Template Driven Performance Modeling of Enterprise Java Beans

Jing Xu, Murray Woodside

Dept. of Systems and Computer Engineering,
Carleton University, Ottawa K1S 5B6, Canada

xujing@sce.carleton.ca, cmw@sce.carleton.ca

Abstract

System designers find it difficult to obtain insight into the potential performance, and performance problems, of enterprise applications based on component technologies like Enterprise Java Beans (EJBs) or .NET. One problem is the presence of layered resources, which have complicated effects on bottlenecks. Layered queueing network (LQN) performance models are able to capture these effects, and have a modular structure close to that of the system. This work describes templates for EJB components that can be instantiated from the platform-independent description of an application, and composed in a component-based LQN. It describes the process of instantiation, and the interpretation of the model predictions.

1. Introduction and motivation

Application servers using component technologies such as Enterprise Java Beans and the J2EE standards [1] [6] [8] promise rapid development and good performance and scalability. Many services are provided by platforms for J2EE and other approaches like .NET (such as support for concurrency, security, and transaction control), leading to substantial overhead costs. Performance shortfalls are a significant concern.

Predictive models of a software design can provide insight into potential problems, and guidance for solutions, as described by Smith and Williams (e.g. [14]) and others (see for example [1] [20]). However modeling is unfamiliar to designers, and takes significant effort. This work sets out to reduce the effort by providing templates which can be tailored to the business logic of the application. They are instantiated and composed into a model of the infrastructure parts such as a J2EE platform, the web server and the database, which are modeled in advance, with parameters to describe the possible deployments. This provides a rapid model-building capability, compatible with the rapid development process.

The process of defining component-based performance models, and of building models from components, has described in [5][19]. The models are layered queueing networks (LQNs) as described in [15][16][21], and the introductory tutorial [17]. Layered queueing is a strategic choice. Compared to other formalisms surveyed in [1], it extends queueing networks to include software resources, and it avoids the state explosion of Markov models based on Petri Nets. Each software component is a distinct model

entity, and contention for logical resources such as threads (which define the concurrency in the server platform) is captured.

This work defines a template-based framework for models of any J2EE application server, and describes in detail how the templates can be applied. The main focus of this paper is the rationale of the LQN templates, i.e. how the templates are derived from the platform-independent description of the application behavior, and how to instantiate the templates to represent a concrete EJB component. The paper also shows the interpretation of results to guide the choice of pool sizes. A companion paper [23] has considered the calibration of a model against real profiling data, and its capability to represent the performance of a small application.

2. Model Framework

Figure 1 shows a layered queueing model for a small web application that provides two business services to the Web Server and further to the Client. Each layer has a large rectangle represents a concurrent entity that may have multiplicity, resources, and behaviour. The right-hand block of each entity (called a "task" in LQNs) represents the entity as a whole; the blocks to its left represent its methods or services exposed to its users (called "entries" in LQNs).
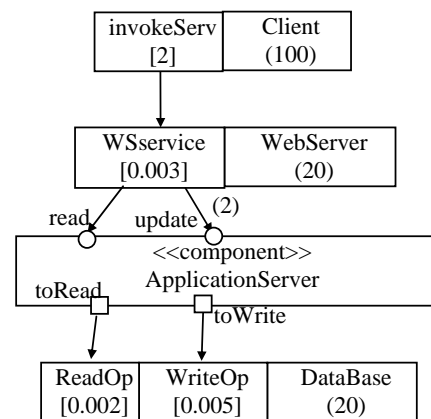
Figure 1 Layered Queueing Model for a web application

The arrows represent calls originating in one entry, to call another. All these calls are synchronous (call-wait-reply)

interactions; asynchronous calls (with no wait/reply) can also be indicated (graphically, by an open arrowhead on the arc).

The parameter within each entry gives its "host demand" (CPU time demand per call); the parameter within each task gives its multiplicity. Thus there are 100 Clients (representing users at their desktops) with client delays of 2 sec., the Web server has 20 threads and WSservice demands 3 ms total to handle each call by a Client (including invoking the application service), and the database has 20 threads, 2ms for a read operation and 5 ms for a write operation. The parameter on each arrow shows the mean number of calls made during one invocation of the calling entry. It is 1 by default if not explicitly shown.

