RI 01006, April 2001 **Computer Science**

IBM Research Report

An Architecture for Virtual Server Farms

Vikas Agarwal, Girish Chafle, Neeran Karnik, Arun Kumar,
Ashish Kundu, Johara Shahabuddin, Pradeep Varma

IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016, India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: **This report has been submitted for publication outside of IBM and will probably be copyrighted is accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T.J. Watson Research Center, Publications, P.O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com).. Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .**

An Architecture for Virtual Server Farms

**Vikas Agarwal, Girish Chafle, Neeran Karnik, Arun Kumar,
Ashish Kundu, Johara Shahabuddin, Pradeep Varma**

IBM India Research Laboratory,
Block - I, Indian Institute of Technology,
Hauz Khas, New Delhi 110016, INDIA

{avikas, cgirish, kneeran, kkarun, kashish, sjohara, pvarma}@in.ibm.com

Abstract We generalize the traditional physical, customer-dedicated application server in server farms to a logical, distributed counterpart. The *virtual servers* thus defined break traditional, static boundaries separating resources (such as physical servers or static partitions thereof) dedicated to customers, and open up new degrees of freedom for optimized and shared use of server farm resources. An additional benefit of shared resources is the substantial savings in redundant capacity required for handling the peak loads of independent customers of the farm – the probability of facing simultaneous peaks goes down multiplicatively. This benefit derives from the observation that, typically, the customer's load is determined dynamically, and that redundancy has to be built at the level of each static resource unit dedicated to a customer (a unit may individually get overloaded, even while other units are not). By co-allocating components of independent virtual servers on the same physical server, and by keeping the boundaries between the co-allocations dynamic, the redundancy requirement on the physical server is shared and brought down. We propose an architecture for a server farm based on this notion of virtual servers. Our system has the benefit of supporting customer-specific Service Level Agreements (SLAs) in terms of simple metrics of user-generated load. Customers need not know about the deployment and configuration details inside the server farm, but simply need to specify the functionality sought by them and negotiate a (load-based) price. Resource deployment, management, and operation are then the responsibilities of the server farm, while the customer is billed for actual usage as per the SLA. We describe the architecture of the virtual server farm and discuss its management and operational details.

Keywords: Virtual Server, Resource management, Server farms, Application Service Providers, Load balancing, Quality of service, Service Level Agreements.

I. INTRODUCTION

Application Service Providers (ASPs) host applications that support multiple customers off a shared infrastructure. This model takes advantage of economies of scale (in terms of redundancy, peak-load handling, IT skills, etc.) and of the need for the companies to focus on their core competencies rather than increasingly complex IT management tasks. Therefore, there is clear trend for corporations to outsource applications, driven by the ASP model. Moreover, since the infrastructure is rented, most of the hardware and software is off-site and shared thus providing the ASP with ample scope for optimization of resource usage.

We propose an architecture for implementing an ASP's server farm such that a customer's application is supported using a *virtual server* – a set of distributed software resources, communicating with each other. A *virtual server farm* is a set of virtual servers coexisting on a set of shared physical machines. We describe mechanisms for automatically managing (allocating, reallocating, provisioning, load-balancing) the resources of such a farm, including dynamically reconfiguring the farm based on current needs of various customers, governed by their Service Level Agreements (SLAs). This system improves resource utilization by reducing the resources required for handling peak loads of multiple customers, while ensuring that SLAs are not violated.

The proposed system allows customer-friendly SLAs to be specified in terms of the request (*hit*) rate for the customer's application. It maps these onto actual resource requirements, using the benchmarked characteristics of the application being hosted. A difficulty of working with hit rates and benchmarks is that actual hits can vary significantly from benchmarking hits. Thus the actual load imposed can be significantly different than the expected load computed using benchmarking hits. Since our system is oriented towards handling of peak loads using shared redundancy, it is able to interpret deviations from expected load as just another variation to be handled using shared redundancy. This allows the use of simpler SLA definitions and reduces the IT awareness required from customers. A customer need not worry about suggesting detailed server configurations. He only specifies a high-level functionality and negotiates a hit-rate-based price function with the ASP. From the perspective of the ASP, our approach allows software/applications to be shared in addition to hardware sharing. This creates more degrees of freedom for optimization and reduces the extent to which redundancy needs to be provided. This results in significant savings for the ASP.

Most current ASPs provide simple services such as web hosting, with customer-dedicated physical servers. Others allow servers to be shared in relatively static mode, with SLAs that are dependent on farm implementation. What is lacking is the technology for dynamic, global, efficient resource allocation that can minimize the cost of hosting and handling peak loads intelligently. Moreover, manual administration costs are high in the running of such server farms and the lack of automation compromises scalability and resultant economies of scale.

Océano [AFF01] improves upon current ASP technology. It provides SLA monitoring, automatic need-based server allocation, and high availability. This enables server farms to automatically handle traffic surges in multiple applications across customers. However, Océano handles allocation and de-allocation in units of entire machines. Besides an inefficiency as far as redundant capacity is concerned, this approach rules out customers

from the small and medium business (SMB) segment whose individual requirements for any given application do not add up to a whole machine. Moreover, only hardware resources are shared – applications are not.

A majority of ASPs that claim to offer shared servers merely use *virtual hosts* as offered by most HTTP servers[1]. They do not guarantee quality of service, high availability, scalability or resource optimization of the hosting farms. Some exceptions like Ensim [ENSIM] do support the concept of machine fractions, which they call *private* servers (PSs). However, system administration for say, migrating customers across PSs or increasing PS resources, must be done manually. In particular, there is no automatic sharing of redundant capacity for handling dynamic fluctuation, or for that matter balancing of load across different servers. Finally, servers are relatively statically partitioned in that redundancy cannot be recovered automatically by pushing back static minimum boundaries of a PS, for supporting another customer.

