NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

OPTIMIZATION OF DISTRIBUTED, OBJECT-ORIENTED ARCHITECTURES

by

William J. Ray

September 2001

Dissertation Supervisor: Dr. Valdis Berzins

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** September 2001	**3. REPORT TYPE AND DATES COVERED** Dissertation	
4. TITLE AND SUBTITLE: Optimization of Distributed, Object-Oriented Architectures			**5. FUNDING NUMBERS**
6. AUTHOR(S) William J. Ray			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

Object-Oriented computing is fast becoming the de-facto standard for software development. Optimal deployment strategies for object servers change given variations in object servers, client applications, operational missions, hardware modifications, and various other changes to the environment.

Once distributed object servers become more prevalent, there will be a need to optimize the deployment of object servers to best serve the end user's changing needs. Having a system that automatically generates object server deployment strategies would allow users to take full advantage of their network of computers.

Many systems have very predictable points in time where the usage of a network changes. These systems are usually characterized by shift changes where the manning and functions preformed change from shift to shift. We propose a pro-active optimization approach that uses predictable indicators like season, mission, and other foreseeable periodic events.

The proposed method profiles object servers, client applications, user inputs and network resources. These profiles determine a system of equations that is solved to produce an optimal deployment strategy for the predicted upcoming usage by the users of the system of computers and servers.

14. SUBJECT TERMS Object-Oriented Programming, Stochastic Optimization, Distributed Computing, Load Balancing, Performance Tuning			**15. NUMBER OF PAGES** 327
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

OPTIMIZATION OF DISTRIBUTED, OBJECT-ORIENTED ARCHITECTURES

William J. Ray
B.S., Purdue University, 1985
M.S., Naval Postgraduate School, 1997

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2001

Author: _____
William J. Ray

Approved by: _____
Dr. Valdis Berzins
Prof., Software Engineering
Dissertation Supervisor, Chairman of Doctoral Committee

_____
Dr. Luqi
Prof., Software Engineering

_____
Dr. Dan Boger
Prof., C4I Department

_____
Dr. Geoffrey Xie
Assist. Prof., Computer Science

_____
Dr. Marvin Langston
SAIC Corporate Development

Approved by: _____
Dr. Luqi, Chair, Department of Software Engineering

Approved by: _____
Carson K. Eoyang, Associate Provost for Academic Affairs

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Object-Oriented computing is fast becoming the de-facto standard for software development. Optimal deployment strategies for object servers change given variations in object servers, client applications, operational missions, hardware modifications, and various other changes to the environment.

Once distributed object servers become more prevalent, there will be a need to optimize the deployment of object servers to best serve the end user's changing needs. Having a system that automatically generates object server deployment strategies would allow users to take full advantage of their network of computers.

Many systems have very predictable points in time where the usage of a network changes. These systems are usually characterized by shift changes where the manning and functions preformed change from shift to shift. We propose a pro-active optimization approach that uses predictable indicators like season, mission, and other foreseeable periodic events.

The proposed method profiles object servers, client applications, user inputs and network resources. These profiles determine a system of equations that is solved to produce an optimal deployment strategy for the predicted upcoming usage by the users of the system of computers and servers.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

ACKNOWLEDGMENTS

The author wishes to acknowledge the Office of Naval Research, ONR, and the Defense Advanced Research Projects Agency, DARPA, for their partial funding of this research.

Office of Naval Research

Code 01A

800 N. Quincy Street

Arlington, VA 22217-5660

Defense Advanced Research Projects Agency

Information Systems Office

3701 N. Fairfax Drive

Arlington, VA 22203-1714

EXECUTIVE SUMMARY

Object-Oriented computing is fast becoming the de-facto standard for software development. Optimal deployment strategies for object servers change given variations in object servers, client applications, operational missions, hardware modifications, and various other changes to the environment.

Once distributed object servers become more prevalent, there will be a need to optimize the deployment of object servers to best serve the end user's changing needs. Having a system that automatically generates object server deployment strategies would allow users to take full advantage of their network of computers.

Many systems have very predictable points in time where the usage of a network changes. These systems are usually characterized by shift changes where the manning and functions preformed change from shift to shift. We propose a pro-active optimization approach that uses predictable indicators like season, mission, and other foreseeable periodic events.

The proposed method profiles object servers, client applications, user inputs and network resources. These profiles determine a system of equations that is solved to

produce an optimal deployment strategy for the predicted upcoming usage by the users of the system of computers and servers.

I. CHAPTER 1

A. INTRODUCTION

Complex computer systems are made up of computers, the networks that connect these computers together, the software that runs on these computers and the users that interact with the applications. The systems tend to be heterogeneous in the hardware and software that comprise their structure. The functions these systems support are diverse as well.

System engineers always want these computer systems to perform at peek efficiency. However, with the constantly changing environment that characterizes these systems, peek efficiency is difficult to maintain.

When these systems serve a set of users that is known and limited, then the possibility of matching the system to the changing environment is achieved. By knowing ahead of time that a limited number of users can access the system, assumptions can be made about queuing delay that make reasoning about this environment possible. Even a simplistic model of this environment can lead to large gains in performance.

To prove this hypothesis, I introduce a methodology for implementing a model of a distributed, object-oriented system with a known set of users on a heterogeneous

environment of hardware. Different scenario reflecting different manning schedules, hardware and software changes where input into the model. The results of these model runs where different deployment assignments for the object servers.

These scenarios where then tested with real software on real hardware in a test environment. Measurements of all possible deployments where collected and compared. The results showed that substantial performance enhancements could be achieved by this approach.

The advancements of object-oriented technology in the past decade have lead to worldwide acceptance of its principles. Today, numerous developers design their systems by modeling the problem domain in terms of communicating entities called objects. Object-oriented systems tend to be more intuitive, easier to maintain, and allow for more re-usable code.

The future of computing is heading for a universe of distributed object servers. The evolution of object servers to distributed object servers will parallel the evolution of the relational databases. Over time, object servers will provide functionality to more client applications than their original applications, just as relational databases were used by more applications than the original application. In both cases, systems optimized for

the original application may not perform well for the new applications. Tools that allow a programmer to model an object and create object servers with all the necessary infrastructure code needed to work as a distributed object server will soon be available. This will lead to an explosion in the number of object servers available to client applications.

A user's network of computers will be in a constantly changing state. Object servers, applications, hardware and user preferences will be in a constant state of flux. No static deployment strategy can adequately take advantage of the assets accessible on the network in such a frequently changing environment. In many cases there exist predictable points in time where the user will know how their network of computers will change. These predictable points in time are usually scheduled. By allowing the user to take advantage of these scheduled changes, the system can be better utilized.

No system can accurately predict user interaction with a system. Two separate users performing the same job will interact with a system differently. The same user may interact differently while performing the same job. For these reason and combinatorial explosion problems, a more dynamic software engineering approach must be taken instead of a static computer science approach. The alternative is a

deployment strategy that is dictated by the system engineer's view of how the system will be utilized. Of course, the system engineer doesn't revisit this strategy every time hardware, software or user interactions change. The goal is to make better deployment choices without the need for a system engineer, since many of these changes will take place without the knowledge of a system engineer or the budget to employ one.

1. Object-Oriented Architectures

Object-oriented systems can be single tier architectures, where the entire system is contained inside of a single class in one executable. They can have n-tier architectures where all tiers execute on a single machine or in a single executable. These two types of architectures are quite easy to develop and deploy. Another architectural type is that of a distributed object-oriented architecture. In this architecture, the first tier usually consists of at least one object server and the remaining tiers consist of at least one application. When the object server and application do not have to be co-located on the same machine, then the architecture is a distributed object-oriented architecture. Distributed, object-oriented architectures are more difficult to develop and deploy, but this architecture is much more applicable to the design of large, complex systems. The model defined in this paper

must be able to reason about distributed, n-tier architectures.

2. Object-Oriented Middleware

There are three primary communication conduits for objects today. The Common Object Request Broker Architecture (CORBA) is the Object Management Group's (OMG) core specification for distributed object interoperability. The protocol used to communicate in CORBA is Internet Inter-ORB Protocol (IIOP). Sun Microsystems's JAVA Remote Method Invocation (RMI) is another protocol commonly used to communicate between distributed objects. The third protocol is Microsoft's Component Object Model (COM) and its derivatives COM, DCOM and COM+.

All of these middlewares offer different advantages and disadvantages, but they are more alike than they are different. A CORBA server, an Enterprise Java Bean (EJB) container, and a COM+ server are all object servers. The methodologies for deploying object servers in this paper will work for all three of these different middlewares.

3. Object-Oriented Languages

There exist many languages to implement object-oriented systems. These include, but are not limited to Smalltalk, C++, Ada95, and JAVA. Although, all the programming in this paper was done in JAVA, the methodologies developed in this

dissertation will work for object servers implemented in different languages.

There are problems associated with redeploying object servers written in different languages on machines with different operating systems. Not all languages are available on all machines. Specialized languages for particular machines will have limited mobility. These restrictions must be reflected in the model.

4. Optimization

The goal of this research is to optimize a distributed, object-oriented architecture to suit users needs. The criterion used to evaluate the optimizations is user response time. By changing the assignment of object servers to different machines with different capabilities, the end result will be a change in the response time to the user. Dynamic optimizations can incur large overheads in computation that can ultimately eat into any savings. I propose a system that re-computes static deployment patterns for predictable points in time to better utilize the targeted system.

5. Predictable Points in Time

How can the user recognize the predictable points in time at which deployment patterns should be re-assessed? Hardware and software changes are usually scheduled for systems in advance. Hence, these changes come at

predictable points in time. These are the most easily recognizable and exist for almost every system. Any tool to optimize a system would have to take into account these changes to the system.

The real power to the optimization detailed here lies in knowing your users. To take full advantage of the methodology detailed in this paper, one needs to model the users of the targeted system. In systems that serve a known universe of users, this methodology can be extremely helpful. By modeling all the different types of users on the targeted system, an optimizing methodology can take advantage of shift schedules, manning changes, mission changes, and other changes in users or their demand patterns to reconfigure for better-targeted performance.

This chapter gave a brief introduction to the problem and the motivation for the research. Chapter II gives an assessment of previous research. Chapter III details the data needed to model the environment. Chapter IV introduces the objective function and illustrates how to map the profiles to the objective function. Chapter V illustrates the different parts of the object function. Chapter VI contains the results of a JAVA RMI implementation. Chapter VII contains the results of a JAVA CORBA implementation. Chapter VIII discusses methodologies for collecting the needed profiles. Chapter IX discusses future refinement to

the model. Chapter X is a collection of conclusions that can be drawn from this research. Appendix A is a list of all the LINGO models that where used in this dissertation. Appendix B is the JAVA RMI code. Appendix C is a listing of the raw data collected from the test environment. Appendix D is a list of LINGO models used for combinatorial timing tests. Appendix E is a listing of the JAVA CORBA code.

II. CHAPTER 2

A. PREVIOUS WORK ASSESSMENT

Distributed, Object-Oriented technology is a relatively new technology. Many of the common tools used to develop distributed object servers haven't been around for many years. CORBA, JAVA, and COM have been around for almost a decade, but actual deployed systems where object servers are involved in the architecture are just now becoming commonplace. Most of the distributed, object-oriented research to date is in the area of making these architectures easier to develop, more reliable, and increasing the performance of the implementing languages and middleware.

There has been little work on deployment strategies for distributed object servers. The closest relevant research is in the fields of load balancing and client/server performance. Relevant work today also exists in the automated generation of object server code.

1. Load Balancing

State of the art load balancing techniques address scheduling of given tasks on a set of given machines. The goal of these techniques is to balance the load across multiple machines. While many of the ideas and terminology

are useful for optimization, the essential goals differ. In optimization, the goal is to decrease the response time to the user. In given situations, this may require having the entire load run on just one computer. Figure 1 depicts the processing that most load balancing research covers where tasks have independence with respect to the location of its execution [1,2,3,4,5,6,7,8].

It is important to remember that object servers do not have independent tasks. All methods in an object are related because they reference the same data. It is often useful to manage all the objects of a given type with a single object manager. Figure 2 represents this fact. While load-balancing research doesn't require independence of tasks, the dependencies that are of interest usually deal with temporal constraints. Tasks of this nature may require that a given task finish executing prior to the start of the related task. Tasks in an object are constrained by locality of data. When duplicate objects are not allowed, it is difficult to spread tasks contained inside of the same object across multiple machines.

Types can be partitioned with some instances in one server and others in a different server. This can work if none of the methods reference more than one instance of a type. The methodology detailed in this dissertation requires all instances of the same type to be in one server.

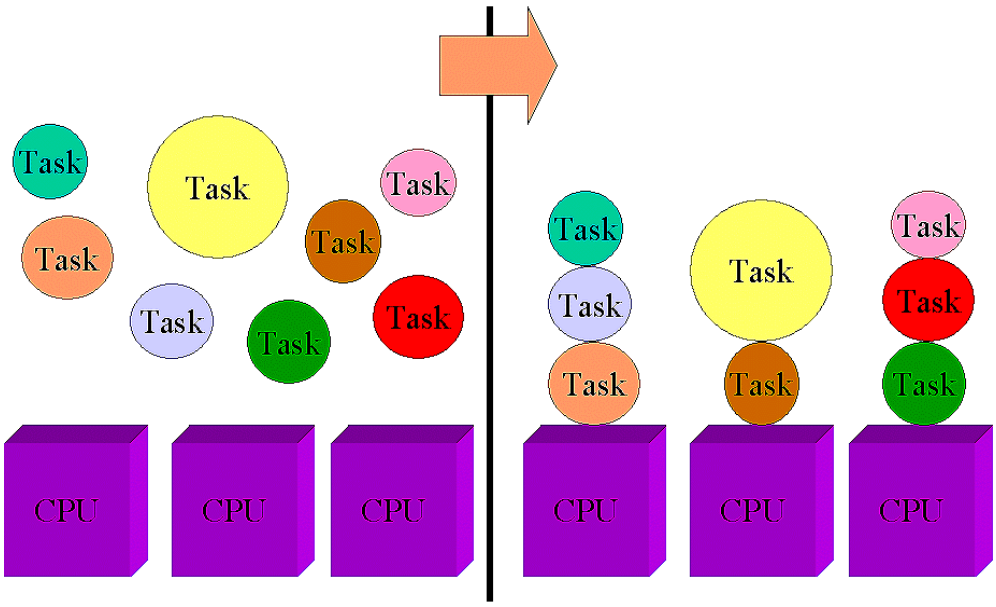

Figure 1: Current Load Balancing Techniques.

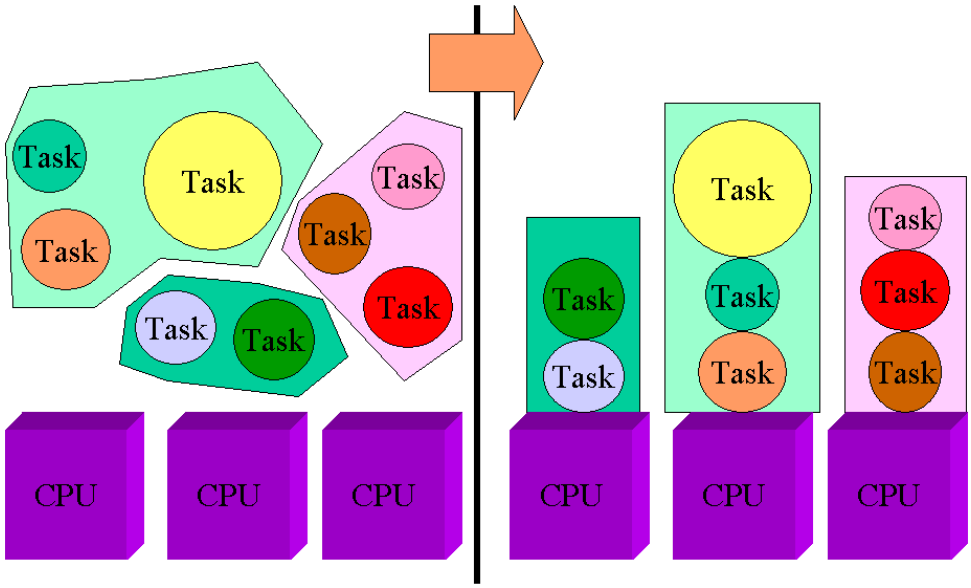

Figure 2: Load Balancing Object Servers.

2. Parallel Processing

Parallel processing research usually looks at changing a sequential computation into a parallel computation to gain speed. This involves breaking down a computation into n parts that can be processed simultaneously on different computers. Finding these parallel components is both difficult and time consuming. The potential benefits of this approach seem limited to complex calculations and large searches. These optimizations would have more than likely have an impact on the internals of the server or methods, than on a methodology for optimizing multiple servers that may or may not be related.

3. Client/Server Performance

In this research, the processing is distributed in a very different way than the way parallel processing research distributes the load. The server functionality usually handles the processing for keeping the data safe. It usually handles functions like persistence, concurrency issues, security, etc. The client code often handles the processing dealing with the graphical user interface.

The number of times a method is called is usually dependent on the interaction with end users, very much like the situation in client/server performance research. Most of the research in this area looks at the internals of the single server relationship with its clients. The caching of

information and use of proxies are examples of performance research in this area [15,19,23,24,25,27]. While some of the ideas in client/server research can be used to optimize object server deployments, the research involved in this paper is targeted at performance to the clients when multiple servers are involved. The internals of the servers are treated as black boxes where no knowledge of the inner workings of the servers is required.

4. Clusters and Replicas

Other approaches to decreasing the average client response time include the use of replicas or clusters. These techniques usually involve making copies of servers and distributing these copies across machines. The optimizations then look at balancing requests across the copies [3,8]. Many commercial middleware and database products use these techniques.

Clusters have no logic internal to the servers to guarantee that the copies remain consistent. Without consistent copies, clients can get different results from the same query. For this reason, many of the commercial products require the objects in the servers to be stateless objects. This means that the objects are essentially read only and the values of the attributes cannot be modified. Even with stateless objects, the addition and deletion of object instances can lead to servers with differences.

Replicas have logic internal to the servers to guarantee that the copies remain consistent [14,16,17,19]. Synchronization of replicas requires two-phase commits in order to guarantee consistency of data [18]. As the update rate increases, the level of performance can deteriorate quickly.

These techniques require additional hardware resources and add complexity to the architecture. A system engineer should evaluate their uses carefully prior to inclusion in their design, especially if performance is the main reason for inclusion. An additional benefit of these techniques is that they give the system fault tolerance. If one of the copies is unavailable, then the system will continue to work. If this is the desired quality, our methodology will still allow these servers to be optimally deployed. Each cluster or replica copy would be treated as a separate server. The model would not know that two servers are identical. It would only know the measurements are identical since we treat each server as a black box. Replica copies would actually be complex servers where the related servers invoke calls on the copies when set methods are invoked.

5. Distributed, Real Time Systems

Research in optimization of distributed, real-time systems is also widely available. This research is aimed at

real-time systems where the optimization is directed at the scheduling of tasks, similar to many load balancing techniques. In non-real time systems where user interactions dictate the majority of the tasks, scheduling of tasks is impractical. Conversely, moving object server locations around in a distributed, real-time system is often impractical. For these reasons, this work is directed at the non real-time arena.

6. Shared-Memory Multiprocessor Systems

Other approaches to improving the performance of servers include hardware improvements. These approaches usually involve shared-memory multiprocessor systems. While research focused on hardware, such as the Cache Coherent Non-Uniform Memory Access (CC-NUMA), does improve the performance of object servers, these solutions are not an option for most system engineers due to the high cost of the systems [31]. Much of the research involved in shared-memory multiprocessor systems relies on the existence of fast, reliable shared-memory, which doesn’t exist in a heterogeneous network of low cost computers. Multi-processor systems are orders of magnitude more expensive than single CPU systems. While these systems may be the only option for large monolithic servers, multi-server architectures can distribute their servers across much cheaper single CPU systems to gain needed performance.

Shared-Memory multiprocessor systems are the clear choice for systems where there is a large amount of objects that are interrelated and where speed is essential. Spreading the objects over multiple machines would incur a network cost that might be too high for some systems.

7. GRID Computing

Research in Grid Computing also has emerged as an important new field in distributed computing. Large-scale resource sharing across multiple organizations increases both the set of available network resources and the complexity of the underlying architecture. The need for authentication, authorization, resource access, resource discovery, and other challenges require applications to conform to “intergrid protocols” [29,30].

While these added complexities would be needed for environments like the Internet, they are not as useful in much smaller, single organization environments. The environment that the grid research is aimed at can be characterized by extremely large tasks where a network delay between computers becomes lessened. While the research in this dissertation can be expanded to include architectures similar to the grid, it is currently aimed at a much smaller architecture of a single local area network.

8. Code Synthesis of Object Servers

The next big push in automating the generation of software is coming in the area of object servers. The Joint Task Force Advanced Technology Demonstration (JTF ATD) Project sponsored by the Defense Advanced Research Projects Agency's (DARPA's) Information System Office (ISO) has been doing research into collaborative, distributed complex object oriented technologies from 1995 to 2000.

The research completed in the JTF ATD Project has resulted in a model driven code generation process and tools that enables system and application developers to build distributed, complex, object oriented systems. The technology has been designated the Next Generation Information Infrastructure (NGII) 2000. It uses the Universal Modeling Language (UML) to describe the models of the system the developer is building along with code scripts that drive the code generation process within the Common Object Request Broker Architecture (CORBA) environment. The researchers have found that the code generation process is extensible to Enterprise JAVA Beans (EJB) as well as Extensible Markup Language (XML) and Extensible Scripting Language (XSL) and systems have been code generated in these domains [32].

The model driven code generation model and tools developed within the JTF ATD to date for the NGII 2000 allows the system and application developers to automatically generate infrastructure services from a UML Class Model, and to create a framework of code to support the application developer in a CORBA environment.

9. Current Practices

Because of the difficulty in producing the infrastructure code necessary to support distributed object computing, many developers produce huge monolithic object servers as seen in Figure 3. A powerful machine is usually needed to adequately handle this server and successful applications that experience large increases in the number of users may outgrow the capabilities of the fastest available single machine. With automated code-generation tools, these servers will be much easier to produce and reconfigure. This will allow servers to be partitioned according to the logical model as Figure 4 illustrates. This allows servers to be decomposed by partitioning unrelated or loosely related objects into different physical servers that can be deployed across the network to take advantage of the available assets. By taking advantage of all the assets on the network, faster response times can be achieved.

Many vendors claim to address optimization within their products. Most of these involve the employment of replicas and clusters embedded in the logic of their EEJB, CORBA or DCOM enterprise tools, like Allaire's Jrun or Borland's Visibroker. These products work best if your system has just a few stateless object classes with numerous instances and plenty of available hardware. In an environment where multiple vendor products are present, they lack the ability to reason outside of their implementation. This dissertation treats each vendor implementation as black box and has the ability to reason over the entire mixed bag of servers.

IBM's Distributed Application Partitioning (DAP) automatically determines how best to place objects in a distributed program. DAP monitors the execution and records how often particular objects communicate with each other. Then it computes an object placement by determining the minimum cut set of a graph. The focus this research is a single application and a single user. Further research is needed to see if this approach has merit in a multiple application, multiple user environment.

While these products have value, they are limited to optimizing servers implemented within their tool. The ability to reason about performance over a mixed bag of object servers regardless of middleware (EEJB, CORBA, DCOM)

implementation was not found in any product or previous research.

Many networks of computers are installed with a single purpose in mind. Over time, these networks support an evolving set of tasks. Even though the original role the network played can change dramatically, rarely does a single system engineer revisit the deployment strategy. What a user ends up with is usually the product of multiple system engineers choices made from his additional changes without regard to the system and its roles as a whole. It is infeasible, because of cost, to hire a system engineer to assess the whole system every time a change occurs. In the end, the user is left with a system that’s deployment strategy boarders on randomness.

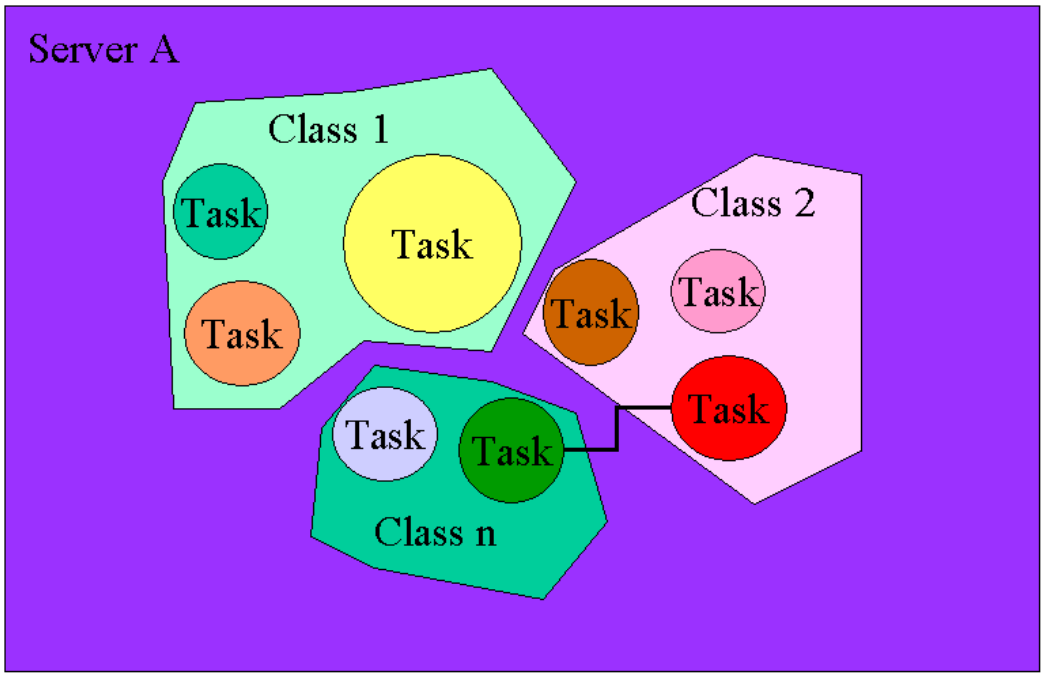

Figure 3: Large Monolithic Object Server.