Component-based modeling for LQNs was described in [19] for assembling sub-models for application elements together with infrastructure sub-models such as a web server, a database, or an application server [13]. The definition of a component sub-model, and its binding into a system model, are illustrated in Figure 1 and Figure 2. In Figure 2, the large rectangle represents the boundary of the component, with its interfaces. The ports represented by circles on the upper edge show *provided* interfaces (with a separate port for each entry within the component), and the ports represented by squares on lower edge show *required* interfaces. In component-based modeling the outer system model is defined with a "slot" having the same interface (shown as the ApplicationServer in Figure 1). The component sub-model is defined separately (as in Figure 2), and then bound to the interfaces and processors of the slot in the system model.
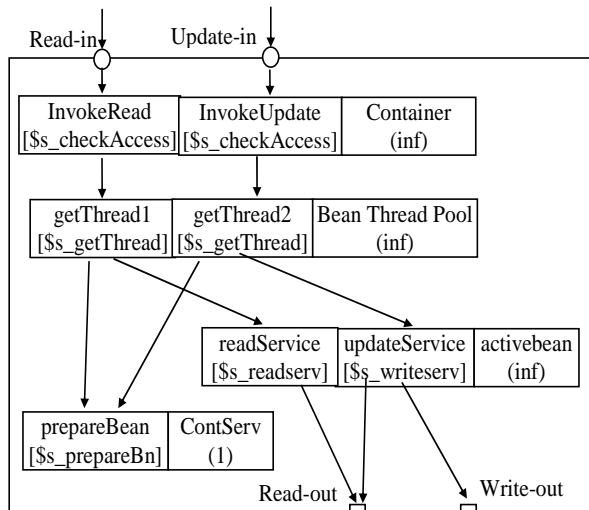
Figure 2 Component Sub-model for an Application Server

The component submodel in Figure 2 represents a Session Bean (based on a template that is described below) taking the place of the application server in Figure 1. We notice that there is a number of internal "tasks", some of which represent the container functions (Container and Bean_Thread_Pool) and some, the application. Infinite multiplicity is attached to fully reentrant objects, multiplicity

1 to a critical section, and other multiplicities, to thread pools. Host demands are described by variables with names beginning with '$' signs.

3. Template Driven LQN Modeling

Model templates provide a general solution for modeling EJB applications in their environment. The template captures common standard structure and parameters and allows variable features to be instantiated both for specific platform and for specific application.

A template has partially fixed internal structure with placeholders and parameters that provide capability of alternative. Instantiation of a template results a LQN component sub-model.

A placeholder is like a piece of schema or meta-model for a LQN model fragment (e.g. entries in a task). When a template is instantiated, the placeholder is replaced by zero or more concrete elements according to application behavior. Relationship between generated concrete elements remains the same as the relationship between their placeholders.

Execution demands and entry invocations (frequency of calls in LQN) can be defined as parameters in a template. When the template is instantiated, the parameters are either replaced by concrete values or kept as variables to be determined later.

Template driven modeling is suitable for analysis of EJB system because all application servers behave alike. Fixed part of a template represents features that are common to all application servers that conform to the J2EE standards. For a particular product, the parameters associated with structural fixed part (mostly container services) can be instantiated by using platform specific data. These data usually can be obtained through profiling or benchmark. Instantiation of the placeholders and their parameters makes the resulted concrete component representing specific application business logic. The data for these parameters can be either obtained by profiling or benchmark, or be assumed or required values in order to get performance prediction.

In the following sections, we will show templates for different types of Enterprise Java Beans (EJBs) and examples on how to use these templates.

4. LQN templates for different EJBs

The three main types of EJB are the Session Bean (used to implement business logic), the Entity Bean (used to represent business entity objects that exist in persistent storage), and the Message Driven Bean (used to respond to an asynchronous invocation). Session Beans are called "stateful" if they maintain the status of a client conversation, or "stateless" if they do not. This section will describe the LQN templates for each type of the EJBs, for cases with Container-Managed Persistence.

4.1 LQN Template for a Stateless Session Bean

A *Session Bean* represents a single client inside the Application Server, and is not sharable. It performs work for its client and is similar to an interactive session, for instance it manages transaction properties. A Session Bean is not

persistent. When the client terminates the session, the session bean is no longer associated with the client.

Figure 3 shows the internal behavior of a Stateless Session Bean. Incoming requests for a business method are captured by the EJB container. A Container thread will be generated for each incoming call. It first checks if the client has access rights to perform this operation on the Session Bean, indicated as a method of the Container. Here we model cases in which the client is authorized. Then the Container thread requests a bean thread from the bean thread manager (BnThreadMng). After obtaining a bean thread, the Container instance enters a critical section described by the behavior fragment in the box labeled "critical", to prepare the thread to execute the method. If the session bean involves transaction operations, it may call external services for initiating or terminating transactions. On exit from the critical section, the Container will invoke the business method on the active bean thread obtained. During execution of the method, external services may be called.

Figure 3 is annotated with performance information according to the UML Profile for Schedulability, Performance and Time [11]. This includes the stereotyping of computation steps (<<PAstep>>) with CPU demands (tagged value PAdemand) and in the case of calling for external transaction services giving the step probability which is shown as tagged value PAprob=$ptranx, and same for PAprob=$pextserv for invoking external services. The stereotyping of the critical section as <<GRMResource>>, with steps to acquire and release it, is an extension of the Profile for logical resources suggested in [12].
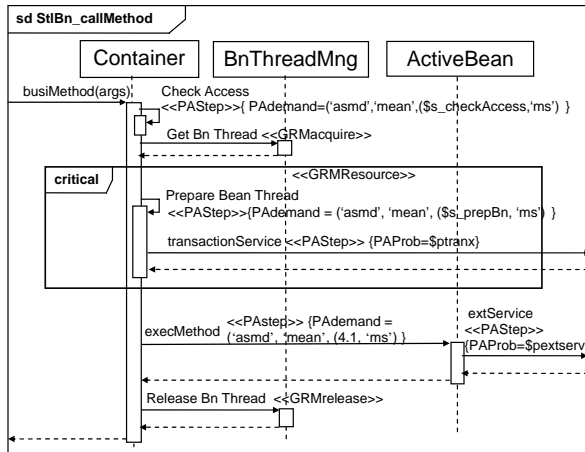