Our system scores over Ensim's and similar solutions by automating resource management, optimizing resource usage, distributing load dynamically based on available capacity, and sharing resources among different customers. The virtual servers in our solution are not static. A virtual server is distributed over multiple machines and its boundaries within any machine are kept dynamic and automatically adjustable. This is useful for reassigning resources to the customers who need it most.

We describe our system as follows. In section II we provide an overview. System architecture is discussed in section III. System initialization, scheduling, and fault handling are discussed in section IV. Section V discusses related work. Finally, conclusions are presented in section VI.

II. OVERVIEW

Resources and Resource Classes: A server farm can be viewed as a collection of *resources*. These include hardware as well as software resources. The hardware ranges from simple PC servers for hosting low-volume websites to high-end mainframes for performing complex financial or scientific analyses. Similarly the software may vary from productivity tools (word processing, spreadsheets) to applications (payroll, online malls) to complex algorithms for specialized domains (games like chess, scientific computations like weather forecasting, etc.).

We partition the applications typically needed by the ASP's customers into a set of *resource classes*. A resource class encapsulates the properties and functionality of a particular type of resource. For example, an IBM DB2 server is an *instance* of the resource class <database server>, Apache's HTTP server is an instance of the resource class <web server>, etc.

Virtual Servers and Virtual Server Farms: An application is composed of a set of resource classes. An "application server" comprises instances of resource classes in that set. Such a server does not correspond to a hardware server in the physical world – we call it a *virtual server* to highlight this. For example, a web-based retail store application could be composed of a front-end <web server>, a middle-tier <e-commerce server> and a

[1] See *http://www.mindsethosting.com/* and *http://www.wwow.com*

back-end <database server>. When this application is deployed in the server farm, multiple instances of each resource class may be created, to enable handling of all the incoming requests in a timely fashion. For example, Acme Inc.'s virtual server may be composed of, say, five instances of the Apache web server, two instances of the Websphere Commerce Server[2] and one DB2 database server. These numbers can vary dynamically, based on the current load on Acme's web-store.

Thus, a customer's application requirements can be converted into the specification of a virtual server. Three such virtual servers are shown in Figure 1. The ASP must map the resource instances onto physical machines. Our approach is to dedicate machines for particular resource classes, thus creating *sub-farms* of machines with identical software resources (see Figure 1). This has the advantage of simplifying the ASP's maintenance chores, and eliminating the interference of other resources in the monitoring of a particular resource. Each resource instance can reside on a single machine, or on a set of machines. Conversely, one machine may host multiple resource instances assigned to different customers. The system treats various instances of the same class to be interchangeable with respect to a customer's application, assuming that resource classes support standard interfaces.

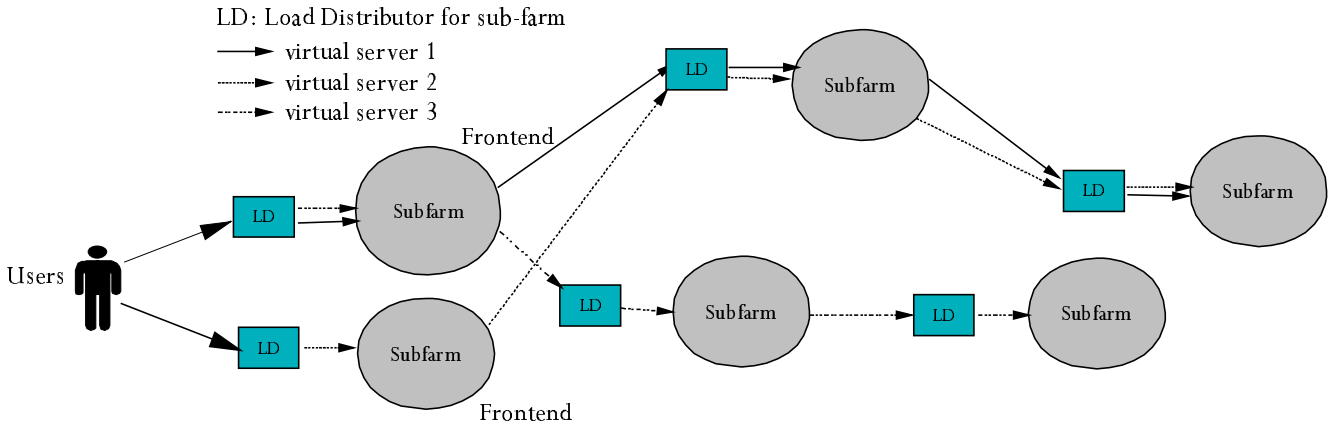

Figure 1. Sub-farms Comprising a Virtual Server Farm

Benchmarking: The *capacity* of a machine, with respect to a resource class, is measured in terms of the hit rate that the resource instance can support, when deployed on that machine. All hosted applications are benchmarked to determine how an input hit rate on the application translates to hit rates on different constituent resource classes. The ratios between these hit rates are termed as *correspondence ratios*. Secondly, we measure the physical resources (such as CPU, RAM, etc.) required by a constituent resource class of an application for a request. This is done for all the machine types in the server farm. The benchmarking results in a k-tuple (one for each physical resource) of usage due to a *reference hit* on a resource class. A reference hit is a notional hit that corresponds to the measured average load. For many popular applications, such benchmarking information is

[2] See http://www.ibm.com/software/webservers/commerce/

available from developers. For other applications, the ASP generates this information by experimentation. Alternatively, the application could be deployed based on liberal initial estimates of the required physical resources.

