Building Online Performance Models of Grid Middleware with Fine-Grained Load-Balancing: A Globus Toolkit Case Study

Ramon Nou[1], Samuel Kounev[2], and Jordi Torres[1]

[1] Barcelona Supercomputing Center (BSC), Technical University of Catalonia (UPC)
Barcelona Spain
{rnou,torres}@ac.upc.edu
[2] University of Cambridge Computer Laboratory, Cambridge, CB3 0FD UK
skounev@acm.org

Abstract. As Grid computing increasingly enters the commercial domain, performance and Quality of Service (QoS) issues are becoming a major concern. To guarantee that QoS requirements are continuously satisfied, the Grid middleware must be capable of predicting the application performance on the fly when deciding how to distribute the workload among the available resources. One way to achieve this is by using online performance models that get generated and analyzed on the fly. In this paper, we present a novel case study with the Globus Toolkit in which we show how performance models can be generated dynamically and used to provide online performance prediction capabilities. We have augmented the Grid middleware with an online performance prediction component that can be called at any time during operation to predict the Grid performance for a given resource allocation and load-balancing strategy. We evaluate the quality of our performance prediction mechanism and present some experimental results that demonstrate its effectiveness and practicality. The framework we propose can be used to design intelligent QoS-aware resource allocation and admission control mechanisms.

1 Introduction

Having established itself as a major computing paradigm for advanced science and engineering, Grid computing is now promising to become the future computing paradigm for enterprise computing and distributed system integration [1,2]. By enabling flexible, secure and coordinated sharing of resources and services among dynamic collections of disparate organizations and parties, Grid computing provides a number of advantages to businesses, for example faster response to changing business needs, better utilization and service level performance, and lower IT operating costs [2]. However, as Grid computing increasingly enters the commercial domain, performance and QoS (Quality of Service) aspects, such as customer observed response times and throughput, are becoming a major concern. The inherent complexity, heterogeneity and dynamics of Grid computing environments pose some challenges in managing their capacity to ensure that QoS requirements are continuously met.

K. Wolter (Ed.): EPEW 2007, LNCS 4748, pp. 125–140, 2007.

Enterprise grids are typically composed of heterogeneous components deployed in disjoint administrative domains, in highly distributed and dynamic environments. The resource allocation and job scheduling mechanisms used at the global and local level play a critical role for the performance and availability of Grid applications [3]. To prevent resource congestion and unavailability, it is essential that admission control mechanisms are employed by local resource managers. Furthermore, to achieve maximum performance, the Grid middleware must be smart enough to schedule tasks in such a way that the workload is load-balanced among the available resources and they are all roughly equally utilized. However, in order to guarantee that QoS requirements are satisfied, the Grid middleware must be capable of predicting the application performance when deciding how to distribute the workload among the available resources. Prediction capabilities are prerequisite to implementing intelligent QoS-aware resource allocation and admission control mechanisms.

Performance prediction in the context of traditional enterprise systems is typically done by means of performance models that capture the major aspects of system behavior under load [4]. Numerous performance prediction and capacity planning techniques for conventional distributed systems, most of them based on analytic or simulation models, have been developed and used in the industry. However, these techniques generally assume that the system is static and that dedicated resources are used. To address the need for performance prediction in Grid environments, new techniques are needed that use performance models generated on the fly to reflect changes in the environment. The term *online performance models* was recently coined for this type of models [5]. The *online* use of performance models defers from their traditional use in capacity planning in that configurations and workloads are analyzed that reflect the real system over relatively short periods of time. Since performance analysis is carried out on the fly, it is essential that the process of generating and analyzing the models is completely automated.

In this paper, we present a case study with the Globus Toolkit [6], the world's leading open-source framework for building Grid infrastructures. We have augmented the Grid middleware with an online performance prediction component that can be called at any time during operation to predict the Grid performance for a given resource allocation and load-balancing strategy. The case study shows how performance models can be generated dynamically and used to provide online performance prediction capabilities. We employ hierarchical queueing Petri net models that are dynamically composed to reflect the system configuration and workload. Queueing Petri nets make it possible to accurately model the behavior of our resource allocation and load balancing mechanism which combines hardware and software aspects of system behavior. Moreover, queueing Petri nets have been shown to lend themselves very well to modeling distributed component-based systems [7] which are commonly used as building blocks of Grid infrastructures [8]. We have evaluated the quality of our online performance prediction mechanism and present some results that demonstrate its effectiveness and practicality. The framework presented in this paper can be used as a basis to implement intelligent mechanisms for QoS-aware resource allocation, load-balancing and admission control. Finally, although our approach is targeted at Grid computing environments, it is not in

any way limited to such environments and can be readily applied in the context of more general Service-Oriented Architectures (SOA).

The paper is structured as follows. We start by introducing the Globus Toolkit and discussing some of our previous work related to the paper in Section 2. Section 3 presents our approach to online performance prediction. In Section 4, we present our case study and the experimental evaluation of our performance prediction mechanism. Finally, in Section 5 we present some concluding remarks and discuss our future work.

