RC22702 (W0301-100) January 22, 2003
Computer Science

IBM Research Report

Commercial Applications of Grid Computing

Catherine H. Crawford, Daniel M. Dias, Arun K. Iyengar, Marcos Novaes, Li Zhang
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

COMMERCIAL APPLICATIONS OF GRID COMPUTING

Catherine Crawford, Daniel Dias, Arun Iyengar, Marcos Novaes, and Li Zhang
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY, 10598
USA
{catcraw,dias,aruni,mnovaes,zhangli}@us.ibm.com

Abstract This paper provides an overview of commercial applications of Grid computing. We discuss Web performance and present a Grid caching architecture. Our Grid caching architecture offloads requests to Grid caches when Web servers become overloaded. We describe performance and traffic modeling techniques which can enhance Grid applications such as caching. We also discuss how Grid computing can be applied to financial applications. A key requirement here is that fast response times are needed. We present a Grid services scheduler that is well suited to commercial applications requiring fast response times.

Keywords: caching, Grid computing, performance modeling, traffic modeling, Web performance.

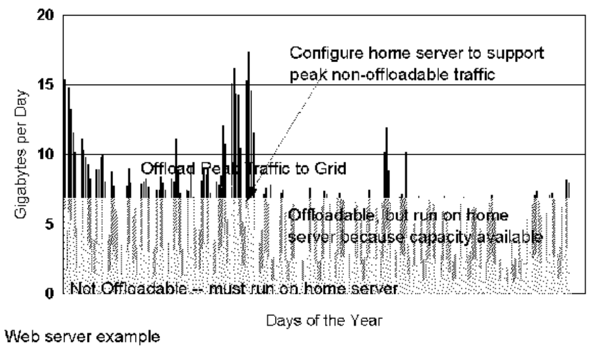
1. Introduction

Grid computing is evolving as the next major distributed computing platform. In the 1990s, large scale cluster computing became a commercial reality [17, 2], with clusters used for both commercial computing, such as for scalable Web Servers [13], and for parallel scientific computing [1]. More recently, cluster computing has evolved to distributed computing on a large scale, across geographic and, in some cases across organizational boundaries, and has been referred to as Grid computing [9]. Many of the early applications of Grid computing were for parallel scientific applications, distributed beyond locally distributed clusters to Grids of multiple clusters or supercomputers across geographically distributed sites. In this paper we examine commercial applications of Grid computing, particularly for Web serving and financial applications.

Concurrently with the emergence of Grid computing, the World Wide Web (Web) has been evolving from primarily that of information access in the 1990s, to a Service Oriented Architecture, using the Web Services paradigm. The Web Services protocols [24]that have been defined can use the Web HTTP protocol for transport, and provide a services-oriented architecture; this includes an interface definition language called Web Services Definition Language (WSDL) [4], for accessing remote services. These two industry trends of Grid computing and Web Services are converging in the recent definition of the Open Grid Services Architecture (OGSA) in the Global Grid Forum [11, 23]. OGSA uses the extensibility defined in WSDL, and specifies a Grid service factory, instance, registry, discovery, life-cycle, service data, among other OGSA service interfaces and behaviors [10].

The initial applications on the Grid were scientific numerically intensive computing applications, that extend the computations from cluster supercomputers to the Grid. For example, the U.K. Scientific Grid (http://www.ercim.org/publication/Ercim_News/enw45/boyd1.html, http://www.escience-grid.org.uk/) had an initial focus on bioscience applications, such as molecular simulation, earth science including climate research and earth observation, and astronomy. The Tera Grid links supercomputing Centers in the U.S, with a focus on open scientific research. One commercially oriented Grid project is the eDiamond Grid for breast cancer screening and diagnosis which is building a "national digital mammography archive for the UK", and a similar mammography Grid with the University of Pennsylvania (http://www.gridtoday.com/02 /1104/100640.html). Another commercial Grid is the Butterfly Grid for multiple concurrent game players (http://www.butterfly.net/). Commercial Grids are in the exploratory stage especially in the financial, life sciences, petroleum, and auto industries.

One of the key motivating factors for using a Grid for commercial applications is illustrated in Figure 1. This figure shows the load on a Web site, in terms of the Gbytes/day served. Also shown is the portion of the data served that is static and can be offloaded to a Web cache; the remaining dynamic data which typically needs to run on the home server is labelled as "non-offloadable". The home server must be configured to handle the peak of the non-offloadable traffic; however, configuring the home server for the peak non-offoadable traffic enables it to serve most or all of the traffic for most of the time, except for peak loads. Thus, one of the key advantages of Grid computing for such commercial applications, is that of being able to configure the home server for the peak non-



Web server example

-- For typical Web serving, about 30% was not offloadable

-- Applies to other applications with different fractions offloadable to Grid

-- For technical computing / compute intensive, offloadable fraction often ~ 100%

-- For high update transaction environments, offloadable fraction small

Figure 1. Web traffic, and the proportion which can be offloaded.

offloadable traffic, and to be able to offload all of the peak traffic to the Grid. In Section 3 we will describe how this can be achieved for Web serving. Note that typical Web caching services offload all of the cacheable traffic, regardless of whether the home server can handle the traffic or not, rather than offloading only the peak traffic, when needed. In Section 4 we show how financial applications can be speeded up by selectively offloading some of the computation to Grid servers.

