Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates

Jing Xu[1], Alexandre Oufimtsev[2], Murray Woodside[1], Liam Murphy[2]

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

Dept. of Computer Science, University College Dublin, Belfield, D4, Ireland

xujing@sce.carleton.ca, alexu@ucd.ie, cmw@sce.carleton.ca, Liam.Murphy@ucd.ie

Abstract

Component technologies, such as Enterprise Java Beans (EJB) and .NET, are used in enterprise servers with requirements for high performance and scalability. This work considers performance prediction from the design of an EJB system, based on the modular structure of an application server and the application components. It uses layered queueing models, which are naturally structured around the software components. This paper describes a framework for constructing such models, based on layered queue templates for EJBs, and for their inclusion in the server. The resulting model is calibrated and validated by comparison with an actual system.

Keywords

Layered Queueing Network, template, Enterprise Java Bean, performance modeling, model calibration, software profiling.

1. Introduction and motivation

The approach to designing application servers based on component technologies such as Enterprise Java Beans and the J2EE standards [1] [5] [6] provides rapid development and the promise of scalability and good performance. J2EE and other approaches such as .NET do this by providing many services which applications require (such as support for concurrency, security, and transaction control) within the platform. As a result however the server platforms also have substantial overhead costs, and performance is a significant concern. Predictive models of a design can provide insight into potential problems and guidance for solutions. The use of predictive modeling to analyze software designs has been described extensively by Smith and Williams (e.g. [10]) and others (see for example [2][17]).

To build predictive models efficiently, the description of the platform should be separated from the components that implement the business logic of the application, the web interface, and the database. The infrastructure parts such as a J2EE platform can be modeled in advance and reused, with embedded parameters to describe possible deployments. When a specific application is designed, its elements are modeled and plugged into the platform sub-model. This provides a rapid model-building capability, in parallel with the rapid development process.

The process of defining component-based performance models, and of building models from components, was described in [4][16].

This work is based on a layered queueing network (LQN) formalism, which is defined in [9][13][14], and the introductory tutorial [15]. Layered queueing is a strategic choice. Compared to other formalisms surveyed in [2], 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.

The central contribution of this work is to demonstrate a sound procedure for modeling EJB applications using a template-based framework, calibrating the model using profiling data and validating it against measurement. Key issues include the relationship between the EJB application components and the platform (which are captured in submodel templates), the feasibility of measuring the model parameters from execution traces, and the accuracy with which the model predicts performance of an implementation under load. The paper [18] describes the design and use of the templates in greater detail.

2. LQN Evaluation

To demonstrate that the LQN model can be applied to this class of system with reasonable accuracy, a simulation model of a system with entity beans due to Llado and Harrison [7] was compared to its LQN equivalent. Figure 1 shows this LQN equivalent model.

The LQN notation is taken from [13][14][15]. Software entities are modeled as *tasks*, offering services called *entries*. In the Figure a task is a rectangle with a rectangular field at the right-hand end for the task name and parameters, and a field for each entry. The entries have parameters in the form "[demand]" for the host (CPU) demand in ms, and have blocking requests to other entries indicated by arrows with labels for the number of requests. The parameters for the tasks are multiplicities, which limit the number of concurrent instances. There are $N clients, $M bean threads, 1 thread for each of $I identical bean instances, and no limit (shown as *inf*) for the Container.

The part of the model within the large rectangle represents one bean Container with its services and beans, and forms a component that can be generated for each bean. The component has input and output interfaces consistent with a component-based model-building framework called CBML [16], which allows complex interacting systems to be built up from modules like this.

The solid arrowheads on the request arcs indicate that the requesting task is blocked (and its task-related resource is held) while the serving task executes. The model captures resource holding patterns, such as (1) the *Container* operation "invokeMethod" requires a bean instance, and (2) startup operations by the *prepareBean* entry of the critical section

psuedo-task *ConServ* (Container Services, with multiplicity 1), represent mutual exclusion.

The bean instances (both active and passive) are represented by a set of $I replicas of the task *Instance* (the shaded boxes in the Figure). Each replica is a single threaded task, showing that requests to the same instance must wait. They are shown as being requested with equal probability (1/$I for each instance).

The probability that a requested entity bean instance must be activated is represented by the probability (1-$p) on calls from the *prepareBean* entry to the call back functions (*activate, passivate, load* and *store*) on the active bean.

