Performance Prediction of COTS Component-based Enterprise Applications

Shiping Chen, Ian Gorton, Anna Liu
Software Architectures and Component Technologies R&D Group, CSIRO CMIS,
Locked Bag 17, North Ryde,
Sydney 2113, Australia
+61 2 9325 3100
{shiping.chen, ian.Gorton, anna.liu}@cmis.csiro.au

Yan Liu
Basser Dept. of Computer Science
University of Sydney
Sydney 2006, Australia
+61 2 93518984
jennyliu@cs.usyd.edu.au

ABSTRACT

One of the major problems in building large-scale enterprise systems is anticipating the performance of the eventual solution before it has been built. This problem is especially germane to modern Internet-based e-business applications, where failure to provide high performance and scalability can lead to application and business failure. The fundamental software engineering problem is compounded by many factors, including application diversity, architectural trade-offs and options, COTS component integration requirements, and differences in performance of various software and hardware infrastructures. This paper investigates the feasibility of providing a novel and practical solution to this problem. The approach as demonstrated, constructs useful models that act as predictors of the performance for component-based systems hosted by middleware infrastructures such as CORBA, COM+ and J2EE.

1. INTRODUCTION

The software industry has realized that robust, reliable and scalable technology is needed to support their enterprise-scale, e-business systems. Middleware and component technologies [1], the *plumbing* of many Internet systems, have emerged in various guises as a base infrastructure for running advanced e-business systems. The predominant ones in use today are CORBA from the Object Management Group, COM+ from Microsoft and J2EE from Sun Microsystems.

This paper explains the inherent difficulties in predicting early in a project lifecycle the performance of applications built using COTS middleware components. In order to narrow the scope of the problem, the focus is on the large class of 4-tier enterprise applications that typically comprise:

1. Web browser based clients
2. A Web Server executing presentation logic
3. An Application Server executing business logic
4. One or more databases and back-end applications that provide data storage

We present a novel approach to this problem of performance prediction. The approach comprises 3 elements:

1. a structured approach to gathering empirical performance results on COTS middleware infrastructures
2. a reasoning framework for understanding architectural trade-offs and relationships to technology features
3. a set of predictive mathematical models that describe the generic behavior of applications using COTS middleware technology

The empirical results established through testing in step 1 are used to feed into COTS product specific parameter values for the models. This then enables us to make predictions based on observed and measured product-specific behaviors. In this paper, we will focus on elements 1 and 3 only. We will also conclude by describing the current status of the project and the outstanding problems that remain to be solved.

2. PERFORMANCE OF COMPONENT-BASED SYSTEMS

The discussion that follows is aimed at component based software infrastructure technologies such as COM+ and J2EE. Very basically, these component architectures provide run-time environments that provide application level components with the many services they require to operate in a distributed system.

These services typically include object location, security, transaction management, integration services, database connection pooling, and so on. The overall aim is to free application level components of the need to manipulate these services directly in their code, hence making them simpler to build and maintain. The component run-time environments, typically called *containers*, provide these services to the application components they host. Individual components are able to request specific levels of service from a container (eg no security, encryption, etc) declaratively, using some form of configuration information that the container reads.

There is an important distinction to be made about the type of components that make up applications built using middleware. As we've already pointed out, COTS middleware components form the infrastructure, or *plumbing* of distributed applications. In effect this infrastructure provides a distributed environment for deploying *application level components* that carry out business-specific processing.

This distinction between infrastructure level components and application level components is crucial. Application level components rely on the COTS middleware-supplied infrastructure components to manage their lifecycle and execution, and to provide them off-the-shelf services such as transactions and security. Hence, an application level component cannot execute outside of a suitable COTS middleware environment. The two are extremely tightly coupled.