One categorization of Grids is in terms of intra-grids, extra-grids and inter-grids. Intra-grids are distributed within an organization. Typically, this would be across geographically distributed sites within the organization, though it may be across multiple clusters or servers on a campus. Most of the commercial applications of Grids today are for intra-grids, often across multiple sites of a company, which have peak loads at different times. One principal reason for this is security, especially the data security in an execution environment. In this paper, given the focus on commercial applications of Grids, we will primarily consider intra-grids. Extra-grids open intra-grids to specific trusted partners. For commercial applications, extra-grids could allow the off-loading of peak traffic to a trusted third party, for example to an application hosting service provider. The scientific Grids mentioned earlier fall into this category. Inter-grids are defined as linking multiple Grids, to allow sharing

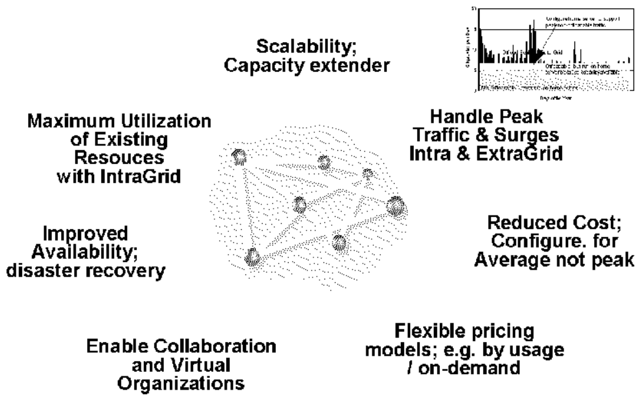


Figure 2. Business value of adopting a Grid model.

of resources across Grids with dynamic resource discovery. Inter-grids are in the research stage, and there are no examples at this time.

The business value of adopting a Grid model is summarized in Figure 2. First, it can provide for scalability, by allowing the use of underutilized resources, both within a company in the intra-grid environment, and via outsourcing to an extra-grid. Most compute environments have a peak load much higher than that of the average load, as discussed earlier. The Grid can be used to off-load the peak traffic, either to available intra-Grid resources, or to an extra-Grid. This could lead to significant cost savings by configuring a system for average rather than peak traffic, and through more efficient usage of existing resources. In configurations with an off-load of peak traffic to an extra-Grid, pricing models based on usage, rather than peak traffic configurations, can also lead to cost savings. Properly configured, a Grid based solution can also provide higher availability and support for disaster recovery.

The remainder of this paper is organized as follows. In Section 2 we describe issues related to high-performance Web serving, a critical commercial application which can benefit from the Grid. In Section 3, we describe how Grid technology can be used for Web caching, especially to handle peak traffic loads. We also discuss how performance modeling and traffic modeling can enhance Grid caching. The use of Grids for financial applications, and a Grid scheduler for these applications, are described in Section 4. Finally, concluding remarks appear in Section 5.

2. High-Performance Web Serving

Web serving is a critically important application that can benefit from parallel processing and Grid computing. Web serving lends itself well to concurrency because requests from different clients can generally be processed independently. A Web site which receives significant traffic would typically contain many servers. The servers might be geographically distributed in order to bring content closer to clients as well as to increase availability.

Requests can consume widely differing amounts of resources to satisfy. If I/O bandwidth is the bottleneck, then large objects become undesirable to serve. Image files can consume significant I/O bandwidth, so limiting the use of images can improve performance considerably. Requests for files are known as static requests and generally consume less overhead than dynamic requests which invoke programs to generate data on-the-fly for satisfying requests. Requests for dynamic data can consume orders of magnitude more CPU time to satisfy than requests for static data. Therefore, even if a Web site serves only a fraction of its requests dynamically, dynamic requests can consume the bulk of the CPU cycles. Static requests are easier to offload to a Grid than dynamic requests.

Encryption can also add significant overhead to a Web site when confidentiality is required. Encryption is typically handled on the Web using the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocol [18, 8]. The SSL protocol requires a handshake at the beginning in order for the client and server to negotiate a session key used for encrypting data via symmetrical cryptography. Session key generation is expensive. The overhead of session key generation is reduced by using the same session key for multiple transactions. In order to limit security exposure, session keys have a limited lifetime after which they must be changed. In Grid computing, security and encryption are often less important for intra-grid environments than for extra-grid or inter-grid environments.