Service Level Agreements: As discussed in Section I, we wish to isolate customers from technical details of the ASP's resources. The customer specifies its requirements in terms of the rate of requests expected for the application. Different values of the hit rate may be specified for different time periods. Based upon this, the customer and ASP negotiate the charges to be levied. In further discussion, we assume that an SLA includes the following:

- The range of hit rates that the application is expected to support, and optionally, an average hit rate
- The price that the customer has agreed to pay for different sub-ranges of this range
- The penalty that the ASP must pay to the customer in case its minimum guarantees are not met

III. SYSTEM ARCHITECTURE

Figure 2 shows a high-level architecture of a virtual server farm. The ASP hosts several customers, whose users access the applications on the ASP infrastructure via the Internet. During the initialization, a customer's application is split into *Resource Classes* as described earlier, and is mapped onto a virtual server. Detailed information about the application and layout of the various resource instances on physical machines is maintained in a *Configuration Repository* (**CR**). The system has many *sub-farms*, each consisting of multiple instances of a resource class. Traffic to a sub-farm is fed through its *Load Distributor* (**LD**), which distributes the load amongst the different instances under it (see Figure 1).

Each resource instance has an associated *Monitor*, which is responsible for collecting metrics on its usage. This information is made available to the *Aggregator*. Based on the current usage, the Aggregator determines changes required in the resource allocation for each customer, and suggests these to the *Global Decision Maker* (**GDM**). The GDM re-computes the resource allocation to customers with a view to optimize parameters such as revenue, resource utilization, or perturbation of the current allocation. The new allocation is provided to the *Resource Manager* (**RM**), which is responsible for actually implementing the changed allocation plan. Depending upon the scale of the server farm, one or more of these system components may be realized as a collection of interacting modules rather than as a single monolithic component.

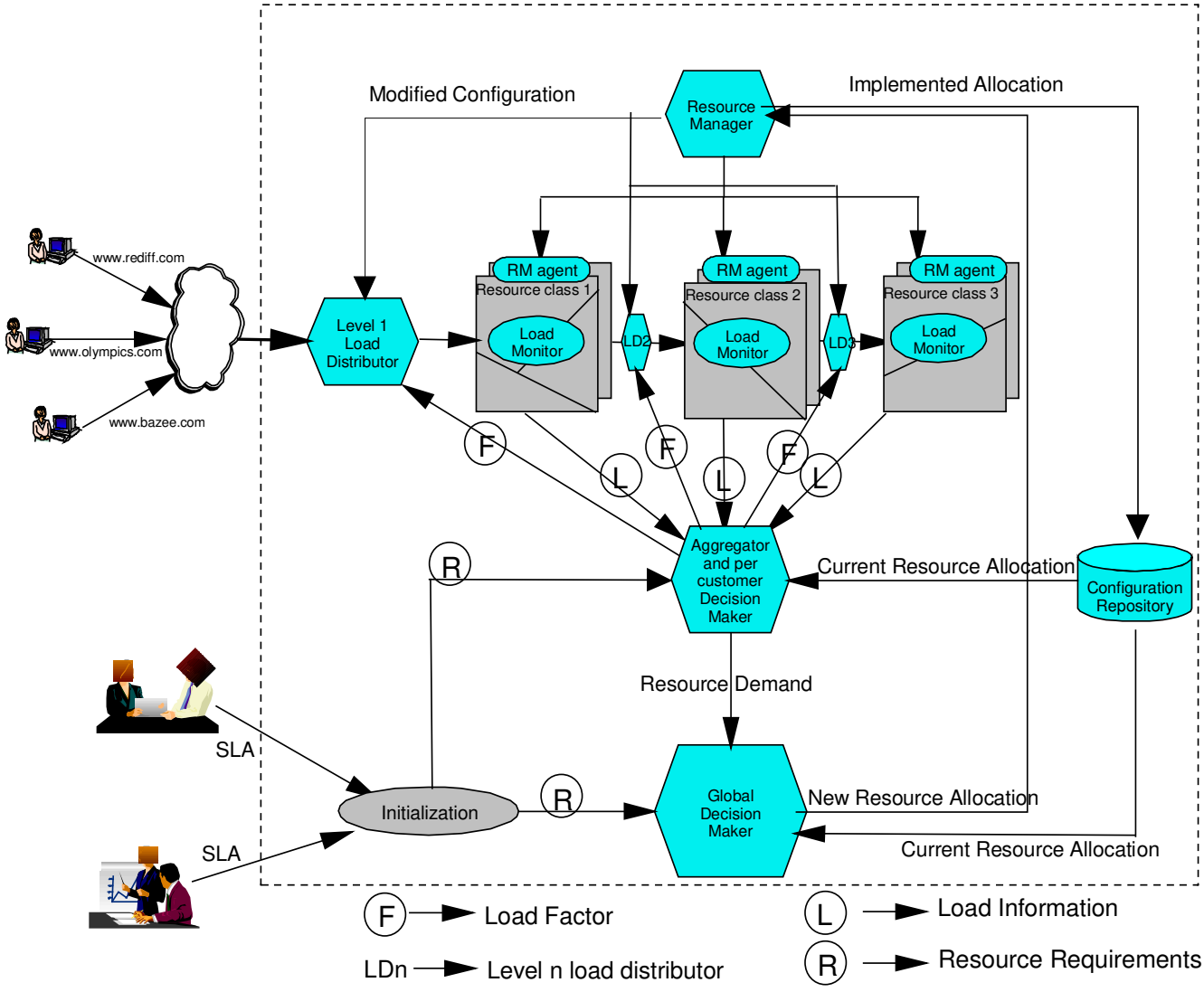

Figure 2. Architecture of the Virtual Server Farm

1. Load Monitors

These are used to monitor usage of various resource instances for management and billing purposes. There are two types of monitors for each resource instance:

- Hit rate monitors: They measure the number of hits on an instance per unit time.
- Machine load monitors: They measure the physical resources consumed, as a k-tuple consisting of parameters such as CPU, memory, etc.

