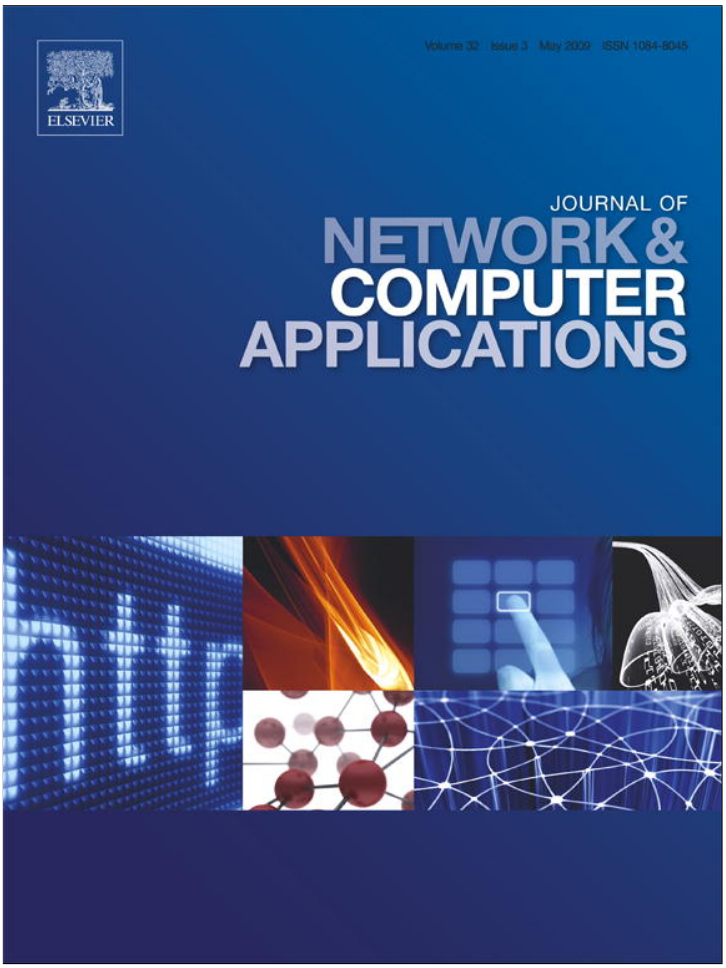

Journal of Network and Computer Applications 32 (2009) 510–524

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Adaptive middleware supporting scalable performance for high-end network services

Byoung-Dai Lee [a,*], Jon B. Weissman [b], Young-Kwang Nam [c]

[a] Next Generation Terminals Lab., Samsung Electronics Co., Ltd., Korea
[b] Department of Computer Science and Engineering, University of Minnesota, Twin Cities, USA
[c] Department of Computer Science, Yonsei University, Korea

ARTICLE INFO

Article history:
Received 9 May 2008
Received in revised form
24 July 2008
Accepted 13 November 2008

Keywords:
Network service
Resource management
Adaptivity
Grid

ABSTRACT

Network service-based computation is a promising paradigm for both scientific and engineering, and enterprise computing. The network service allows users to focus on their application and obtain services when needed, simply by invoking the service across the network. In this paper, we show that an adaptive, general-purpose run-time infrastructure in support of effective resource management can be built for a wide range of high-end network services running in a single-site cluster and in a Grid. The primary components of the run-time infrastructure are: (1) dynamic performance prediction; (2) adaptive intra-site resource management; and (3) adaptive inter-site resource management. The novel aspect of our approach is that the run-time system is able to dynamically select the most appropriate performance predictor or resource management strategy over time. This capability not only improves the performance, but also makes the infrastructure reusable across different high-end services. To evaluate the effectiveness and applicability of our approach, we have transformed two different classes of high-end applications—data parallel and distributed applications—into network services using the infrastructure. The experimental results show that the network services running on the infrastructure significantly reduce the overall service times under dynamically varying circumstances.

1. Introduction

Recently, due to the availability of high-speed networks and advances in packaging and interface technologies, there has been considerable interest in transforming the high-performance applications into ***network services***, thus promoting both software and hardware sharing (see Fig. 1). Providing efficient, reliable, secure, and scalable services requires a significant run-time infrastructure. Resource management is an essential part of the infrastructure because effectively allocating resources across competing service requests directly relates to how fast service request can be processed. To deliver high performance, the infrastructure must be able to adapt to dynamically varying circumstances (e.g., resource availability, user demands). Furthermore, we believe that high-end network service itself must also be adaptive with respect to system resources, which we call ***service-level adaptivity***. For example, in order to balance resource allocation, resources must be dynamically shared among competing service requests. Adaptivity is also useful for keeping the infrastructure as general as possible. For example, instead of hardwiring a certain technique or policy into the system, by enabling the system to select the most appropriate ones dynamically over time, the system not only provides good performance, but also makes itself as general as possible.

In this paper, we show that an adaptive, general-purpose run-time infrastructure supporting effective resource management can be built for a wide range of high-end network services running in a single-site cluster and in a Grid. Among many of challenging research issues such as reusability, reliability, and security, we focus on the development of a general-purpose run-time infrastructure supporting scalable performance as a fundamental, yet important component necessary for transforming various high-performance applications into network services.

The approach we have taken to develop such infrastructure is to first identify the characteristics of each high-end network service class and then to divide them into service-dependent and service-independent parts. For the service-dependent part, we simply provide interfaces and templates so that the service providers can implement them for their needs. An example of a service-dependent part would be adaptation routines for service-level adaptivity. When resources are added or removed in the middle of service execution, the adaptation routine stores and redistributes the states of the network service to continue execution, which differs from one service to another. For the service-independent part, we provide reusable middleware, software components, and libraries. One of the core functionalities

* Corresponding author.
E-mail addresses: byoungdai.lee@samsung.com (B.-D. Lee), jon@cs.umn.edu (J.B. Weissman), yknam@yonsei.ac.kr (Y.-K. Nam).

1084-8045/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jnca.2008.11.002

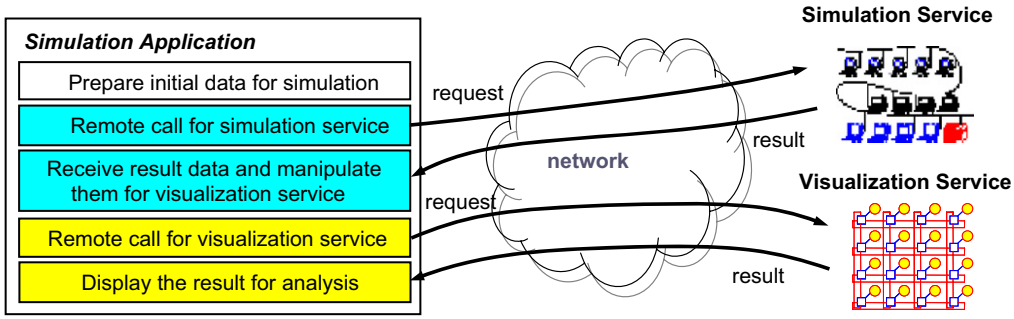

Fig. 1. Network service-based computation: complex and time-consuming parts of an application can be replaced with remote services; thus, application development can be easy. In the above example, the simulation application simply plays the role of a container for remote service calls.

provided by the middleware is adaptive resource management for service-level adaptivity. For example, the middleware makes decisions on the candidate requests that the resources be taken away to achieve the performance objective. Throughout this paper, the performance objective is to minimize the overall service time because it is a representative metric for users to evaluate the quality of the service. When making adaptation decisions, performance gain and loss, as well as the overhead due to adaptation, must be taken into account. Therefore, performance information is crucial to adaptive resource management. As an essential component of reusable middleware, we propose a novel technique for performance prediction, called history-based dynamic performance prediction, which can be applied across different network services.

For high-demand network services, the Grid (Global Grid Forum) is an ideal environment to provide scalable performance. Resource management for network services running in a Grid becomes more complex because resources are distributed and heterogeneous. The presence of multiple sites, each offering heterogeneous resources, implies that the first job of resource management is to select an appropriate site for handling the service request. The novel aspect of our approach is to allow the system to select appropriate resource selection policies dynamically over time, based on past performance history.

Overall, the contributions of this paper are two-fold. First, we have developed several adaptive algorithms and techniques allowing the run-time infrastructure to adapt itself under dynamically changing circumstances and to be reusable across different high-end network services. Second, we have completed the implementation of the run-time infrastructure and have deployed two prototypical services in a single-site cluster and a Grid. They are an N-body simulation service and a library-to-library gene sequence comparison service.

The organization of this paper is as follows. Section 2 presents related work in existing middleware solutions for high-end network services and a variety of work on resource management for network services. Section 3 describes the service and resource sharing models. Section 4 presents the proposed system architecture in detail, and Section 5 describes the adaptive multi-level resource management techniques. Section 6 describes two prototypical services and experimental results obtained from those services. Finally, Section 7 provides a summary and proposal for future work.

2. Related work

2.1. Existing middleware solutions

Many groups have built their own infrastructures for providing domain-specific high-end network services. NetSolve (Casanova and Dongarra, 1997) provides complete run-time infrastructure, as well as server management tools and client interfaces to C, Fortran, Java, and MATLAB. PUNCH (Kapadia and Fortes, 1999) is a network-based computing system that allows users to run unmodified software application via standard World Wide Web browsers. OGSA (Foster et al.) is built upon two core technologies: the Globus toolkit (The Globus Toolkit) and Web services (Vaughan-Nichols, 2002). It defines uniformly exposed service semantics, defines standard mechanisms for creating, naming, and discovering transient service instances, provides local transparency and multiple protocol binding for service instances, and supports integration with underlying native facilities. Along with these above-mentioned systems, several service-based architectures for Grid computing environments have been proposed, such as CCA (Armstrong et al., 1999), XCAT (Govindaraju et al., 2002), Partitionable Services (Ivan et al., 2002), H_2O (Sunderanm and Kurzyniec, 2002), and Service Grid (Weissman and Lee, 2002), to name a few.