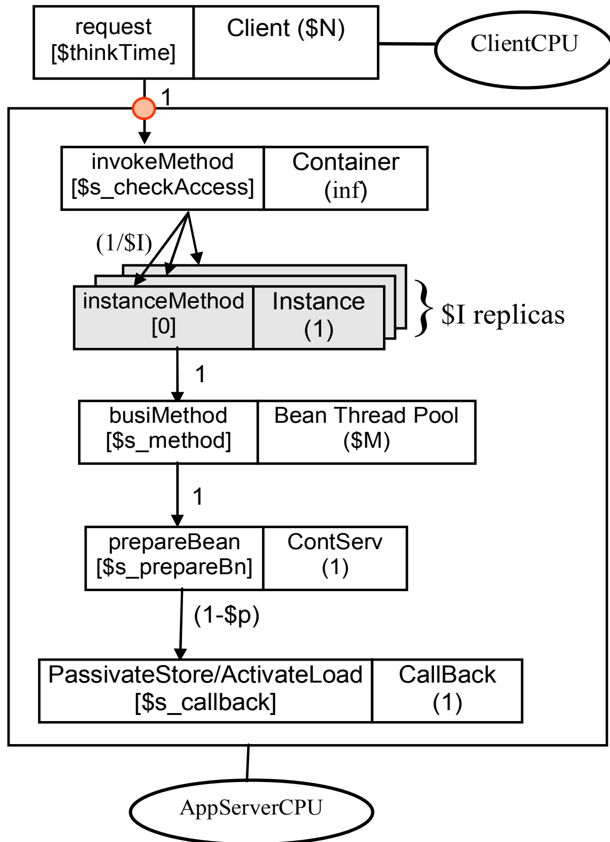

Figure 1 LQN model for the system with Entity Beans in [8]

With the same parameter values as were used in [8], the LQN model was solved with 20 instances ($I=20), pool size of 6 ($M=6), negligible execution demand for *invokeMethod*, and *prepareBean* ($s_checkAccess = 0.001, $s_prepareBn = 0.001) and business method (*busiMethod*) time of 4.1 ($s_method = 4.1). All the call back functions were aggregated to a single entry with a total demand of 0.4 (i.e. $s_callback = 0.4).

Figure 2 compares the simulation results from [8] with the LQN model. The results show approximately 6% differences between the two models with the LQN being a little pessimistic.

Llado and Harrison describe an extended queueing model for this system in [7], using decomposition and 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 becomes more complex with multiple interacting beans. The current approach tries to overcome this by the use of a standardized model framework and a systematic model-building process based on LQN templates for the different kinds of beans.

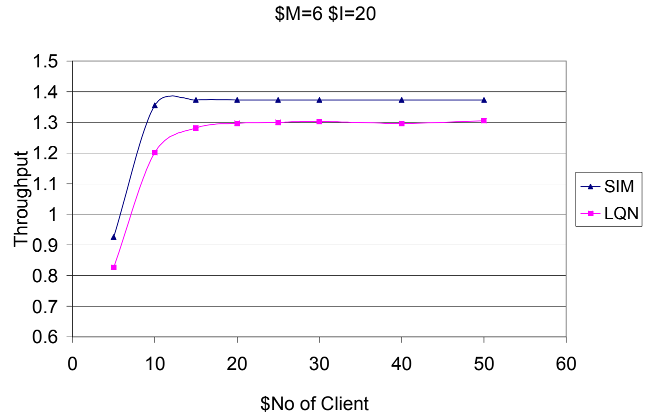

Figure 2 LQN model predictions compared with Simulation Results [8]

Figure 3 shows another set of results, which compares the throughput for different numbers of bean instances $I, with the same pool size $M = 6. It can be noted that the number of bean instances makes little difference since the system is saturated at the small sized thread pool. This behavior corresponds to the results obtained from [8].

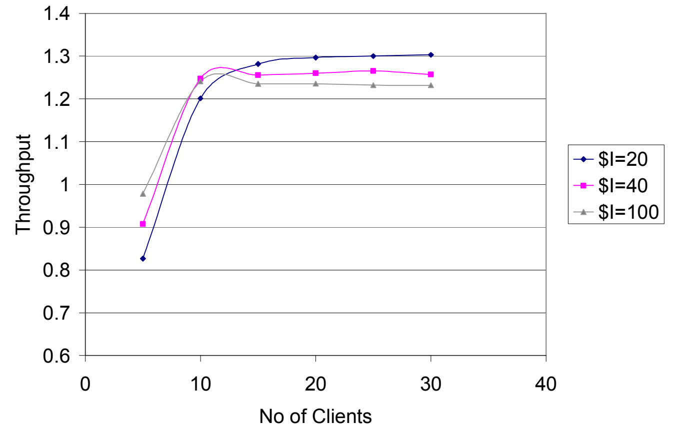

Figure 3 Results for Different Numbers of Instances

3. LQN sub-model templates

This section describes a LQN model template for each type of EJB. These templates follow the LQN component concept described in [16]. They can be instantiated according to specific function requirement in each scenario for system usage, and then be assembled into a complete LQN model for the whole scenario. An example on how to use these templates to build a complete LQN model for a business scenario will be shown in section 4.