Hit rate monitoring is done by the *load distributor* module (discussed later). It measures the hit rate for each resource instance by aggregating the number of hits forwarded. Machine load monitoring can be done using system programs such as *ps* or *top* in Unix, OS-level information, third-party monitoring tools, or load information provided by the application itself. In addition, fault monitoring of resources can be done using a heartbeat mechanism. Both load and fault monitoring interfaces can be standardized using standards like Java Management Extensions [JMX] for supporting a plug and serve approach. Also, well-known techniques such as

[IBMSP] can be used in accumulating monitored data. Finally, simple rule-based decisions can be used which follow guidelines such as "declare a machine faulty if X% of its resource instances are unavailable", etc.

2. Aggregators

Aggregators collect load information from monitors and aggregate them for efficient decision making. There are two levels of aggregators as shown in Figure 3.

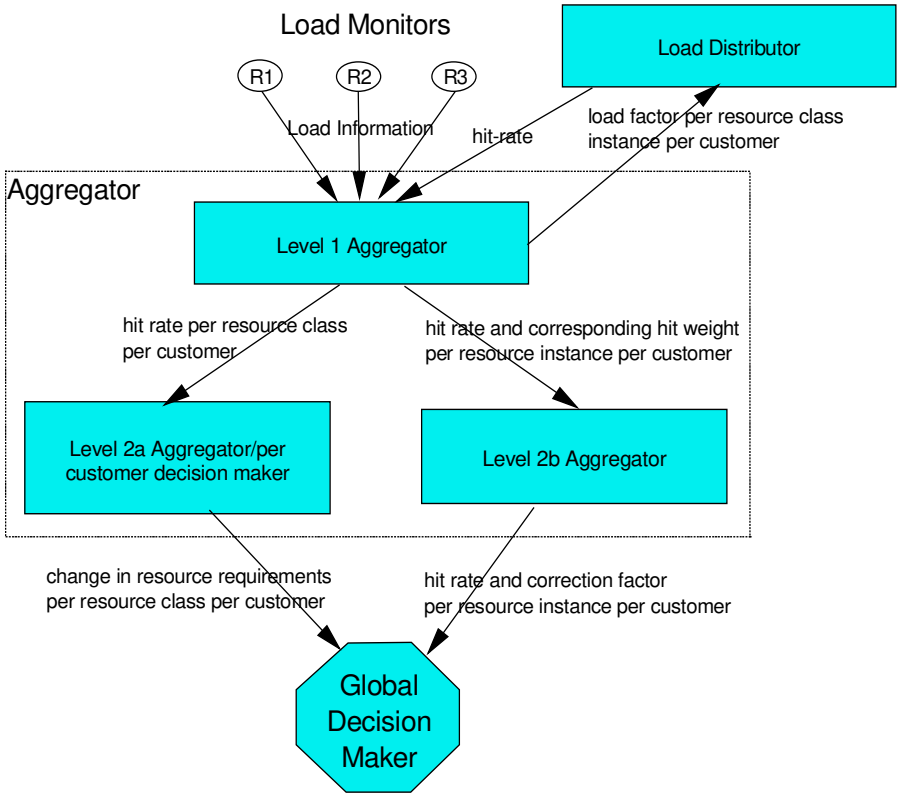

Figure 3. Aggregation of Load Data

Level-1 aggregator receives monitored data (current hit rate and the resource usage k-tuple) for each resource instance and aggregates it on per-resource class, per-customer basis. The aggregated hit rate, of each resource class of a customer, is obtained by adding the hit rates of each instance of that class. It sends this information to Level-2a aggregator. Secondly, it computes the *hit weight* per resource instance as a ratio of actual resource usage per hit to the resource usage of a reference hit. The hit weight represents the deviation of load generated by observed hits from the benchmarking information. It is sent to Level-2b aggregator along with the corresponding hit rate. Finally, it computes the *load factor* for each resource instance as the actually used fraction of the allocated resources and sends it to LD, which uses it to distribute the load proportionally.

Level-2a aggregator acts as *per customer decision maker* and makes demand projections for each customer, in units of reference hits, to the GDM. Whenever a customer's load reaches within a factor α of its current allocation, an increase β in allocation is demanded for that customer. Similarly, γ and δ are the corresponding parameters for de-allocation of resources. These demand projections are compared against the

customer's SLA and pruned down, if needed. The parameters α, β, γ and δ may be constants or functions of variables such as load, current resource allocation or SLA limits, and are configurable for each customer.

A demand for the *front-end* resource class (which receives requests from the users) of the customer's application is treated differently. Such a request gets translated into a set of requests consisting of appropriate number of hits for all the resource classes of the application. To compute this, it uses the correspondence ratio, whose current value is obtained from CR. If there is a significant change in the observed ratio over a period of time, the value in the CR is updated.

Level-2b aggregator computes *correction factors* for each resource instance. These factors compensate for deviations of actual hit load from the reference-hit load. The aggregator obtains the previous hit-weight from the CR, and current hit rate and hit weight from Level-1 aggregator to compute the correction factor as

*Correction factor = (current hit weight – previous hit weight) * number of reference hits allocated*

It also computes the total deviation by aggregating these correction factors across all instances of each resource class for each customer, and sends it to the GDM along with the correction factors.

