Adaptive Resource Sharing in a Web Services Environment

Vijay K. Naik[1], Swaminathan Sivasubramanian[2], and Sriram Krishnan[3]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
vkn@us.ibm.com
[2] Vrije Universiteit, Amsterdam, The Netherlands
swami@cs.vu.nl
[3] Indiana University, Bloomington, IN 47405
srikrish@cs.indiana.edu

Abstract. One effect of the push towards business process automation and IT consolidation is that low-level resources from multiple administrative domains are shared among multiple workloads and the middleware is called upon to bring about the integration while masking the details of sharing such resources. Web services and grid based technologies hold promise for developing such middleware. However, existing solutions do not extend well when resources to be shared belong to multiple administrative domains and when resource sharing is governed by local policies. In this paper, we describe an architecture for adaptive resource sharing among two types of workloads: (i) local resource specific workload and (ii) global web services based grid workload. Each resource can set its own policies regarding how the resource is to be shared. Our approach leverages both the grid and the web services based technologies and overcomes the limitations of existing solutions by providing an additional layer of middleware. This layer provides services for dynamic discovery and aggregation of resources, policy based and transparent management of resources, and dynamic workload scheduling using the concept of virtualized resources. We discuss some of the design choices we made and present performance results to show the effects of policy-based resource sharing on the throughput delivered to the grid workload.

1 Introduction

In an Enterprise many different types of applications are used to deliver IT services (e.g., financial, accounting, supply-chain management, e-commerce, billing, customer relations). Each serves its own workload and typically each service is run on servers dedicated to provide that service. One drawback of using a dedicated set of servers is that sufficient server capacity must be allocated to handle peak workloads in order to meet the response time and throughput guarantees. Since not all workloads are correlated, when one service is dealing with peak workload, demand on other services may be at or below average. Therefore, if resources can be shared by two or more non-correlated services, average utilization of the resources can be raised and higher workload demands can be met without provisioning resources for the worst case scenarios.

H.-A. Jacobsen (Ed.): Middleware 2004, LNCS 3231, pp. 311–330, 2004.

While resource sharing is beneficial from IT consolidation point of view, sharing of resources by different types of applications and their workload is not straightforward even when resource capacity is not an issue. This can be because of application-legacy reasons, workload-resource affinity issues, unpredictability in the workload arrival patterns, security and isolation needs, and so on. Obviously, applications that require special hardware or non-generic platform configurations cannot easily share their resources with other applications. Workloads that have affinity to specific resources (e.g., desktop users sending their interactive workload to their own desktops) cannot be sent over to other resources, but it may be possible to share the underlying resources with other types of workloads. In this paper, one of our workloads is of this type. Unpredictability in the demand leads application and system administrators to take conservative approaches and to over-provision resources. However, even limited predictability about the demand can help administrators in setting policies so the underlying resources can be shared adaptively and dynamically in anticipation of the predicted demand. We use such an approach in our present work. Security and isolation are real concerns even when all workloads are generated within the same organization. Resource sharing can lead to security related compromises. In this paper, we describe an approach that uses hypervisor-based virtual machines. This is an extension of our earlier work described in [11]. Specifically, we make use of the VMWare Workstation product [4] to isolate the two types of workloads. Hypervisor-based virtual machines such as these address OS and application level security concerns and thus facilitate resource sharing without compromising isolation.

In this work, we describe a middleware that enables resource sharing among two classes of workloads that characterize a typical enterprise environment: (i) resource specific workload, in particular, the interactive workload submitted by users to their desktops and (ii) workload for applications that conform to the J2EE programming model and, in particular, to the Web Services programming model. In an enterprise environment, desktops, in aggregate, represent one of the largest set of underutilized resources and their raw capacities are increasingly matching server capacities. Secondly, the Web Services programming model is increasing being adopted by enterprises to automate many of their business processes and many different types of workloads can be handled by Web Services [5]. The Web Services abstraction allows development of general purpose containers that can be supported by a variety of platforms. General availability of such containers makes it possible to deploy and process Web Service requests on a variety of platforms.

