



# The University of Sydney

## **Design Level Performance Modeling of Component-based Applications**

Technical Report Number 543

November, 2003

**Yan Liu, Alan Fekete**

*School of Information Technologies  
University of Sydney*

**Ian Gorton**

*Pacific Northwest National Laboratory  
Richland WA, USA*

ISBN 1 86487 615 8

**School of Information Technologies  
University of Sydney NSW 2006**

# Design Level Performance Modeling of Component-based Applications

Yan Liu<sup>1</sup>, Alan Fekete<sup>1</sup> and Ian Gorton<sup>2</sup>

<sup>1</sup>[[jennyliu,fekete@it.usyd.edu.au](mailto:jennyliu,fekete@it.usyd.edu.au)];<sup>2</sup> [ian.gorton@pnl.gov](mailto:ian.gorton@pnl.gov)

<sup>1</sup>*School of Information Technologies*

*University of Sydney – Australia*

<sup>2</sup>*Pacific Northwest National Laboratory*

*Richland, WA 99352, USA*

**Abstract.** In this paper, we present an approach to predict the performance of component-based applications in the design phase of development. We can build a quantitative performance model for a proposed system design. The inputs needed to produce this performance prediction are a state diagram showing the main waiting and resource usage aspects of the proposed system architecture, and measurements taken on the middleware infrastructure using a simple benchmark application which is much cheaper to implement than the full system. The performance model allows the system designer to make decisions between alternative architectures and implementation approaches, in terms of their scalability, and ability to achieve required service levels. We show our method in action using a J2EE application, Stock-Online, and validate these predictions by implementing the design and measuring its performance. The modeling approach is applicable to applications built on common middleware technologies such as CORBA, J2EE and COM+/.NET.

## 1. Introduction

The component-based approach has proved successful in the construction of enterprise-scale systems. A range of technologies such as J2EE, CORBA, COM+/.NET, each allow designers to deploy separate components in an environment where middleware infrastructure provides essential support for aspects such as naming/binding, messaging, persistence, transactions and security [4]. While the middleware reduces much of the complexity of coding a distributed application, it has been a challenge to design a system with confidence that it will perform well enough to meet Service Level Agreements (SLAs). The overall performance of a deployed system depends on architectural decisions in the design of components, on the particular implementation of the middleware, and on the many tunable parameters of the middleware, as well as on the client load [5]. Furthermore, these aspects interact in subtle ways, so (for example) the relative scalability of three different software architectures may be reversed if the hosting middleware product is changed to another which implements the same technology [6].

Some of the decisions taken in the development of a component-based system can be adjusted easily, late in the development lifecycle. For example, in order to improve performance, the size of a thread pool can be configured at deployment. However some decisions must be taken early, and change is thereafter very expensive: for example the choice of the technology (J2EE vs .NET) and the choice of system architecture need to be made irrevocably, long before substantial

coding takes place. An unwise decision at design-time could make it impossible to achieve the required performance level once the system has been delivered. Consequently, the designer needs to be able to predict the performance of the finished system, working from a design but without access to a complete implementation of the application.

An architect could use a performance model to answer questions such as the following:

- What are the maximum load levels that the system will be able to handle? If client load increases by a given amount, what level of extra hardware is needed to maintain the required performance?
- What are the average response time, throughput and resource utilization under the expected workload?
- Which components have the largest effect on the overall system performance and are they potential bottlenecks?
- What performance benefit would be obtained by various design changes? Can tuning achieve the required performance?

The common practice to address these problems is to build a prototype. For a complex application, this is expensive and time-consuming. Our approach avoids the need for a prototype, since we can determine the overall form of the performance equation from the design description. We can then estimate the numeric parameters of the equation by measuring the middleware product performance with a simple benchmark, which is much simpler in both code and architecture than the proposed design.

This research fits within the goals of prediction-enabled component technology [8]. Our work is more narrowly focused than [8] as we deal only with performance rather than general quality attributes. However, models for the class of middleware-hosted applications our work addresses must deal with the subtle interactions between the infrastructure and domain components. A particular contribution of this paper is our ability to disentangle the influences of performance characteristics of the infrastructure from those of the application's architecture. This is achieved without the need to inspect or instrument the middleware infrastructure code itself.

Our work is also related to the tradition of analytical performance modeling of computer systems, based particularly on queuing theory [9][12][15][16]. These techniques have been applied to component-based middleware-hosted applications [14]. One approach has been to model the complete system at a very detailed, physical level [2]. In contrast, other researchers have worked at the abstract level of the software architecture [3][10][13]. These efforts generally lead to a performance model giving the overall form of the equation. However, a quantitative prediction requires application-specific parameters, which can be obtained only from a substantial prototype implementation.

Another thread of research on the design of component-based applications to meet performance goals has been aimed at more qualitative prediction [1]. This research is often based on detailed measurements of a benchmark application, but the outcome is insight into the aspects of a design with performance consequences [1][5].

In this paper, section 2 describes our new approach to provide a quantitative performance model. This is followed by a case study where we predict the performance of an application used in expository literature. We describe the Stock-Online application in section 3, and in section 4

demonstrate our performance model for three different architectures in a J2EE implementation of Stock-Online. We also validate the predictions by coding these designs and measuring actual performance.

## 2. Framework of Our Approach

An overview of our modeling approach is shown in Figure 1. We follow five phases:

1. **Modeling.** We establish a general model for the chosen technology, by identifying the main components of the system, and noting where queuing delays can be expected. We abstract details of the infrastructure components and their communication.
2. **Calibrating.** An architectural choice can be mapped to a set of infrastructure components and their communication pattern[3]. The operations of service components can be further aggregated into computing modules. Calibrating the performance model means deriving mathematical models with parameters characterizing those computing modules.
3. **Characterizing.** For a given application, we can determine how often each module is executed. This will depend on the business logic, which tells us how often methods are called, and what operations are performed by what computing modules.
4. **Benchmarking.** The above produces a performance prediction for the designed system in the form of an equation with parameters. Some of the parameters represent observable or tunable features of the configuration, but other parameters reflect internal details of the middleware platform. We therefore implement a simple application, with minimal business logic and a simple architecture, on the target middleware platform, and measure its performance. Solving the performance model corresponding to the simple application allows us to determine the parameter values, which we describe as the performance profile of the platform.
5. **Populating.** The parameters of the middleware performance profile can be substituted into the performance model of the designed system, giving the required quantitative prediction of performance of that system.



Figure 1 The performance prediction framework

### 3. Predicting Stock-Online Performance

#### 3.1 Motivation

Stock-Online [4] is a simulation of an on-line stock-broking system. It models typical e-commerce application functions and aims to exercise the core features of a component container. Stock-Online supports six business transactions and it enables users to buy and sell stock, inquire about the up-to-date prices of particular stocks, and get a holding statement detailing the stocks they currently own. The supporting database has features to track customers, their transactions, payments, and so on. There are four database tables to store details for accounts, stock items, holdings and transaction history. The transaction mix can be configured to model a lightweight system with read-mostly operations, or a heavyweight one with intensive update operations. The process of determining the transaction mix of the workload requires an understanding of the business domain, a data model, and some expectations as to the patterns of usage of the data. Two business models of Stock-Online are and listed in Table 1, one for the *read-only intensive* usage pattern of data and the other one for increasing demand of data update, i.e. the system has doubled updates compared to the first model.

**Table 1 Two Stock-Online business models**

| Usage Pattern   | Transaction      | Transaction mix<br>( <i>Read-only intensive</i> ) | Transaction mix<br>( <i>Doubled updates</i> ) |
|---|------------------|---|---|
| Read-only point<br>(read one record)  | Query stock      | 70%   | 52%   |
| Read-only multiple<br>(read a collection of records)  | Get stockholding | 12%   | 12%   |
| Read-write (inserting records,<br>and updating records; all<br>requiring transactional support) | Create account   | 2%  | 4%  |
|   | Update account   | 2%  | 4%  |
|   | Buy stock        | 7%  | 14%   |
|   | Sell stock       | 7%  | 14%   |

The deployment environment consists of a J2EE application server as the container for Stock-Online and a database for persistence. The container and database execute on separate machines. In the test environment, simulated clients mimic the behavior of web server hosted components under a full, sustained request load. Clients also execute on their own machine.

Suppose that Stock-Online is to be implemented using EJB components and assembled into a single J2EE application. A common solution architecture could use Container Managed Persistence (CMP) entity beans, using the standard EJB design pattern of a session bean as a façade to the entity beans. A single session bean implements all transaction methods. Four entity beans, one each for the database tables, manage the database access. Transactions are container managed.

While this session-façade architecture is certainly a valid approach, there are others could be considered. For example, recall that the *read-only intensive* business model for Stock-Online has 82% read-only transactions. This makes a solution based on the read-mostly pattern [1] a strong alternative to a pure CMP-based design. In this pattern, read-only and read-write operations are separated into two different entity beans. By exploiting the J2EE container's entity bean caching ability, reads from read-only beans are not blocked, reducing the transactional overhead in synchronizing the cache with the persistent data store.