3. Global Decision Maker (GDM)

The GDM accepts a set of latest demands as input and computes an allocation plan for the server farm, with the aim to maximize the ASP's revenue. The plan computation matches the supply of machines to customer demands in various resource classes. We model this as a mixed integer linear program [BHM77]. The global optimization problem is hard to solve in an exact manner so we rely on a heuristic solution. There are many candidate approaches such as standard LP techniques, problem LP relaxation, cutting planes, branch-and-bound, and column generation techniques, that can generate efficient, approximate solutions. We choose a simple heuristic comprising of solving using LP relaxation and then re-solving the simplified problem after setting the integral variables to rounded off (but feasible) values. Appendix A contains details of our model. A preprocessing module simplifies variables and constraints so as to speed up the solution. The solution process is halted after a short time for the best solution obtained till then. The new allocation plan is given to the Resource Manager for deployment.

4. Resource Manager

The Resource Manager (**RM**) translates the GDM's allocation plans into actions performed on physical machines and actual processes, to arrive at the desired configuration. It creates and deletes resource instances, interfaces with OS-level mechanisms to change resource allocations, and creates and deletes sub-farms. All these actions are transparent to the customers' applications.

The RM has an active component (**RM Agent**) on each machine in the farm, which provides an interface for managing resource instances on that machine. When the machine boots up, the agent starts executing and registers its location in the Configuration Repository (CR). It then queries the CR to determine the low-level *startup* and *shutdown* commands for the resource class on its machine.

A machine in a sub-farm may host multiple instances of one resource class, each having a unique identifier (**instanceID**). Each instance may support hits from multiple customers, and each customer may be allocated a different maximal hit rate on that instance. The RM Agent for the machine maintains a table for each instance, containing these hit-rate allocations per customer. The agent presents the following interface to the RM:

Startup( )	Start a resource instance and return its instanceID
shutdown(instanceID)	Shut down the specified resource instance
setAllocation(instanceID, customerName, hitRate)	Set a customer's allocation on the specified instance, to the given maximal hit rate

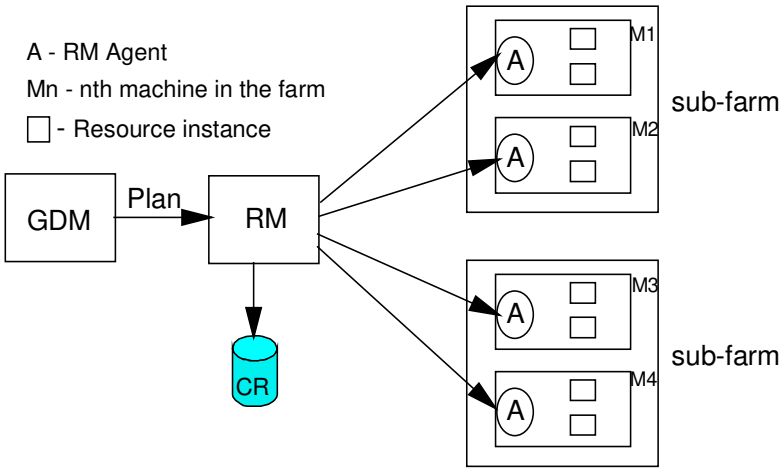

Figure 4. Resource Manager interactions

Implementing an allocation plan: The RM has a current plan reflecting the current status of the farm, and a new plan provided by the GDM. At farm startup time, the current plan indicates that no resources are allocated to any customer. The RM waits until a fresh allocation plan is generated by the GDM. This plan is a three-dimensional table. For each customer, it lists the allocation for each instance of every resource class that the customer requires. The RM compares the resource instance allocations of the new plan (*new_alloc*) with the ones in the current plan *(curr_alloc)*. Several cases are possible:

Case I: ***curr_alloc = new_alloc***. Because there is no change in allocation, no action needs to be taken.

Case II: ***curr_alloc* = 0, *new_alloc* ≠ 0**. A new instance may have to be created (if all customers' current allocations on it are zero). The RM invokes the **startup** operation on the appropriate agent. The agent creates the instance and either starts a new load monitor for it, or assigns the responsibility of monitoring it to an existing monitor. It then sends an acknowledgment to the RM, which subsequently uses the agent's **setAllocation** operation to initialize the customer's hit-rate allocation to the *new* value. The agent then sends a message to the Load Distributor of its sub-farm, informing it about the creation of the new instance, so that incoming requests can be forwarded to it.

Case III: ***curr_alloc* ≠ 0, *new_alloc* = 0**. The instance may have to be destroyed if it is no longer needed (i.e. if all other customers' allocations are also reduced to zero). The RM then invokes the **shutdown** operation on the appropriate RM agent. The agent first directs the sub-farm's Load Distributor to stop sending hits to that instance, and then uses the resource's *shutdown* script to stop the instance and its associated monitor. The instance can shut itself down cleanly after servicing any pending requests.

Case IV: ***curr_alloc* ≠ *new_alloc* ≠ 0**. The RM uses the **setAllocation** operation on its agent to modify the allocation. The agent updates its internal tables accordingly and sends a message to the Load Distributor indicating the new allocation. The LD is responsible for enforcing this limit by throttling incoming requests if necessary. If *new_alloc* > *curr_alloc*, the allocation has been *increased*. An increase in a resource without a corresponding increase in the next resource to which it sends hits, is not useful – it only results in throttling of the excess requests until that next resource has also been expanded. Because of the unpredictable delay involved in sending messages to the respective agents, the LD must wait until all dependent resource instances have also received their increases before allowing more hits through. For this purpose, RM agents send acknowledgment messages to the RM, which implements barrier synchronization. After all the agents have confirmed that their instances have been expanded, the RM issues a "*switch plan*" message to all the LDs, which then switch to the increased allocation.