Although desktop systems are resource rich and they are highly underutilized, harnessing the available cycles from desktop based resources and applying them in an aggregate manner for supporting business processes is a hard problem. The problems are primarily related to the conflicting requirements placed by the desktop users and by the workload for the Web Services. For the desktop users, interactivity, responsiveness, and security are of prime concern. From the point of view of the business process clients, 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 Web Services clients it is important that a sufficient number of resources be available at all times to handle the workload. As we discuss in Section 2.3, desktop users may set policies, independent of other users, that determine how their desktop system is to be shared with some other workload. It is impractical for the Web Services clients to go hunting for resources that may be idle at that instance. Because of the unpredictability in the desktop resource availability, a mechanism is needed to identify resources available for processing the Web Services dynamically and to route the requests to the appropriate resources transparent to the client(s) generating the workload. It is also equally important to mask the changes in the resource pool available for sharing.

Transparent management of resources that belong to multiple administrative domains is at the core of grid computing architecture. 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 [8]. With this service oriented approach of OGSA, Grid computing has become an attractive platform for deploying business applications and for supporting commercial workload. 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 to determine the manner in which a resource is to be shared by grid and non-grid workload. Thus, grid based technologies, and OGSA in particular, are ideal candidates for enabling desktop (and in general, any resource) sharing among multiple types of workloads. However, 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 2.2.

In this paper, we describe a middleware architecture that overcomes the above discussed resource sharing difficulties. In our architecture, a virtual service layer is created that provides an additional layer of indirection. This helps to mask the actual changes in the physical resource layer from the workload for the Web Services. To reduce the overhead of this additional layer of indirection, the dynamic effects of the policies governing the physical resources (in this case the desktop systems) transcend the multiple layers and make their effect visible to the request scheduling layer without reducing the transparency or the administrative independence of the individual resources.

The rest of the paper is organized as follows. In the next section, we first briefly discuss the J2EE programming model and describe the characteristics of the transactional business oriented workload. In Section 2.3, we discuss the requirements of desktop systems, when used as shared resources. In Section 3, we describe the highlights of our middleware architecture. In Section 4, we discuss the design and implementation of the Gateway that acts as the coordinator between the deployed Web Services and the Grid clients. Performance results from an implementation of our architecture are described in Section 5. In Section 6, we discuss how our work relates to other work in the literature. Finally, we present our conclusions in Section 7.

2 Preliminaries

2.1 J2EE Programming Model

Java 2 Enterprise Edition (J2EE) is a standard for developing enterprise applications in a composable manner using reusable Java components, called Enterprise Java Beans (EJBs). The standard spells out programming interfaces that the EJB components conform to. The EJBs provide services to local and remote clients. The EJB components and their life cycle are managed within a special container provided by an application server such as IBM's WebSphere Application Server [3]. The J2EE standard makes it possible so that EJBs can be developed in a container independent manner and the a J2EE compliant application server can handle its life cycle by providing a set of standard runtime services specified by J2EE. These services include naming and directory services, authentication and authorization, state management, interfacing with a Web server, and so on. The application server handles all the platform and vendor specific details such as those involved initializing and maintaining interactions with the OS, databases, and network subsystems. The J2EE programming model provides an abstraction for the multi-tier architecture implicit in many business processes. The persistent state is maintained in the database tier, processing of this state is performed in the business logic tier, and the results of which may be rendered in a presentation tier before being sent to a remote client. The remote client itself could be an application. This makes it possible to integrate multiple business processes using standard technologies.

For more information on the J2EE standards and the associated programming models, we refer interested readers to [10]. In the context of our work, Web Services are specialized EJB components. Web Services further standardize the distributed interactions by conforming to Web based technologies. Complex business processes can be modeled and deployed using reusable components. However, application servers such as IBM WebSphere Application Servers need to be deployed in order to run the Web Services and handle their workload.

2.2 Characteristics of Transactional Applications

In the following, we highlight the requirements posed by the transactional business applications and services. We contrast these against the requirements posed by the scientific and engineering applications, which have motivated the development of the classical grid related middleware work.

Transactional business services exhibit *high degree of interactivity* with human operators and/or with databases where business state information is held in a persistent manner. 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 handle the peak demand, necessary resources must be allocated dynamically and on demand. Moreover, the resource management mechanisms must be sensitive to 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. To provide these functionalities, middleware services are needed 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 re-

sources needed to satisfy the demand. Such a middleware 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 3 we describe our architecture that responds to response time, throughput, and availability requirements.