Figure 3 Internal behavior of a Stateless Session Bean

Figure 4 shows the LQN template for a stateless Session Bean derived from the behaviour. The container services are separated into 2 tasks: a *Container* task with infinite multiplicity represents the unconstrained operations on the incoming calls, including the check access operation, and a single threaded task *ContServ* models the critical section for preparing the bean thread. The contention for active bean instances is represented as requests to the *BeanThreadPool* task, with multiplicity parameter $M for the pool size.

The elements with bold lines are placeholders, which, in this case, include all provided and required ports, entries *invokeMethod*, *getThread*, *busiMethod* and all the calls that with at least one end connected to these entries. Parameters are annotated by a '$' sign followed by a name, such as $s_prepareBn for the CPU demand of the entry *prepareBean* and $ptranx for mean number of calls made to external transaction services from *prepareBean*.

The general structure of this template represents the platform independent behaviour of a session bean, while the parameters $s_checkAccess, $s_getThread, $s_prepareBn can be filled with values according a specific middleware solution. $M is a tunable parameter of the runtime configuration. The business logic of an application determines the instantiation of the placeholders and their associated parameters, including the instantiation of required or provided interfaces (the placeholder ServiceRequest or methodInvoke) and calls to or from them. Options in the business logic will also determine the use of the required transactionService interface.
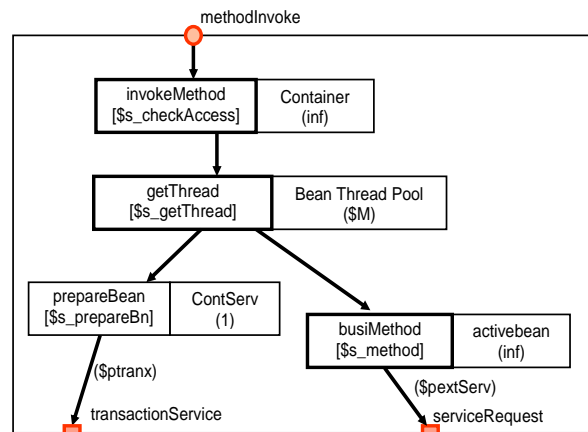

Figure 4 LQN template for Stateless Session Bean

To instantiate the template, each placeholder is replaced by one or more instance entities. The chain of entries invokeMethod, getThread, busiMethod is instantiated for each separate business method, along with its input port and the arcs joining the entries. The result is an LQN component sub-model. Figure 2 shows an instantiation of the template in Figure 4, with two ports connected to two business methods.

The template *methodInvoke* port is instantiated twice into ports *Read-in* and *Update-in*, along with the entry chain, *invokeMethod*, *getThread* and *busiMethod.* The required port *serviceRequest* is instantiated twice. The call from *busiMethod* is instantiated once for *readService*, and twice for *updateService* with calls to both required ports (the call number $pextserv =1 for both). Since no external transaction service is required, the outgoing call from entry *prepareBn* and its port are omitted in Figure 2 (i.e $ptranx=0). The CPU demands $s_checkAccess and $s_getThread are the same on both paths since they representing platform operations, whereas $s_method is instantiated separately in the instance entries since each business method has its own demands.

4.2 LQN Template for a Stateful Session Bean

A Stateful Session Bean is different in that it maintains the status of its client conversation. In order to achieve this while maintaining efficiency on sharing a limited thread pool, the status of a session bean may be swapped out from memory and stored in a file system when it is not in use and the container claims its thread resource. This procedure is called passivation of a bean instance. When its client requires its service again, an empty thread will be acquired from the container and its status information will swapped into memory again, called activation of the instance. This may incur swapping out another bean instance.
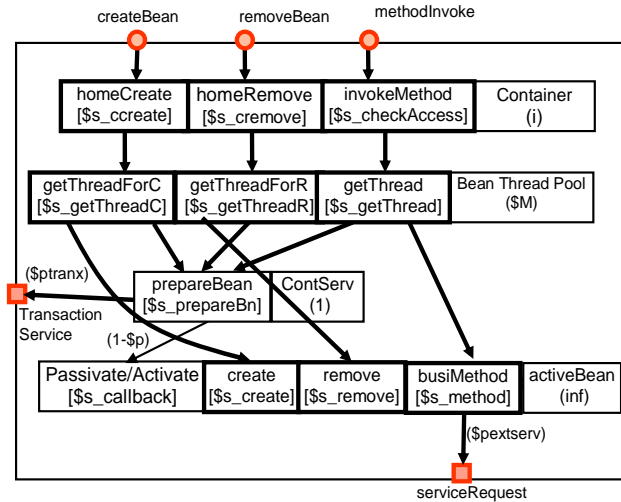