2 The Globus Toolkit

The Globus Toolkit (GT) is a community-based, open-architecture, open-source set of services and software libraries that support Grids and Grid applications [6]. The toolkit addresses issues of security, information discovery, resource management, data management, communication, fault detection, and portability. Globus Toolkit mechanisms are in use at hundreds of sites and by dozens of major Grid projects worldwide.

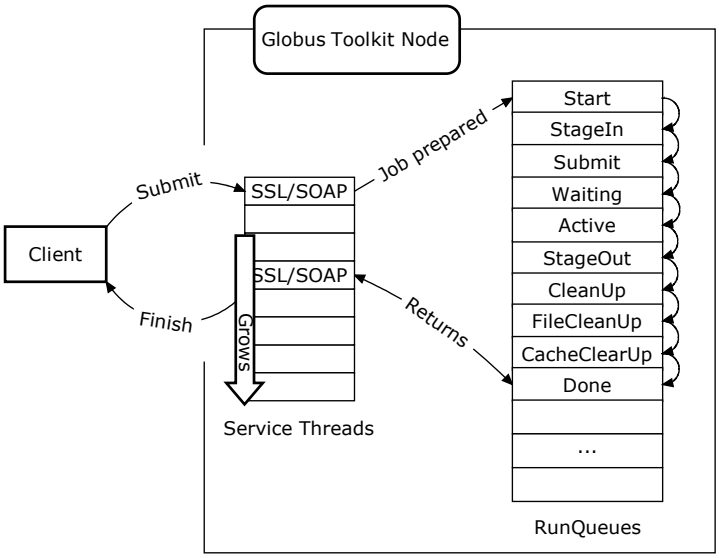

Fig. 1. Job Workflow in Globus Toolkit 4

Unfortunately, despite its popularity and success, the current implementation of the Globus Toolkit (GT4) exhibits very poor performance and reliability when the Grid middleware is overloaded. In our previous work [9,10], we have studied the behavior of Globus under heavy load and proposed enhancing the Grid middleware with a self-management layer to improve its performance and reliability under load [11]. In this paper, we show how the Grid middleware can be further enhanced with online performance prediction capabilities that are required in order to implement intelligent QoS

control mechanisms. We plan to use the online performance prediction framework proposed in this paper to extend our self-management layer with some more sophisticated QoS-aware resource allocation and admission control mechanisms.

We now take a brief look at the internal flow of control in Globus when processing jobs. Figure 1 shows the workflow of a job executed by Globus. When a client submits a job to a Globus server (using the *globusrun-ws* interface in its non-batch working mode), the job is picked by a ServiceThread which performs an SSL handshake. After the handshake, the SOAP request is parsed preparing the job for execution. The job is then started and proceeds through several stages as shown in Figure 1. At each stage the job is placed in a RunQueue and processed by a pool of threads. The most important stage is when the job is executed at the OS level. This is done by a separate OS process forked by Globus. After the job execution finishes, it goes through several clean up stages (RunQueues) and finally a ServiceThread generates the SOAP response and sends it back to the client.

3 Modeling Approach

Formally, a Grid environment based on the Globus Toolkit can be represented as a 4-tuple $G = (S,V,F,C)$ where:

- $\mathbf{S} = \{s_1, s_2, ..., s_m\}$ is the set of Grid servers,
- $\mathbf{V} = \{v_1, v_2, ..., v_n\}$ is the overall set of services offered by the Grid servers,
- $\mathbf{F} \in [S \longrightarrow 2^V]$[1] is a function assigning a set of services to each Grid server. Since Grids are typically heterogeneous in nature, we assume that, depending on the platform they are running on, Grid servers might offer different subsets of the overall set of services,
- $\mathbf{C} = \{c_1, c_2, ..., c_l\}$ is the set of currently active client sessions. Each session $c \in C$ is a tuple (v, λ) where $v \in V$ is the service used and λ is the rate at which requests for the service arrive.

3.1 Scheduling Mechanism

We have implemented a configurable *service request dispatcher* that provides a flexible scheduling and load-balancing mechanism for service requests. It is assumed that for each client session, a given number of threads (from 0 to unlimited) is allocated on each Grid server offering the respective service. Incoming service requests are then load-balanced across the servers according to thread availability. Threads serve to limit the concurrent requests executed on each server, so that different scheduling strategies can be enforced.