3.1 LQN template for an Entity Bean

Figure 4 shows a LQN template for an Entity Bean. The entry *busiMethod* of the *activebean* is a placeholder for one or more methods of the Bean, and the entries *InvokeMethod*,

InstanceMethod and *getThread* are placeholders for resource requests on the calling path. All of these entries must be instantiated for each business method of the Bean, with calls between them as shown, connected to the interfaces.

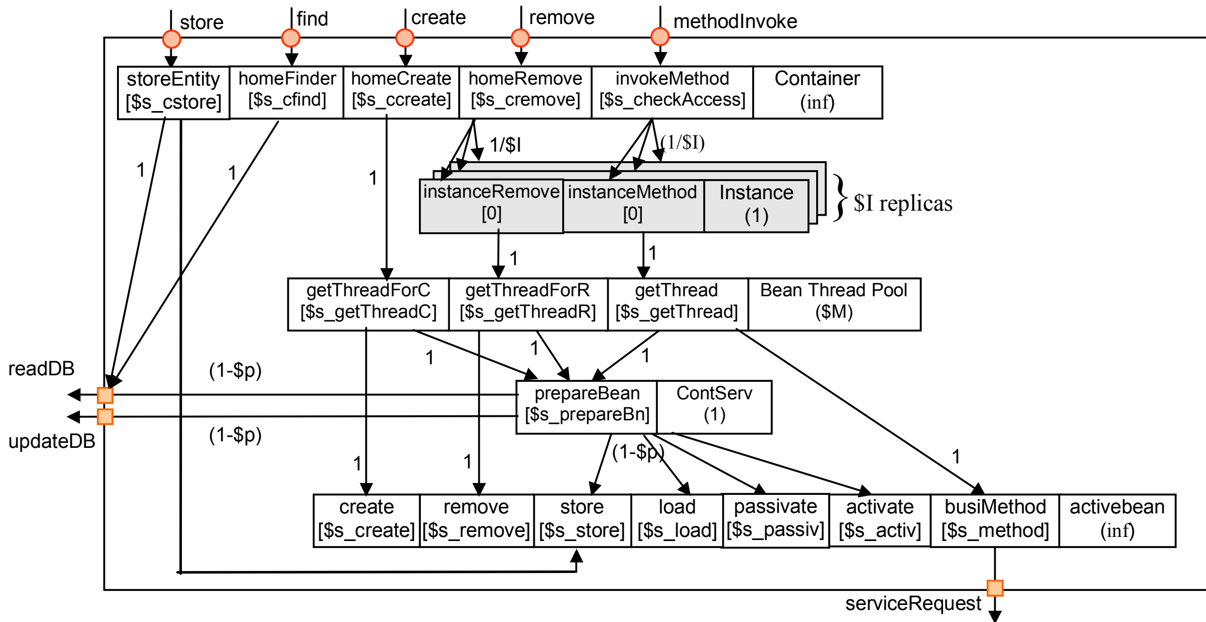

Figure 4 Template for an Entity Bean

The upper interfaces define provided services, with the placeholder *methodInvoke* for the Bean business methods. S*tore*, *find*, *create* and *remove* represent the Container Home Interfaces of the Bean. The *store* interface is used when a request to update the Entity state into the database is issued by another EJB component, for instance during a transaction-committing step of a Session Bean.

The lower interfaces define required services. S*erviceRequest* is a placeholder for function requests issued by the entity bean during its operation. The *readDB* and *updateDB* interfaces represent database operations during service and bean-instance context swapping.

When the template is instantiated, placeholders are instantiated as required for different services.

3.2 LQN template for a Stateless Session Bean

Figure 5 shows a LQN template for Stateless Session Bean. As it does not retain any state for a given client, each request can be directed to any available bean thread. After a service is finished, the bean thread is put back to the bean thread pool and ready for serving next request immediately. There can be outgoing requests.

For Stateless Session Beans the creation and removing of bean threads are controlled by the container. Clients do not create or remove bean threads.

3.3 LQN Template for a Stateful Session Bean

Figure 6 shows the LQN template for a Stateful Session Bean, which resembles that for an Entity Bean. However the passivation and activation operations use local storage rather than a database. The passivation and activation operations are aggregated and shown as callback functions from container to bean, represented by a pseudo-task *CallBack* whose workload is actually performed by the bean instance that is executing in the *ContServ* critical section. They 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.