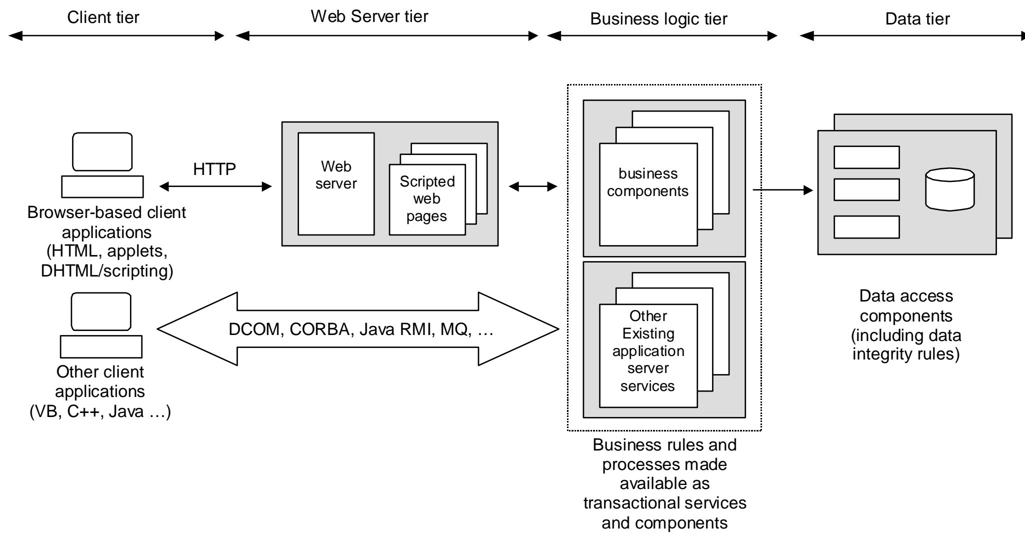


Figure 1 Typical 4-tier Enterprise Application Architecture

An important implication of this is that the behavior of application components is completely dependent upon the behavior of the infrastructure components. The two cannot be divorced in any meaningful way. The entire application's behavior is the combination of the behavior of the application – the business logic - and infrastructure components as depicted in Figure 2.

All this has profound implications upon component assessment, certification and software engineering. No matter how high the quality of the application components, the COTS middleware infrastructure becomes the most crucial component in most systems. If the COTS middleware is naively architected or implemented, has subtle errors in some services, or is simply inefficient and lacking in features, then the application components inevitably pay the price.

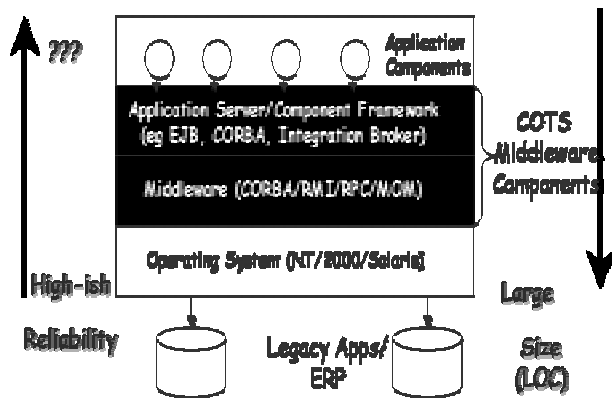


Figure 2 Anatomy of a COTS Middleware Application

Interestingly, open standard COTS middleware infrastructures such as CORBA and J2EE actually exacerbate this problem. With CORBA and J2EE technologies, many vendors sell their own versions of the middleware infrastructure. These are all implemented differently, and hence behave and perform differently [2]. This means, for example, a J2EE-based application component's performance is dependent upon the actual J2EE product that it runs on. The same application component may perform very differently indeed on two different J2EE implementations [2], depending on the quality and features of the product. This of course is not the case with single-source component technologies such as Microsoft's COM+.

This tight coupling of application and infrastructure invalidates traditional approaches to application performance measurement, and makes prediction of the effects of various architectural trade-offs complex. It is no longer possible to execute the application components independently and measure their performance. Nor is it possible to inspect the code in an attempt to analyze performance, as the infrastructure source code is rarely available.[1]

This last point is absolutely crucial. Approaches such as that exemplified in [3] are naïve. They assume an idealized implementation of a component infrastructure technology such as Enterprise Java Beans (EJB), and base their performance models on this. Unfortunately, no such idealized implementation exists. CSIRO's MTE project clearly demonstrates the performance differences between different EJB products.

In fact, a trivial test case can demonstrate why the queuing model proposed in [3] not valid for a real technology. Figure 3 illustrates the results of executing identical application components on six different COTS components infrastructures based on the J2EE open standard. All the tests are executed on the same physical hardware and software environment, and the products are configured to achieve optimal performance on the test hardware available.

The graph shows the application throughput achieved in terms of transactions per second (tps) for client loads varying between 100 and 1000. The performance differences are significant, both in the peak throughput achieved for each COTS technology, and their ability to scale to handle increasing client loads. These differences become more even more significant as the same test case is scaled out to run on more application server machines in an attempt to improve application throughput [2].