2.3 Sharing Desktop-Based Resources

The primary objective of desktop systems is to provide a high degree of interactivity to desktop users 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. On a desktop system, there are many unused cycles, but their frequency and duration are highly unpredictable. Moreover, desktop users (or administrators) may set policies that enforce conditions under which desktop resources may be used for processing other workload. 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 Web Services at any given time, (iii) allow processing of grid workload only when the interactive workload is below a certain threshold, (iv) allow participation in the Grid computations only during certain time of the day or on certain days of the week, and so on. 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 individual 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.

Clearly, for effective utilization of desktop systems in a Grid like environment, 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 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 Web 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 middleware 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.

3 Architecture Overview

3.1 Requirements

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 is throughput and response time. This means the architecture should be able to deliver a service requested by the clients on demand and it should be able to adjust the capacity of the underlying resource 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.

Intuitively, the desired architecture needs to facilitate (i) deployment of appropriate Web Services on the desktop resources, and (ii) route client requests to appropriate Web Service instances. These tasks are made challenging because of (i) the uncertainties in the resource availability for deploying a Web Service at any given instance in time and (ii) the uncertainties in the client demand on a Web 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 deploy Web Service instances on appropriate desktop resources so as to empower them with the desired capacity just-in-time for delivering the service, and (ii) to identify and match Grid client requests with Web Service instances with appropriate capacity (i.e., with ability to respond within prescribed time limits).

Even when sufficient desktop resources are available on an aggregate basis, efficient identification of the once that can provide the desired capabilities at any given time requires good prediction mechanisms. Given the ability to discover, monitor, and predict resource availability and demand, 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. Note that the predictions need to be only reasonably accurate and not highly accurate.

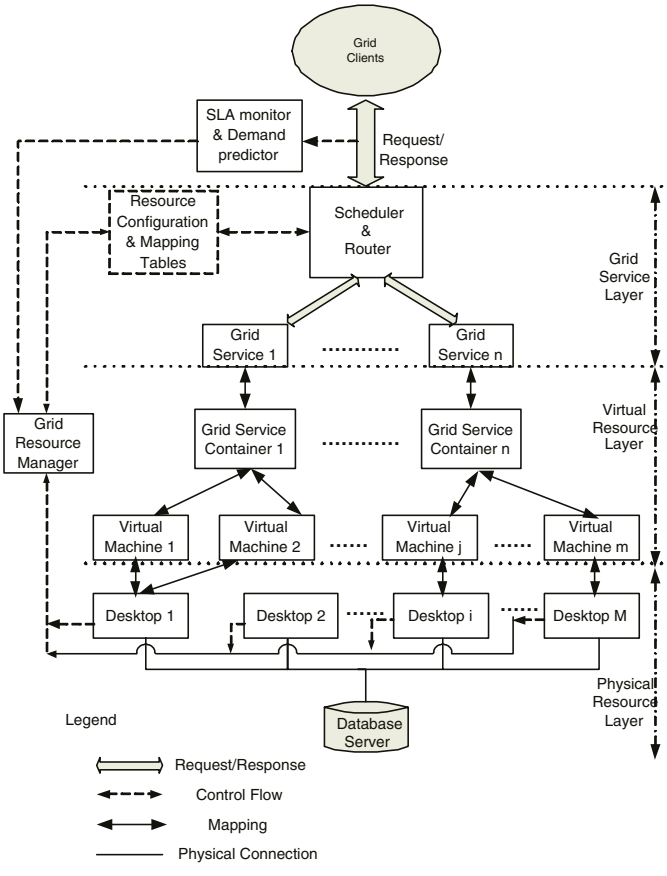

Fig. 1. Layered architecture for the enabling policy-based resource sharing. The interactive workload for the desktops is not shown.

3.2 Highlights of Our Architecture

The architecture is defined using a layered approach. This allows addressing the requirements of the transactional workload 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.

