Community Services: A Toolkit for Rapid Deployment of Network Services

Byoung-Dai Lee and Jon B. Weissman
Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN U.S.A
{blee, jon}@cs.umn.edu

Abstract

Advances in packaging and interface technologies have made it possible for software components to be shared across the network through encapsulation and offered as network services. They allow the end-users to focus on their applications and obtain remote services when needed simply by invoking the services across the network. Many groups have built significant infrastructure for providing domain-specific high performance services. However, Transforming high performance applications into network services is labor-intensive and time consuming because there is little existing infrastructure to utilize. In this paper, we propose a software toolkit and run-time infrastructure for rapid deployment of network services.

1. Introduction

High performance applications are used in many areas to model and solve complex problems. Examples of such problems include modeling large DNA structures, global weather forecasting and predicting motion of the astronomical bodies in space, to name a few. A number of codes and libraries that support high performance applications have been developed and are actively being used by the community. Some popular examples include Netlib, a collection of numerical libraries and CHARMM, a code for dynamic macro-molecular simulations. However, in order to develop new high performance applications using such supporting codes, it is necessary to acquire sources codes of the libraries, compile them on local machines, and tune the performance of the libraries. To make things worse, those supporting codes often require the need to install a code base such as MPI.

Fortunately, advances in packaging and interface technologies ([1][2][3]) have made it possible for software components to be shared across the network through encapsulation and offered as network services. They allow the end-users to focus on their applications and obtain remote services when needed simply by invoking remote services across the network

We believe the next dominant paradigm for high performance computing will be based on high performance network services. To support this vision, significant amount of high performance applications will be transformed into network services, which we call ***Community Services***. However, transforming high performance applications into network services is labor-intensive and time consuming because there is little existing infrastructure to utilize.

To address this difficulty, we propose a software toolkit and run-time infrastructure for rapid deployment of network services. However, in this paper, among the many issues to consider when building such infrastructures, we'll focus on resource management because efficient resource management is indispensable to providing acceptable performance for a wide-spectrum of users.

2. Systems Architecture

The community service infrastructure that we propose is an open architecture that consists of three software components designed to manage services (Figure 1.).

The **meta-service** is responsible for the overall management of the service providers environment. It

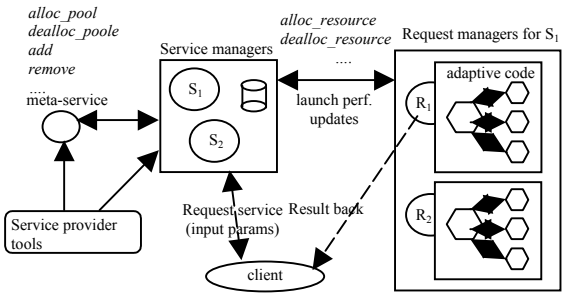

Figure 1. Community service architecture

provides facilities to add and remove services, and performs global management of the site resources by allocating and deallocating resource pool to separate services. The **service manager** handles client requests for a particular service. It tracks the current requests for the service, gathers incremental performance updates from the request manager, and makes resource allocation decision on behalf of user requests within its resource pool. It also maintains a performance history of the service. The service manager provides an initial resource allocation to the request manager. However, it is possible that the resources can be added or removed later as part of dynamic resource management. The **request manager** is a front-end linked in with the service code and is the place where adaptive actions such as resource addition and removal are actually implemented.

4. Adaptive Resource Management

Network services imply the potential of multiple concurrent users across the network and heterogeneous service requests (e.g. requests with different input parameters). Therefore, the run-time infrastructure must provide acceptable performance for a wide-spectrum of users. The novel aspect of our resource management system is ***resource harvesting***, where the resource management system is able to add/remove resources dynamically to/from the active service requests to support high performance. Therefore, performance prediction is crucial to adaptive resource management because deciding how to best allocate resources dynamically depends on the estimation of service execution time.

Resource harvesting raises two fundamental questions that must be addressed.

- *From which service requests should resources be harvested?*
- *How many resources should be harvested?*

One way to address the first question is the service providers establish priority classes of its user base so that higher priority requests can harvest resources from lower priority requests. The second question is more complex because the performance gain achieved by resource harvesting must be able to amortize the performance loss experienced by requests from which resources are taken away.

4.1. Performance Prediction

Performance prediction is needed by the service manager to estimate the cost of executing a specific instance of a service request on a given amount of resources. The service execution time depends on not only the number of resources but also the input parameters to the service. We introduce local linear regression and clustering technique to predict run-time, where local linear regression is applied using a subset of prior performance data that are clustered near the new data point. Performance history is organized in a two dimensional matrix, where each column represents the resource set and each row represents performance of the service given a set of input parameters on each resource set. To fill in the matrix, a configurable parameter, *cluster range*, is used. It determines whether or not two different input parameters can be regarded as similar in terms of performance when the same resource set is used. When a triple <input parameters, resource set, performance> is reported by a request manager, the system locates the corresponding cell in the matrix and test the following condition:

$$\alpha \le \beta \pm (\beta \times Cluster\ Range)$$

$$\alpha : new\ performance\ data$$

$$\beta : the\ average\ value\ of\ the\ performance\ in\ the\ cell$$

If the condition is satisfied, then the new performance data is stored into the cell. Otherwise, a new row for the input parameters is created. Note that each cell maintains several performance data.

