Developing Performance Models from Non-intrusive Monitoring Traces

Daniela Mania, John Murphy, Jennifer McManis
Performance Engineering Laboratory,
Dublin City University, Ireland
Phone: +353-1-7007644, fax: +353-1-700-5508
e-mail: {maniad, murphyj, mcmanisj}@eeng.dcu.ie

Abstract: The performance of large-scale component-based applications is an important issue to be considered. Middleware performance offered by technologies such as EJB, .NET, CORBA does not guarantee the fulfilment of performance requirements. We propose a methodology that automatically builds analytical models and drives possible performance improvements for the systems under study. We outline a methodology of discovering the main transactions within the system at run-time using non-intrusive monitoring. Our belief is this methodology will help in the understanding of performance problems within complex systems.

1 Introduction

Developing large-scale complex component-based applications is an emerging strategy in the software industry. Time to market requirements have lead to developers concentrating more on the functionality and less on the performance of applications. The increasing complexity of middleware and the reuse of COTS (Commercial Of The Shelf) components often make the behaviour of an application difficult to understand. Furthermore, when the system is deployed and running, if there are performance problems, it is difficult if not impossible to determine their cause.

Middleware solutions such as Sun's Enterprise Java Beans (EJB), Microsoft's .NET, and OMG's CORBA Component Model (CCM) make complex component-based applications easier to develop. Enterprise Java Bean (EJB) technology has gained significant acceptance among enterprise development teams and platform providers. This is because EJB is a server-side component framework that simplifies the process of building scalable, reliable and secure applications without the need to re-implement complex distributed component services [1]. The Enterprise Java Bean server/container provides automatic support for middleware services such as security, transactions, persistence, remote accessibility, resource management and database connectivity, thus reducing the complexity of application development.

We propose a technique that enables performance metrics such as response time and throughput to be derived automatically from traces obtained by non-intrusive monitoring of an EJB-based e-commerce application. Moreover, the methodology provides behavioural information about the application under study.

The rest of the paper is organized as follows: related work and an overview of EJB technology are presented in Section 2. Section 3 describes the problem that we consider in this paper. The methodology that we propose is outlined in Section 4. Section 5 presents an overview of the performance model that we intend to use. Conclusions and future work are described in Section 6.

2 Background

2.1 Related work

There has been a significant amount of research in developing performance models from traces. However, none of this work is related to EJB-based systems and none of it is uses traces from a real-time non-intrusive monitoring system.

Trace-based Load Characterization (TLC) is a technique proposed by Curtis Hrischuk [2]. This technique uses a special kind of trace called “angio-trace” [3], automatically generated from instrumentation. The “angio-trace” is based on the concept of using a dye to capture the cause and effect relationships for each user-defined and communication event. TLC is based on matching templates to event graphs to recognize event patterns. The shortcoming of this technique is due to the use of templates, since they are limited to the imagination of the developer.

The Model Builder technique proposed by El-Sayed [4] develops performance models through traces obtained from an SDL design tool. Some assumptions are made to simplify the problem. For example, it is assumed that the queues are FIFO with no priorities, thus limiting the ability of the technique to create models beyond this criterion.

Tim Souder et al. [5] proposed a framework FORM for creating views of program executions. Tools are constructed that analyse the run-time behaviour of stand-alone and distributed software systems. UML sequence diagrams are generated to describe system execution. The information used is obtained by profiling Java-based distributed systems, and is thus gained in an intrusive fashion.

2.2 EJB Overview

The Enterprise JavaBean (EJB) standard specifies how server-side components are written and provides a contract (i.e. a set of interfaces) between components and the application server that manages them [1]. There are three types of Enterprise Java Beans as defined in the EJB 2.0 specification [6]: entity beans, session beans and message-driven beans. Entity beans model business data. They represent persistent objects that can be stored in permanent storage. Session beans model business processes. They represent objects that implement business logic, business rules, algorithms and workflow. Message-driven beans are special types of EJB components that can receive Java Message Service (JMS) messages. One of the goals of

message-driven beans is to allow asynchronous programming in EJB-based applications since the entity and session beans use synchronous communication.