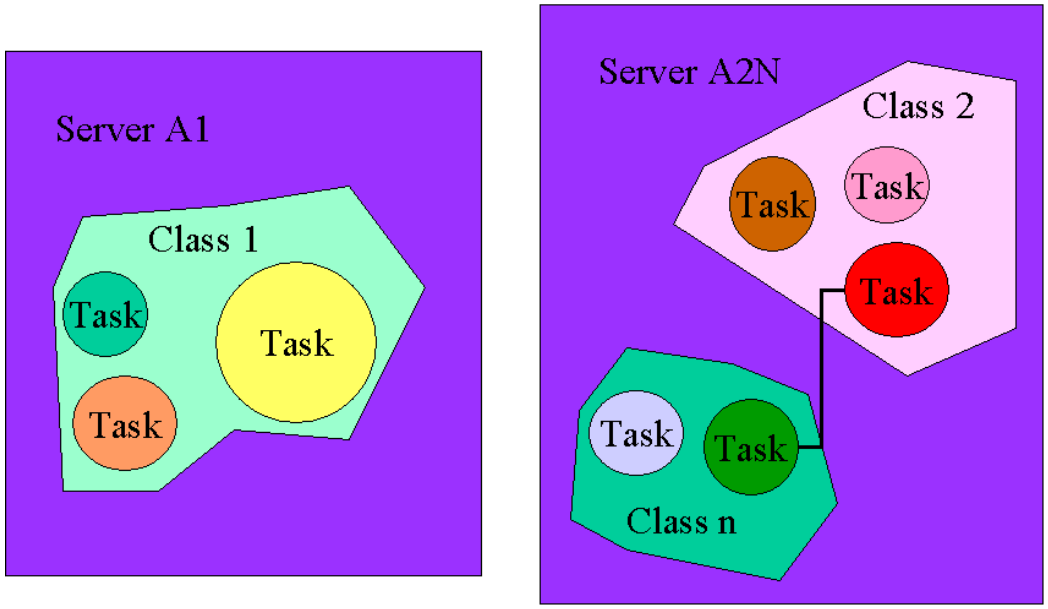

Figure 4: Decomposed Object Servers.

THIS PAGE INTENTIONALLY LEFT BLANK

III. CHAPTER 3

A. PROFILING ARCHITECTURAL COMPONENTS

The goal of this chapter is to describe a methodology that can repeatedly generate distributed object oriented server deployment architectures to take advantage of network resources for the purpose of reducing client response time to complete his job. The system must be able to reason about deployment strategies of loosely related objects. Finally, the system must allow an end user to set priorities on end user tasks.

The data collected from the system is stored in a table called a profile. There are profiles for each machine, server, application and user type. There is also a profile that describes the network. The system then must map all of these profiles into equations to minimize response time.

The actual elements of a profile and the methodologies for collecting these profiles are described in later chapters. This chapter is here to expose the reader to the concept of profiles and their uses.

1. Evolution

Over time, a collection of hardware, software and user requirements will change in a given environment. Common hardware changes consist of adding new computers, removing

old computers, upgrading CPUs, modifying RAM and modifying network bandwidth capacity. Each of these hardware changes will produce an event that would trigger the system to re-evaluate its deployment strategy.

Software can also be quite dynamic in nature. New object servers and applications can appear. Old ones can be removed. Existing object schemata and methods can be changed. Each of these changes would trigger an event to re-evaluate the deployment strategy.

2. Loosely Related Objects

Objects that invoke methods on other objects are said to be related objects. Not all objects that are related must necessarily be contained in a single object server. There is a point where the performance of the system would improve by moving the object into a different server. This is usually the case when none of the application code exercises the relationship or exercises it only very rarely. The approach will be able to reason about not only deploying object servers, but also recommend the schema supported by these object servers.

3. Use Patterns

User requirements can also be in a state of flux. Most computer systems are used to support multiple jobs. Business-hour requirements can differ greatly from after-hours computational requirements. A developer’s network of

computers can support multiple projects, but may need to be optimized for a single project for demonstrations. In the military, the operational mission being supported can change significantly. For example, a set of distributed object servers could be used to support many applications aboard a ship. These applications could handle such tasks as Anti-Submarine Warfare (ASW), Anti-Surface Warfare (ASUW), Anti-Air Warfare (AAW), Electronic Warfare (EW), humanitarian missions and rescue missions. The relative computational activity of these applications could differ significantly on different missions of the ship.

Optimizing a system of object servers for all possible roles would not be optimal when the system is only performing a couple of missions at a time. By profiling each role, the user could choose to optimize his deployment to increase the response time of the user chosen roles. In this way, the user could tune his system to give peak performance for the task he is currently trying to perform.

4. Profiles

A profile is an abstraction of a given characteristic of the system. The elements in the profile are the raw data that the model will use to reason about the given characteristic. The tricky part is to figure out what elements are needed in the different profiles, how to map these profiles into equations and then model how these

profiles interact with each other. The more complex the modeling of the hardware becomes the more computationally intensive the approach will become. Initially, we explore an approach with rather simplistic profiles to demonstrate its capabilities.

a) Hardware Profiles

The aspects being modeled in the hardware profiles include characteristics of each computer such as CPU speed, RAM size and disk capacity. The hardware profile also models the network speed between each pair of computers. Current hardware profiles do not directly support multi-processor computers, but they could be modeled as groups of separate nodes with very high “network speeds” between them.

b) Object Server Profiles

Object servers need to be profiled for metrics associated with each method call in each object. The computational time of each method call should be captured and normalized to a specific hardware architecture. Since object servers ideally run continuously, the RAM and disk usage of the object server must also be measured and summarized. The hardware profile and the object profile are sufficient to optimize the server deployment for the case where all the functionality contained in all the objects is of equal value to the user. Metrics can be collected

easily with a small client application that exercises each method call and records the data. Thus, actual implementation code for the application isn't needed to estimate the object server profiles.

c) Client Application Profiles

Profiling becomes more difficult if the application code is not available. When no source code is available, then the system must allow a user to create a task and record all the events that occur in the task. This could be done by simulation or monitoring calls to the object servers when the system is in a training mode. The plus side to this method is that the user could profile more complex tasks involving many user interactions into a single profile.

d) User Profiles

The more difficult part is profiling user criteria for optimizing the system. The way a user interacts with a system can be characterized, but not precisely predicted. For this reason, we are left with an optimization approach with stochastic variability.

The most straightforward approach involves making a table for each user-initiated task an application can perform. The call map for this table entry can be scanned for calls to object servers. The actual local processing of

the client code is not factored into the objective function to be optimized, since the user predetermines the location of the client software when he initiates the application on a machine. The user then has a table of all tasks each application can perform. The user would then create a new role and select the tasks that are of most importance to that role. The user then selects a subset of roles and has the system come up with an optimal deployment strategy to meet these criteria. A more refined profile could include frequency information for the tasks and calls for each task, and response time goals for each task.

IV. CHAPTER 4

A. METHODOLOGY FOR TURNING PROFILES INTO MODEL VALUES

In order to compute the optimal deployment strategy from a given set of profiles, one needs to map these profiles into equations that can be solved for minimum response time. To illustrate the mappings, we present an example. The example consists of four machines, four object servers containing a single object and three client applications.

1. Hardware Profile Examples

Table 1 shows the hardware profile of the four machines. In the profile is an entry for the physical RAM of each machine measured in bits and an entry for the speed of its CPU measured in megahertz.

Table 1: Machine Profile.

MACHINE	RAM (b)	CPU Speed (MHz)
W	512,000,000	300
X	1,024,000,000	200
Y	1,024,000,000	400
Z	2,048,000,000	500

Table 2 shows the network bandwidth available to communicate from each machine to the other. In this

example, the machines will have equal bandwidth between each other, as is the case when all servers are running on the same local LAN. The speed of communications between servers on the same machine is more difficult to measure. These speeds usually lie in the area bounded by the speed of the machines back plane and the speed of the network. It is dependent on the operating system, implementation of the middleware, and other factors. For this reason, the system currently assumes that intra-machine communication is twice as fast as inter-machine communication as a nominal representative case. Of course, the analysis can be run with best and worst-case scenarios by inputting the boundary values stated above, or with values determined from measurements of the actual hardware.

Table 2: Network Speed Profile.

Machine to Machine Speed (bps)	W	X	Y	Z
W	2,000,000	1,000,000	1,000,000	1,000,000
X	1,000,000	2,000,000	1,000,000	1,000,000
Y	1,000,000	1,000,000	2,000,000	1,000,000
Z	1,000,000	1,000,000	1,000,000	2,000,000

2. Object Server Profile Examples

Besides the hardware profiles, we need to have the server profiles. For simplicity, each server will have only one object type. Later, we will show how multiple object types in a server are handled. We will need three different tables to depict the profile of object servers. These tables are a RAM usage table, a normalized table of CPU usage and message size for each call, and a table of calls for server-to-server method invocations. Table 3 lists each server's RAM requirements.

Table 3: Object Server RAM Profile.

SERVER	RAM Required (b)
A	352,000,000
B	264,000,000
C	528,000,000
D	352,000,000

The second table of information we need for object server profiles is the timing of each individual method call available in each server. In this example, server A has four methods, server B has two methods, server C has three methods and server D has four methods. Table four gives the profiles for these servers.

Table 4: Object Server Performance Profile.

SERVER	Method	CPU time (s)	Average Size of Message (b)
A	1	0.0056	14000
A	2	0.01454	2300
A	3	0.0034	5600
A	4	0.0123	22000
B	1	0.0089	500000
B	2	0.0124	340000
C	1	0.0122	40000
C	2	0.0141	500000
C	3	0.0034	50000
D	1	0.0333	33000
D	2	0.0102	2700
D	3	0.0183	35000
D	4	0.0383	40900

The last table of information we need to describe our object server profiles is a list of server-to-server method invocations. This is a list of methods that call other server methods from within a server. In this example, every time method B.2 is called from a client application, B.2

invokes method C.1. There must not be any cycles in this table or it will be impossible to normalize Table 4 correctly. The values in Table 4 must contain only a value for the processing on that server. Any time spent waiting on return calls from other servers must be removed. Also, the measured time must be comparable between machines. If the data is collected on different machines, then the values must be normalized for comparability. A simple solution to this problem is to multiply the measured value by a ratio of the machine speed of the measured machine by the machine speed of a normalized machine. Table 5 lists these methods of which there is only one for this example.

Table 5: Complex Object Server Profile.

Primary Method	Secondary Method
B.2	C.1

3. Client Application Profile Examples

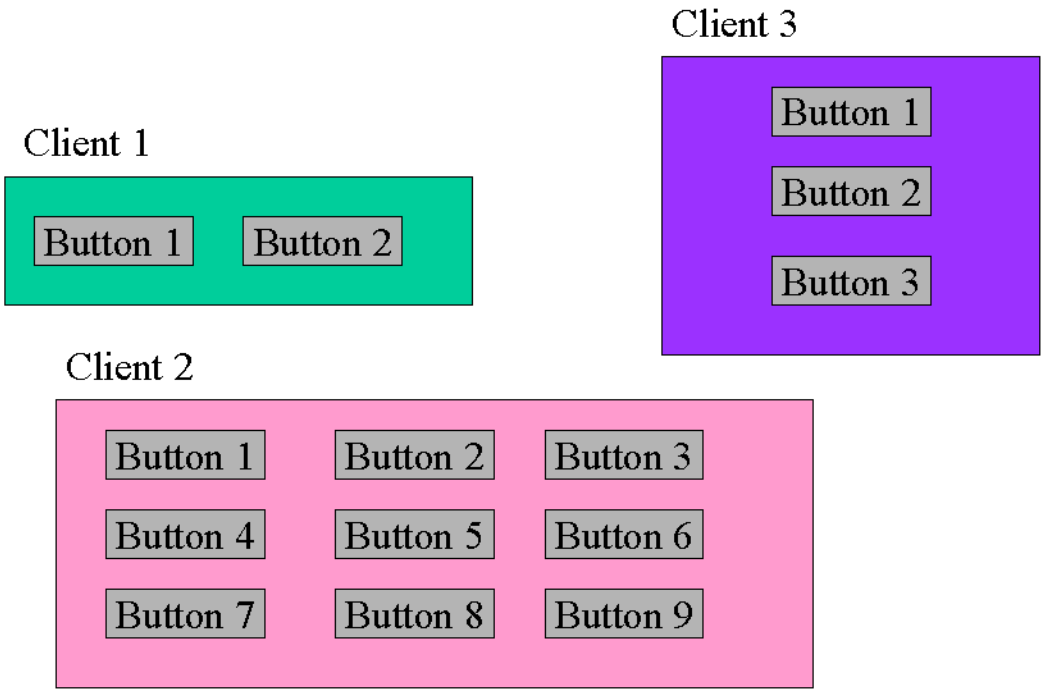

Figure 5: Client Applications' Interface.

Table 6: User Interface Call Chart.

Button	Methods Called
C1.B1	A.1
C1.B2	A.2 + B.1
C2.B1	C.1 + C.2 + D.1
C2.B2	C.3
C2.B3	C.2
C2.B4	C.3 + D.4
C2.B5	A.1 + *B.2* + D.3
C2.B6	*B.2*
C2.B7	A.4
C2.B8	D.1 + D.2 + D.3
C2.B9	A.1 + A.2 + A.3 + *B.2*
C3.B1	C.1 + D.3
C3.B2	B.1 + *B.2*
C3.B3	C.2 + D.4

In Figure 5, the user interfaces for three client applications are shown. In Table 6, we list the object server call trees for each action a user can initiate. For example, if the user were to use client application number 3 and click on Button 3, then the client application code would invoke the second method on object server C and the fourth method on object server D. Notice that all of the B.2 method calls are in italics. This indicates that *B.2* is actually equal to B.2 and C.1 since B.2 contains a call to C.1.

4. User Profile Examples

Let’s assume that this network of computers supports three different roles for the user and the following is the use pattern over a period of time. Role 1 may pertain to a daytime shift. Role 2 may pertain to a nighttime shift and Role 3 may pertain to an end of the month inventory function. From the table below we get that during the daytime shift or Role 1, the average user clicked on [Application 1, Button 1] 50 times, [Application 1, Button 2] one time, [Application 2, Button 1] 25 times, and [Application 2, Button 6] one time on average over a 100 second period.

Table 7: User Roles Profiles.

ROLE	CALL PATTERN (per 100 second period)
Role 1 [Daytime]	50 C1.B1 + 1 C1.B2 + 25 C2.B1 + C2.B6
Role 2 [Nighttime]	10 C1.B1 + 40 C1.B2 + 24 C3.B2
Role 3 [Month End]	50 C2.B5 + 10 C2.B9 + 30 C2.B3 + 1 C2.B2 + 1 C3.B2

5. Objective Function

The objective function that needs to be minimized is the sum of all of the response times for a given call pattern over a given time interval. Since we want to allow the user the freedom to run client applications from anywhere on the network, we will ignore all processing on the client machines and all network delay between client machines and server machines. The only factors we will consider for optimizing our server deployment are the processing of the object server and the network delay between object servers. Therefore, the function that we wish to minimize is:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * R_n * S_{norm}}{S_m} + \sum_{i=0}^{N} \sum_{j=0}^{N} \frac{B_{ij}}{Q_{ij}} \right]$$

subject to the following four constraints:

1. Object Servers cannot be split across machines.

   a_{nm} = 1, iff server n is running on machine
   0, otherwise

2. Each Server can run on only one machine [no multiple instances of the same server.

$$\forall n \left[\sum_{m=0}^{M} a_{nm} \equiv 1 \right]$$

3. RAM usage by the object servers cannot pass a set threshold on each machine.

$$\forall m \left[\sum_{n=0}^{N} a_{nm} * V_n \leq T_m * U \right]$$

4. CPU time on a given machine cannot surpass the corresponding real time interval.

$$\forall m \left[\sum_{n=0}^{N} \frac{a_{nm} * R_n * S_{norm}}{S_m} \leq C \right]$$

where

N = Number of object servers
M = Number of physical machines
R_n = Normalized machine load of server n (seconds, s)
S_{norm} = Speed of the normalizing machine (MHz)
S_m = Speed of machine m (MHz)
B_{ij} = Data sent between server i to server j (bits, b)
Q_{ij} = Network Speed between server i to server j (bps)
T_m = Physical RAM on machine m (bits, b)
V_n = Memory allocated by server n (bits, b)

U = Multiple to limit RAM utilization [0.1 < U < 3.0]
C = Time Interval [seconds, s]

Note that the optimization process ranges over all possible combinations for a_{nm} and finds the minimum based on the above objective function and constraints. Q_{ij} is dependent on a_{nm}. Its value is a function of the relative location of the two servers. Depending on this function, the system of equations may be linear or non-linear. For the examples in this dissertation, $Q_{ij} = \left(1 + \sum_{m=0}^{M} a_{im} * a_{jm}\right) * L$, where L is the LAN speed. All other terms are fixed either by measurement or input.

a) Processing Speed Term

The processing speed term of the objective function is:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * R_n * S_{norm}}{S_m} \right]$$

This part of the function looks at all possible deployment patterns. a_{nm} is used to keep track of the deployments. a_{nm} is zero if SERVER n is not located on MACHINE m. If SERVER n is located on MACHINE m, then a_{nm} is one. S_{norm} is the CPU clock rate of the machine used to

normalize the object server profile. S_m is the CPU clock rate of MACHINE m. R_n is actually the workload on the CPU for SERVER n. This is a sum of the products of all the methods in SERVER n times the number of times a user ROLE calls that method. This term of the objective function ends up being expressed in units of seconds.

b) Network Speed Term

The network speed term of the objective function is:

$$Minimize \left[\sum_{i=0}^{N} \sum_{j=0}^{N} \frac{B_{ij}}{Q_{ij}} \right]$$

The network speed term just adds some time for each time a server-to-server method is called. The number of bits is divided by the rate of transmission. B_{ij} is expressed in bits. Q_{ij} is expressed in bits per second. This term of the objective function ends up being expressed in units of seconds.

c) RAM Limits

In the model, we must have some logic for not overloading a machine so much that the processing of the machine bogs down. To accomplish this task, we limit the

amount of a machines RAM that can be used by the object servers. Constraint 3 in the objective function servers this purpose.

This constraint basically states that the total memory usage of all object servers loaded on a machine will be less than a percentage of the memory on that machine.

d) CPU Limits

We also have to limit the loading of the CPU. Since all of the processing measurements are averages and the user profiles are averages over time, we cannot exceed 100% CPU loading. Even though the CPU can queue tasks when overloaded, it doesn't have the chance to catch up if the user profiles truly reflect the user requests. Constraint 4 servers this purpose.

e) Specializing the Objective Function for Role 1

Role 1 consists of 50 C1.B1 calls, one C1.B2 call, 25 C2.B1 calls, and one C2.B6 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 5.

50 [A.1] + 1 [A.2 + B.1] + 25 [C.1 + C.2 + D.1] + 1 [*B.2*] =
50 [A.1] + 1 [A.2 + B.1] + 25 [C.1 + C.2 + D.1] + 1 [B.2 + C.1] =
50 A.1 + A.2 + B.1 + 25 C.1 + 25 C.2 + 25 D.1 + B.2 + C.1 =
50 A.1 + A.2 + B.1 + B.2 + 26 C.1 + 25 C.2 + 25 D.1

This leads to the following values for R, the normalized machine loads for each server. In the following syntax, R(A) will stand for the load on server A, R(B) for the load on server B, R(C) for the load on server C, and R(D) for the load on server D.

R(A) = 50 [A.1 values for CPU] + 1 [A.2 value for CPU]
= 50 [5.6] + 1 [14.54]
= 294.54

R(B) = 1 [B.1 values for CPU] + 1 [B.2 value for CPU]
= 1 [8.9] + 1 [12.4]
= 21.3

R(C) = 26 [C.1 values for CPU] + 25 [C.2 value for CPU]
= 26 [12.2] + 25 [14.1]
= 669.7

R(D) = 25 [D.1 values for CPU]
= 25 [33.3]
= 832.5

There is only one italicized method call prior to substitution, so there is only one network value to deal with. There is also only one complex server call. The syntax BITS[I,J] stands for the data bits sent from server I to server J.

BITS[B,C] = 1 [*B.2* message in bits]
= 40000

The full LINGO model for Role 1 can be found in Appendix A labeled ADOA6.1.1.

f) Specializing the Objective Function for Role 2

Role 2 consists of 10 C1.B1 calls, 40 C1.B2 call, and 24 C3.B2 calls. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 5.

10 [A.1] + 40 [A.2 + B.1] + 24 [B.1 + *B.2*] =
10 [A.1] + 40 [A.2 + B.1] + 24 [B.1 + B.2 + C.1] =
10 A.1 + 40 A.2 + 40 B.1 + 24 B.1 + 24 B.2 + 24 C.1 =
10 A.1 + 40 A.2 + 64 B.1 + 24 B.2 + 24 C.1

This leads to the following values for R.

R(A) = 10 [A.1 values for CPU] + 40 [A.2 value for CPU]
= 10 [5.6] + 40 [14.54]
= 637.6

R(B) = 64 [B.1 values for CPU] + 24 [B.2 value for CPU]
= 64 [8.9] + 24 [12.4]
= 867.2

R(C) = 24 [C.1 values for CPU]
= 24 [12.2]

= 292.8

R(D) = 0

There is only one italicized method call prior to substitution, so there is only one network value to deal with. However, it is called 24 times.

BITS[B,C] = 24 [*B.2* message in bits]
= 24 [40000]
= 960000

The full LINGO model for Role 2 can be found in Appendix A labeled ADOA6.2.1.

g) Specializing the Objective Function for Role 3

Role 3 consists of 50 C2.B5 calls, 10 C2.B9 calls, 30 C2.B3 calls, one C2.B2 call, and one C3.B2 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 5.

50 [A.1 + *B.2* + D.3] + 10 [A.1 + A.2 + A.3 + *B.2*] + 30 [C.2] + 1 [C.3] + 1 [B.1 + *B.2*] =

50 A.1 + 50 *B.2* + 50 D.3 + 10 A.1 + 10 A.2 + 10 A.3 + 10 *B.2* + 30 C.2 + C.3 + B.1 + *B.2* =

60 A.1 + 10 A.2 + 10 A.3 + B.1 + 61 *B.2* + 30 C.2 + C.3 + 50 D.3 =

60 A.1 + 10 A.2 + 10 A.3 + B.1 + 61 [B.2 + C.1] + 30 C.2 + C.3 + 50 D.3 =

60 A.1 + 10 A.2 + 10 A.3 + B.1 + 61 B.2 + 61 C.1 + 30 C.2 + C.3 + 50 D.3

This leads to the following values for R.

R(A) = 60 [A.1 values for CPU] + 10 [A.2 value for CPU] + 10 [A.3 value for CPU]
= 60 [5.6] + 10 [14.54] + 10 [3.4]
= 515.4

R(B) = 1 [B.1 values for CPU] + 61 [B.2 value for CPU]
= 1 [8.9] + 61 [12.4]
= 765.3

R(C) = 61 [C.1 values for CPU] + 30 [C.2 value for CPU] + 1 [C.3 value for CPU]
= 61 [12.2] + 30 [14.1] + 1 [3.4]
= 1170.6

R(D) = 50 [D.3 values for CPU]
= 50 [18.3]
= 915

There is only one italicized method call prior to substitution, so there is only one network value to deal with.

BITS[B,C] = 61 [*B.2* message in bits]
= 61 [40000]
= 2440000

The full LINGO model for Role 3 can be found in Appendix A labeled ADOA6.3.1.

V. CHAPTER 5

A. EXERCISING OF OPTIMIZATION MODELS

In order to better understand the final model used to optimize the distributed, object-oriented system, it is valuable to follow the steps along the way. By explaining each addition to the original model one at a time, and then showing the impact of the addition to the outcome of the optimization, one gets a better understanding of each refinement to the model. In the base model, we assume that all methods in each server are called once in a given period. This is a common assumption used to deploy servers when the systems engineer doesn’t have prior knowledge of the usage of the object servers.

The solution for the optimal deployment strategy for the example given in the previous chapter is that all servers would run on machine Z since machine Z is the fastest and has available RAM to support all of these servers. Our initial model assumes that response time depends only on processor speed, although realistically the transaction rate is also a significant factor. However, we can impose limits on the amount a RAM available to object servers on a particular machine.

1. RAM Limits Refinement

There are many reasons why we might want to impose RAM limits. The machines available for hosting these servers may have other jobs. The RAM limit in this way saves part of the systems RAM to handle these background jobs. Also, the processing speed of a machine can be significantly affected when the RAM utilization approaches one hundred percent and the system starts to rely heavily on virtual memory. The following tables show the deployments when different percentages of a machine's RAM are available to object servers.

Table 8: Model Output with 66% RAM utilization.

Machine	Server
W	None
X	None
Y	A
Z	B,C,D

Table 9: Model Output with 50% RAM utilization.

Machine	Server
W	None
X	B
Y	A

Z	C,D

Table 10: Model Output with 40% RAM utilization.

Machine	Server
W	None
X	A
Y	D
Z	B, C

Note that there is no viable solution to this problem when the system is limited to 39 percent of the RAM available. There seems to be sufficient assets available, but this isn't the case. The model does not allow running part of a server on one machine and the rest on another. In the model, a server is an atomic element and cannot be split among machines.