Figure 5 Template for a Stateful Session Bean

Figure 5 shows the LQN template for a Stateful Session Bean. The passivation and activation operations are aggregated and shown as callback functions from the critical section of the container service *ContServ* to the active bean. These calls inform the bean that the container is about to passivate or activate the bean instance, so that the bean instance can release or acquire corresponding resources such as sockets, database connection, etc., and they include the passivation/activation overhead as well. The "hit rate" p is the probability that a required bean instance is currently active (in memory), so $(1-\$p)$ is the probability that passivation/activation is invoked on a new request.

A Stateful Session Bean also provides *home* interfaces that allow clients to control creation and removal of a bean instance. Elements representing these interfaces and related container services are shown in the template.

4.3 LQN Template for a Message Driven Bean

A *Message Driven Bean* is similar to stateless session bean except that it processes messages asynchronously. It normally acts as a Java Message Service (JMS) listener which can process either JMS messages or other kinds of messages. The messages can be sent to any J2EE component by a JMS application, including systems that do not use J2EE technologies. A *Message Driven Bean* is useful for implementing asynchronous business logic.

The LQN template for a message driven bean is the same as the template of a stateless session bean, except its incoming calls are asynchronous messages to the *invokeMethod* entry.
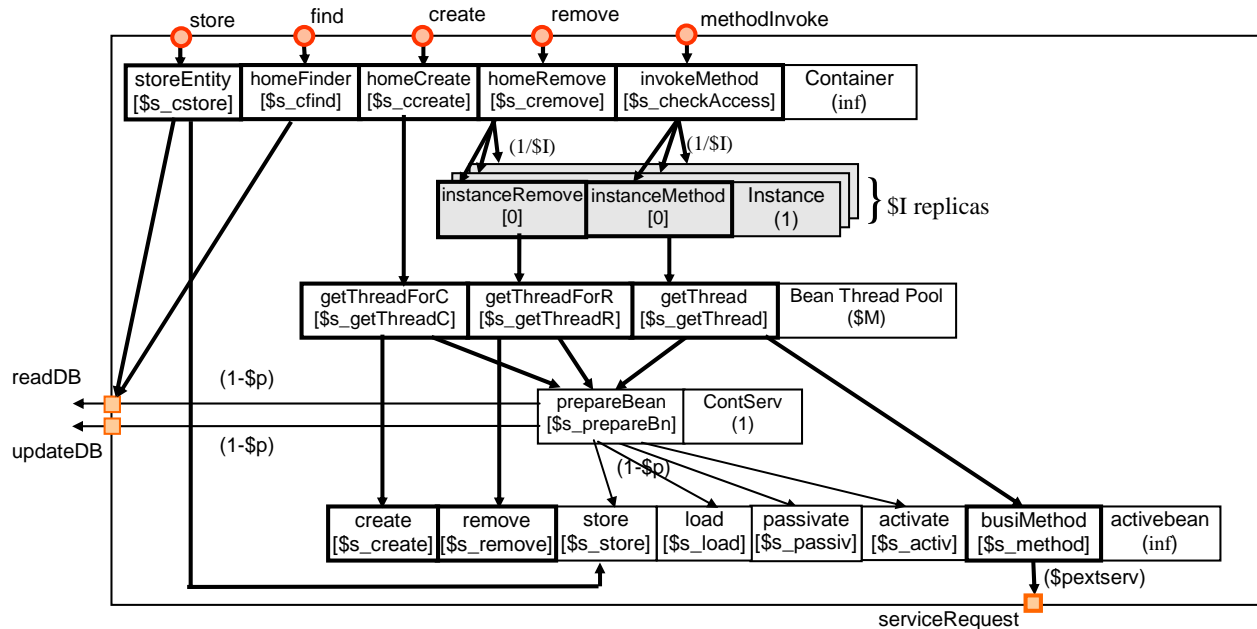

Figure 6 Template for an Entity Bean

4.4 LQN Template for an Entity Bean

The Entity Bean has the most complex functional and resource behaviour, often resulting in performance issues. Besides competing for thread pool and critical container services, requests may contend for data objects. When an instance of an Entity Bean is in use by a client, other clients requiring the same instance (i.e. the same data) must wait. In the LQN template this contention is represented by requests to a replication pool of pseudo-tasks called *Instance*, with one task for each Entity Bean instance. A request to a busy *Instance* must wait for it to become free. The probability of accessing each replica in the pool is assumed equal here, i.e. probabilities of calls into entries of each replica are the same (1/$I in the diagram). In the case of some data instances may be accessed more frequently than others, separate tasks with different accessing rate need to be added.

Besides the home interfaces for creating and removing an instance, a *find* interface is also provided for looking up data in database and returning the handle of a bean instance which represents the data. The *store* interface is used when a request to update the Entity state into the database is issued by another EJB component in the same application server, for instance during a transaction-committing step of a Session bean.