Another option is optimistic concurrency control [11], which is implemented by most J2EE application servers for persistence service. The idea is that no lock is held during a transaction in the

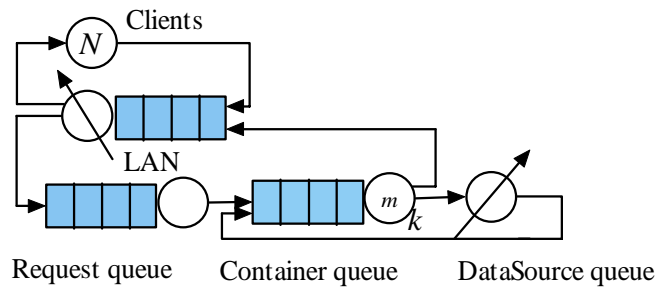
container and the database. The container issues a *predicated update* clause to detect the confliction of transactions at the commit time. If the data in a transaction has been updated by other transactions, the transaction is rolled back.

For an architect, knowledge about which design is most suitable for the application without building all the three solutions is desirable, as is determination of the level of performance that the system provides under load.

### 3.2 Performance modeling

#### 3.2.1 A closed QNM for an application server

A queuing network model (QNM) is used for the performance model. The model captures an application server's behavior in processing a request from a client. The QNM of an application server with a fixed size server thread pool<sup>1</sup> is shown in Figure 2. A closed QNM is appropriate for application servers with finite thread pool capacity as this effectively limits the maximum requests active in the server.



**Figure 2 A closed QNM for an application server**

The clients in the model represent the ‘proxy clients’ of an application server, such as servlets in a web server. Consequently, a client is considered as a delay resource and its service time equals the thinking time between two successive requests. The request handler and container are modeled as single server queues respectively with no load-dependency.

Application servers provide a database pooling mechanism to facilitate connection reuse across requests. The only database clients are in fact the EJBs within the application server, and hence the connection pool can be set to the size of the server thread pool. Database access is then modeled as a delay server with load dependency. The operation time at the database tier contributes to the service demand of the *DataSource* queue.

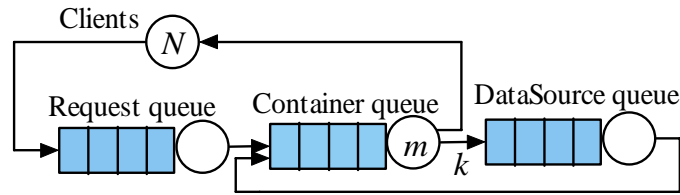
Simultaneous resource possession (SRP) occurs at the *Container* and *DataSource* queues. A server thread is blocked while awaiting replies from the database, and hence is not available to other requests. The key point is that the service demand of Container queue can overlap with the service demand of *DataSource* queue.

A model with SRP can be solved using the Method of Surrogates [9]. In this paper however we use an approximation technique to solve the model using benchmarking. We wish to represent exactly once every service demand and every source of queuing delay exhibiting SRP. When the

<sup>1</sup> For an application server, which dynamically spawns a thread to service each request, the maximum active requests in the server vary depending on the workload in the system. It is important for the application to place limits on the number of concurrent callers into the application server.

workload in the overall network is stable, the utilization of the *Container* queue can be determined. Consequently, the active number of server threads invoking requests at the database can be determined. At this point, we can consider the *DataSource* queue as a load-independent server with a constraint on its client population. The delay at the *Container* queue mainly results from the delay awaiting server thread availability, while the *DataSource* queue delay is mainly due to contention.

The approximation made here is that a logical queuing delay is substituted for the contention delay at the *DataSource* queue. The original model can now be revised into a logically separable QNM, as in Figure 3. We add the constraint value  $m$  to *Container* queue to represent the active number of server threads and  $k$  to the *DataSource* queue to represent its actual client population. The network service demand is ignored because the effects of network traffic are uniform between architectural choices.



**Figure 3** A separable QNM of the application server

This model focuses on the software infrastructure of an application, as the aim is to predict the performance of different architectures before the system is built. To carry out capacity planning at the hardware level, the model can be decomposed to map to hardware resources (e.g. CPU, disk) using Hierarchical Decomposition [12] or Layered Queuing Models [15][16].

### 3.2.2 Model analysis

The performance metric of interest is the average server side response time. We focus on the average response time under different usage patterns, as this simplifies the tasks of modeling and measurement. Our presentation is based on the following notation:

- $N$ : Average requests/client population in the QNM.
- $K$ : Number of queues.
- $X_0$ : Average throughput of the queuing network.
- $D_i$ : Service demand of a single queue, i.e. the amount of time required for a request to be served at queue  $i$ .<sup>2</sup>
- $R$ : Average response time, equal to the total residence time over all queues.
- $u_i$ : Average utilization of queue  $i$ , defined as the fraction of time the resources of queue  $i$  is busy.
- $s$ : Server thread pool size.
- $m$ : Average number of active server threads.
- $k$ : Average number of active database connections.

<sup>2</sup> Hereafter, we denote subscript  $1, 2, 3$  for *Request*, *Container* and *DataSource* queue respectively.

We make approximations when the *Container* queue has multiple threads concurrently servicing the requests. Let  $D_1'$ ,  $D_2'$  and  $D_3'$  be the effective service demand of each queue. The *Request* queue is still a single server queue in this case. Therefore:

$$D_1' = D_1 \quad (1)$$

We assume the service demand of the *Container* queue is effectively divided by  $m$  because the single queue structure guarantees that the requests will be serviced in a first-come-first-server fashion, and no request will wait if any thread is idle.

$$D_2' = \frac{D_2}{m} \quad m \leq s \quad (2)$$

We assume the effective service demand of the *DataSource* queue is a linear function of  $k$ . The database service time is actually affected by various factors, (e.g. transactional attributes, lock management). Those settings are static and hence not explicitly modeled.

$$D_3' = kD_3 \quad k \leq m \quad (3)$$

Using Utilization and Little's Law [12], we have

$$D_i' = \frac{u_i}{X_o} \Rightarrow \sum_{i=1}^K D_i' = \frac{\sum_{i=1}^K u_i}{X_o} \quad (4)$$

$$R = \frac{N \sum_{i=1}^K D_i'}{\sum_{i=1}^K u_i} \quad (5)$$

This model can be solved using Approximate Mean Value Analysis (MVA) for Closed QNM [14].  $D_i'$  is the input parameter and  $R$  and  $u_i$  are the output. Eq.(1)-(3) show that  $D_i'$  depends on  $D_i$ ,  $m$  and  $k$ .

To solve the model, the challenge is to obtain  $D_i'$ , the separated service demand of each queue with single resource. We introduce a notation for the usage of a queue, denoted as  $U_i$   $i \in K$ .  $U_i$  is the percentage of the total service time the application server spent on queue  $i$  during the measurement. It does not include the fraction of waiting time on the processing of services at lower levels.  $U_i$  can be measured by benchmarking. Let  $T_i$  be the service time the application server spent on queue  $i$  during the measurement and  $T$  be the total time of the measurement. In a multithreaded application environment,  $T_i$  is the average time of all the running server threads. The following trivial relationship holds:

$$\begin{aligned} U_i &= \frac{T_i}{\sum_{i=1}^K T_i} \\ u_i &= \frac{T_i}{T} \\ U_i &= \frac{u_i}{\sum_{i=1}^K u_i} \end{aligned} \quad (6)$$

Therefore, using Utilization and Little's Law and Eq.(6) we can have

$$D_i' = \frac{(U_i \sum_{i=1}^K u_i) R}{N} \quad (7)$$

### 3.3 Modeling EJB architectures

We now need to calibrate the application server technology that will host the three alternative designs. Thus we add a superscript to quantities to indicate the relevant architecture. For example,  $D_1^{cmp}$  is the service demand of queue1 (*Request* queue) in the CMP architecture. It proves convenient to abstract the application server's state in order to analyze its service time. We designate that a request resides in one of the following states:

- Request dispatching (RDISP): A request is accepted by the server and dispatched to the *Container* queue.
- Server initialization processing (SINIT): When a thread is available, the request steps through an invocation chain in the server.
- Bean object skeleton processing (BSP): Before the actual business method is invoked, the skeleton code of a session bean object is first invoked.
- Pre-method-invocation processing (PREP): A session bean instance from the pool is associated with the request. The container registers the request with the transaction manager.
- Method processing (MP): The business logic is executed.
- Post-method-invocation processing (POSTP): The container finalizes the request processing. Data updates are committed.

Clearly, the service demand of *Request* queue equals the service time at state RDISP, which can be regarded as a constant. Therefore the three architectures have the same *Request* queue service demand, i.e.  $D_1^{cmp} = D_1^{rm} = D_1^{occ}$ .

SINIT, BSP, PREP and POSTP are modules that a request must pass through. If the time for database operations is deducted, their total service time is constant for all the three architectures, denoted as  $T_0$ . For convenience, these states as a whole are referred to as a composite, CS.