The RM agent is also responsible for interfacing with any available operating-system mechanisms for partitioning the machine amongst customers. Such partitioning mechanisms (e.g., [ENSIM, WLM]) typically enforce strict limits on each application in terms of CPU usage, disk space, network bandwidth and other OS-level resources, and also provide secure isolation of applications from each other. However, even if such a mechanism is not available, the system's throttling-based mechanism (performed by LD) loosely enforces the allocation limits.

When all the RM agents have performed the requested reallocations, the RM commits the new allocation plan to the Configuration Repository.

Sub-farm Creation and Removal: When a customer requests the use of a resource class that isn't currently available on the server farm, a new sub-farm must be created automatically to execute instances of that resource class. The RM uses the Configuration Repository to identify the appropriate unused machines, designates them as part of a new sub-farm, and sets up a Load Distributor to manage the incoming requests. No resource instances are actually created at this stage. The instances get created as outlined earlier, when the GDM allocates non-zero hit-rates to them.

A sub-farm may be removed if no customers require the resource class installed on it. The RM uses its agents to shut down all resource instances running on that sub-farm, as well as its Load Distributor. Once the sub-farm is made inactive, it updates the status of each machine in the CR, to indicate that it is part of the free pool.

5. Load Distributor

There is a Load Distributor (**LD**) associated with each sub-farm, which distributes load among the instances assigned to a customer. An LD has a queue for incoming requests, and data structures to store the allocated hits per instance per customer and the amount of capacity consumed during the current time interval. It removes a request at the head of the queue, identifies the customer C for which the request is meant, and finds the instance **p** assigned to C that has the least load factor. If the allocated capacity of **p** is fully consumed, the request is throttled, else it is forwarded to **p**. Similar load balancing techniques are described in [CCY99b, HGKM98].

LD receives load and fault information about an instance from the aggregator and new resource allocations from the RM agents. Any reduction in allocation for a customer is taken into account immediately. An increase in allocated capacity, however, is taken into account only after the corresponding "*switch plan*" message is received, as described earlier.

6. Configuration Repository

The Configuration Repository acts as a central storage for the server farm. It stores information on hardware availability, software dependency and requirements, machine capacities, the allocation table, application benchmark characteristics, SLAs, etc.

IV. SYSTEM INITIALIZATION, SCHEDULING, AND FAULT-HANDLING

System Initialization: At startup time, the machines designated to run the system management components (Global Decision Maker, Aggregators, Load Monitors, Resource Manager, Load Distributors) are booted up with predefined scripts that start and initialize the respective components. All applications are installed on a distributed file system, and a uniform view of the file system is provided on all the machines.

Each new customer who joins the server farm is first hosted on an experimental setup to benchmark his applications, if this information is not already available. This information is then added to the CR. Next, the sub-farms are configured for the customer. Expected resource demands are fed to the GDM, in order to generate an allocation plan. The Resource Manager then deploys the GDM's plan.

Scheduling: The system uses a simple schedule having a monitoring sweep, followed by an aggregation sweep, a GDM plan computation, and finally a plan deployment by the resource manager. After this the cycle repeats. Since plan computation and deployment are relatively slow processes, a cycle like this has a relatively large time period. Monitoring and aggregation for feeding back the load information to Load Distributors require a relatively smaller time period. To cater to this, multiple cycles of feedback to Load Distributors are executed per cycle involving the GDM. Thus, the majority of events in monitor output are oriented towards feedback to Load Distributors with only a small minority guiding re-computation of the allocation plan.

Handling Faults: Monitors pass a discovered fault to aggregators for further action. The aggregators post a notice to the system administrator and inform the affected Load Distributors to stop directing hits to the faulty

resource. It is then up to the LDs to distribute the hits to alternative resources, or throttle some hits if they cannot be handled. The aggregators also pass fault data to the GDM so that the GDM can remove the faulty resource from its allocation plans. Prior to preparing an allocation plan, the GDM collects the latest status of the free pool of machines (there may be new or newly-fixed machines added to the pool which can be used) and takes into account all faulty machines that have to be removed.

V. RELATED WORK

During the 1990s, research in cluster computing focused mainly on providing computing environments with high availability and scalability needs [BSSDRP95, FGCBG97]. A survey and classification of server clustering techniques can be found in [SGR00]. More recently, research in this area has been directed towards using clusters for web-based applications [AYI97, RA99, SLR98]. Systems such as [RA99] aim to improve scalability through mirroring and migration whereas [SLR98] and others provide fault tolerance. The Flex system [C99] performs locality-aware load balancing for server farms to improve performance, but applies only to static web sites. All these systems focus more on scalability and reliability issues rather than on resource optimization and mostly deal with static web content.

Mounties [FJNRV00] is similar to the presented work in that it focuses on the management of software and hardware resources by developing plans using a centralized optimizer, with an infrastructure for deploying the plans. The notion of a virtual server corresponds to the notion of resource groups in Mounties. However, in contrast to Mounties, a virtual server is generally distributed over multiple machines as opposed to residing on one machine. Finally, the present work discusses server farms, including issues like directing/throttling incoming load, which is not the subject of Mounties.

The MultiSpace [GWB99] system is similar to ours in that it provides a mechanism for providing transparently scalable services to clients. It however requires the use of a client-side stub to perform dynamic load redirection, and is only useful for freshly developed Java applications.