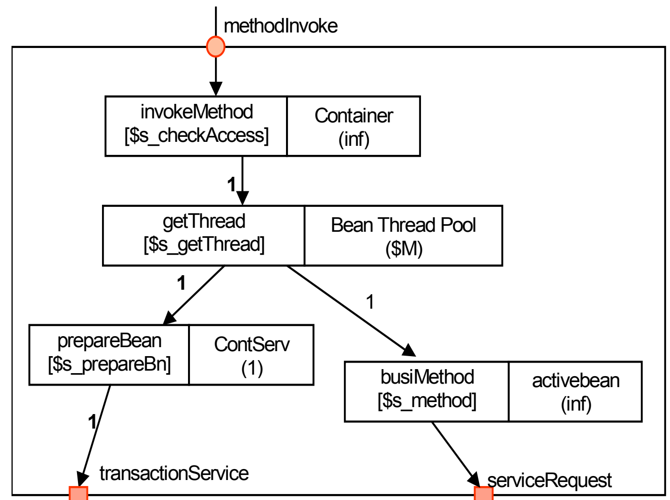

Figure 5 Template for a Stateless Session Bean

Assuming that each client has its own session bean there is no contention for a single bean instance, so the replica tasks for the instances are not modeled here. The thread pool is modeled as

executing the bean methods directly, with instances activated as necessary.

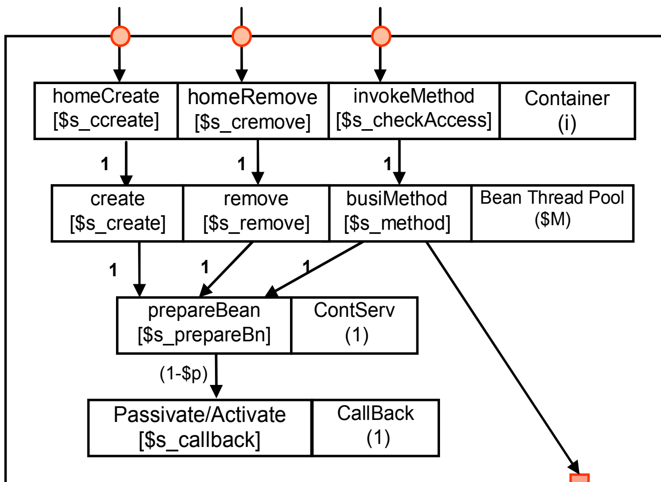

Figure 6 Template for a Stateful Session Bean

3.4 LQN Template for a Message Driven Bean

A message driven bean is similar to a stateless session bean, but its incoming calls are asynchronous messages (denoted by an arc with open arrowhead as shown in Figure 7).

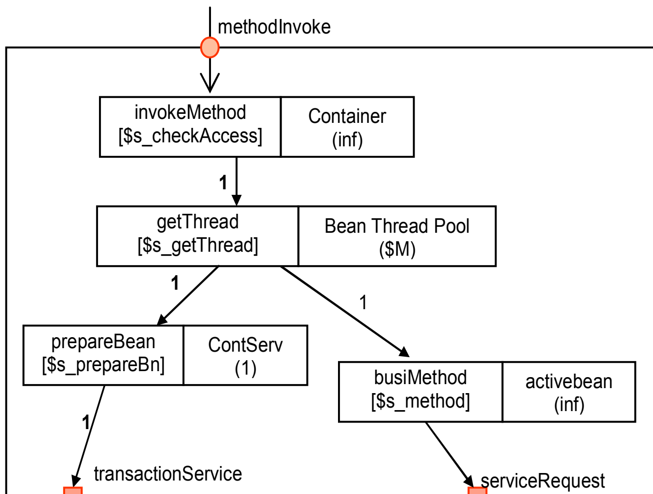

Figure 7 Template for a Message Driven Bean

4. Model Construction

A system is modeled by first modeling the beans as tasks with estimated parameters, then instantiating the template to wrap each class of beans in a container, and finally adding the execution environment including the database. Calls between beans, and calls to the database, are part of the final assembly. The model may be calibrated from running data, or by combining

- knowledge of the operations of each bean
- pre-calibrated workload parameters for container and database operations.

To illustrate this procedure, a simple system with a stateless session bean class and an entity bean class is used; this model was calibrated in the tests described in the next section.

The system chosen was based on the well-known Duke's Bank Application which is shipped with the J2EE documentation provided by Sun Microsystems [3]. The "Update Customer Information" Use Case was specialized to update e-mail information for customers. Figure 8 shows the scenario. It follows the EJB session façade pattern in which a Stateless Session Bean *CustomerControllerBean* delegates service requests from clients to an Entity Bean *CustomerBean*. The Session Bean first finds the required Entity Bean instance by Primary Key (PK) and then updates its email information.

