RC22839 (W0307-031) July 3, 2003
Computer Science

IBM Research Report

Adaptive Resource Management and Workload Scheduling for a Peer Grid

Vijay K. Naik, Swaminathan Sivasubramanian, Sriram Krishnan
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Adaptive Resource Management and Workload Scheduling for a Peer Grid

Vijay K. Naik[*] Swaminathan Sivasubramanian[†] Sriram Krishnan[‡]

Abstract: Business applications are typically run on dedicated servers belonging to a single administrative domain. The advent of the Open Grid Services Architecture (OGSA) has opened new avenues for executing business workloads on a platform consisting of non-dedicated resources from multiple administrative domains, using the Grid paradigm. However, the popular implementations of OGSA do not adequately address the needs of business and commercial applications. In this paper, we discuss the key requirements imposed by transactional business applications and how these requirements affect the underlying grid architecture. Instead of using resources that are dedicated for a single type of grid computations, we consider a grid architecture that pools together resources that may be shared according to individual local policies. We discuss the logical architecture of a such a grid with particular emphasis on transparent resource management and workload scheduling. We also discuss some of the design choices we have made and present performance results to show the effects of transient resources (because of policy-based sharing) on the throughput delivered to grid workload.

1. Introduction

The Open Grid Services Architecture (OGSA) defines a uniform service semantics (the *Grid service*) and standardized basic mechanisms required for creating and composing distributed computing systems in terms of web services related concepts [6]. With this service oriented approach of OGSA, Grid computing has become an attractive platform for deploying business applications and for supporting commercial workload. Standardized Web-based protocols such as HTTP for transport and Web services for application life-cycle management provide easy to adopt programming models for application developers without sacrificing the scalability offered by the Web. OGSA makes it possible to manage and administer large scale systems, in a standardized manner, across multiple administrative domains. Using the Grid architecture it is possible to associate individual policies with each participating Grid resource and these policies can determine the manner in which a resource is be shared by grid and non-grid workload.

This opens the door to the possibility of developing a distributed computing platform for delivering business services with certain level of guarantees using inexpensive but large number of underutilized resources such as the desktop systems. Over the past few years, desktop systems have become more powerful than mid-size systems of the yesteryear. The operating systems and middleware technologies make it possible to harness their inherent capabilities. Thus, in principle, the desktop systems in a corporate intranet form an ideal set of resources for off-loading peak demands on backend servers, for testing and deploying new releases of backend applications, or for improving availability and responsiveness of existing mission critical backend infrastructure.

However, harnessing these resources and applying them in an aggregate manner for supporting business processes is a hard problem. The prob-

[*]Correpsonding author, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598. *vkn@us.ibm.com*

[†]Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands. *swami@cs.vu.nl*

[‡]Dept of Computer Science, Indiana University, Bloomington, IN 47405.*srikrish@cs.indiana.edu*

lems are primarily related to the conflicting requirements placed by the desktop users and the users of the business applications (i.e., the Grid users). For the desktop users, interactivity, responsiveness, and security are of prime concern. From the point of view of the Grid client, discovery, responsiveness, and security are of prime concern. While the desktop users want the entire system at their disposal when they want to use it, for the Grid clients it is cumbersome to go hunting for idle resources, suitable for their computations, over the intranet. Moreover, existing grid enabling toolkits such as Globus (versions 2.x and 3.0) do not adequately address the business application requirements, which we explain in more detail in Section 2. Similarly, the job and resource management tools commonly used by the scientific community do use the same performance metric as those demanded by the commercial workload.

In [9], we have defined an architecture that extends the service-oriented architecture of OGSA to address the specific requirements discussed in this paper. In that paper, we presented a three layer architecture consisting of Grid Service Layer, Logical Resource Layer, and Physical Resource Layer, with a focus Physical Resource Layer. The Physical Resource Layer has the necessary mechanisms for monitoring and predicting the future interactive workload.

In this paper, we focus on the Grid Service Layer of the architecture and describe in detail the manner in which architecture is laid out and the system is designed so transactional Grid services can be scheduled and run in a predictable manner while using unpredictable set of desktop resources. Using the services provide by this layer, Grid services can adapt to the changing set of resources and provide uninterrupted service to Grid clients. We discuss the design of this layer and present performance data to show that the system can efficiently utilize available cycles by deploying Grid services on idle desktop systems just-in-time to satisfy the demand from Grid clients.

We refer to the grid defined by our architecture as peer grid because, using this architecture, it is possible to develop a grid entirely on top of resources at the edge of the network, such the desktop systems. No dedicated resources are needed even for supporting management services.

The rest of the paper is organized as follows. In the next section, we describe the requirements placed on the architecture by the transactional business oriented workload and also the requirements arising from the use of shared resources such as desktop systems. We then describe the highlights of the architecture with emphasis on the Grid Service Layer. In Section 3, we discuss the design and implementation of the Gateway that acts as the coordinator between the deployed Grid services and the Grid clients. Performance results from an implementation of our architecture are described in Section 4. In Section 5, we discuss how our work relates to other work in the literature. Finally, we present our conclusions in Section 6.

