been made. The UI manager component uses this information to present a visualization of the configuration of the storage system, and then in combination with the aggregated measurements it is also used to identify performance bottlenecks. The dynamic information, which is a time series of measurements from the storage infrastructure, is aggregated for the use in the policy-based alert management.

The ATMU is designed and implemented as an autonomic computing element, as outlined in [9]; it presents itself to other systems as a Grid Service conforming to the Open Grid Services Architecture [10] for the purposes of management requests (such as setting a policy or requesting a new data container be allocated). All inter-module and external communication in ALOMS-Tango is via the exchange of XML documents through standard transport protocols.

Policy Execution Environment

The policy execution environment in ALOMS-Tango has been built with concepts and the framework developed in IETF/DMTF policy work [11][12]. It has three subcomponents: a policy agent, a translator, and a rule engine. The policy agent retrieves relevant policies from a policy repository (in an XML schema) and uses the translator

to convert them into a form that is suitable for the rule engine. Rules are executed either in a periodic mode or in a request-response mode, as appropriate.

When the policy manager in the present prototype is initialized, its policy agent sends a request to the policy repository, and in response obtains alert policies that are valid, in-force, and applicable. In addition, the policy agent subscribes to receive updates for future changes to the alert policies.

The policy agent then submits the retrieved policies to the policy-to-rule translator, which performs translation-time checks on the policies. If there are no errors, it creates a set of rules that can be executed by the rule engine. Translation-time checks typically involve range checks and checks to ensure that the policies expressed are ones that the rule engine can execute.

With the generated rules, the policy manager, through the policy agent, invokes the rule engine to evaluate the rules at regular intervals or in response to events, using a new set of measurements for each invocation. If the conditional part of a rule evaluates to true, then the action indicated by the action part is carried out. The rule engine used in ALOMS-Tango is from an intelligent agent construction framework called ABLE [13].

The rule engine references certain variables and functions in evaluating the condition part of a rule, and (when necessary) in carrying out the action part. These typically represent input values received from sensors, and actions such as creating an entry in a log. For example, the alert policy shown in Section 2 may result in the following rule:

```
if (observedValue("pmdo1", "throughput", "minutes", 10)
<  95% of expectedValue("pmdo1", "throughput")) then
(createAlert("logEntry", "pmdo1 throughput is below 95%
of specified value…"))
```

This rule states that if the measured throughput on a policy managed storage object called `pmdo1`, measured in ten-minute intervals, falls below 95% of its expected value, then an entry should be created in the alert log.

In this rule, the functions `observedValue` and `expectedValue` provide access to measurements of the storage infrastructure and service class definitions respectively. The function `createAlert` produces the alert.

Since we wish to use a general-purpose rule engine and to design the policy manager for extensibility, we cleanly separate the rule engine from the rest of system by providing a well-defined interface for resolving rule engine references to variables and functions. This interface consists of callbacks and mapping functions.

For example, for the alert policy above, `observedValue`, `expectedValue`, and `createAlert` calls become callbacks into the alert management part of ATMU, where they are mapped to the appropriate methods in the sensors, the service class definition repository, and the alerting system. This clean separation of policies, rule execution, and variable/function mapping from one another, easily extend the policy execution environment to support new policies, as described in [14].

4 The Prototype

The ALOMS-Tango prototype has been implemented to support several storage infrastructure configurations including IBM SSA drives attached to an AIX server and IBM Enterprise Storage Subsystem (ESS) connected to an AIX server via a SAN switch. Here we will describe the support for the SAN-based configuration.

Figure 2 shows the SAN-based configuration we used for the prototype. An AIX server hosting DB2 is connected to a SAN switch and the IBM ESS is also connected to the SAN switch. The ALOMS-Tango management unit (ATMU) runs in a separate Linux server.

