Performance Prediction & Physical Design of J2EE Based Web Applications

UMESH BELLUR, AKHILESH SHIRABATE
School of Information Technology
Indian Institute of Technology, Bombay
Powai, Mumbai - 400076
INDIA

Abstract: - With the widespread use of N tier computing technologies in the enterprise and the increased dependence on the services provided by these applications, greater emphasis is now being placed on the performance and health of these applications. It's now not enough that the application merely functions, but it needs to meet the QoS[1] metrics that is expected of it. It turns out that QoS is dependent, not just on how well the application has been designed and built but also on how is gets deployed and distributed. We present here ongoing work in the area of autonomic computing of distributed component based applications – an effort that seeks to make distributed enterprise applications and the environments in which they run self-configuring and self-healing.

Key-Words: - J2EE, performance prediction, analytical models, queueing Petri-nets, QoS of Web Applications.

[1] QoS – Quality of Service

1. Introduction

As distributed applications become mainstream enterprise solutions, there have been considerable advances in making the development of these applications simpler. The development of server side component models followed by standardization of server side "software containers" to host these components have helped considerably shorten the development lifecycles of large applications. Indeed it is not uncommon to see release cycles of 6 months or less in the enterprise for major features and 3 months or less for minor feature adds. In addition, the expectation on the application performance and reliability has gone up to the point where well defined QoS measures are expected to be adhered to by these complex distributed applications. The impact of these rapid application development paradigms has shifted the complexity from what used to be application development to deployment and beyond – tasks that are commonly handled by the IT Operations staff in the enterprise. Once the application has been developed, the first task would be to map it to a physical architecture given the expected workloads and the availability of shared physical resources (CPU, disk, network bandwidth etc.). Once resource mapping is done, the various resources need to be configured with the appropriate parameters to handle the application. This in itself is a task of great complexity not only because of the dependencies between the various components making up an application but also because one needs to map any QoS requirements of the application (such as response times and uptime) to the selection of the different physical components that the application will run on. For example, network QoS may have to be negotiated appropriately since network communication quality can have a significant impact on application performance of distributed applications. The complexity also arises from the numbers of parameters that have to be tuned on resources such as application servers and relational databases. The modern J2EE[2] application server has over 300 parameters that have to be tuned in order to extract the best value. Of late, there has been an increased focus on "autonomic computing" techniques – techniques that determine how application environments can configure and heal themselves in the event of problems. For example, an application server (or middleware server) can have over a hundred different parameters that have to be tuned and the configuration needs to be consistent with that of other servers that it may depend on.

We have started an effort to focus autonomic computing techniques on the system design, application deployment and problem diagnosis and correction of enterprise class distributed applications. For simplicity, we are looking only at J2EE applications currently although the techniques are likely to be useful across a variety of similar component models such as .NET and CORBA. This is termed the Lights-out Automated Management of Distributed Applications (LAMDA). As part of LAMDA we are investigating increasingly complex N tier architectural models for autonomic computing (self configuration and healing purposes) starting at the low end of a web application that needs no other resources such as the DB or any other business logic to the high end which consist of N tiers of servers with business logic, workflow, rules engines as well as a relational DB.

The first of these models is that of a web application that consists of a simple Web Server/Servlet container that hosts dynamic content generation pieces of Java code known as Servlets. As a simplification in our first model, Servlets execute independently and don't need to access backend resources for either business logic or data. Given a servlet, we are able to model it's execution analytically using Queuing Petri-net models which can be used to predict performance based on reward rates of the underlying Markov chains. We are using this information to then configure the Web Server and container according to the QoS needs of the application.

The rest of this paper is organized as follows. The next section gives an introduction to LAMDA and the different aspects of LAMDA that we are working on. It also defines the notions of "service" versus "application" and other terms that are used in the rest of the paper. After that, we discuss the different methods of analytical modeling that we can chose for such a problem. We then turn our attention to the specific problem of predicting

[2] J2EE is a trademark of SUN and denotes the server side Java component architecture commonly used to build enterprise applications today.

performance of JSPs/Servlets using analytical models in the selected modeling techniques. Finally we conclude with a discussion on the related efforts that we are aware of and the current status of the effort and future directions.