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 Web 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 Web Service type. Based on the predicted arrivals and available Web 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 Web 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 Web 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 Web Service. This are described in more detail in [11]. 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. One function provided by GRM is to discover desktop resources that are available and are capable of deploying one or more Web Services. This is accomplished by coordinating and heart-beating with a CMC known as a Virtual Machine Manager (VMM). One instance of VMM runs on each physical desktop resource as long as that resource is participating as a grid resource. It controls and monitors all virtual machines on that resource. Using the same mechanisms, it also detects when a desktop resource is no longer provisioning a particular Web Service or is no longer available as grid resource. A second key function provided by GRM is to allocate the predicted capacities of each participating desktop resource to Web Services requiring the resource during that future time interval. It tries to locate and allocate as many resources to each Web Service as possible making sure that the conflicts caused by sharing are minimized. To perform this task, GRM collects from each VMM the usage and policy related data for its resource and predicted availabilities. The VMM in turn uses a platform specific CMC known as the Host Agent for monitoring, gathering, and aggregation of performance and usage related data. GRM normalizes the raw capacity of each desktop against a standard platform. In case the desktop node is to be shared among multiple Web Services, it further reduces the available capacity in proportion to the share made available for other Web Services. This represents the maximum normalized capacity available to a particular Web 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 Web Service. This forms the basis for the allocation of resources to the expected workload. This information is represented in the Resource Configuration & Mapping Tables shown in Figure 1 and is dynamically updated by GRM.

The Scheduler uses this information to determine the number of service instances to deploy for each Web 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 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 Web Service.

4 Gateway Architecture and Design

4.1 Gateway Requirements

An important component of our architecture is the *request scheduler* responsible for scheduling 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 an appropriate end point resource (also referred to as a Grid node), where the request is processed. Recall that this may be a shared resource. The request router monitors the execution status of each request it has dispatched. 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 request is not dropped.

The request scheduler and router components together form a single logical component referred to as the *Gateway*, which serves as the entry point for the requests from Grid clients. It is a logically centralized component. However, if necessary (for reasons of scalability), it can be implemented as a federation of gateways, as described in section 4.2.

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 coordinates with various CMCs of physical and Grid layers, and smoothes 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, since for transactional workloads the execution times are 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.

4.2 Gateway Design and Implementation

In our design, we make a clear demarcation of resource prediction models from on-line allocation and scheduling components. Such a demarcation 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.
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 client requests independent of the changes in the participating Grid nodes. However, as noted in the preceding section, the Gateway (through the SLA monitor) must communicate with GRM and inform the expected demand for a particular Web Service, to ensure that enough resources are allocated for provisioning each type of Web Service requested by clients.

Design Choices. The design of Gateway can be done in several ways: For example, Gateway can be built using network-level redirector 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 redirectors that cluster together the available servers into a logical resource pool and route requests to the cluster [9]. APIs are provided to allocate or deallocate servers from the pool. 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 Web Services. If not, then it requires the use of one network-level redirector for each Web Service. Thus, this approach lacks flexibility in adding new Web 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 requests using 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 types of Web 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 Web 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.

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 submit their requests with 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.

4.3 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.

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 [6]. 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.

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.

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.

As we noted earlier, a Grid request can be executed in multiple Grid nodes for reasons of fault-tolerance. Replicating a transaction is straightforward if the transaction does not update the backend database. However if the transaction modifies the database, then the system must commit only one of the replicated transactions. Such scenarios warrant a database middleware that identifies such replicated transactions (using an unique transaction identifier) and commits only one of them. Our current implementation supports only replication of read-only transactions. For update transactions, we plan to build such database middleware for replicated transactions in the future.

5 Performance Evaluation

5.1 Performance Modeling

We now describe a model for quantifying the variability in the capabilities of the resources that are used to form the resource grid. 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 quantifies the capabilities of a computing resource, and is the ratio of the capability 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 availabilities 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 availability (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.

5.2 Experiment Setup

We tested the performance of our system on a small scale with a set of five Grid nodes. We logged the CPU utilization 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 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.

5.3 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 Table 2. 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.

Table 1.

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

Table 2.

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

From Tables 1 and 2, 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, from our experiments, we observe an efficiency factor of 0.92, which implies that the system is able to utilize over 90% of the unutilized desktop resources for running transactional workloads. This also implies that (i) the overheads introduced by request redirection at the Gateway is minimal, and (ii) the prediction algorithm is effective enough to handle the bursty desktop