The business logic is processed in the MP state, and it comprises the majority of the service demands on the *Container* queue and *DataSource* queue. The service time of operations in the MP state are modeled as  $f_c$  and  $f_d$  for the *Container* and *DataSource* queues respectively. From the above analysis, we know that  $f_c$  and  $f_d$  are determined by the component architecture, denoted as superscript  $A$ , thus

$$\begin{aligned} D_2^A &= T_0 + f_c^A \\ D_3^A &= f_d^A \end{aligned}$$

Now the problem is to find the expression of  $f_c^A$  and  $f_d^A$ . We first consider only one component involved in a transaction, and model two scenarios:

- *findByPrimaryKey* is invoked on the component's home interface. Only one component is identified by the primary key.
- *findByNonPrimaryKey* is called on the component's home interface. A collection of references is returned.

### 3.3.1 Modeling CMP

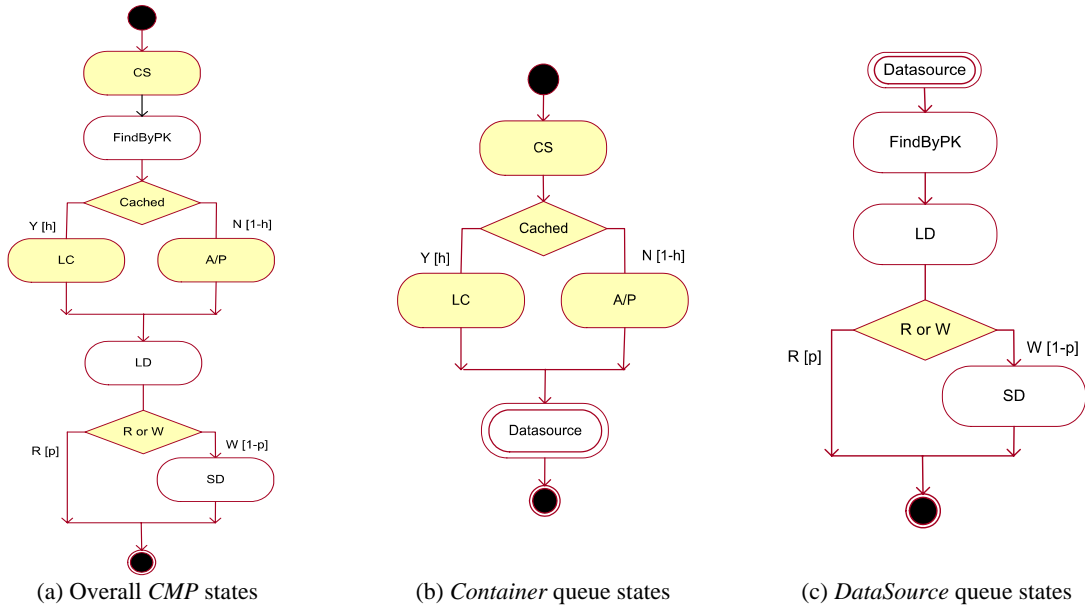
A session bean gets a reference to an entity bean by invoking its *findByPrimaryKey* method in state *FindByPK* in Figure 4(a). If the primary key is present in the database, a reference to the entity bean is returned to the session bean. The container checks its cache to see whether a corresponding bean instance (identified by its primary key) exists. If so it transitions to state *LC*, in which a cached instance is returned. We define the cache hit ratio as  $h$ , so this transition occurs with probability  $h$ . Otherwise, the container transitions to state *A/P*. *ejbActivate*, which incurs expensive serialization operations, is called to associate a pooled instance with a primary key. In the case that the pool is full, *ejbPassivate* is called to serialize a victim entity bean instance to secondary storage. Both the *LC* and *A/P* states are followed by state *LD*, in which a call to *ejbLoad* is made to synchronize the state of the entity bean cache with the underlying database. At the end of the transaction, *ejbStore* is called to store any updates made in the transaction. This is represented by state *SD*.

As the absolute order in the state machine is not crucial for the service demand calculation, we compact *FindByPK*, *LD* and *SD* into a compound state *DataSource* and separate it from states involving the *Container* queue, (see Figure 4(b),(c)). Consequently, we can represent the *CMP* architecture effect on *Container* queue as:

$$f_c^{cmp} = hT_1 + (1-h)T_2 \quad (8)$$

where  $T_1$  and  $T_2$  are the service time of state *LD* and *A/P* respectively. Similarly we model the *CMP* architecture effect on the *DataSource* queue as in Eq.(9), where,  $T_{find}$ ,  $T_{load}$  and  $T_{store}$  are the service time to find, load data into and store updates to an entity bean.  $p$  is the ratio of read-only requests.

$$f_d^{cmp} = T_{find} + T_{load} + (1-p)T_{store} \quad (9)$$



**Figure 4** *CMP* state machine

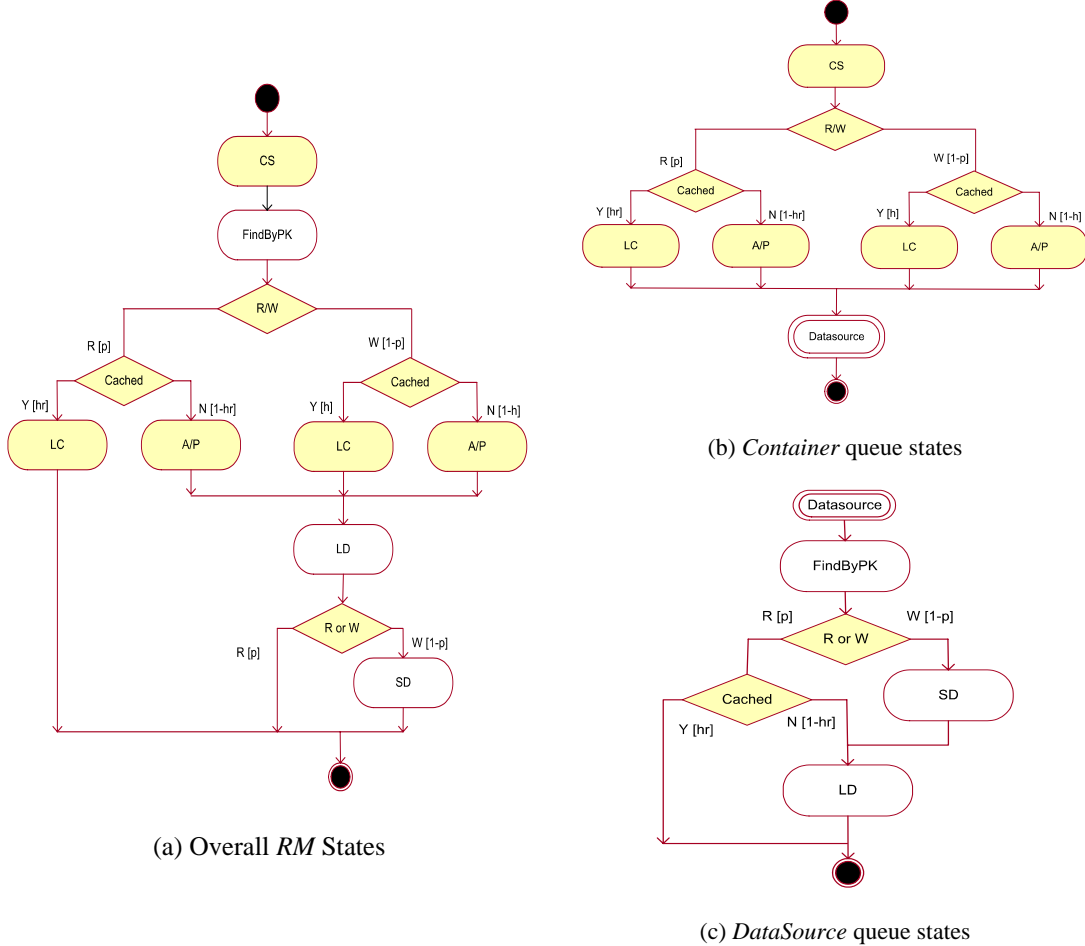
### 3.3.2 Modeling the read-mostly pattern

For *RM*, the container is aware of the request type and accesses the read-only or read-write entity bean cache. For a read-only bean, the container goes to state *LD* only when a cache miss

occurs. We denote the cache hit ratio of read-only beans by  $h_r$ . The container deals with read-write requests in the same manner as the *CMP* architecture. The state machine for this behavior is shown in Figure 5. We therefore model the *RM* architecture as follows:

$$f_c^{rm} = [ph_r + (1-p)h] \cdot T_1 + [p(1-h_r) + (1-p)(1-h)] \cdot T_2$$

$$f_d^{rm} = T_{find} + [p(1-h_r) + (1-p)] \cdot T_{load} + (1-p) \cdot T_{store}$$