Each EJB component consists of well-defined interfaces which obey certain rules, a bean class, and a deployment descriptor. The bean class contains the implementation details of the EJB component. The remote interface defines the exposed business methods of the EJB. The home interface defines methods for creating, destroying and locating beans. The local interfaces are high performance versions of the home/remote interface. By using local interface the overhead introduced by Remote Method Invocation – Internet Inter-ORB Protocol (RMI-IIOP) is avoided for beans situated in the same Java Virtual Machine. The deployment descriptor is an eXtensible Markup Language (XML) file that specifies the bean's middleware requirements: bean management and lifecycle, persistence, and transaction and security requirements.

An EJB component resides in an EJB container that, transparently to the application developer, provides security, concurrency, transactions, persistence, resource management and database connectivity. A client will access a bean instance only through its container.

3 Problem definition

EJB-based e-commerce applications are composed of interdependent EJBs. Potentially, some are developed in-house, some are commercially available, and others are developed by the customer. The EJBs and their interconnections typically vary for each design. Performance prediction is important for such applications to help the designers select a better design and to adjust the software architecture for better performance.

As mentioned, we are developing a technique that builds performance models from traces generated by non-intrusive monitoring of an EJB-based e-commerce application. How the non-intrusive monitoring is being performed is not described in this study and is presented in [7].

A trace is a record of a sequence of events. Tracing a program is used to understand how the program executes and reveals the dynamic details of the design. A timestamp is attached to every event recorded in the trace to maintain its chronological order. We assume that the correct chronological order of events can be achieved through clock synchronization.

cTime	ID	Duration
1	EJB1.inst1.meth1	70
2	EJB1.inst2.meth2	50
3	EJB2.inst1.meth1	20
4	EJB2.inst2.meth1	25
12	EJB3.inst1.meth2	10
14	EJB3.inst2.meth1	5
24	EJB2.inst2.meth3	25
26	EJB3.inst1.meth3	7
35	EJB3.inst1.meth4	14
40	EJB3.inst2.meth2	20
75	EJB1.inst1.meth2	53
…….		

Figure 1 Trace file

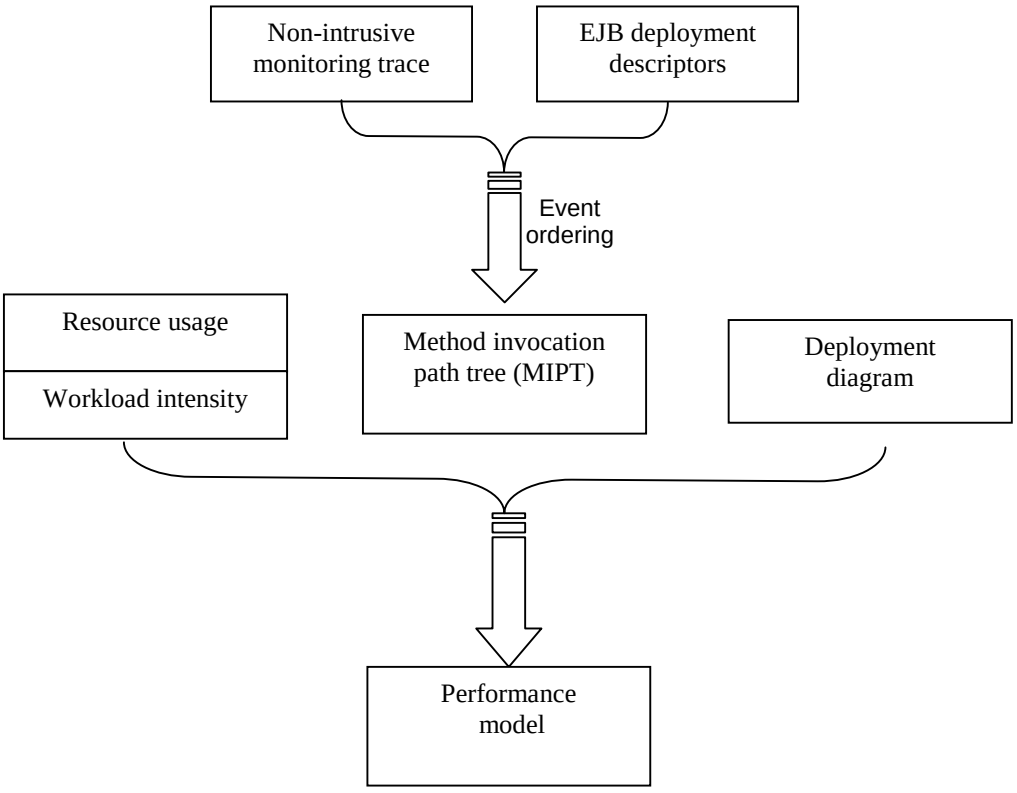
Using a non-intrusive monitoring approach we can obtain the following fields for the events recorded:

1. Identification: the qualified name of an EJB method. This includes the EJB name, instance number, method name;
2. Current time (timestamp): time when the event took place (the method was called);
3. Duration: the cumulative total time spent in that method (including that of all method called);
4. Actual parameters (optional): the value of the method's parameters.

An example of such a trace is presented in Figure 1. It should be noted that traces are observed sequences of events and do not imply causality. It is therefore necessary to attempt to infer causality from the available (incomplete) information.

We propose a technique for building a performance model for an EJB-based system based on our observation of event traces. The technique consists of several steps, shown in Figure 2.

Fig.2 Steps to obtain a performance model



The first step is to build a Method Invocation Path Tree (MIPT) from the non-intrusive monitoring trace using the information offered by the deployment descriptors. The MIPT is a node-labelled, directed, acyclic graph whose structure is based on the causal relationship between events. Each MIPT represents a main transaction within the system. A *transaction* is a causal sequence or sequences of events that correspond to user level tasks such as "browse the catalogue" or "add item to shopping cart". The labelled nodes in the MIPT represent the EJB methods that belong to the transaction represented. The arcs in the MIPT represent the "cause

and effect" relationship. Due to the fact that more than one event sequence may be associated with the same transaction, there is nondeterminism in the MIPT representation. Details of the construction of MIPTs are given in Section 4.

The next step in obtaining a performance model is to combine the MIPTs with the workload model, the resource utilisation function for each EJB method, and the deployment diagram. The workload model comprises two parts: resource usage (i.e. CPU usage, I/O activities, logical resources) and workload intensity (i.e. frequency distribution of requests, request inter-arrival time). The deployment diagram presents the configuration of a set of run-time processing nodes and the components running on each node [11]. We will define an algorithm with the MIPTs, workload model and resource usage model as inputs and the performance model as output. Our performance model will exhibit a high degree of accuracy because the workload parameters and the resource usage have real values obtained by the real-time monitoring. Details of the construction of this model are left to future work, although possible representations of such a model are discussed in Section 5.

4 Method Invocation Path Tree detection

As stated previously, the Method Invocation Path Tree (MIPT) builds up a probabilistic model of event sequences associated with a given user transaction. First, we use our observed event trace, EJB specific information such as which beans may call other beans, and temporal information to determine possible sequences of events associated with a transaction. Then we use our observations over time to build a probability profile denoting the likelihood of specific event sequences being associated with the transaction.

4.1 Definitions

The MIPT is obtained using the information provided by the trace and the deployment descriptors. Due to the way the non-intrusive monitoring is done [7], the context of one method invocation cannot be captured. Therefore the caller of an EJB's method is unknown. However, we can obtain all the EJBs that can potentially call a method of another EJB, namely using the EJB references.

The EJB deployment descriptor declares the EJB references (i.e. the local/remote references from one bean to another). They indicate which EJBs could be called by the EJB that declared them in its deployment descriptor. This information allows us to restrict the number of possible causal sequences. Other information, that we do not use, is also available in the deployment descriptors such as: name and type of the EJB, the business methods that our EJB is exposing and the EJB resources (i.e. Java Database Connectivity driver, J2EE Connector Architecture driver).

For each causal sequence we determine, we can define a likelihood of method invocations. These probabilities must be determined by observing frequencies over a long period of time.

The last piece of information we may have at our disposal is a set of constraints specific to the technology we use (in this case EJB).