In a scalable Web site, requests are distributed to multiple servers by a load balancer. The Web servers may access one or more databases or other back-end systems for creating content. The Web servers would typically contain replicated content so that a request could be directed to any server in the cluster. One way to share static files across multiple servers is to use a distributed file system such as AFS or DFS [15]. Copies of files may be cached in servers for faster access. This approach works if the number of Web servers is not too large and data doesn't change frequently. For large numbers of servers for which data updates are frequent, distributed file systems can be highly inefficient. Part of the reason for this is the strong consistency model imposed by distributed file systems. Shared file systems require copies of files to be strongly consistent. In order to update a file in one server, all other copies of the file need to be invalidated before the update can take place. These invalidation messages add overhead and latency. At some Web sites, the number of objects updated in temporal proximity to each other can be quite large. During periods of peak updates, the system might fail to perform adequately. Shared file systems would be more appropriate for an intra-grid environment than an extra-grid one.

Another method of distributing content which avoids some of the problems of distributed file systems is to propagate updates to servers without requiring the strict consistency guarantees of distributed file systems. Using this approach, updates are propagated to servers without first invalidating all existing copies. This means that at the time an update is made, data may be inconsistent between servers for a little while. For many Web sites, these inconsistencies are not a problem, and the performance benefits from relaxing the consistency requirements can be significant. This approach for distributing content is easier to apply to a Grid environment than that of shared file systems.

Load balancers distribute requests among multiple Web servers. One method of load balancing requests to servers is via DNS servers. DNS servers provide clients with the IP address of one of the site's content delivery nodes. When a request is made to a Web site such as *http://www.ibm.com/employment/*, "www.ibm.com" must be translated to an IP address, and DNS servers perform this translation. A name affiliated with a Web site can map to multiple IP addresses, each associated with a different Web server. DNS servers can select one of these servers using a policy such as round robin [3].

One of the problems with load balancing using DNS is that name-to-IP mappings resulting from a DNS lookup may be cached anywhere along the path between a client and a server. This can cause load imbalance because client requests can then bypass the DNS server entirely and go directly to a server [7]. Name-to-IP address mappings have time-to-live attributes (TTL) associated with them which indicate when they are no longer valid. Small TTL values can limit load imbalances due to caching. The problem with this approach is that it can increase response times [19]. Another problem with this approach is that not all entities caching name-to-IP address mappings obey TTL's which are too short.

Caching of name-to-IP address mappings is a problem if DNS is used to route request to Grid servers or caches because a Grid server cannot easily be removed; clients may continue to route requests to a former Grid server even if the DNS has stopped sending requests to the server.

Another approach to load balancing is using a connection router in front of several back-end servers. Connection routers hide the IP addresses of the back-end servers. That way, IP addresses of individual servers won't be cached, eliminating the problem experienced with DNS load balancing. Connection routing can be used in combination with DNS routing for handling large numbers of requests. A DNS server can route requests to multiple connection routers. The DNS server provides coarse grained load balancing, while the connection routers provide finer grained load balancing. Connection routers also simplify the management of a Web site because back-end servers can be added and removed transparently.

IBM's Network Dispatcher [12]is one example of a connection router which hides the IP address of back-end servers. Network Dispatcher uses Weighted Round Robin for load balancing requests. Using this algorithm, servers are assigned weights. All servers with the same weight receive a new connection before any server with a lesser weight receives a new connection. Servers with higher weights get more connections than those with lower weights, and servers with equal weights get an equal distribution of new connections.

With Network Dispatcher, requests from the back-end servers go directly back to the client. This reduces overhead at the connection router. By contrast, some connection routers function as proxies between the client and server in which all responses from servers go through the connection router to clients.

In the next section, we show how load balancing is used to send requests to Web servers as well as to Grid caches which are used when the Web servers become overloaded. The load balancer is configured to route requests to new caches in response to increasing load.

Caching on the Web exists in a number of different forms. Client browsers may cache objects so that they don't have to be fetched from remote sources. Proxy caches are caches which exist on Web proxy servers. When a client connects to the Web, the client typically goes through a proxy server which is shared by many clients. The proxy server may have a cache which is shared by several clients. Static documents which are designated as being cacheable may be stored in the proxy cache when first requested by a client. A subsequent request for the object from another client sharing the cache would obtain the object from the proxy cache instead of going out to the network to fetch the object.

Content distribution networks (CDN) cache content remotely from servers at edges of the network. Web sites will pay to use a CDN to offload requests and move content closer to clients. By contrast, proxy caches function on behalf of clients, and a Web site does not pay to use a proxy cache.

3. Performance Modeling and Web Caching Grids

We propose a *Web caching Grid* in which remote servers on a Grid may function as caches when request rates are high and quality of service (QoS) for response times are in danger of degradation. As illustrated in Figure 1 in Section 1, the fraction of the network bandwidth that can be offloaded from a typical Web server is high. Traffic can be appropriately monitored in order to determine when requests should be offloaded to remote Grid caches. Grid caches would typically store static Web data which does not need to be encrypted. That way, security is preserved, and sophisticated back-end processing is not required for Grid caches to serve data, while response time goals can be maintained.

In this section, we are concerned with the following scenario. A high volume Web site customer can dynamically configure servers (i.e. provision additional servers) with sufficient advanced warning (i.e. minutes to hours) that the demand for capacity is increasing. In order to generate this advanced warning we need an on-line load measurement system, a real time prediction engine, and an event generation mechanism. The addition of forecasted data has the additional benefit of decreasing the frequency with which provisioning, quite possibly an expensive task, occurs since our decision window has increased.