[1] Even if it were, the complexity of the infrastructure code would make this infeasible.

It therefore should be quite apparent that any performance prediction approach that does not take in to account differences between actual products is doomed to failure.

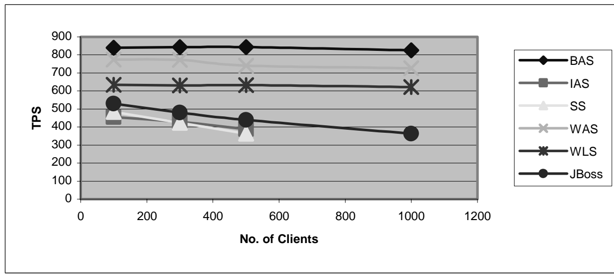


Figure 3 J2EE Performance Comparisons

Given all these issues, software engineers in practice must resort to experimentally discovering application configurations that provide acceptable levels of performance. This can be a time-consuming and expensive process, as it requires detailed measurements to be recorded from test runs across a number of different configurations. As COTS middleware technologies are highly configurable, often with tens of inter-related configuration options, this is rarely a trivial performance tuning exercise.

3. PERFORMANCE PREDICTION METHODOLOGY

As depicted in Figure 4, the performance prediction methodology has two aims.

The first is to create a COTS product-specific performance profile that describes how the various components of the middleware product affect performance. This profile is aimed at analyzing the behavior and performance of a middleware product in a generic manner that is not related to any particular application requirements. Using this profile, it is possible to use a set of generic mathematical models to predict the behavior of the middleware infrastructure under various configurations.

The second aim is to construct a reasoning framework for understanding architectural trade-offs and their relationships to specific technology features. This reasoning framework provides the architect with insights into how the different quality attributes interact with each other, and it helps the architect reason about the effects of their architectural decisions.

The third and final aim is to create an application-specific configuration that takes in to account the behavioral characteristics of the application at hand. The application architect describes the application behavior in terms of client loads, business logic complexity, transaction mix, database requirements, and so on. By inputting these parameters in to the generic performance models, it should be possible to predict the application configuration settings required to achieve high performance.

The remainder of this paper focuses on the first and third step of this methodology, and the design of a set of test cases that can be used to characterize the performance of a COTS middleware technology. In particular, we use a J2EE/EJB application server technology as an example.

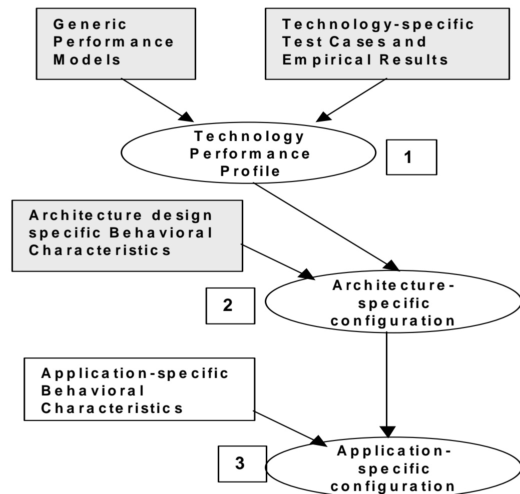


Figure 4 Performance Prediction Methodology Outline

Central to the J2EE specification is the Enterprise JavaBeans framework. EJBs are server-side components, written in Java, that typically execute the application business logic in an N-tier application. An EJB container is required to execute EJB components. The container provides EJBs with a set of ready to use services including security, transactions and object persistence. Importantly, EJBs call on these services declaratively by specifying the level of service they require in an associated XML file known as a deployment descriptor. This means that EJBs do not need to contain explicit code to handle infrastructure issues such as transactions and security.

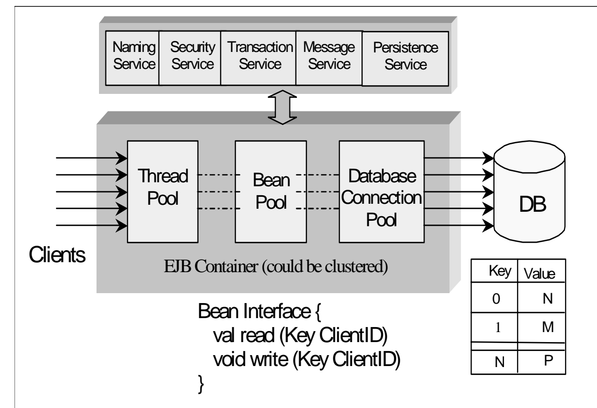


Figure 5 J2EE/EJB Test Case Design

An EJB container also provides internal mechanisms for managing the concurrent execution of multiple EJBs in an efficient manner. EJBs themselves are not allowed to explicitly manage concurrency, and hence must rely on the container for efficient threading and resource usage.