2 An Approach for Developing a Peer Grid Using Desktop Resources

2.1 Transactional Business Services

As mentioned in the introductory section, this work is driven by two objectives:

i. To deploy and enable transactional business services as Grid services (e.g., financial, accounting, billing, e-commerce, customer relations, or supply-chain management related transactions).

ii. To use desktop based resources to provision such transactional services.

Both of these objectives give rise to design parameters that differ significantly from those that are common to traditional Grid designs that cater to scientific and engineering applications. In the following paragraphs, we highlight the requirements posed by the transactional business applications and services and contrast these against the requirements posed by the scientific and engineering applications.

Transactional business services exhibit *high degree of interactivity* with human operators and/or with databases that hold business state information. The time spent interacting with the external environment is typically comparable to the time spent in performing local computations. Moreover, the frequency of interactions with the external environment is relatively high. On the other hand, typical scientific and engineering applications start with a state encapsulated in a small number of static files or other objects and evolve that state over a period of time and/or space. Such computations can continue in batch mode without significant interactions with a database or with a human operator. The time spent in batch mode can be order of magnitude higher than the time spent interacting with the external environment.

One effect of the interactivity is that business services need to be much more sensitive to *response time constraints* posed by the users. This is not only because of the human factors involved (e.g., on-line shoppers may not have patience for long response time delays), but also because of the role played by these applications in time sensitive business processes. In such cases, any processing delays can result in financial losses and/or competitive disadvantages. Typical response times are of the order of seconds or minutes.

The flip side of response time is the throughput, which is a measure of the number of transactions performed per unit time. Although many types of business interactions tend to be bursty (i.e., low activity followed by sudden rise in the demand, which is again followed by weak demand), they also require that the service throughput should rise with demand, without deteriorating the response time. Unless enough resources are allocated at all times to handled the peak demand, this means that necessary resources must be allocated dynamically and on demand. Moreover, the resource management mechanisms must be sensitive in the changes in the workload and must respond rapidly so the response time and overall throughput do not deteriorate.

Finally, many of the business processes are mission critical. This means the business services and the state information they process, must be available to corporate customers at all times – 24 hours a day and 7 days a week. This results *in the high availability* requirements on the on-line business services. At the minimum, services need to recover gracefully from failures and user data is not to be lost.

In contrast, typical scientific and engineering applications have low response time requirements and no availability requirements to speak of, but they do have reliability and service time requirements. This means users of such applications, who many times are also the application developers, are more flexible about the turnaround time as long as their applications run to completion in a reliable manner. The job arrival patterns are much less bursty and demand on resources fluctuates within a narrow range. Because of these characteristics, Grid systems catering to users of scientific applications emphasize services related to reservation mechanisms, job queuing, launching, checkpointing, migration, file transfers, and so on.

In short, typical Grid systems used for scientific computing workloads provide services that focus on maintaining high utilization of Grid resources. But such systems provide inadequate or no support for response time guarantees, continuous availability of applications, work-flow type of application setup, or for dynamic provisioning of resources in response to changes in the request arrival patterns. As pointed out above, such services are important in the context of transactional business applications. To provide these services, the Grid architecture needs to provide monitoring mechanisms to evaluate the rate at which different types of requests are processed, analytic capabilities to determine if these processing rates are adequate, prediction capabilities to anticipate future demand and resources needed to satisfy the demand. Such a Grid architecture also needs to provide support for deploying services that can persist and remain available even if the underlying resources become unavailable for some reason.

In Section 2.3 we describe a Grid architecture that responds to response time and availability requirements.

2.2 Desktop-based Resources

The second objective of our work is to utilize unused desktop-based resources and, more generally, to effectively share resources across multiple domains.

The primary objective of desktop systems is to provide the desired level of interactivity and to create an environment that is conducive to high-levels of productivity in a collaborative environment. The nature of the desktop-based interactive applications is such that the demand on the desktop resources occurs in frequent, but short bursts and the load dissipates rapidly.

Thus, there are many unused cycles, but their frequency and duration are highly unpredictable. Moreover, the desktop users (or administrators) may set policies that enforce conditions under which desktop resources may be used for deploying Grid services.

Clearly, for effective utilization of desktop systems in a Grid, one needs to take into account the following requirements: (i) Utilize the desktop system whenever conditions allow it to be used in Grid computations, and (ii) not to schedule any computations on a desktop system, beyond its available capacity.

The first requirement implies that a mechanism is needed to accurately predict when a desktop system becomes available for Grid computations. The second requirement implies that the Grid workload assigned to a desktop-based Grid node should match the available capacity. Note that desktop policies may not allow full utilization of the maximum available capacity of a desktop system. For example, a policy may specify the maximum fraction of the CPU, memory, and network bandwidth that a particular Grid service may use at any given time.

Clearly, the desktop resource availability and capability is more predictable when the desktop user is away from the system (e.g., in the evening and night hours). This information can be gathered and analyzed by running a monitoring agent on the desktop to understand the daily, weekly, monthly and seasonal patterns in "macro" usage of the system. Even when the desktop system is being used by the desktop user, there are many opportunities for running Grid workload on the system under the specified policies. However, because of the unpredictable nature of the interactive usage, only short term predictions about the future usage by interactive applications can be made with a given level of confidence.