Consider the scenario shown in Figure 3. Here we have a load balancer distributing Web requests to multiple nodes in a Web serving cluster. More precisely, in the N node cluster, m nodes comprise the content server, and $N - m$ nodes act as caching proxies for the server. Furthermore, client machines not included in the cluster, i.e. nodes on the Grid, have registered with the cluster management server to also be proxies for the Web server. We envision that such registration would utilize OGSA-based interfaces and infrastructure.

During normal, or, more precisely, low-load operation, all requests are routed to the configured Web server. Once an overload of requests is predicted, where overload has been defined using known server capacity, forecasted arrival rates, and a specified quality of service (QoS), an event

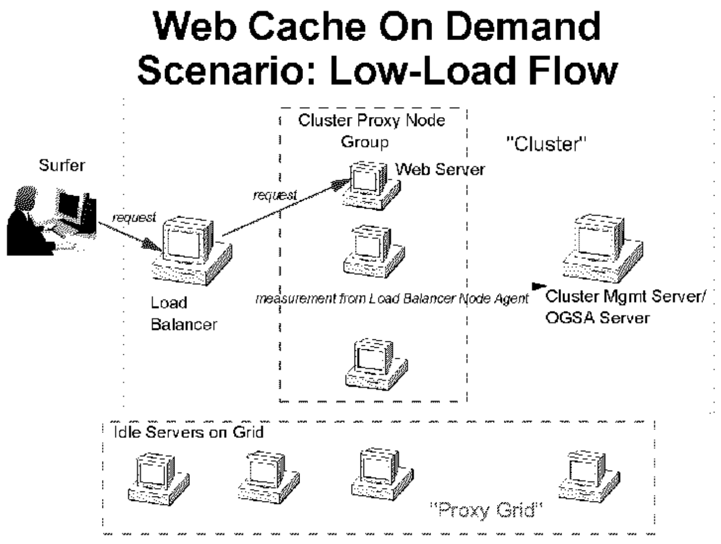


Figure 3. Under low and moderate loads, all requests are sent to Web servers.

is sent to the cluster management server. This predicted overload event is forwarded to a provisioner which can then add a proxy to the cluster (via node and load balancer configuration) to accommodate the forecasted demand without risking the QoS for incoming requests. This scenario is shown in Figure 4.

If all of the proxies in the cluster have been provisioned and increased demand is again predicted (a cluster overload condition), the provisioner now moves to contacting registered proxies in the virtual organization, or Grid, once the overload event is received. This scenario is shown in Figure 5.

As was stated previously, a key component to the caching Grid is on-line modeling and prediction. In the caching Grid that we have built we utilize the eModel tool to fulfill this necessary function. In the following paragraphs we briefly summarize the eModel tool.

In autonomic computing, where each system component monitors its own performance data, and accordingly takes necessary actions, (making it self-healing, self-configuring and self-optimizing) closer integration of various analytic tools for modeling its online performance with the underlying runtime systems is a requirement. In contrast, traditional focus on tooling has been on developing sophisticated off-line or on-line tools capturing as much of the details of an environment as possible ([5], [20]). Integrating these types of tools as run-time *components* of a system de-

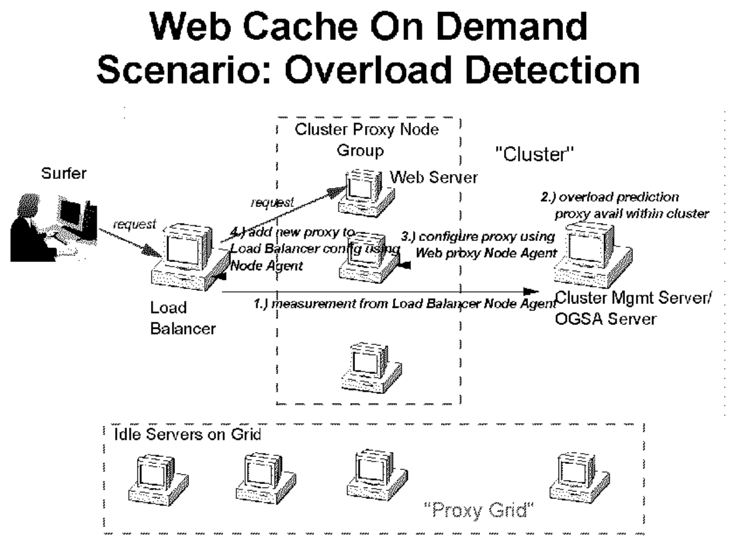

Web Cache On Demand
Scenario: Overload Detection
Cluster Proxy Node Group
"Cluster"
Surfer
Web Server
request
request
2.) overload prediction proxy avail within cluster
4.) add new proxy to Load Balancer config using Node Agent
3.) configure proxy using Web proxy Node Agent
Load Balancer
1.) measurement from Load Balancer Node Agent
Cluster Mgmt Server/ OGSA Server
Idle Servers on Grid
"Proxy Grid"

Figure 4. Handling overload when a proxy is available within the cluster.