2. Network Speed

The above results were computed with a model that assumed that no communication was present between object servers. However, this may not be the case. In some architectures, there will be servers that are loosely related to other servers. In this case, the speed of the network has a more dramatic role on the deployment strategy. The interaction between client applications and servers is

ignored since we are allowing the client host to be arbitrary. If the second method call in server B always called the first method in server C, then the network speed would become much more of a factor. In this case, we keep RAM limit at fifty percent and adjust network speed from one Mbps to half of that speed. The following tables show how this refinement changes the deployment pattern.

Table 11: Model Output with 1Mbps, 50% RAM utilization.

Machine	Server
W	None
X	B
Y	A
Z	C,D

Table 12: Model Output with 0.5 Mbps, 50% RAM utilization.

Machine	Server
W	None
X	A
Y	D
Z	B,C

3. Usage Patterns

The last two scenarios optimized the system for each server and its methods having equal transaction frequency. However, this isn't very realistic usage for the object servers and in some cases it isn't even a possible usage pattern. The user interfaces may not allow each method in each server to be called an equal number of times.

The following scenarios add roles to the list of profiles for the model to optimize. These roles have more realistic use patterns for the different jobs a user would actually perform on the system.

For this example, we use the user interfaces and three roles defined in the previous chapter. These three usage patterns lead to three different deployment patterns as show in Table 13.

Table 13: Model Outputs with Different User Roles.

Machine	Role 1	Role 2	Role 3
W	None	None	None
X	B	D	A
Y	A	A	D
Z	C,D	B,C	B,C

4. Concurrent Users

Another adjustment to the model is limiting the load on a CPU. Since usage patterns for a role are over a period of time, we cannot allow the CPU to be overburdened. The model works equally well if you were to combine multiple different roles. When we limit CPU, we get the following tables of deployment patterns. The RAM limit was set to 80 percent and the bandwidth was set at 1Mbps for these computations.

Table 14: Model Outputs with Concurrent Role 1Users.

Machine	Role 1 (1 user)	Role 1 (100 users)	Role 1 (119 users)
W	None	A	A
X	None	None	B
Y	None	B, C	C
Z	A, B, C, D	D	D

Table 15: Model Outputs with Concurrent Role 3 Users.

Machine	Role 3 (1 user)	Role 3 (50 users)	Role 3 (75 users)
W	None	None	B
X	None	None	A
Y	None	A, D	D
Z	A, B, C, D	B, C	C

B. CONCLUSIONS

The model reacts in a logical fashion to the changes that are placed on the environment. At this point, it is time to test the model against a real system to validate the model.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CHAPTER 6

A. JAVA RMI EXPERIMENTATION RESULTS

We tested the validity of the model presented in Chapter 4 by experimental measurement. A testbed was created with Windows 2000 machines that match the characteristics of the machines in the following tables. Servers were created using JDK 1.3 and RMI as the middleware.

Software to simulate the three different users was also created. The users were simulated with a random choice for button selection that has a uniform distribution similar to the roles. This simulation software was instrumented to measure the actual time the software was blocked waiting for an object server method call to response. All 27 different configurations were established and the average response time for each configuration was measured and recorded. Between each simulation, the testbed machines were rebooted.

All 27 configurations were tested twice. One tested the configuration with the object servers using much less than the stated memory needs. Another tested the configuration with the object servers using all of the stated memory needs. Some configurations strained the machines memory limits. These configurations resulted in

system failures in the test with the object servers using all of the stated memory needs. These system failures are listed as error in the tables of results. It should be noted that Windows 2000 did a much better job of swapping when memory utilization exceeded 100% than a previously tested operating system, Windows NT.

1. Experiment Characteristics

The hardware was purchased for this experiment with the following CPU clock rates and RAM. Each machine had a 10/100Mbps Ethernet card installed. The machines were connected via a Ethernet hub rated for 10/100Mbps. The LED indicators on the Ethernet hub verified the rate of 100Mbps data transfer rate.

Table 16: Machine Profile for JAVA RMI Experiments.

MACHINE	RAM (bits)	CPU Speed (MHz)
SIX	512,000,000 = 64MB	600
BR733	1,024,000,000 = 128MB	733
GIGA	1,024,000,000 = 128MB	1000

Table 17: Network Speed Profile for JAVA RMI Experiments.

Machine to Machine Speed (bps)	SIX	BR733	GIGA
SIX	200,000,000	100,000,000	100,000,000
BR733	100,000,000	200,000,000	100,000,000
GIGA	100,000,000	100,000,000	200,000,000

Table 18: Object Server RAM Profile for JAVA RMI Experiments.

SERVER	RAM Required (bits)
A	352,000,000 = 44MB
B	480,000,000 = 60MB
C	528,000,000 = 66MB

Table 19: Object Server Performance Profile for JAVA RMI Experiments.

SERVER	Method	CPU time (s)	Average Size of Message (b)
A	1	0.5796	112000
A	2	2.6203	18400
A	3	1.18175	44800
A	4	2.0264	176000
B	1	1.76655	4000000
B	2	3.70085	2720000
C	1	3.0043	320000
C	2	4.8040	4000000
C	3	0.48815	400000

Table 20: Complex Object Server Profile for JAVA RMI Experiments.

Complex Method	Exterior Calls
B.2	C.1

Table 21: User Role Profiles for JAVA RMI Experiments.

ROLE	CALL PATTERN (observation interval is 990 seconds)
Role 1	50 C1.B1 + 1 C1.B2 + 1 C2.B1 + 1 C2.B6
Role 2	10 C1.B1 + 40 C1.B2 + 24 C3.B2
Role 3	50 C2.B5 + 10 C2.B9 + 30 C2.B3 + 1 C2.B2 + 1 C3.B2

Table 22: Application Call Chart for JAVA RMI Experiments.

Button	Methods Called
C1.B1	A.1
C1.B2	A.2 + B.1
C2.B1	C.1 + C.2
C2.B2	C.3
C2.B3	C.2
C2.B4	C.3
C2.B5	A.1 + *B*.2

C2.B6	*B*.2
C2.B7	A.4
C2.B8	C.3 + A.3
C2.B9	A.1 + A.2 + A.3 + *B*.2
C3.B1	C.1
C3.B2	B.1 + *B*.2
C3.B3	C.2

Table 23: Expanded User Role Profile for JAVA RMI Experiments.

ROLE	**Methods Called in Role**
Role 1	50 * (A.1) + 1 * (A.2 + B.1) + 1 * (C.1 + C.2) + 1 * (*B*.2)
Role 2	10 * (A.1) + 40 * (A.2 + B.1) + 24 * (B.1 + *B*.2)
Role 3	50 * (A.1 + *B*.2) + 10 * (A.1 + A.2 + A.3 + *B*.2) + 30 * (C.2) + 1 * (C.3) + 1 * (B.1 + *B*.2)

2. Specializing the Objective Function for Role 1

Role 1 consists of 50 C1.B1 calls, one C1.B2 call, one C2.B1 call, and one C2.B6 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 4.

50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [*B.2*] =
50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [B.2 + C.1] =
50 A.1 + A.2 + B.1 + C.1 + C.2 + B.2 + C.1 =
50 A.1 + A.2 + B.1 + B.2 + 2 C.1 + C.2

This leads to the following values for the array R for the optimization equation.

R(A) = 50 [A.1 values for CPU] + 1 [A.2 value for CPU]
= 50 [579.6] + 1 [2620.3]
= 31600.3

R(B) = 1 [B.1 values for CPU] + 1 [B.2 value for CPU]
= 1 [1766.55] + 1 [3700.85]
= 5467.4

R(C) = 2 [C.1 values for CPU] + 1 [C.2 value for CPU]
= 2 [3004.3] + 1 [4804.0]
= 10812.6

There is only one italicized method call prior to substitution, so there is only one network value to deal with.

BITS[B,C] = 1 [*B.2* message in bits]
= 320000

3. Specializing the Objective Function for Role 2

Using the same approach as in 4.6.1, we get the following for Role 2:

R(A) = 110608
R(B) = 201879.6
R(C) = 72103.2

There is only one italicized method call prior to substitution, so there is only one network value to deal with. However, it is called 24 times.

BITS[B,C] = 24 [*B.2* message in bits]
= 24 [320000]
= 7680000

4. Specializing the Objective Function for Role 3

R (A) = 72796.5
R (B) = 227518.4
R (C) = 327870.45
BITS[B,C] = 19520000

5. Model Outputs

The optimization model determines the following deployment strategies for the different roles when setting

different RAM limits and keeping all other variables the same as in the last example. Solving the optimization problem defined in previous section with the parameter values determined derives these results. The LINGO models for the below results can be seen in Appendix A. These models are listed as ADOA8.1 for Role 1(1 user), ADOA8.2 for Role 2(1 user) and ADOA8.3 for Role 3(1 user). Table 24 gives the results when RAM utilization was set at 1.5 times the physical RAM of the machines.

Table 24: Model Outputs, 150% RAM util., JAVA RMI Experiments.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	None	None	None
GIGA	A, B, C	A, B, C	A, B, C

Table 25 used the same LINGO models as the above table, but with the RAM utilization set at 1.0 times the physical RAM of the machines.

Table 25: Model Outputs, 100% RAM util., JAVA RMI Experiments.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	B	C	A
GIGA	A, C	A, B	B, C

Table 26 used LINGO models that represented having multiple concurrent users exercising the object servers. These models are listed in Appendix A as ADOA8.1.28 for Role

1 (28 users), ADOA8.2.4 for Role 2 (4 users), and ADOA8.3.3 for Role 3 (3 users). The RAM utilization was set at 1.0 times the physical RAM of the machines.

Table 26: Model Outputs, Concurrent Users, JAVA RMI Experiments.

Machine	Role 1 (28 users)	Role 2 (4 users)	Role 3 (3 users)
SIX	None	A	A
BR733	B, C	C	B
GIGA	A	B	C

6. Role 1 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 1 is present and the servers are using memory at significantly less than their profiled needs. The actual memory use of the minimal memory tests was about 5 megabits.

Table 27: Measured Role1 Min Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 1	CALLS	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	976.331	1000	930	19	27	24
2	GIGA	GIGA	BR733	899.344	1000	952	15	12	21
3	GIGA	BR733	GIGA	960.811	1000	939	20	26	15
4	GIGA	BR733	BR733	1079.641	1000	936	18	23	23
5	BR733	GIGA	GIGA	1140.796	1000	938	22	22	18
6	BR733	GIGA	BR733	1218.875	1000	939	21	23	17
7	BR733	BR733	GIGA	1119.092	1000	949	11	16	24
8	BR733	BR733	BR733	1186.861	1000	947	17	15	21
9	GIGA	GIGA	SIX	991.531	1000	951	12	18	19

10	GIGA	SIX	GIGA	878.782	1000	954	19	13	14
11	GIGA	SIX	SIX	1157.765	1000	933	27	17	23
12	SIX	GIGA	GIGA	1274.376	1000	945	22	22	11
13	SIX	GIGA	SIX	1402.687	1000	945	15	16	24
14	SIX	SIX	GIGA	1413.983	1000	934	22	24	20
15	SIX	SIX	SIX	1642.232	1000	950	14	14	22
16	BR733	BR733	SIX	1197.423	1000	953	14	16	17
17	BR733	SIX	BR733	1306.374	1000	937	20	24	19
18	BR733	SIX	SIX	1305.296	1000	945	17	23	15
19	SIX	BR733	BR733	1291.719	1000	953	20	8	19
20	SIX	BR733	SIX	1467.437	1000	941	21	19	19
21	SIX	SIX	BR733	1441.421	1000	943	14	23	20
22	GIGA	BR733	SIX	1114.344	1000	937	23	22	18
23	GIGA	SIX	BR733	1068.765	1000	944	11	29	16
24	BR733	GIGA	SIX	1246.361	1000	941	19	16	24
25	BR733	SIX	GIGA	1304.703	1000	925	22	30	23
26	SIX	GIGA	BR733	1355.594	1000	944	20	21	15
27	SIX	BR733	GIGA	1306.687	1000	945	19	18	18

The models chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the third fastest average response time in this test. The fact that pattern 10 was the fastest average response time in this test run is a result of the variability of the simulation. Pattern 10 had 15 more calls to Call 1 and 13 less calls to Call 3 than pattern 3 had in the test run. Call 3 in this test had 16 times the server load than Call 1. Since Pattern 3 is a provably superior deployment than Pattern 10, this variability in the simulation of the user is the only possible explanation.

Pattern 1 was the fourth fastest on this run even though it was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Swapping was not an issue since memory usage was low, the explanation lies in the fact that just like Pattern 3, it had a higher number of calls to Call 3 and a lower number of calls to Call 1. More interesting from a software engineering standpoint was the fact that the model proposed a configuration that outperformed most configurations from 10 to 44 percent.

7. Role 1 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 1 is present and the servers are using memory at their profiled needs.

Table 28: Measured Role1 Max Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 1	CALLS	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	977.343	1000	949	13	20	18
2	GIGA	GIGA	BR733	942.984	1000	957	12	13	18
3	GIGA	BR733	GIGA	887.031	1000	954	17	12	17
4	GIGA	BR733	BR733	1041.391	1000	956	11	16	17
5	BR733	GIGA	GIGA	1144.672	1000	942	22	21	15
6	BR733	GIGA	BR733	1282.643	1000	941	17	24	18
7	BR733	BR733	GIGA	1228.031	1000	940	17	17	26
8	BR733	BR733	BR733	1409.515	1000	947	20	15	18
9	GIGA	GIGA	SIX	1039.298	1000	947	24	13	16
10	GIGA	SIX	GIGA	962.609	1000	949	21	16	14

11	GIGA	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	1348.828	1000	952	17	17	14
13	SIX	GIGA	SIX	error	N/A	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	error	N/A	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	1262.703	1000	960	14	12	14
17	BR733	SIX	BR733	1439.251	1000	935	18	18	29
18	BR733	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	1535.657	1000	942	23	14	21
20	SIX	BR733	SIX	error	N/A	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	error	N/A	N/A	N/A	N/A	N/A
22	GIGA	BR733	SIX	982.687	1000	956	16	11	17
23	GIGA	SIX	BR733	1131.969	1000	949	18	15	18
24	BR733	GIGA	SIX	1311.905	1000	941	19	21	19
25	BR733	SIX	GIGA	1189.655	1000	942	18	19	21
26	SIX	GIGA	BR733	1390.297	1000	948	16	19	17
27	SIX	BR733	GIGA	1344.611	1000	949	21	17	13

The models chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the fastest average response time in the stated memory run. Pattern 1 was the fourth fastest on this run even though it was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. More interesting from a software engineering standpoint was the fact that the model proposed a configuration that outperformed most configurations from 10 to 44 percent and that the recommended patterns were free from failures.

8. Role 2 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 2 is present and the servers are using memory at significantly less than their profiled needs.

Table 29: Measured Role2 Min Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 2	CALLS	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	5150.362	1000	145	532	323
2	GIGA	GIGA	BR733	5530.329	1000	132	586	282
3	GIGA	BR733	GIGA	6417.171	1000	123	547	330
4	GIGA	BR733	BR733	6686.376	1000	129	551	320
5	BR733	GIGA	GIGA	5953.015	1000	129	554	317
6	BR733	GIGA	BR733	6233.064	1000	140	542	318
7	BR733	BR733	GIGA	6877.968	1000	129	554	317
8	BR733	BR733	BR733	7238.876	1000	143	523	334
9	GIGA	GIGA	SIX	5958.547	1000	146	536	318
10	GIGA	SIX	GIGA	7176.861	1000	124	549	327
11	GIGA	SIX	SIX	7852.795	1000	126	543	331
12	SIX	GIGA	GIGA	6375.549	1000	139	524	337
13	SIX	GIGA	SIX	6969.187	1000	138	549	313
14	SIX	SIX	GIGA	8211.857	1000	128	533	339
15	SIX	SIX	SIX	8644.362	1000	133	551	316
16	BR733	BR733	SIX	7342.092	1000	133	569	298
17	BR733	SIX	BR733	7862.331	1000	133	565	302
18	BR733	SIX	SIX	8514.078	1000	113	548	339
19	SIX	BR733	BR733	7601.829	1000	136	550	314
20	SIX	BR733	SIX	8033.173	1000	120	559	321
21	SIX	SIX	BR733	8222.031	1000	148	552	300
22	GIGA	BR733	SIX	6987.719	1000	120	563	317
23	GIGA	SIX	BR733	7423.048	1000	139	535	326
24	BR733	GIGA	SIX	6515.812	1000	152	518	330

25	BR733	SIX	GIGA	7783.171	1000	129	524	347
26	SIX	GIGA	BR733	6752.499	1000	139	527	334
27	SIX	BR733	GIGA	7380.828	1000	137	519	344

The model predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 2 was the second fastest average response time. Pattern 1 was the fastest average response in this run, which is the predicted configuration when RAM usage is 150% of physical RAM. Since swapping was not an issue in this test run, the model correctly predicted the right configuration. Again, the configuration chosen by the model outperformed most configurations from 10 to 38 percent.

9. Role 2 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 2 is present and the servers are using memory at their profiled needs.

Table 30: Measured Role2 Max Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 2	CALLS	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	5120.184	1000	154	523	323
2	GIGA	GIGA	BR733	5580.438	1000	155	541	304
3	GIGA	BR733	GIGA	6349.859	1000	129	553	318

4	GIGA	BR733	BR733	6696.141	1000	128	553	319
5	BR733	GIGA	GIGA	5874.642	1000	150	533	317
6	BR733	GIGA	BR733	6204.922	1000	134	568	298
7	BR733	BR733	GIGA	6838.001	1000	143	534	323
8	BR733	BR733	BR733	7215.576	1000	140	541	319
9	GIGA	GIGA	SIX	5916.187	1000	148	541	311
10	GIGA	SIX	GIGA	7288.954	1000	132	519	349
11	GIGA	SIX	SIX	error	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	6424.484	1000	135	528	337
13	SIX	GIGA	SIX	error	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	error	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	error	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	7322.595	1000	146	557	297
17	BR733	SIX	BR733	8148.969	1000	112	563	325
18	BR733	SIX	SIX	error	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	7742.921	1000	120	559	321
20	SIX	BR733	SIX	error	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	error	N/A	N/A	N/A	N/A
22	GIGA	BR733	SIX	6967.624	1000	133	546	321
23	GIGA	SIX	BR733	7343.782	1000	145	537	318
24	BR733	GIGA	SIX	6613.031	1000	128	547	325
25	BR733	SIX	GIGA	7548.561	1000	150	524	326
26	SIX	GIGA	BR733	6772.453	1000	135	534	331
27	SIX	BR733	GIGA	7457.968	1000	125	532	343

The model predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 2 was the second fastest average response time. Pattern 1 was the fastest average response in this run, which is the predicted configuration when RAM usage is 150% of physical RAM. Again, the configuration chosen by the model outperformed

most configurations from 10 to 38 percent and that the recommended patterns were free from failures.

10. Role 3 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 3 is present and the servers are using memory at significantly less than their profiled needs.

Table 31: Measured Role3 Min Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 3	CALLS	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	6741.948	1000	525	109	338	16	12
2	GIGA	GIGA	BR733	8266.516	1000	533	116	336	8	7
3	GIGA	BR733	GIGA	7802.172	1000	552	97	324	11	16
4	GIGA	BR733	BR733	9124.938	1000	537	118	327	12	6
5	BR733	GIGA	GIGA	7413.343	1000	521	137	323	9	10
6	BR733	GIGA	BR733	8508.343	1000	542	108	335	8	7
7	BR733	BR733	GIGA	8142.719	1000	530	117	334	9	10
8	BR733	BR733	BR733	9428.658	1000	559	108	309	13	11
9	GIGA	GIGA	SIX	9259.221	1000	527	108	340	11	14
10	GIGA	SIX	GIGA	8627.407	1000	540	117	321	9	13
11	GIGA	SIX	SIX	10712.98	1000	544	86	349	11	10
12	SIX	GIGA	GIGA	7332.718	1000	534	101	347	13	5
13	SIX	GIGA	SIX	9838.221	1000	514	119	342	12	13
14	SIX	SIX	GIGA	8972.002	1000	567	88	324	9	12
15	SIX	SIX	SIX	12131.09	1000	542	110	320	12	16
16	BR733	BR733	SIX	10387.13	1000	536	113	331	14	6
17	BR733	SIX	BR733	10360.99	1000	570	120	284	11	15
18	BR733	SIX	SIX	11067.39	1000	541	104	326	17	12
19	SIX	BR733	BR733	9591.424	1000	518	113	349	8	12
20	SIX	BR733	SIX	10590.13	1000	539	106	325	18	12
21	SIX	SIX	BR733	10185.45	1000	537	97	343	12	11

22	GIGA	BR733	SIX	10259.39	1000	575	120	287	13	5
23	GIGA	SIX	BR733	9834.875	1000	554	116	304	19	7
24	BR733	GIGA	SIX	9563.001	1000	537	114	328	12	9
25	BR733	SIX	GIGA	8743.235	1000	526	110	340	12	12
26	SIX	GIGA	BR733	8625.439	1000	548	93	340	10	9
27	SIX	BR733	GIGA	8259.047	1000	551	98	321	16	14

The models predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 5 was the third fastest average response time. Pattern 1, the fastest average response time in this run, was the predicted configuration when RAM usage was set to 150% of physical RAM. The fact that pattern 12 was the second fastest time in this test run is a result of the variability of the simulation. Pattern 12 is a provably inferior deployment than pattern 5. Pattern 5 had 36 more calls to Call 2 and 24 less calls to Call 3 than pattern 12 had in the test run. Call 2 has more than twice the server load than Call 3. Again, the model proposed configuration outperformed most configurations from 10 to 44 percent.

11. Role 3 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one

user of type Role 3 is present and the servers are using memory at their profiled needs.

Table 32: Measured Role3 Max Memory Results, JAVA RMI Experiments.

PAT	SERVER A	SERVER B	SERVER C	ROLE 3	CALLS	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	6776.846	1000	532	115	325	19	9
2	GIGA	GIGA	BR733	8213.157	1000	550	98	327	10	15
3	GIGA	BR733	GIGA	7900.562	1000	554	112	312	11	11
4	GIGA	BR733	BR733	9217.953	1000	553	123	301	14	9
5	BR733	GIGA	GIGA	7267.639	1000	574	100	308	9	9
6	BR733	GIGA	BR733	8519.844	1000	542	105	327	9	17
7	BR733	BR733	GIGA	8232.064	1000	528	130	325	11	6
8	BR733	BR733	BR733	9373.861	1000	540	107	332	11	10
9	GIGA	GIGA	SIX	9463.079	1000	508	149	321	9	13
10	GIGA	SIX	GIGA	8532.983	1000	535	111	329	13	12
11	GIGA	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	7346.219	1000	503	111	352	20	14
13	SIX	GIGA	SIX	error	N/A	N/A	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	error	N/A	N/A	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	10529.611	1000	532	129	313	15	11
17	BR733	SIX	BR733	10123.563	1000	529	120	331	13	7
18	BR733	SIX	SIX	error	N/A	N/A	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	9770.578	1000	528	132	314	15	11
20	SIX	BR733	SIX	error	N/A	N/A	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	error	N/A	N/A	N/A	N/A	N/A	N/A
22	GIGA	BR733	SIX	10193.641	1000	534	122	326	11	7
23	GIGA	SIX	BR733	9804.983	1000	557	91	330	5	17
24	BR733	GIGA	SIX	9617.297	1000	516	129	330	15	10
25	BR733	SIX	GIGA	8865.811	1000	527	118	332	9	14
26	SIX	GIGA	BR733	8860.094	1000	544	118	323	6	9
27	SIX	BR733	GIGA	8328.064	1000	548	104	323	12	13

The model predicted a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. In this

run, the model predicted configuration of pattern 5 was the second fastest average response time. Pattern 1, the fastest average response time in this run, was the predicted configuration when RAM usage was set to 150% of physical RAM. Again, the model proposed configuration outperformed most configurations from 10 to 44 percent and that the recommended patterns were free from failures.

12. Four Concurrent Role 2 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when four concurrent users of type Role 2 are present and the servers are using memory at significantly less than their profiled needs. More detailed information on each individual user is available in the appendix.

Table 33: Measured 4 Role 2 Users Min Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	14603.393
2	GIGA	GIGA	BR733	11746.102
3	GIGA	BR733	GIGA	11711.421
4	GIGA	BR733	BR733	14333.221
5	BR733	GIGA	GIGA	11335.303
6	BR733	GIGA	BR733	11666.615
7	BR733	BR733	GIGA	17066.677
8	BR733	BR733	BR733	21134.671
9	GIGA	GIGA	SIX	12355.078
10	GIGA	SIX	GIGA	14302.569
11	GIGA	SIX	SIX	18378.255

12	SIX	GIGA	GIGA	12035.296
13	SIX	GIGA	SIX	13884.880
14	SIX	SIX	GIGA	20878.541
15	SIX	SIX	SIX	28119.431
16	BR733	BR733	SIX	17406.758
17	BR733	SIX	BR733	15659.077
18	BR733	SIX	SIX	19011.373
19	SIX	BR733	BR733	15652.578
20	SIX	BR733	SIX	15407.319
21	SIX	SIX	BR733	22150.555
22	GIGA	BR733	SIX	11524.385
23	GIGA	SIX	BR733	13739.013
24	BR733	GIGA	SIX	10201.602
25	BR733	SIX	GIGA	14089.308
26	SIX	GIGA	BR733	10544.218
27	SIX	BR733	GIGA	12569.524

The model predicted a configuration of pattern 26 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 26 was the second fastest average response time. Pattern 24, the fastest average response time in this run, was the result of variability in the simulation of users. Both of these patterns have the servers spread across all machines and Server B located on the fastest processor available since Role 2 is a heavy user of Server B. The predicted pattern outperformed most of the other patterns by 10 to 100 percent.