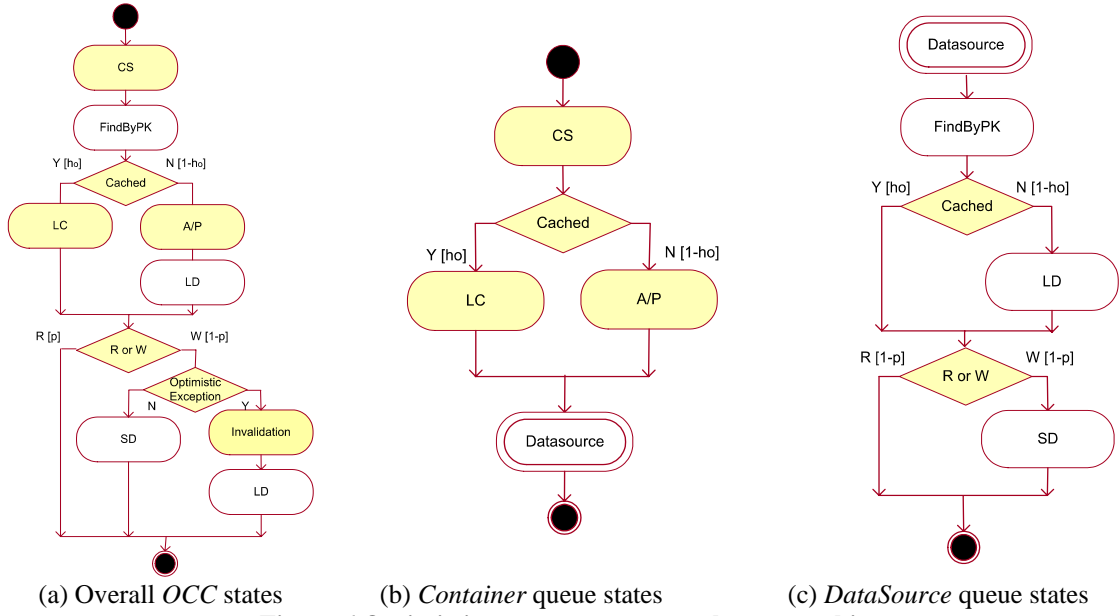
**Figure 5 Read-mostly pattern state machine**

### 3.3.3 Modeling Optimistic Concurrency Control

The state machine of *OCC* is shown in Figure 6. The container invalidates the entity bean instance, whose operation results in the conflicting transaction. *ejbLoad* method is called the next time this invalidated entity is invoked, which is modelled as *Invalidation* state followed by *LD* state. The context information of an invalidated bean is kept valid, which can save the overhead of activating/passivating a bean instance. If there is no conflict detected, *ejbStore* is called at the end of the read-write transaction. *OCC* is desirable if there is a low probability of write contention. We assume that the overhead of invalidation and rolling back a transaction can be ignored when the data size of uniform access is large enough and the ratio of read-write transactions is low. The cache hit ratio of *OCC* entity cache is  $h_o$ . We model *OCC* architecture as follows:

$$f_c^{occ} = h_o T_1 + (1-h_o) T_2$$

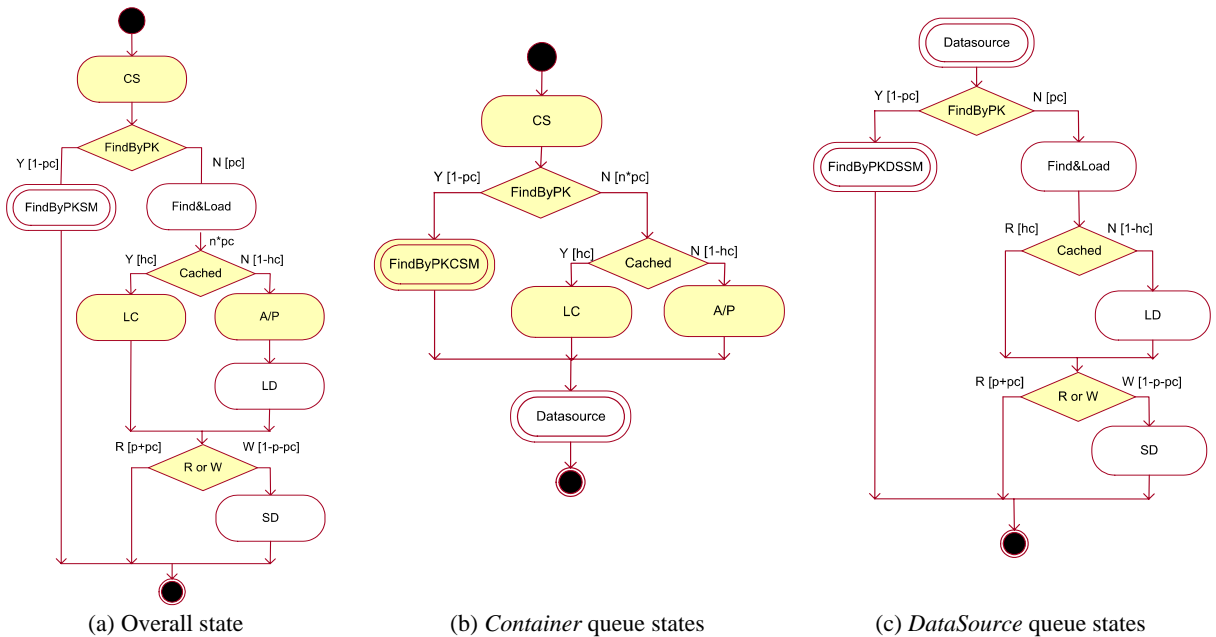
$$f_d^{occ} = T_{find} + (1-h_o) T_{load} + (1-p) T_{store}$$



(a) Overall OCC states (b) Container queue states (c) DataSource queue states  
**Figure 6 Optimistic concurrency control state machine**

### 3.3.4 Modeling find-by-non-primary-key

The *FindByNonPrimaryKey* method of the *EntityHome* interface can return a collection of entity bean references. The container loads all the matched entity beans into the cache when *findByNonPrimaryKey* is called. This is modeled by state *Find&Load* in Figure 7. When *getValue* is called the container uses the cached beans. As bean instances are accessed iteratively, we denote the ratio of state transition as  $np_c$  where  $p_c$  is the ratio of *findByNonPrimaryKey* transactions. Other states are encapsulated into a compound state, *findByPKSM*. In the decomposition of the overall state machine, the sub-state machines of the *Container* and *DataSource* queues with *findByPrimaryKey* transactions are abstracted as states *findByPKCSM* and *findByPKDSM* respectively. The formulae for  $f_c^{col}$  and  $f_d^{col}$  in different architecture models with *findByNonPrimaryKey* transactions are listed in Table 2.



(a) Overall state (b) Container queue states (c) DataSource queue states  
**Figure 7 State machine for *findByNonPrimaryKey* transactions**

**Table 2 Architecture models with *findByNonPrimaryKey* transactions**

|         |                  |   |
|---------|------------------|---|
| CMP_COL | $f_c^{col\_cmp}$ | $(1 - p_c + np_c)h_c T_1 + [(1 - p_c + np_c)(1 - h_c)]T_2$                              |
|         | $f_d^{col\_cmp}$ | $[(1 - p_c) + np_c(1 - h_c)]T_{load} + (1 - p_c - p)T_{store} + T_{find}$               |
| RM_COL  | $f_c^{col\_rm}$  | $[(p + np_c)h_r + (1 - p_c - p)h]T_1 + [(p + np_c)(1 - h_r) + (1 - p_c - p)(1 - h)]T_2$ |
|         | $f_d^{col\_rm}$  | $[1 - p_c - ph_r + np_c(1 - h_r)]T_{load} + (1 - p_c - p)T_{store} + T_{find}$          |
| OCC_COL | $f_c^{col\_occ}$ | $(1 - p_c + np_c)h_c T_1 + [(1 - p_c + np_c)(1 - h_c)]T_2$                              |
|         | $f_d^{col\_occ}$ | $[1 - p_c - ph_c + np_c(1 - h_c)]T_{load} + (1 - p_c - p)T_{store} + T_{find}$          |

### 3.3.5 Modeling multiple EJBs

Now we generalize the analysis for an application with multiple session and entity beans. In the Session-Facade architecture, we assume the instance pool size is large enough, reducing the waiting time to obtain a session bean instance to zero. Let  $p_s$  be the probability of an operation of session bean  $s$  being accessed and  $T_s$  be the average service time of its operation, which doesn't include the time waiting for replies from nested beans' operations. The service time of all session beans is  $\sum p_s T_s$ . For an entity bean, operations can be categorized into four groups, creating a bean, removing a bean, getting value(s) and setting value(s). The time for a container to create and remove an entity bean is denoted as  $T_{create}$  and  $T_{remove}$  respectively. Let  $p_{e,o}$  be the probability of operation  $o$  in entity bean  $e$  being invoked and  $T_{e,o}$  be the average service time. If  $o$  is a getter or setter method,  $T_{e,o}$  is determined by the entity architecture model  $f_c^A$  derived in the above sections.  $T_{e,o}$  also does not include the time spent in nested beans operations.