2. LAMDA

There are several facets to autonomic computing all of which form part of the LAMDA vision.

a. Physical Design and deployment – Self Configuration. There are two aspects to this – static and dynamic. Static design lays out certain constraints on location of the application components and maps it initially to a physical topology. The dynamic version ensures that these constraints continue to be met and may move application components, add or remove computing resources and reconfigure the infrastructure.

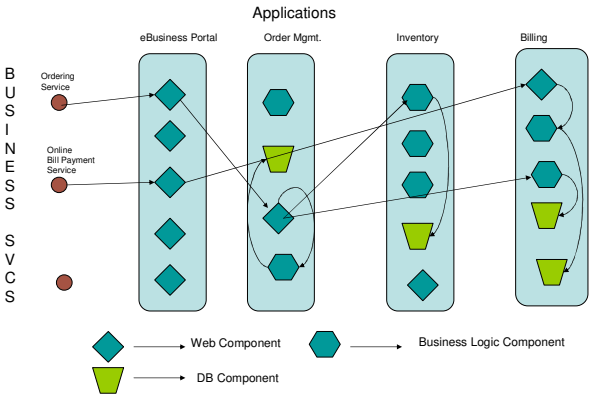
b. Root Cause Isolation and correction - Self Healing. Self healing can be for the purposes of correcting a structural constraint or property that has been broken such as those related to performance, availability or capacity.

c. Self Protection – Related to the second facet, this is for the purposes of healing a security breach that has occurred. The techniques and the basis for self protection are often very different from those used for self healing and so will be considered separately.

As a part of this effort (especially part a), we have also developed meta models for describing application and service QoS parameters and resource needs which we use in trying to come up with the physical design.

2.1 The Basis of LAMDA

2.1.1 Application versus Service



In LAMDA, we differentiate between applications and services as follows. Applications are considered as units of deployment which bind together a set of components to be deployed as a group. For example the Order Management application can have 2 EJBs[3] representing order processing business logic, a DB component representing the order schema and a set of JSPs[4] that represents the interface into ordering, order status determination etc.

Business services are transactions that have a clear customer access point such as a web site link or a GUI button that can start the transaction. Business services thread through various applications touching individual components along the way. Quality of Service (QoS) requirements should exist on business services such as the bill payment service will have availability of 99.9% with 85% of the transactions exhibiting response times of less than 1 second! Applications themselves may have individual QoS but that is relatively less important.

The two of these concepts are orthogonal. Developers are concerned with applications that encapsulate some functionality while IT administrators are concerned with managing services as seen by the customer.

2.1.2 Structural Basis - Topology

The starting point for self-healing or self

[3] EJB stands for Enterprise Java Beans which are server side components in the J2EE architecture.

[4] JSP stands for Java server pages which are server side pages than are used to generate dynamic web pages in web applications.

configuration is to know one self and so determining the topology of the application in relation to its execution environment is critical. An application cannot be deployed without knowledge of the various components that make it up. Both the static parts of the component (viz, it's packaging) as well as it's physical footprint need to be well understood for problem isolation and correction.
Topology therefore is a description of:

a. The infrastructure (both physical such as compute servers as well as logical such as server component containers), its configuration and its dependence on the underlying network.
b. The static view application components and their configurations.
c. The dynamic or run time view of application components that execute on the infrastructure. This specifies the physical footprint that the component exhibits at run time. For example, an EJB can be deployed on several J2EE containers either as a cluster or singly.
d. Dependencies that exist between application components, between application components and infrastructure (software, hardware and network).

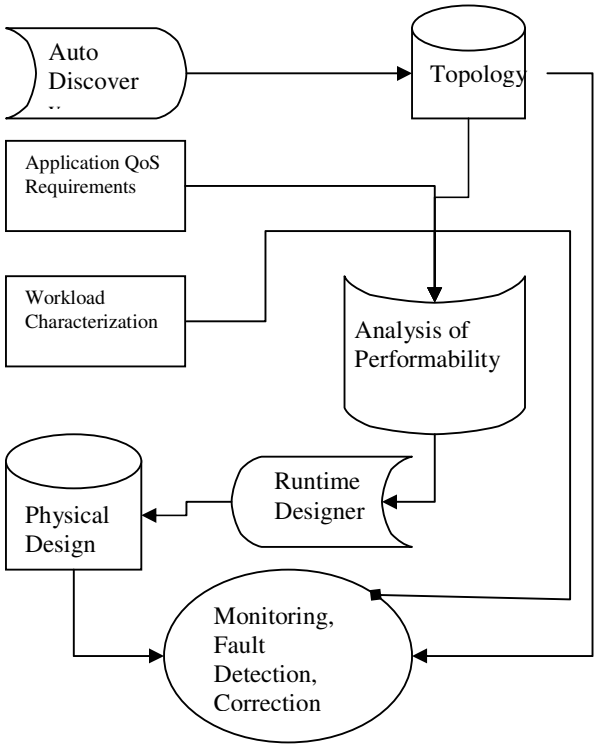
Topology is a realization of the meta-model that characterizes applications and their execution environments and provides a canonical language for common understanding of what an application is and what it depends on. Every tool in the LAMDA arsenal works off of topology. Since the topology of a distributed shared execution environment is constantly changing (applications are being added, removed or updated, machines are upgraded or added, the network is being tuned etc.), we need a process that will keep up-to-date the topology of the existing environment including any applications that are currently executing on it.