A scheduling strategy can be represented by a function $T \in [C \times S \longrightarrow \mathbb{N}_0 \cup \{\infty\}]$ which will be referred to as *thread allocation function*. The service request dispatcher queues incoming service requests (as part of a client session) and schedules them for service at the Grid servers as threads become available. Note that threads are used here

[1] 2^V denotes the set of all possible subsets of V, i.e. the power set.

as a *logical* entity to enforce the desired concurrency level on each server. Thread management is done entirely by the service request dispatcher and there is no need for Grid servers to know anything about the client sessions and how many threads are allocated to each of them. While the service request dispatcher might use a separate physical thread for each logical thread allocated to a session, this is not required by the architecture and there are many ways to avoid doing this in the interest of performance. For maximum scalability, multiple service request dispatchers can be instantiated and they can be distributed across multiple machines if needed.

Service request dispatchers completely decouple the Grid clients from the Grid servers which provides some important advantages that are especially relevant to commercial Grid environments. First of all, the decoupling enables us to introduce *fine-grained* load-balancing at the service request level, as opposed to the session level. Second, service request dispatchers make it possible to load-balance requests across heterogeneous server resources without relying on any platform-specific scheduling or load-balancing mechanisms. Finally, since clients do not interact with the servers directly, it is possible to adjust the resource allocation and load-balancing strategies dynamically.

3.2 Online Performance Prediction

In order to enhance the Grid middleware with online performance prediction capabilities, we have developed an *online performance prediction component* that can be called at any time during operation to predict the Grid performance for a given scheduling strategy represented by a thread allocation function T. Performance prediction is carried out by means of an online performance model generated and analyzed on the fly. The online performance prediction component can be used to find an optimal scheduling strategy that satisfies the client SLAs under given resource utilization constraints. Based on this, intelligent QoS-aware admission control mechanisms can be developed. For example, when a client sends a request to start a new session, the scheduler can reject the request if it is not able to find a scheduling strategy that satisfies the client SLAs. The design of intelligent mechanisms for QoS control is outside the scope of this paper. In the following, we focus on the performance prediction component and evaluate its effectiveness in the context of a real-life Globus deployment.

The performance prediction component is made of two subcomponents - *model generator* and *model solver*. The model generator automatically constructs a performance model based on the active client sessions and the available Grid servers. The model solver is used to analyze the model either analytically or through simulation. Different types of performance models can be used to implement the performance prediction component. In this paper, we use Queueing Petri Nets (QPNs) which provide greater modeling power and expressiveness than conventional modeling formalisms like queueing networks, extended queueing networks and generalized stochastic Petri nets [12,13,14]. In [7], it was shown that QPN models lend themselves very well to modeling distributed component-based systems and provide a number of important benefits such as improved modeling accuracy and representativeness. The expressiveness that QPNs models offer makes it possible to model the logical threads used in our scheduling mechanism accurately. Depending on the size of QPN models, different

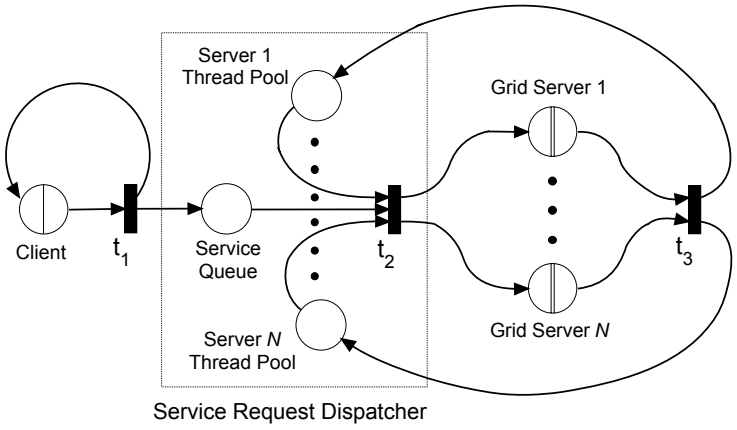

Fig. 2. High-level QPN model of the Grid environment

methods can be used for their analysis, from product-form analytical solution methods [15] to highly optimized simulation techniques [16].

Figure 2 shows a hierarchical QPN model of a set of Grid servers accessed through our service request dispatcher. The Grid servers are modeled with nested QPNs represented as subnet places. The *Client* place contains a $G/G/\infty/IS$ queue which models the arrival of service requests sent by clients. Service requests are modeled using tokens of different colors, each color representing a client session. For each active session, there is always one token in the Client place. When the token leaves the Client queue, transition t_1 fires moving the token to place *Service Queue* (representing the arrival of a service request) and depositing a new copy of it in the Client queue. This new token represents the next service request which is delayed in the Client queue for the request interarrival time. An arbitrary request interarrival time distribution can be used. For each Grid server, the service request dispatcher has a *Server Thread Pool* place containing tokens representing the logical threads on this server allocated to the different sessions (using colors to distinguish between them). An arriving service request is queued at place Service Queue and waits until a thread for its session becomes available. When this happens, the request is sent to the subnet place representing the respective Grid server. After the request is processed, the logical service thread is returned back to the thread pool from where it was taken. By encapsulating the internal details of Grid servers in separate nested QPNs, we decouple them from the high-level performance model. Different servers can be modeled at different level of detail depending on the complexity of the services they offer.