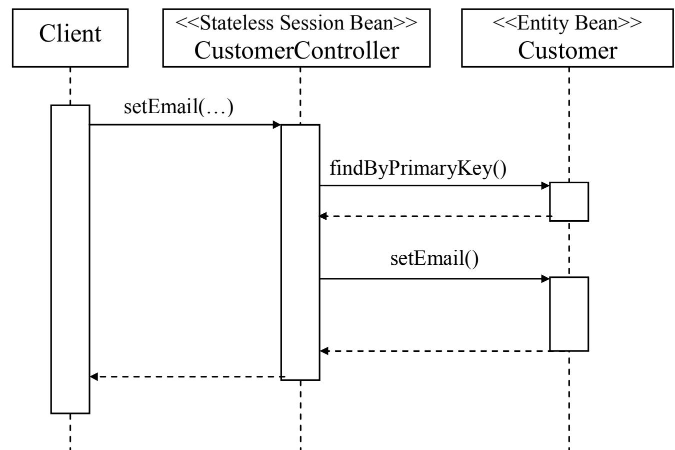

Figure 8 Update Customer Information Scenario

Figure 9 shows the completed LQN model for this scenario. The *CustomerControllerBean* sub-model instantiates the Stateless Session Bean template shown in Figure 5. The *CustomerBean* sub-model instantiates the Entity Bean template shown in Figure 4. The entries related to business methods are instantiated by the entries named *InvokeSetEmail*, *InstanceSetEmail* and *SetEmail*, with their parameters such as execution demands. The *CustomerControllerBean* requires two external services during the *SetEmail* operation, to find the customer by its primary key and update the email of the customer, giving two instances of the *serviceRequest* outgoing interface which are connected to the incoming interfaces of the *CustomerBean*.

This application uses Container Managed Persistence (CMP) strategy in which transactions are managed by containers. A transaction is started at the beginning of an invocation on the session bean *CustomerController* and is committed and ended right before the operation on session bean is completed. Any change on entity data is updated into database during the transaction committing stage. Therefore, the entity store operation is actually invoked by the session bean during its critical section for bean context swapping (represented by *prepareBean* in the model).

Underlying services from execution environment including *DataBase* and processing resource ApplicationCPU (*AppCPU*) are finally added to complete the model structure. Unused services such as bean creation/removal entries are omitted in the model.

The size for Bean Thread Pool $M can be obtained from container configuration during system deployment. The replica parameter $I represents the number of data records in the

database. Assuming that bean instances are accessed with equal probability, the hit rate $p = $M/$I. Alternatively, $p can be observed by measurement or benchmark. The experimental data shows that the hit rate quickly approaches its limit even at initial warmup phases when the number of replica tasks in the model is substantially greater than $M. Therefore, its value was limited to $I' = 3*$M for simplicity of the model.

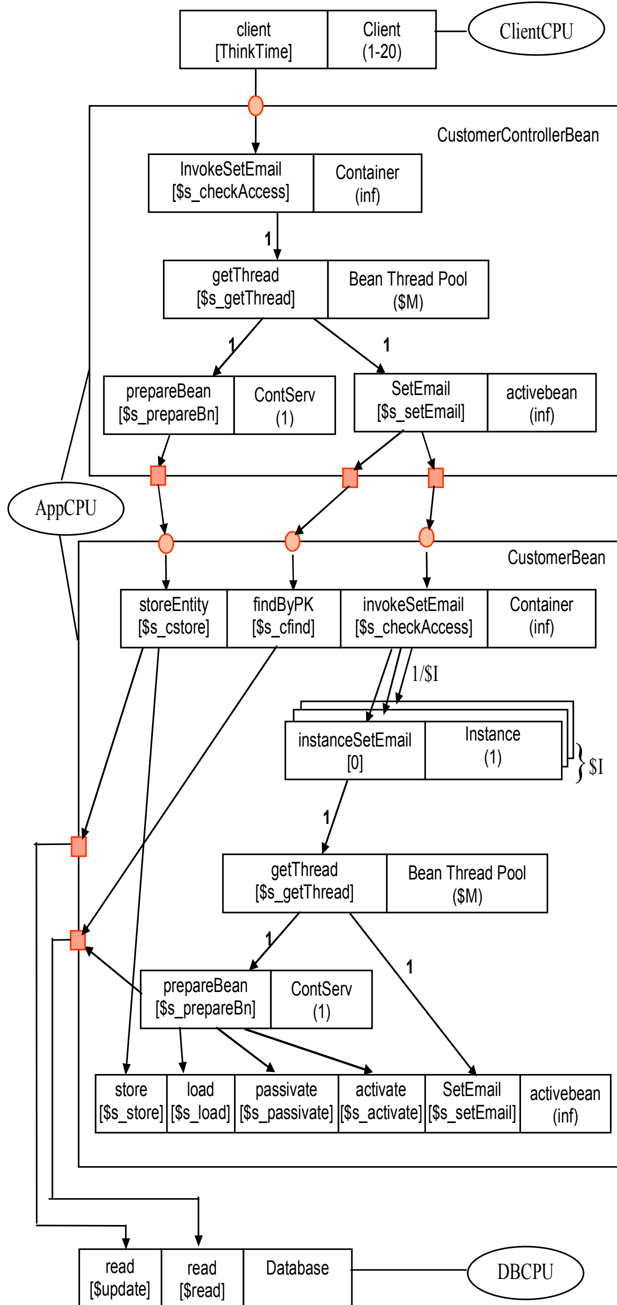

Figure 9 Completed LQN model for Update Customer Information Scenario

5. Application Profiling and Measurement

The Duke's Bank application [3] was modified in two ways: the Entity Beans were modified to use Container-Managed Persistence (CMP), and support for multiple concurrent clients was added.