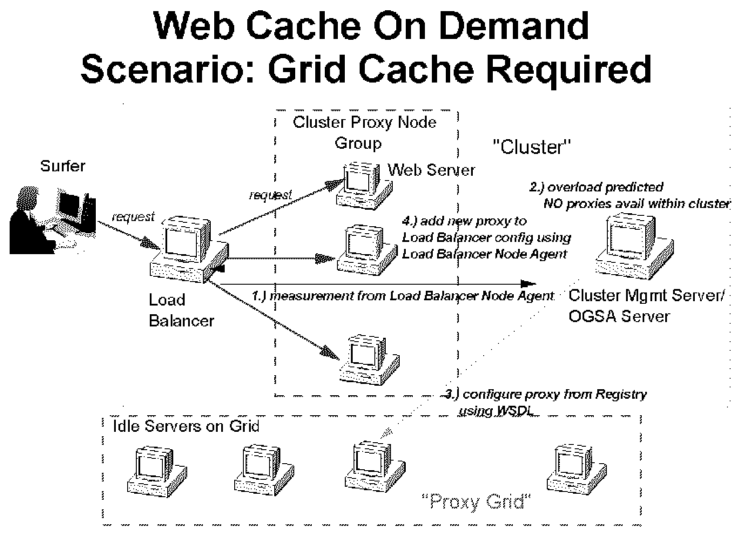

Web Cache On Demand
Scenario: Grid Cache Required
Cluster Proxy Node Group
"Cluster"
Surfer
Web Server
request
request
2.) overload predicted NO proxies avail within cluster
4.) add new proxy to Load Balancer config using Load Balancer Node Agent
Load Balancer
1.) measurement from Load Balancer Node Agent
Cluster Mgmt Server/ OGSA Server
3.) configure proxy from Registry using WSDL
Idle Servers on Grid
"Proxy Grid"

Figure 5. Handling overload when a Grid cache is needed.

mands that we pay as much attention to the integration framework as to the models themselves. Ease-of-use continues to remain an important issue, however, not necessarily only as an issue in running the tool by a human operator but also as an issue in setting up the tool for system development/deployment. Here we introduce a flexible architecture and corresponding tool implementation, *eModel*, for such a modeling framework and demonstrate a usage scenario using a Web caching Grid.

An autonomic system component typically monitors and reconfigures itself to comply with service level agreements (SLAs see [16]for an example SLA specification) on its usage, established with the clients of this system. SLAs are established by clients with the service systems in order to receive a guarantee on various service level objectives, e.g., average response time for supported throughput level during certain time periods, availability of services, etc. The eModel architecture is used during all phases of SLA life-cycles: creation, deployment and runtime monitoring and enforcement. In order to establish an SLA, a client needs an understanding of its expected workload, perhaps predicted from past workload history. The data may be available in predefined formats (i.e., as a file or database table). However, when such data are not available *a priori* or an SLA needs to be renegotiated to reflect changing needs, the data may be collected from a running system. As the data collection occurs, workload models are built, and new SLAs can be constructed. We refer to this specific usage of eModel as an SLA Advisor. From a service provider perspective, during deployment and/or for making a commitment to a client SLA, it needs to understand its available capacity, and analyze its risk in accepting this SLA. The eModel tool framework can also be used as a Risk Analyzer. During actual service invocation, the eModel framework can be used to measure and monitor runtime performance, and predict potential violations of service level objectives. We refer to this usage of eModel framework as an SLA Monitor. Note that sophisticated SLA monitoring may involve both computation of aggregated run-time parameters via metric composition (e.g., computing average from individual response times) and/or online prediction of future values of composed or component parameters. The eModel framework can also be used to further monitor an individual or a collection of resources, to watch for (current or predicted) problem states, e.g., high utilization, system bottlenecks, etc. We will refer to this use of eModel framework as a Resource Monitor. Finally, observed service performance data and/or observed customer workload can be used to adjust risk analysis, and/or to trigger renegotiation of existing SLAs.

In all of these scenarios, the overall analysis/modeling tool needs to support the following features. First and foremost, it needs to provide ease-of-use in setting up the modeling framework, where a model may be a single plug-in or composed from a collection of plug-ins in tandem. From the point of view of on-line integration of this modeling framework with varied data sources (e.g., database tables, online calls to runtime systems, etc.), we need flexible ways of specifying data sources to be used and their access details. Similarly, the eModel framework needs to support flexible ways of delivering analysis data to other runtime system components (e.g., via database, direct calls, etc.). A detailed set up describing the usage of specific model(s) can be specified declaratively using an XML document, which can be created via a GUI. Then a runtime program can parse the XML document and create the corresponding on-line modeling structure and appropriate hooks into measurement and management tools.