At each point in time, the online performance prediction component keeps track of the active client sessions and the currently available Grid servers. It is assumed that when servers are added to the Grid, for every server a performance model is provided in the form of a nested QPN that captures the server capacity and its internal behavior when processing service requests. When invoked, the performance prediction component uses the models of the Grid servers to dynamically construct an up-to-date model

of the Grid environment that reflects the current workload (in terms of active client sessions) and the currently available server resources. The model is constructed by integrating the Grid server models into the high-level QPN model discussed above. The model generation and analysis is completely automated and happens on the fly.

4 Case Study

In this section, we present our case study of a deployment of GT4 which we have enhanced with online performance prediction functionality as described in the previous section. We evaluate the quality of our performance prediction mechanism and present some experimental results that demonstrate its effectiveness. Our testing environment depicted in Figure 3 consists of two heterogeneous Grid servers, the first one 2-way Pentium Xeon at 2.4 GHz with 2 GB of memory and the second one 4-way Pentium Xeon at 1.4 GHz with 4 GB of memory. Both servers run Globus Toolkit 4.0.3 (with the latest patches) on a Sun 1.5.0_06 JVM. The Grid clients are emulated on a separate machine with identical hardware as the first Grid server. The machines communicate over a Gigabit network.

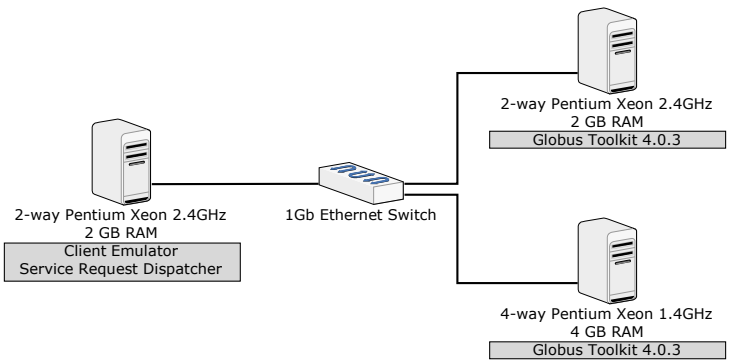

Fig. 3. High-level view of our testing environment

4.1 Workload Characterization

As a basis for our experiments, we use several sample jobs each executing some business logic requiring a given amount of CPU time. Some of the jobs include calls to external (third-party) service providers that are not part of the Grid environment. In order to build performance models of the two Grid servers, we must first characterize their workload in terms of the service demands of the jobs they execute.

There are several approaches to determining the CPU service demands. The most reliable method is to use a Globus profiler to measure the CPU service times directly. We can use the BSC Monitoring Framework (BSC-MF) [9] developed at the Barcelona Supercomputing Center in conjunction with the Paraver performance analysis tool [17]. Another approach which does not require profiling Globus is to estimate the service

times based on measured CPU utilization and job throughput data. This approach is very general and does not require any profiling tools. For each job type, an experiment is run injecting jobs of the respective type. Based on the utilization law [18], we can then compute the average job service demand D as the ratio of the measured CPU utilization U to the job throughput X. In certain cases, techniques can be employed that help to estimate the service demands without the need to do any measurements on the system [19]. Such techniques are based on analyzing the business logic that jobs execute at the source code level.

Table 1 shows the service demands of the sample jobs we analyzed. For each job type, the internal job processing CPU time is shown, the time waited for external service providers, the measured total job CPU service demand and the job management overhead introduced by Globus. The overhead is consistently around 1 second across the seven job types.

Table 1. Estimated job service demands and Globus processing overhead (sec)

Job	A	B	C	D	E	F	G
Internal job processing CPU time	20.00	10.00	6.00	5.00	4.00	1.00	0.50
External service provider time	5.00	0.00	2.00	0.00	3.00	0.20	0.00
Job CPU service demand	21.00	11.00	6.89	5.84	4.79	1.93	1.54
Globus management overhead	1.00	1.00	0.89	0.84	0.79	0.93	1.04

In the rest of the case study, we concentrate on the middle three jobs (C, D and E), which we analyze in more detail. We assume that these jobs are exposed as three separate *services* offered by the Grid servers. Table 2 shows the service demands of the three services at the two Grid servers.