$$T_{e,o} = \begin{cases} f_c^A(p_{e,o}, T_1, T_2) & o \in \{get, set\} \\ p_{e,o} T_{create} & o \in \{create\} \\ p_{e,o} T_{remove} & o \in \{remove\} \end{cases}$$

$T_s$  can be regarded as the total service time of all but state *SINIT* in *CS*, whose service time is  $T_{SINIT}$ . Hence, the overall service demand of the *Container* queue is

$$D_2 = T_{SINIT} + \sum_s p_s T_s + \sum_e \sum_o T_{e,o} \quad (10)$$

Similarly let  $t_{e,o}$  be the service time of operation  $o$  in entity bean  $e$  to access the database through the *DataSource* queue. This gives:

$$t_{e,o} = \begin{cases} f_d^A(p_{e,o}, T_{find}, T_{load}, T_{store}) & o \in \{get, set\} \\ p_{e,o} T_{insert} & o \in \{create\} \\ p_{e,o} T_{delete} & o \in \{remove\} \end{cases}$$

Here  $T_{insert}$  and  $T_{delete}$  are the time for the database to insert and delete an entity record respectively. The overall service demand of *DataSource* queue is

$$D_3 = \sum_e \sum_o t_{e,o} \quad (11)$$

### 3.4 Characterizing Stock-Online

The above analysis has identified the infrastructure components involved in a specific architecture and established the linkage between architecture characteristics and the overall

performance. Now the task is to characterize the application and determine how much it utilizes the operations of infrastructure components.

We use the concept of a scenario, which proves useful in understanding system and software behavior[7], to define the performance expectations for an application. A scenario traces through the application and can be derived from use cases or class diagrams. It has attributes to specify its name, visit count, type (e.g. read, write, create, remove) and think time (e.g. idle time between two requests in milliseconds). A scenario comprises multiple calls. A call equals a message in sequence diagrams. It has attributes caller, callee, caller scenario and callee scenario to specify the origin and destination of a call. A call also has other attributes such as the number of calls (or invocation count) in a transaction, the type (e.g. synchronous or asynchronous) and iteration count. If a call is remote, bytes sent and received can be specified to estimate the overhead of the network communication.

For example, *BuyStock* transaction in Figure 8 queries an instance of *Account* entity by its primary key. A getter method of *Account* entity, *getCredit* is invoked inside *BuyStock* transaction. The access expectation of *getCredit* method equals to the number of calls in the transaction multiplied by the transaction’s frequency in the transaction mix (7%). Similarly we can derive the access expectation of other entities involved in *BuyStock* transaction. Since an entity bean such as *Account* entity is used by more than one transaction, e.g. *BuyStock*, *SellStock*, *UpdateAccount* etc., we aggregate the access expectation for all transactions that use one entity, giving the metric for each entity bean shown in Table 3 for the two business models we introduced earlier. Note that the transaction frequencies sum to 1, but the access expectations sum to more than 1 (since each transaction access one or more methods).

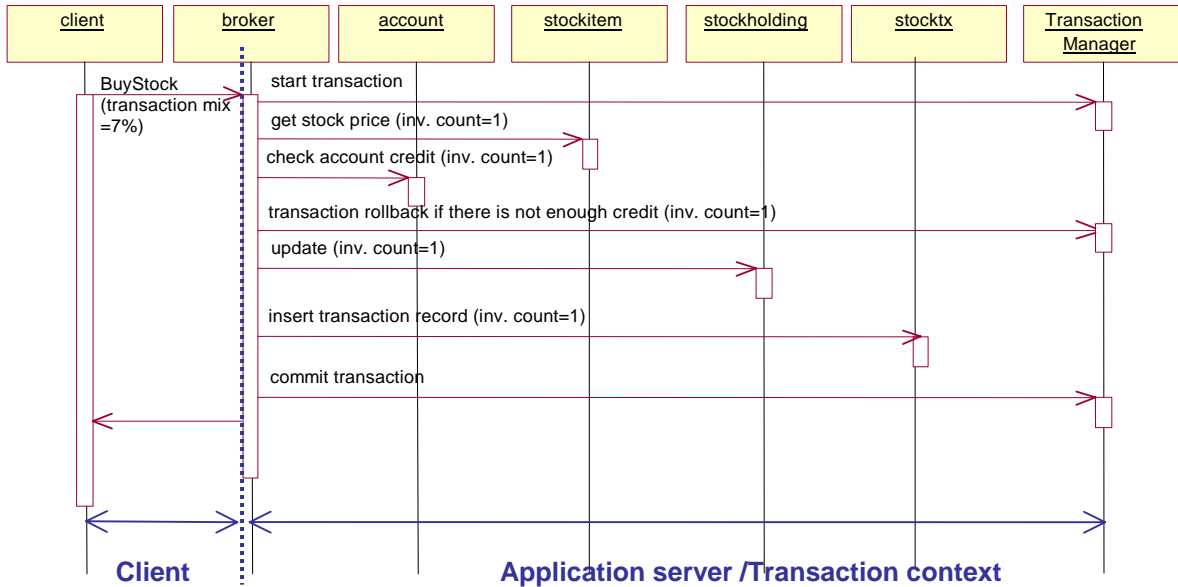


Figure 8 Sequence diagram of *BuyStock*

As the service demand of the *Request* queue is constant, therefore  $D_1^{StockOnline} = D_1^{cmp} = D_1^{rm} \cdot D_2^{StockOnline}$  and  $D_3^{StockOnline}$  can be calculated using Eq.(10) and Eq.(11). Since there is only one session bean involved in Stock-Online, it is easy to conclude that  $T_{SINIT} + \sum_s p_s T_s = T_{SINIT} + T_s = T_0$ .

Three of the Stock-Online entity beans, *Account*, *StockItem* and *StockTx* can be modeled using *findByPrimaryKey*, as their instances are obtained by primary key. *StockHolding* has a scenario to access a collection of entity beans and the collection size is 20 and it is consequently modeled by *findByNonPrimaryKey*. Now it is clear that to predict the performance of Stock-Online with architecture *CMP* and *RM*, the following parameters have to be populated:  $T_0$ ,  $T_1$ ,  $T_{create}$ ,  $T_{find}$ ,  $T_{insert}$ ,  $T_{load}$ ,  $T_{store}$ , and also the cache hit ratio of each architecture must be known.

**Table 3 Metrics of Stock-Online entity beans**

| Entity       | Operation          | Access Expectation<br>(Read-only intensive) | Access Expectation<br>(Doubled updates) |
|--------------|--------------------|---|---|
| Account      | Get                | 13.956%                                     | 27.912%                                 |
|              | Set                | 16.282%                                     | 32.564%                                 |
|              | Create             | 2.326%                                      | 4.652%                                  |
| StockHolding | Get                | 6.978%                                      | 13.956%                                 |
|              | Set                | 13.956%                                     | 27.912%                                 |
|              | Get (iteration=20) | 11.630%                                     | 11.630%                                 |
| StockItem    | Get                | 83.736%                                     | 79.084%                                 |
| StockTx      | Create             | 13.956%                                     | 27.912%                                 |

### 3.5 Benchmarking

Benchmarking is used to populate the performance parameters needed to solve the model.

#### 3.5.1 The benchmark design

The benchmark suite consists of four modules, namely a workload generator, benchmark application, monitoring utility and profiling toolkit.

The benchmark clients simulate active requests from proxy applications, such as a servlet engine in a web server. This kind of proxy client has an ignorable interval between two successive requests. Its population in a steady state is bounded<sup>3</sup>. Hence the client spawns a fixed number of threads in each test. Each thread submits a new service request immediately after the results are returned from the previous request to the application server. The ‘thinking time’ of the client is effectively zero.

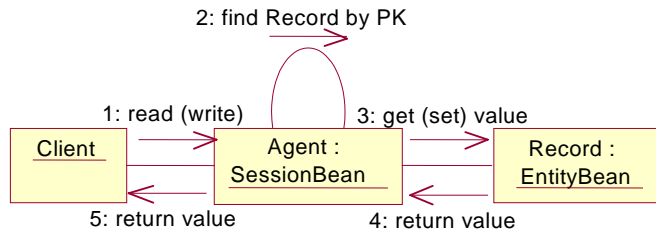
The implementation of the simple application involves a session bean object *Agent* and an entity bean object *Record*. *CMP* persistence is used and transactions are container-managed. The example collaboration diagram in Figure 9 shows the sequence of calls in the benchmark application for a *read/write* request. In Figure 10 *getRecords* event simulates the scenario of an entity, which has *findByNonPrimaryKey* transactions.

The monitoring utility is implemented using the Java Management Extensions (JMX) remote API. It collects performance metrics at runtime, including the number of active server threads and active database connections.