2.2 LAMDA Architecture

LAMDA is essentially a closed loop optimization process. The input to this process is a set of applications along with their QoS needs and expected workloads. Initial physical design is a byproduct of the analysis and optimization process of the architecture but we expect this is a continual process driven by changes in the underlying infrastructure as well as workloads.

The underlying infrastructure which is pre-built based on our knowledge of the functioning of the containers, is augmented with the knowledge about the topology of the application. So, for example, if the application calls for a particular servlet to talk with a specific DB schema, then we can build the underling analytical model for performance analysis. We then solve the analytical model and obtain the expected QoS under a particular physical design. This is iteratively refined by moving around components to optimize for the QoS parameters till we meet or beat the expected QoS of the application.

Of course, this optimization has to be performed with all the applications that share a common infrastructure, else it will not be of much use in a real environment. The same approach can be used to optimize the number of resources used as well and output the best expected QoS from the application.



3. Analytical Modeling Approach and Justification

Many modeling formalisms exist, some developed to do quantitative modeling such as queuing networks and some developed to do qualitative modeling such as Petri-Nets.

As developed originally, queuing networks were limited by their lack of exclusivity of synchronization constructs. The extended queuing networks (EQNs)[[12]] are an effort in the direction to remove this limitation of queuing network theory. EQNs are QNs that have been augmented with passive resources, fork nodes, join nodes and split nodes. Each passive resource consists of a number of tokens representing the resource units available for customers arriving at the allocation node. Special nodes are defined in the extended notation where customers can acquire, release, create or destroy a resource token. Besides EQN, there are various other models which extend the queuing models. A model called Queuing system with flag mechanisms [[14]] was proposed by D. Mailles et el. The model was based on the integration of concepts from petri nets into queuing networks.

Petri Nets or Place Transition nets on the other hand originated to perform qualitative (reachability) analysis and petri nets have simple ways of arranging the places and transitions to give various constructs. Some common constructs are sequence, choice, concurrency and synchronization. For further details about the place/transition nets, refer [13]. Petri Nets were then augmented with timing information for quantitative analysis. There are two principal ways of integrating timing aspects into Petri nets:

- specification of a dwelling time for tokens on a place (Timed places Petri nets - TPPNs)
- specification of a firing delay for enabled transitions (Timed transition Petri nets - TTPNs)

The most important representatives of TTPNs are stochastic Petri nets, e.g. generalized stochastic Petri nets (GSPNs), describing Markov processes. The main disadvantage of these time augmented Petri nets is the very difficult description of scheduling strategies. For easier description of scheduling strategies additional elements are often integrated into the stochastic Petri net world, like e.g. inhibitor arcs, with the problem of raising the modeling power up to Turing machines even for the untimed model causing the undecidability of important analysis problems [13].

These issues were gotten around by integrating the notion of queues into Petri net places and adding the notion of timed transitions to create Queued Petri Nets (QPN). The main idea in creation of the QPN [11] was to add timing aspects to the places of a (coloured) Petri net. In QPNs time is integrated in a more powerful way, because we don't restrict our model to the specification of a dwelling time for tokens. In QPNs a whole queue (station) may be integrated into the definition of a place. Such a timed place consists of two components, the queue (station) and a "repository of deposit" for served tokens (customers). The behaviour of the net is like follows. Tokens, fired by the input transitions of such a timed place, are inserted into the queue due to the specified scheduling strategy. Tokens in a queue are not available for the transitions of the QPN. After completion of service the token (customer) is placed on the "place of deposit". Tokens on this "repository" are available for all output transitions of the timed place. Like in TTPNs, an enabled timed transition will fire after a certain delay specified by a random variable. Enabled immediate transitions will fire due to relative firing frequencies. We assume that no token is generated or destroyed in a queue, so that qualitative analysis can be partially done by analyzing the underlying colored Petri net. We believe that these are the most effective modeling formalisms for the kinds of entities (Web servers and application servers) that need to be modeled. We are however interested mainly in the quantitative results delivered by solving the underlying Markov Chains.