The *readDB* and *updateDB* interfaces represent database operations during service and bean-instance context swapping (passivate/activate).

5. Using the LQN templates

An EJB system is modeled by first modeling the beans as tasks with estimated parameters, then instantiating the template to wrap each class of bean in a model of its container, and finally modeling the execution environment including processors (CPUs) and database. Calls between beans, and calls to the database, are part of the final assembly. The model may be calibrated directly from operational data such as profiling, or by combining designer knowledge of the operations of each bean with pre-calibrated workload parameters for container and database operations.

The model can then be solved by LQN solvers either analytically or by simulation, to evaluate throughputs, response times, and resource utilizations. The results can be used to guide choices of EJB patterns and deployment configurations.

Two examples will be shown in this section. The first example describes the LQN model for a three-tier client-server system with only Entity Beans. The model was solved and the results were compared with a previous study by simulation. The second example describes how to build a model for a more complex system with different type of EJBs, but (to save space) it only shows parts of the model.

5.1 An Entity Bean example for the use of the template

To demonstrate that the LQN model can be applied to this class of system with reasonable accuracy, we revisit a simulation study done by Llado and Harrison for a system with entity beans [9] [10].

Figure 7 A three-tier client-server system in [9] [10]

Figure 7 shows the architecture of their three-tier client-server system. The client requests database operations through Entity Beans which reside in the application server. There is only one class of Entity Bean involved with a single type of business method. No home operations are required on the Entity Beans.

Figure 8 shows the LQN model for this system. The client and database are modeled by tasks. The Entity Bean template was instantiated into an "EJB Component" sub-model and then was assembled in the slot of the application server. Finally the component is bound to the ServerCPU which is shared with the Database. In order to focus on performance of software components and eliminate the affect of hardware, the ServerCPU was set at infinite multiplicity (ample multiple CPUs).
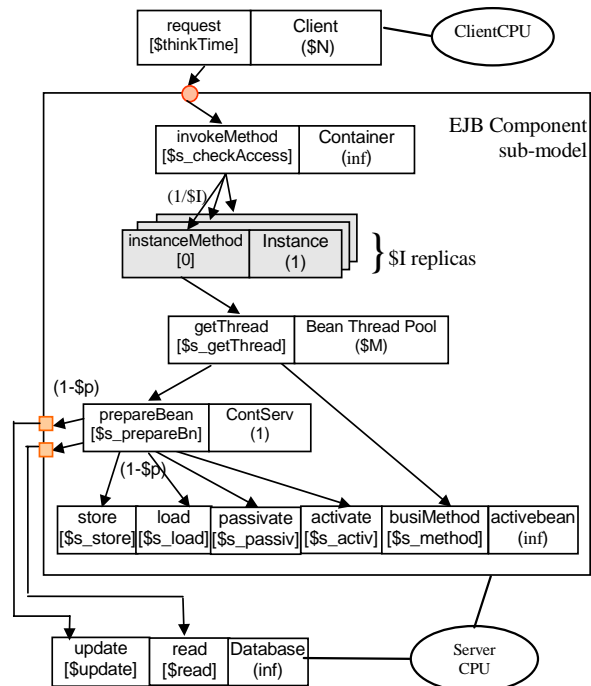

Figure 8 LQN model for the system in Figure 7

Using the same parameter values as in [10], the LQN model was solved with 40 instances ($I=40), a pool size of 6 ($M=6), negligible execution demand for *invokeMethod*, *getThread*, and *prepareBean* ($s_checkAccess = 0.001ms, $s_getThread = 0, $s_prepareBn = 0.00ms) and business method (*busiMethod*) time of 4.1ms ($s_method = 4.1ms). The underlying Database services and call back functions were aggregated to a total demand of 0.4ms (i.e. $update + $read + $s_store + $s_load + $passiv + $activ = 0.4ms).

Figure 9 compares the simulation results from [10] with the LQN model. The difference between these two results is about 6%, with the LQN being a little pessimistic.

From the results we can learn that the system is saturated with about 10 clients giving a throughput of 1.3/ms. The bottleneck is at the bean thread pool, which has a utilization of 98.8%. These results imply that the configuration of the bean thread pool size should be increased in order to achieve higher performance if more than 10 concurrent clients are expected.
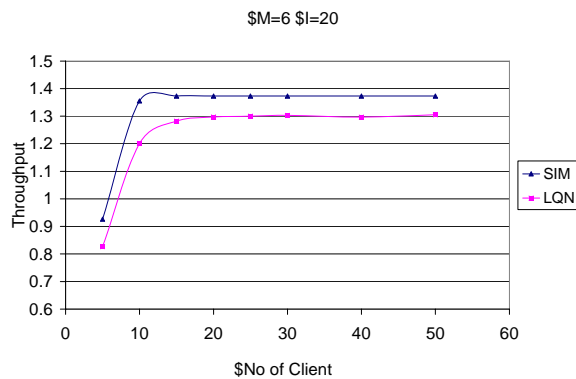
Figure 9 LQN model predictions compared with Simulation Results [10]