2.2. Performance prediction

Application run-time information is a fundamental component for resource management in Grid environment, as well as most application and job scheduling approaches for parallel and distributed systems. In order to achieve accurate predictions of the application run-time, many conventional approaches used in scheduling research (Maheswaran and Siegel, 1998; Subramani et al., 2002; Takefusa et al., 2001; Vadhiyar and Dongarra, 2002; Weissman, 2002) assume that accurate performance models are available that include low-level details of the applications (e.g., operation counts, message size). Kapadia et al. (1999) use three instance-based learning algorithms to predict resource usage for each application run. In their approach, instead of using static performance models, subsets of past performance information are used for performance prediction; however, they do not consider parallel applications. Furthermore, they implicitly assume that system- and application-specific information does not change significantly. Smith et al. (1999) propose a run-time prediction technique for parallel applications using historical information. They use a greedy search and a genetic algorithm search to find the past history that has strongest similarity to the current job.

2.3. Resource management in Grids

Significant research has been conducted in the area of resource management in Grids. In AppLes project (Berman et al., 2003), each application is integrated with its own AppLes agent, which uses the performance model and dynamic information regarding resources to predict the run-time of its application on a given set of resources. Among a set of available possible candidate

schedules, AppLes agent selects the one that is predicted to provide the best performance. Condor matchmaking (Raman et al., 1998) uses a semi-structured data model—ClassAd data model—to describe resource properties. A matchmaking service matches ClassAds in a manner that satisfies the constraints specified in both ads and informs the relevant entities of the match. Ernemann et al. (2002) introduces an adaptive multi-site scheduling algorithm for fragmentable jobs. Given a job with its requested number of resources, the algorithm decides the number of fragments of the job that will be executed on multiple sites in parallel to minimize the number of fragments. The basic idea of the dynP scheduler (Streit, 2002) is to change between different scheduling policies over time. Although dynP shows the feasibility of dynamic policy switching, it can achieve improved performance only when the accurate information of run-times of jobs are available.

As Grids become more mature and popular, one promising resource management policy is driven by a real-world economic model (Buyya et al., 2002; Carman et al., 2002; Grid Economic Services Architecture; Keller et al., 2002; Wolski et al., 2001). The economic model complements our approach since it can be used to implement service-level adaptivity. For example, in the case of requests that paid "more", the system may take resources more aggressively from the requests that paid "less".

The difference between our approach and the above-mentioned work is two-fold. First, our solution supports service-level adaptivity to serve multiple concurrent service requests. Second, by enabling the run-time system to choose the most appropriate techniques or strategies over time, our solution not only improve the performance, but also makes the resource management system to be reusable across different high-end network services.

3. System model

3.1. Service model

Virtually everything, ranging from simple content-delivery to scientific and engineering computations to the resource itself can be offered as network services. In this paper, however, we consider *stateless*, *non-composite application services* that demand significant computational resources because many of the existing high-performance applications belong to this class. In addition, datasets that are required to process user requests are assumed to exist at the same place that the services are running. This assumption is reasonable because in many cases, the service providers may not want to distribute the datasets across the network for security reasons.

"Stateless" means that the network services do not maintain complete or partial results between service invocation. The output of the network service depends only on user-supplied information (e.g., input parameters to the service). "Non-composite" services are those services that do not require invocation to other services. Resource management for composite services becomes more complex because the relationship to other services, as well as individual services, must be considered. Developing effective resource management for composite services is one of our future research objectives.

When a user submits a request to a network service, a service instance will be created and will execute the service code on behalf of the user. The run-time infrastructure of the network service will assign the necessary resources before the instance starts executing, and the assignment typically remains unchanged until the instance finishes the user request. However, in order to balance the resource allocation among competing service requests, we believe that service-level adaptivity must be supported. Service-level adaptivity enables computational resources to be dynamically added to/removed from the service instances while they are running. For example, if there are insufficient resources available when a new request arrives, the resources that are being used by the active service instances may be taken away and allocated to the new service request in order to enable it to start immediately. Although service-level adaptivity looks promising, it introduces additional overhead to the service time. For example, when resources are removed, the states of the processes running on those resources may need to be saved and redistributed to the remaining resources. During the execution of the service instance with input parameter, $\vec{P}$, if the number of resources changes from N_1 to N_2 to, …, N_k, the service time with service-level adaptivity is defined below:

$$T_{service}(\vec{P}, N_1, \ldots, N_k) = T_{comm} + T_{wait} + \sum_{i=1}^{i=k} T_{run}(\vec{P}, N_i) + \sum_{i=1}^{i=k-1} T_{overhead}(N_i, N_{i+1}) \quad (1)$$

where T_{comm} is the data communication time between the users and the network services, T_{wait} the period of time spent in the wait queue until enough resources are available, $T_{run}(\vec{P}, N)$ the time consumed to process the service request, $T_{overhead}(N_i, N_{i+1})$ the overhead due to change of resources from N_i to N_{i+1}.

In what follows, we do not consider T_{comm}, since we assume that the computation time of the service (e.g., T_{run}+T_{wait}) is a dominant component of the service time. Due to the overhead (see Eq. (1)), not every class of high-end network services are suitable for adaptation. For example, those services with complex workflows may suffer from significant overhead for data collection and decomposition. For such services that cause excessive overhead for service-level adaptation, other resource management schemes are more appropriate; consequently, it must be the run-time system's responsibility to select the appropriate resource management schemes for the given services.

3.2. Resource sharing model

Executing high-end services requires a large amount of resources, including processors, disk storage, and networks. In this paper, we limit our interest to managing computational resources (e.g., processors) because in our service model, we assume that the computation time of the network service is a dominant component of the service time. Network services can be deployed on a variety of platforms, but we are especially interested in a cluster setting and Grids. As technology advances, powerful but inexpensive processors are commonly available. A cluster setting of such processors interconnected with a faster network has become an attractive platform to meet both price and performance requirements. We assume that a cluster is homogeneous in that all resources of the cluster are of the same type (e.g., all nodes have processors of the same architecture, the same amount of memories and disk spaces). In our model, a Grid is comprised of distributed, logical sites, each offering a homogeneous cluster. Clusters at different sites, however, may be heterogeneous. Each logical site contains only one cluster; for example, if a site contains two clusters, it is viewed as two logical sites, each of which provides a cluster.

Computational resources on which the network services will run can be categorized into different classes based on the following criteria:

(1) Degree of multiprogramming—*space-shared/time-shared*
On space-shared resources, once resources are assigned to a service instance, those resources are available only after the service instance finishes its job. On the other hand,

time-shared resources allow multiple service instances to run on the resources simultaneously.

(2) Existence of external load—*dedicated/non-dedicated*
On dedicated resources, only service requests compete for resources while on non-dedicated resources, other users are allowed to use the resources as well.

In this paper, we assume that clusters are space-shared and dedicated. We believe that this assumption is reasonable because service providers want to have complete control over their resources to provide guaranteed performance to their users. Furthermore, the resources provided by many of production Grid systems are space-shared; once the resources are allocated to users, they are dedicated until the jobs are finished. In order to support time-shared and non-dedicated resources, the resource management systems may use existing resource monitoring systems, such as Ganglia, Network Weather System (Wolski, 1998), and Remos (Dinda et al., 2001) to predict short- or medium-term resource availability.

4. System architecture

The run-time infrastructure that we propose is an open architecture and it consists of two primary components: ***a service manager*** that manages the network service and ***a request manager*** that manages each user request (see Fig. 2). The service manager keeps track of the current requests for the service, gathers incremental performance updates from the request managers, and makes resource allocation decisions on behalf of the user requests within its resource pool. When a service request arrives, *the local scheduler* of the service manager decides how many resources will be assigned to that request based on the scheduling policy it employs and *the service instance implementor* instantiates the request by creating a request manager. The service manager is also responsible for making adaptation decisions for service-level adaptivity. Once the request manager is created, the service manager passes the information regarding the selected resource sets as well as the input parameters supplied by the user. The request manager then creates corresponding processes on those resources, and the service instance starts executing the service code. Finally, when the service instance is completed, the request manager collects the results and sends them directly to the user. The request manager is a front-end linked in with the service code and is the place where service-level adaptation, such as resource addition and resource removal, are actually implemented. In addition, when a service instance is finished, the request manager associated with the instance reports the performance of the request to the service manager so that the service manager can maintain up-to-date performance information. To support service-level adaptivity, the service code must be linked with the adaptation routines of adaptive code libraries. Although the implementations of both the service code and the adaptive code library must be provided by service code developers, the interfaces of the functions in adaptive code library are pre-defined since the functions are called by the middleware.

For high-demand network services, we envision that it will be often the case that the network service is installed at multiple sites so that each participating site can handle different user requests. We label such network services as ***Grid-enabled network services.*** The novel aspect of our system is that it can be extended with ease to accommodate Grid-enabled network services. As shown in Fig. 2, the network service is installed at M sites, each of which is running the middleware that manages a cluster within the site, and one of the site is selected as *a home-site*, where users can submit service requests. An important role of the home-site service manager is to redistribute the service requests to appropriate sites to actually handle the requests and, therefore, it maintains additional middleware components, called ***the network service front-end***. *The global scheduler* of the network service front-end decides to which site it will forward the requests, while the site database maintains up-to-date status information (e.g., current queue length) of each site. When a number of requests arrive at the front-end simultaneously, the front-end may become a performance bottleneck. One of the possible approaches to address such problem is to dynamically replicate and delete the network services based on the client demands, which is not the scope of this paper.