4. Tomcat and It's QPN Model

In order to study the performance and subsequently apply it to physical design, we use the Apache Tomcat Servlet container as our reference architecture.

4.1 The Tomcat Concrete Architecture

Conceptually, TOMCAT is split into two parts - a connector which is tasked with handling the communication protocol and it's details and a backend server which is the actual Servlet

container.

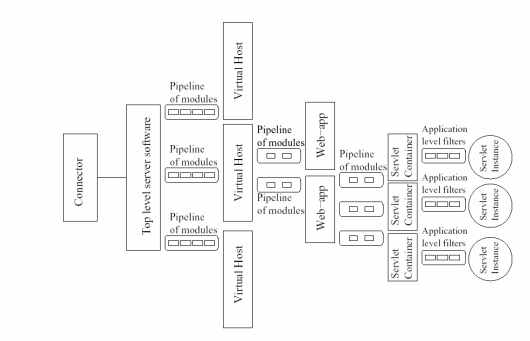


Figure 1: The Architecture of Tomcat

The server itself is built in a pipelined fashion making it possible to have multiple requests flowing through the system even as multiple threads are used to concurrently process requests. The concrete architecture of Tomcat is shown in Figure 1. When a HTTP request arrives at the Tomcat, these are the set of steps that occur in sequence:

1. It is handled by the Coyote protocol adapter.
2. The adapter assigns a thread from the thread pool.
3. The request is then associated with HttpRequest and Response objects also obtained from a pool. The request HTML is parsed and the individual fields in the Request and Response are filled in. Parsing may be just-in-time as well.
4. The request is passed through a user defined pipeline of filters where each step of the pipeline does some (user defined) processing on the `Request` and `Response` objects.
5. It then gets mapped to a virtual host which then processes the request through its own pipeline of filters.
6. The request is then associated with the context of the web application with which it is bound.
7. The appropriate Java classes are loaded using the right class loader. This step may be skipped if the classes have already been loaded and have not been invalidated by a new deployment.
8. Finally, the request is associated with an instance of the servelet and the `service()` method of the servlet is called, which in turn generally maps the type of HTTP request to appropriate method of the servlet.
9. After the servlet finishes processing, the response object flows through the same path, freeing up resources which it had earlier acquired and returning objects to respective pools.
10. Finally, the Response is converted back to HTTP response over the socket stream.

The above description shows that simply modeling the execution of a servlet as a single queue is not appropriate and we need to ensure that the different aspects of request processing need to be taken into account in the analytical model for it to be accurate.

4.2 The QPN Model of Tomcat

In this section, we present the QPN model of Tomcat which works as described in the previous section.

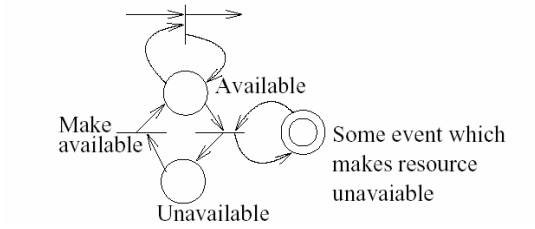


Figure 2: Modeling Resource Unavailability

The model takes explicit care to create QPN constructs which can be reused often. For example, the QPN construct for the blocking on access to hash table can be seen repeatedly in the Tomcat QPN.

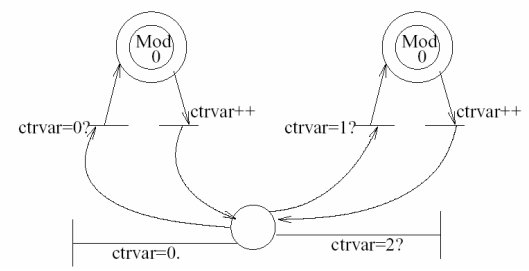


Figure 3: Modeling a closed loop chain of responsibility

Besides, blocking on service unavailability, pooling, pipelining or chain of modules are patterns which appear repeatedly in the Tomcat architecture and logical processing flow of the requests. We have created a library of such high level modeling patterns that we use to create larger models for efficiency purposes.

timed transitions are populated with the distribution of the execution times. These parameters have been culled out of observation largely and will decide on the accuracy of our model. We are now going through a cycle of verifying the accuracy of these parameters.