In [9] Llado and Harrison describe another analytic model for this system using decomposition, with a custom-built solution strategy, which provides an even closer match to the simulation results. However the effort of creating such a model must be repeated for every configuration, and would be even more complex with multiple interacting beans. The advantage we seek with the LQN is the use of a standardized model framework and solution strategy, and a systematic model-building process based on templates for different kinds of beans.
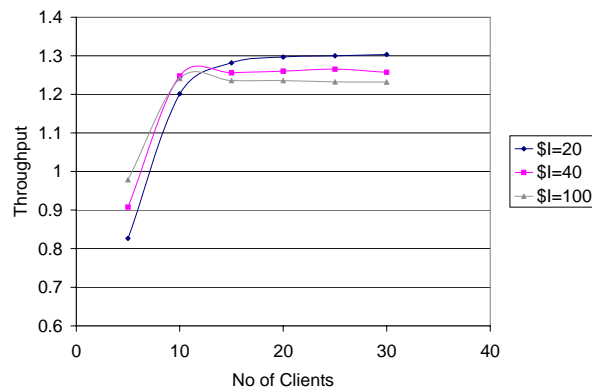
Figure 10 Results for Different Numbers of Instances

Another set of results in Figure 10 compares the throughput for different numbers of bean instances $I, with the same pool size $M = 6. We can see that the number of bean instances makes little difference because the system is limited by the small thread pool. This also corresponds to

Llado's results. Before reaching saturation, the system with a larger number of instances gives higher throughputs because of less competition for each data instance (based on an equal probability 1/$I of accessing each instance, which is small for large $I). On the other hand, after the bean thread pool is saturated, the throughput for the case with small number of instances is higher, because the hitting rate on an active bean instance is lower ($p is small), which results in more overhead on swapping bean instance.

5.2 Example on constructing a LQN component model containing different types of EJBs

In this section, we model a more complex EJB system called RADS Book Store. Due to space limitations, we only show the internal structure of the application server and some but not all of the EJB components.

The RADS Book Store is a web-based system providing basic online store services including user inquiry, purchase, and inventory management. The system was implemented on Weblogic 8.0 platform in Windows environment.

Figure 11 shows the sequence diagram for one of its scenarios, the Checkout scenario. It follows the EJB session façade pattern and involves three types of EJB: Stateless Session Bean (Controller), Stateful Session Bean (Shopping Cart) and Entity Bean (Order, OrderLine, and Book). We will model this scenario.
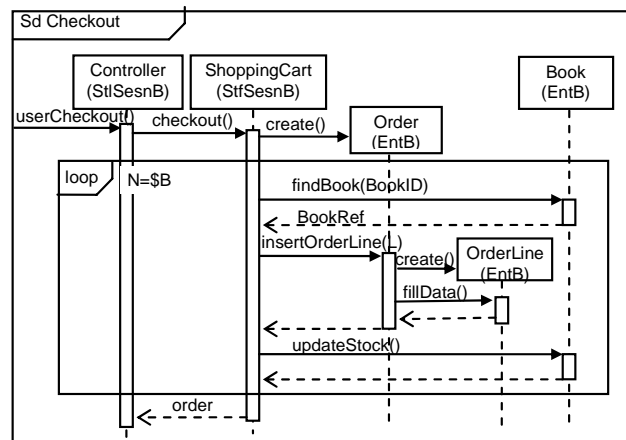
Figure 11 Checkout Scenario for RDS Book Store System

Figure 12 shows the LQN component model for the application server with slots to fit in EJB components. It has a provided interface (userCheckout) which will be connected to the client component, and 2 required interfaces (readDB and updateDB) that will be connected to database component in higher level LQN model.

Figures 13-15 shows the internal structure of the Controller Bean, Shopping Cart Bean and Book Bean instantiated from different EJB templates, as described in section 4.

In the case of the session façade pattern with container managed persistence, transactions are entirely managed by the container. A transaction is started at the beginning of an invocation on the session bean ShoppingCart, and is committed and ended right before the operation on

ShoppingCart is completed. Any change on entity data is updated into database during the transaction committing stage. Therefore, the store operation on entities is actually invoked by ShoppingCart during its critical section for bean context swapping (represented by prepareBean in the model).