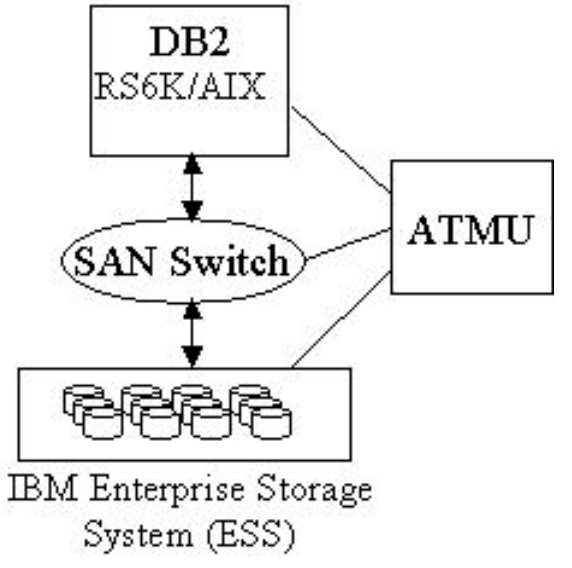

Fig. 2. The SAN-based ESS configuration

The provisioner needs to build a logical model of this SAN-based ESS infrastructure configuration. In the current prototype this building of a logical model is implemented statically by writing a customized "SAN-based ESS" module in the provisioner component of the ATMU that understands this configuration. In the future we hope to develop a generic module that can discover and build a logical model based on standardized management interfaces such as the SMI-S.

For this configuration there are two sets of sensors and effectors, one set for the storage infrastructure in the AIX server, and another set for the IBM ESS. We did not manage the SAN switch hence we did not require sensors and effectors for it. The logical model built for this configuration can be best seen in a screen shot of the configuration of a policy managed data object allocated in this configuration (Figure 3).

Figure 3 shows how a policy managed data object ("PMDO_2") is built from the elements of the storage infrastructure, including the lowest level disk resources. At the bottom of the tree are these physical disks, which are aggregated into a disk group

called dg8, from which a RAID5 array called vs0 is configured. From this RAID5 array a logical disk ("LUN") called 1000 was created and was provided to the AIX server as a physical volume named hdisk2. This physical disk is configured under a pseudo-device called vpath0 to mask multiple paths to the logical disk (Figure 3 shows only one data path). A volume group called tVG_0 is created on this pseudo-device, from which an AIX logical volume named tLV_3 is created as the realization of the policy-managed data object pmdo_2. The dashed line shows the separation between the ESS and the AIX server. The construction of this logical model is performed in a module in the provisioner that is specialized for this configuration.

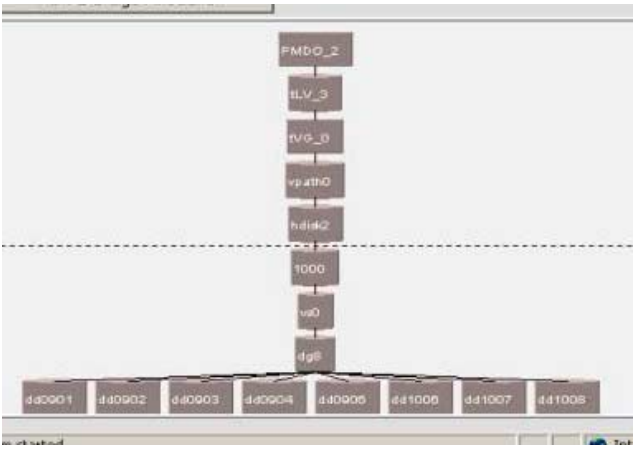

Fig. 3. A logical model of the storage infrastructure for the SAN-based ESS configuration

Having built this logical model, the provisioner then develops a capacity management strategy using this model. In the present prototype, the provisioner creates a fixed number of storage pools (or "bins"), from which it allocates resources to meet incoming creation requests.

To broadly apply this strategy, in the present prototype we create these storage pools at the AIX volume group level (i.e. at the tVG_0 level in Figure 3). Each volume group is assumed to have certain performance and space capacity based on how they are built from the infrastructure elements below it. Figure 4 shows creation of four volume groups each corresponding to a RAID5 array in the IBM ESS.

Logical volumes are created out of these volume groups so as to meet service class goals specified in storage allocation request. These logical volumes are the actual data handles using which the requesting applications can make use of the storage resources. How different requested allocations are assigned to different bins is at the heart of an allocation algorithm. We used a simple "worst-fit" algorithm by which each of the incoming requests is distributed as well as possible across the four bins. In Figure 4, we show how two "Fast" (premium) class storage allocations and five "Slow" (ordinary) class allocations are distributed among the storage pools. Note that the "size" of each allocation, as shown in Figure 4, represents performance requirements, not the more traditional size requirement.