A complete review of the eModel framework design and architecture can be found in [6]. Here, we briefly summarize the eModel features which include:

1. a Java and XML based architecture for portability;
2. a Java API for user defined measurement collection (on- or off-line), model functions, and application or system management interfaces (e.g. events, logging, etc.);
3. a run-time manager consisting of an object based container and a thread pool (for multiple workload tracking and modeling instances);
4. a set of static classes and corresponding API for access to measured data and predicted values either from run-time cache or the persisted database; and
5. a plotting tool for real-time visualization of data, predictions and triggered events.

In the remainder of this section, we describe a sophisticated modeling methodology that could be employed within the eModel framework for our Web caching Grid.

Under the general Grid architecture, the dynamic and heterogeneous nature of the available resources makes it a challenging task for the provisioner to make the resource allocation related decisions such as determining the number and type of Grid machines to allocate for the requested service. The underlying mathematical problem is fundamentally difficult due to the complex request arrival patterns and diverse service mechanisms. We need to apply sophisticated statistics and modeling techniques for analyzing the request arrival patterns to the system, forecasting how these arrivals will change over time, and constructing system models for the request service processes on different platforms. COMPASS is a set of tools to help address these issues based on advanced statistics, stochastic processes, queueing, control and optimization theories. COMPASS stands for Control and Optimization based on Modeling, Prediction and AnalySiS. Its main functions include workload characterization, system and application modeling, automatic model building and on-line optimal control.

Request arrival information is passed on to the workload characterization module, which makes predictions of the Web request process based on the past access patterns and the constantly changing volume using time series models. Furthermore, the key characteristics that have strong impact on the server's performance are also extracted by the workload characterization module. Studies [22, 21]have shown that the correlation characteristics such as short-range and long-range dependence have significant impact on the response time measures. So does the burstiness characteristics such as the variability and heavy-tailness of the request distributions. The response time measures under long-range dependent and heavy-tailed request processes can degrade by orders of magnitude, and have a fundamentally different decay rate compared with traditional Poisson models. These key parameters including the correlation factor, the marginal distributions, the detected user access patterns, the visit page sequences and think times, and the various matching distribution parameters are calculated by the workload characterization module to establish a complete workload profile to be used as the input to the other modeling modules.

A common approach to model the request serving process in many service systems is to build queueing models [14]. As the user behaviors and the Web service functions become more and more complex, so are the structures of the Web systems. To model how the customer requests, or *transactions*, are served by such complex systems, a single server queue is far from adequate. The system and application module constructs flexible queueing network models to capture the Web serving process. Each of the multiple components within the server system can be represented as a queue or a more complex sub-queueing network. For example, one can use a queue to model the network component within a system, and use a single server queue to model a database, etc. Different routing mechanisms such as round robin and probabilistic routing, and different service disciplines such as processor sharing or priority policies can be used for each queueing or server component within the model to mimic the component's service behavior. Users can be categorized into multiple classes based on their access behaviors. For given routing and service parameters of such a queueing system, the system and application modeling module readily obtains the performance related measures such as throughput, utilization and response times, by simulations and queueing network theories. The workload profile into the system can be the original as well as the forecasted profile from the workload models. Using markup languages, the constructed models can be easily described and customized for the many different platforms within the heterogeneous environment.

One set of crucial parameters of such queueing network models is the set of service time requirements for different job classes at each server in the model. These parameters can be calibrated ahead of time for different platforms. This process may require considerable time and experience and is difficult to automate. Recent research [25]has provided a general methodology to infer these per-class service time parameters at different servers based on the server throughput, utilization and the per-class end-to-end response time measurements. The service time parameters are the solution to an optimization problem with queueing-theoretic formulas in the objective and constraints. This is the main function provided in the automatic model building module.

Based on the appropriate data of sufficient detail and accuracy, we can construct workload, system and performance models automatically and integrate to complete the control loop. To achieve the given QoS objective, the on-line optimal control module activates a controller to dynamically change the scheduling and resource allocation policies within the servers based on these models. Furthermore, the optimization functions in the modeling modules map the given QoS requirements into the most cost and operation efficient hardware and software configurations. The results of these actions are reflected from the monitored performance measures. These performance measures together with the changing workload will again influence the control decisions. With all these functions in place, the system is empowered with self-managing capabilities.

4. Case Study: Grid Computing in Finance

In this section, we explore a particular aspect of commercial applications which are common in finance-oriented environments. In particular, we will discuss the demands of applications used in investment banking, such as portfolio pricing and portfolio optimization. These applications involve mathematical operations which are also common in the field of scientific and technical computing, such as Monte Carlo optimization and stochastic modeling. However, there is one key characteristic which is special to the field of finance: the sensitiveness to the response time. In this section we argue that the traditional architecture used in scientific and technical computing is not designed to support the response time requirements in the technical field, and suggest an alternative architecture that is more suitable to applications which are extremely sensitive to the response time.