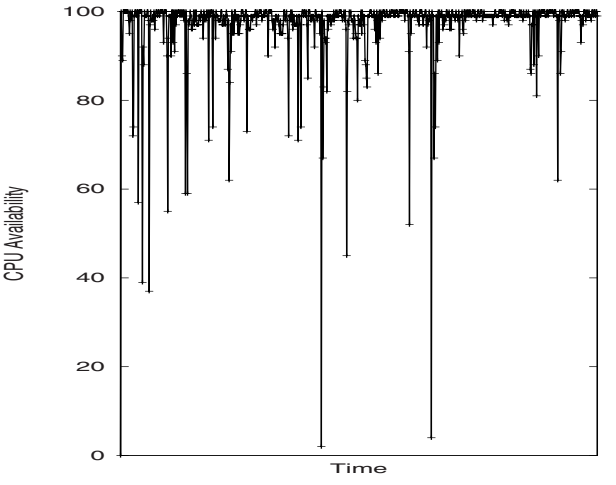

Fig. 2. The actual usage pattern for Node 3.

workloads with minimal overhead. We note that our studies are simple in nature as they were conducted with a relatively small number of desktops. We are planning to extend this study to include a larger number of desktops with more comprehensive workload scenarios. We are also planning to develop a simulator to analyze our model with a large set of parameters.

6 Related Work

There are several groups working on resource management in the context of grid and peer computing, although their motivations and approaches differ from ours.

One of the leading projects addressing scheduling for the Grid is Condor. Condor is a specialized workload management system for compute-intensive jobs [14]. 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 [13], SETI@home, `distributed.net`, Entropia [7], 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 grid 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 [9] 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 using a Network Dispatcher (ND) [2], which is a load-balancing switch that distributes requests across a server cluster. Hence, it is not very suitable to deal with requests to multiple services in the same resource pool. Currently, there is no prediction information being used, although it is part of their long term goals. Crawford et al [6] 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.

7 Concluding Remarks

In this paper, we have described a middleware architecture that enables policy based sharing of enterprise resources by two types of workloads: (i) a resource specific workload and (ii) a Web Services based grid workload. We have designed and implemented the system where desktop based resources are the shared resources. These type of resources represent a rich source of underutilized resources in many organizations and also because the resource specific workload (in this case, the interactive workload submitted by desktop users) is more likely to be non-correlated with any external grid workload. We address the key concerns of desktop users for responsiveness, privacy and protection by isolating the grid

workload in a virtual machine in the desktop. The proposed architecture and its key design concepts are equally applicable to other types of resources in an enterprise, e.g., they can be the servers that are primarily used for running backend applications that can be shared by their primary workload and the grid workload.

The salient components of our middleware architecture are: (i) Grid Resource Manager, (ii) End-point resource agents, and (iii) Scheduler and Router. Together, these components enable dynamic discovery of resources, identification of services offered to grid workload on the resources, predicted capacity available to grid workload, policy management, deployment and provisioning of services, and transparent scheduling and routing of the grid workload. With this middleware, individual resources are free to join and disconnect, based on their local policies, from the pool of resources available to the grid workload. The middleware manages these changes transparently from the grid clients as well as from the other resources in the pool. The scheduling of grid workload adapts dynamically to the availability and sharing capabilities of the individual resources.

To deploy and process Web Services, we use IBM WebSphere Application Server – a J2EE container – on each shared resource. Use of such a container has advantages and disadvantages. The container masks the heterogeneity in the underlying resources and creates a homogeneous environment so a request can be processed on any one of resources. For Web Services based workload this choice works out very well. However, for any workload that is not supported by the current container technology, the middleware discussed here may not be readily suitable. We also note that in the current release of WebSphere Network Deployment Edition (Release 5), multiple application servers can be configured to form a cell and requests can be processed on any one of the members of such a cell, transparent to the client making the request [3]. Such a cell is essentially a static cluster of application servers that cannot dynamically join and disconnect from the cell without affecting the operations in the rest of the cell. In our approach, we do not use WebSphere's network deployment mechanisms. Our middleware is designed specifically to handle dynamic changes in the resource availability. The main difference between our system and a statically configured system is that in our system the actual resources providing a service may enter and exit the resource pool, but the system continues to service client requests and clients are unaware of the low level dynamic changes taking place in the system. However, many of the reliability issues are handled in a manner similar to those in a statically configured system. In addition, the Gateway maintains the client request state until a response is sent back. By continuously monitoring the availability of the underlying resources providing a service and by discovering and provisioning additional resources the system tries to masks the changes from the clients of the service.