Parameters such as sizes of the pools for threads and other objects are fed directly from the configuration information from the Tomcat configuration file. This model is rather simplistic - it does not for example take into account that a percentage of requests may need to use the DB or other resources outside Tomcat. However the extension is fairly simple, we need to introduce the notion of colors within our Petri-Net and come up with the probability distribution of requests of different types. We are currently working on verifying the model before moving on to extending it.

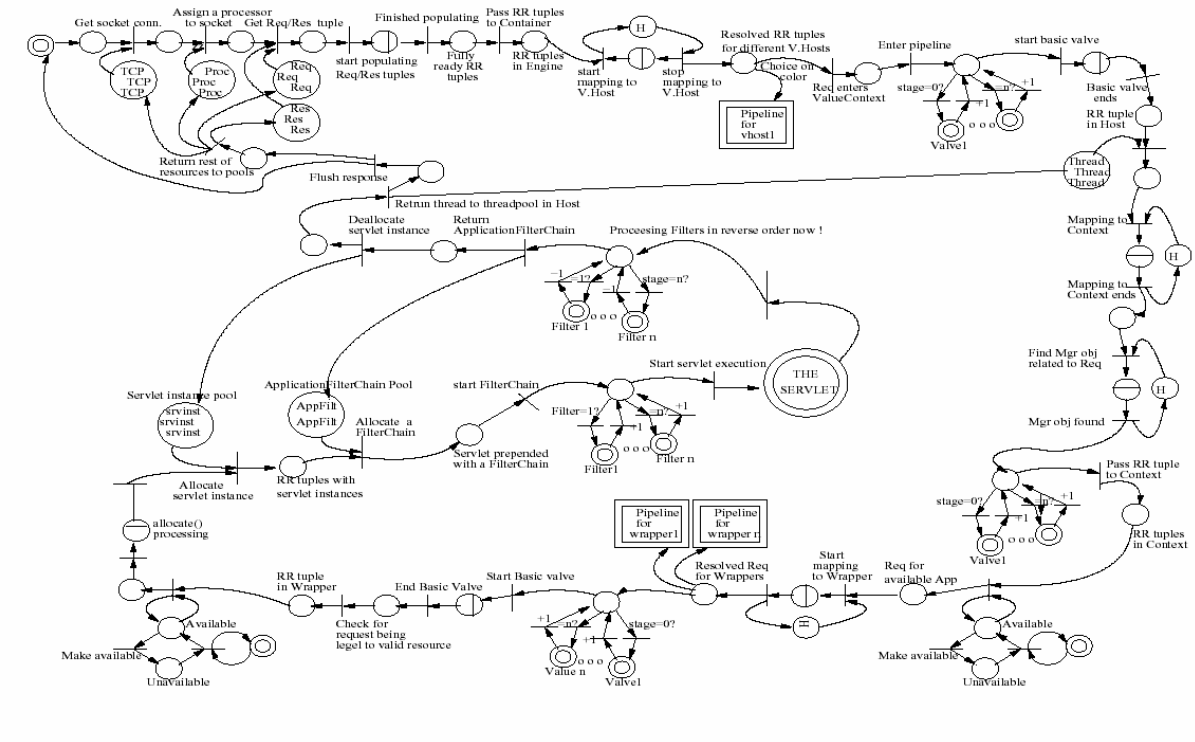


Figure 4: The QPN Model of Servlet Execution in Tomcat

Figure 4 shows the QPN Model of executing a servlet in Tomcat. It is essentially a composite of a host of QPN Patterns the likes of which we described earlier. Each queued place is populated with the arrival distributions and

5. Related Work

Our project spans multiple domains, that of performance prediction, that of autonomic computing and that of resource management. We will briefly describe related work in the former two areas as they are most closely related.