Thus, to effectively utilize desktop-based resources, the Grid architecture needs to provide support for monitoring desktop resource usage patterns, both for the interactive workload as well as for the Grid workload. It needs to incorporate analytical mechanisms to predict resource availability and capabilities at various time intervals in the future. Furthermore, the system needs to be able to bound the uncertainties in the predictions. We now describe our architecture that takes into account these requirements.

2.3 Architecture Overview

Availability and responsiveness to the changes in the client demands are the key criteria that a transactional service provider must meet. The primary figure-of-merit (i.e., expected QoS) for such services is throughput and response time. This means the architecture should be able to deliver a requested service on demand from the clients and it should be able to adjust the capacity of each service so as to meet the intensity of the demand. The client requests can be complex (e.g., requests resulting in a work-flow), request arrival rates can be unpredictable, and clients may have multiple levels of service-level-agreements (SLA) with the service provider. The architecture needs to address these requirements.

The use of desktop based resources gives rise to a different set of requirements. The primary purpose of desktops is to serve the desktop users by providing a high degree of interactivity and responsiveness. These resources are to be used to provision the transactional services according to some policy defined by the desktop user or by system administrators. Each desktop may have a unique local policy, which may change over time. Examples of local desktop policies include: (i) interactive workload always has the highest priority, (ii) allocate no more than a certain percent of the desktop resources to Grid services at any given time, (iii) dedicate certain fraction of the resources for Grid computations, (iv) allow participation in the Grid computations only during certain time of the day or on certain days of the week. Thus, policy enforcement requires evaluation of certain conditions, which may be static and predictable or dynamic and unpredictable such as the current interactive workload. Moreover, policies may be defined using a combination of static and dynamic conditions. The architecture needs to take into account policies and the heterogeneity in the capacities associated with each desktop resource while addressing the availability, throughput, and responsiveness requirements associated with the transactional services.

Intuitively, the desired architecture needs to facilitate (i) deployment of appropriate Grid services on the desktop resources, and (ii) route client requests to appropriate Grid service instances. These tasks are made challenging because of (i) the uncertainties in the resource availability for deploying a Grid service at any given instance in time and (ii) the uncertainties in the client demand on a Grid service at any instance in time. If we assume that there are enough idle desktop-based resources available to meet demand at any given time, then the task of the architecture is (i) to identify and match Grid client requests with Grid service instances with appropriate capacity (i.e., with ability to respond within prescribed time limits), and (ii) to deploy Grid service instances on appropriate desktop resources so as to empower them with the desired capacity just-in-time for delivering the service.

Although there are enough desktop resources available on an aggregate basis, identifying the once that can provide the desired capabilities at any given time requires good prediction mechanisms. Since there is an element of randomness in the demand and in the resource availability, it is also important to quantify the uncertainties in the predictions. One way to achieve this is by monitoring the behavior of each desktop resource over a sufficiently large time interval and then by comparing the observed behavior with behavior predicted by the prediction model (after suitably priming the predictor with initial conditions). The extent of the mismatch between the two is a measure of the quality of the predictor and hence can be used to determine the uncertainty in the predicted values by the predictor for that system. Since these systems are dynamic, uncertainties in the prediction models need to be evaluated continuously.

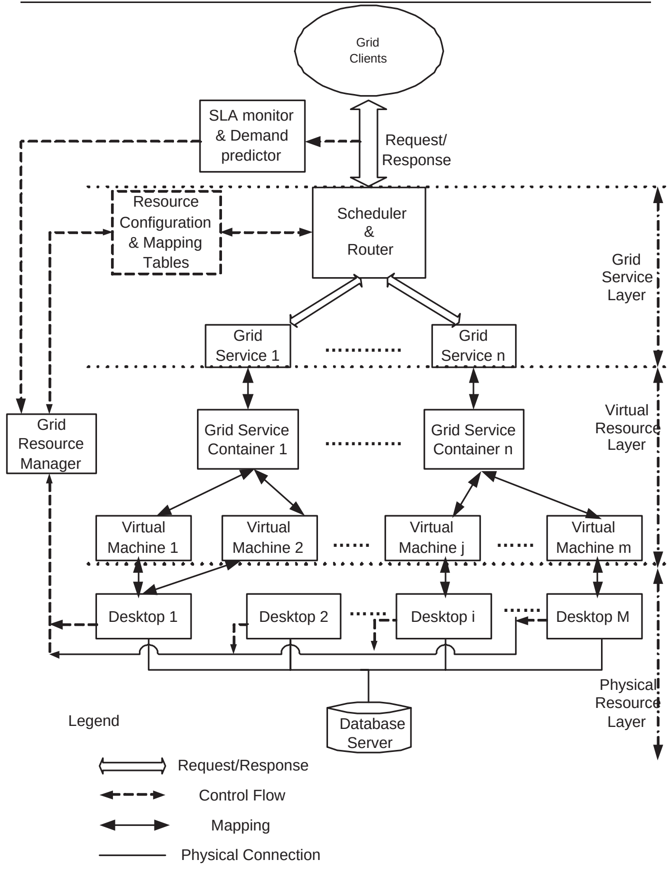

Figure 1. Layered architecture for the Peer Grid. The Grid Service Layer consists of the Grid Services, Scheduling and Routing Services. Logical Resource Layer consists of the Container for Web Services, Virtual Machines, and the Virtual Machine Managers. Physical Resource Layer consists of the physical resources and the Host Agents.