5.1 Hardware platform

The testing environment includes three x86 machines described in Table 1: one for the EJB server, one for the database server, and one for client request generation. All the machines are connected to a dedicated switched 100 Mbps network. The client machine is more powerful than the servers to make sure that it does not become a bottleneck when generating the test load.

Due to a limited number of database entries in the application, the database server is lightly loaded.

Table 1. Testing Environment Specification

Machine Type	Processor	Memory	HDD I/O System	OS
App Server	PIII-866 Mhz	512 Mb	20 Gb IDE	Debian Gnu/Linux 3.1 (Sarge)
Database Server	PIII-800 Mhz	512 Mb	20 Gb IDE	Debian Gnu/Linux 3.1 (Sarge)
Client	PIV-2.2 Ghz	512 Mb	80 Gb IDE	Debian Gnu/Linux 3.1 (Sarge)

5.2 The software environment

The following software was used for testing purposes:

- operating system: Debian GNU/Linux 3.1 "sarge", kernel v 2.6.8-2
- database server: MySQL v. 4.0.23-7
- application server: JBoss v. 4.01sp1, connected to the database with Mysql Connector/J v 3.0.16.
- JVM: Java2SDK 1.4.2_03 for the application server, Java2SDK v5.0 for the client.
- client: a multithreaded testing suite developed in-house, that calls EJBs in the application server via RMI.

The newer version of SDK introduced some problems to the testbed, so SDK 5.0 was not used for the server setup.

Measurements on the container and program execution were obtained by running JProbe 5.2.1 Freeware profiler for Linux. Additional data, such as pool instance numbers and cache instance numbers were obtained using XMBean examples from the JBoss advanced tutorials. The UNIX sar utility was used to obtain various data available on the performance of the operating systems, including CPU, disk, and network usage.

The following options were used for JVM startup:

- the initial Java heap size was 384 MB to prevent performance loss in variations of the java heap size, with option -Xms384m,
- parameter -XX:+PrintCompilation was set to monitor the runtime behavior of the JVM. Generally, compilation messages suggest JVM is still adapting to certain type of workload.

- parameter -XX:CompileThreshhold was used to monitor the system's behavior with no JVM runtime optimizations

5.3 Testing Scenarios

The following changes were made to the application to emphasize contention for resources and to determine their overhead costs:

- container-managed persistence (CMP) entity beans are used instead of bean-managed persistence (BMP);
- multiple users are supported. The original Duke's bank only supports a single user;
- stateful session beans were converted to stateless session beans;
- the deployment descriptors were modified to limit the size of caches and pools for the beans to 10 to enforce activation/passivation, and artificial congestion at the pool/cache level.
- timeouts for bean passivation and removal was also significantly decreased to 5 seconds to allow shorter waiting time before next batch of client requests.

Two patterns of access to the data were followed by the simulated users. In **sequential access** all the beans are accessed in ascending order. In **random access**, the bean IDs are selected randomly. For profiler measurements a single request was entered and followed, once for each pattern.

For overall performance measurements the load was gradually increased from 1 to 20 users, with step size 1. 20 users for 10 entity beans were necessary to create an artificial situation of activation/passivation for a limited workload. To ensure that the bottleneck remains at the application server, the number of records in the database was kept to a low number of 300.

Every client performed the following loggable sequence in between warm-up and cool-down periods:

1. Update each customer record in ascending order (300 records in total);
2. Wait for other clients to finish.
3. Update a random customer record 300 times;
4. Wait for other clients to finish.

Average response time for each client thread was logged separately for random and sequential calls. Each set of measurements was carried out at least ten times after the warmup to check the results consistency. The thread pool and the cache of the entity beans were also monitored to ensure JBoss's compliance with imposed configuration restrictions. The database host has been 'warmed up' to achieve a constant response time to queries. The application server's operating system was warmed up to minimize its effect on the measurements. JBoss was restarted before the system, so JVM runtime adaptations could be observed.

6. Model Calibration and Comparison with Measurement Results

The measurement results are shown in Figures 10 and 11. Figure 10 shows an average throughput of the system. It can be seen that with 4 clients the system reaches the saturation point. The results produced are very consistent, with standard deviation (stdev) of 1.25 and 95% confidence intervals (CI95) of ±0.77. Average response time is shown in Figure 11. Both random and sequential access patterns are represented. For the random pattern, CI95 = ±1.94, and for the sequential pattern CI95 = ±2.01.

Clearly the test system is bottlenecked for as few as two customers, because the synthetic clients had zero "think time" between requests. Thus the throughput levels out and the response time becomes steadily greater as clients are added, due to waiting for the saturated processor.