3.1 A Bottom-up Testing Approach

In order to examine the infrastructure component quantitatively, and independently from the application behaviour, we use a simple test application, known as the *identity* application. The identity application has several important characteristics that make it an good test application for examining infrastructure quality. The identity application has the following characteristics:

- There is only one table in the relational database
- The table contains 2 fields only: the unique identifier (or key) and a value field that contains a number
- A single application component (e.g. an EJBean) with a read and write method in its interface
- The read(Key ClientID) method simply reads in the number in the value field, given the identifier
- The write(Key ClientID) method increments the value field given the identifier
- Each client using the server-side identity application business logic has a unique id

Using the identity application, we can thus remove any unpredictability in timing due to database contention and application and or business logic complexities. In the absence of database bottlenecks, we can observe the COTS middleware components behaviour and quality. The insight into the infrastructure component assists with building the prediction model.

Initially, tests aim to exercise the basic infrastructure of the COTS components using the identity application. Additional service components such as transaction and or security service components can then be introduced in to the tests. The multi-staged testing approach enables us to observe the differences between a basic system (with no additional services) and another that utilizes an additional service component.

Parameter Types	**Sample Test Parameters**
External variable stimulus	Client request load, transaction types, transaction request frequency
Configurable System Parameter	Thread pool size, database connection pool size, application cache size,
Measurable/Observable parameters	Throughput (transactions per sec) Client response time, service time
Deducible System Property	Optimal thread pool size for achieving maximum throughput, optimal database connection pool size for achieving maximum throughput, optimal application component cache size for achieving minimal response time

Table 1 Parameter Types and Sample Test Cases

4. CASE STUDY

In order to demonstrate the viability of this approach, this section describes a cost model that estimates the overhead of contention in a generic server component interacting with a database.

The presentation of this case study follows the performance prediction process:

1. Construct a mathematical model that captures generic behavioral properties, and uses coefficients to abstract away the constant environmental factors that holds true for a given test environment
2. The parameter values for the models are 'discovered' through empirical testing of specific COTS technologies
3. Once the parameter values have been populated, and coefficients representing environmental factors have been determined, the model is used in a 'predictive' fashion, to help the system architect in making important design decisions early in the project lifecycle

4.1 Test Setup and Model

For simplicity, we consider only one server, which creates a set of threads to serve requests concurrently. Figure 6 illustrates the process of handling concurrent requests in the server.

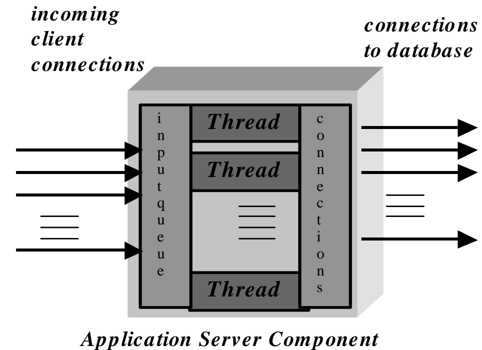


Figure 6 Handling Concurrent Requests

As shown in Figure 6, the overall contention overhead for a server component basically results from three sources. First, contention occurs when all concurrent requests compete for service by the server component. This contention includes the network bandwidth and underlying transport mechanisms, typically socket ports in TCP/IP-based protocols. Second, contention arises when all accepted requests compete for a thread to execute the server component's business logic. Finally, contention results from concurrent access to the database by the concurrent server threads. `This involves database connection handling and locks on data in the database.

If we represent the number of clients as x and the number of server threads as y, we obtain the following contention cost model:

$$C = ax + \frac{bx}{y} + cy \qquad (1)$$

Where:

- C *is the overall contention overhead in time;*
- a *is the network contention overhead ratio per concurrent client;*
- b *is the contention overhead ratio per concurrent client for service from a single server thread;*
- c *is the contention overhead ratio per server thread to access database;*

Note that in this contention prediction model, x is the external variable stimulus, y is the configurable system parameter, coefficients of x and y, i.e. a, b and c are abstractions of the environmental variables for a given test environment. To make the contention overhead independent from testing execution times and transaction types, we define the contention overhead *C* as the average response time for all transactions, *i.e. C=T/N,* where *T* is the execution time of a test and *N* is the total number of transactions.