5. Adaptive multi-level resource management

5.1. Performance prediction

Performance prediction is crucial to adaptive resource management because deciding how to best allocate resources dynamically depends on the estimation of the run-time of the service,

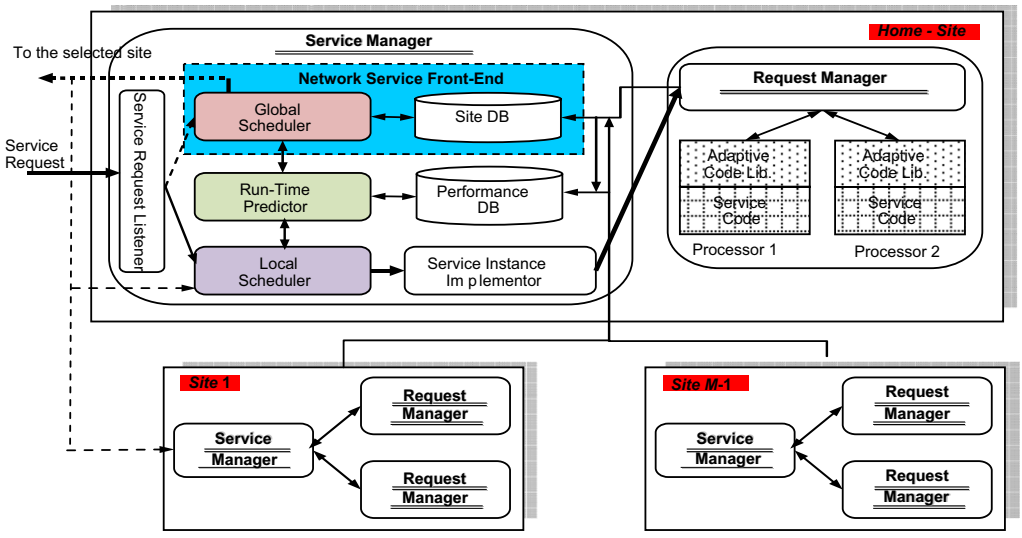

Fig. 2. System architecture: the network service is installed at M sites, each offering a dedicated cluster. Dashed lines are applied only to the home-site service manager.

which depends on not only the number of resources, but also on the input parameters to the service. For simple services, such as matrix multiplication services, it is relatively easy to acquire accurate performance models. However, in general it is not always possible to obtain such accurate performance models of complex services, as they require a thorough understanding of the service codes as well as a significant number of experiments to validate the models developed. To overcome the difficulties in generating accurate performance models for complex services, several well-known learning techniques have been developed which can be applied to performance prediction, including instance-based learning technique (Kapadia et al., 1999), global parametric learning techniques and filtering and local linear regression technique (Lee and Schopf), to name a few. However, given a service, it is difficult to know which predictor generates the most accurate predictions in advance, as it is highly dependent on the service.

The novel aspect of our approach is to allow the system to select a best predictor dynamically over time for the service deployed by using previous prediction history. The run-time history database in the service manager stores the past prediction history of each predictor, and it is organized in a two-dimensional matrix, where columns represent the performance predictors and the variables that affect the run-time, and each row represents the predicted run-times of the predictors and the actual run-time of a service instance. When the prediction is needed, the dynamic performance predictor runs all of the performance predictors in parallel that are appropriate. An appropriate predictor is one that can be applied (e.g., if the service has m dependent parameters, then the predictor must work with m parameters). The dynamic performance predictor then gathers most recent k prediction history of each predictor and selects the one that shows the best prediction accuracy. Once the request is completed, the actual run-time reported by the request manager will be stored in the performance history database, along with the predictions of all predictors selected. We used the average prediction error rate (*APE*) and the normalized prediction error rate (*NPE*) to measure prediction accuracy:

$$APE = \frac{\sum(RunTime_{measured} - RunTime_{predicted}/RunTime_{measured}) \times 100}{k}$$
$$NPE = \frac{\sum_1^k |RunTime_{measured} - RunTime_{predicted}| \times 100}{k \times AVG(RunTime_{measured})} \quad (2)$$

where $AVG(RunTime_{measured})$ is the average of k measured run-times. One of the primary benefits of this approach is that whenever a new prediction technique is developed, we can utilize the technique by incorporating it into the dynamic performance predictor. In order to add new predictors without recompiling the middleware, the meta-information regarding the predictors must be provided. The meta-information includes, for example, the number of dependent parameters required and the data types of parameters. The information, along with the pointers to the predictors, is stored in the middleware. Therefore, the predictors that will be added later are required to exist as active entities such as dynamic libraries or software components (see Lee, 2003) for more details on implementation.

5.2. Intra-site resource management

The local scheduler of the service manager is responsible for making adaptation decisions for service-level adaptivity. In order to support service-level adaptivity, both service code developer and the middleware of the network service must be involved. The service code developer should provide service-specific adaptation routines that are responsible for saving and redistributing the states of the network service, whereas the middleware is responsible for making adaptation decisions for dynamic resource sharing, which we call ***resource harvesting***. Therefore, the middleware must address several fundamental questions to support service-level adaptivity:

- How many resources should be initially allocated?
- From which service instances should resources be harvested?
- How many resources should be harvested
- Where freed resources should be allocated?

Depending on the answers to the above questions, several harvesting policies can exist. Note that the overhead of harvesting must be measured within the service and must be made available as part of the adaptation decision process. In what follows, we introduce two harvesting policies based on performance prediction.

5.2.1. Shortest-remaining time harvesting

The basic idea of shortest-remaining time harvesting (*SRT_Harvest*) is that when a new service request, S, can finish earlier than an active service instance, R, using the resources of R, then S can harvest resources from R to enable it to run. Therefore, this approach gives higher priorities to short-running jobs. The behavior of the algorithm is determined by two configurable parameters:

- *Harvesting parameter (HP)*: controls how aggressively the system can harvest resources from active service instances.
- *Wait time parameter (WP)*: determines the maximum wait time threshold for each service request; it is proportional to the minimum run-time of the request.

Before harvesting resources, the service manager contacts the request managers of active service instances to acquire the current status information as to progress. Then, for each request in the wait queue, the algorithm computes the best performance achievable using resource harvesting. The service manager can harvest resources from a request R for request S only when the following condition is satisfied:

$$EstimatedRunTime(S) \times HP < RemainingRunTime(R) \quad (HP \geqslant 1.0)$$

This condition determines both the victim requests from which resources will be harvested and the number of resources from those requests. Thus, as long as a request S can finish earlier than R, S can harvest resources from R. If not, it cannot harvest any of the resources of R. Note that the estimated run-time of S and the remaining time of R include the overhead of adding and releasing resources. If there are multiple requests in the wait queue that satisfy the requirement, the one that can finish the earliest is selected. Furthermore, if a request R has finished before a request S returns the resources harvested from R, those resources will be allocated to service instances currently running to make them finish earlier, instead of assigning the resources to requests in the wait queue. Since the algorithm favors short-running requests, long-running requests may not acquire resources forever. To prevent the starvation of long-running requests, whenever resources are available, the system checks if there are any pending request, Z, whose total wait time exceeds the maximum wait time threshold, defined by $EstimatedMinimumRunTime(Z) \times WP$. If so, resources are allocated to those requests, and they are marked as "Non-harvestable" so that no requests can take resources away from the requests.

5.2.2. Impact-based harvesting

In contrast to shortest-remaining-time harvesting, impact-based harvesting (*IB_Harvest*) focuses on resource lenders rather than resource borrowers. The resources of an active service instance, *R*, can be harvested for a new service request, *S*, only when the impact of resource harvesting that *R* will experience is below a threshold. The impact is defined in terms of the service time. In order to compute the service time threshold for each request, the algorithm uses a configurable parameter, *impact parameter (IP)*, and the optimal run-time of the request, which is defined as the minimum run-time achievable. The service time threshold for each request is defined by $EstimatedOptimalRunTime(R) \times IP$.

If there are not enough resources available for a new request, for each active service instances, the service manager computes the number of resources that can be harvested from each of them. The number of harvestable resources of each active service instance is defined as follows:

$$max\{n : ElapsedTime + RemainingTime(n) < ServiceTimeThreahold\}$$

where *ElapsedTime* is the *Current Time–Service Start Time* and $RemainingTime(n)$ the estimated remaining run-time if *n* resources are used.

The number of resources that will be allocated to the new request is the minimum number of resources with which the request can finish earlier than its service time threshold. Once the harvestable service instances and the number of resources to harvest from them are determined, the service manager collects resources randomly from each of the selected service instances until the desired number of resources is collected. When a service instance finishes, it returns the harvested resources to service instances where the resources were collected. If the requests have already finished, those resources will be allocated to requests in the wait queue to reduce their wait time. If there are no requests in the wait queue, the resources will be allocated to service instances currently running.

5.3. Inter-site resource management

When the service manager of the home-site receives the request, the first decision to be made is to select the appropriate site to handle the request. The global scheduler may choose a site using the estimated completion times of the request that are predicted by participating service managers. However, since requests serviced by each site may not be the same, the quality of estimation generated by all sites is not equal. For example, if a site has serviced a large number of prior requests that are similar to the candidate request, the site is more likely to provide an accurate estimation than one that has serviced fewer requests. Due to this inequality of estimation, simply comparing the estimation is not sufficient. Another approach for site selection problem is that the global scheduler itself predicts the completion time of the request. That is, the global scheduler collects the status information of all sites and simulates the resource management systems of the sites to predict the completion time. This approach suffers less from prediction quality disparity, but it is computationally expensive. In particular, if participating sites are running dynamic resource management systems where the request priorities can change dynamically over time, reordering of the request execution can happen frequently, which in turn causes a large overhead to the global scheduler. As more sites are included, this problem becomes more significant. In this paper, we present two adaptive site selection heuristics that do not depend on accurate predictions of completion times of service requests.

5.3.1. Weighted queue length-based site selection

The basic idea of weighted queue length-based site selection (WQL) is that it presumes that a site with a shorter queue will typically finish a request earlier. Therefore, WQL distributes service requests across all sites in such a way that the number of queued requests of the sites are in proportion to their resource capacities, which is represented by the weight associated with the site. The weight can be computed by comparing the resource specifications of the site (e.g., processor speed, network bandwidth) against a base resource, or by running a suite of service requests on the site at bootstrap time and comparing the service times of the requests.

When a service request arrives, the global scheduler will select a site using the following equation:

$$\min_{1 \leqslant i \leqslant m} \{(n^i_{queued} + n^i_{forwarded}) \times w_i\}$$

where n^i_{queued} is the number of requests queued at site *i*, $n^i_{forwarded}$ the number of requests forwarded to site *i* during the current time interval, w_i the weight associated with site *i* and *m* the total number of sites.

n^i_{queued} is reported by each site periodically, and it remains until the new value is reported. Once the global scheduler selects a site and forwards the request to the site, it increments $n^i_{forwarded}$ and when each site reports its status to the front-end, the $n^i_{forwarded}$ of each site is reset to 0. Therefore, $(n^i_{queued} + n^i_{forwarded})$ is an approximation of the worst case queue length of site *i*. Since each site has a different resource capacity, the estimated queue length is normalized with the associated weight.