13. Four Concurrent Role 2 Users Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where

tested and the results are listed for the case when four concurrent users of type Role 2 are present and the servers are using memory at their profiled needs. More detailed information on each individual user is available in the appendix.

Table 34: Measured 4 Role 2 Users Max Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	14927.946
2	GIGA	GIGA	BR733	11927.037
3	GIGA	BR733	GIGA	11834.462
4	GIGA	BR733	BR733	14640.246
5	BR733	GIGA	GIGA	11270.985
6	BR733	GIGA	BR733	11347.985
7	BR733	BR733	GIGA	16819.333
8	BR733	BR733	BR733	21089.873
9	GIGA	GIGA	SIX	12299.154
10	GIGA	SIX	GIGA	14122.558
11	GIGA	SIX	SIX	ERROR
12	SIX	GIGA	GIGA	11818.231
13	SIX	GIGA	SIX	ERROR
14	SIX	SIX	GIGA	ERROR
15	SIX	SIX	SIX	ERROR
16	BR733	BR733	SIX	17629.400
17	BR733	SIX	BR733	15442.842
18	BR733	SIX	SIX	ERROR
19	SIX	BR733	BR733	15503.059
20	SIX	BR733	SIX	ERROR
21	SIX	SIX	BR733	ERROR
22	GIGA	BR733	SIX	11484.678
23	GIGA	SIX	BR733	14197.207
24	BR733	GIGA	SIX	10200.213
25	BR733	SIX	GIGA	14054.867
26	SIX	GIGA	BR733	10697.057
27	SIX	BR733	GIGA	12373.780

The model predicted a configuration of pattern 26 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 26 was the second fastest average response time. Pattern 24, the fastest average response time in this run, was the result of variability in the simulation of users. Both of these patterns have the servers spread across all machines and Server B located on the fastest processor available since Role 2 is a heavy user of Server B. The predicted pattern outperformed most of the other patterns by 10 to 100 percent and successfully avoided patterns that lead to system errors.

14. Three Concurrent Role 3 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when three concurrent users of type Role 3 are present and the servers are using memory at significantly less than their profiled needs. More detailed information on each individual user is available in the appendix.

Table 35: Measured 3 Role 3 Users Min Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	15978.641
2	GIGA	GIGA	BR733	13925.953
3	GIGA	BR733	GIGA	13066.211
4	GIGA	BR733	BR733	20415.474

5	BR733	GIGA	GIGA	14614.580
6	BR733	GIGA	BR733	15729.781
7	BR733	BR733	GIGA	13616.035
8	BR733	BR733	BR733	23320.024
9	GIGA	GIGA	SIX	16637.454
10	GIGA	SIX	GIGA	14247.811
11	GIGA	SIX	SIX	25796.666
12	SIX	GIGA	GIGA	14553.052
13	SIX	GIGA	SIX	19029.670
14	SIX	SIX	GIGA	15860.547
15	SIX	SIX	SIX	30349.109
16	BR733	BR733	SIX	18143.637
17	BR733	SIX	BR733	17679.588
18	BR733	SIX	SIX	25890.508
19	SIX	BR733	BR733	20733.072
20	SIX	BR733	SIX	19881.596
21	SIX	SIX	BR733	18053.782
22	GIGA	BR733	SIX	16933.835
23	GIGA	SIX	BR733	15992.606
24	BR733	GIGA	SIX	16031.549
25	BR733	SIX	GIGA	13661.996
26	SIX	GIGA	BR733	13839.297
27	SIX	BR733	GIGA	12488.024

The model predicted a configuration of pattern 27 when RAM was limited to 100% utilization. In this run, the model predicted configuration of pattern 27 was the fastest average response time. The predicted pattern outperformed most of the other patterns by 10 to 150 percent.

15. Three Concurrent Role 3 Users Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when three

concurrent users of type Role 3 are present and the servers are using memory at their profiled needs. More detailed information on each individual user is available in the appendix.

Table 36: Measured 3 Role 3 Users Max Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	ERROR
2	GIGA	GIGA	BR733	14105.719
3	GIGA	BR733	GIGA	13317.196
4	GIGA	BR733	BR733	20540.183
5	BR733	GIGA	GIGA	14471.595
6	BR733	GIGA	BR733	15387.582
7	BR733	BR733	GIGA	13482.313
8	BR733	BR733	BR733	ERROR
9	GIGA	GIGA	SIX	16615.449
10	GIGA	SIX	GIGA	14480.759
11	GIGA	SIX	SIX	ERROR
12	SIX	GIGA	GIGA	14450.947
13	SIX	GIGA	SIX	ERROR
14	SIX	SIX	GIGA	ERROR
15	SIX	SIX	SIX	ERROR
16	BR733	BR733	SIX	17953.958
17	BR733	SIX	BR733	17636.885
18	BR733	SIX	SIX	ERROR
19	SIX	BR733	BR733	20719.032
20	SIX	BR733	SIX	ERROR
21	SIX	SIX	BR733	ERROR
22	GIGA	BR733	SIX	16896.695
23	GIGA	SIX	BR733	15828.615
24	BR733	GIGA	SIX	16190.401
25	BR733	SIX	GIGA	13688.695
26	SIX	GIGA	BR733	13690.884
27	SIX	BR733	GIGA	12481.280

The model predicted a configuration of pattern 27 when RAM was limited to 100% utilization. In this run, the model

predicted configuration of pattern 27 was the fastest average response time. The predicted pattern outperformed most of the other patterns by 10 to 60 percent and successfully avoided patterns that lead to system errors.

16. Twenty Eight Concurrent Role 1 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. The 13 most relevant combinations where tested and the results are listed for the case when twenty eight concurrent users of type Role 1 are present and the servers are using memory at significantly less than their profiled needs.

Table 37: Measured 28 Role 1 Users Min Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	9307.177
2	GIGA	GIGA	BR733	4964.728
3	GIGA	BR733	GIGA	4333.767
4	GIGA	BR733	BR733	3789.347
5	BR733	GIGA	GIGA	7005.968
6	BR733	GIGA	BR733	14435.578
7	BR733	BR733	GIGA	10810.608
22	GIGA	BR733	SIX	3548.850
23	GIGA	SIX	BR733	3014.110
24	BR733	GIGA	SIX	7413.570
25	BR733	SIX	GIGA	6807.109
26	SIX	GIGA	BR733	11117.019
27	SIX	BR733	GIGA	12042.323

The model predicted that pattern 4 would be the optimal deployment. In the experimental test bed, pattern 4 had the

third lowest average response time. Pattern 23 and pattern 22 both outperformed pattern 4 in the test bed. It is interesting to note that these three patterns are the only patterns that had Server A assigned to the fastest machine, GIGA, with no other servers assigned to that machine. The load on Server A is so much higher than the loads on the other servers that the differences in the three deployments is well below the fidelity of the model.

17. Five Concurrent Role 3 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. The 13 most relevant combinations where tested and the results are listed for the case when five concurrent users of type Role 3 are present and the servers are using memory at significantly less than their profiled needs. More detailed information on each individual user is available in the appendix.

Table 38: Measured 5 Role 3 Users Min Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	29292.842
2	GIGA	GIGA	BR733	21508.580
3	GIGA	BR733	GIGA	20086.958
4	GIGA	BR733	BR733	36014.683
5	BR733	GIGA	GIGA	25289.885
6	BR733	GIGA	BR733	25185.096
7	BR733	BR733	GIGA	20293.638
22	GIGA	BR733	SIX	26617.734
23	GIGA	SIX	BR733	23814.951

24	BR733	GIGA	SIX	25853.817
25	BR733	SIX	GIGA	20167.128
26	SIX	GIGA	BR733	21026.894
27	SIX	BR733	GIGA	18015.532

The results of this test are quite interesting. The system was at saturation when there were just three concurrent users. Two more concurrent users of the same usage pattern were added to the load to see how the system responded and to see if there was any significance that could be determined from the test results. It is interesting to note that pattern 27 continued to be the best when more users of the same type were added beyond what the system could theoretically handle.

18. Two Concurrent Role 3 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. The 13 most relevant combinations where tested and the results are listed for the case when two concurrent users of type Role 3 are present and the servers are using memory at significantly less than their profiled needs. More detailed information on each individual user is available in the appendix. The model, ADOA8.3.2, predicts that at 100 percent RAM utilization, that pattern 3 will be optimal.

Table 39: Measured 2 Role 3 Users Min Mem, JAVA RMI Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	10681.153
2	GIGA	GIGA	BR733	10746.357
3	GIGA	BR733	GIGA	10078.811
4	GIGA	BR733	BR733	13987.569
5	BR733	GIGA	GIGA	10255.211
6	BR733	GIGA	BR733	11559.820
7	BR733	BR733	GIGA	10554.952
22	GIGA	BR733	SIX	13197.936
23	GIGA	SIX	BR733	12331.096
24	BR733	GIGA	SIX	12367.663
25	BR733	SIX	GIGA	11114.565
26	SIX	GIGA	BR733	10724.390
27	SIX	BR733	GIGA	10272.231

The results of the test run do in fact show that pattern 3 does have the fastest average response time. It is as much as 25 percent better than other possible deployments that were tested.

B. CONCLUSIONS

The results of the JAVA RMI experimentations lead to some very interesting results. The predictions made by the model were very accurate, leading to good choices for server deployment. However, more striking conclusions are drawn from looking at groups of experiments.

1. Scheduled Re-Deployments

Although the model does a good job of predicting performance for a single point, the true strength of this approach is chaining these points together. By taking

advantage of changes to the system at predictable points in time, we can do better than any single statically assigned server placement.

Table 40: Shift Changes.

PAT	SERV A	SERV B	SERV C	ROLE 1	ROLE 2	ROLE 3	R2 (4)	R3 (3)	R1 (28)
2	GIGA	GIGA	BR733	899.34	5530.33	8266.52	11746.10	13925.95	4964.73
3	GIGA	BR733	GIGA	960.81	6417.17	7802.17	11711.42	13066.21	4333.77
4	GIGA	BR733	BR733	1079.64	6686.38	9124.94	14333.22	20415.47	3789.35
5	BR733	GIGA	GIGA	1140.80	5953.02	7413.34	11335.30	14614.58	7005.97
26	SIX	GIGA	BR733	1355.59	6752.50	8625.44	10544.22	13839.30	11117.11
27	SIX	BR733	GIGA	1306.69	7380.83	8259.05	12569.52	12488.02	12042.34

If we assume that we have a shift schedule that has the following six unique manning requirements over the duration of the schedule, then we can initiate object server re-deployments to coincide with the shift changes. The shaded areas in Table 40 indicate the deployment pattern recommended by the model. The numbers in the matrix are the actual measured values for these deployments.

We are only interested in the six deployment patterns listed in Table 40. If we were to institute a static deployment for our system, then we would be forced to pick just one of the deployment patterns listed above. The system engineer would be forced into some logic that mitigated a worst-case scenario.

However, since we have the ability to reason about different manning schedules, then we can take advantage of

this capability. By allowing the system to adjust the location of its object servers at shift changes, we gain substantial improvements to the system.

By comparing the models recommended deployment pattern versus the other six deployment patterns in Table 40, we can quantify this improvement. By dividing the model predicted patterns measured performance by the measured performance of the other patterns in the same column, we get the performance improvement for each shift. Table 41 below contains these values.

Table 41: Shift improvements.

PAT	SERV A	SERV B	SERV C	ROLE 1	ROLE 2	ROLE 3	R2 (4)	R3 (3)	R1 (28)
2	GIGA	GIGA	BR733	-7%	0%	10%	10%	10%	24%
3	GIGA	BR733	GIGA	0%	14%	5%	10%	4%	13%
4	GIGA	BR733	BR733	11%	17%	18%	26%	39%	0%
5	BR733	GIGA	GIGA	16%	7%	0%	7%	15%	46%
26	SIX	GIGA	BR733	29%	18%	14%	0%	10%	66%
27	SIX	BR733	GIGA	26%	25%	10%	16%	0%	68%

Interesting to note is that we are only comparing deployment patterns that are of high probability of actually being used. Only one entry in the table has a negative value, all other entries have a substantial performance improvement. Clearly from Table 41, any organization with known manning schedules that fluctuate would benefit from this approach.

2. Saturation Testing

Another interesting observation can be ascertained by viewing all of the tests involving Role 3 users. Tests were conducted with 1, 2, 3 and 5 concurrent Role 3 users even though the model was at saturation point with three concurrent Role 3 users.

Table 42: Saturation Testing.

PAT	SERV A	SERV B	SERV C	1 User	2 Users	3 Users	5 Users
3	GIGA	BR733	GIGA	7802.2	10078.8	13066.2	20086.9
5	BR733	GIGA	GIGA	7413.3	10255.2	14614.5	25289.8
27	SIX	BR733	GIGA	8259.1	10272.2	12488.0	18015.5

Again, the shaded area indicates the deployment pattern predicted by the model. At saturation point, and additional load of a similar pattern does not change the deployment pattern. The quantified improvements in performance results are expressed in the values in Table 43.

Table 43: Quantified Improvements, saturation testing.

PAT	SERV A	SERV B	SERV C	1 User	2 Users	3 Users	5 Users
3	GIGA	BR733	GIGA	5%	0%	4%	10%
5	BR733	GIGA	GIGA	0%	2%	14%	29%
27	SIX	BR733	GIGA	10%	2%	0%	0%

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CHAPTER 7

A. JAVA CORBA EXPERIMENTATION RESULTS

We tested the validity of the model presented in Chapter 4 by experimental measurement a second time with a different middleware. A testbed was created with Windows 2000 machines that match the characteristics of the machines in the following tables. Servers were created using JDK 1.3 and CORBA as the middleware. A commercial CORBA implementation, Visibroker 4.5.1, was used in this testbed.

Software to simulate the three different users was also created. The users were simulated with a random choice for button selection that has a uniform distribution similar to the roles. This simulation software was instrumented to measure the actual time the software was blocked waiting for an object server method call to response. All 27 different configurations were established and the average response time for each configuration was measured and recorded. Between each simulation, the testbed machines were rebooted.

All 27 configurations were tested twice. One tested the configuration with the object servers using much less than the stated memory needs. Another tested the configuration with the object servers using all of the stated memory needs. Some configurations strained the

machines memory limits. These configurations resulted in system failures in the test with the object servers using all of the stated memory needs. These system failures are listed as error in the tables of results. It should be noted that Windows 2000 did a much better job of swapping when memory utilization exceeded 100% than a previously tested operating system, Windows NT.

1. Experiment Characteristics

The hardware was purchased for this experiment with the following CPU clock rates and RAM. Each machine had a 10/100Mbps Ethernet card installed. The machines were connected via a Ethernet hub rated for 10/100Mbps. The LED indicators on the Ethernet hub verified the rate of 100Mbps data transfer rate.

Table 44: Machine Profile, JAVA CORBA Experiments.

MACHINE	RAM (bits)	CPU Speed (MHz)
SIX	512,000,000 = 64MB	600
BR733	1,024,000,000 = 128MB	733
GIGA	1,024,000,000 = 128MB	1000

Table 45: Network Speed Profile, JAVA CORBA Experiments.

Machine to Machine Speed (bps)	SIX	BR733	GIGA
SIX	200,000,000	100,000,000	100,000,000
BR733	100,000,000	200,000,000	100,000,000
GIGA	100,000,000	100,000,000	200,000,000

Table 46: Object Server RAM Profile, JAVA CORBA Experiments.

SERVER	RAM Required (bits)
A	352,000,000 = 44MB
B	480,000,000 = 60MB
C	528,000,000 = 66MB

Table 47: Object Server Performance Profile, JAVA CORBA Experiments.

SERVER	Method	CPU time (s)	Average Size of Message (b)
A	1	0.5523	536
A	2	2.5742	104
A	3	1.16165	104
A	4	1.9828	320
B	1	1.59275	536
B	2	3.52555	104
C	1	2.94475	536
C	2	4.5665	104
C	3	0.47065	104

Table 48: Complex Server Profile, JAVA CORBA Experiments.

Complex Method	Exterior Calls
B.2	C.1

Table 49: User Role Profiles, JAVA CORBA Experiments.

ROLE	CALL PATTERN (observation interval is 990 seconds)
Role 1	50 C1.B1 + 1 C1.B2 + 1 C2.B1 + 1 C2.B6
Role 2	10 C1.B1 + 40 C1.B2 + 24 C3.B2
Role 3	50 C2.B5 + 10 C2.B9 + 30 C2.B3 + 1 C2.B2 + 1 C3.B2

Table 50: Application Call Chart, JAVA CORBA Experiments.

Button	Methods Called
C1.B1	A.1
C1.B2	A.2 + B.1
C2.B1	C.1 + C.2
C2.B2	C.3
C2.B3	C.2

C2.B4	C.3
C2.B5	A.1 + *B*.2
C2.B6	*B*.2
C2.B7	A.4
C2.B8	C.3 + A.3
C2.B9	A.1 + A.2 + A.3 + *B*.2
C3.B1	C.1
C3.B2	B.1 + *B*.2
C3.B3	C.2

Table 51: Expanded User Roles, JAVA CORBA Experiments.

ROLE	**Methods Called in Role**
Role 1	50 * (A.1) + 1 * (A.2 + B.1) + 1 * (C.1 + C.2) + 1 * (*B*.2)
Role 2	10 * (A.1) + 40 * (A.2 + B.1) + 24 * (B.1 + *B*.2)
Role 3	50 * (A.1 + *B*.2) + 10 * (A.1 + A.2 + A.3 + *B*.2) + 30 * (C.2) + 1 * (C.3) + 1 * (B.1 + *B*.2)

2. Specializing the Objective Function for Role 1

Role 1 consists of 50 C1.B1 calls, one C1.B2 call, one C2.B1 call, and one C2.B6 call. The first step is to convert all of the button calls into method calls by substituting the values for the calls from Table 4.

50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [*B.2*] =
50 [A.1] + 1 [A.2 + B.1] + 1 [C.1 + C.2] + 1 [B.2 + C.1] =
50 A.1 + A.2 + B.1 + C.1 + C.2 + B.2 + C.1 =
50 A.1 + A.2 + B.1 + B.2 + 2 C.1 + C.2

This leads to the following values for the array R for the optimization equation.

R(A) = 50 [A.1 values for CPU] + 1 [A.2 value for CPU]
= 50 [552.3] + 1 [2574.2]
= 30189.2

R(B) = 1 [B.1 values for CPU] + 1 [B.2 value for CPU]

= 1 [1592.75] + 1 [3525.55]
= 5118.3

R(C) = 2 [C.1 values for CPU] + 1 [C.2 value for CPU]
= 2 [2944.75] + 1 [4566.5]
= 10456.0

There is only one italicized method call prior to substitution, so there is only one network value to deal with.

BITS[B,C] = 1 [*B.2* message in bits]
= 536

3. Specializing the Objective Function for Role 2

Using the same approach, we get the following for Role 2:

R(A) = 108491.0
R(B) = 186549.2
R(C) = 70674.0

There is only one italicized method call prior to substitution, so there is only one network value to deal with. However, it is called 24 times.

BITS[B,C] = 24 [*B.2* message in bits]
= 24 [536]
= 1344

4. Specializing the Objective Function for Role 3

R (A) = 70496.5
R (B) = 216651.3
R (C) = 317095.4
BITS[B,C] = 32696

5. Model Outputs

The optimization model determines the following deployment strategies for the different roles when setting different RAM limits and keeping all other variables the same as in the last example. Solving the optimization problem defined in previous section with the parameter values determined derives these results. The LINGO models for the below results can be seen in Appendix A. These models are listed as ADOA9.1 for Role 1(1 user), ADOA9.2 for Role 2(1 user) and ADOA9.3 for Role 3(1 user). Table 52 lists the results when RAM utilization was set at 1.5 times the physical RAM of the machines.

Table 52: Model Outputs, 150% RAM util., JAVA CORBA Experiments.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	None	None	None
GIGA	A, B, C	A, B, C	A, B, C

The next table used the same LINGO models as the above table, but with the RAM utilization set at 1.0 times the physical RAM of the machines.

Table 53: Model Outputs, 100% RAM util., JAVA CORBA Experiments.

Machine	Role 1 (1 user)	Role 2 (1 user)	Role 3 (1 user)
SIX	None	None	None
BR733	B	C	A
GIGA	A, C	A, B	B, C

Table 54 used LINGO models that represented having multiple concurrent users exercising the object servers. These models are listed in Appendix A as ADOA9.1.28 for Role 1 (28 users), ADOA9.2.4 for Role 2 (4 users), and ADOA9.3.3 for Role 3 (3 users). The RAM utilization was set at 1.0 times the physical RAM of the machines.

Table 54: Model Outputs, Concurrent Users, JAVA CORBA Experiments.

Machine	Role 1 (28 users)	Role 2 (4 users)	Role 3 (3 users)
SIX	None	A	A
BR733	C	C	B
GIGA	A, B	B	C

6. Role 1 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 1 is present and the servers are using memory at significantly less than their profiled needs. Minimal memory in the CORBA tests usually indicated a RAM usage of 6 megabits.

Table 55: Measured Role 1 User Min Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 1	CALLS	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	812.143	1000	954	14	20	12
2	GIGA	GIGA	BR733	906.811	1000	945	19	13	23
3	GIGA	BR733	GIGA	887.391	1000	942	26	14	18

4	GIGA	BR733	BR733	923.566	1000	951	13	14	22
5	BR733	GIGA	GIGA	1046.038	1000	950	15	19	16
6	BR733	GIGA	BR733	1204.761	1000	935	20	19	26
7	BR733	BR733	GIGA	1051.965	1000	953	17	12	18
8	BR733	BR733	BR733	1322.773	1000	929	16	27	28
9	GIGA	GIGA	SIX	1056.029	1000	938	17	22	23
10	GIGA	SIX	GIGA	965.497	1000	938	20	20	22
11	GIGA	SIX	SIX	1166.363	1000	926	32	21	21
12	SIX	GIGA	GIGA	1273.043	1000	939	16	21	24
13	SIX	GIGA	SIX	1315.116	1000	949	20	15	16
14	SIX	SIX	GIGA	1347.371	1000	936	27	23	14
15	SIX	SIX	SIX	1304.101	1000	958	16	15	11
16	BR733	BR733	SIX	1252.422	1000	939	26	18	17
17	BR733	SIX	BR733	1179.988	1000	942	31	13	14
18	BR733	SIX	SIX	1339.852	1000	938	17	21	24
19	SIX	BR733	BR733	1281.810	1000	951	20	14	15
20	SIX	BR733	SIX	1422.636	1000	942	21	21	16
21	SIX	SIX	BR733	1453.406	1000	935	21	24	20
22	GIGA	BR733	SIX	996.187	1000	949	15	19	17
23	GIGA	SIX	BR733	1078.838	1000	938	13	27	22
24	BR733	GIGA	SIX	1327.667	1000	929	20	22	29
25	BR733	SIX	GIGA	1120.733	1000	946	22	13	19
26	SIX	GIGA	BR733	1265.486	1000	952	14	18	16
27	SIX	BR733	GIGA	1269.271	1000	946	13	21	20

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the second fastest average response time in this test. Pattern 1 was the fastest on this run which was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Swapping was not an issue since memory usage was low. More interesting from a software engineering standpoint was the fact that the model proposed a

configuration that outperformed most configurations from 10 to 45 percent.

7. Role 1 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 1 is present and the servers are using memory at their profiled needs.

Table 56: Measured Role 1 User Max Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 1	CALLS	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	833.919	1000	949	18	14	19
2	GIGA	GIGA	BR733	886.201	1000	955	10	19	16
3	GIGA	BR733	GIGA	841.651	1000	955	12	18	15
4	GIGA	BR733	BR733	949.210	1000	947	17	18	18
5	BR733	GIGA	GIGA	1190.435	1000	923	26	22	29
6	BR733	GIGA	BR733	1181.708	1000	940	16	18	26
7	BR733	BR733	GIGA	1093.800	1000	946	22	12	20
8	BR733	BR733	BR733	1169.140	1000	944	23	14	19
9	GIGA	GIGA	SIX	926.091	1000	952	14	13	21
10	GIGA	SIX	GIGA	926.692	1000	943	19	23	15
11	GIGA	SIX	SIX	ERROR	1000	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	1221.178	1000	946	17	7	30
13	SIX	GIGA	SIX	ERROR	1000	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	ERROR	1000	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	ERROR	1000	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	1187.428	1000	954	10	18	18
17	BR733	SIX	BR733	1294.509	1000	935	19	22	24
18	BR733	SIX	SIX	ERROR	1000	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	1329.086	1000	946	18	14	22
20	SIX	BR733	SIX	ERROR	1000	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	ERROR	1000	N/A	N/A	N/A	N/A
22	GIGA	BR733	SIX	1137.129	1000	937	12	24	27

23	GIGA	SIX	BR733	1022.474	1000	940	21	23	16
24	BR733	GIGA	SIX	1258.190	1000	933	26	17	24
25	BR733	SIX	GIGA	1180.871	1000	935	29	21	15
26	SIX	GIGA	BR733	1354.999	1000	937	23	20	20
27	SIX	BR733	GIGA	1202.858	1000	955	17	21	7

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 3 when RAM was limited to 100% utilization. Pattern 3 was the second fastest average response time in this test. Pattern 1 was the fastest on this run, which was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Again, the model proposed a configuration that outperformed most configurations from 10 to 45 percent and successfully avoided any pattern that would result in a system error.