Parameter Types	**Test Parameters**

External variable stimulus	Number of clients
Configurable System Parameter	System thread pool size
Measurable/Observable parameters	Throughput, response time
Deducible System Property	Contention overhead ratio per server thread to access database

Table 2 Test Case Parameters for Deducing System Contention

The contention experienced for a server component is of course application and hardware dependent. Many factors may contribute to or reduce the overheads of contention, such as the transaction types, machine capacity, network traffic, and so on. This makes it impractical or impossible to build a 'perfectly precise' cost model that takes all factors into account. While our cost model focuses on two contention factors, namely the number of concurrent requests and the number of server threads (*x* and *y*), all the other factors are absorbed in the model parameters (*a, b* and *c*). With our contention cost model (1), it is consequently possible to derive the optimal number of server threads. That is:

$$y^* = \sqrt{\frac{bx}{c}} \qquad (2)$$

where x is the given number of clients; b and c are the model parameters reflecting the characters of a specific application and platform.

This result can be further explained as follows:

- The more concurrent clients result in higher contention at all points. The degree of the concurrency of a server component is proportional to the square root of the number of concurrent clients.
- Increased contention on the database in practise restricts a server component from using an unbounded number of concurrent database connections.

4.2 Empirical Testing and Parameter

For a given test application using a commercially available application server technology, a number of tests were run. Each fixed the number of server threads, e.g. y = 1, 2, 4 and so on. Then, we ran a varied number of test clients for different tests runs against the server component. Each client executes 430 transactions in the test [2]. From the test results, the contention overheads were derived from the execution times measured at the client-side by using C=T/N, where T is the execution time in milliseconds and N = 430. In this way, we obtained the first measure of contention overheads as shown in Table 3.

From Table 3, we can observe that while the contention overheads increase as the number of concurrent clients increase, the contention overheads change in a parabola shape as the number of server threads increase monotonically. This verifies the assumption that too high a degree of concurrency in a server component will degrade the overall performance. Therefore, an optimal degree of concurrency for this application exists, and the engineering problem is to discover this and achieve high performance.

Num. of	Num. of Concurrent Clients			
Server Threads	100	200	400	800
1	1251	2447	5216	10405
2	841	1605	3096	6385
4	730	1443	2966	5910
8	724	1427	2945	5913
16	737	1435	2894	5773
32	760	1574	2974	5814
64	782	1579	3158	6178
128	813	1629	3389	6849

Table 3 Contention overheads in milliseconds

The statistical software package, *Splus* [5], was then used to fit our non-linear cost model. Based on the experimental results in Table 3, we obtained the following parameters for cost model (1) above, for the test application implemented in the application server technology used in the tests:

a = 6.65 msec / client

b = 5.42 msec · thread / client

c = 5.24 msec / thread

Then the cost model with the three parameters was used to estimate contention overheads, which are compared with the experimental results in Figure 7. The graphs show the cost model represents the pattern and trend of the contention overhead and thus can be used to predict the degrees of the concurrency in the server components.

4.3 Deriving Optimal Concurrency

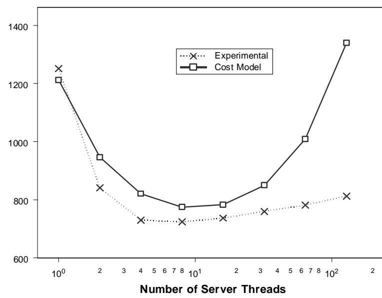
Based on the above parameters, we can derive the optimal number of server threads by applying (2). A set of theoretical optimal numbers of server threads for different number of concurrent clients is listed in Table 4. As shown in Figure 7, basically, the theoretical optimal numbers of server threads falls in the area where the best performance could be achieved.