WQL has a limitation in that it considers only the number of queued requests, and not the run-times of those requests. Under dynamic resource management systems, a site with a shorter queue does not always guarantee shorter service time. For example, if each site is running shortest-job-first scheduling, for a short-running request, selecting a site (*A*) with many long-running requests pending will better than a site (*B*) with a few short-running requests pending since it is more likely that the request will have a higher priority over the requests pending at site (*A*). Therefore, WQL performs well when the run-times of the service requests do not fluctuate significantly; however, in the opposite situation, the performance gains may be small.

5.3.2. Multi-level queue-based site selection

The underlying principle of multi-level queue-based site selection (MLQ) is that by grouping requests with similar characteristics (e.g., the run-time of the request) together and forwarding them to the same site, requests with higher priorities (e.g., short-running requests) will be less influenced by dynamic changes of priorities of requests with low priorities (e.g., long-running requests).

In MLQ, the range of run-time is assigned to each site. When a new request arrives, the global scheduler predicts the run-time of the request and forwards it to the site whose range includes the predicted run-time of the request. The range will be assigned in such a way that the site providing faster resources will handle long-running requests. Note that MLQ predicts the run-times of requests in order to characterize the requests, and the predicted run-time is the time that the request will take on base machine (e.g., home-site cluster), and not the time that the request will take on the selected site. Thus, other information, such as input parameters to the service, also could be used for request characterization.

When many new requests have similar run-times, for example, a certain site will become overloaded quickly. To address this problem, MLQ checks the status of each site periodically and if the ratio of maximum queue length is above a threshold, it changes

the range of each site adaptively so that the overloaded site will serve smaller fractions of client requests. The proportion of the range that a site i will handle is defined as follows:

$$\left[\frac{Q_{total}/Q(i)}{\sum_{j=1}^{m} Q_{total}/Q(j)}\right] \times (\max(RunTime) - \min(RunTime)) \quad (3)$$

where $Q(i)$ is the normalized queue length of site i ($= n^i_{queued} \times w_i$). $Q_{total} = \sum_{i=1}^{m} Q(i)$ where w_i is the weight associated with site i, m the total number of sites, $max(RunTime)$ the maximum of optional run-times of requests during the previous time interval, $min(RunTime)$ the minimum of optional run-times of requests during the previous time interval.

In addition, if a site remains overloaded even after its range has been changed, MLQ will eventually mark the site as "Unavailable" until the site finishes most of the service requests queued at the site. To do so, MLQ uses a threshold called the adjustment threshold. Before adjusting the ranges, the global scheduler computes the ratio of the queue length to the minimum queue length, $Q(i)\big/\min(Q(j))_{1 \leqslant j \leqslant m}$ for each site i, and if the value is above the adjustment threshold, the global scheduler marks the site as "Unavailable". As in WQL, the queue length of each site is normalized with the weight associated with the site because each site has a different resource capacity. Fig. 3 illustrates how MLQ works. At T_1, the global scheduler detects that B is overloaded compared to other sites. It decides to shrink the range assigned to B. As a result, some service requests will be forwarded either A or C, which otherwise would have been directed to B. At T_2, B is still overloaded, which makes the global scheduler block B. Now, only A and C are actively serving the incoming service requests. Since both A and C handle requests that would have been forwarded to B, B will eventually finish most of its queued requests (T_3). At T_n, the global scheduler re-assigns the range to B so that all of the sites handle incoming requests. Note that the range of each site at T_n is not necessarily the same as those at T_1. It depends on the client workload observed.

5.3.3. Dynamic policy switching

Several site selection heuristics can exist, including WQL and MLQ. However, which heuristics is best depends on both resource capacity and the current status (e.g., the number of active requests and queue requests) of each site, as well as the characteristics of the incoming requests (e.g., client workloads: the run-times of requests, inter-request arrival time). In addition, it is necessary to factor in how resources will be allocated to the requests by the resource management system of the selected site. Therefore, without accurate information about the completion times of requests, the site selection heuristics will show different performance under different circumstances, which will be shown in Section 6.

To address this problem, we propose an adaptive site selection algorithm that supports dynamic policy switching, enabling the global scheduler to choose appropriate site selection heuristics dynamically over time, based on the past performance history. It is more complex to develop such a scheduler because, unlike the dynamic performance prediction, it is not possible to run several heuristics in parallel, since this would require different executions of the same service requests.

To achieve dynamic policy switching, the global scheduler periodically measures and records information to capture the performance of a site selection heuristics used during each time interval. The information consists of three tuples: $\langle$*heuristic ID*, *performance*, *workload*$\rangle$. "*heuristic ID*" is used to identify the site selection heuristic used for the current time interval, "*performance*" represents the performance of the heuristic, while "*workload*" represents the client workload observed. Note that those values for performance and workload characterizations are applied for each time interval; therefore, they are not accumulative. Since we assume that resources are space-shared, one possible metric for performance characterization is to use the queue length. The performance of a site selection heuristic used for time interval $\langle T_t, T_{t+1} \rangle$ is defined by

$$performance = [AWQ_t^{start}, AWQ_t^{end}] \quad (4)$$

where AWQ_t^{start} is the AWQ measured at the start of time interval $\langle T_t, T_{t+1} \rangle$, AWQ_t^{end} the AWQ measured at the end of time interval $\langle T_t, T_{t+1} \rangle$, $AWQ = \sum_{i=1}^{m} Q(i)/\sum_{i=1}^{m} w^i$ the weighted queue length, w^i the weight associated with site i, and m the total number of sites.

The client workload is characterized by two values: the average run-time of requests and the inter-request arrival time. For example, the client workload during $\langle T_t, T_{t+1} \rangle$ in Fig. 4 is computed by

$$Workload_t = [ART^t, RAT^t] \quad (5)$$

$ART_t = (\sum_{i=1}^{r} RunTime(R_{t,i})/r)$ is the average run-time of requests received during $\langle T_t\ T_{t+1} \rangle$, $RAT_t = \sum_{i=1}^{r-1}(Arrival(R_{t,i+1}) - Arrival(R_{t,i+1}))/r - 1$ the inter – request arrival time during $\langle T_t\ T_{t+1} \rangle$, $Arrival(R)$ the arrival time of request R, and r the total number of requests received during $\langle T_t, T_{t+1} \rangle$.

When the global scheduler receives a request, it predicts the run-time of the request that would take on the base machine, and uses the prediction to compute the average run-time of requests received during the time interval. Since the prediction is used only

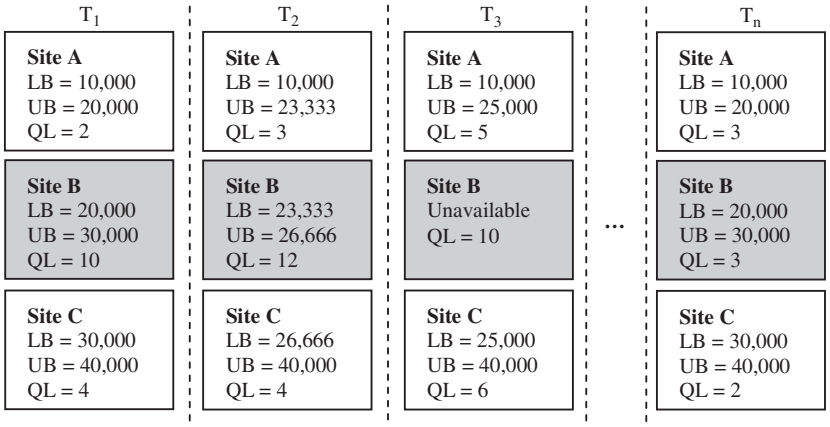

Fig. 3. Multi-level queue-based site selection: LB and UB represent the lower bound and the upper bound of the run-times assigned to each site, respectively. QL denotes the queue length reported at the start of each time period.

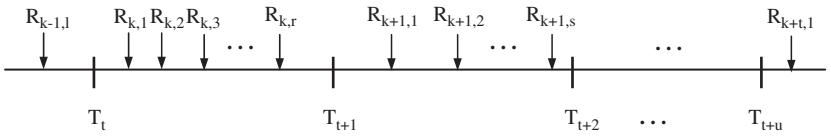

Fig. 4. Characterization of client workloads: $R_{i,j}$ represents the jth service request received during time period $\langle T_i, T_{i+1} \rangle$.

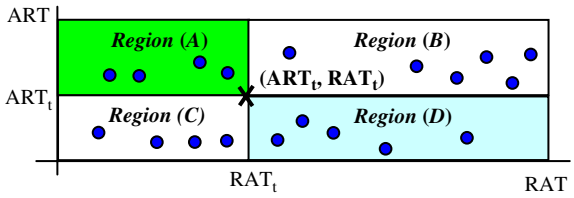

Fig. 5. Division of performance history space: the x-axis represents the average of run-times of requests, and the y-axis represents the inter-request arrival time. Given the current client workloads (ART_t, RAT_t), the past client workloads can be divided into four regions. The points represent three tuples $\langle$*heuristic ID*, *performance*, *workload*$\rangle$ measured.

for characterizing the client workload, the same base machine must be used.

Given the current observation, [ART_t, RAT_t, AWQ_t^{start}, AWQ_t^{end}], the global scheduler divides the past performance history space into four regions, based on the current client workload, [ART_t, RAT_t], and identifies those regions that can be directly compared with the current workload (see Fig. 5). It is apparent that points in region (A) have heavier workloads than the current workload, and points in region (D) have lighter workloads than the current workload. If the client workload for the next time interval becomes heavier than the current one, those heuristics showing better performance than the current heuristic under the heavier workload must be used. To select such heuristics, for each data point i in region (A), if the performance of the data point is better than the current performance, the heuristic of the data point will be selected. Since we do not know how the current heuristic will perform under the heavier workload, the candidate heuristics for the next time interval must also include the current heuristic. H_{recom} in the below equation represents the set of heuristics recommended for the next time interval:

$$H_{recom} = \{HID(i), i \in region(A) : (AWQ_i^{start} > AWQ_t^{start}) \text{ and } (AWQ_i^{end} \leqslant AWQ_t^{end})\} \cup \{HID(current)\} \quad (6)$$

$HID(i)$ is the heuristic ID of data point i in the performance history.