Table 2. Service demands of workload services (sec)

Service	Service 1 (Job C)	Service 2 (Job E)	Service 3 (Job D)
CPU service demand on the 2-way server	6.89	4.79	5.84
CPU service demand on the 4-way server	7.72	5.68	6.49
External service provider time	2.00	3.00	0.00

4.2 Grid Server Models

We assume that when Grid servers join the Grid they first register with the online performance prediction component responsible for the local environment. Each server provides a performance model in the form of a nested QPN that captures its internal behavior when processing service requests. When invoked, the performance prediction component dynamically constructs an up-to-date model of the Grid environment by integrating the Grid server models into the high-level model presented in Section 3.2 (see Figure 2). The two servers used in our case study were each modeled using a

nested QPN as shown in Figure 4. Service requests arriving at a Grid server circulate between queueing place *Server CPUs* and queueing place *Service Providers*, which model the time spent using the server CPUs and the time spent waiting for external service providers, respectively. Place *Server CPUs* contains a $G/M/m/PS$ queue where m is the number of CPUs, whereas place *Service Providers* contains a $G/M/\infty/IS$ queue. The service times of service requests at these queues are set according to the measured service demands shown on Table 2. For simplicity, we assume that the service times at the server CPUs, the request interarrival times and the times spent waiting for external service providers are all exponentially distributed. In the general case this is not required. The firing weights of transition t_2 are set in such a way that place *Service Providers* is visited one time for Services 1 and 2 and it is not visited for Service 3. The model solver component of the performance prediction component was implemented using SimQPN - our highly optimized simulation engine for QPNs [16].

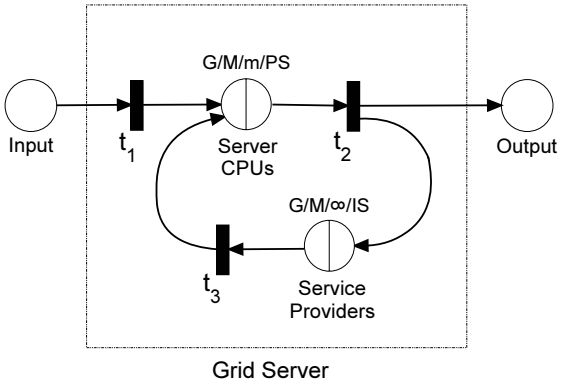

Fig. 4. Grid server QPN model

Before the Grid server models can be used for performance prediction they must be validated. This is normally done by comparing the model predictions against measurements on the real system. Our initial attempts to validate the model revealed that predictions were accurate in scenarios with no limitation on the number of concurrently scheduled requests and much less accurate in scenarios with limited number of threads allocated to client sessions (see Section 3.1). Table 3 shows four of the scenarios we considered. Given that for some scenarios the error was higher than 15%, the models could not pass our initial validation attempt. To investigate the problem, we analyzed the internal behavior of Globus when processing service requests. Figure 5 shows the stages and the CPU usage during the processing of a request for Service 3 (job D) in isolation. This view was obtained from a trace generated by BSC-MF [9] and processed using Paraver [17]. The black zones show periods when a CPU was used and the zones in between, marked with red horizontal lines, correspond to periods during which all CPUs were idle (in total about 1 sec). The idle CPU periods were occurring during job

state transitions. Note that the service was executed in single user mode and therefore the idle CPU periods were not being caused by contention for software or hardware resources, neither were they being caused by I/O, since the disk utilization was negligible. Taking these "hidden internal delays" introduced by Globus into account helped us to understand why the model predictions were much less accurate in the case with limited concurrency. Indeed in this case, given that there are limited threads available and client requests have to wait at the service request dispatcher to obtain a thread, a difference of 1 sec in the time spent by jobs on the server obviously would have a much bigger impact on the overall response time than in the case with unlimited threads. Monitoring Globus under load with multiple concurrent requests revealed that the idle CPU periods during service execution were pretty much constant and were not affected much by the workload intensity or transaction mix. Having concluded this, we decided to calibrate our models by introducing an additional 1 sec delay during service execution. For simplicity, we added this delay to the time waited in place *Service Providers*. After this calibration, the discrepancy between the model predictions and measurements on the real system disappeared.

Table 3. Model predictions before calibration

Services	No of threads allocated	Request interarrival time (sec)	Request response time (sec)		Error (%)
			measured	predicted	
2	unlimited	4	11.43	10.47±0.033	8.3%
1—3	unlimited	8 / 8	13.66 / 12.91	12.21±0.019 / 11.17±0.031	11% / 13%
3	5	2.5	10.93	8.14±0.030	**25%**
1—3	2/2	8 / 8	18.15 / 9.79	15.58±0.23 / 7.8±0.05	14.1% / **20.3%**

Fig. 5. Paraver view of a job execution inside Globus

An alternative approach to model the Grid environment is to use a general purpose simulation system such as OMNeT++ [20] which is based on message-passing. We have used OMNeT++ successfully to model a Tomcat Web server [21] with software admission control. Figure 6 compares the precision of interval estimates provided by SimQPN and OMNeT++ when simulating a model of our Grid environment described above with several concurrent client sessions. The precision is measured in terms of the maximum width of 95% confidence intervals for job response times. For run times below 1 second, SimQPN provided slightly wider confidence intervals than OMNeT++, however, there was hardly any difference for run times greater than 1 second. At the same time, while OMNeT++ results were limited to job response times, SimQPN results were more comprehensive and included estimates of job throughputs, server utilization,