8. Role 2 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 2 is present and the servers are using memory at significantly less than their profiled needs.

Table 57: Measured Role 2 User Min Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 2	CALLS	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	5002.236	1000	113	570	317
2	GIGA	GIGA	BR733	5456.493	1000	111	550	339
3	GIGA	BR733	GIGA	5927.456	1000	126	545	329

4	GIGA	BR733	BR733	6229.919	1000	130	547	323
5	BR733	GIGA	GIGA	5545.803	1000	125	544	331
6	BR733	GIGA	BR733	5720.117	1000	147	543	310
7	BR733	BR733	GIGA	6192.044	1000	156	545	299
8	BR733	BR733	BR733	6790.937	1000	151	503	346
9	GIGA	GIGA	SIX	5478.762	1000	140	550	310
10	GIGA	SIX	GIGA	6531.051	1000	142	542	316
11	GIGA	SIX	SIX	7108.123	1000	135	558	307
12	SIX	GIGA	GIGA	5999.068	1000	124	541	335
13	SIX	GIGA	SIX	6419.965	1000	145	551	304
14	SIX	SIX	GIGA	7638.231	1000	126	551	323
15	SIX	SIX	SIX	8200.049	1000	136	543	321
16	BR733	BR733	SIX	7047.470	1000	137	535	328
17	BR733	SIX	BR733	7354.101	1000	157	523	320
18	BR733	SIX	SIX	7652.154	1000	142	547	311
19	SIX	BR733	BR733	7202.273	1000	135	538	327
20	SIX	BR733	SIX	7446.842	1000	147	521	332
21	SIX	SIX	BR733	8159.049	1000	107	559	334
22	GIGA	BR733	SIX	6292.001	1000	132	574	294
23	GIGA	SIX	BR733	7111.723	1000	131	527	342
24	BR733	GIGA	SIX	6045.871	1000	142	541	317
25	BR733	SIX	GIGA	7323.822	1000	125	528	347
26	SIX	GIGA	BR733	6266.378	1000	145	516	339
27	SIX	BR733	GIGA	6803.448	1000	139	540	321

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. Pattern 2 was the second fastest average response time in this test. Pattern 1 was the fastest on this run, which was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Swapping was not an issue since memory usage was low. More interesting from a software engineering standpoint was the fact that the model proposed a

configuration that outperformed most configurations from 10 to 30 percent.

9. Role 2 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 2 is present and the servers are using memory at their profiled needs.

Table 58: Measured Role 2 User Max Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 2	CALLS	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	4783.474	1000	136	582	282
2	GIGA	GIGA	BR733	5451.146	1000	118	539	343
3	GIGA	BR733	GIGA	5941.058	1000	128	539	333
4	GIGA	BR733	BR733	6292.611	1000	139	524	337
5	BR733	GIGA	GIGA	5576.746	1000	127	529	344
6	BR733	GIGA	BR733	5863.812	1000	134	537	329
7	BR733	BR733	GIGA	6379.604	1000	127	564	309
8	BR733	BR733	BR733	6769.637	1000	128	552	320
9	GIGA	GIGA	SIX	5432.729	1000	143	554	303
10	GIGA	SIX	GIGA	6603.610	1000	147	523	330
11	GIGA	SIX	SIX	ERROR	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	5905.165	1000	141	532	327
13	SIX	GIGA	SIX	ERROR	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	ERROR	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	ERROR	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	7068.025	1000	139	531	330
17	BR733	SIX	BR733	7719.824	1000	123	530	347
18	BR733	SIX	SIX	ERROR	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	7262.279	1000	128	542	330
20	SIX	BR733	SIX	ERROR	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	ERROR	N/A	N/A	N/A	N/A

22	GIGA	BR733	SIX	6573.312	1000	136	529	335
23	GIGA	SIX	BR733	6905.221	1000	138	545	317
24	BR733	GIGA	SIX	5903.918	1000	150	553	297
25	BR733	SIX	GIGA	7082.142	1000	145	535	320
26	SIX	GIGA	BR733	6222.291	1000	135	555	310
27	SIX	BR733	GIGA	6843.850	1000	144	518	338

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 2 when RAM was limited to 100% utilization. Pattern 2 was the third fastest average response time in this test. Pattern 9 was the second fastest average response time in this test. Pattern 9 is provably inferior to pattern 2 and is a result of the variability of the simulation. Pattern 1 was the fastest on this run, which was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Again, the model proposed a configuration that outperformed most configurations from 10 to 30 percent and successfully avoided any pattern that would result in a system error.

10. Role 3 Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 3 is present and the servers are using memory at significantly less than their profiled needs.

Table 59: Measured Role 3 User Min Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 3	CALLS	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	6501.536	1000	549	95	335	11	10
2	GIGA	GIGA	BR733	7779.399	1000	543	103	333	13	8
3	GIGA	BR733	GIGA	7425.086	1000	542	108	331	9	10
4	GIGA	BR733	BR733	8763.950	1000	531	130	324	11	4
5	BR733	GIGA	GIGA	6847.776	1000	547	107	325	11	10
6	BR733	GIGA	BR733	8059.274	1000	550	102	326	13	9
7	BR733	BR733	GIGA	7912.398	1000	561	118	296	8	17
8	BR733	BR733	BR733	8991.030	1000	518	121	331	15	15
9	GIGA	GIGA	SIX	8787.180	1000	533	113	328	20	6
10	GIGA	SIX	GIGA	8018.179	1000	523	110	341	16	10
11	GIGA	SIX	SIX	10311.860	1000	521	106	353	10	10
12	SIX	GIGA	GIGA	7063.876	1000	529	107	339	8	17
13	SIX	GIGA	SIX	9364.904	1000	537	108	334	9	12
14	SIX	SIX	GIGA	8862.786	1000	557	120	300	10	13
15	SIX	SIX	SIX	10956.668	1000	542	111	324	11	12
16	BR733	BR733	SIX	9874.777	1000	532	106	331	20	11
17	BR733	SIX	BR733	9555.933	1000	533	99	348	8	12
18	BR733	SIX	SIX	10634.238	1000	516	117	341	15	11
19	SIX	BR733	BR733	9134.159	1000	554	96	326	10	14
20	SIX	BR733	SIX	10156.106	1000	555	95	332	7	11
21	SIX	SIX	BR733	9949.506	1000	532	121	328	12	7
22	GIGA	BR733	SIX	9732.875	1000	572	102	304	12	10
23	GIGA	SIX	BR733	9301.973	1000	528	106	344	11	11
24	BR733	GIGA	SIX	9097.136	1000	552	104	321	13	10
25	BR733	SIX	GIGA	8297.218	1000	552	90	335	9	14
26	SIX	GIGA	BR733	8438.224	1000	526	121	328	8	17
27	SIX	BR733	GIGA	7945.419	1000	539	116	321	18	6

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. Pattern 5 was the second fastest average response time in this test. Pattern 1 was the fastest on this run, which was the predicted configuration when RAM usage was set to 150% of physical RAM

in the model. Again, the model proposed a configuration that outperformed most configurations from 10 to 38 percent.

11. Role 3 Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when only one user of type Role 3 is present and the servers are using memory at their profiled needs.

Table 60: Measured Role 3 User Max Mem, JAVA CORBA Experiment.

PAT	SERVER A	SERVER B	SERVER C	ROLE 3	CALLS	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	6623.287	1000	508	122	347	5	18
2	GIGA	GIGA	BR733	7832.773	1000	555	105	321	10	9
3	GIGA	BR733	GIGA	7254.935	1000	542	87	344	15	12
4	GIGA	BR733	BR733	8641.256	1000	511	108	356	10	15
5	BR733	GIGA	GIGA	6895.644	1000	572	103	299	12	14
6	BR733	GIGA	BR733	8053.885	1000	536	100	335	13	16
7	BR733	BR733	GIGA	7645.818	1000	557	97	331	9	6
8	BR733	BR733	BR733	8926.149	1000	531	103	334	12	20
9	GIGA	GIGA	SIX	8924.096	1000	550	117	311	11	11
10	GIGA	SIX	GIGA	8045.043	1000	549	95	342	7	7
11	GIGA	SIX	SIX	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
12	SIX	GIGA	GIGA	7079.384	1000	526	109	339	8	18
13	SIX	GIGA	SIX	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
14	SIX	SIX	GIGA	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
15	SIX	SIX	SIX	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
16	BR733	BR733	SIX	10035.334	1000	557	102	321	8	12
17	BR733	SIX	BR733	9565.464	1000	544	102	330	17	7
18	BR733	SIX	SIX	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
19	SIX	BR733	BR733	9254.666	1000	521	123	332	13	11
20	SIX	BR733	SIX	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
21	SIX	SIX	BR733	ERROR	N/A	N/A	N/A	N/A	N/A	N/A
22	GIGA	BR733	SIX	9698.045	1000	538	107	332	12	11

23	GIGA	SIX	BR733	9311.344	1000	527	104	333	19	17
24	BR733	GIGA	SIX	9081.706	1000	532	107	331	14	16
25	BR733	SIX	GIGA	8279.541	1000	537	101	333	19	10
26	SIX	GIGA	BR733	8187.599	1000	519	99	357	14	11
27	SIX	BR733	GIGA	8063.269	1000	542	121	314	12	11

The model chose a configuration of pattern 1 when RAM was set at 150% utilization and a configuration of pattern 5 when RAM was limited to 100% utilization. Pattern 5 was the second fastest average response time in this test. Pattern 1 was the fastest on this run, which was the predicted configuration when RAM usage was set to 150% of physical RAM in the model. Again, the model proposed a configuration that outperformed most configurations from 10 to 38 percent and successfully avoided any pattern that would result in a system error.

12. Three Concurrent Role 3 Users Minimal Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when three concurrent users of type Role 3 are present and the servers are using memory at significantly less than their profiled needs. Detailed information about the individual characteristics of each user can be found in the appendix.

Table 61: Measured 3 Role 3 Users Min Mem, JAVA CORBA Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	15877.702
2	GIGA	GIGA	BR733	13886.899
3	GIGA	BR733	GIGA	12654.546
4	GIGA	BR733	BR733	19894.807
5	BR733	GIGA	GIGA	14149.033
6	BR733	GIGA	BR733	15326.239
7	BR733	BR733	GIGA	13277.221
8	BR733	BR733	BR733	22593.446
9	GIGA	GIGA	SIX	16043.708
10	GIGA	SIX	GIGA	14039.925
11	GIGA	SIX	SIX	25316.147
12	SIX	GIGA	GIGA	14369.772
13	SIX	GIGA	SIX	18758.802
14	SIX	SIX	GIGA	15383.009
15	SIX	SIX	SIX	28872.999
16	BR733	BR733	SIX	18000.066
17	BR733	SIX	BR733	17391.755
18	BR733	SIX	SIX	25055.715
19	SIX	BR733	BR733	20333.473
20	SIX	BR733	SIX	19321.708
21	SIX	SIX	BR733	17489.952
22	GIGA	BR733	SIX	16853.736
23	GIGA	SIX	BR733	15620.748
24	BR733	GIGA	SIX	15732.665
25	BR733	SIX	GIGA	13635.231
26	SIX	GIGA	BR733	13652.896
27	SIX	BR733	GIGA	12076.358

The model chose a configuration of pattern 27 when RAM was limited to 100% utilization. Pattern 27 was the fastest average response time in this test. Again, the model proposed a configuration that outperformed most configurations from 10 to 100 percent.

13. Three Concurrent Role 3 Users Maximum Memory

The following table is actual measured results from a test bed that implemented servers with the characteristics of the given example. All 27 possible combinations where tested and the results are listed for the case when three concurrent users of type Role 3 are present and the servers are using memory at their profiled needs. Detailed information about the individual characteristics of each user can be found in the appendix.

Table 62: Measured 3 Role 3 Users Max Mem, JAVA CORBA Experiment.

PATTERN	SERVER A	SERVER B	SERVER C	AVERAGE
1	GIGA	GIGA	GIGA	15770.141
2	GIGA	GIGA	BR733	13781.662
3	GIGA	BR733	GIGA	12534.210
4	GIGA	BR733	BR733	19937.813
5	BR733	GIGA	GIGA	14171.030
6	BR733	GIGA	BR733	15056.097
7	BR733	BR733	GIGA	12868.518
8	BR733	BR733	BR733	22810.510
9	GIGA	GIGA	SIX	16314.067
10	GIGA	SIX	GIGA	13853.769
11	GIGA	SIX	SIX	ERROR
12	SIX	GIGA	GIGA	14173.931
13	SIX	GIGA	SIX	ERROR
14	SIX	SIX	GIGA	ERROR
15	SIX	SIX	SIX	ERROR
16	BR733	BR733	SIX	17857.033
17	BR733	SIX	BR733	17155.762
18	BR733	SIX	SIX	ERROR
19	SIX	BR733	BR733	20096.231
20	SIX	BR733	SIX	ERROR
21	SIX	SIX	BR733	ERROR
22	GIGA	BR733	SIX	16813.626
23	GIGA	SIX	BR733	15423.078

24	BR733	GIGA	SIX	15895.970
25	BR733	SIX	GIGA	13285.579
26	SIX	GIGA	BR733	13477.260
27	SIX	BR733	GIGA	12011.614

The model chose a configuration of pattern 27 when RAM was limited to 100% utilization. Pattern 27 was the fastest average response time in this test. Again, the model proposed a configuration that outperformed most configurations from 10 to 70 percent and successfully avoided any pattern that would result in a system error.

B. CONCLUSIONS

The results of the JAVA CORBA experimentations reinforced the conclusions presented in the previous chapter. The predictions made by the model were very accurate, leading to good choices for server deployment. However, more striking conclusions are drawn from looking at groups of experiments.

1. Scheduled Re-Deployments

Although the model does a good job of predicting performance for a single point, the true strength of this approach is chaining these points together. By taking advantage of changes to the system at predictable points in time, we can do better than any single statically assigned server placement.

Table 63: Shift Changes, CORBA experiments.

PAT	SERV A	SERV B	SERV C	ROLE 1	ROLE 2	ROLE 3	R3 (3)
2	GIGA	GIGA	BR733	906.81	5456.49	7779.40	13886.90
3	GIGA	BR733	GIGA	887.39	5927.46	7425.09	12654.55
5	BR733	GIGA	GIGA	1046.04	5545.80	6847.78	14149.03
27	SIX	BR733	GIGA	1269.27	6803.45	7945.42	12076.36

If we assume that we have a shift schedule that has the following four unique manning requirements over the duration of the schedule, then we can initiate object server re-deployments to coincide with the shift changes. The shaded areas in Table 63 indicate the deployment pattern recommended by the model. The numbers in the matrix are the actual measured values for these deployments.

We are only interested in the four deployment patterns listed in Table 63. If we were to institute a static deployment for our system, then we would be forced to pick just one of the deployment patterns listed above. The system engineer would be forced into some logic that mitigated a worst-case scenario.

However, since we have the ability to reason about different manning schedules, then we can take advantage of this capability. By allowing the system to adjust the location of its object servers at shift changes, we gain substantial improvements to the system.

By comparing the models recommended deployment pattern versus the other four deployment patterns in Table 63, we can quantify this improvement. By dividing the model predicted patterns measured performance by the measured performance of the other patterns in the same column, we get the performance improvement for each shift. Table 64 below contains these values.

Table 64: Quantifiable Improvements in Shift Changes.

PAT	SERV A	SERV B	SERV C	ROLE 1	ROLE 2	ROLE 3	R3 (3)
2	GIGA	GIGA	BR733	2%	0%	12%	13%
3	GIGA	BR733	GIGA	0%	8%	8%	5%
5	BR733	GIGA	GIGA	15%	2%	0%	15%
27	SIX	BR733	GIGA	30%	20%	14%	0%

Interesting to note is that we are only comparing deployment patterns that are of high probability of actually being used. None of the entries in the table have negative values, seven of the twelve other entries have a double digit performance improvement. Clearly from Table 64, any organization with known manning schedules that fluctuate would benefit from this approach.

2. Middleware Independence

Since the results of the JAVA CORBA experiments reflect the same results as the JAVA CORBA experiments, it would lead to the conclusion that this approach is independent of middleware implementation. In the future, testing of

experiments with a mixed bag of middleware implementations in one system would be useful.

VIII. CHAPTER 8

A. PROFILING

With any tool that is to be used by software developers, if the information needed to feed the tool is not readily available, then the tool's usefulness suffers. Many tool designers fail to understand this point. The approach detailed in this dissertation always had this goal in mind during its development.

Not all profiles are required to get meaningful results. The only required profiles are the hardware profiles of the machines and the profiles of the object servers. With just these profiles, the system provides results. However, when client software profiles and user profiles are added, then the strength of the tool is realized in the form of more accurate results.

Since this approach is abstracted above any middleware or implementation language, there isn't a given methodology or toolset for gathering these profiles. Instead, I will enumerate different approaches for profiling different areas and give examples in certain cases. I will start with the profiles that are required. These also happen to be the easiest to collect.

1. Hardware Profiles

Hardware profiles can be collected with manual or automated methods. The manual method requires the user to individually collect the profiles for each machine. The required information may be obtained from purchasing information or collected from the machine itself. When collecting the information from the machine, it may be labeled on the machine or may be queried while the machine is running. The approaches vary and differ depending on the operating system of the machine.

Automated systems also exist for profiling a network of computers. Commercially available tools like SolarWinds.net's Network Management Tools are too numerous to list. These systems usually consist of software that runs on a machine connected to the network and actively sniff out other hardware. The detail of the information obtained is dependent on the tool.

2. Server Profile

Profiling object servers is a bit more involved than hardware profiles, but still not complicated. The key here is to make sure that all values are normalized. To normalize the values for your object server profile, it is preferable to collect all the values by running the object server timing tests on the same machine. However, if this isn't possible, then the object server timing values must be

normalized after collection. This can be accomplished by multiplying all data collected by a ratio of the machines speed and a set machine speed.

To collect object server profiles, test software is usually required. This test software consists of a call to all exposed methods of all classes while collecting metrics on each call. An example of this test software for the object servers in this dissertation can be found in Appendix B.

Creating this test software by hand is simple enough when the number of classes and methods is relatively small. However, if this isn't the case, then a more automated approach is desirable. If the UML model of the object servers is available, then a tool like Quava can be used to generate the test software. Quava can generate the test software for object servers, even if the object server wasn't created with Quava [32].

3. Client Application Profiles

There are numerous ways to collect the profiles of the applications involved. If the source code is available, then the code can be visually inspected, parsed, or instrumented to produce the application profiles. For parsing or instrumenting the code, there exists numerous tools and parsing languages to accomplish this task. The

choice of tool will depend on the implementation language of the application and your preference of tools.

If the source code contains conditionals, loops or recursion, then had choices need to be made. The most common approaches are to either take the average or the maximum number of times the call can be made. Both approaches have merit.

Since the client applications for the examples in this paper were implemented in JAVA and my preference was to instrument the application code to create their profile, the technology I decided on was Aspect-Oriented Programming, which allows for simple rules to be written that crosscut over the entire application. Thus, very large applications can be instrumented rather efficiently. The tool used was AspectJ, version 0.8. The software for the clients and the aspect can be found in Appendix B. The client software is Client1.java, Client2.java and Client3.java. The aspect software is Profile.java.

If the application software is not available, then other methods must be used. These involve registering events external to the application and mapping them back to the application. In CORBA object servers, interceptors can be used to log these events. In EEJB containers, the same mechanism is available. If the source code is available for the object servers, then they can be instrumented to provide

these logs. The profiles can even be obtained at a lower level, by monitoring network packet traffic. David Luckham's at Stanford on complex event processing is one such approach [10,11,12].

4. User Profiles

If application software is available, then the applications can be instrumented to gather the interactions of the users with the applications. The methods available are similar to the ones mentioned in the application profile section.

Some Graphical User Interface (GUI) tools offer options to track user actions. Both X-Windows and Microsoft Windows have commercial tools that will track user actions. These tools were originally designed to be Big Brother watching every move an employee makes by logging their interactions with their computer. However, they can server a more useful service if their data ultimately used to benefit the employee by speeding up the system with which he interacts.

Of course, visual observation of users and manually logging their interaction will work. All that is needed is paper and pen. A matrix of possible user interacts can be made prior to an exercise or shift. The user or an independent third party can log the frequency of interactions.

B. CONCLUSIONS

There are many ways to collect the required data, but the key is finding the most accurate and automated approach. Ultimately, a cost-benefit analysis will have to be made on whether or not to implement this approach on a host system. The collection of these profiles will be a primary source of the cost associated with its implementation.

IX. CHAPTER 9

A. FUTURE REFINEMENTS TO THE MODEL

The model presented in this dissertation is robust enough to demonstrate the validity of the approach. However, there are multiple ways that the model could be improved.

1. Weights

Weights could be added at various places to allow for more critical operations to have higher significance. There are basically four levels that weights could be added to the methodology.

Weights could be added at the user interaction level. Basically, weights could be added to each interaction that a user could have with the system. At this level, we would see formulas such as WEIGHT[X]* APPLICATION[WORD, SAVE] and WEIGHT[Y] * APPLICATION[EXCEL, EXIT].

Weights could be added at the application level. This would mean that all interactions initiated from a particular application would carry more weight than interactions initiated from another application. These formulas would have the form of WEIGHT[X] * APPLICATION[WORD] and WEIGHT[X] * APPLICATION[EXCEL].

Another place that weights could be inserted is more in line with the object-oriented theme of this dissertation. Each method call in each object server could be assigned a weight. This would have the form WEIGHT[X] * A.M1 and WEIGHT[Y] * A.M2.

The last place that weights could be added is to assign a weight to each object server. In essence, one would say that object server A is more important than object server B. This adjustment could be made to the objective function directly, whereas the other weighting would be internal to the evaluation of the load value of each server. This would adjust the first term of the objective function to:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * W_n * R_n * S_{norm}}{S_m} \right]$$

All of these weighting changes could be made separately on in some combination concurrently. The important thing to remember is the impact it will have on the optimization. All weights must be positive. Weights with values less than one could lead to queuing delay that is not evaluated. Weights greater than one will lead to CPU slack time, but this is the safest approach.

2. Queuing Delay

The model could reason better about deployments if it knew how queuing delay was affecting the system. When more

users are present and actively using an object server, then there is a higher probability of being delayed in the queue prior to having a request being serviced. The probability of a queuing delay is directly related to the utilization of the CPU. A term that added load to the objective function when there were more users and CPU utilization was high could be added. Conversely, a term, that limited CPU utilization as more users were actively engaged with the system, could be added to the CPU constraint.

Classical queue theory makes many assumptions about the arrival rate and service time of tasks. These assumptions may not be valid when modeling users that exhibit certain behavioral habits. Many users that interact with reactive systems generally fall into a looping model of pushing a button, waiting for the reply, then observing the results. When this model of user behavior is assumed, then regardless of the arrival rate or service time, the queue length will always be bounded by the number of users currently engaged with the system.

For systems that serve large number of users with unscheduled usage patterns like those systems that are common to the Internet, then this observation would not be of value. However, in systems where the total number of scheduled users is known ahead of time, then this observation is much more meaningful.

The addition of logic to evaluate queuing delay would have the added benefit of removing the problem of not reasoning correctly about a special case for optimization. If all of the machines have equal RAM and CPU speed and there is no inter-server communication, then the current objective function doesn’t guarantee the correct deployment. In this case, each server should run on a separate machine to minimize the effect of queuing delay. Queuing delay logic would remove this special case.

3. Automated RAM limits

The model could reason better about deployments if it knew how each machine actually responded when the resources of the machine were being depleted. Since we do not have a function that accurately depicts the effect of RAM utilization on processing speed, we could substitute functions that bound the upper and lower limits of the function. These bounding functions could then be incorporated into the model to allow the model to determine how RAM will be utilized.

The model ADOA4 is an illustration of this change. In the ADOA4 model, the RAM limit was removed. The model was allowed to use up to twice the amount of memory on a machine than there was RAM to allow for virtual memory. The function used to increase processing when RAM utilization rises was simply to multiply by 3 raised to the power of the

RAM utilization percentage. This is a safe choice for an upper bound to the actual function. The results of the model were the following deployment.

Table 65: Model Outputs, Automated RAM Function.

MACHINE	SERVER	RAM Utilization
W	None	0.00
X	None	0.00
Y	D	0.3437500
Z	A, B, C	0.5585938

Of course, the problem is finding out what the actual function may be and finding functions that bound the actual function without over exaggerating.

Another problem with introducing an automated RAM function is the variations that different operating systems have in dealing with swapping virtual memory to physical memory. Each operating system could have different RAM functions. When a heterogeneous environment is being modeled the objective function would have to change considerably.

4. Asymmetric Communications

The model is easily changed to handle asymmetric communications by replacing the function used to compute Q.

By changing $Q_{ij} = \left(1 + \sum_{m=0}^{M} a_{im} * a_{jm}\right) * L$, where L is the LAN speed, to $Q_{ij} = \sum_{m=0}^{M} \sum_{m'=0}^{M} a_{im} * a_{jm'} * P_{mm'}$, where P is a matrix of values, then the model can handle almost any kind of communications configuration.

It should be noted that when the network speed is a constant regardless of the location of the servers, then the second term of the objective function serves no real purpose since it becomes a constant and can be eliminated. In this case, the objective function becomes linear. Since most middleware vendors take care to implement efficient code, the point is irrelevant.

5. Unreachable Deployments

There are some deployment patterns that cannot be reached because of the CPU and RAM limits imposed on the model. In a three machine, three-server model there are 27 different possible deployment patterns. Of these 27 possible deployment patterns, some of them may not be available because of the RAM and CPU limits.