Similarly, if a heuristic showed worse performance than the current heuristic under a lighter workload, that heuristic will be avoided for the next time period. The current heuristic is not included in such heuristics in order to prevent frequent heuristic change. H_{avoid} in the equation below includes those heuristics:

$$H_{recom} = \{HID(i), i \in region(D) : (AWQ_i^{start} < AWQ_t^{start}) \text{ and } (AWQ_i^{end} \geqslant AWQ_t^{end})\} \quad (7)$$

Note that the performance histories belonging to regions B and C can also contain valuable information. However, comparing those histories with the current observation may require significant information regarding the workloads. For example, given the current observation [$ART_t = 10$, $RAT = 20_t$, $AWQ_t^{start} = 5$, $AWQ_t^{end} = 7$], suppose that performance history X in region B has [$ART_t = 20$, $RAT = 30_t$, $AWQ_t^{start} = 5$, $AWQ_t^{end} = 7$]. Using this information, it is not easy to determine whether or not X had operated under a heavier or lighter workload than the current one. In order to do so, more complete trace information of the client workload is required, which dramatically increases both storage and computational costs. Our approach, therefore, requires fewer storage and computational costs; yet, it can extract useful feedback from limited information.

Once having identified two sets of heuristics (H_{recom}, H_{avoid}), the global scheduler first randomly selects one of the heuristics that belongs to H_{recom}, but not H_{avoid}. If there are none satisfying the condition, it then randomly selects one of the heuristics that belongs to H_{recom}. If there are no such heuristics, then it selects one of those heuristics that do not belong to H_{avoid}. Finally, if there is no heuristics that satisfy any of the above conditions, the global scheduler selects one heuristic randomly from the policy space.

6. Experimental results

In order to evaluate the effectiveness of our approach, we have developed two prototypical high-end network services: N-body simulation service and library-to-library gene sequence comparison service. N-body simulation service is widely used in many scientific areas to explore the evolution of a system of N bodies (e.g., galaxies, particles) and it is a representative data parallel service requiring interactions between the constituent processes for either data communication or synchronization, and the communication patterns are symmetric. The N-body simulation service that we have deployed is an implementation of $O(n^2)$ algorithm, where n is the size of the input bodies. Library-to-library gene sequence comparison, on the other and, is used for determining the structure and function of biological sequences, which is an important problem in molecular biology. Library-to-library gene sequence comparison service is an example of distributed services. Distributed services are similar to data parallel services, but they differ from data parallel services wherein they do not require data exchange or synchronization among the distributed processes. Instead, a large set of independent tasks are distributed across a set of processors which communicate results back to a master node. The library-to-library gene sequence comparison service requires a source library of sequences as an input parameter and the time complexity of the service is $O(m)$, where m is the size of the source library of sequences. The feature that both of these types of services have in common is that they require significant computational resources as the problem size increases.

The experimental environment consists of a cluster setting and a Grid-like setting. A cluster setting uses a single Linux cluster, while the Grid-like setting includes three heterogeneous clusters (Linux, Solaris, SGI) interconnected with a fast LAN.

6.1. Performance prediction

Fig. 6 shows the prediction accuracies of several performance predictors for the prototypical services. For this experiment, we ran both services in a cluster setting, and the input parameters to the services were randomly generated. **O**(**N**), **O**(**N***N**), and **O**(**N***N***N**) are predictors using the static performance models of $O(n)$, $O(n^2)$, and $O(n^3)$, respectively. **FILTERING** uses the filtering and local linear regression technique (Lee and Schopf) for prediction using the number of processors and the input to the service (e.g., the size of input bodies, the size of the source library) as a filter. The dynamic performance predictors run these four

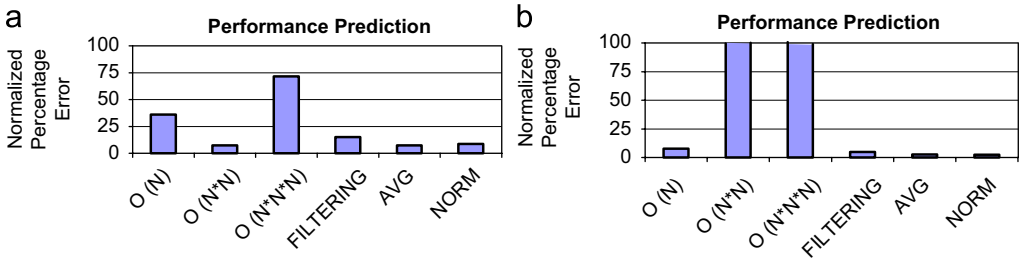

Fig. 6. Comparison of prediction accuracy: (a) N-body simulation service and (b) gene sequence comparison service.

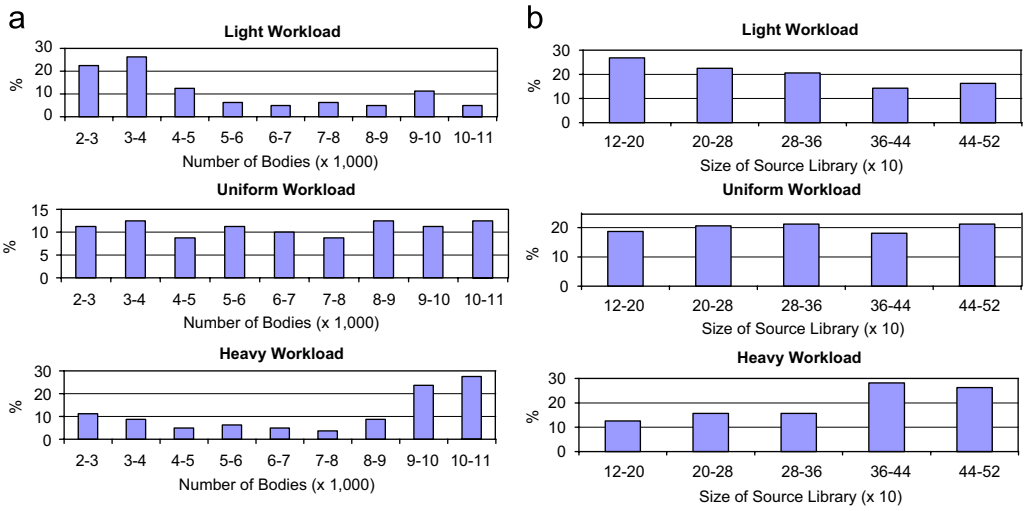

Fig. 7. Synthetic workloads for the intra-site resource management: (a) N-body simulation service and (b) gene sequence comparison service.

predictors in parallel when predictions are required. The difference involves which metric is used for ranking those predictors. **AVG** represents the dynamic performance predictor using the average prediction error rate of the most recent k predictions, and **NORM** represents the dynamic performance predictor using the normalized percentage error of the most recent k predictions (see Eq. (2)). We have conducted several experiments while varying the values of k, and the results shown throughout this paper used the value $k = 10$. For comparison, we measured the normalized percentage error of each performance predictor.

O(N*N) and **O(N)** can generate accurate predictions for the N-body simulation service and the library-to-library gene sequence comparison service, respectively and they can achieve the prediction error rate to within 7% and 9%. For both services, **FILTERING** can learn the relationships among input parameters, the resource set, and the service time. Therefore, after the learning phase, the service time can be predicted accurately.

The dynamic performance predictors generate accurate predictions, achieving prediction error rates to within 10% and 4% for the N-body simulation service and the gene sequence comparison service, respectively. These are close to the error rates that can be achieved by the best predictor for each service. The performance difference between **AVG** and **NORM** was negligible. This result can provide good evidence that the dynamic performance predictor can dynamically choose the most accurate predictor over time.

6.2. Intra-site resource management

6.2.1. Workload

To assess the performance of resource harvesting, we generated three synthetic workloads for both services: light, uniform, and heavy workloads (see Fig. 7). For the N-body simulation service, we ran 15 clients for the experiment, and each client waited from 1 to 5 min randomly after receiving the results for the previous request from the service. The x-axis represents the number of bodies submitted to the service, and the y-axis represents the percentage of the number of requests for each category in the total number of requests. Similarly, for the library-to-library gene sequence comparison service, we ran 25 clients because the run-times of this service is relatively small; therefore, it requires more simultaneous requests to make the service overloaded. The x-axis represents the number of sequences in the source libraries. The target library contains 500 sequences, and it is in all of the nodes of the Linux cluster.

6.2.2. Performance comparison

We compared the performance of resource harvesting against two simple scheduling policies: MODABLE and IDEAL, as shown in Fig. 8. MODABLE assigns idle resources up to the optimal number of resources for each request. The optimal number of resources is the number of resources that makes the run-time of the request minimal and is determined by the performance predictor. If there are no available resources, the request is queued. In IDEAL, only the optimal number of resources is assigned to the request. Therefore, if the number of resources available is less than the optimal number of resources, the request waits until the optimal number of resources is available. Both scheduling policies do not use resource harvesting. We conducted several experiments for each harvesting policy while varying the configurable parameters. Table 1 shows the configurable parameters of *SRT_Harvest* and *IB_Harvest* used in Fig. 8. The effects of the parameters on the behaviors of the harvesting policies will be shown later.