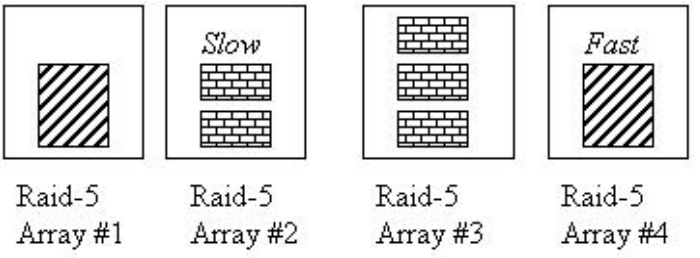

Fig. 4. A capacity allocation strategy used in the prototype

Both the capacity planning strategy and the request allocation algorithm could be improved significantly. For example, the "worst-fit" algorithm is prone to local minima, which may be sub optimal.

5 Conclusion

From this early but complete prototype we make the following observations:

1. The tedious, error prone, and mundane tasks in the storage allocation process can be automated;
2. The ability to build a common logical model of storage for heterogeneous storage subsystems requires a common way of describing many elements in a typical storage infrastructure, including the operating system support;
3. There is a need to characterize important architectural characteristics of a storage subsystem using a standard model so that it can be properly configured, using management software, to achieve desired quality of service. Alternatively, storage subsystems may directly support quality-of-service, by virtue of having internal policy-based autonomic systems. The latter only changes the location of where the autonomic manager runs (i.e. inside rather than the outside of a storage subsystem), and therefore the need for an explicit description of the storage subsystem architecture remains.

Acknowledgements

The authors wish to thank Norm Pass and Jai Menon (both from IBM Almaden Research Center), Lorraine Herger, Steve White, Dinesh Verma, and Hoi Chan (all from IBM Watson Research Center); Jack Gelb and Jimmy Strickland (both from IBM Systems Group) for their invaluable guidance and technical help. Special thanks are due to Al Stuart (IBM Systems Group) for support and for facilitating acquisition and set up of our test environment.

References

[1] Jeffrey O. Kephart and David M. Chess, "The Vision of Autonomic Computing," Computer Magazine, IEEE, Jan 2003.

[2] Dinesh C. Verma, "Policy-Based Networking: Architecture and Algorithms," New Riders Publishing, 2001.

[3] SNIA (Storage Networking Industry Association), "SMI-S Specification, Public Review Draft (v. 1615)," 15 Apr 2003, available at http://www.snia.org/tech_activities/SMI/bluefin

[4] Jack P. Gelb, "System-Managed Storage," IBM Systems Journal, Vol 28, No 1, 1989.

[5] Guillermo A. Alvarez, John Wilkes, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, and Alistair Veitch, "Minerva: An automated resource provisioning tool for large-scale storage systems," ACM Transactions on Computer Systems, Vol. 19, No. 4, pp 483-518, November 2001.

[6] Eric Anderson, et al, "Hippodrome: running circles around storage administration," Proc. of USENIX FAST '02 conference, June 2002.

[7] Barry Mellish, John Aschoff, Bryan Cox, and Dawn Seymour, IBM ESS and IBM DB2 UDB Working Together, IBM Redbook SG24-6262-00, IBM International Technical Support Organization, October 2001 (available on the Internet at ibm.com/redbooks).

[8] Murthy Devarakonda, Jack Gelb, Avi Saha, and Jimmy Strickland, "A Framework for Policy-Based Storage Management," in Proceedings of Policy 2002 (Intl Workshop on Policies for Distributed Systems and Networks), Monterrey, CA, June 2002.

[9] David W. Levine *et al*, "A Toolkit for Autonomic Computing," IBM Developerworks Live, 2003.

[10] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman; "Grid Service Specification," Open Grid Service Infrastructure WG, Global Grid Forum, Draft 2, 7/17/2002.

[11] DMTF, "IETF/DMTF policy framework," http://www.dmtf.org/download/presentations/junedev01/track/0613-01_policy.pdf

[12] DMTF, "CIM Core Policy Model white paper," DSP0108, February 2001, http://www.dmtf.org/education/whitepapers.php

[13] Joseph P. Bigus, Jennifer Bigus, "Constructing Intelligent Agents Using Java," Second Edition, John Wiley & Sons, Inc., 2001.

[14] Murthy Devarakonda, Alla Segal, and David Chess "A Toolkit-Based Approach to Policy-Managed Storage," in Proceedings of Policy 2003 workshop (Intl Workshop on Policies for Distributed Systems and Networks), Lake Como, Italy, June 2003.

[15] Paul Horn, "Autonomic Computing: IBM's Perspective on The State of Information Technology", IBM Corporation, http://www.research.ibm.com/autonomic/manifesto, October 2001.