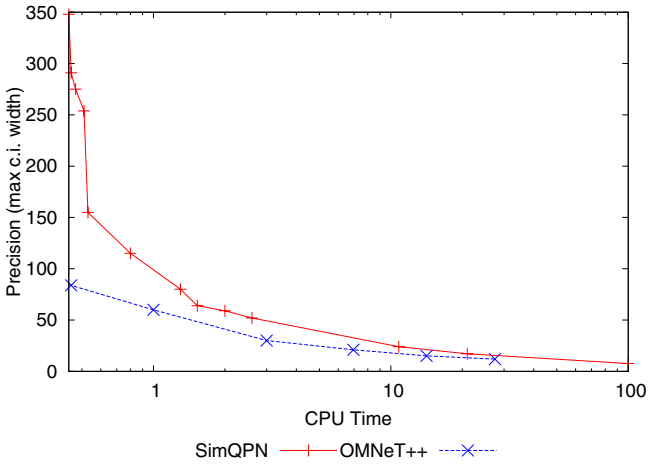

Fig. 6. Precision of interval estimates provided by SimQPN and OMNeT++ for a given simulation run time (sec)

queue lengths, etc. Moreover, QPN models have the advantage that they are much easier to build and, as discussed in Section 3.2, they can be hierarchically composed which facilitates the dynamic model construction. The hierarchical composition is essential since it introduces a clear separation of the high-level system model from the individual Grid server models. The latter can be developed independantly without knowing in which environment they will be used.

4.3 Experimental Results

We now evaluate the quality of our online performance prediction mechanism in the context of the scenario described above. We have developed a client emulator framework that emulates client sessions sending requests to the Grid environment. The user can configure the target session mix specifying for each session the service used and the time between successive service requests (the interarrival time). Client requests are received by the service request dispatcher and forwarded to the Grid servers according to the configured scheduling strategy as described in Section 3.1. Figure 7 illustrates the flow of control when processing service requests.

Whenever the online performance prediction component is invoked, it uses the QPN models of the Grid servers to dynamically construct a QPN model of the Grid environment that reflects the current workload in terms of active client sessions and the selected scheduling strategy. The generated model is then analyzed by means of simulation using SimQPN. The method of non-overlapping batch means was used with a batch size of 300 and the simulation was configured to run sequentially until the half-widths of 95% confidence intervals for response times dropped below half a second.

We used the online performance prediction component to predict the Grid performance under a number of different workload and configuration scenarios varying the

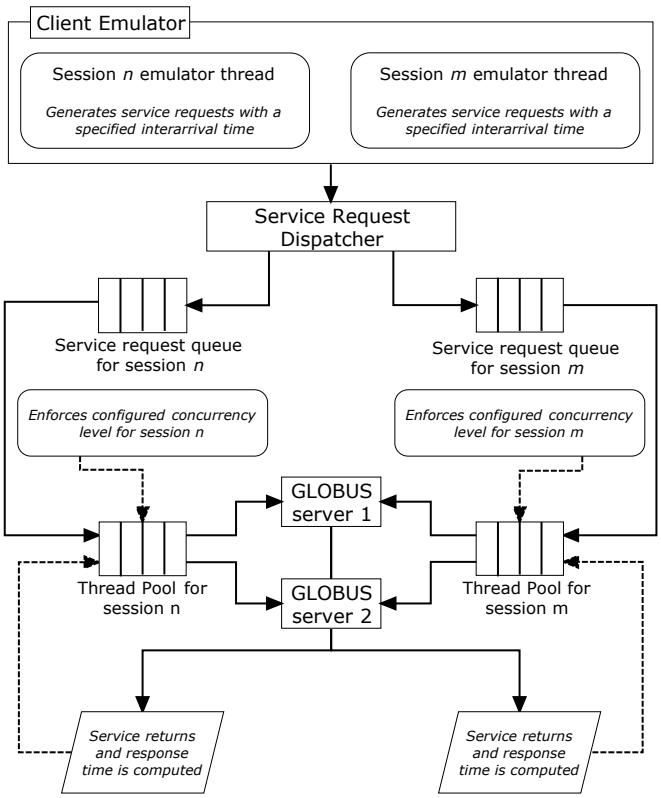

Fig. 7. Flow of control when processing service requests

session mix, the scheduling strategy and the number of servers available. Table 4 presents the results from an experiment in which the workload evolved through five different stages each one with different session mix and duration between 1200 and 3600 seconds. In the beginning of each stage, the scheduling strategy was modified dynamically and the online performance prediction component was called to predict the system performance on the fly. The average time needed to predict the system performance (including model generation and analysis) was 6.12 seconds, the maximum was 12 seconds. Performance predictions were recorded and were later compared against the actual performance measured during the run. The experiment was repeated multiple times and exhibited negligible variation in the measured performance metrics. Table 4 shows the results from comparing the model predictions against the measurements on the system. 95% confidence intervals for response times are provided. The mean modeling error was only 7.9% (with standard deviation 6.08) and it did not exceed 22.5% across all scenarios. We conducted a similar experiment for a number of different workload and configuration scenarios. The results from the analysis were of similar accuracy