Given the ability to monitor, predict, and estimate the uncertainties in the predictions, the architecture is basically reduced to scheduling appropriate number of service instances, mapping the service instances on to the physical resources, and routing client requests to appropriate service instances. The rest of this section gives an overview of our architecture and briefly describes the orchestration of monitoring analysis & predictions, allocation, mapping, scheduling operations described above.

The architecture is defined using a layered approach. This allows addressing the requirements of Grid workload and of transactional Grid services separately from the requirements of interactive workload and desktop related policies. The architecture, as shown in Figure 1, has three layers: (i) The Grid Service Layer, (ii) The Logical Resource Layer, and (iii) The Physical Resource Layer. In the following, we describe the salient features and functionality of each layer. For details on the three layers of the architecture, please refer to [9].

Each layer is associated with Control and Management Components (CMCs). The interactions among the CMCs and the functionality they provide largely define the architecture. The SLA Monitor and Demand Predictor, shown in Figure 1 is one such CMC. This component monitors request arrivals per Grid service type and per Grid client class basis. It also monitors SLA violations on a per client basis. In addition, predictions on future arrival rates are made for each Grid service type. Based on the predicted arrivals and available Grid service capabilities, a scheduling strategy for request processing is adopted to meet the SLA requirements. This process is repeated frequently as arrival patterns change and/or as the Grid service capabilities change. Some of examples of scheduling strategies are weighted round robin, priority based scheduling (with priorities derived from SLAs), one-to-many scheduling (i.e., simultaneous processing of a request on multiple Grid service instance to overcome uncertainties in service capabilities), and so on.

The CMCs in the Physical Resource Layer enforce desktop related policies, monitor and analyze the interactive workload, and predict the short range availability and capability of the desktop system for a particular Grid service. This are described in more detail in [9]. The CMCs in the Logical Resource Layer act as coordinators between the Grid Service Layer and Physical Resource Layer.

The Grid Resource Manager (GRM) shown in Figure 1 acts as a facilitator across all three layers. The main function provided by GRM is to discover desktop resources that are available and capable of deploying one or more Grid services. It also detects when a desktop resource is no longer available for deploying Grid services. A second key function provided by GRM is to allocate the predicted capacities of each participating desktop resource to Grid services requiring the resource during that future time interval. It tries to locate and allocate as many resources to each Grid service as possible making sure that the conflicts caused by sharing are minimized. To perform this task, GRM collects from each desktop the desktop usage and policy related data and predicted availabilities from the CMC (known as Host Agent) running on that desktop. It normalizes the raw capacity of the desktop against a standard platform. In case the desktop node is to be shared among multiple Grid services, it further reduces the available capacity in proportion to the share made available for other Grid services. This represents the maximum normalized capacity available to a particular Grid service. It then takes into account the predicted available capacity as a fraction of the total capacity and uses that to compute the predicted available capacity from a desktop resource for each Grid service. This forms the predicted allocation of desktop resources to predicted Grid services requiring resources. It also computes the uncertainty in each prediction and makes this information available to the Scheduler & Router component of the Grid Service Layer. This information is represented by the Resource Configuration & Mapping Tables shown in Figure 1.

The Scheduler uses this information to determine the number of service instances to deploy for each Grid service for which it anticipates demand. The number of instances deployed is proportional to the allocated capacity and to the expected demand. When requests arrive, the Router routes those requests to the physical resources where the service instance is actually deployed.

The Scheduler also takes into account the uncertainty in the predicted allocations. When the uncertainty is high, it may decide to schedule a request on more than one service instance simultaneously, making sure that the the service instances are mapped on-to distinct physical resources. In such cases, the Router replicates a request and multicasts it to multiple instances of the same Grid service.

In the above, for simplicity we have described the architecture with one Grid system to serve all the Grid clients. However, the architecture described here is more general than that. In its generality, when a class of Grid services are to be deployed in anticipation of a particular Grid workload, a Scheduler & Router object is spawned off. This can be accomplished using a Grid service factory method described in [6]. As described above, the Scheduler is provided with a list of Grid Services for which requests are expected and a set of Mapping Tables are provided. In creating the Mapping Tables, sharing of physical resources by other virtual Grid environments and desktop users is taken into account. Based on the priorities and policies, GRM calculates the availability and capability factors for each Grid environment separately. It also assigns probabilities to indicate the uncertainties in its performance predictions. Note that the same physical resource may participate in two different different virtual grid organizations and it may be assigned different capability and uncertainty factors in the two virtual grid organizations.

3 Gateway Architecture and Design

In this section, we discuss the functionalities of our Scheduler & Router component. We also discuss in detail the choices available for designing this component and present their relative merits and demerits. Finally, we describe the prototype implementation of our design.