[CCY99a] describes a technique for sharing load among replicated web servers. It relies on the static redirection mechanisms built into DNS and HTTP. In our system in contrast, we perform load redirection at several levels (one per sub-farm), and the redirection is controlled by the latest available load information.

Darwin [CFKNSTZ98] provides resource management mechanisms for value-added network services. The network-centric view of Darwin percolates through the kind of flow graphs used to specify resource requests -nodes are services, edges are communication flows. Resource availability itself is cast as a discovery problem in the network. In contrast, our work is server-farm centric with a pool of local, reusable resources for fast, dynamic allocation. This causes us to solve different kinds of optimization problems. For example, optimization of network communication among resources, or in choosing among resources is not our focus. This also causes our optimization plans and the plan-implementing infrastructure to differ from the work in Darwin.

VI. CONCLUSION

Escalating costs of IT infrastructure, its management and ever changing technologies make it imperative to exploit economies of scale by sharing IT resources. In this paper, we have proposed an architecture that addresses this need by automated sharing of hardware and software resources at the intra and inter-machine level. The sharing is carried out using global and local optimization/decision-making. The architecture does the decision-making and deployment automatically in compliance with pre-defined SLAs that are expressive, yet simple and user friendly. The architecture comprises a notion of virtual servers that are a set of distributed software resources communicating with each other. A preliminary prototype of the architecture has been implemented and further work is planned.

ACKNOWLEDGEMENTS: We would like to thank Vikram Singh Beniwal, Abhay Chrungoo, Vishu Gupta, Abhinav Roongta, and Saurabh Sood for their work in building a preliminary prototype of the proposed architecture. We also thank Dr. Sugata Ghosal for his meticulous and insightful comments on the paper.

REFERENCES

[AFF01] Appleby, Fakhouri, Fong et al. *Océano – SLA Based Management of a Computing Utility*. To appear in Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management.

[AYI97] Daniel Anderson, Tao Yang, and Oscar H Ibarra. *Towards a Scalable Distributed WWW Server on Workstation Clusters.* Journal of Parallel and Distributed Computing (JPDC), September 1997.

[BHM77] S.P.Bradley, A.C.Hax and T.L.Magnanti. *Applied Mathematical Programming*, Addison-Wesley, 1977, pp.379-380.

[BSSDRP95] Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. *Beowulf:A Parallel Workstation for Scientific Computation*. In Proceedings of International Conference on Parallel Processing, 1995.

[C99] Ludmila Cherkasova. *FLEX: Design and Management Strategy for Scalable Web Hosting Service.* Technical Report HPL-1999-64R1, HP Labs, 1999.

[CCY99a] Valeria Cardellini, Michele Colajanni, Philip S. Yu. *Redirection Algorithms for Load Sharing in Distributed Web-server Systems*. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS99), pp. 528-535, May / June 1999.

[CCY99b] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. *Dynamic Load Balancing on Web-server Systems.* IEEE Internet Computing, May-June 1999.

[CFKNSTZ98] P. Chandra, A. Fisher, C. Kosak, E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, *Darwin: Customizable Resource Management for Value-Added Network Services*. In Proceedings of 6th International Conference on Network Protocols, pp. 177-188, Oct. 1998.

[ENSIM] Ensim Corp. *ServerXchange white paper*, http://*www.ensim.com/pdf/wp.pdf*

[FJNRV00] Sameh A. Fakhouri, William F. Jerome, Vijay K. Naik, Ajay Raina, and Pradeep Varma. *Active Middleware Services in a Decision Support System for Managing Highly Available Distributed Resources*. In LNCS #1795, Middleware 2000, New York, NY, USA, April 2000.

[FGCBG97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. *Cluster Based Scalable Network Services*. In Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, October 1997.

[GWB99] Steven D. Gribble, Matt Welsh, Eric A. Brewer, David Culler. *The MultiSpace: An Evolutionary Platform for Infrastructural Services*. In Proceedings of the USENIX Annual Technical Conference, Monterey, USA. June 6-9, 1999.

[HGKM98] G. Hunt, G. Goldszmidt, R. King and R. Mukherjee. *Network Dispatcher: A Connection Router for Scalable Internet Services.* In Proceedings of the 7th International World Wide Web Conference, 1998.

[IBMSP] IBM Corp., *RS/6000 SP Monitoring: Keeping It Alive*, IBM Publication# SG24-4873, 1997.

[JMX] Java Management Extensions (JMX), http://java.sun.com/products/JavaManagement/, 2000.

[RA99] Michael Rabinovich and Amit Aggarwal. *RaDaR: A Scalable Architecture for a Global Web Hosting Service*. In Proceedings of the Eighth International World Wide Web Conference, 1999.

[RNC96] H. Raghav Rao, Kichen Nam and A. Chaudhary. *Information Systems Outsourcing*, Special section in Communications of the ACM, 39(7), July 1996.

[SGR00] Trevor Schroeder, Steve Goddard, Byrav Ramamurthy. *Scalable Web Server Clustering Technologies*. IEEE Network, pp 38-45, May-June 2000.

[SLR98] Ashish Singhai, Swee-Boon Lim and Sanjay R. Radia. *The SunSCALR Framework for Internet Servers*. In Proceedings of the IEEE 28th International Symposium on Fault Tolerant Computing, 1998.