There is a slight decrease in response time in the neighborhood of 4 users. This is due to the JVM adaptation (such as real-time compilation) that takes place during the initial part of the test. By the time the responses are recorded for 5 - 7 users the system has stabilized and thereafter the response time grows linearly with the number of users.

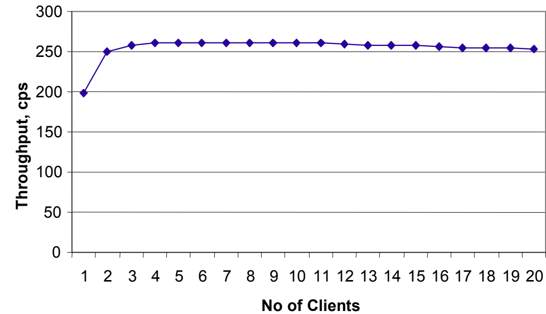

Figure 10 Observed Throughput vs Number of Clients

In Figure 11 the random access times are lower, probably because in sequential access every bean request requires an activation, while for random access there is a probability that the bean is already active.

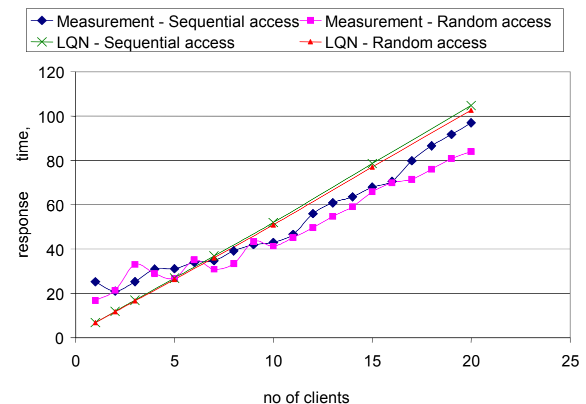

Figure 11 Comparison of LQN results with measurement results

6.1 Model Calibration

The model constructed in section 4 was calibrated from the profiling data under a single-client workload. Two factors are used to adjust the parameter values when doing the calibration.

First, because the JProbe profiling tool itself introduced significant overhead on the execution, the execution demand values extracted from profiling data are adjusted to remove the contribution of overhead. This was done by using a *Profiling Ratio Factor* (PFC) based on the assumption that the profiling

overhead is proportionally distributed across the operations within some section of the scenario. The factor was obtained for each section by measuring the service time with and without profiling and taking the ratio. For Session Bean container services, PFC was 2.18. For Entity Bean container services, two PFC values were found. For finder operations, PFC was 3.78; for business method related operations, PFC was 7.81. The reason for the difference is that many more of the underlying methods are profiled than business methods. For general container services, such as security check operations, PFC was 2.98. For low-level operations, in which no underlying services are profiled, PFC=1.

Second, during the measurements, the system warm up was observed when the same operations are repeated a number of times. JVM runtime optimizations are described in [11],[12]. For the server type JVM used here these include, but are not limited to, dead code elimination, loop invariant hoisting, common sub-expression elimination, constant propagation, null and range check eliminations, as well as full inlining and native code compilation.

The effect of JVM optimizations on the response time for the customer information update scenario used in this study is shown in Figure 12a. After initial volatility the response time stabilizes as shown in Figure 12b. The sporadic delays of about 150 ms are due to Garbage Collection. Setting the "CompileThreshold" parameter of JVM sufficiently high effectively turns the optimizations off.

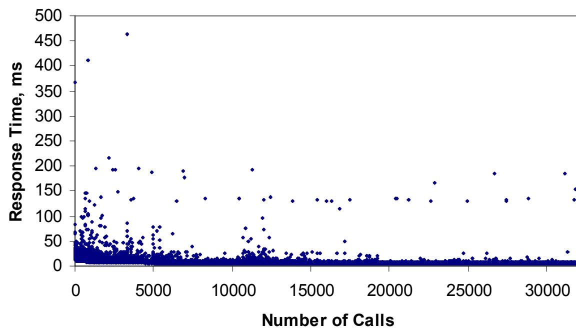

Figure 12a Response time Variation

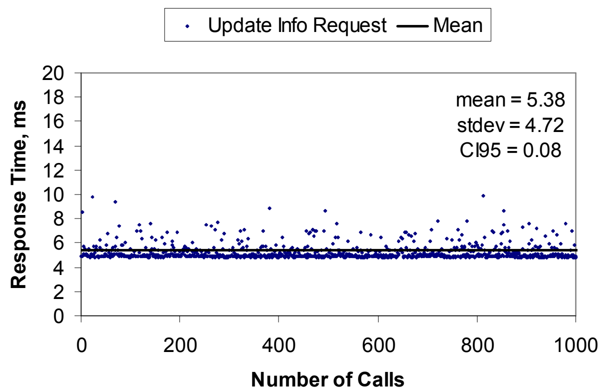

Figure 12b Response time with an optimized JVM

An unoptimized JVM becomes stable after just a couple dozen calls with a mean of 15 ms and sporadic Garbage Collection delays of approximately 220 ms. This initial delay is mainly caused by lazy loading of classes at JVM. Additional possible factors include OS processes, such as Virtual Memory Manager (VMM) moving swapped out pages back to RAM, the process and IO scheduler optimizing for the load, and connection establishment to the machine that runs the MySQL database.