We make the following definitions to capture our information regarding, method causality, likelihood of causality, and technology specific constraints.

1) S_C as the set of all EJBs within the system, where each EJB is a set of its business methods;

$$S_C = \{EJB_1, EJB_2, ..., EJB_n\} \ , \ EJB_i = \{m_1, m_2, ..., m_{n_i}\} \ , \ i = \overline{1, n_i} \ , \ N = \sum_{i=1}^{n} n_i \quad (1)$$

2) $C \in M_{n \times n}(S_C)$ as EJB matrix that specifies for every EJB which are the EJB that it can call;

$$C(i, j) = \begin{cases} 1, & EJB_i \xrightarrow{call} EJB_j \\ 0, & \text{otherwise} \end{cases} , \ 0 \le i, j < n \quad (2)$$

E.g.

$$\begin{aligned} &C(i, j) = 1 \ \text{ if } \ EJB_j \in \{ejb\_ref(EJB_i) \cup ejb\_local\_ref(EJB_i)\} \\ &C(i,i) = 0 \ \text{ if } \ EJB_i \in \{sessionBean, message-drivenBean\} \\ &C(i,i) = 1 \ \text{ if } \ EJB_i \in \{reentrant \ \ entityBean\} \end{aligned} \quad (3)$$

3) S_T as the set of the main transactions within the system. Each transaction has associated a probability matrix P_k. The element $P_k(i, j)$ defines the probability that method i is calling method j within transaction T_k.

$$S_T = \{T_1, T_2, ..., T_r\} \ , \ \ T_k \xleftarrow{associate} P_k \in M_{N \times N}(S_C), \ 1 \le k \le r \quad (4)$$

$$P_k(i, j) = \begin{cases} 0, & (m_i \not\rightarrow m_j) \,|\, T_k \\ p, & (m_i \rightarrow m_j) \,|\, T_k \end{cases} \quad (5)$$

4) Δ as the set of constraints specific to EJB technology (i.e. subsequent methods of one EJB should be part of the same instance, lifecycle events are not part of a transaction)

4.2 Training process

In order to determine the main transactions within the system we propose the following methodology. First, we have a training process that provides us the main transactions within the system. The training process consists of parsing the trace file and determines transaction files using intermediate log files. A transaction file will contain all the possible paths of the same transaction within the system.

A) Building step:

1. Select a method that belongs to an EJB (EJB_j) that is an "entry" in the system (i.e. satisfying the relation: $C(i, j) = 0, \forall 0 \le i < n$). Set a variable ct to cTime (see Fig. 1) of the method selected from the trace file. Extract all the rows until cTime > ct + d and create a log file.
2. Parse the trace in order to detect another occurrences of the method selected in the Step 1 and update ct.
3. Repeat 2, 3 until the end of the trace.

This step builds a log file containing all possible paths through the system that begin with the method selected in step A1 and eliminating all events that could not be part of the transaction's event sequence. We repeat the procedure presented above for every "entry" method within the system. Generally, the front end of an EJB system consists of the web-components such as servlets or JSP files. We can assume without loss of generality that the entry points in the system are known.

B) Refining step:

1. Parse each "transaction" within the log file from bottom to top and eliminate all methods that cannot be part of a real transaction due to timing constraints (i.e. methods x that satisfy the relation: $(\text{cTime} + \text{Duration})_x \geq (\text{cTime} + \text{Duration})_1$);
2. Construct in a backtracking manner all possible candidate real transactions, taking into account cTime and Duration of each method;
3. Eliminate the candidate real transactions obtained before that do not conform to the set of constraints ($\boldsymbol{\Delta}$) specific to the technology used;
4. Select the next candidate transaction from the log file and repeat steps B2, B3 and B4 until there are no more candidate transactions to be processed.

The above procedure is repeated for all the log files obtained in step A. The refining step can be combined with frequency pattern algorithms to improve the results.

4.3 Merging and recognition process

The next process is the merging process in which we build the MIPT for each transaction. A MIPT for one transaction is built by merging the possible paths through the system obtained within the transaction file. An example is presented in Figure 3. The graph nodes are the methods within the transaction. They are annotated with metrics such as mean response time and method call count. The edges represent the calls between the two methods. We consider the time proceeding from top to bottom and from left to right. In order to determine the nested method calls within a transaction one has to traverse the graph in a prefix manner.