To predict the performance of the service given (input parameters:*a*, resource set:*b*), the service manager first locates the corresponding cell in the matrix indexed by (*a, b*). If it is populated, then it simply returns the average value of the performance data in the cell. Otherwise, it finds the nearest two cells on the same row as the cell and applies local linear regression using the performance data in the two cells (row-wise prediction). After that, it finds the nearest two cells on the same column as the cell and does the same operation (column-wise prediction). Finally, the service manager returns the average value of the two estimated values. Row-wise prediction reflects the performance change depending on resource set given the input parameters, whereas column-wise prediction reflects the performance change depending on input parameters given the same resource set.

4.2. Shortest-Remaining-Time Harvesting

To address two fundamental questions mentioned above, we introduce a resource harvesting algorithm, which we call ***SRT_Harvest***. The idea behind this algorithm is when a new service request, *S*, can finish earlier than other service requests that are currently running, then *S* can harvest resources from those service requests to enable it to run. The behavior of the algorithm is determined by two configurable parameters:

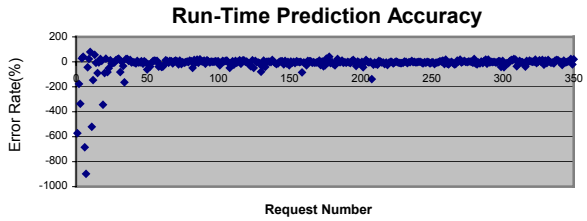

Figure 2. Accuracy of run-time prediction

- *HP(Harvesting Parameter)*: controls how aggressively the system can harvest resources from running request.
- *WP(Wait Time Parameter)*: defines the maximum wait time threshold for each request. It is proportional to the minimum run-time of the request.

To prevent the starvation of longer requests, whenever resources are available, the system checks if there are any pending requests whose total wait time exceeds the maximum threshold, defined by (*EstimatedMinimumRunTime* * *WP*). If so, resources are allocated to those requests and the resources are marked as "Non-Harvestable" so that no requests can take resources away from them.

Before harvesting resources, the service manager contacts the request managers of active requests to acquire the current status information as to progress. Using this information, the service manager computes the available time of the resources that are currently being used by active requests. Then, for each request in the wait queue, the algorithm computes the best performance achievable using resource harvesting. It can harvest resources from *R* for request *S* only when the following condition is satisfied:

$$Run\ Time\ of\ S \times HP < Remaining\ Time\ of\ R, (HP \geq 1.0)$$

This condition determines the candidate requests from which resources will be taken away and the number of resources from those requests. If only a subset of resources are harvested, there could be requests in the wait queue that can finish earlier than the request using the remaining resource set. Since the algorithm favors shorter requests, in such case, the active request should relinquish its resources to the one in the wait queue.

5. Experiments

We have built a Linux-based community service prototype and have deployed an N-body simulation service to test the performance of the approach. The objective of N-body simulation is to find the positions and movements of bodies in space that are subject to gravitational forces from other bodies using Newtonian laws of physics. In the prototype, the N-body simulation is implemented using Master/Slave paradigm, where the master maintain a bag of tasks and slaves repeatedly get tasks, update the bodies, then return the results to the master. We deployed the prototype service on a Linux Cluster consisting of 10 dual cpu PCs and the cluster is dedicated to the service.

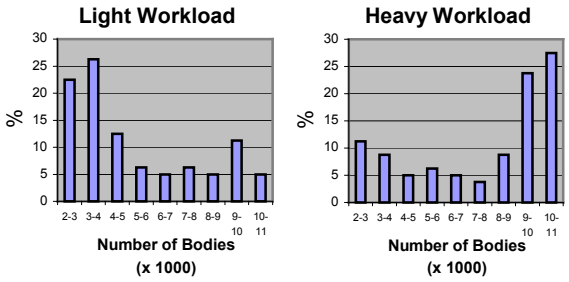

Figure 3. Synthetic workloads

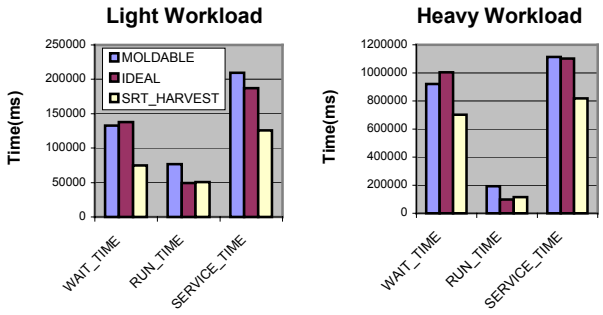

Figure 4. Comparative performance

Figure 2. shows run-time prediction accuracy. For this experiment, we generated input parameters to the service randomly and the cluster range is set to 0.05. The experimental results show that our prediction system can achieve prediction accuracy to within 4% error rate. Since initially there are not enough data in the performance history matrix, the error rates of the first few predictions are high. However, as clients requests are served, the prediction system learns the relationship between input parameters, resource set, and the run-time. Therefore, after the learning phase, it can predict the run-time accurately.