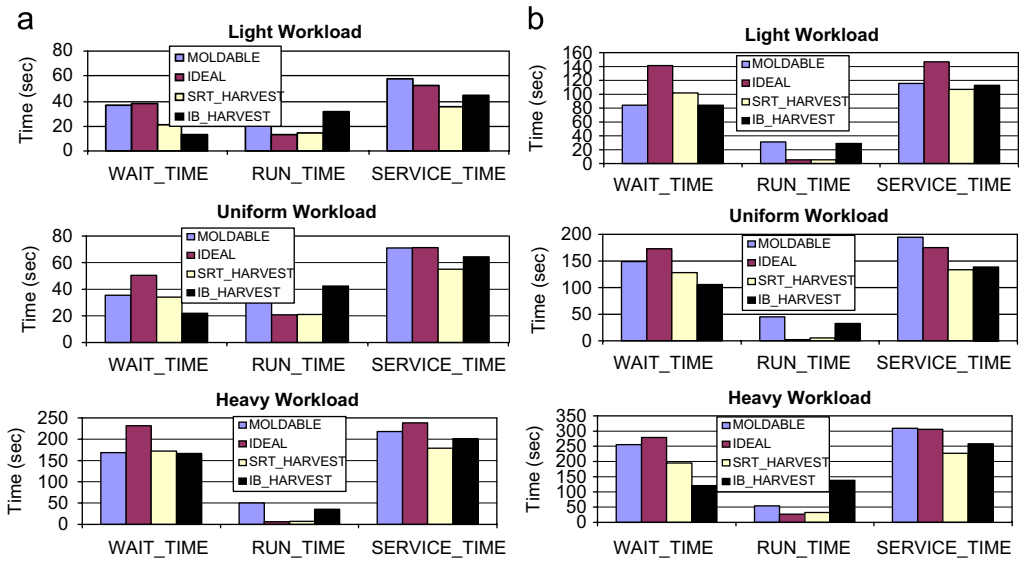

Fig. 8. Comparative performance for different harvesting policies: (a) N-body simulation service and (b) gene sequence comparison service.

Table 1
Configurable parameters for resource harvesting algorithms.

	Light workload	Uniform workload	Heavy workload
HP (harvesting parameter)	1.5	1.8	1.2
WP (wait time parameter)	12.0	8.0	12.0
IP (impact parameter)	1.7	1.1	1.3

As IDEAL always waits until the optimal number of resources is available, its wait time is the highest, but the average run-time of IDEAL is the smallest for the same reason. In *SRT_Harvest*, by executing short-running requests earlier than long-running requests, the wait time is decreased significantly. In addition, since *SRT_Harvest* assigns the idle resources to the active service instances, the average run-time can be reduced. As in *SRT_Harvest*, *IB_Harvest* also dynamically collects resources for new requests whenever there are not enough resources for them. Therefore, its average wait time is also smaller than those of simple policies. Moreover, unlike *SRT_Harvest*, since *IB_Harvest* favors requests in the wait queue, the average wait time is even smaller than that of *SRT_Harvest*. However, due to resource harvesting, each request can use only the minimum number of resources, which leads to increased run-time. Therefore, when the wait time is a more significant metric for evaluating the quality of service, *IB_Harvest* is more appropriate than *SRT_Harvest*. On the other hand, if the overall service time is more significant then *SRT_Harvest* must be used. Table 2 shows the maximum performance improvement achieved through resource harvesting for each service.

6.2.3. Sensitivity to configurable parameters

In theory, running requests with the smallest run-time first always reduces the average wait time. Therefore, in the shortest-remaining time harvesting, $HP = 1.0$ should provide the best performance However, due to *WP* short-running requests may wait until non-harvestable requests are finished. Furthermore, a smaller *HP* makes the wait time of long-running requests reach the maximum wait time threshold faster because long-running requests either may not be selected for execution or may frequently relinquish all of their resources to short-running requests. These two behaviors make the wait time of short-running requests longer if they arrive when non-harvestable requests are using all of the system resources (Fig. 9). However, as *HP* increases, the total wait time also increases because short-running requests may not be executed, even though long-running requests are using resources. The reason for choosing a larger value as *WP* for a heavy workload is that as the run-time of each request is relatively high in the heavy workload, small *WP* makes the total wait time of each request quickly exceed the maximum wait time threshold. Therefore, shortest-remaining time harvesting may not take advantage of resource harvesting.

Fig. 10 shows that as *WP* increases, the average performance improves in all workloads. This is quite straightforward because with a very large *WP*, whenever short-running requests arrive, they acquire resources from long-running requests. On the other hand, if a small *WP* is used, as the maximum wait time threshold of each request becomes small, they quickly become non-harvestable. Therefore, even if short-running requests arrive, they may not acquire resources, which results in increased wait time.

A larger *IP* allows more frequent resource harvesting. Therefore, as *IP* increases, the average wait time decreases. However, at a certain point, since most of the active service instances are using the minimum number of resources, the decrements of average wait times cannot compromise the increments of the average run-time (see Fig. 11).

We acknowledge that the harvesting policies require different values for their configurable parameters to achieve the best performance regardless of varying circumstances. One of our future research objectives is to develop algorithms for the dynamic selection of configurable parameters of the harvesting policies.

6.3. Adaptive resource management for a Grid

To evaluate the performance of site selection heuristics, we have deployed the N-body simulation service and the library-to-library gene sequence comparison service on a Grid-like setting. All of the participating sites run the shortest-remaining time harvesting as intra-site resource management for the local cluster, and the Linux cluster plays the role of the home-site. *HP* and *WP*

Table 2
Performance improvement using resource harvesting.

Service type	Harvesting	Workload		
		Light workload (%)	Uniform workload (%)	Heavy workload (%)
N-body simulation service	*SRT_Harvest*	40	20	27
	IB_Harvest	23	10	16
Gene sequence comparison service	*SRT_Harvest*	27	25	24
	IB_Harvest	23	23	15

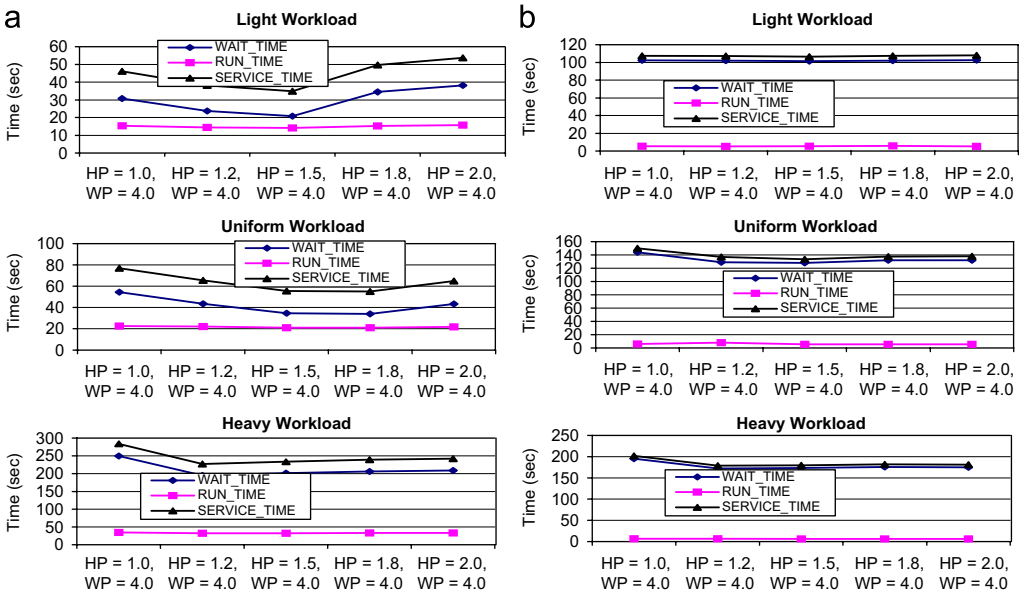

Fig. 9. Sensitivity to harvesting parameter: (a) N-body simulation service and (b) gene sequence comparison service.

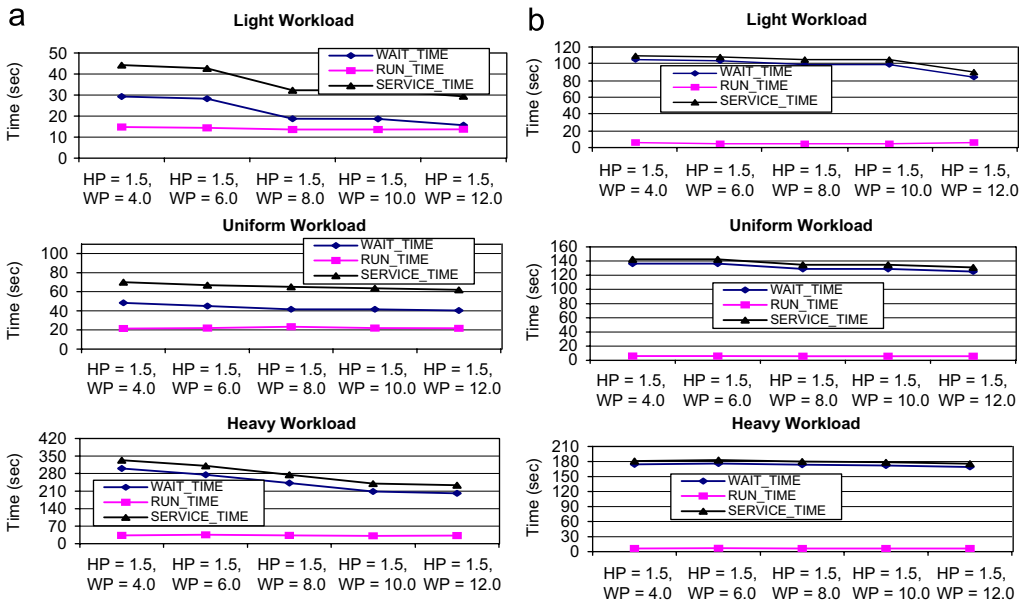

Fig. 10. Sensitivity to wait time parameter: (a) N-body simulation service and (b) gene sequence comparison service.

for the harvesting algorithm were set to 1.8 and 12.0, respectively; all sites used the same values for the parameters. Fig. 12 shows the workloads used.