3.1 Gateway Requirements

An important component of our architecture is the *request scheduler* responsible for scheduling request requests from Grid clients, based on the relative priorities of the requests and their SLA requirements. Another essential component is the *request router*, which routes the client request to the Grid node that actually performs the transaction. It also maintains the status of various requests. This component works closely with the scheduler and monitors the status of execution of each request. It is capable of restarting a request, if a request fails (possibly due to a Grid node failure, network failure etc.) to ensure that the Grid clients get the guaranteed QoS.

In our system, we view the above two components that perform these two functionalities (scheduling and Router) as a single logical component and refer to it as the *Gateway*. Gateway is the entry point for the Grid clients, to which they submit their service requests. It can be viewed as a logically centralized component. However, if required (for reasons of scalability), it can be implemented as a federation of gateways, as described in section 3.2.2.

In a peer Grid using desktop resources, in addition to scheduling and request tracking, Gateway must be capable of handling the variability in resource availability and smoothen it so that Grid clients do not see the effects of variability. In our system, this is handled by the Grid Resource Manager, which co-ordinates with various CMCs of physical and Grid layers, and smoothens the variability in the resource availability of the desktops. As noted before, the process of handling this variability can be done in two ways: (i) predicting the resource availability in a Grid node and scheduling based on that prediction or (ii) schedule assuming they are available all the time and migrate the request, if they become unavailable during the course of execution of a request. In our system we adopt the first approach, as our Grid applications are transactional workloads, where the execution time is much smaller compared to those of the batch-computing applications. The overhead introduced in migrating an active transaction can be comparable to the service time of a transaction itself. However, the success of the first approach design relies on the accuracy of the prediction mechanisms.

3.2 Gateway Design and Implementation

In this subsection, we concentrate on the design of the Gateway and its interaction with GRM.

In our design, we make a clear demarkation of resource prediction models from on-line allocation and scheduling components. Such a demarkation is required for the following two reasons: (i) The availability of each Grid node is governed not only by its (interactive workload) usage pattern but also by the local policy set by the desktop user. Hence, resource prediction must be done separately taking these factors into account and (ii) Separation of resource prediction components from allocation and scheduling components allows the system to use different resource prediction algorithms without affecting other system behavior.

In our system, the scheduling and resource allocation components are also separated. Thus, the Gateway schedules the request onto logical resource pools and routes it to the actual Grid nodes based on the routing and mapping tables populated by the GRM. The GRM (resource allocator) is responsible for allocating the Grid nodes onto these logical resource pools such that the overall Grid throughput is maximized. The advantage of this approach is that the Gateway can schedule the calls oblivious of change in the constituents of the Grid nodes. However, as noted in the preceding section, the Gateway (through the SLA monitor) must communicate with GRM and inform its expected Grid service demand, to ensure that enough resources are allocated to each of Grid services serviced by it.

3.2.1 Design Choices

The design of Gateway can be done in several ways: For example, Gateway can be built using network-level redirector such as such as IBM Network Dispatcher [2]. The other way to build the Gateway is by modifying the application-level transaction scheduler. In the following, we discuss the relative merits and demerits of these two approaches:

- *Network-level solution:* Gateway Router can be built using network-level redirector. Then, logical resource pools are built as a cluster of servers managed by these redirector. The requests to the pool can be routed using network-level redirectors. GRM can allocate or deallocate nodes by invoking the appropriate APIs provided by the redirectors. A similar approach is used in the system proposed by [7]. The primary advantage of this approach is its performance. Since all the routing is done in network level, it does not suffer the overhead of call marshaling and unmarshaling. However, it has the following disadvantages: It assumes that all the servers are equally capable of serving all Grid services. If not, then it requires the use of one network-level redirector for each Grid service. Thus, this approach lacks flexibility in adding new Grid services dynamically. Further, network redirectors are built for server clusters and is not suitable for our desktop pools, where the maximum number of desktop nodes in a pool can be relatively high, with dynamic change in their availability.

- *Application-level solution:* Gateway can also be built using application level redirector. In this approach, the Gateway receives service requests from clients, unmarshals them and based on the type of service required, it schedules the routing table populated by the GRM for that service. GRM, by populating a per-service routing table, essentially creates a per-service resource pool, based on which the Gateway Router schedules the service requests. The primary advantage of this approach is that this design can support different type of Grid services, without any changes or addition of new hardware. Further, this design can possibly manage a larger resource pool. However, this approach introduces some processing overhead as it needs to marshal and unmarshal a service request.

In our system, we adopt the application-level solution as it is more flexible to add/remove more Grid services dynamically. We have also observed that the overhead introduced by processing the request at application-level is negligible compared to the overall service execution time.

3.2.2 Gateway Implementation

In our system, we have implemented GRM as a Web service and is deployed in IBM WebSphere Application Server [3]. Similarly, Gateway Router is also implemented and deployed as a Web service. The Grid clients make their Grid service calls as Web service calls to the Gateway Router. However, since the Grid clients make Web service calls as if the service is running on the gateway, we have to implement our router in such a way that forwarding of request from the Gateway to the Grid node is transparent to the client.