We define two special nodes: a "fake node" (FN)and a "fake root" (FR). The "fake node" is used to describe different paths through the system of the same transaction. It will split the transaction graph in two sub-graphs. Each of it may be executed with a certain probability. The "fake root" is introduced as a root to a new sub-graph. In Figure 3, the fake root replaces the node labelled EJB2.meth3 that is the real root of the sub-graph defined by the fake node.

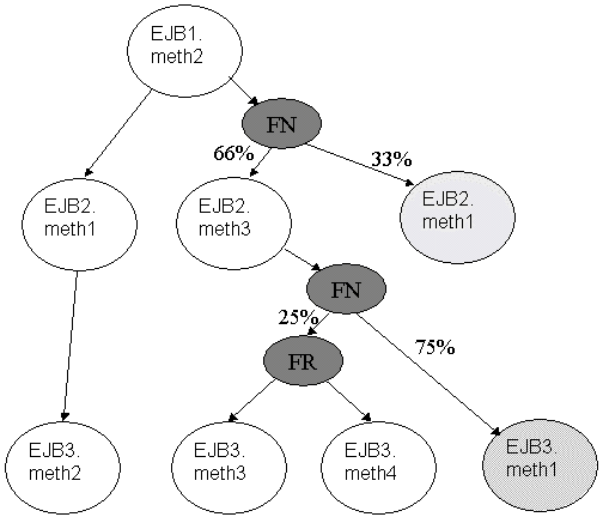


Figure 3 Method Invocation Path Tree example

The last step in our methodology is the recognition process step. This involves the recognition of the current state of the model in real time. We will keep "learning" the details of the system's transaction. Moreover, we will detect the inter-arrival time distribution for each transaction, the method call counts, the mean response time within each transaction, and so on. All the information obtained in this step will be used in constructing the analytical model of the system.

5 Performance Models

Queueing models have been used successfully in performance analysis of software systems. A shortcoming of queuing models is that they can model only software components that demand one-at-a-time resources. Simultaneous resource demands and parallel sub-paths require a more sophisticated model, such as Extended Queueing Networks, Performance Petri Nets, Stochastic Activity Networks, and Stochastic Process Algebras. Layered Queueing Networks (LQN) [8,9] are a new adaptation of the Extended Queueing Network defined specifically to represent the fact that software servers are executed on top of other layers of servers and processors, giving complex combinations of simultaneous requests for resources. They are used to detect and eliminate software bottlenecks, for scalability analysis and capacity planning and for real-time systems with software deadlines. LQN is capable of modelling most of the features that are important from a performance point of view such as multi-threaded processes, devices, locks, and communication. For this reason, we choose LQN for our performance model.

LQN defines a system in terms of its objects (software and hardware). These objects are represented as *tasks* that are divided into three categories: *client tasks* (only sends requests), *active server tasks* (can receive and send requests) and *pure server tasks* (only receives requests). Each task accepts service requests as *entries* that can correspond to methods exposed by the software objects. Each entry has its own execution time and demands other services. A task has its own queue with a queue discipline (i.e. FIFO) for all the services it exposes.

LQN presents three communication protocol types: asynchronous (a task that requests a service and does not wait for the reply of the request), synchronous (a task that requests a service and waits for the reply), forwarding (a task that sends a synchronous request to another task that forwards it in an asynchronous way to another task). The term layered does not imply a strict layering of tasks. A task may call another task from the same layer or may skip a task.

Surrogate delays are used to solve LQN to resolve the simultaneous resource possession problem arising from the nested calling pattern in the system under study. The model is divided in small MVA sub-models that are resolved and iterated until the convergence in waiting times is reached. The interaction between MVA sub-models occurs by the use in a sub-model of other sub-models results in the calculation of service and thinking times. Typical LQN results are response times, throughputs, utilization and queueing delays.