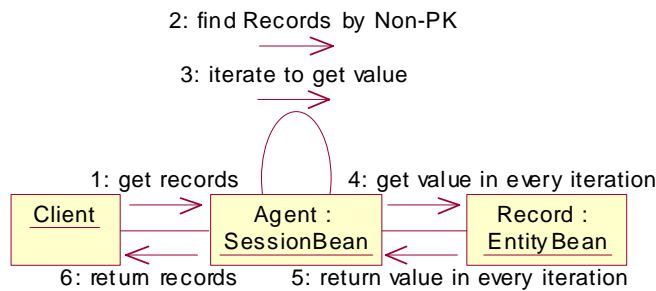
A profiling toolkit *OptimizeIt* is also employed. *OptimizeIt* obtains profiling data from the Java virtual machine, and helps in tracing the execution path and collecting statistics such as the method invocation count and duration of each invocation. Profiling tools are necessary for COTS component-based systems, when instrumentation of the source code is not possible.

<sup>3</sup> A web server has configuration parameters to limit the active workload. For example, Apache uses *MaxClient* to control the maximum number of workers, thus the concurrent requests to the application server are bounded.

The same fundamental design for the benchmark suite should apply to different technologies. It will be the precise implementation details that differ. However, it should be possible to use the same suite for applications that adhere to the same middleware, e.g. J2EE in this case.



**Figure 9 Benchmark application: read/write event**



**Figure 10 Benchmark application: getRecords event**

### 3.5.2 Performance measurement

The benchmarking environment comprises three 4-CPU machines connected by a 100 Mbps isolated network. The client, application server and the database server are deployed on separate machines. The application server used is WebLogic Server 7.0 and the server thread pool size  $s$  is 20. The JVM is Sun's JDK 1.3.1 with settings  $-hotspot$ ,  $-Xms512m$  and  $-Xmx1024m$ . The database server is Oracle 8.1.7 and the Oracle thin JDBC driver is used. Initially, there are 1000 records in the database table. All three machines are running on Windows 2000 server.

Each experiment has three phases, *rampUp*, *steadyState* and *rampDown*. The system is started and initialized in the *rampUp* stage for 1 minute. The system then transfers to *steadyState* for 10 minutes. The statistical results are measured during this time. Each experiment is run several times to guarantee that the difference between runs is minor.

**Table 4 Parameters measured by benchmarking**

| Parameter  | Access Expectation<br>(Read-only intensive) | Access Expectation<br>(Doubled updates) |
|--|---|---|
| Workload ( $N$ )                                     | 100   | 100                                     |
| Throughput (tps) ( $X$ )                             | 513.2                                       | 457.8                                   |
| Response time (ms) ( $R$ )                           | 194.3                                       | 218.4                                   |
| Active # of server threads ( $m$ )                   | 19.97                                       | 19.69                                   |
| Active # of db connections ( $k$ )                   | 9.79  | 10.88                                   |
| CMP cache hit ratio ( $h$ )                          | 50%   | 50%                                     |
| RM cache hit ratio ( $h_r$ )                         | 99.6%                                       | 99%                                     |
| OCC cache hit ratio ( $h_o$ )                        | 60%   | 63%                                     |
| <i>findByNonPriamryKey</i> cache hit ratio ( $h_c$ ) | 96%   | 97%                                     |
| Request queue utilization ( $U_1$ )                  | 15.93%                                      | 14.16%                                  |
| Container queue utilization ( $U_2$ )                | 57.21%                                      | 51.10%                                  |
| DataSource queue utilization ( $U_3$ )               | 26.86%                                      | 34.74%                                  |

In order to measure parameters to solve the performance model, we need to select the workload so that it is large enough to efficiently utilize the server's physical resources but small enough to prevent overload and performance degradation. We scaled the workload from 1 to 1000 clients and measured the throughput. We observed the throughput was 513 transactions per second (tps) when  $N=100$ . The application server CPU utilization was 89% and the database server utilization was 23%. The peak throughput was 527 tps for all clients and the degradation of throughput was 2.66%. We therefore decided to use  $N=100$  as base workload. In order to measure the cache hit ratio, the benchmark application is implemented using *CMP*, *RM* and *OCC* architecture. *getRecords* events are injected to simulate the *StockHolding* entity's usage pattern with *findByNonPrimaryKey* transactions. Table 4 lists the parameters obtained by benchmarking.

### 3.6 Populating parameters

Recall that  $D_i' = \frac{(U_i \sum_{i=1}^K u_i)R}{N}$ . Thus  $D_i'$  depends on  $u_i$  however  $u_i$  is unknown. We further introduce intermediate parameters as

$$D_i'' = \frac{D_i'}{\sum_{i=1}^K u_i} = \frac{U_i R}{N}$$

It can be proved that  $u_i = u_i''$ , where  $u_i''$  is the utilization of queue  $i$  under the service demand as  $D_i''$ .  $D_i''$  can be calculated using data in Table 4, then  $u_i''$  can be solved by MVA algorithm. Consequently  $u_i$  is available and  $D_i'$  can be solved using Eq.(7). The service demand of the benchmark application can be solved in Table 5 using Eq.(1)-(3), from which we can derive other performance parameters.

**Table 5 Benchmark service demands (calculated)**

| Service demand     | Transaction mix<br>(Read-only intensive) | Transaction mix<br>(Doubled updates) |
|--------------------|--|--------------------------------------|
| $D_1^{cmp\_bench}$ | 0.5409                                   | 0.5956                               |
| $D_2^{cmp\_bench}$ | 38.7962                                  | 42.993                               |
| $D_3^{cmp\_bench}$ | 0.09317                                  | 0.1343                               |

Our analysis treated  $D_1^{cmp\_bench}$  and  $D_2^{cmp\_bench}$  as independent of the workload mix  $p$ . The measurement shows that the error is 9.18% for  $D_1^{cmp\_bench}$  and 9.76% for  $D_2^{cmp\_bench}$  when  $p=50\%$ . Approximating  $T_{find} = T_{load}$  in Eq.(8), we get

$$\begin{cases} T_{find} + T_{load} + (1-0.9)T_{store} = 0.09317 \\ T_{find} + T_{load} + (1-0.5)T_{store} = 0.1343 \\ T_{find} = T_{load} \end{cases} \Rightarrow \begin{cases} T_{find} = 0.04135 \\ T_{load} = 0.04135 \\ T_{store} = 0.1032 \end{cases}$$

As reading data from cache is faster than reading data from the database, we use  $T_{load}$  as a substitute for  $T_1$ . Thus  $T_2$  can be solved by Eq.(8). We further assume that  $T_{insert} = T_{store}$ . We also found from the JVM profiling that average service time of state *CS* is about 11.816% of the *Container* queue's service demand, so we estimate  $T_0 = 11.816\% \times D_2^{cmp\_bench} = 4.5842$ . Similarly we can have  $T_{create} = 0.02082$ . Service demands of Stock-Online are calculated in Table 6 using Eq.(10)(11).

**Table 6 Stock-Online service demands (calculated)**

| Service demand      | Business model<br>( <i>Read-only intensive</i> ) |           |            | Business model<br>( <i>Doubled updates</i> ) |           |            |
|---------------------|--|-----------|------------|--|-----------|------------|
|                     | <i>CMP</i>                                       | <i>RM</i> | <i>OCC</i> | <i>CMP</i>                                   | <i>RM</i> | <i>OCC</i> |
| $D_1^{StockOnline}$ | 0.5409   | 0.5409    | 0.5409     | 0.5409                                       | 0.5409    | 0.5409     |
| $D_2^{StockOnline}$ | 50.6077  | 15.0702   | 42.8199    | 58.1064                                      | 27.3369   | 48.5704    |
| $D_3^{StockOnline}$ | 0.1481   | 0.0976    | 0.1149     | 0.2653                                       | 0.2153    | 0.2139     |

The utilization  $u_i$  of a software task is a useful performance metric, from which we can estimate the average number of threads that are busy. We only measure  $m$  and  $k$  when  $N=100$  clients. The values for other workloads are estimated as follows:

The maximum number of requests being served simultaneously by the server threads is the thread pool size  $s$ . When  $N \geq s$ , the capacity of the server threads approaches saturation,  $m_{100}$  and  $k_{100}$  can be used for  $m_N$  and  $k_N$  as approximations. When  $N < s$ , we can regard  $u_2$  as the ratio of active number of threads  $m$  not idle waiting for requests to the thread pool size, i.e.  $u_2 \approx m/s$ ;  $u_3$  as the ratio of the number of active database connections to the number of active threads, i.e.  $u_3 \approx k/m$ . Therefore

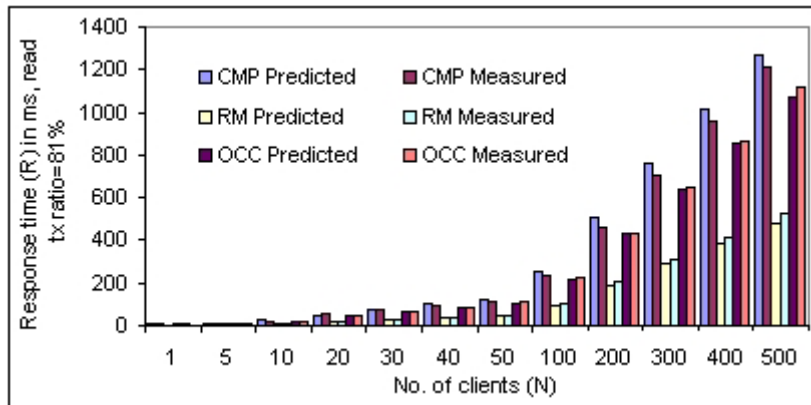
$$\begin{cases} m_N = m_{100}, k_N = k_{100} & N \geq s \\ m_N = \min(m_{100}, s \times u_2(N)), k_N = \min(k_{100}, m_N \times u_3(N)) & N < s \end{cases}$$

### 3.7 Predicting Stock-Online performance

To verify our approach, three versions of Stock-Online were implemented, one using the *CMP* architecture, one using the *RM* architecture and another *OCC* architecture. Each was deployed and run on the experimental environment used for benchmarking. The predicted server side response time was then compared to the empirical results.