Due to limited space, the component models for the Order and OrderLine entity beans are not shown here. Instantiation of the entity bean template for them is similar to that for the Book bean. The model would be completed by binding each component into its corresponding slot in Figure 12.
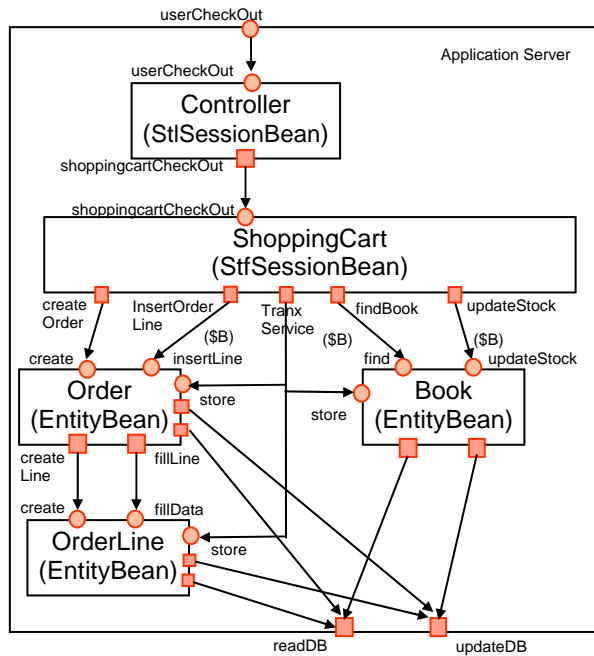

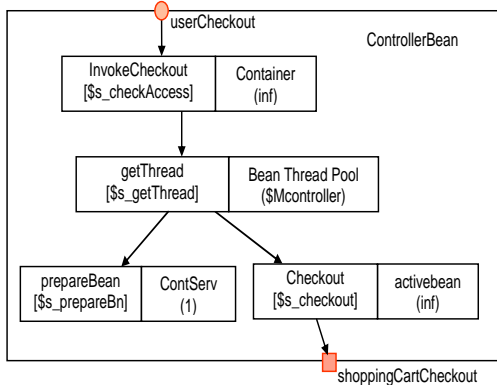**Figure 12 LQN model with Slots for EJB**

Figure 13 LQN component model for the Controller (Stateless Session Bean)

Figure 14 LQN component model for the Shopping Cart (Stateful Session Bean)
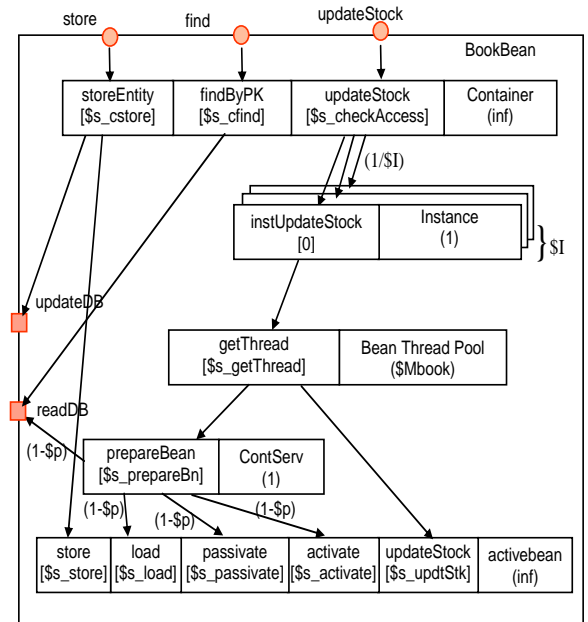
Figure 15 LQN component model for Book (Entity Bean)

6. Conclusions

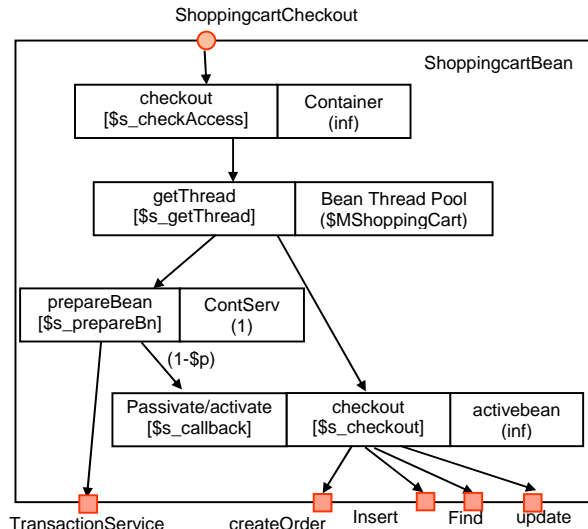This paper has described the process of defining predictive performance models for J2EE-based systems, using templates for EJB containers, and Layered Queueing with component-based features.

The modeler needs to define models only for platform-independent objects. These are then incorporated in template

instances which are assembled into a system model. Most of the model, representing the J2EE platform, can be pre-calibrated, and the application description (in terms of its use of services) can be dropped in. This is a kind of PIM-to-PSM (Platform-Independent Model to Platform-Specific Model) transformation, in model space. Automation of the transformation would be a useful next step.

The examples described in Section 5 demonstrate that the model gives useful accuracy, comparable to other approaches, and show how a complex system is handled.

The templates described here are for Enterprise Java Beans in a J2EE application server, but a similar approach could be applied to other technologies like .NET. The templates could be further extended to include the operating system by capturing common features of different operating systems.

The process of building models is supported by tools for component-based model-building [19][22]. However, the sub-model of the application logic represented by a bean is inserted into a template instantiated to contain it, with appropriate parameters for the instantiation. This is different from other examples of infrastructure which may run as a service layer to the application elements, for example in [18].