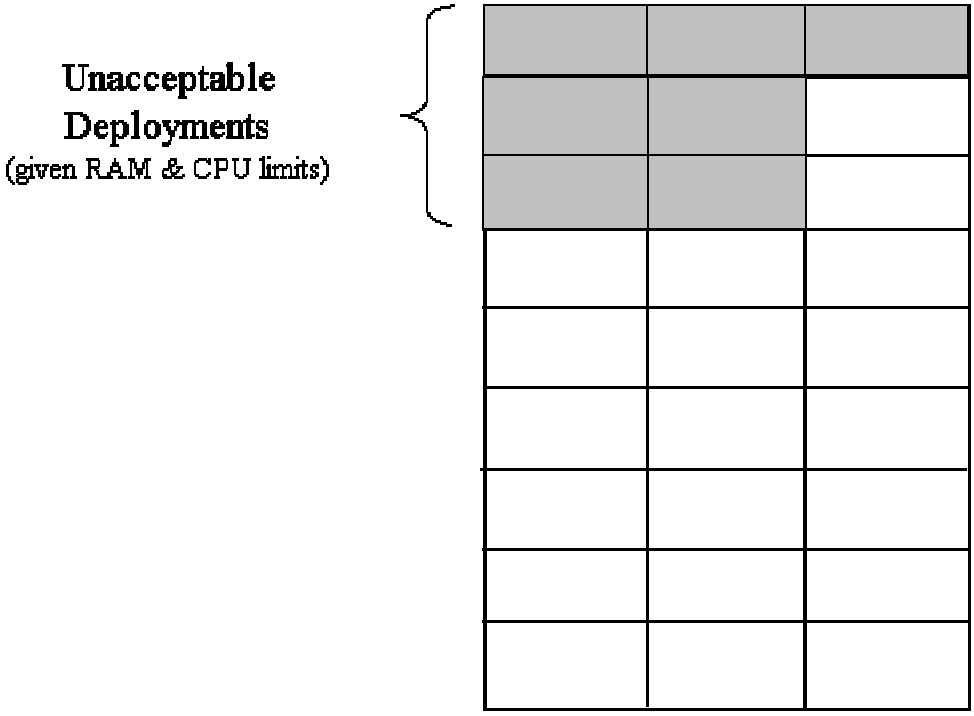

Figure 6: Unacceptable Deployments.

If these unreachable deployments could be removed from consideration from the model ahead of time, then the processing speed of the model could be improved. This becomes more important when the size of the model increases.

6. Provably Inferior Deployments

Besides deployments that cannot be reached because of the constraints placed on the CPU and RAM utilization, there are other deployments that can sometimes be dismissed. In general, a faster machine should never be void of a server assignment while a slower machine possesses such an assignment. However, this is not entirely the case.

When a faster machine has greater or equal amounts of RAM than a slower machine, then it should never be the case that the faster machine is void of a server assignment while the slower machine is assigned a server. This logic could be used to lower the number of deployment possibilities that must be reasoned about, thus decreasing the computational time of the model.

7. Optimal Zone

Because of errors in data used, there may be zones of deployments that are within a few standard deviations of the optimal solution. This zone is called the optimal zone. In the figure below, the bull's-eye area represents the data collected for model to compute. The inner circle of the area represents the area covered if the data collected is within one standard deviation. The middle ring represents the area covered within two standard deviations and the outer ring represents the area covered within three standard deviations.

Again the boxes in the grid represent possible deployments for the servers. The figure is a visual representation of the possibility that errors in the data could lead to ambiguity in the solution space. With sufficient error in the data, any of the shaded boxes in the optimal zone could be the correct deployment.

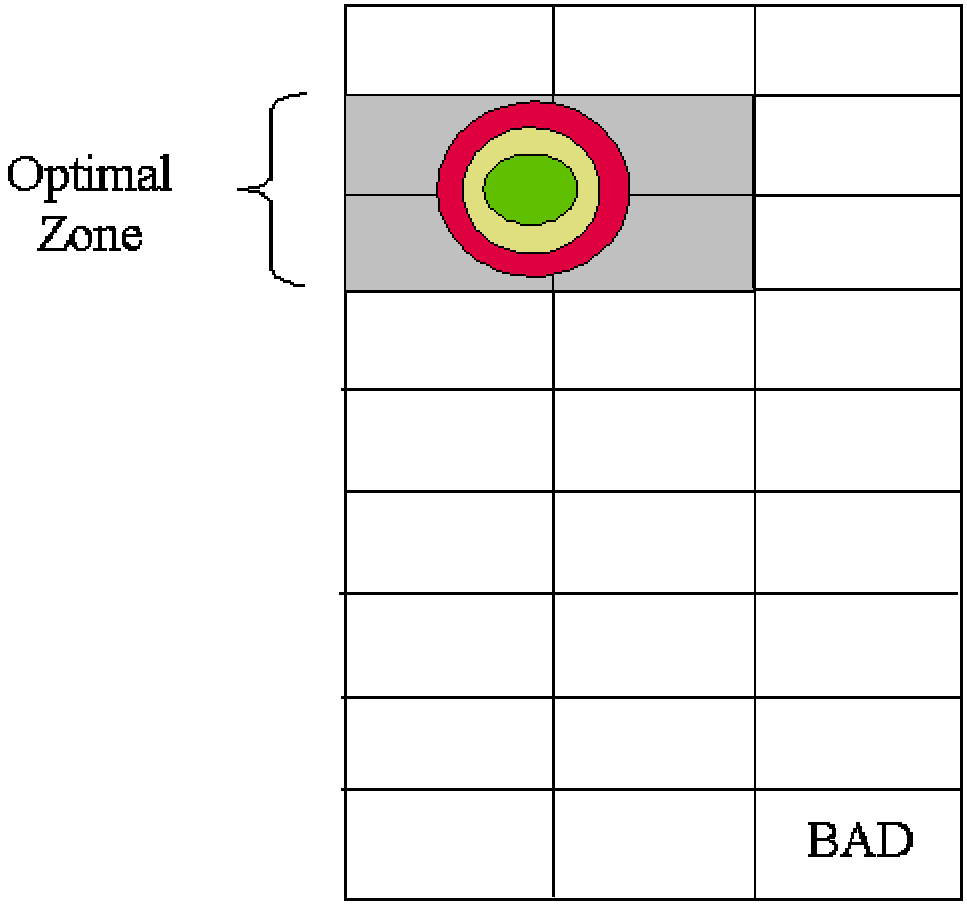

Figure 7: Optimal Zone.

The interesting question is whether it is wise to move from box in the optimal zone to another box in the optimal zone given the fidelity of the model and data.

8. Instance Distribution

A possible enhancement is to add a strategy for splitting the object servers for a given class. This strategy involves using a hash function to partition the population of a class into disjoint buckets, and to have a

separate object server for each bucket. This approach is useful when the load on an object server exceeds the capacity of the fastest available machine. The strategy depends on the assumption that each object in the class is an independent entity. The assumption is a good one if there are no methods that require more than one argument of the class type (including the “self” object).

9. Enterprise Resource Planning (ERP)

An additional capability that can be added to the model is in the area of enterprise resource planning. By adding a table of possible upgrades like adding additional memory, upgrading the CPU, or adding an additional machine along with all the expenses involved in each upgrade, the model could reason as to which upgrades would give the most performance for the least amount of money.

The system could also be used to estimate how many concurrent users the system could support. This would allow the system engineer to reason about scalability issues of the deployed system.

10. Code Generation from UML and ADL

Since this methodology is driven at the class level, it can easily be combined with a tool like Quava and an Architectural Description Language (ADL) to produce multiple servers from a large UML class diagram [32]. In many cases, a UML class diagram may contain hundreds of classes. A

single server generated from this class diagram may be poor performing. The classic solution to this problem today is to either buy faster hardware or create multiple copies of the server and spread the request across multiple machines.

If faster hardware isn’t available and if the objects in the server have state, then a better solution may be to decompose the server into smaller servers and spread these smaller servers across multiple machines. This task was difficult in the past. However, with automated server generators like Quava, this task is much simpler. The logic of the methodology described in this paper could be added to a system like Quava and used to generate multiple object servers from a single UML class diagram based on a given ADL.

THIS PAGE INTENTIONALLY LEFT BLANK

X. CHAPTER 10

A. CONCLUSIONS

1. Model Performance

The model's performance was quite impressive in all of the scenarios tested. It consistently predicted a top performing deployment pattern. As a single static deployment tool, the model has value. Combined with the concept of changing usage patterns with predictable points in time, the value of scheduling deployment changes has even greater value.

By looking at only the deployment patterns that the model has given for different scenarios, we can see how the ability to change server deployments at shift changes will increase the overall performance of a system. A tool that synchronizes the shifting of server deployments at given time could be easily implemented.

2. Targeted Behavior

User profiles can be targeted to the actual behavior that is most critical to his job. Extraneous tasks can be ignored when profiling a user to give greater importance to a desired task. The person profiling the user has total control over these options.

For example, say a system supports ticket agents. The tasks performed by a ticket agent include ticket orders, timecard entry, web browsing, etc. We can profile the ticket agent only when taking a ticket order. Let's say that a shift has 5 ticket agents assigned to the system. If we optimize the system for the targeted user profile with 5 agents, then the system will give the quickest response when 5 simulations orders are being executed. The system may not be optimal for when the ticket agents are doing other tasks, but the act of taking ticket orders was deemed of primary importance.

In a dynamic system by past performance, this ability is not available. If the 5 agents had been surfing the web for a few hours prior to 5 customers calling with ticket orders, then the system would have been tuned for the web surfing task. Worse, a dynamic redeployment may occur during the taking of the ticket orders, causing further delays.

3. Accuracy of Information

The accuracy of the data used as input to the model is very important to the output of the model. Knowing this, it is very important to be as accurate as possible when collecting the data.

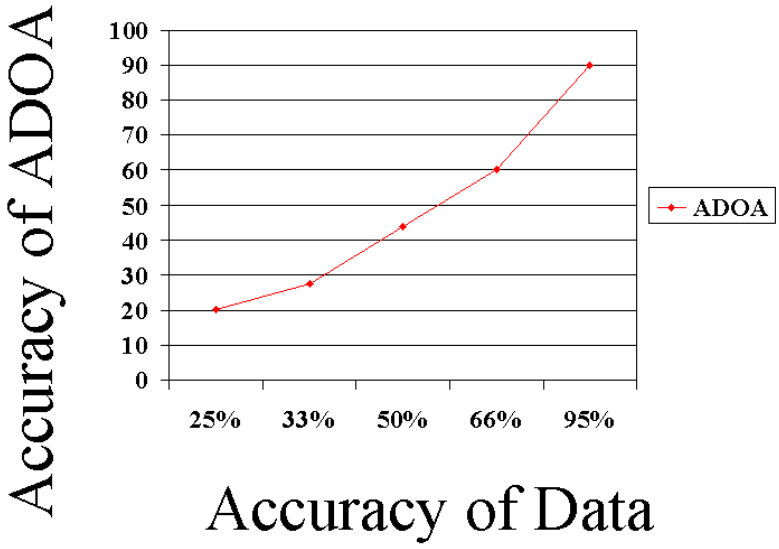

Figure 8: Accuracy of Input Information.

The usage patterns will be the most difficult to accurately predict. Take the example of a simple two server, two-machine architecture to support a military operation. Both machines have the same amount of RAM. Lets say that one server handles track data and another server handles logistics data. Now let's say that we have different missions to perform. These missions may be a high intensity conflict with multiple tracks [Persian Gulf War], a low intensity conflict with few tracks [Somalia], and a humanitarian relief project [Bangladesh Floods]. The high intensity conflict should have the track server running on the fastest machine and the logistics server running on the slower machine, but if the information used to calculate the

optimal deployment is inaccurate, then the deployment may have both servers running on the same machine as depicted below. The deployment labeled BAD is having the two servers running on the slowest machine. This should never happen, regardless of the missions being conducted.

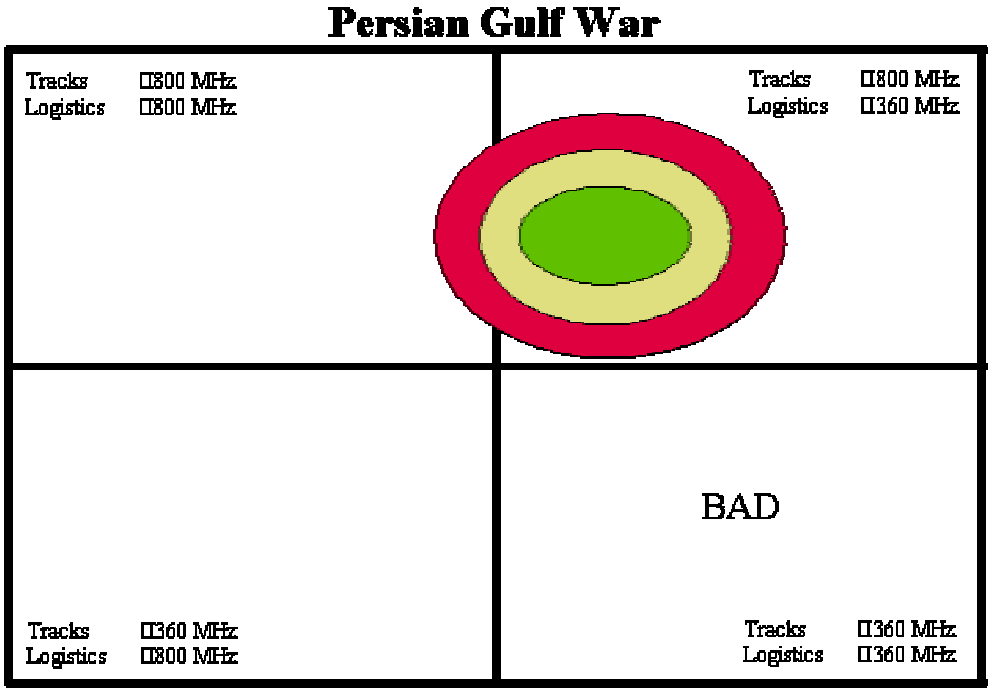

Figure 9: Information Accuracy, Persian Gulf Scenario.

In the lower intensity conflict as depicted by the Somalia Conflict, it may be that the most optimal solution would be to have both servers running on the fastest machine because the track usage is low and the logistics are important to the completion of the mission. However, if the

information used to run the model is inaccurate, then the deployment may suffer. In this case, poor accuracy could lead to two different deployments other than the one desired as shown below.

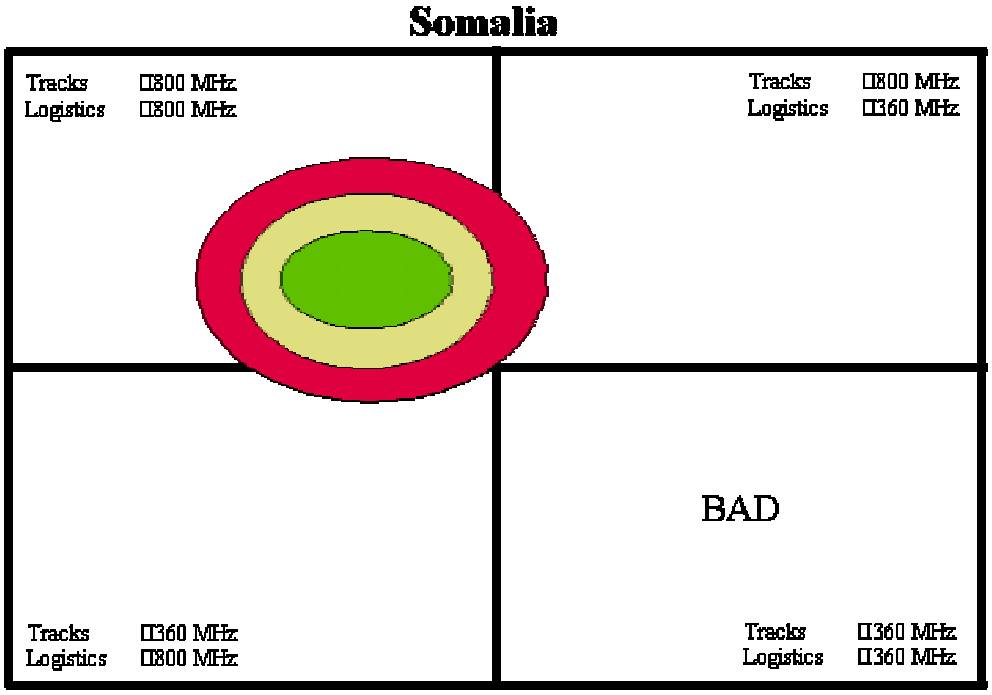

Figure 10: Information Accuracy, Somalia Scenario.

The last mission may have almost no track data to process, but huge logistic problems to correct. This is typical in a humanitarian relief project where food, medicine and shelter are the most crucial obstacles to overcome. In this situation, the most advantageous deployment would have the logistics server running on the fastest machine, and the track server running on the slowest

machine. However, if the information fed to the model is inaccurate, then other deployments are possible.

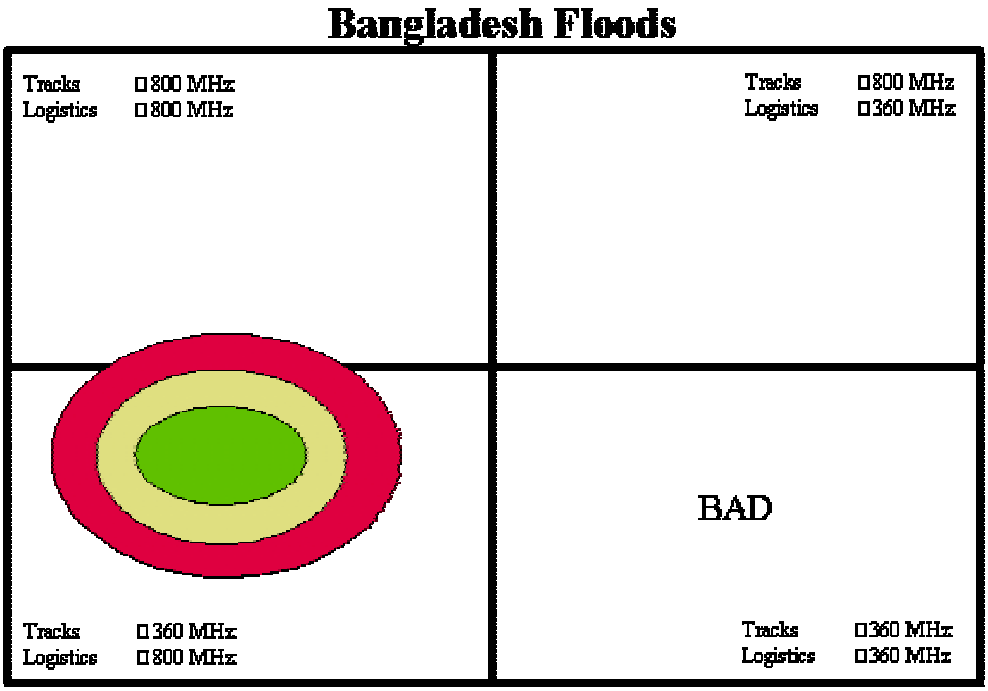

Figure 11: Information Accuracy, Bangladesh Floods Scenario.

4. Combinatorial Explosion

Of course, the problem with combinatorial explosion is also an issue. Larger systems can cause significant delays in computing deployment strategies. The more machines and servers there are to reason about, the more processing time and memory are used by the model to compute an optimal deployment.

Given N object servers and M machines, the possible number of deployment patterns is M raised to the N power.

The table below lists how quickly the number of possible deployment grows. All possible abstractions of the world have been made to reduce the factors to just these parameters. No additional abstractions can be made without invalidating the model. The times were collected by running LINGO models on a 1100MHz AMD machine with 128MB of RAM. These models are listed in Appendix C.

Table 66: Model Computational Time.

SERVERS N	MACHINES M	DEPLOYMENTS	Computational TIME (hh:mm:ss)
4	4	256	00:00:01
5	4	1024	00:00:08
5	5	3125	00:00:15
6	6	46656	00:01:51
7	7	823543	00:04:30
8	8	16777216	00:26:04
9	9	387420489	01:03:50
10	10	10000000000	03:59:44
11	11	285311670611	26:48:44

5. Usefulness of the Model

Although the current version is rather simplistic, the approach seems to have merit. The system responds in a reasonable way with changes is the environment, constraints placed on the system, and different roles that a user might want. Since all of these changes take place on a given network of computers, a single static deployment strategy

will never utilize the assets available to better support the end user.

Predicting exactly how a user will interact with a system that supports multiple roles will always be an inexact science. Using the model in this dissertation to deploy object servers is a software engineering approach to a real world problem that currently exists without a better solution. No solution can be exact because of the limitations inherent in modeling users, software, hardware, etc.

APPENDIX A

This appendix contains an explanation of the transformation between the objective function and the language used by LINGO. Listings of all of the LINGO models used in this dissertation are also present.

A. TRANSFORMATION TO LINGO

The transformation from the objective function to a tool specialized to the task of solving systems of nonlinear equations is unique to each tool or math library used. In this dissertation, the tool of choice was LINGO. The mappings from the objective function to LINGO are thus included.

1. Processing Speed Term

In the objective function, the processing speed term was defined as:

$$Minimize \left[\sum_{n=0}^{N} \sum_{m=0}^{M} \frac{a_{nm} * R_n * S_{norm}}{S_m} \right]$$

In LINGO, the above term is represented by:

```
PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
             V ( I, J ) * MULTIPLIER ( J ) *
                                  NORM_SPEED / SPEED( I ));
```

Note that MULTIPLIER in LINGO is equivalent to R_n, NORM_SPEED is equated to S_{norm}, V is equivalent to a_{nm}, and

SPEED is matched to S_m. More descriptive identifiers were used in the LINGO model in keeping with good programming practices.

2. Network Speed Term

The network speed term in the objective function was:

$$Minimize \left[\sum_{i=0}^{N} \sum_{j=0}^{N} \frac{B_{ij}}{Q_{ij}} \right]$$

In the LINGO models, this term is represented in a couple of ways. If the speed internal to a machine was twice as fast as external, then the following represented this case in LINGO:

```
NET_SPEED = 32696000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;

 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
);
```

In this example, the term B_{ij} was replaced with the actual bit count, 32696000. The term Q_{ij} is comprised of the logic in the denominator of the NET_SPEED equation.