We compared the performance of WQL and MLQ against a baseline-Round-Robin selection policy. Each participating site reports its current queue length every 5 min. The adjustment

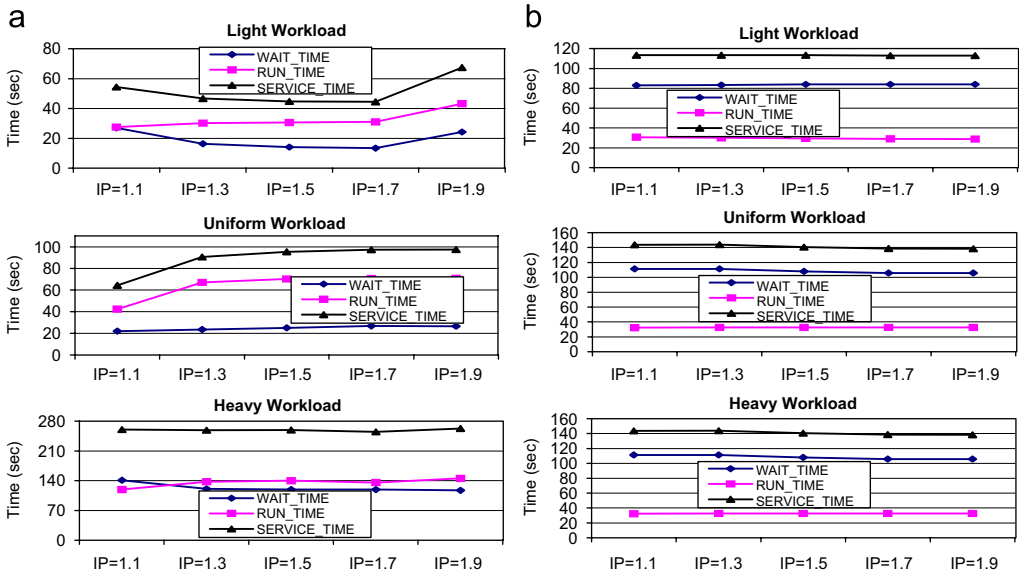

Fig. 11. Sensitivity to impact parameter: (a) N-body simulation service and (b) gene sequence comparison service.

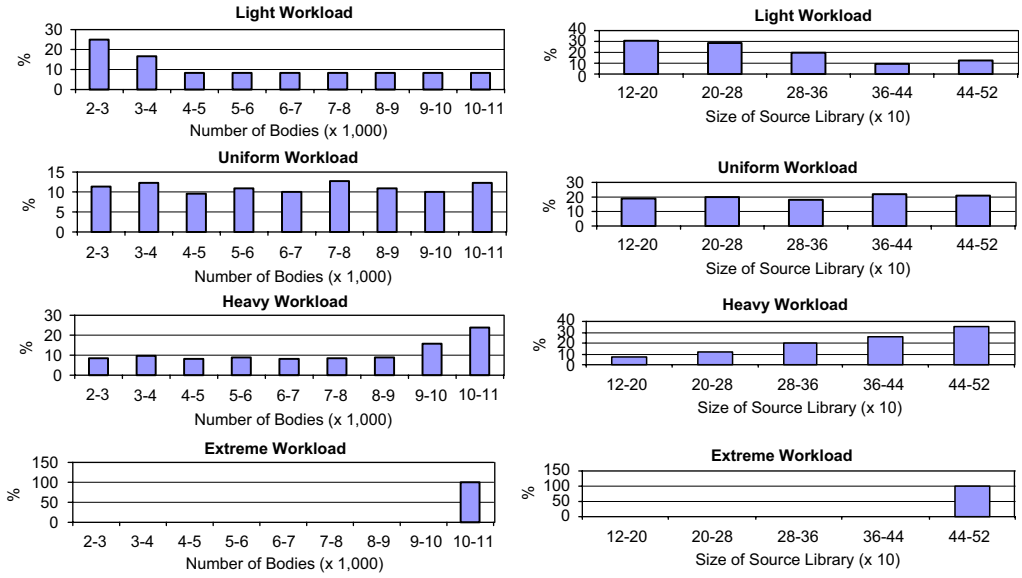

Fig. 12. Synthetic workloads for inter-site resource management: (a) N-body simulation service and (b) gene sequence comparison service.

threshold for MLQ is set to 2.0, and the weights associated with Linux, Solaris, and SGI clusters are 1.2, 1.4, and 3.1, respectively. These values were selected by pre-processing of a related set of experiments and were fixed throughout.

As shown in Fig. 13, for each workload, both WQL and MLQ outperformed the Round-Robin selection policy, since they considered the resource capacity and the current status of each site when scheduling the service requests. Furthermore, each heuristic shows different performances under different workloads. For example, WQL shows better performance under light and extreme workloads, while MLQ outperforms other policies under uniform and heavy workloads for the N-body simulation service. Table 3 shows the performance improvement of each heuristic against the Round-Robin policy.

This experiment also shows that the performance of each heuristic can be dependent on the characteristics of the services. Unlike the N-body simulation service, for the library-to-library gene sequence comparison service, WQL outperforms other selection policies, regardless of the workloads (see Fig. 13). This result occurs because the run-times of the requests to the service are relatively small and do not fluctuate significantly regardless of different input parameters. For example, the requests comparing a source library whose size is 120 take 4.1 s using 7 nodes of the Linux cluster; those comparing a source library whose size is 520 take 8.3 s. On the other hand, the requests to compute the positions of 2000 bodies take 7.5 s using 7 nodes of the Linux cluster; those to compute the positions of 11,000 bodies take 33.9 s. Therefore, the queue length can be used as a good substitute for the estimated completion time.

6.3.1. *Dynamic policy switching*

Fig. 14 shows synthetic workloads used to evaluate the performance of dynamic policy switching: hybrid workload and hybrid-extreme workload. The hybrid workload includes three different types of workloads: light, heavy, and uniform workloads.

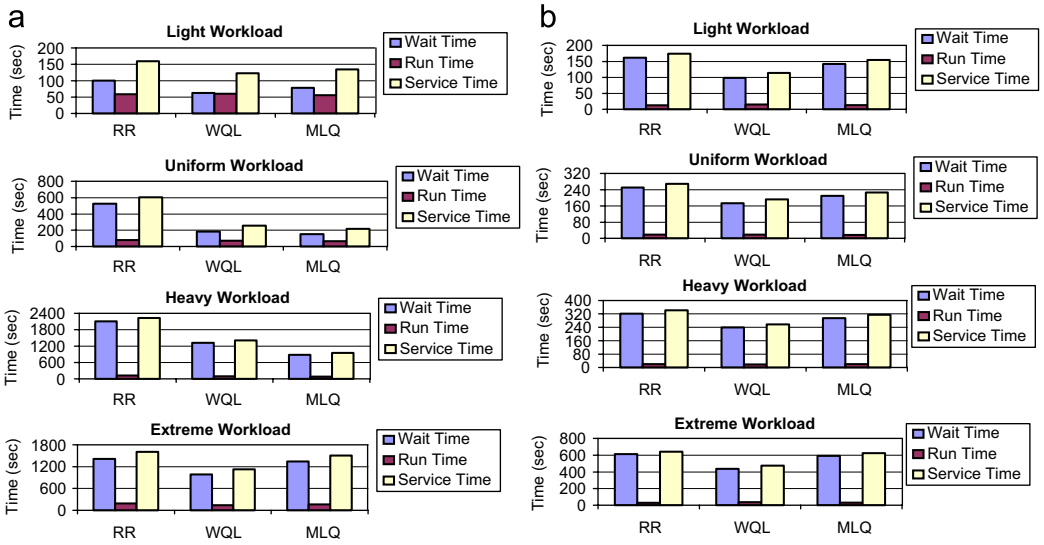

Fig. 13. Comparative performance under different selection policies: (a) N-body simulation service and (b) gene sequence comparison service.

Table 3
Performance improvement using WQL and MLQ.

Service type	Selection	Workload			
		Light workload (%)	Uniform workload (%)	Heavy workload (%)	Extreme workload (%)
N-body simulation service	WQL	23	57	37	30
	MLQ	22	71	57	6
Gene sequence comparison service	WQL	35	29	24	26
	MLQ	11	16	9	3

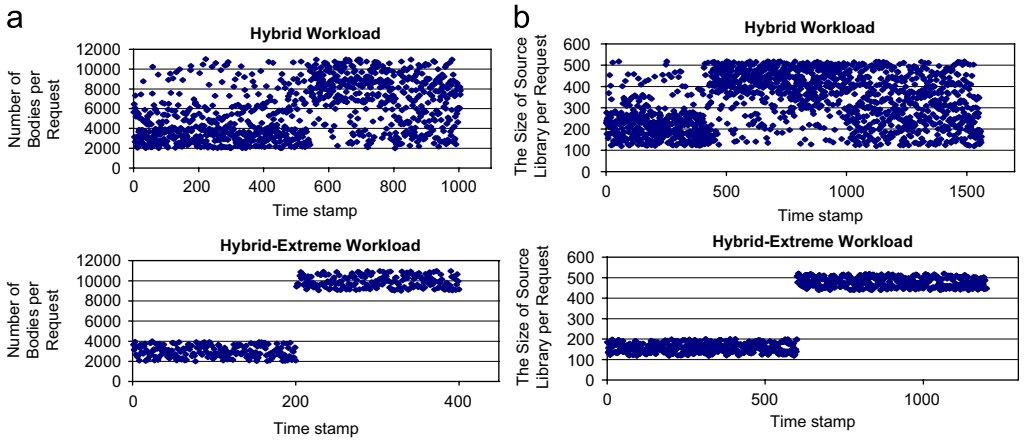

Fig. 14. Synthetic workloads for dynamic policy switching: (a) N-body simulation service and (b) gene sequence comparison service.

In the hybrid-extreme workload, all of the service requests have similar run-times for a certain time periods. For this experiment, each site reports its current queue length every 5 min. The global scheduler characterizes the performance of the site selection heuristic and client workload every 5 min, and makes a decision on which site selection policy to use among Round-Robin, MLQ, and WQL every 10 min. Other configurations, such as inter-request generation time, are the same as those in the previous experiments.

Figs. 15 and 16 show the performance of dynamic policy switching and other site selection heuristics under two workloads. Table 4 shows the maximum performance improvement of each site selection policy against the Round-Robin policy.