Table 4. Comparison of model predictions against measurements on the real system

Services	No of threads allocated		Request interarrival time (sec)	Request response time (sec)		Error (sec)	Avg. CPU utilization	
	server 1	server 2		measured	predicted		measured	predicted
1	3	2	12.5	13.86	13.23±0.591	+0.63	0.66	0.65
1	5	3	13	14.57	13.41±0.622	+1.16		
2	2	4	11	11.36	10.63±0.432	+0.73		
2	1	2	12	11.07	10.49±0.452	+0.58		
2	5	4	15	12.49	10.99±0.453	+1.5		
3	1	5	13	9.37	8.36±0.331	+1.01		
3	1	2	16	9.41	8.53±0.363	+0.88		
3	4	4	16	10.17	9.79±0.520	+0.38		
1	1	10	12.5	11.61	11.13±0.597	+0.48	0.56	0.58
1	4	7	15	11.95	12.57±0.736	-0.62		
2	8	6	15	12.20	10.45±0.567	+1.75		
2	4	6	9	10.67	10.45±0.466	+0.22		
2	3	8	15.5	10.59	10.39±0.542	+0.2		
3	7	3	16	10.90	8.93±0.577	+1.97		
3	10	2	12	11.67	9.01±0.508	+2.66		
1	5	2	12.5	13.81	13.81±0.669	+0	0.64	0.66
1	2	1	13	13.76	14.19±0.750	-0.43		
2	1	4	11	10.4	10.2±0.532	+0.2		
2	1	5	12	10.3	10.18±0.507	+0.12		
2	4	1	15	13.45	11.80±0.479	+1.65		
3	1	1	13	9.61	9.7±0.504	-0.09		
3	3	2	16	10.9	9.82±0.627	+1.08		
3	2	2	16	9.51	9.57±0.617	-0.06		
1	1	1	14.5	12.36	14.28±0.443	-1.92	0.61	0.63
1	3	2	17	13.40	13.2±0.393	+0.2		
1	5	3	18	15.51	13.2±0.438	+2.31		
2	5	1	15	13.19	11.5±0.336	+1.69		
2	4	3	25	11.88	11.16±0.405	+0.72		
2	1	3	15	10.29	10.13±0.302	+0.16		
2	5	2	25.5	13.3	11.11±0.412	+2.19		
3	4	4	16	9.48	9.66±0.317	-0.18		
3	1	4	16	8.25	8.38±0.234	-0.13		
3	3	3	25	8.65	9.56±0.340	-0.91		
1	4	2	18.5	13.35	14.77±0.623	-1.42	0.64	0.65
1	4	5	15	12.05	14.56±0.618	-2.51		
1	4	4	16	12.58	14.50±0.673	-1.91		
2	1	2	15	10.67	10.78±0.460	-0.11		
2	5	1	17	11.86	12.59±0.534	-0.73		
2	2	2	19	11.09	11.51±0.492	-0.42		
2	4	2	15.5	11.53	11.99±0.528	-0.46		
3	2	5	16	9.42	10.1±0.475	-0.68		
3	1	4	19	8.98	9.14±0.413	-0.16		
3	3	5	23	8.92	10.57±0.535	-1.65		

as the ones presented here and demonstrated the effectiveness of our performance prediction mechanism. The computational overhead of the online performance prediction component was measured to be less than 60 sec for scenarios with up to 40 Grid servers and 80 concurrent sessions.

5 Conclusions and Future Work

In this paper, we presented a novel case study with the Globus Toolkit, the world's leading open-source framework for building Grid infrastructures, in which we showed how performance models can be generated dynamically and used to provide online performance prediction capabilities. We have augmented the Grid middleware with an online performance prediction component that can be called at any time during operation to predict the Grid performance for a given resource allocation and load-balancing strategy. The quality of our performance prediction mechanism has been evaluated under a number of different workload and configuration scenarios varying the session mix, the scheduling strategy and the number of servers available. We presented some experimental results that demonstrated the effectiveness, practicality and performance of our approach. The modeling error of predicted response times was only 7.9% on average (with standard deviation of 6.08) and it did not exceed 22.5% across all considered scenarios. The framework we propose provides a basis for designing intelligent QoS-aware resource allocation and admission control mechanisms.

Our performance prediction mechanism is based on hierarchical queueing Petri net models that are dynamically composed to reflect the system configuration and workload. Using queueing Petri nets we could accurately model the resource allocation and load balancing mechanism which combines hardware and software aspects of system behavior. Moreover, queueing Petri nets provide great flexibility in choosing the level of detail and accuracy at which system components are modeled. To the best of our knowledge, this is the first application of queueing Petri nets as online performance models.