3. RAM Limits Constraint

The constraint in the objective function that limited the RAM utilization was:

$$\forall m \left[\sum_{n=0}^{N} a_{nm} * V_n \leq T_m * U \right]$$

In the LINGO models, this constraint is represented in the form of:

```
@FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
       );
```

4. CPU Limit Constraint

In the objective function, the CPU utilization was limited by the term:

$$\forall m \left[\sum_{n=0}^{N} \frac{a_{nm} * R_n * S_{norm}}{S_m} \leq C \right]$$

In LINGO, this logic was represented in the form of:

```
@FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*
                          NORM_SPEED/SPEED(R)) = Q(R);
        Q(R) < CPU_TIME;
       );
```

5. Whole Server Constraint

In the objective function, the constraint that stated that a server couldn't be split across multiple machines was simply to state that a_{nm} must equal zero or one. In LINGO, this is represented by:

```
@FOR (DEPLOYMENT: @BIN(V));
```

6. Single Copy Constraint

In the objective function, the constraint that stated that a server couldn't have multiple copies was stated as:

$$\forall n \left[\sum_{m=0}^{M} a_{nm} \equiv 1 \right]$$

In LINGO, this logic is represented by:

```
@FOR (SERVER(K):
V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) +
V(@INDEX(Z),K) = 1; );
```

B. LINGO MODELS

1. ADOA3

```
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     35.84      44
     21.3       33
     29.7       66
    100.1       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
ENDDATA
```

```
 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 40000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

END
```

2. ADOA3.1

```
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     294.54      44
      21.3       33
     669.7       66
     832.5       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 40000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
```

```
                V(@INDEX(X), K)*V(@INDEX(X),L) +
                V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

END
```

3. ADOA3.2

```
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
```

```
   MULTIPLIER MEMORYUSE =
     637.6       44
     867.2       33
     292.8       66
       0.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
!   24 * 40000 * 1000  = 960000000
;
!
;
 NET_SPEED = 960000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
```

```
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                                 ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

END
```

4. ADOA3.3

```
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     515.4       44
     765.3       33
    1170.6       66
     915.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
```

```
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
!  61 * 40000 * 1000 = 2440000000
;
 NET_SPEED = 2440000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

END
```

5. ADOA6.1.1

```
!                                          ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS     ;
!                                          ;
MODEL:
```

```
SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     294.54      44
      21.3       33
     669.7       66
     832.5       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 40000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
```

```
                V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

6. ADOA6.1.100

```
!                                        ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS   ;
!                                        ;
MODEL:
```

```
SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     29454       44
      2130       33
     66970       66
     83250       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 4000000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
```

```
                V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

7. ADOA6.1.119

```
!                                       ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS  ;
!                                       ;
MODEL:

SETS:
```

```
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     35050.26      44
      2534.7       33
     79694.3       66
     99067.5       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 4760000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
```

```
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

8. ADOA6.2.1

```
!                                       ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS  ;
!                                       ;
MODEL:

SETS:
   MACHINE / W X Y Z/:
```

```
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     637.6       44
     867.2       33
     292.8       66
       0.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 960000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
```

```
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

9. ADOA6.3.1

```
!                                        ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS   ;
!                                        ;
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
```

```
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     515.4       44
     765.3       33
    1170.6       66
     915.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 2440000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
```

```
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

10. ADOA6.3.50

```
!                                       ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS  ;
!                                       ;
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
```

```
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     25770.0       44
     38265.0       33
     58530.0       66
     45750.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 122000000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
```

```
!  A server cannot be split over multiple machines              ;
!                                                               ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                               ;
!  Each server can only run on one machine.                     ;
!                                                               ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                ;
!  Constraint for limiting the CPU load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

11. ADOA6.3.75

```
!                                        ;
!  SAME AS ADOA3 WITH CPU USAGE LIMITS   ;
!                                        ;
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
```

```
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
   MULTIPLIER MEMORYUSE =
     38655.0       44
     57397.5       33
     87795.0       66
     68625.0       44;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 183000000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
```

```
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R)  < MEMORY(R)*MEM_LIMIT;
   );

!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     Q(R) = ( V(R, @INDEX(A))*MULTIPLIER(@INDEX(A))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(B))*MULTIPLIER(@INDEX(B))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(C))*MULTIPLIER(@INDEX(C))* NORM_SPEED /
SPEED( R ) ) +
            ( V(R, @INDEX(D))*MULTIPLIER(@INDEX(D))* NORM_SPEED /
SPEED( R ) );
     Q(R)  < CPU_TIME;
   );

END
```

12. ADOA4

```
!                                                                ;
! SAME AS ADOA3 WITH SIMPLISTIC RAM FUNCTION BUILT IN            ;
!                                                                ;
MODEL:

SETS:
   MACHINE / W X Y Z/:
      MEMORY, SPEED;
   SERVER  / A B C D/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
```

```
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   MEM_PER (MACHINE): S;
ENDSETS

DATA:
!                                                    ;
!        MACHINE METRICS                             ;
!                                                    ;
   NORM_SPEED = 500;
   MEMORY SPEED =
     64    300
    128    200
    128    400
    256    500;
!                                                     ;
!        SERVER METRICS                               ;
!                                                     ;
   MULTIPLIER MEMORYUSE =
     35.84      44
     21.3       33
     29.7       66
    100.1       44;
   NET_BW = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
             (3^S(I)) * V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED /
SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 40000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                  ;
!  Figure out if two servers are running on the same machine.      ;
!                                                                  ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        U(K,L) = V(@INDEX(W), K)*V(@INDEX(W),L) +
                 V(@INDEX(X), K)*V(@INDEX(X),L) +
                 V(@INDEX(Y), K)*V(@INDEX(Y),L) +
                 V(@INDEX(Z), K)*V(@INDEX(Z),L) + 1;
```

```
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
     V(@INDEX(W),K) + V(@INDEX(X),K) + V(@INDEX(Y), K) + V(@INDEX(Z),K)
= 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
     T(R) = V(R, @INDEX(A))*MEMORYUSE(@INDEX(A)) +
            V(R, @INDEX(B))*MEMORYUSE(@INDEX(B))  +
            V(R, @INDEX(C))*MEMORYUSE(@INDEX(C))  +
            V(R, @INDEX(D))*MEMORYUSE(@INDEX(D));
     T(R) < 2*MEMORY(R);        !  NOTE:  VIRTUAL RAM
;
     S(R) = T(R)/MEMORY(R);     !  NOTE:  SIMPLISTIC RAM USAGE
MULTIPLIER ;
   );

END
```

13. ADOA8.1

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
```

```
   MULTIPLIER MEMORYUSE =
     31600.3      44
      5467.4      60
     10812.6      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 320000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.          ;
```

```
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

14. ADOA8.2

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     110608.0      44
     201879.6      60
      72103.2      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
```

```
!
;
 NET_SPEED = 7680000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

15. ADOA8.3

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
```

```
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     72796.5      44
    227518.4      60
    327870.45     66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 19520000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
```

```
!                                                                    ;
!  Constraint for limiting the RAM load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                    ;
!  Constraint for limiting the CPU load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

16. ADOA8.1.28

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     884808.4      44
     153087.2      60
     302752.8      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
```

```
             V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 8960000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

17. ADOA8.2.4

```
MODEL:
```

```
SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
```

```
!  A server cannot be split over multiple machines              ;
!                                                               ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                               ;
!  Each server can only run on one machine.                     ;
!                                                               ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                ;
!  Constraint for limiting the CPU load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

18. ADOA8.3.3

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
    218389.5      44
    682555.2      60
```

```
    983611.35     66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 58560000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
```

```
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

19. ADOA8.3.2

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     145593.0      44
     455036.8      60
     655740.9      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
```

```
 NET_SPEED = 19520000000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

20. ADOA9.1

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS
```

```
DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     30189.2      44
      5118.3      60
     10456.0      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 536000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
```

```
!                                                                    ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                    ;
!  Constraint for limiting the CPU load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

21. ADOA9.2

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     108491.0      44
     186549.2      60
      70674.0      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
```

```
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 1344000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

22. ADOA9.3

```
MODEL:
```

```
SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     70496.5      44
    216651.3      60
    317095.4      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 32696000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
```

```
!  A server cannot be split over multiple machines              ;
!                                                               ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                               ;
!  Each server can only run on one machine.                     ;
!                                                               ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                ;
!  Constraint for limiting the CPU load on a single machine.     ;
!                                                                ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

23. ADOA9.1.28

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     845297.6      44
```

```
    143312.4      60
    292768.0      66;
  MEM_LIMIT = ?;
  NET_BW = ?;
  CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 15008000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
```

```
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );
END
```

24. ADOA9.2.4

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     433964.0      44
     746196.8      60
     282696.0      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 5376000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
```

```
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

25. ADOA9.3.3

```
MODEL:

SETS:
   MACHINE / SIX BR733 GIGA/:
      MEMORY, SPEED;
   SERVER  / A B C/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS
```

```
DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000;
   MULTIPLIER MEMORYUSE =
     211489.5      44
     649953.9      60
     951286.2      66;
   MEM_LIMIT = ?;
   NET_BW = ?;
   CPU_TIME = ?;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 98088000/(U(@INDEX(B),@INDEX(C))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
```

```
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                    ;
!  Constraint for limiting the CPU load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B

A. JAVA RMI CODE

1. Server A Code

a) A.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface A extends Remote {
        int memory_add(int counts) throws RemoteException;
        int memory_del(int counts) throws RemoteException;
        StringBuffer m1() throws RemoteException;
        StringBuffer m2() throws RemoteException;
        StringBuffer m3() throws RemoteException;
        StringBuffer m4() throws RemoteException;
}
```

b) Aimpl.java

```
import java.util.Vector;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class AImpl extends UnicastRemoteObject implements A {

  public AImpl() throws RemoteException {         super();     }

  StringBuffer a1;
  StringBuffer a2;
  StringBuffer a3;
  StringBuffer a4;
  Vector medium_memory;

  public static int factorial(int x) {
     if (x <= 0)
       return 0;
     else
       return factorial(x-1) + x;
```

```
}


public int memory_add(int counts) {
   int cnt = 0;

   while (cnt < counts)
   {
      StringBuffer billy = new StringBuffer(1000000);
      medium_memory.add(billy);
      cnt = cnt + 1;
   }

   return cnt;
}

public int memory_del(int counts) {
   int cnt = 0;

   while (cnt < counts)
   {
      if (medium_memory.isEmpty())
        return -1;

      medium_memory.remove(medium_memory.lastElement());
      cnt = cnt + 1;
   }

   return cnt;
}

public StringBuffer m1() {
  int count;

  for (int i = 0; i < 360; i++)
    for (int j = 0; j < 360; j++)
       count = factorial(i) * factorial(j);

      return a1;
}

public StringBuffer m2() {
  int count;

  for (int i = 0; i < 600; i++)
    for (int j = 0; j < 600; j++)
       count = factorial(i) * factorial(j);

      return a2;
}
```

```
  public StringBuffer m3() {
    int count;

    for (int i = 0; i < 460; i++)
      for (int j = 0; j < 460; j++)
         count = factorial(i) * factorial(j);

       return a3;
  }

  public StringBuffer m4() {
    int count;

    for (int i = 0; i < 550; i++)
      for (int j = 0; j < 550; j++)
         count = factorial(i) * factorial(j);

       return a4;
  }

  public void init() {
   a1 = new StringBuffer(14000);
   a2 = new StringBuffer(2300);
   a3 = new StringBuffer(5600);
   a4 = new StringBuffer(22000);
   medium_memory = new Vector();
  }

  public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            AImpl obj = new AImpl();
            obj.init();
            // Bind this object instance to the name "AServer"
            Naming.rebind("//giga/AServer", obj);
            System.out.println("AServer bound in registry");
        } catch (Exception e) {
             System.out.println("AImpl err: " + e.getMessage());
             e.printStackTrace();
        }
    }
}
```

2. Server B

a) B.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface B extends Remote {
        int memory_add(int counts) throws RemoteException;
        int memory_del(int counts) throws RemoteException;
        StringBuffer m1() throws RemoteException;
        StringBuffer m2() throws RemoteException;
}
```

b) Bimpl.java

```
import java.util.Vector;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class BImpl extends UnicastRemoteObject implements B {

  public BImpl() throws RemoteException {         super();     }

  StringBuffer b1;
  StringBuffer b2;
  Vector medium_memory;
  C objC = null;

  public static int factorial(int x) {
     if (x <= 0)
       return 0;
     else
       return factorial(x-1) + x;
  }

  public int memory_add(int counts) {
     int cnt = 0;

     while (cnt < counts)
     {
        StringBuffer billy = new StringBuffer(1000000);
        medium_memory.add(billy);
        cnt = cnt + 1;
```

```
    }

    return cnt;
 }

 public int memory_del(int counts) {
    int cnt = 0;

    while (cnt < counts)
    {
       if (medium_memory.isEmpty())
         return -1;

       medium_memory.remove(medium_memory.lastElement());
       cnt = cnt + 1;
    }

    return cnt;
 }

 public StringBuffer m1() {
   int count;

   for (int i = 0; i < 511; i++)
     for (int j = 0; j < 511; j++)
        count = factorial(i) * factorial(j);

       return b1;
 }

 public StringBuffer m2() {
   int count;
StringBuffer answer;

   for (int i = 0; i < 666; i++)
     for (int j = 0; j < 666; j++)
        count = factorial(i) * factorial(j);

       try {
             answer = objC.m1();
  }
 catch (Exception exc) {
        System.out.println("BServer exception: " +
                           exc.getMessage());
   exc.printStackTrace();
 }

       return b2;
 }

 public void init() {
```

```
   b1 = new StringBuffer(500000);
   b2 = new StringBuffer(340000);
   medium_memory = new Vector();
   try {
      objC = (C)Naming.lookup("//giga/CServer");
   } catch (Exception e) {
       System.out.println("BServer exception: " +
                          e.getMessage());
    e.printStackTrace();
   }
  }

  public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            BImpl obj = new BImpl();
            obj.init();
            // Bind this object instance to the name "BServer"
            Naming.rebind("//giga/BServer", obj);
            System.out.println("BServer bound in registry");
        } catch (Exception e) {
             System.out.println("BImpl err: " + e.getMessage());
             e.printStackTrace();
        }
    }
}
```

3. Server C

a) C.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface C extends Remote {
        int memory_add(int counts) throws RemoteException;
        int memory_del(int counts) throws RemoteException;
        StringBuffer m1() throws RemoteException;
        StringBuffer m2() throws RemoteException;
        StringBuffer m3() throws RemoteException;
}
```

b) Cimpl.java

```
import java.util.Vector;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;

public class CImpl extends UnicastRemoteObject implements C {

  public CImpl() throws RemoteException {         super();     }

  StringBuffer c1;
  StringBuffer c2;
  StringBuffer c3;
  Vector medium_memory;

  public static int factorial(int x) {
     if (x <= 0)
       return 0;
     else
       return factorial(x-1) + x;
  }

  public int memory_add(int counts) {
     int cnt = 0;

     while (cnt < counts)
     {
        StringBuffer billy = new StringBuffer(1000000);
        medium_memory.add(billy);
        cnt = cnt + 1;
     }

     return cnt;
  }


  public int memory_del(int counts) {
     int cnt = 0;

     while (cnt < counts)
     {
        if (medium_memory.isEmpty())
          return -1;

        medium_memory.remove(medium_memory.lastElement());
        cnt = cnt + 1;
     }
```

```
   return cnt;
}

public StringBuffer m1() {
  int count;

  for (int i = 0; i < 627; i++)
    for (int j = 0; j < 627; j++)
       count = factorial(i) * factorial(j);

      return c1;
}

public StringBuffer m2() {
  int count;

  for (int i = 0; i < 726; i++)
    for (int j = 0; j < 726; j++)
       count = factorial(i) * factorial(j);

      return c2;
}

public StringBuffer m3() {
  int count;

  for (int i = 0; i < 340; i++)
    for (int j = 0; j < 340; j++)
       count = factorial(i) * factorial(j);

      return c3;
}

public void init() {
 c1 = new StringBuffer(40000);
 c2 = new StringBuffer(500000);
 c3 = new StringBuffer(50000);
 medium_memory = new Vector();
}

public static void main(String args[]) {
      // Create and install a security manager
      if (System.getSecurityManager() == null) {
          System.setSecurityManager(new RMISecurityManager());
      }
      try {
          CImpl obj = new CImpl();
          obj.init();
          // Bind this object instance to the name "CServer"
          Naming.rebind("//giga/CServer", obj);
          System.out.println("CServer bound in registry");
```

```
        } catch (Exception e) {
             System.out.println("CImpl err: " + e.getMessage());
             e.printStackTrace();
        }
    }
}
```

4. User Simulation Code

a) Timer.java

```
/**
 * A utility class to help time internal operations.
 *
 */

public class Timer {

  private long startMillis;
  private long endMillis;

// constructor
  public Timer() {
    reset();
  }

// constructor
  public Timer(boolean running) {
    reset();
    if (!running) {
      endMillis = startMillis;
    }
  }

  public void reset() {
    startMillis = getMillis();
    endMillis = 0;
  }

  public void stop() {
    endMillis = getMillis();
  }

  public long elapsedms() {
    long millis = delta();
 return millis;
  }
```

```
  public String elapsed() {
    long millis = delta();
    if (millis > 1000L)  {
      char xx[] = new char[2];
      xx[0] = (char)('0' + (millis % 1000)/100);
      xx[1] = (char)('0' + ((millis+5) % 100)/10);
      return ("" + millis/1000 + "." + new String(xx) + " seconds");
    } else {
      return ("" + millis + " milliseconds");
    }
  }

  public void add(Timer tim) {
    if (endMillis == 0) {
      throw new Error("Can only add to a stopped Timer");
    }
    endMillis += tim.delta();
  }

  private long delta() {
    if (endMillis == 0) {
      stop();
    }
    return (endMillis - startMillis);
  }

  private long getMillis() {
    return System.currentTimeMillis();
  }

}
```

b) Roles.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class Roles {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A objA = null;
    B objB = null;
```

```
   C objC = null;

   public void init() {
    try {
        objA = (A)Naming.lookup("//giga/AServer");
        objB = (B)Naming.lookup("//giga/BServer");
        objC = (C)Naming.lookup("//giga/CServer");

     } catch (Exception e) {
         System.out.println("Roles exception: " +
                                  e.getMessage());
         e.printStackTrace();
   RUNNING = false;
     }
}

   public void set_memory() {
    int val;
    try {
        val = objA.memory_add(44);
        System.out.println("MB set in server A: " + val);
        val = objB.memory_add(60);
        System.out.println("MB set in server B: " + val);
        val = objC.memory_add(66);
        System.out.println("MB set in server C: " + val);

     } catch (Exception e) {
         System.out.println("Roles exception: " +
                                  e.getMessage());
         e.printStackTrace();
   RUNNING = false;
     }
   }

public void run_test1(int max_run) {
   int choice = 1;
   double average;
          boolean RUN1 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
          int cnt4 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN1)
   {
      choice = simulate.nextInt(53);
      calltime.reset();
   try {
```

```
    if (choice < 50)
 {
            objA.m1();
                            cnt1 = cnt1 + 1;
 }
 else if (choice < 51)
 {
            objA.m2();
            objB.m1();
                            cnt2 = cnt2 + 1;
 }
 else if (choice < 52)
 {
            objC.m1();
            objC.m2();
                            cnt3 = cnt3 + 1;
 }
 else if (choice < 53)
 {
            objB.m2();
                            cnt4 = cnt4 + 1;
 }
 else
 {
      System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("Roles exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Roles exception on sleep: " +
                 inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
```

```
   }
   }
   testtime.stop();
   System.out.println("Test 1 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   System.out.println("  4   Number of calls is " + cnt4);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

public void run_test2(int max_run) {
   int choice = 1;
   double average;
          boolean RUN2 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN2)
   {
      choice = simulate.nextInt(74);
      calltime.reset();
   try {
       if (choice < 10)
    {
               objA.m1();
                              cnt1 = cnt1 + 1;
    }
    else if (choice < 50)
    {
               objA.m2();
               objB.m1();
                              cnt2 = cnt2 + 1;
    }
    else if (choice < 74)
    {
               objB.m1();
               objB.m2();
                              cnt3 = cnt3 + 1;
    }
    else
    {
         System.out.println("Got choice out of bounds " + choice);
    }
```

```
      }
    catch (Exception exc) {
        System.out.println("Roles exception: " +
                    exc.getMessage());
     exc.printStackTrace();
     RUN2 = false;
     RUNNING = false;
    }
    calltime.stop();
    duration = duration + calltime.elapsedms();
    count = count + 1;
    if (count < max_run)
    {
       try {
         Thread.currentThread().sleep(1000);
        }
       catch (Exception inter) {
          System.out.println("Roles exception on sleep: " +
                      inter.getMessage());
       inter.printStackTrace();
       }
    }
    else
    {
       RUN2 = false;
    }
    }
    testtime.stop();
    System.out.println("Test 2 duration is " + testtime.elapsed());
    System.out.println("Total number of calls is " + count);
    System.out.println("  1   Number of calls is " + cnt1);
    System.out.println("  2   Number of calls is " + cnt2);
    System.out.println("  3   Number of calls is " + cnt3);
    average = (double) duration / (double) count;
    System.out.println("Average response time is " + average);
    System.out.println("");
}

public void run_test3(int max_run) {
    int choice = 1;
    double average;
           boolean RUN3 = true;
    int count = 0;
           int cnt1 = 0;
           int cnt2 = 0;
           int cnt3 = 0;
           int cnt4 = 0;
           int cnt5 = 0;
    long duration = 0;

    testtime.reset();
```

```
   while (RUN3)
{
   choice = simulate.nextInt(92);
   calltime.reset();
try {
    if (choice < 50)
 {
            objA.m1();
            objB.m2();
                           cnt1 = cnt1 + 1;
 }
 else if (choice < 60)
 {
            objA.m1();
            objA.m2();
            objA.m3();
            objB.m2();
                           cnt2 = cnt2 + 1;
 }
 else if (choice < 90)
 {
            objC.m2();
                           cnt3 = cnt3 + 1;
 }
 else if (choice < 91)
 {
            objC.m3();
                           cnt4 = cnt4 + 1;
 }
 else if (choice < 92)
 {
            objB.m1();
            objB.m2();
                           cnt5 = cnt5 + 1;
 }
 else
 {
     System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("Roles exception: " +
                  exc.getMessage());
 exc.printStackTrace();
 RUN3 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
```

```
   {
      try {
        Thread.currentThread().sleep(1000);
       }
      catch (Exception inter) {
         System.out.println("Roles exception on sleep: " +
                     inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN3 = false;
   }
   }
   testtime.stop();
   System.out.println("Test 3 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   System.out.println("  4   Number of calls is " + cnt4);
   System.out.println("  5   Number of calls is " + cnt5);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

   public static void main(String s[]) {
/*
*/

Roles test = new Roles();
calltime = new Timer();
testtime = new Timer();
simulate = new Random();


   test.init();
   test.run_test1(1000);
   test.run_test2(1000);
   test.run_test3(1000);
   test.set_memory();
   System.out.println("");
   System.out.println("Bumping the memory on the servers...");
   System.out.println("");
   test.run_test1(1000);
   test.run_test2(1000);
   test.run_test3(1000);

  }
```

```
}
```

c) R1.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class R1 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A objA = null;
    B objB = null;
    C objC = null;

    public void init() {
     try {
         objA = (A)Naming.lookup("//giga/AServer");
         objB = (B)Naming.lookup("//giga/BServer");
         objC = (C)Naming.lookup("//giga/CServer");

      } catch (Exception e) {
          System.out.println("R1 exception: " +
                                  e.getMessage());
          e.printStackTrace();
    RUNNING = false;
      }
 }

    public void set_memory() {
     int val;
     try {
         val = objA.memory_add(44);
         System.out.println("MB set in server A: " + val);
         val = objB.memory_add(60);
         System.out.println("MB set in server B: " + val);
         val = objC.memory_add(66);
         System.out.println("MB set in server C: " + val);

      } catch (Exception e) {
          System.out.println("R1 exception: " +
                                  e.getMessage());
          e.printStackTrace();
    RUNNING = false;
```

```
     }
   }

public void run_test1(int max_run) {
   int choice = 1;
   double average;
          boolean RUN1 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
          int cnt4 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN1)
   {
      choice = simulate.nextInt(53);
      calltime.reset();
   try {
       if (choice < 50)
    {
               objA.m1();
                              cnt1 = cnt1 + 1;
    }
    else if (choice < 51)
    {
               objA.m2();
               objB.m1();
                              cnt2 = cnt2 + 1;
    }
    else if (choice < 52)
    {
               objC.m1();
               objC.m2();
                              cnt3 = cnt3 + 1;
    }
    else if (choice < 53)
    {
               objB.m2();
                              cnt4 = cnt4 + 1;
    }
    else
    {
         System.out.println("Got choice out of bounds " + choice);
    }
     }
   catch (Exception exc) {
       System.out.println("R1 exception: " +
                     exc.getMessage());
    exc.printStackTrace();
```

```
    RUN1 = false;
    RUNNING = false;
   }
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < max_run)
   {
      try {
        Thread.currentThread().sleep(15500);
       }
      catch (Exception inter) {
         System.out.println("R1 exception on sleep: " +
                     inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN1 = false;
   }
   }
   testtime.stop();
   System.out.println("Test 1 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   System.out.println("  4   Number of calls is " + cnt4);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

   public static void main(String s[]) {
/*
*/

R1 test = new R1();
calltime = new Timer();
testtime = new Timer();
simulate = new Random();


   test.init();
   test.run_test1(848);
   test.set_memory();
   System.out.println("");
   System.out.println("Bumping the memory on the servers...");
   System.out.println("");
   test.run_test1(848);
```

```
   }
}
```

d) R2.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class R2 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A objA = null;
    B objB = null;
    C objC = null;

    public void init() {
     try {
         objA = (A)Naming.lookup("//giga/AServer");
         objB = (B)Naming.lookup("//giga/BServer");
         objC = (C)Naming.lookup("//giga/CServer");

      } catch (Exception e) {
          System.out.println("R2 exception: " +
                                  e.getMessage());
          e.printStackTrace();
    RUNNING = false;
      }
 }

    public void set_memory() {
     int val;
     try {
         val = objA.memory_add(44);
         System.out.println("MB set in server A: " + val);
         val = objB.memory_add(60);
         System.out.println("MB set in server B: " + val);
         val = objC.memory_add(66);
         System.out.println("MB set in server C: " + val);

      } catch (Exception e) {
          System.out.println("R2 exception: " +
                                  e.getMessage());
```

```
          e.printStackTrace();
      RUNNING = false;
        }
      }

  public void run_test2(int max_run) {
      int choice = 1;
      double average;
             boolean RUN2 = true;
      int count = 0;
             int cnt1 = 0;
             int cnt2 = 0;
             int cnt3 = 0;
      long duration = 0;

      testtime.reset();
         while (RUN2)
      {
         choice = simulate.nextInt(74);
         calltime.reset();
      try {
          if (choice < 10)
       {
                  objA.m1();
                                  cnt1 = cnt1 + 1;
       }
       else if (choice < 50)
       {
                  objA.m2();
                  objB.m1();
                                  cnt2 = cnt2 + 1;
       }
       else if (choice < 74)
       {
                  objB.m1();
                  objB.m2();
                                  cnt3 = cnt3 + 1;
       }
       else
       {
            System.out.println("Got choice out of bounds " + choice);
       }
        }
      catch (Exception exc) {
          System.out.println("R2 exception: " +
                       exc.getMessage());
       exc.printStackTrace();
       RUN2 = false;
       RUNNING = false;
      }
      calltime.stop();
```

```
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < max_run)
   {
      try {
        Thread.currentThread().sleep(6500);
       }
      catch (Exception inter) {
         System.out.println("R2 exception on sleep: " +
                     inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN2 = false;
   }
   }
   testtime.stop();
   System.out.println("Test 2 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

   public static void main(String s[]) {
/*
*/

R2 test = new R2();
calltime = new Timer();
testtime = new Timer();
simulate = new Random();


   test.init();
   test.run_test2(1184);
   test.set_memory();
   System.out.println("");
   System.out.println("Bumping the memory on the servers...");
   System.out.println("");
   test.run_test2(1184);

  }
}
```

e) R3.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class R3 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A objA = null;
    B objB = null;
    C objC = null;

    public void init() {
     try {
         objA = (A)Naming.lookup("//giga/AServer");
         objB = (B)Naming.lookup("//giga/BServer");
         objC = (C)Naming.lookup("//giga/CServer");

      } catch (Exception e) {
          System.out.println("R3 exception: " +
                                  e.getMessage());
          e.printStackTrace();
    RUNNING = false;
      }
 }

    public void set_memory() {
     int val;
     try {
         val = objA.memory_add(44);
         System.out.println("MB set in server A: " + val);
         val = objB.memory_add(60);
         System.out.println("MB set in server B: " + val);
         val = objC.memory_add(66);
         System.out.println("MB set in server C: " + val);

      } catch (Exception e) {
          System.out.println("R3 exception: " +
                                  e.getMessage());
          e.printStackTrace();
    RUNNING = false;
      }
    }
```