In WebSphere Application Server (versions 4.x and 5.x), every Web service call is being trapped by its appropriate RPCProvider, as defined in Apache SOAP [1]. This provider is responsible for locating the actual class and method that needs to be invoked to make a (Web-service based) transaction. In our system, we implemented a new provider (which is in conformance with regulations of Apache SOAP specifications) that similarly receives this request at the Gateway. However instead of finding a method to invoke, it makes a call to a *forward* method of Gateway Router Service. This method receives the call object and consults the routing table for that service request, and forwards requests to different Grid nodes on a weighted round robin fashion.

By implementing a new provider, we have made no changes to WebSphere or its SOAP implementation and have just added a new plug-in to support our new provider. Thus, Grid clients make service requests as normal Web service calls with no change in their code. The GRM Web service populates the routing tables of the Gateway Router by making a standard Web service call.

3.3 Design Scalability

The design described above assumes a centralized Gateway and a centralized GRM. Such a system will have scalability problems as the number of desktop nodes increases and/or the number of Grid clients rises. However, this scalability issue can be addressed in several different ways. In the following, we briefly describe some of these concepts.

3.3.1 Federated Gateways

One way to alleviate the Gateway congestion is to provide multiple Gateways, each responsible for serving a subset of Grid clients using a subset of Grid nodes. The problem to address here is that of load balancing among the Gateways. One possibility is to use DNS servers and another possibility is to use a network dispatcher type of mechanism in front of the Gateways. Both of these approaches suffer the shortcomings described above and in [4]. We now describe third approach which is more appropriate when Grid clients perform many transactions within a session. When a new session is to begin, a Grid client registers for that session with a single well known Grid Registry. As a part of the registration the client receives address to one of the multiple Gateways that is capable of serving the client requests. The Registry keeps track of the current load on multiple Gateways and randomizes new client requests among lightly loaded possible Gateways.

Similarly, GRM allocates Grid nodes among multiple Gateways by knowing the current load among the Gateways. If it detects that some of the Gateways are not able to keep up with their demand, then it readjusts the current allocations among the Gateways and resets the Mapping Tables provided to each Scheduler & Router.

3.3.2 Hierarchical Control Structure

Another potential source of bottleneck in scaling up the system is the GRM and associated control structure. Here again the answer is to provide an hierarchy of GRMs. At the lowest level, each GRM looks after a manageable number of Grid nodes and then it forwards the allocation information to the GRM at the next higher level. The GRM at the top level has the consolidated information from all GRMs. This is then forwarded to the one or more Gateways in the system.

3.3.3 Databases

In case of commercial applications, client state is typically stored in backend database servers. This information may be accessed multiple times when a single transaction is being processed. Thus, in a large Grid system, a single backend database server can be a source of bottleneck. If the database is mostly used for retrieving information (e.g., content distribution or page serving), then the bottleneck problem can be alleviated by replication and periodic refresh. However, when transactions result in database update, the backend databases need to consistent with one another. While the database community has developed solutions to provide concurrent database systems, we admit that for a large scale system, the database subsystem may prove to be the true source of bottleneck for certain class of applications.

4 Performance Evaluation

4.1 Performance Modeling

We model the inherent variability and disparate capabilities of the resources that are part of our Peer Grid in the following manner. From our model, we try to infer the maximum throughput that is deliverable to a Grid client by our system, and compare it with our observations.

Assuming that a set of resources $0..m$ are available to be utilized, we associate a normalization factor, f_i, with each resource i. This factor qualifies the capabilities of a computing resource, and is the ratio of the capabilty of a particular resource with that of the best one available. Thus f_i varies in the set $(0..1]$.

We assume a set of types of requests from Grid clients $0..n$. For each request, we define the normalized service time, s_j, which is the time required by a Grid node with $f_i = 1$ to service a request of type j. In addition, we define the node service time, s_{ij}, as the time required by the node i to service a request of type j. It follows from the definition that $s_{ij} = s_j/f_i$. We note here that both s_j and s_{ij} are defined assuming that the nodes on which they are running are fully available for the Grid workload, without any timesharing or multitasking.

The availabilties of each resource are predicted at regular intervals, δt. This availability is a function of time (which varies from 0 to T). We define $p_i(t)$ as the fraction of the i^{th} resource available at time t. $p_i(t)$ varies from 0 (when the machine is not available to the Grid) to 1 (when the machine can be fully dedicated to the Grid workload).

If $a_i(t)$ is the actual fraction of resource i available at time t, and δa is the time interval between our observations of resource availabilty (note that δa need not necessarily be the same as δt), $A_{ij}(q)$, the maximum number of requests for service j that can potentially be processed by node i over time $0..q$ is equal to $\sum_{t=0}^{q/\delta a} a_i(t*\delta a)*f_i*\delta a/s_j$. The maximum number of requests that can potentially be processed by the Grid, $A_j(q)$, equals $\sum_{i=0}^{m} A_{ij}(q)$.

If $O_j(q)$ is the observed number of requests for service j that are processed in our implementation during time $0..q$, we can define the observed efficiency of our system, $o_j(q)$, as $O_j(q)/A_j(q)$.