Num of Clients (*x*)	Optimal Num. of Server Threads (y^*)
100	10
200	14
400	20
800	29

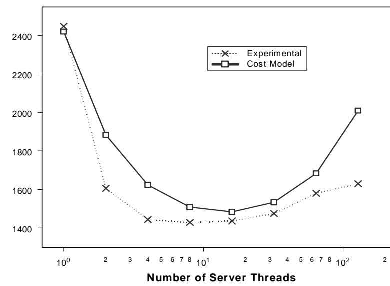
Table 4 Theoretical optimal numbers of server threads

5. CONCLUSIONS

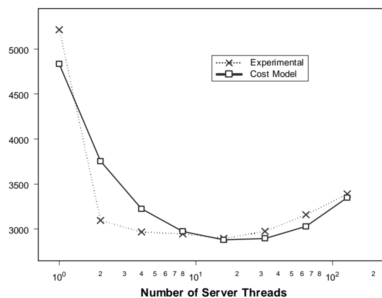
The approach explained in this paper explicitly recognizes the inherent problems of component based systems, and provides a possible solution based on empirical testing and mathematical modeling. The models describe generic behaviors of application server components running on COTS middleware technologies. The parameter values in the models are necessarily different for each different product, as all products have unique performance and behavioral characteristics. These values must therefore be discovered through empirical testing. To this end, a set of test cases are defined and executed for each different COTS middleware product. The results of these tests make it possible to solve the models for each product, so that performance prediction for a given product becomes possible.



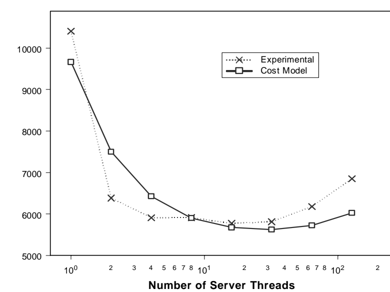
(a) 100 Clients



(b) 200 Clients



(c) 400 Clients



(d) 800 Clients

Figure 7 Experimental Results vs. Cost Models

This approach has been validated in two case studies, one of which is described in the paper. The execution of the complete set of test cases to characterize two different J2EE products is close to completion, and the results are being used to populate a number of generic models.

In the medium term, there remain a number of complex problems to solve. Incorporating application-specific behavior in to the equation in a practical manner is an open problem. This is necessary to make the approach useable in wide-scale engineering. It also remains to be seen how far the results from the empirical testing can be generalized across different hardware and software platforms. This is important, as it profoundly affects the cost of executing the test cases. It may be that, if a huge number of test cases need to be executed on different platforms, this approach may be economically impractical in an environment of rapid change and evolution.

6. REFERENCES

[1] Wolfgang Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000. ISBN: 0-471-98657-7.

[2] Ian Gorton. *Middleware Technology Evaluation Series.* CSIRO, Australia, 2000. www.cmis.csiro.edu.au/adsat.html

[3] Catalina M. Liado, Peter G.Harrison. *Performance Evaluation of an Enterprise JavaBean Server Implementation*. Proceedings on the second international workshop on software and performance, September 17-20, 2000, Ottawa, Canada.

[4] Len Bass, Paul Clements, Rick Kazman. Software Architecture in Practice. Addison-Wesley, 1998. ISBN: 0201199300

[5] Longhow Lam. *An Introduction to S-PLUS for windows. CANdiensten*, 1999. ISBN 90-804652-2-4

[6] Shiping Chen and Jinling Xue. *Partitioning and scheduling loops on NOWs*. Journal of Computer Communications, 22(11): 1017-1033. Elsevier, Netherlands, July 1999.

[7] Dino. P. Chiesa. *Tuning Encina Applications: A Practical Guide.* IBM white paper, www.transarc.ibm.com/Support/encina/white_papers/Index.html

[8] Irfan Pyarali et al. *Optimizing Thread-Pool Strategies for Real-Time CORBA*, Procs of ACM SIGPLAN Workshop on Optimization of Middleware and Distributed System (OM 2001), Utah, June 2001.

[9] Ian Gorton. *Enterprise Transaction Processing Systems: Putting the CORBA OTS, Encina++ and OrbixOTM to work.* Addison-Wesley, 2000. ISBN: 0201398591

[10] Grundy, J.C. and Liu, A. *Directions in Engineering Non-Functional Requirement Compliant Middleware Applications*, In Procs of the 3rd Australasian Workshop on Software and Systems Architectures, Australia, Nov 2000, Monash Univ.

[11] Anna Liu. *An Approach for Constructing High Performance, Scalable Distributed Object Systems*, Proceedings of International Conference on Software Engineering, 4-11 June, 2000, Limerick, Ireland.

[12] John Grundy, Yuhong Cai, Anna Liu. *Generation of Distributed System test-beds from High-Level Software Architecture Descriptions*, in Proceedings of 15th IEEE International Conference on Automated Software Engineering, 2001, San Diego USA.