The main requirement that drives Grid applications in investment financing can be simply stated: minimal response time. This is a key motivation for moving finance applications to the Grid. The gain in a few seconds in response time represents a significant advantage over a competitor. Therefore, the goal of Grid-enabled applications in finance is to reduce the response time. This goal is usually achieved using code parallelization. Fortunately, most of the mathematical operations used in investment banking are inherently parallelizable. Each portfolio is typically an aggregation of individual items which can be used as parameters to independent calculations. This is usually done using a scatter/gather model, in which operations are scattered to several nodes in the Grid and then a master node performs a gathering operation, and combines the partial results into a portfolio value. Using this technique it is usually possible to reduce the response time by a factor linear to the number of nodes. Linear scaling means that a result that took 60 seconds using one node may take only 6 seconds using 10 nodes. This represents a huge advantage in real time trading.

We now turn our attention to the ways that are available for scheduling the kind of parallel operations just discussed. In traditional high performance computing the parallel scheduling architecture is batch oriented. Consider, for example, that the Globus Toolkit offers several job managers for batch schedulers, such as Load Leveler, Condor, LSF and PBS. This is because Globus was initially developed by the scientific and technical community. Most of the technical applications are batch oriented and are not very sensitive to response time. Also, most units of work (batch) in scientific and technical computing are generally long (several hours or more), and so the response time can be significantly longer than in financial computing..

The batch computing model has a major disadvantage when dealing with the class of short lived operations just described. One such performance impact is the cost of launching and executing a new set of parallel processes for every request. The task of launching a parallel process is very costly, involving the cost of spawning new process instances, the cost of the initialization of the application code itself, and may also possibly involve security-related costs, such as user authentication. In order to avoid such costs, it is imperative that the application code be maintained loaded in the memory of each Grid node, running as a persistent process. Another advantage of using persistent processes is the fact that they provide persistent state for the application. This persistent state can be used, for instance, as a way of caching intermediate results in a pipelined operation. Such pipelined operations (one in which the results of a previous computation are used as input for another computational step of the pipeline) are very common in investment banking applications. The batch programming model is unable to take advantage of persistent runtime state, and is also not able to be used by interactive applications.

In order to overcome the limitations of the batch programming model, it is necessary to deploy persistent applications, or services. One of the major features of the new Open Grid Services Architecture (OGSA) is the support of persistent Grid services. The OGSA architecture defines a specific interface for lifecycle management of such services which is designed for the management and control of persistent services. The lifecycle interface defines the rules for service instance creation and destruction, which are essential for the management of persistent service instances. This architecture enables a programming model which offers a major improvement over the traditional batch processing model: interactive, persistent parallel services.

The remainder of this section proposes an architecture for deploying persistent parallel services in a Grid. This architecture is implemented in a Grid scheduler prototype we have developed called Topology Aware Grid Services scheduler (TAGSS). The TAGSS architecture is derived from concepts introduced by the OGSA standard. The OGSA standard mandates that Grid services implement the mentioned life cycle interface, which specifies a creation method, also called the factory interface. The TAGSS architecture defines a basic service, which is the TAGSS Container Factory. The TAGSS container factory can construct individual TAGSS containers, or collections of containers named Grid Container Arrays. The Grid Container Array is also a persistent Grid Service, which is in turn a factory for Grid Object Arrays. The Grid Object Array is the main construct which exports the scheduling functionality of the TAGSS architecture.

The Grid Object Array is a construct similar to an object in object-oriented languages such as Java. The difference is that a method invocation on a Grid Object Array results in method invocations in each object in the array. In order to pass the arguments for the method invocation on a Grid Object Array, an auxiliary data structure called the Grid Data Set is used. The Grid Data Set is basically a matrix where each row corresponds to an argument list for a method invocation. There are three different method invocation semantics which can be used when invoking a method on a Grid Object Array:

1 MULTICAST: in this mode the Grid Data Set contains one row, and the method invocation is done using the arguments in the single row to all objects in the Grid Object Array. In other words, all objects execute the same method with the same arguments. This mode is used to synchronize state in the objects in the array.

2 ANYCAST: in this mode the Grid Data Set typically contains many more rows than the number of objects in the Grid Object Array. For example, this mode can be used with a Grid Data Set of thousands of rows and a small number of objects in the array. The semantics of ANYCAST dictate that it does not matter which object works on a particular row of the Grid Data Set; any object can work on any row. This method of execution can be used to implement the typical scatter/gather function which is of particular interest in investment banking, as discussed at the end of the section.

3 BARRIER ENFORCED METHOD INVOCATION: this mode of method invocation specifies that there is a one to one correspondence between each row of the Grid Data Set and each object. The objects in the Grid Object Array are indexed, and row zero on the Grid Data Set is used to invoke a method on object zero, row one on object one and so forth. This method is useful when all objects have to complete a specific operation with different arguments, and the invocation is synchronized by a barrier condition.

Figure 6 depicts the basic elements of the TAGSS architecture.

The TAGSS architecture has a unique feature which makes it particularly suitable for Grid deployment. When a Grid Container Array is created, one of the containers is chosen as a coordination point; it creates a small microscheduler object called the Grid Container Array controller. This object is not directly seen or controlled by the client application, and it has the function of enforcing the proper method invocation semantics described above. The microscheduler is especially important in the realization of the ANYCAST method invocation mode, because this mode requires the monitoring of the completion of tasks (rows in the Grid Data Set) and assigns new tasks (rows) to the objects in the Grid Object Array. This component actually implements a real time scheduling environment.