It is worth noting that our model has a few limitations. In particular, it assumes no latencies between the Grid client and the Gateway, and between the Gateway and the Grid node. In addition, we neglect the scheduling overhead at the Gateway.

4.2 Experiment Setup

We tested the performance of our system on a small scale with a set of five Grid nodes. We logged the CPU utilizations of the interactive workloads of desktops used by the administrative personnel in our lab. These logs were used to simulate the interactive workloads on three of our Grid nodes. By doing this, we are able to simulate a real world situation where idle cycles can be used from desktops serving common users. These desktops

Node	Service Times (ms)
Node 1	914
Node 2	912
Node 3	1060
Node 4	1384
Node 5	1652

Table 1. Individual service times for each Grid node

Node	Availability	Prediction Accuracy
Node 1	100%	-
Node 2	100%	-
Node 3	99%	90%
Node 4	99%	89%
Node 5	99%	93%

Table 2. Average Availability of each Grid node and the Prediction Accuracy

will typically be highly available for Grid users as compared to the ones serving as development and production machines. We assumed two of our Grid nodes to be available all the time.

From our experiments, we compare the observed throughput ($O_j(q)$) with the maximum available throughput ($A_j(q)$) and determine the efficiency of our system. This efficiency depends on the accuracy of our predictions, and the associated overheads (as noted in the preceding subsection). Also, we verify that our predictions are reasonably accurate for the type of workloads we used in our experiments.

In order to measure the maximum observed throughput, we had to generate enough requests to keep the Grid nodes busy at all times they were available. To do so, we created a traffic generator that would generate a request as soon as it would receive a response to its prior call. In addition, this traffic generator is multi-threaded ensuring that multiple requests can be made in parallel, in order to keep all the Grid nodes busy at all available times.

4.3 Performance Analysis

The individual service times for our transaction on different Grid nodes is as shown in Table 1. We calculate the normalization factors f_i for each of the Grid nodes from the individual service times. We computed the actual availabilities of the individual Grid nodes, $a_i(t)$, from the utilization logs used for simulating the interactive workload on the desktops and is given in 2. As seen in the table, the average availability of the three non-dedicated Grid nodes is close to 100%. This is because the CPU utilization of interactive workload is bursty in nature and lasts for a short period, thereby providing a high *average* availability. For example, the actual availability of Node 3 can be seen in Figure 2, and its bursty usage pattern is apparent from it.

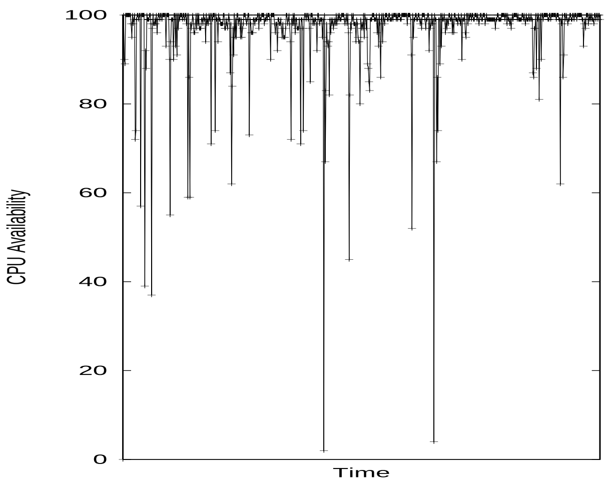

Figure 2. The actual usage pattern for Node 3

From the Table 2 and Table 1, we computed the maximum number of requests that can be potentially served $A_j(q)$, where q, the duration of the experiment, is 3570 seconds. $A_j(q)$ is found to be equal to 15872.

We observed that our system was able to service 14594 requests during the same time ($O_j(q)$). Thus, the efficiency of our system, $o_j(q)$, is 0.92. Apart from the overheads in the system, a potential factor that can cause a decrease in the efficiency is a faulty prediction scheme. From Table

2, we can see that our predictions are accurate for around 90% of the times for each of the Grid nodes.

Thus, in our simple study, we observe an efficiency factor of 0.92, which implies that it is able to utilization 90% of the unutilized desktop resources for running transaction workloads. However, we note that our studies are very simple in nature as they are conducted with relatively small number of desktops. We are planning to simulate this setup with more number of desktops using our model and analyze for higher number of grid nodes, in future.

5. Related Work

There are several groups working on resource management for Peer Grids, although their approaches differ due to their varying motivations and requirements.

One of the leading projects addressing scheduling for the Grid is Condor. Condor is a specialized workload management system for compute-intensive jobs [11]. It provides mechanisms for job queuing, scheduling, resource monitoring and resource management. The *ClassAd* mechanism provides a way of matching job requirements with resource offers. A central manager is responsible for scheduling the jobs on resources by matching these ClassAds. Certain types of jobs can also be checkpointed and migrated if the availability of the resources change during the course of execution of the job. Our target workloads are not the typical long-running scientific workloads that Condor targets, but are instead transactional workloads that have shorter turn-around times. Hence, migration does not make much sense in our case. In addition, evaluation of complicated ClassAds may be too much of an overhead for transactional workloads. In our case, requests from Grid clients for these transactions may arrive at a high rate. This necessitates replication of services so that requests from Grid clients can be processed in parallel. Condor is not based on such a request-response model, and does not need to replicate any jobs explicitly. To ensure higher throughput, it is also imperative that we predict the availability of our resources. Condor does not do any prediction of resource availability, and this makes sense in the case of long-running computational workloads, since the availability of resources can not be accurately predicted over long lengths of time. However, in our case, each request from a Grid client can be serviced in a short period of time, and predictions can be made reasonably accurately for shorter time intervals.