Our approach to autonomic computing and self-

configuration has an analytical basis in modeling application execution as Markov processes, solving the models to get an idea of the expected QoS and then optimizing the model for the needed QoS. Other approaches include biological and adaptive system models which take inspiration from the human autonomous nervous system for modeling system architectures [5], real-time monitoring and dynamic adjustment based on localized optimizations [4] and models inspired from physical models of gravity for placement of objects in an application on distributed systems [1].

Regarding performance prediction, there have been a multitude of efforts mainly using Queuing network theory for predicting performance of web applications but most of these are for CGI based programs where the model will be significantly different than that of Servlets and J2EE. In particular, Falko Bause et. al, [11] have provided a interesting and usable tool to model Petri nets hierarchically which will be extremely useful in describing large models. Kounev and Buchmann in [15] have used this tool to do coarse grained modeling of enterprise applications - the significant difference between their work and ours is the level of detail that we feel is necessary to achieve realistic results. Our approach is extremely fine grained.

6. Current Status and Future Work

As outlined we are currently working on the simplest of the architectural models – one in which dynamic web content is provided through servlets which don't access any external resources. We have completed the model and populated it with the various parameters using observation. We are in the process of verifying the accuracy of our model by comparing the predicted results with actual performance as gathered from an instrumented version of Tomcat which we have. Obviously the simplicity of the model is unrealistic and we intend to take this model through two steps of refinement:

a. Extend the model to include a data base which is also modeled at the same level of detail. Web requests here go to the DB in some percentage of the cases and so will be significantly be more expensive. This will also cause a refinement of the Tomcat model itself since it will introduce resources such as connection pools which don't exist in our current QPN Model.
b. Model full blown enterprise applications using middle tier business logic technologies in addition to web front ends and DB back ends. This might also include workflow and rule engines as well.

In parallel, we are proceeding along the track of using the performance prediction for optimized usage of resources.

7. References

[1] Mario Tokoro, Computational field model: Toward a new computing model/methodology for open distributed environment. In proceeding of ObjectOriented Programming Systems, Languages and Applications, October 1990

[2] Dashofy E. M.m et al., "Towards Architecture-based Self-Healing Systems", ACM WOSS, Charleston, SC, USA., 21-26, Nov., 2002.

[3] Fox A. and Patterson, D., "When Does Fast Recovery Trump High Reliability?", *Proceedings of the EASY 2002*, San Jose, CA, October 2002.

[4] Garlan, D. and Schmerl, B., "Model-based Adaptation for Self-Healing Systems", ACM WOSS, Charleston, SC, USA., 27-32, Nov., 2002.

[5] George S., et al., "A Biologically Inspired Programming Model for Self-Healing Systems", ACM WOSS, Charleston, SC, USA., 102-104, Nov., 2002.

[6] Vaidyanathan, K., Selvamuthu, D., and Trivedi, K. S., Analysis of Inspection-Based Preventive Maintenance in Operational Software Systems, Intl. Symposium on Reliable Distributed Systems, SRDS 2002, Osaka, Japan, October 2002

[7] Probability and Statistics with Reliability, Queueing and Computer Science Applications, Kishore S. Trivedi, ISBN 0-471-33341-7, John Wiley and Sons.

[8] Jakarta Project at Apache Software Foundation, http://jakarta.apache.org/tomcat/

[9] The J2EE Specification version 1.4, http://java.sun.com/products/j2ee/1.4/

[10] The Servlet Specification version 2.4, http://java.sun.com/products/servlet/

[11] F. Bause, P. Buchholz and P. Kemper – QPN Tool for the Specification and Analysis of Hierarchically Combined Queueing Petri Nets. Quantitative Evaluation of Computing and Communication Systems, Lecture Notes in Computer Science No. 977, Springer-Verlag, 1995

[12] M.Vernon, J. Zahorjan, E.D Lazowska. A comparison of performance Petri nets and queueing network models. Proceedings on the International Workshop on Modeling Techniques and Performance Evaluation, Paris(France), 1987. pp. 191- 202.

[13] G. Rozenberg (ed.). Advances in Petri nets. Lecture Notes in Computer Science 424, 1989, pp. 1-29.

[14] D.Mailles and S. Fidda. Queueing systems with flag mechanisms, Proceedings on the International Workshop on Modelling techniques and Performance Evaluation, Paris(France), 1987. pp. 167-190.

[15] Alejandro Buchmann and Samuel Kounev, Performance Modeling of Distributed E-Business Applications using Queuing Petri Nets, Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2003.