Since the performance results were obtained by repeatedly invoking the same scenario, they are mostly warm system results. However, the profiling was done for a "cold" state with a single client in the system. A *Warm System Factor* (WSF) was introduced to adjust the service time values extracted from the profiling results, defined as the ratio of cold system state times over warm system state times. Based on the way performance measurements varied as the system warmed up, a factor of 3 is used for WSF.

After applying these factors, the following parameters were used in the model calibration:

- For CustomerControllerBean:
  $M = 10 $c_CheckAccess = 0.817ms
  $s_getThread = 0.002ms $s_prepareBn = 0.280ms
  $s_setEmail = 0.010ms

- For CustomerBean:
  $M = 10 $I' = 30 replicas
  $s_cstore = 0.303 $s_cfind = 1.740ms
  $s_checkAccess = 0.513ms $s_getThread = 0.003ms
  $s_prepareBn = 0.257ms $s_store = 0.003ms
  $s_load = 0.003ms $s_setEmail = 0.120ms
  $s_passivate = 0.001ms $s_activate = 0.001ms

- For Database
  $update = 2ms $read = 0.4ms

For sequential access, the hit rate $p should be 0 since the required bean instance is always in passive mode and needs context swapping every time. For random access, $p should 0.033, which is $M / $I = 10/300.

6.2 Model Predictions and Accuracy

Response time prediction results obtained from solving the model calibrated with above parameters are also shown in Figure 11. LQN predictions are low for small $N and high for large $N, which is due to progressive adaptation of the JVM during the experiment. For large $N the differences are between 6.2% to 23.9% for the sequential access case and 2.1% to 24.5% for the random access case.

The constant WSF for the impact of system warm-up is a compromise between the colder states at the left and the warmer states at the right. The greatest WSF that was observed in measurement is about 5, but it was not stable.

Software running in JVM continuously goes through optimizations by the VM. It is therefore proposed to use a range of values for CPU demand prediction, corresponding to warm system factor (WSF) values between 1 and 5. The value of 3 was used as a compromise in the middle of the range.

The LQN model also predicts resource utilizations which can be used to diagnose bottleneck. Model results show that with parameters taken in a cold state, the application processor is the bottleneck with 80%-95% utilization. With parameters that adjusted by WSF = 3, the bottleneck moves to the Bean Thread Pool of the Session Bean. Both these predictions are verified by measurement results. For a few clients the thread pool is not

saturated, but for many clients it is. The saturated thread pool does not slow down the system (or reduce the processor utilization) since the database was artificially configured to be very fast through a decrease of records in corresponding tables. This thread pool just limits the number of beans which are actively being processed at one time. However with a slower database, it could limit performance severely.

7. Conclusions

A template-based framework has been described for rapidly building predictive models of applications based on J2EE middleware. 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. The paper shows a complete procedure of constructing, calibrating, solving and analysis of the model for a real system.

The approach here has been customized to Enterprise Java Beans in a J2EE application server, but a similar approach can be applied to other technologies such as .NET. The framework uses Layered Queueing Network models, which can represent the various types of resources.

The LQN correctly predicts resource saturation of processor and thread resources. Predictions are affected by JVM adaptation, which must be taken into account when calibrating a model. CPU demand parameters measured on a cold system are up to 5 times of those on a warm system.

Acknowledgement

The authors are grateful for the support of the Informatics Research Initiative of Enterprise Ireland (Mr. Oufimtsev and Dr. Murphy), and of Communications and Information Technology Ontario (Ms. Xu and Prof. Woodside).

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

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

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

[7] 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.

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

[9] 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

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

[11] Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani, "Evolution of a Java just-in-time compiler for IA-32 platforms," IBM Journal of Research and Development, IBM Research in Asia Issue, Vol. 48, No. 5/6, pp. 767-795, 2004.

[12] Sun Microsystems, "*The Java Hotspot Virtual Machine, v.1.4.1, d2"*, online white paper at http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf, Sep 2002

[13] 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

[14] 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

[15] 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)

[16] 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.

[17] 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.

[18] J. Xu, M. Woodside, "Template-Driven Performance Modeling of Enterprise Java Beans", to appear in Proc. Workshop on Middleware for Web Services, Enschede, Netherlands, Sept. 2005.