EJB technology can be modelled using Layered Queueing Networks [10]. Figure 4 presents an entity EJB as three-layer LQN model. The first task will represent the entity EJB and business methods exposed in remote interfaces will be the entries of this task. The second layer is represented by data management system (DBMS) that accepts database connection requests from entity EJB, processes SQL statement and retrieves/updates data. The third layer is represented by the hardware, in this case disk subsystem.

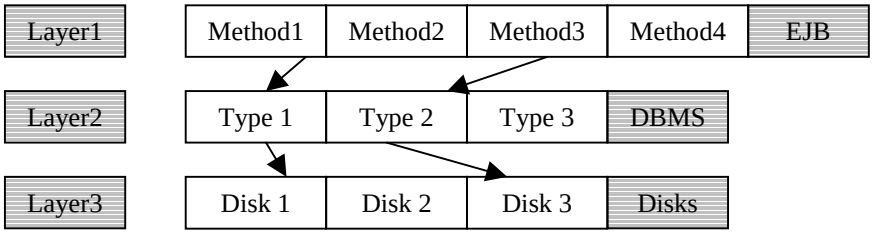


Fig.4 Layer Queueing Model for entity EJB

There are services in EJB technologies provided by the middleware that are difficult to model using the layered queueing. We will concentrate in modelling EJB based application using queueing theory taking into account the issues that this technology raised in the performance-modelling domain.

6 Conclusion

Performance of EJB-based e-commerce applications is a relevant issue. We have proposed a technique that extracts a performance model of an application from traces obtained using non-intrusive monitoring techniques. A complete procedure is still under development. Future work includes the improvement of the technique, the implementation of the algorithm that automatically constructs the MIPT and the development of the second algorithm that automatically transforms the method invocation tree into a Layered Queueing Model.

7 Acknowledgement

We are grateful for the support provided by Enterprise Ireland's Informatics Research Initiative.

References

[1] Ed Roman, S.W. Amber and T. Jewell, "Mastering Enterprise JavaBeans", second edition, John Wiley&Sons, Inc., 2002.
[2] C. Hrischuk, C.M. Woodside, J. Rolia and R. Iversen, "Trace-based load characterization for generaling software performance models", IEEE Transactions on Software Engineering, vol. 25, no. 1, January 1999.
[3] C. Hrischuk, J. Rolia, C.M. Woodside, "Automated generation of software performance model using an object-oriented prototype", International Workshop on Modeling and Simulation, Analysis, Simulation of Computer and Telecommunication Systems (MASCOTS '95), pp. 399-409, Durham, NC, 1995.
[4] Hesham M. El-Sayed, "A Framework For Automated Performance Engineering of Distributed Real-Time Systems", PHD Thesis, Dept. of Systems and computer engineering, Carleton University, 1999.
[5] Tim Souder, Spiros Mancoridis and Maher Salah, "FORM: A framework for creating views of program executions",
[6] Sun Microsystems, "Enterprise JavaBeans Specification, version 2.0", August 2001.
[7] Adrian Mos, John Murphy "Performance Monitoring Of Java Component-Oriented Distributed Applications", IEEE 9th International Conference on Software, Telecommunications and Computer Networks - SoftCOM 2001, Croatia/Italy, October 9-12, 2001
[8] J. R. Rolia and Kenneth Sevcik, "The method of layers", IEEE Transactions on Software Engineering, Vol. 21, No. 8, pp. 689-700,1995.
[9] C.M. Woodside and G. Ragunath, "General Bypass Architecture for High-Performance Distributed Algorithms", *Proc. 6th IFIP Conference on Performance of Computer Networks*, Istanbul, Oct. 23-26, 1995, in "Data Communications and their Performance", eds. S.Fdida and R.U. Onvural, Chapman and Hall, 1996, pp 51-65.
[10] Te-Kai Liu, Santhosh Kumaran, Zongwei Luo, "Layered Queueing Models for Enterprise Java Beans Applications", IBM Research Report, June 2001.
[11] G. Booch, J. Rumbaugh and J. Jacobson, "The Unified Modeling Language Reference Manual", Addison-Wesley Object Technology Series, 1999.
[12] R.G. Franks, "Performance Analysis of Distributed Server Systems", PhD. Thesis, Dept. of Systems and computer engineering, Carleton University, 1999.