Since dynamic policy switching utilizes only subsets of past performance history, some significant past history belonging to regions *B* and *C* may not be considered (see Fig. 5). This situation may cause incorrect selection, such as picking up the Round-Robin policy. However, in general, dynamic policy switching can choose the most appropriate site selection strategies for the current workload and the service. The quality of dynamic selection will increase as more diverse past performance histories are collected. For example, after processing light and heavy workloads, MLQ is

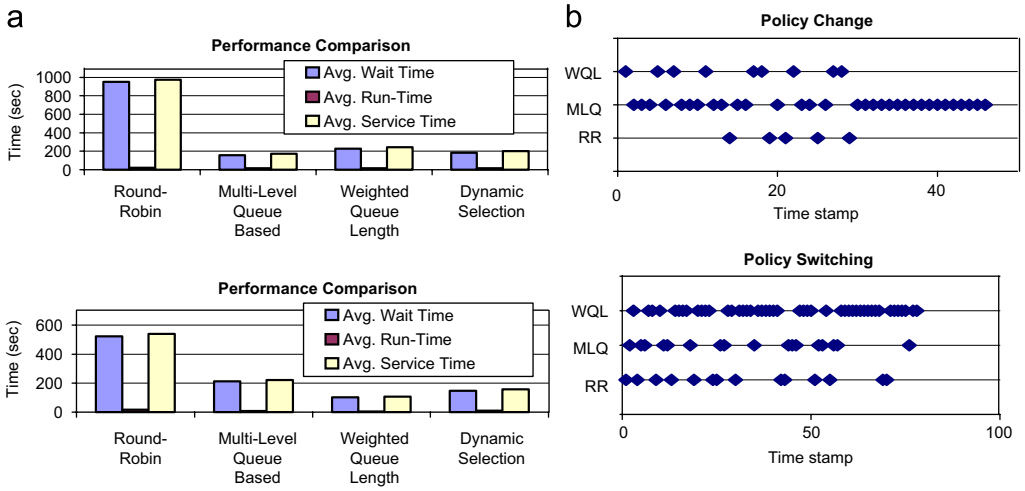

Fig. 15. Performance of dynamic policy switching under the hybrid workloads: (a) N-body simulation service and (b) gene sequence comparison service.

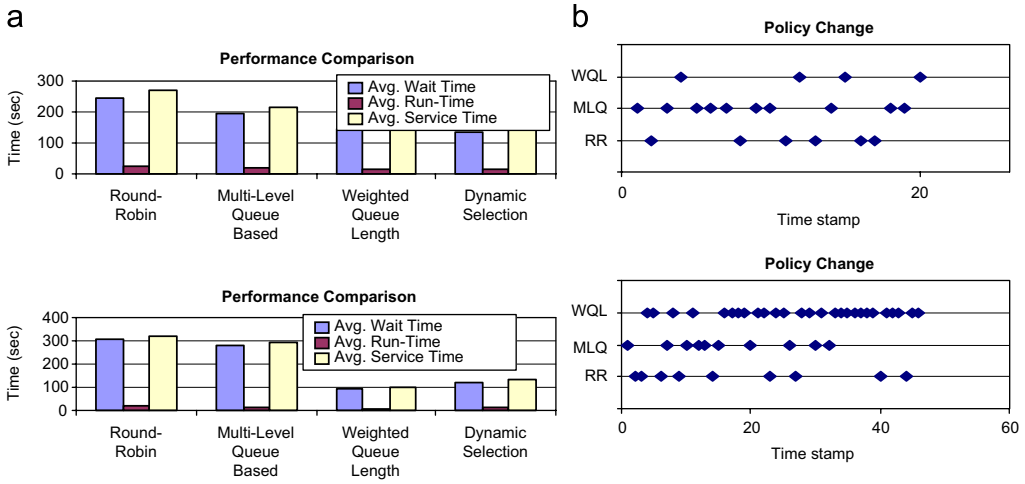

Fig. 16. Performance of dynamic policy switching under the hybrid-extreme workloads: (a) N-body simulation service and (b) gene sequence comparison service.

Table 4
Performance improvement of adaptive site selection policies.

Service type	Selection	Workload	
		Hybrid workload (%)	Hybrid-extreme workload (%)
N-body simulation service	WQL	75	44
	MLQ	82	20
	Dynamic switching	80	61
Gene sequence comparison service	WQL	80	70
	MLQ	58	10
	Dynamic switching	70	56

more frequently selected under a uniform workload for the N-body simulation service. Similarly, the frequency of WQL also increases during a uniform workload for the library-to-library gene sequence comparison service.

7. Summary and future work

In this paper, we have proposed the general-purpose run-time infrastructure for high-end network services. We have focused on effective resource management of the infrastructure which must be realized in order to deliver scalable performance. The run-time infrastructure that we propose is built upon three important components: (1) dynamic performance prediction; (2) adaptive intra-site resource management; and (3) adaptive inter-site resource management. Performance information is a fundamental component for resource selection in a Grid environment, as well as most application and job scheduling approaches. The novel aspect of our approach for performance prediction is to allow the run-time system to select the most appropriate one among a pool

of performance predictors for the given service. In dynamic environments such as Grids, dynamic and unpredictable resource availability requires the service itself to be adaptive with respect to system resources. We presented promising techniques to implement dynamic resource sharing. Finally, as Grids evolve, it will often be the case that the network service will be installed at multiple sites to deliver scalable performance. We proposed adaptive multi-level resource management to control distributed and heterogeneous resources for such services. The hierarchical structure can ensure scalability of the service and autonomy of local sites. In addition, we proposed a novel technique enabling the run-time system to dynamically select the most appropriate resource management strategies based on performance feedback.

Future work lies in several areas. Although the resource harvesting technique provides acceptable performance, its behaviors depend on the configurable parameters. One of our future research objectives is to develop algorithms for the dynamic selection of configurable parameters of the harvesting policies. The target services that we considered are stateless, non-composite services whose computation time is a dominant factor of the service time. The future work will extend this assumption to support tactful, composite services. The idea we plan to investigate is that the global scheduler may maintain a DAG annotated by the amount of computation for the participating services and the amount of communication expected between those services. At completion of each service, the global scheduler may update the DAG and reassign new services for the next steps, based on dynamic information regarding resources and services.

References

Armstrong R, et al. Towards a common component architecture for high-performance scientific computing. In: Proceedings of the Eighth IEEE International Symposiums on High Performance Distributed Computing, 1999.

Berman F, et al. Adaptive computing on the Grid using AppLes. IEEE Transactions on parallel and distributed systems 2003;14(14):369–82.

Buyya R, et al. Economic models for resource management and scheduling in grid computing. Concurrency: Practice and Experience (Special Issue on Grid Computing Environment), 2002.

Carman M, et al. Towards an economy-based optimization of file access and replication on a data grid. In: Proceedings of the Second International Symposium on Cluster Computing and the Grid, 2002.

Casanova H, Dongarra J. NetSolve: a network server for solving computational science problems. International Journal of Supercomputing Applications and High Performance Computing 1997;11(3).

Dinda P, et al. The architecture of the Remos system. In: Proceedings of the 10th IEEE International Symposiums on High Performance Distributed Systems, 2001.

Ernemann C, et al. Enhanced algorithms for multi-site scheduling. In: Proceedings of the Third International Workshop on Grid Computing, 2002.

Foster I, et al. The physiology of the grid: an open Grid service architecture for distributed systems integration. Available at 〈http://www.globus.org〉.

Ganglia, 〈http://ganglia.sourceforge.net/〉.

Global Grid Forum, 〈http://www.ggf.org〉.

The Globus Toolkit, 〈http://www.globus.org〉.

Govindaraju M, et al. XCAT 2.0: a component-based programming model for grid web services. Technical Report, TR-563, Department of Computer Science, Indiana University, 2002.

Grid Economic Services Architecture, 〈http://www.ggf.org/3_SRM/gesa.html〉.

Ivan A, et al. Partitionable services: a framework for seamlessly adapting distributed applications to heterogeneous environments. In: Proceedings of the 11th IEEE International Symposiums on High Performance Distributed Computing, 2002.

Kapadia NH, Fortes JAB. PUNCH: an architecture for web-enabled wide-area network computing. Cluster Computing 1999;2.

Kapadia NH, et al. Predictive application-performance modeling in a computational grid environment. In: Proceedings of the Eighth IEEE International Symposiums on High Performance Distributed Computing, 1999.

Keller A, et al. Managing dynamic services: a contract based approach to a conceptual architecture. In: Proceedings of the Eighth IEEE/IFIP Network Operations and Management Symposium, 2002.

Lee B. Adaptive middleware for high-end network services, PhD dissertation, 2003.

Lee BD, Schopf JM. Run-time prediction of parallel applications on shared environments. Proceedings of international conference on cluster computing, 1–4 December 2003. p. 487–491.

Maheswaran M, Siegel H. A dynamic matchmaking and scheduling algorithm for heterogeneous computing systems. In: Proceedings of the Seventh Heterogeneous Computing Workshop, 1998.

Raman R, et al. Matchmaking: distributed resource management for high throughput computing. In: Proceedings of the Seventh IEEE International Symposiums on High Performance Distributed Computing, 1998.

Smith W, et al. Predicting application run-times using historical information. In: Proceedings of IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing, 1999.

Streit A. A self-tuning job scheduler family with dynamic policy switching. In: Proceedings of the Eighth Workshop on Job Scheduling Strategies for Parallel Processing, 2002.

Subramani V, et al. Distributed job scheduling on computation grids using multiple simultaneous requests. In: Proceedings of the 11th IEEE International Symposiums on High Performance Distributed Computing, 2002.

Sunderanm V, Kurzyniec D. Lightweight self-organizing framework for metacomputing. In: Proceedings of the 11th IEEE International Symposiums on High Performance Distributed Computing, 2002.

Takefusa A, et al. A study of deadline scheduling for client–server systems on the computation grid. In: Proceedings of the 10th IEEE International Symposiums on High Performance Distributed Computing, 2001.

Vadhiyar SS, Dongarra J. A meta-scheduler for the grid. In: Proceedings of the 11th IEEE International Symposiums on High Performance Distributed Computing, 2002.

Vaughan-Nichols SJ. Web services: beyond the hype. IEEE Computer 2002.

Weissman JB. Predicting the cost and benefit for adapting data parallel applications on clusters. Journal of Parallel and Distributed Computing 2002;62(8).

Weissman JB, Lee B. The virtual service grid: an architecture for delivering high-end network services. Concurrency: Practice and Experience 2002;14(4).

Wolski R. Dynamically forecasting network performance using the network weather service. Cluster Computing: Networks, Software Tools, and Applications 1998.

Wolski R, et al. Analyzing market-based resource application strategies for the computational grid. International Journal of High Performance Computing Applications 2001;15(3).