[WLM] AIX 5L Workload Manager (WLM), http://www.redbooks.ibm.com/redbooks/SG245977.html

APPENDIX A: Optimization Model

Inputs: from aggregators: (1) Resource demands, in terms of reference hits (2) Correction factor demands. Inputs from configuration repository: (1) Correspondence ratios (2) Maximum allowable number of instances of each resource class for a customer (3) Cost of starting a new instance and shutting down an existing one (4) machines available to each resource class, (5) machine capacities with respect to a resource class (computed after taking into account O/S and other overheads), and (6) limits on the number of instances allowed on each machine (7) Current allocation plan (8) Customer SLA information. The GDM computes the allocation plan, while aiming to maximize ASP's revenues, using a mixed integer linear programming model of the system.

Problem formulation: The subscripts i, j, k vary from 1 to N_i, N_j, N_k, respectively, where N_i is the total number of customers, N_j is the total number of resource classes, and N_k is the total number of machines.

$\mathbf{D_{ij}}$ denotes the total demand by customer i for resource class j. It is the sum of demands from Level-2a and Level-2b aggregators in reference hit units. D_{ij} may be negative. $\mathbf{A_{ijk}}$ denotes the current allocations, in terms of reference hits, for customer i on resource class j, machine k. We solve for the final allocations $\mathbf{B_{ijk}}$.

$\mathbf{P_{ijk}}$ denotes the sum charged per unit of resource B_{ijk} allotted to a customer i, resource class j and machine k combination. The value of P_{ijk} conforms to the customer's SLA. $\mathbf{T_{ijk}}$ represents the system costs (second term, equation (1)) of adding (setup cost) or removing a customer ('cleaning' cost). $\mathbf{U_{ij}}$ denotes the system cost associated with having multiple instances of a resource class for a customer (third term, equation (1)). Thus, the plan computation problem can be formulated as follows.

(1) $$\textbf{Maximize } \Sigma_i\Sigma_j\Sigma_k (P_{ij}*B_{ijk}) - \Sigma_i\Sigma_j\Sigma_k T_{ijk}*|Y_{ijk}-Z_{ijk}| - \Sigma_i\Sigma_j(U_{ij}*\Sigma_k Z_{ijk}).$$

Here $\mathbf{Y_{ijk}}$ **in {0,1}** indicates if $A_{ijk}>0$, and $\mathbf{Z_{ijk}}$ **in {0,1}** indicates if $B_{ijk}>0$ (i.e. it indicates allocation of instance on machine). Let $\mathbf{S_{jk}}$ denote the capacity of machine k for resource class j. Then the following equations assign values to Y_{ijk} and Z_{ijk}. Only the second is a constraint:

$$\mathbf{Y_{ijk} >= A_{ijk}/S_{jk},}$$

$$\mathbf{Z_{ijk} >= B_{ijk}/S_{jk}.}$$

Final allocation is bounded by the sum of capacity demanded and the existing allocation. Also, in case the system does not have sufficient capacity, this allows dropping of part of the demand due to customers with low priority. The demand constraint is:

(2) $$\Sigma_k B_{ijk} <= \Sigma_k A_{ijk} + D_{ij}, \quad \textbf{for all i, j.}$$

The supply constraint is modeled as follows. Each machine k can be allocated no more than its capacity for the j^{th} resource class, S_{jk}. The variable $\mathbf{R_{jk}}$ **in {0,1}** indicates if resource class j exists on machine k.

(3) $$\Sigma_i B_{ijk} <= S_{jk} * R_{jk}, \quad \textbf{for all j, k.}$$

Only one resource class is assigned to each machine.

(4) $$\Sigma_j R_{jk} <= 1, \quad \textbf{for all k.}$$

There is a limit $\mathbf{L_{jk}}$ on the number of resource instances on each machine. This limits the overheads arising out of supporting a large number of instances on a single machine.

(5) $\Sigma_i Z_{ijk} <= L_{jk}$, **for all j, k.**

There is a limit M_{ij} on the number of instances of a given resource class, for each customer. This is because it gets harder to manage multiple instances as the number of machines over which a customer is spread increases.

(6) $\Sigma_k Z_{ijk} <= M_{ij}$, **for all i, j.**

The next constraint ensures that whenever requirements from different resource classes are dependent on each other (e.g., where one resource class calls another), the amounts are allocated in proportion (see equation (7)). This is because, if an application cannot be granted its demands for a particular resource class then it is meaningless to reserve corresponding resources for the other resource classes of that application. Although a linear model is not an accurate one, it is an approximation of the actual dependence between the two resources. To ensure that the final allocations conform to the correspondence ratios $C_{jj'}$, we have

(7) $\Sigma_k B_{ij'k} = C_{jj'} * \Sigma_k B_{ijk}$ **for all dependencies j, j'.**

Each customer must be allocated a minimum requirement according to the SLA promised to it.

(8) $\Sigma_k B_{ijk} >= Min_{ij}$, **for all i, j.**

We try to ensure that the supply of machines at any time is enough to ensure feasibility of constraint (8). However, in some cases such as a major system fault, (8) may not be feasible. Thus, the above problem is first solved for the following minimum in equation (8):

$$Min_{ij} = \text{maximum (minimum specified by the SLA, } (\Sigma_k A_{ijk} + \Sigma_k V_{ijk})),$$

where, $(\Sigma_k A_{ijk} + \Sigma_k V_{ijk})$ represents the sum of the current allocation and its corresponding hit-weight demand V_{ijk} which is also specified in terms of reference hits. This option supports a higher quality service to all customers since existing allocations in the system would be upheld**.** In case this is infeasible, it is replaced by the original equation (8). If that is also infeasible, there is a penalty because the system cannot handle the customer's minimum requirements. In this case, this equation (8) is dropped and the model solved again without it, and with the penalties instead. There is a standard model to incorporate penalties in linear programming, which is being omitted for ease of presentation (refer [BHM77]).