One disadvantage of using managed container such as that provided by WebSphere is that configuring and deploying such a container is not trivial and requires some domain specific expertise. In our approach, we deploy virtual ma-

chines that are pre-configured with the WebSphere Application Server. This reduces some of the system management complexities and improves the degree to which the resource configuration can be automated. However, because of the bandwidth requirements, our solution is well suited for resources in an intranet environment and not over a wide area network. Recently efficient mechanisms have been proposed for migrating virtual machine states [12] and we are investigating the applicability of such approaches to our work. Similarly, in the three-tier distributed computing model (i.e., client logic, business logic, and data logic) that is inherent to most business processes and is supported by the J2EE technology, a Web Service providing the business logic may require access to a database for state information. In this work, we assume that a Web Service can access the necessary state information from any of the resources where it may be deployed. Again for this reason, the solution discussed here is better suited for an intranet environment. Note that resources over the intranet can belong to multiple administrative domains.

Another point to note here is that in our approach, the request to a Web Service is trapped in the Gateway before it is routed to the most suitable resource available at that time. This requires partial processing of the request at the "Network Layer 7" and this may introduce significant overhead especially when the service time of the request is relatively short. However, the advantages of processing requests at Layer 7 are obvious as reflected in our architecture.

Resource sharing across multiple workloads, as described in this paper, is effective when the capacity prediction mechanisms are accurate. Primarily, we rely on predictions about the available capacity on each shared resource for the grid workload. When a resource is to be shared with an interactive workload, as we do in this work, the prediction mechanisms have certain limitations. By sampling over a long period of time and by using techniques based on time series analysis, one can identify daily, weekly, and seasonal patterns in the utilization for each type of resource for a specific workload. For example, desktop utilization during evening and night hours is mostly zero while during the day time on a weekdays it is significantly higher. In the case of interactive workload, for the intervals where there are no long term patterns, fairly accurate short term predictions may be possible. We discuss one such mechanism in [11] that we found to perform reasonably well. However, the effectiveness of such an approach diminishes for large time intervals (e.g., larger than tens of minutes) and intervals that are further away in the future. We note that in case of many business application that run on backend servers, the workload is more predictable than the interactive workloads on the desktops. In such cases, the resource sharing can be more straightforward. Similarly, predictions about workload arrival patterns can improve the effectiveness of dynamic resource provisioning. We note that the predictions need not be highly accurate, but higher accuracy can result in better performance (i.e., response time or throughput) using a smaller number of resources.

Our results and experience so far with the system we have developed have been encouraging and lead us believe that our middleware provides a flexible

design for sharing enterprise resources by a resource specific workload and a global Web Services based workload. In an enterprise environment, such a system can be used for off-loading peak demands at data centers and 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 expand the scope of this word along the following dimensions: (i) to share a resource across multiple types workloads, (ii) use of a light weight virtual machine, (iii) development and testing of other prediction algorithms, and (iv) affinity based request routing.

References

1. Apache SOAP, as of July 2004. http://ws.apache.org/soap/.
2. IBM Network Dispatcher User's Guide, as of July 2004. ftp://ftp.software.ibm.com/software/websphere/info/edgeserver/ndugv3-us.pdf.
3. IBM WebSphere, as of July 2004. http://www.ibm.com/websphere/.
4. VMWare, as of July 2004. http://www.vmware.com/.
5. J. Chung, K. Lin, and R. Mathieu. Guest Editor's Introduction–Web Services Computing: Advancing Software Interoperability. *Computer 36(10)*, 2003.
6. 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.
7. Entropia PC Grid Computing. DCGrid Platform, as of July 2004. http://www.entropia.com/dcgrid_platform.asp.
8. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.
9. 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.
10. S. Microsystems. J2EE Platform Specification, 2002. http://java.sun.com/j2ee/.
11. V. K. Naik, S. Sivasubramanian, D. F. Bantz, and S. Krishnan. Harmony: A Desktop Grid for Delivering Enterprise Computations. November 2003.
12. C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.
13. L. Sarmenta. Web-based Volunteer Computing using Java. In *Proc. 2nd Intl. Conference on Worldwide Computing and its Applications*, 1998.
14. T. Tannenbaum, D. Wright, K. Miller, and M. Livny. *Beowulf Cluster Computing with Linux*, chapter 15, Condor - A Distributed Job Scheduler. MIT Press, 2002.