The area considered in this paper has many different facets that will be subject of future work. We are currently working on extending the Globus Toolkit with online QoS control functionality. Based on the online performance prediction mechanism proposed in this paper, we are building a framework for QoS-aware resource allocation and admission control. The framework includes an intelligent QoS broker component that negotiates QoS goals and SLAs with Grid clients before making a commitment. Taken collectively these enhancements will not only provide sophisticated QoS control capabilities but can also be exploited to make the Grid middleware self-configurable and adaptable to changes in the system environment and workload. Another aspect we intend to investigate is how our framework can be extended to take into account the costs associated with using the Grid resources when negotiating QoS targets.

Acknowledgments

This work was supported by the Spanish Ministry of Science and Technology, the European Union under contract TIN2004-07739-C02-01, and the German Research Foundation under grant KO 3445/1-1. We acknowledge the support of our colleague Ferran Julià from the Technical University of Catalonia in resolving many technical issues.

References

1. Foster, I., Kesselman, C., Nick, J.M., Tuecke, S.: Grid Services for Distributed System Integration. Computer 35(6), 37–46 (2002)
2. OGF: Open Grid Forum, `http://www.ogf.org`

3. Menascé, D., Casalicchio, E.: A Framework for Resource Allocation in Grid Computing. In: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, IEEE Computer Society Press, Los Alamitos (2004)
4. Menascé, D.A., Almeida, V.A.F., Dowdy, L.W.: Performance by Design. Prentice-Hall, Englewood Cliffs (2004)
5. Menascé, D., Bennani, M., Ruan, H.: On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems (chapter On the Use of Online Analytic Performance Models in Self-Managing and Self-Organizing Computer Systems). In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A., van Steen, M. (eds.) Self-star Properties in Complex Information Systems. LNCS, vol. 3460, Springer, Heidelberg (2005)
6. Foster, I.T.: Globus Toolkit Version 4: Software for Service-Oriented Systems. In: Proceedings of the 2005 IFIP International Conference on Network and Parallel Computing, pp. 2–13 (2005)
7. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. IEEE Transactions on Software Engineering 32(7), 486–502 (2006)
8. Foster, I., Kesselman, C.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (2003)
9. Nou, R., Julia, F., Carrera, D., Hogan, K., Caubet, J., Labarta, J., Torres, J.: Monitoring and analysis framework for grid middleware. In: PDP, pp. 129–133. IEEE Computer Society, Los Alamitos (2007)
10. Nou, R., Juliá, F., Torres, J.: Should the grid middleware look to self-managing capabilities? In: The 8th International Symposium on Autonomous Decentralized Systems (ISADS 2007), Sedona, Arizona (2007)
11. Nou, R., Juliá, F., Torres, J.: The need for self-managed access nodes in grid environments. In: 4th IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2007), IEEE Computer Society Press, Los Alamitos (2007)
12. Bause, F.: "QN + PN = QPN" - Combining Queueing Networks and Petri Nets. Technical report no.461, Department of CS, University of Dortmund, Germany (1993)
13. Bause, F., Buchholz, P., Kemper, P.: Integrating Software and Hardware Performance Models Using Hierarchical Queueing Petri Nets. In: Proc. of the 9. ITG / GI - Fachtagung Messung, Modellierung und Bewertung von Rechen- und Kommunikationssystemen (1997)
14. Kounev, S., Buchmann, A.: Performance modelling of distributed e-business applications using queuing petri nets. In: Proc. of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03), mar 2003, IEEE Computer Society Press, Los Alamitos (2003)
15. Bause, F., Buchholz, P.: Queueing Petri Nets with Product Form Solution. Performance Evaluation 32(4), 265–299 (1998)
16. Kounev, S., Buchmann, A.: SimQPN - a tool and methodology for analyzing queueing Petri net models by means of simulation. Performance Evaluation 63(4-5), 364–394 (2006)
17. Jost, G., Jin, H., Labarta, J., Gimenez, J., Caubet, J.: Performance analysis of multilevel parallel applications on shared memory architectures. International Parallel and Distributed Processing Symposium (IPDPS), Nice, France (2003)
18. Denning, P.J., Buzen, J.P.: The Operational Analysis of Queueing Network Models. ACM Computing Surveys 10(3), 225–261 (1978)

19. Menascé, D., Gomaa, H.: A Method for Desigh and Performance Modeling of Client/Server Systems. IEEE Transactions on Software Engineering 26(11) (2000)
20. Varga, A.: The OMNeT++ discrete event simulation system. In: European Simulation Multiconference (ESM'2001) (June 2001)
21. Nou, R., Guitart, J., Torres, J.: Simulating and modeling secure web applications. In: Alexandrov, V.N., van Albada, G.D., Sloot, P.M.A., Dongarra, J.J. (eds.) ICCS 2006. LNCS, vol. 3991, pp. 84–91. Springer, Heidelberg (2006)