#### 3.7.1 Single application server performance

Figure 11 shows the performance results of three implementations of Stock-Online. The error of prediction is around 5-9% and the worst case except for  $N=1$  is about 13% for the three implementations. The prediction is that *RM* architecture significantly improves performance by about 60-65% over *CMP*, while *OCC* architecture has limited performance optimization over *CMP* by about 5-15%. Measurement confirms this prediction.



**Figure 11 Stock-Online Performance (*Read-only intensive* business model)**

When the business model has twice as many updates, we predict the performance in Figure 12. The error of prediction is around 5-17% except  $N=1$ . The error of predicting *OCC* performance is up to 17% when  $N=500$ . We observed that more transactions rolled back as the workload increases, which results in more overhead of the container. It can be seen from the prediction that the advantage of *RM* architecture diminishes as the ratio of read-only transactions decreases. *RM* architecture and *OCC* architecture still produce better performance than *CMP* architecture.

The model can also help to size the capacity of the system. Suppose that the requirement for the application response time is sub-1 second. We can estimate that the maximum workload of *CMP* to meet this requirement is  $N=400$ , and this degrades to  $N=380$  when the system has twice as many updates.

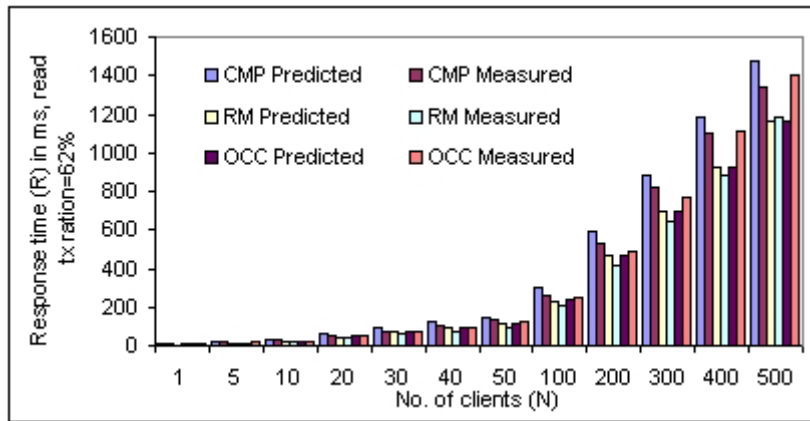
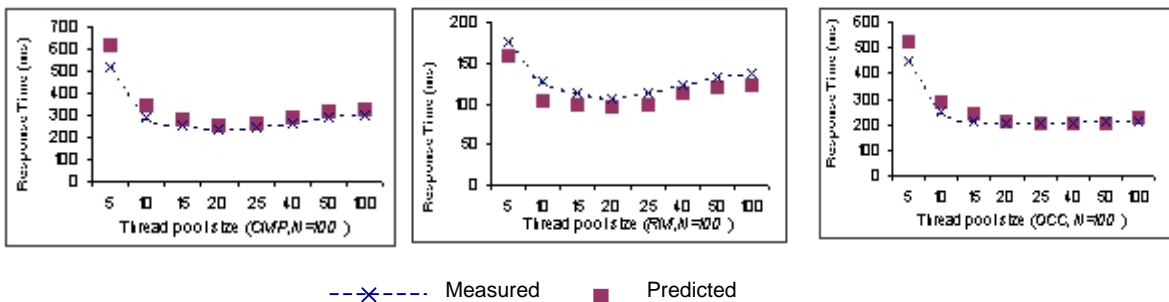


Figure 12 Stock-Online Performance (Doubled updates business model)

The utilization of each queue in the QNM is depicted in Figure 14. The prediction is that in the *CMP* and *OCC* architecture, the *Container* queue is the most demanding subsystem. It is the bottleneck software component as its utilization is approaching 100%. In the contrary, the *RM* architecture speeds up the container by optimizing the cache usage of read-only data. Thus the bottleneck is transferred to the *DataSource* queue whose utilization is saturated.

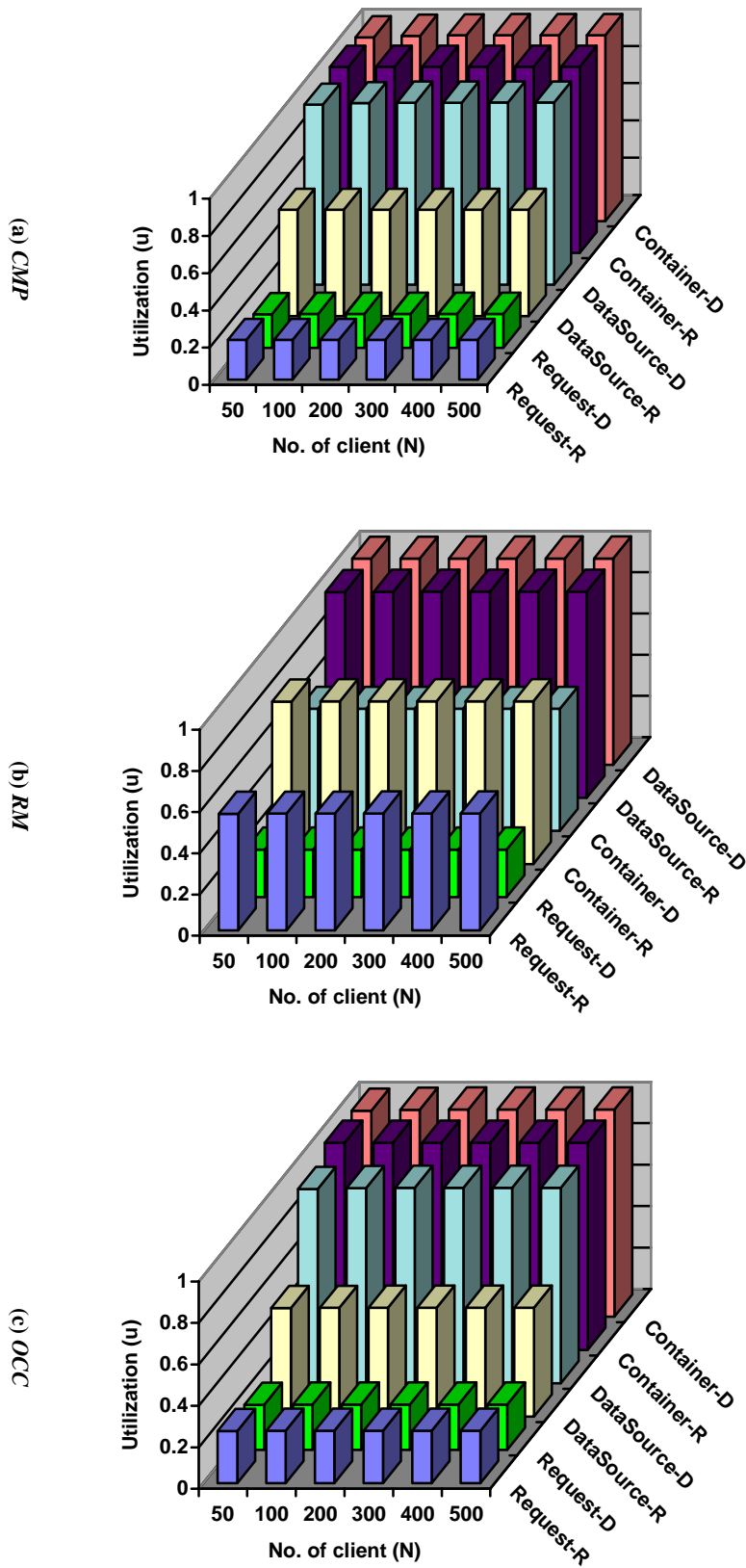
### 3.7.2 Tuning the thread pool size

The configuration of thread pool size represents an important tuning option. Too few threads will limit performance in the application server by serializing much of the application processing. Too many threads will consume resources and increase contention, again reducing application performance. Fortunately, the model can help us find the optimal configuration for the thread pool size. The model accurately predicts in Figure 13 that the optimal value is 20 for both *CMP* and *RM* and 25 for *OCC* instead of the default setting 15.