The present approach has been tested on a couple of example systems, including the well-known Duke's Bank Application which is shipped with J2EE documentation provided by Sun Microsystems [3]. A companion paper [23] describes experience calibrating a model and predicting saturation and delay. Saturation was correctly predicted and response time prediction errors ranged from about 2% to about 25%, with better accuracy for more clients.

References

[1] E. Armstrong, J. Ball, S. Bodoff, D. Carson, I. Evans, D. Green, K. Haase, E. Jendrock, *The J2EE 1.4 Tutorial*, on-line document at java.sun.com/j2ee/1.4/docs/tutorial/doc, Sun MicroSystems, Dec. 16, 2004.

[2] S. Balsamo, A. DiMarco, P. Inverardi, and M. Simeoni, "Model-based Performance Prediction in Software Development," *IEEE Trans. on Software Eng.*, vol. 30, no. 5 pp. 295-310, May 2004.

[3] S. Bodoff, D. Green, E. Jendrock, M. Pawlan, *The Dukes Bank Application*, on-line document at java.sun.com/j2ee/tutorial/1_3-fcs/doc/E-bank.html, Sun MicroSystems.

[4] G. Franks, A. Hubbard, S. Majumdar, J. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", *Performance Evaluation*, vol. 24, pp117-136, 1995

[5] V. Grassi, R. Mirandola, "Towards Automatic Compositional Analysis of Component Based Systems", *Proc Fourth Int.Workshop on Software and Performance*, Redwoood Shores, CA, Jan. 2004, 00 59-63.

[6] Java Community Process, "*J2EE 1.4 Specification*", on-line document at http://java.sun.com/j2ee/1.4/download.html#platformspec, Nov. 24, 2003

[7] Prasad Jogalekar, Murray Woodside, "Evaluating the Scalability of Distributed Systems", *IEEE Trans. on Parallel and Distributed Systems*, v 11 n 6 pp 589-603, June 2000.

[8] R. Johnson, *J2EE Design and Development*, Wiley Publishing Inc., Indianapolis.

[9] C.M. Llado, P.G. Harrison, "Performance Evaluation of an Enterprise Java Bean Server Implementation", *Proc second Int. Workshop on Software and Performance* (WOSP 2000), Ottawa, September 2000, pp 180-188.

[10] C.M. Llado, PhD thesis, Imperial College, London, 2001.

[11] Object Management Group, "UML Profile for Schedulability, Performance, and Time Specification," OMG Adopted Specification ptc/02-03-02, July 1, 2002.

[12] D. B. Petriu and M. Woodside, "A Metamodel for Generating Performance Models from UML Designs," *Proc. UML 2004*, v. 3273 of Lecture Notes in Computer Science (LNCS 3273), Lisbon, Oct 2004, pp. 41-53.

[13] Erik Putrycz, Murray Woodside, and Xiuping Wu, "Performance Techniques for COTS Systems", *IEEE Software*, v. 22, n 4, pp. 36–44, July-August 2005.

[14] C. U. Smith and L. G. Williams, *Performance Solutions*. Addison-Wesley, 2002.

[15] C.M. Woodside, E. Neron, E.D.S. Ho, and B. Mondoux, "An ``Active-Server'' Model for the Performance of Parallel Programs Written Using Rendezvous," *J. Systems and Software,* pp. 125-131, 1986

[16] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", *IEEE Transactions on Computers*, Vol. 44, No. 1, January 1995, pp. 20-34

[17] M. Woodside, "*Tutorial Introduction to Layered Modeling of Software Performance*", Edition 3.0, May 2002 (Accessible from http://www.sce.carleton.ca/rads/lqn/lqn-documentation/tutorialg.pdf)

[18] M. Woodside, D.B. Petriu, K. H. Siddiqui, "Performance-related Completions for Software Specifications", *Proc 24th Int. Conf. on Software Engineering* (ICSE 2002), Orlando. May 2002.

[19] X.P. Wu and M. Woodside, "Performance Modeling from Software Components," in *Proc. 4th Int. Workshop on Software and Performance* (WOSP 04), Redwood Shores, Calif., Jan 2004, pp. 290-301.

[20] J. Xu, M. Woodside, and D.C. Petriu, "Performance Analysis of a Software Design using the UML Profile for Schedulability, Performance and Time," in *Proc. 13th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (TOOLS 03), Urbana, USA, Sept. 2003

[21] J. A. Rolia and K. C. Sevcik, "The Method of Layers," *IEEE Trans. on Software Engineering*, vol. 21, no. 8 pp. 689-700, August 1995

[22] E. Putrycz, M. Woodside, X. Wu, "Performance Techniques for COTS Systems", *IEEE Software*, to appear, 2005.

[23] Jing Xu, Alexandre Outfimtsev, Murray Woodside, Liam Murphy, "Performance Modeling and Prediction of Enterprise Java Beans with Layered Queueing Network Templates", to appear in Proc. of SAVCBS'05, Lisbon, Portugal, Sept. 2005.