Another class of applications that are related to our work are the several projects dealing with *Volunteer Computing*, viz. Bayanihan [10], SETI@home, `distributed.net`, Entropia [5], etc. Typically, all such applications try to leverage cycles from voluntary underutilized resources on the Internet, and deal with applications that are embarrassingly parallel. In general, no guarantees are provided for the performance that can be obtained from such a set of resources. The scheduling policies of most such systems are not very complicated, since the participating resources *pull* work from a centralized *Work Manager* as and when they run out of work to execute. There is generally enough work to be pulled from such Work Managers to keep all the resources busy when they would otherwise be idle. In our case, we don't have a pool of work to keep distributing among the Peer resources. Instead, the amount of work to be done depends on the outstanding requests from the Grid clients. Thus, our work differs from traditional Volunteer Computing in the type of workloads that we target.

Leff et al [7] try to address delivering Service Level Agreements (SLAs) for commercial (transactional) workloads. However, their emphasis is not on leveraging idle cycles from resources, but reconfiguring resources inside a resource pool so that the number of resources that are currently serving customer requests are optimal for the SLAs agreed upon. They provide the scheduling of requests at a network level, using a Network Dispatcher (ND) [2], which is a load-balancing switch that distributes requests across a server cluster. However, a ND deals with requests at the packet level, and is oblivious to the type of service being requested. Hence, it is not very suitable to deal with requests to multiple services in the same resource pool. In addition, since it is at the network level, it is not very conducive for any kind of application-level scheduling. Currently, there is no prediction information being used, although it is part of their long term goals. Crawford et al [4] have also discussed a Grid using dedicated set of servers for deploying financial and content distribution type of applications. They describe a Topology Aware Grid Services Scheduler (TAGSS) for dynamic creation and deployment of Grid services.

6 Conclusions

In our previous work [9], we presented a three layer architecture for building a Desktop-based Peer Grid consisting of the Grid Service Layer, Logical Resource Layer, and Physical Resource Layer, with an emphasis on the Physical Layer. In this paper, we focussed on the Grid Service Layer, and presented how this layer provides the requisite services so that Grid clients can use the varying set of resources effectively. We described the design and architecture of the Gateway, which relies on performance prediction to compute the idle cycles that can be provided by the desktop machines, and route requests from Grid clients to the resource most likely to remain idle during the duration of the execution. We also presented a simple model for our system, and verify it with our experiments. We found that our simple prediction algorithm performs reasonably well for the type of resources that we target.

Our preliminary results are encouraging and lead us believe that the concepts presented here can be used for off-loading peak demands on backend servers, for testing and deploying new releases of backend applications or for improving the availability of existing mission critical backend infrastructure, by sharing underutilized resources across an organization.

In the future, we plan to test our system with several other prediction algorithms. We also plan to perform experiments involving more desktop machines in our department to verify the scalability of our architecture, and identify possible bottlenecks.

References

[1] Apache SOAP, as of July 2003. http://ws.apache.org/soap/.

[2] IBM Network Dispatcher, as of July 2003. http://www-3.ibm.com/software/network/dispatcher/.

[3] IBM WebSphere, as of July 2003. http://www.ibm.com/websphere/.

[4] C. H. Crawford, D. M. Dias, A. K. Iyengar, M. Novaes, and L. Zhang. Commercial Applications of Grid Computing, Jan. 2003. IBM Research Report, RC22702, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA.

[5] Entropia PC Grid Computing. DC-Grid Platform, as of July 2003. http://www.entropia.com/dcgrid_platform.asp.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[7] A. Leff, J. T. Rayfield, and D. M. Dias. Service-Level Agreements and Commercial Grids. In *IEEE Internet Computing, Special Issue on Grid Computing*, July 2003.

[8] Z. Luo, S. Chen, S. Kumaran, L. Zhang, J. Chung, and H. Chang. A Web-Service-Based Deployed Framework in Grid Computing Environment, May 2002. IBM Research Report, RC22470, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA.

[9] V. K. Naik, S. Sivasubramanian, D. F. Bantz, and S. Krishnan. Harmony: A Desktop Grid for Delivering Enterprise Computations, *To appear* in the Proceedings of Grid 2003. Also available as IBM Research Report, RC22832, IBM T. J. Watson Research Center, Yorktown Heights, NY, USA.

[10] L. Sarmenta. Web-based Volunteer Computing using Java. In *Proc. 2nd Intl. Conference on Worldwide Computing and its Applications*, 1998.

[11] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. *Beowulf Cluster Computing with Linux*, chapter 15, Condor - A Distributed Job Scheduler. MIT Press, 2002.