--x-- Measured    ■ Predicted

Figure 13 Stock-Online thread pool size tuning



**Figure 14 Predicted utilization of each queue in a single server.** For one architecture, the figure shows the utilization of each queue under two business models: *Read-only intensive* with the column named in the form of *Queue-R* and *Doubled updates* with the column named in the form of *Queue-D*

### 3.7.3 Two-node server clustering

Most middleware products support server clustering to provide scalability and availability. The application components are replicated on the servers within the cluster. We applied our approach to explore the performance of *CMP* architecture in a two-node server cluster. The *RM* architecture in a cluster environment requires more complicated infrastructure services to manage the consistency of beans replicas. The container is responsible for the invalidation of cached entity beans involved in conflicting transactions. The *RM* architecture model for a single server consequently has to be extended to represent this change in a clustered environment, and this is beyond the scope of this paper.

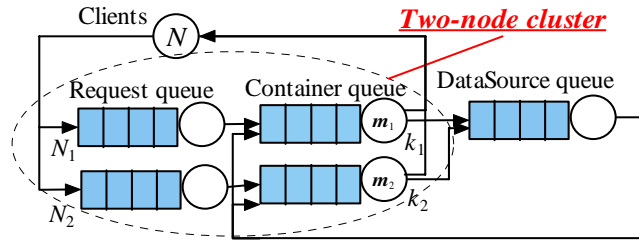


Figure 15 QNM of a two-node server cluster

We assume the two server machines are identical and they access a shared database server. The extended QNM is depicted in Figure 15. The workload is balanced between these two servers. The constraint of the client population of the *DataSource* queue is the total number of active database connections used by the two servers. We further assume that  $m_1=m_2$  and  $k_1=k_2$ . This model can be solved using Hierarchy Decomposition [12]. The server thread pool size is 20 and the parameters derived from a single server can be used to solve the model. The results are shown in Figure 16. The prediction error is 2-6% for various tested workloads. Using the model we can predict that adding an identical server, the maximum workload is  $N=660$  while meeting the requirement for a sub-second response time.

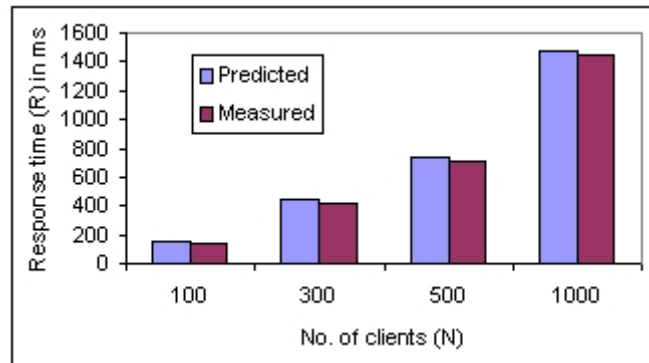


Figure 16 Stock-Online performance (two-node cluster)

The model can also produce the utilization of each queue in the QNM of a two-node cluster. It is clearly shows in Figure 17 that the utilization of two *Request* queues is similar to each other in the two application servers, which is also true for the utilization of the two *Container* queues. This is because the workload is balanced between these two servers and active number of server threads and database connections are almost identical. We can see that now the *DataSource* queue is about 100% utilized, which indicates it is the bottleneck software. Thus without needing more experiments, we can reason that adding extra nodes beyond 2 to the application server cluster would

not be useful. To further improve the performance, either the database could be transferred to a more powerful machine or more database servers should be clustered to provide service.

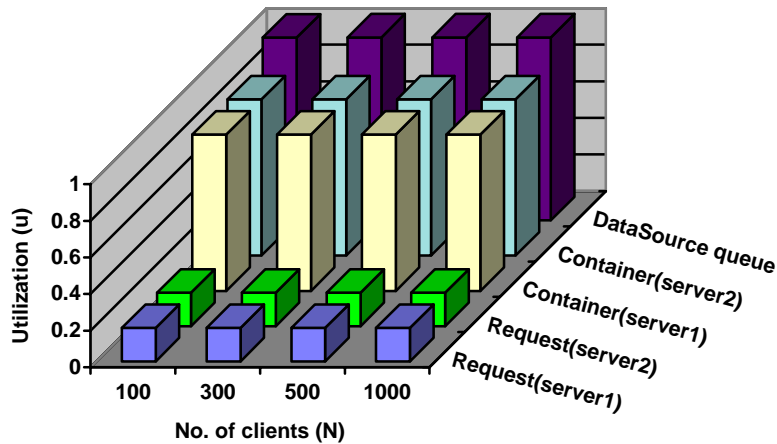


Figure 17 Predicted utilization of each queue in a cluster

## 4. Conclusion

In this paper we report a significant contribution to software developers who need to make sensible architectural choices during the design phase for a component-based, middleware-hosted application, in order to achieve desired performance goals. We have shown that one can predict eventual performance quite closely (within 10% on most measures for our example system). Our approach derives a quantitative performance model for the design, with parameters that reflect properties of the infrastructure platform. These parameters can be measured by running a simple benchmark application on the platform.

We have demonstrated our approach by predicting the performance of three competing designs for an application from the literature. The predictions were good enough to choose between the designs. The example design used EJB and J2EE technologies, and dealt with a CMP design, a read-mostly optimization design and optimistic concurrency control design. So far, we only model synchronous processes and we only derive the average response time of all classes of requests. The next step is to model asynchronous processes and to derive performance metrics of each class of requests. We also plan to investigate models that deal more precisely with the database server, which we have approximated as a simple queue. Case studies will be carried out on other middleware based applications. We are confident that the approach is general, and could be applied to other component technologies.

## 5. Acknowledgements

We would like to thank Software Architectures and Component Technologies (SACT) group in CSIRO Australia for providing the experiment environment for this work, especially Dr. Shiping Chen for his technical support.

## 6. References

- [1] Bachmann, F.; Bass, L.; Klein, M. Deriving architectural tactics, CMU/SEI-2003-TR-004, 2003.
- [2] Dilley, J.A.; Friedrich, R. J.; Jin, T. Y.; Rolia, J. Measurement tools and modeling techniques for evaluating web server performance, HPL-96-161, 1996.

- [3] Gooma, H.; Menascé, D.A. Design and performance modeling of component interconnection patterns for distributed software architectures, Proc. Workshop on Software and Performance (WOSP), 2000, pp.117-126.
- [4] Gorton, I. Enterprise Transaction Processing Systems, Addison-Wesley, 2000.
- [5] Gorton, I.; Liu, A. Performance evaluation of alternative component architectures for EJB applications, IEEE Internet Computing, vol.7, no. 3,2003, pp.18-23.
- [6] Gorton, I.; Liu, A.; Brebner, P. Rigorous evaluation of COTS middleware technology, IEEE Computer, vol. 36, no.3, 2003, pp. 50-55.
- [7] Harel, D.; Kugler, H.; Marelly, R.; Pnueli, A. Smart play-out, 18<sup>th</sup> Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 03), pp. 68-69, 2003.
- [8] Hissam, S., Moreno, G., Stafford, J., Wallman, K., Packaging predictable assembly, Component Deployment: IFIP/ACM Working Conference, LNCS 2370, 2002, pp 108-224.
- [9] Jacobson, P. A. and Lazowska, E. D., Analyzing queueing networks with simultaneous resource possession, Comm. of the ACM, vol 25, no. 2, 1982, pp.142-151.
- [10] Kounev, S., Buchmann, A., Performance modeling of distributed E-Business applications using queuing petri nets, Proc. of IEEE Int'l Symp on Performance Analysis of Systems and Software, 2003.
- [11] Kung, H. T. and Robinson, J. T. On Optimistic Methods for Concurrency Control, ACM Transactions on Database Systems, vol. 6, No. 2, 213 - 226, 1981.
- [12] Lazowska, E., Zahorjan, J., Graham, S., Sevcik, K., Quantitative System Performance, Prentice Hall, 1984.
- [13] Liu, T.K., Behroozi, A., Kumaran, S. A performance model for a business process integration middleware, IEEE Int'l Conf. on E-Commerce, 2003, pp. 191-198.
- [14] Menascé, D.; Almeida, V.A.F. Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning. Prentice-Hall, 2000.
- [15] Rolia, J. A., Sevik, K.C., 1995. The method of layers, IEEE Transaction on Software Engineering, vol. 21, no. 8. 1995, p.689-700.
- [16] Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S., 1995. The Stochastic Rendezvous Network Model for Performa of Synchronous Client-Server-Like Distributed Software, IEEE Transactions on Computers, vol. 44, no. 1, January, 1995, pp. 20-34.
- [17] Xu, J., Woodside, C. M., Petriu, D., Performance analysis of a software design using the UML profile for schedulability, performance and time, Proc. Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS), LNCS 2794, 2003, pp 291-310.