The TAGSS architecture is well suited for the Grid for a number of reasons. In the first place it creates a distinct Grid Container Array for

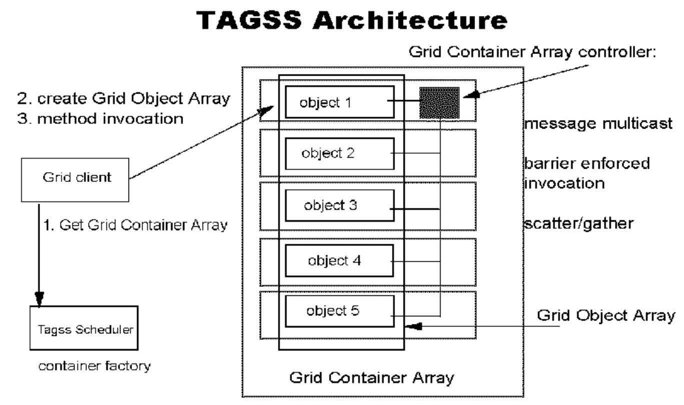


Figure 6. The TAGSS architecture.

each client, and therefore each client has its private scheduling domain. This is consistent with the Grid architecture, because the set of resources available to any given user is determined by the privileges associated with the user's Grid certificate. Therefore, the resource set for any given user is potentially distinct, and the scheduling domain is equally distinct. This aspect makes the deployment of a centralized scheduling architecture particularly difficult, because the scheduling procedure would have to consider a different set of resources and constraints for each user. The TAGSS architecture, on the other hand, deploys a microscheduler for every Grid Container Array. It can therefore be considered a fully distributed scheduling architecture, which is more suitable for deployment in the Grid. Another reason that makes the TAGSS architecture appropriate for Grid deployment is that it is designed to be used for the deployment of persistent stateful services, which, as mentioned in the beginning of this section, are necessary to fulfill the minimal response time requirements which are important in commercial Grid applications, and in particular to applications in the field of investment banking.

5. Conclusion

We have presented several issues related to commercial applications of the Grid. Web serving is an example of an application which can benefit from the Grid. We presented a caching Grid which offloads traffic to remote caches when servers become overloaded. Unlike prior caching techniques such as proxy caching and content distribution networks, Grid caches are only used when servers become overloaded.

We described techniques for performance and traffic modeling which can be applied to Grid caches. We also discussed financial applications of Grid computing. A key requirement for financial applications is to have short response times. We presented a scheduler for Grid applications which is specifically targeted to applications which require fast response times.

References

[1] R. Agarwal et al. High Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2. *IBM Systems Journal*, 34(2):263–272, 1995.

[2] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. Sp2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[3] T. Brisco. DNS Support for Load Balancing. Technical Report RFC 1974, Rutgers University, April 1995.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language 1.1, March 2001. http://www.w3.org/TR/wsdl.

[5] Peakstone Corporation. Peakstone eAssurance^{TM} eBusiness capacity management product features, descriptions & benefits. http://www.peakstone.com/pdf/FDB.pdf.

[6] Catherine H. Crawford and Asit Dan. eModel: Addressing the need for a flexible modeling framework in autonomic computing. Research Report RC 22464, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, May 2002.

[7] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[8] T. Dierksand and C. Allen. The TLS Protocol (RFC 2246). http://www.ietf.org/rfc/.

[9] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.

[10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed systems integration. *IEEE Computer*, 35(6):37–46, June 2002.

[11] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002. http://www.globus.org/research/papers/ogsa.pdf.

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

[13] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), March/April 2000.

[14] L. Kleinrock. *Queueing Systems, Volume II, Computer Applications*. John Wiley and Sons, 1976.

[15] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.

[16] H. Ludwig, A. Keller, A. Dan, and R. King. A service level agreement language for dynamic electronic services. In *WECWIS*, June 26-28 2002.

[17] G. Pfister. *In Search of Clusters*. Prentice Hall, 1998.

[18] E. Resorla. HTTP Over TLS (RFC 2818). http://www.ietf.org/rfc/.

[19] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM 2001*, 2001.

[20] Sodalia. NetTraffic^{TM}. http://www.sodalia.com/pdf/nettraffic.pdf.

[21] M.S. Squillante, D.D. Yao, and L. Zhang. Web traffic modeling and web server performance analysis. In *IEEE Conference on Decision and Control (CDC)*, 1999.

[22] M.S. Squillante, D.D. Yao, and L. Zhang. *System Performance Evaluation: Methodologies and Applications*, chapter Internet Traffic: Periodicity, Tail Behavior and Performance Implications. CRC Press, 2000. E. Gelenbe, book editor.

[23] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid service specification. In *Open Grid Service Infrastructure WG, Global Grid Forum*, July 2002.

[24] W3C. Web services. http://www.w3.org/2002/ws/.

[25] Li Zhang, C. H. Xia, Mark S. Squillante, and W. N. Mills III. Workload service requirements analysis: A queueing network optimization approach. In *Tenth IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2002.