To assess the performance of our resource management system, we generated two synthetic workloads: light workload and heavy workload (Figure 3.). The X-axis in the graph represents the number of bodies submitted to the service to compute movement. Note that since we fixed other parameters to the service, the number of bodies controls the run-time.

We compared performance against two simple scheduling policies: MOLDABLE and IDEAL (Figure 4.). MOLDABLE 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 minimum. If there are no available resources, the request is queued.

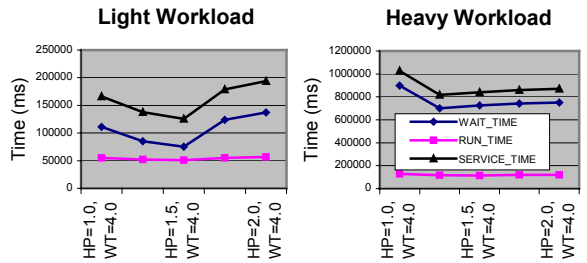

Figure 5. Sensitivity to HP

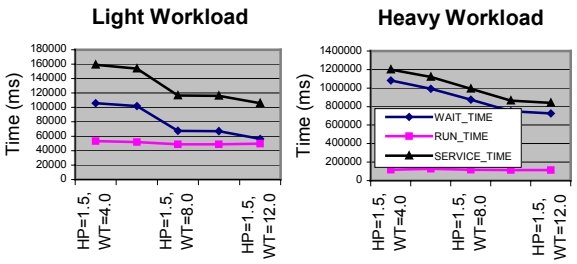

Figure 6. Sensitivity to WP

Otherwise, the request is "molded" to the available resources. In IDEAL, only the optimal number of resources are assigned to the request. Therefore, if the number of available resources less than the optimal number of resources, the request waits until the optimal number of resources are available. Both scheduling policies do not use resource harvesting. Table 1. shows configurable parameters of SRT_Harvest algorithm used in the experiment.

Table 1. Configurable parameters of SRT_Harvest algorithm

	Light Workload	Heavy Workload
Cluster Range	0.05	0.05
HP	1.5	1.2
WP	12.0	12.0

For each workload, we measured average values of wait time, run-time, and service time of requests under different resource management policies. Wait time denotes a period of time in the wait queue, whereas run-time represents the time consumed to process the request. The service time is the sum of wait time and run-time. As IDEAL always waits until the optimal number of resources are available, its wait time is the highest but the average run-time is the smallest for the same reason. In SRT_Harvest, by executing shorter requests earlier than longer requests, the wait time is decreased significantly. In addition, because request may not always run using the optimal number of resources, its average run-time is higher than that of IDEAL. In this experiment, SRT_Harvest achieved service time improvement up to 50% and 30% for light workload and heavy workload respectively.

Figure 5-6 show the behavior of SRT_Harvest algorithm as varying configurable parameters. In theory, running shortest requests first always reduces the average wait time. Therefore, *HP=1.0* should provide best performance. However, due to *WP*, shorter requests may wait until non-harvestable requests finish. Furthermore, smaller *HP* makes the wait time of longer requests reach the maximum wait time threshold faster because longer requests either may not be selected for execution or may relinquish all of their resources frequently to shorter requests. These two behaviors make the wait time of shorter requests longer if they arrive when non-harvestable requests are using all of the system resources (Figure 5.). However, as *HP* increases, the total wait time also increases because shorter requests may not be executed even though harvestable longer requests are using resources. Figure 6. shows that as *WP* increases, the average performance improves in both workloads. This is quite straightforward because with a very large *WP*, whenever shorter requests arrive, they acquire resources from longer requests. On the other hand, if a small *WP* is used, as the maximum wait time threshold of each request becomes small, they become non-harvestable quickly.

6. Conclusions

Network services make it easy to share software components across the network. However, deploying network services is not an easy task since there is no existing infrastructure to utilize. In this paper, we proposed adaptive resource management system that such infrastructure should possess. Our preliminary results show that resource harvesting can achieve up to 50% performance improvement.

Acknowledgements

This work was sponsored in part by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-2-0014.

7. References

[1] R. Armstrong et al., "Towards a Common Components Architecture for High-Performance Scientific Computing", Proceedings of the 8th International Symposiums on High Performance Distributed Computing, 1999.

[2] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", IEEE Communication Magazine, 14(2), 1997.

[3] Web Service Description Languages(WSDL) 1.1 W3C Note, http://www.w3.org/TR/wsdl.