```
public void run_test3(int max_run) {
   int choice = 1;
   double average;
          boolean RUN3 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
          int cnt4 = 0;
          int cnt5 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN3)
   {
      choice = simulate.nextInt(92);
      calltime.reset();
   try {
       if (choice < 50)
    {
               objA.m1();
               objB.m2();
                              cnt1 = cnt1 + 1;
    }
    else if (choice < 60)
    {
               objA.m1();
               objA.m2();
               objA.m3();
               objB.m2();
                              cnt2 = cnt2 + 1;
    }
    else if (choice < 90)
    {
               objC.m2();
                              cnt3 = cnt3 + 1;
    }
    else if (choice < 91)
    {
               objC.m3();
                              cnt4 = cnt4 + 1;
    }
    else if (choice < 92)
    {
               objB.m1();
               objB.m2();
                              cnt5 = cnt5 + 1;
    }
    else
    {
         System.out.println("Got choice out of bounds " + choice);
```

```
     }
      }
    catch (Exception exc) {
        System.out.println("R3 exception: " +
                    exc.getMessage());
     exc.printStackTrace();
     RUN3 = false;
     RUNNING = false;
    }
    calltime.stop();
    duration = duration + calltime.elapsedms();
    count = count + 1;
    if (count < max_run)
    {
       try {
         Thread.currentThread().sleep(4500);
        }
       catch (Exception inter) {
          System.out.println("R3 exception on sleep: " +
                      inter.getMessage());
       inter.printStackTrace();
       }
    }
    else
    {
       RUN3 = false;
    }
    }
    testtime.stop();
    System.out.println("Test 3 duration is " + testtime.elapsed());
    System.out.println("Total number of calls is " + count);
    System.out.println("  1   Number of calls is " + cnt1);
    System.out.println("  2   Number of calls is " + cnt2);
    System.out.println("  3   Number of calls is " + cnt3);
    System.out.println("  4   Number of calls is " + cnt4);
    System.out.println("  5   Number of calls is " + cnt5);
    average = (double) duration / (double) count;
    System.out.println("Average response time is " + average);
    System.out.println("");
}

    public static void main(String s[]) {
/*
*/

R3 test = new R3();
calltime = new Timer();
testtime = new Timer();
simulate = new Random();
```

```
    test.init();
    test.run_test3(1312);
    test.set_memory();
    System.out.println("");
    System.out.println("Bumping the memory on the servers...");
    System.out.println("");
    test.run_test3(1312);

   }
}
```

5. Client Side Code

a) Client1.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Client1 extends JPanel {
    static JFrame frame;

    static String a1= "Button 1";
    static String a2= "Button 2";
    static String one = "1";
    static String two = "22";


    JRadioButton a1Button, a2Button;

    A objA = null;
    B objB = null;
    int val = 0;

    public Client1() {
 // Create the buttons.

 a1Button = new JRadioButton(a1);
 a1Button.setActionCommand(one);
 a2Button = new JRadioButton(a2);
 a2Button.setActionCommand(two);


 // Group the radio buttons.
 ButtonGroup group = new ButtonGroup();
 group.add(a1Button);
 group.add(a2Button);
```

```
       // Register a listener for the radio buttons.
RadioListener myListener = new RadioListener();
a1Button.addActionListener(myListener);
a2Button.addActionListener(myListener);

add(a1Button);
add(a2Button);

   }

public void init() {
    try {
        objA = (A)Naming.lookup("//coltsfan/AServer");
        objB = (B)Naming.lookup("//coltsfan/BServer");

     } catch (Exception e) {
         System.out.println("Client1 exception: " +
                                  e.getMessage());
         e.printStackTrace();
     }
 }

   /** An ActionListener that listens to the radio buttons. */
   class RadioListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
    String choice = e.getActionCommand();
           int choicelen = choice.length();

           try {
             switch(choicelen) {
               case 1:
                  objA.m1();
                  break;
               case 2:
                  objA.m2();
                  objB.m1();
                  break;
               default:
      break;
             }
           }
    catch (Exception exc) {
              System.out.println("Client1 exception: " +
                                  exc.getMessage());
               exc.printStackTrace();
       JRadioButton button = (JRadioButton)e.getSource();
       button.setEnabled(false);
           }

}
```

```
    }

    public static void main(String s[]) {
 /*
 */

 Client1 panel = new Client1();


        panel.init();

 frame = new JFrame("Client1");
 frame.addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e) {System.exit(0);}
 });
 frame.getContentPane().add("Center", panel);
 frame.pack();
 frame.setVisible(true);

    }
}
```

b) Client2.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Client2 extends JPanel {
    static JFrame frame;

    static String a1= "Button 1";
    static String a2= "Button 2";
    static String a3= "Button 3";
    static String a4= "Button 4";
    static String b1= "Button 5";
    static String b2= "Button 6";
    static String c1= "Button 7";
    static String c2= "Button 8";
    static String c3= "Button 9";
    static String one = "1";
    static String two = "22";
    static String thr = "333";
    static String fou = "4444";
    static String fiv = "55555";
    static String six = "666666";
 static String sev = "7777777";
```

```
static String eig = "88888888";
static String nin = "999999999";


   JRadioButton a1Button, a2Button, a3Button, a4Button;
   JRadioButton b1Button, b2Button;
   JRadioButton c1Button, c2Button, c3Button;

   A objA = null;
   B objB = null;
   C objC = null;
   int val = 0;

   public Client2() {
// Create the buttons.

a1Button = new JRadioButton(a1);
a1Button.setActionCommand(one);
a2Button = new JRadioButton(a2);
a2Button.setActionCommand(two);
a3Button = new JRadioButton(a3);
a3Button.setActionCommand(thr);
a4Button = new JRadioButton(a4);
a4Button.setActionCommand(fou);

b1Button = new JRadioButton(b1);
b1Button.setActionCommand(fiv);
b2Button = new JRadioButton(b2);
b2Button.setActionCommand(six);

c1Button = new JRadioButton(c1);
c1Button.setActionCommand(sev);
c2Button = new JRadioButton(c2);
c2Button.setActionCommand(eig);
c3Button = new JRadioButton(c3);
c3Button.setActionCommand(nin);


// Group the radio buttons.
ButtonGroup group = new ButtonGroup();
group.add(a1Button);
group.add(a2Button);
group.add(a3Button);
group.add(a4Button);
group.add(b1Button);
group.add(b2Button);
group.add(c1Button);
group.add(c2Button);
group.add(c3Button);

       // Register a listener for the radio buttons.
```

```
RadioListener myListener = new RadioListener();
a1Button.addActionListener(myListener);
a2Button.addActionListener(myListener);
a3Button.addActionListener(myListener);
a4Button.addActionListener(myListener);
b1Button.addActionListener(myListener);
b2Button.addActionListener(myListener);
c1Button.addActionListener(myListener);
c2Button.addActionListener(myListener);
c3Button.addActionListener(myListener);

add(a1Button);
add(a2Button);
add(a3Button);
add(a4Button);
add(b1Button);
add(b2Button);
add(c1Button);
add(c2Button);
add(c3Button);

   }

public void init() {
    try {
        objA = (A)Naming.lookup("//coltsfan/AServer");
        objB = (B)Naming.lookup("//coltsfan/BServer");
        objC = (C)Naming.lookup("//coltsfan/CServer");

     } catch (Exception e) {
         System.out.println("Client2 exception: " +
                                  e.getMessage());
         e.printStackTrace();
     }
 }

   /** An ActionListener that listens to the radio buttons. */
   class RadioListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
    String choice = e.getActionCommand();
           int choicelen = choice.length();

           try {
             switch(choicelen) {
               case 1:
                  objC.m1();
                  objC.m2();
                  break;
               case 2:
                  objC.m3();
                  break;
```

```
                case 3:
                   objC.m2();
                   break;
                case 4:
                   objC.m3();
                   break;
                case 5:
                   objA.m1();
                   objB.m2();
                   break;
                case 6:
                   objB.m2();
                   break;
                case 7:
                   objA.m4();
                   break;
                case 8:
                   objC.m3();
                   objA.m3();
                   break;
                case 9:
                   objA.m1();
                   objA.m2();
                   objA.m3();
                   objB.m2();
                   break;
                default:
      break;
             }
          }
    catch (Exception exc) {
              System.out.println("Client2 exception: " +
                                 exc.getMessage());
              exc.printStackTrace();
       JRadioButton button = (JRadioButton)e.getSource();
       button.setEnabled(false);
          }

}
   }

   public static void main(String s[]) {
/*
*/

Client2 panel = new Client2();


       panel.init();

frame = new JFrame("Client2");
```

```
 frame.addWindowListener(new WindowAdapter() {
     public void windowClosing(WindowEvent e) {System.exit(0);}
 });
 frame.getContentPane().add("Center", panel);
 frame.pack();
 frame.setVisible(true);

    }
}
```

c) Client3.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Client3 extends JPanel {
    static JFrame frame;

    static String a1= "Button 1";
    static String a2= "Button 2";
    static String a3= "Button 3";
    static String one = "1";
    static String two = "22";
    static String thr = "333";


    JRadioButton a1Button, a2Button, a3Button;

    B objB = null;
    C objC = null;
    int val = 0;

    public Client3() {
 // Create the buttons.

 a1Button = new JRadioButton(a1);
 a1Button.setActionCommand(one);
 a2Button = new JRadioButton(a2);
 a2Button.setActionCommand(two);
 a3Button = new JRadioButton(a3);
 a3Button.setActionCommand(thr);

 // Group the radio buttons.
 ButtonGroup group = new ButtonGroup();
 group.add(a1Button);
 group.add(a2Button);
```

```
group.add(a3Button);

       // Register a listener for the radio buttons.
RadioListener myListener = new RadioListener();
a1Button.addActionListener(myListener);
a2Button.addActionListener(myListener);
a3Button.addActionListener(myListener);

add(a1Button);
add(a2Button);
add(a3Button);

   }

public void init() {
    try {
        objB = (B)Naming.lookup("//coltsfan/BServer");
        objC = (C)Naming.lookup("//coltsfan/CServer");

     } catch (Exception e) {
         System.out.println("Client3 exception: " +
                                  e.getMessage());
        e.printStackTrace();
     }
 }

   /** An ActionListener that listens to the radio buttons. */
   class RadioListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
    String choice = e.getActionCommand();
           int choicelen = choice.length();

           try {
             switch(choicelen) {
               case 1:
                  objC.m1();
                  break;
               case 2:
                  objB.m1();
                  objB.m2();
                  break;
               case 3:
                  objC.m2();
                  break;
               default:
      break;
             }
           }
    catch (Exception exc) {
              System.out.println("Client3 exception: " +
                                  exc.getMessage());
```

```
                exc.printStackTrace();
            JRadioButton button = (JRadioButton)e.getSource();
            button.setEnabled(false);
                }

    }
       }

       public static void main(String s[]) {
    /*
    */

    Client3 panel = new Client3();


            panel.init();

    frame = new JFrame("Client3");
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {System.exit(0);}
    });
    frame.getContentPane().add("Center", panel);
    frame.pack();
    frame.setVisible(true);

       }
}
```

d) Profile.java

```
aspect Profile {

    pointcut mellons(java.awt.event.ActionEvent event): executions(*
actionPerformed (event));
 pointcut ballsA(A Aobj): calls(* Aobj.*(..));
 pointcut ballsB(B Bobj): calls(* Bobj.*(..));
 pointcut ballsC(C Cobj): calls(* Cobj.*(..));

    before(java.awt.event.ActionEvent event): mellons(event) {
  System.out.println(" *** Event Start at " +
System.currentTimeMillis());
        System.out.println(thisJoinPoint);
  System.out.println(event);
    }
    after(java.awt.event.ActionEvent event): mellons(event) {
  System.out.println(" *** Event End " );
  System.out.println();
 }
 before(A Aobj): ballsA(Aobj) {
```

```
        System.out.println(thisJoinPoint);
 }
 before(B Bobj): ballsB(Bobj) {
        System.out.println(thisJoinPoint);
 }
 before(C Cobj): ballsC(Cobj) {
        System.out.println(thisJoinPoint);
 }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C

A. DETAILED EXPERIMENTAL RESULTS

This section is a detailed listing of the actual numbers measured from the experiments in the testbed. Some of the tables listed earlier where collected from these tables.

1. 4 Concurrent Users, Role 2 (Minimal Memory)

Table 67: Concurrent User 1 of 4 for Role 2 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 1	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14451.689	176	649	359
2	GIGA	GIGA	BR733	11655.629	187	640	357
3	GIGA	BR733	GIGA	11761.639	174	631	379
4	GIGA	BR733	BR733	14199.362	175	647	362
5	BR733	GIGA	GIGA	11660.064	144	632	408
6	BR733	GIGA	BR733	11774.479	151	646	387
7	BR733	BR733	GIGA	16772.198	155	678	351
8	BR733	BR733	BR733	21161.106	177	615	392
9	GIGA	GIGA	SIX	12672.601	156	633	395
10	GIGA	SIX	GIGA	14173.722	162	646	376
11	GIGA	SIX	SIX	18818.979	176	607	401
12	SIX	GIGA	GIGA	12209.472	169	625	390
13	SIX	GIGA	SIX	13791.529	168	628	388
14	SIX	SIX	GIGA	20606.696	176	631	377
15	SIX	SIX	SIX	28740.316	136	638	387
16	BR733	BR733	SIX	17456.344	159	649	376
17	BR733	SIX	BR733	15346.151	173	653	358
18	BR733	SIX	SIX	18496.188	166	654	364
19	SIX	BR733	BR733	15987.398	171	609	404
20	SIX	BR733	SIX	15239.851	179	621	384
21	SIX	SIX	BR733	22227.631	163	636	385
22	GIGA	BR733	SIX	11549.421	156	656	372
23	GIGA	SIX	BR733	14235.364	157	617	410
24	BR733	GIGA	SIX	10365.736	155	630	399

25	BR733	SIX	GIGA	14139.427	148	663	373
26	SIX	GIGA	BR733	10571.078	159	661	364
27	SIX	BR733	GIGA	12565.601	161	630	393

Table 68: Concurrent User 2 of 4 for Role 2 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 2	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14545.361	155	635	394
2	GIGA	GIGA	BR733	11819.111	164	645	375
3	GIGA	BR733	GIGA	11815.286	152	638	394
4	GIGA	BR733	BR733	14774.072	155	650	379
5	BR733	GIGA	GIGA	11346.611	150	653	381
6	BR733	GIGA	BR733	11760.701	157	630	397
7	BR733	BR733	GIGA	17227.631	159	630	395
8	BR733	BR733	BR733	21403.992	170	620	394
9	GIGA	GIGA	SIX	12505.802	151	639	394
10	GIGA	SIX	GIGA	14480.084	163	621	400
11	GIGA	SIX	SIX	18019.294	162	661	361
12	SIX	GIGA	GIGA	12096.166	163	625	396
13	SIX	GIGA	SIX	14411.424	146	636	402
14	SIX	SIX	GIGA	20762.944	171	618	395
15	SIX	SIX	SIX	28669.331	135	621	398
16	BR733	BR733	SIX	17350.652	168	635	381
17	BR733	SIX	BR733	16275.682	162	615	407
18	BR733	SIX	SIX	18960.491	156	649	379
19	SIX	BR733	BR733	15648.187	161	666	357
20	SIX	BR733	SIX	15558.432	159	649	371
21	SIX	SIX	BR733	21830.567	162	648	374
22	GIGA	BR733	SIX	11654.785	161	653	370
23	GIGA	SIX	BR733	13402.626	170	673	341
24	BR733	GIGA	SIX	10350.559	152	635	397
25	BR733	SIX	GIGA	14375.937	145	657	382
26	SIX	GIGA	BR733	10785.552	162	604	418
27	SIX	BR733	GIGA	12905.591	149	630	405

Table 69: Concurrent User 3 of 4 for Role 2 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 3	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14823.641	153	652	379

2	GIGA	GIGA	BR733	11663.735	157	655	372
3	GIGA	BR733	GIGA	11708.873	172	623	389
4	GIGA	BR733	BR733	14195.511	162	648	374
5	BR733	GIGA	GIGA	11233.892	157	641	386
6	BR733	GIGA	BR733	11404.122	177	628	379
7	BR733	BR733	GIGA	17289.391	137	661	386
8	BR733	BR733	BR733	20935.519	159	661	364
9	GIGA	GIGA	SIX	11911.613	142	691	351
10	GIGA	SIX	GIGA	14230.758	151	647	386
11	GIGA	SIX	SIX	18921.204	159	628	392
12	SIX	GIGA	GIGA	11772.084	167	628	389
13	SIX	GIGA	SIX	13689.168	148	647	389
14	SIX	SIX	GIGA	20817.592	163	642	379
15	SIX	SIX	SIX	27637.496	175	623	386
16	BR733	BR733	SIX	17407.274	170	619	395
17	BR733	SIX	BR733	15552.021	162	629	393
18	BR733	SIX	SIX	18821.888	167	636	381
19	SIX	BR733	BR733	15248.937	162	651	371
20	SIX	BR733	SIX	15419.307	149	654	381
21	SIX	SIX	BR733	22058.594	148	630	406
22	GIGA	BR733	SIX	11092.807	178	648	358
23	GIGA	SIX	BR733	13639.815	160	670	354
24	BR733	GIGA	SIX	9915.239	156	647	381
25	BR733	SIX	GIGA	14153.888	158	620	406
26	SIX	GIGA	BR733	10428.958	148	661	375
27	SIX	BR733	GIGA	12638.207	143	655	386

Table 70: Concurrent User 4 of 4 for Role 2 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 4	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14592.881	154	645	385
2	GIGA	GIGA	BR733	11845.931	161	608	415
3	GIGA	BR733	GIGA	11559.884	152	663	369
4	GIGA	BR733	BR733	14163.937	160	651	373
5	BR733	GIGA	GIGA	11100.644	147	642	395
6	BR733	GIGA	BR733	11727.157	171	598	415
7	BR733	BR733	GIGA	16977.487	158	637	389
8	BR733	BR733	BR733	21038.068	163	651	370
9	GIGA	GIGA	SIX	12330.296	137	663	384
10	GIGA	SIX	GIGA	14325.713	147	660	377

11	GIGA	SIX	SIX	17753.542	167	670	347
12	SIX	GIGA	GIGA	12063.461	138	619	427
13	SIX	GIGA	SIX	13647.399	143	666	375
14	SIX	SIX	GIGA	21326.932	138	662	384
15	SIX	SIX	SIX	27430.582	160	671	353
16	BR733	BR733	SIX	17412.761	162	636	386
17	BR733	SIX	BR733	15462.453	162	651	371
18	BR733	SIX	SIX	19766.926	149	635	388
19	SIX	BR733	BR733	15725.789	164	632	388
20	SIX	BR733	SIX	15411.685	162	642	380
21	SIX	SIX	BR733	22485.427	136	649	399
22	GIGA	BR733	SIX	11800.525	163	630	391
23	GIGA	SIX	BR733	13678.247	174	634	376
24	BR733	GIGA	SIX	10174.872	159	622	403
25	BR733	SIX	GIGA	13687.979	164	657	363
26	SIX	GIGA	BR733	10391.285	146	671	367
27	SIX	BR733	GIGA	12168.697	167	652	365

2. 4 Concurrent Users, Role 2 (Maximum Memory)

Table 71: Concurrent User 1 of 4 for Role 2 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 1	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14883.671	158	648	378
2	GIGA	GIGA	BR733	12132.167	155	646	383
3	GIGA	BR733	GIGA	12009.064	154	630	400
4	GIGA	BR733	BR733	15056.073	150	632	402
5	BR733	GIGA	GIGA	11183.435	154	638	392
6	BR733	GIGA	BR733	11263.501	186	638	360
7	BR733	BR733	GIGA	16864.718	166	634	384
8	BR733	BR733	BR733	21332.836	151	649	384
9	GIGA	GIGA	SIX	12595.811	143	627	414
10	GIGA	SIX	GIGA	14243.983	156	643	385
11	GIGA	SIX	SIX	error	error	error	error
12	SIX	GIGA	GIGA	11924.226	154	629	401
13	SIX	GIGA	SIX	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error
16	BR733	BR733	SIX	17713.959	157	629	398
17	BR733	SIX	BR733	15740.461	149	659	376
18	BR733	SIX	SIX	error	error	error	error

19	SIX	BR733	BR733	15621.095	151	649	388
20	SIX	BR733	SIX	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error
22	GIGA	BR733	SIX	11644.662	164	650	370
23	GIGA	SIX	BR733	14034.273	169	617	398
24	BR733	GIGA	SIX	10280.603	171	603	410
25	BR733	SIX	GIGA	14339.144	155	627	402
26	SIX	GIGA	BR733	10798.431	154	650	380
27	SIX	BR733	GIGA	12519.662	149	669	366

Table 72: Concurrent User 2 of 4 for Role 2 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 2	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14933.714	152	634	398
2	GIGA	GIGA	BR733	12020.929	146	653	385
3	GIGA	BR733	GIGA	11648.953	177	628	379
4	GIGA	BR733	BR733	14670.187	170	620	394
5	BR733	GIGA	GIGA	11445.787	152	621	411
6	BR733	GIGA	BR733	11170.028	182	648	354
7	BR733	BR733	GIGA	16622.558	180	641	363
8	BR733	BR733	BR733	21044.405	142	683	359
9	GIGA	GIGA	SIX	12136.321	164	677	343
10	GIGA	SIX	GIGA	13979.017	164	653	367
11	GIGA	SIX	SIX	error	error	error	error
12	SIX	GIGA	GIGA	11810.413	153	669	362
13	SIX	GIGA	SIX	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error
16	BR733	BR733	SIX	17526.961	162	655	367
17	BR733	SIX	BR733	14876.307	191	651	342
18	BR733	SIX	SIX	error	error	error	error
19	SIX	BR733	BR733	15820.009	148	650	386
20	SIX	BR733	SIX	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error
22	GIGA	BR733	SIX	11641.059	173	631	380
23	GIGA	SIX	BR733	14130.437	174	623	387
24	BR733	GIGA	SIX	10323.572	156	623	405
25	BR733	SIX	GIGA	14028.612	160	646	378
26	SIX	GIGA	BR733	10569.229	177	638	369
27	SIX	BR733	GIGA	12379.026	163	646	375

Table 73: Concurrent User 3 of 4 for Role 2 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 3	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	14860.921	159	621	404
2	GIGA	GIGA	BR733	11971.122	160	628	396
3	GIGA	BR733	GIGA	11876.305	143	643	398
4	GIGA	BR733	BR733	14221.736	178	637	369
5	BR733	GIGA	GIGA	11172.515	155	623	406
6	BR733	GIGA	BR733	11452.427	156	647	381
7	BR733	BR733	GIGA	16790.307	171	618	395
8	BR733	BR733	BR733	20901.703	172	644	368
9	GIGA	GIGA	SIX	12233.711	157	653	374
10	GIGA	SIX	GIGA	14275.818	159	631	394
11	GIGA	SIX	SIX	error	error	error	error
12	SIX	GIGA	GIGA	11834.329	156	649	379
13	SIX	GIGA	SIX	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error
16	BR733	BR733	SIX	17705.823	154	620	410
17	BR733	SIX	BR733	15442.351	161	655	368
18	BR733	SIX	SIX	error	error	error	error
19	SIX	BR733	BR733	15515.414	159	646	379
20	SIX	BR733	SIX	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error
22	GIGA	BR733	SIX	11214.995	179	639	366
23	GIGA	SIX	BR733	14339.631	156	632	396
24	BR733	GIGA	SIX	9854.642	157	669	358
25	BR733	SIX	GIGA	13729.628	155	658	371
26	SIX	GIGA	BR733	10651.652	164	636	384
27	SIX	BR733	GIGA	12399.781	149	638	397

Table 74: Concurrent User 4 of 4 for Role 2 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 2 - 4	CALL 1	CALL 2	CALL 3
1	GIGA	GIGA	GIGA	15033.478	135	648	401
2	GIGA	GIGA	BR733	11583.931	183	638	363
3	GIGA	BR733	GIGA	11803.527	154	620	410
4	GIGA	BR733	BR733	14612.986	144	656	384

5	BR733	GIGA	GIGA	11282.204	159	648	377
6	BR733	GIGA	BR733	11505.982	151	645	388
7	BR733	BR733	GIGA	16999.747	161	630	393
8	BR733	BR733	BR733	21080.549	163	647	374
9	GIGA	GIGA	SIX	12230.774	139	663	382
10	GIGA	SIX	GIGA	13991.415	182	619	383
11	GIGA	SIX	SIX	error	error	error	error
12	SIX	GIGA	GIGA	11703.954	165	637	382
13	SIX	GIGA	SIX	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error
16	BR733	BR733	SIX	17570.856	150	650	384
17	BR733	SIX	BR733	15712.248	162	634	388
18	BR733	SIX	SIX	error	error	error	error
19	SIX	BR733	BR733	15055.717	168	657	359
20	SIX	BR733	SIX	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error
22	GIGA	BR733	SIX	11437.994	176	612	396
23	GIGA	SIX	BR733	14284.486	145	649	390
24	BR733	GIGA	SIX	10342.034	161	596	427
25	BR733	SIX	GIGA	14122.083	155	658	371
26	SIX	GIGA	BR733	10768.916	154	640	390
27	SIX	BR733	GIGA	12196.652	179	624	381

3. 3 Concurrent Users, Role 3 (Minimal Memory)

Table 75: Concurrent User 1 of 3 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	16246.644	686	151	446	17	12
2	GIGA	GIGA	BR733	13963.951	725	142	417	13	15
3	GIGA	BR733	GIGA	13267.373	724	154	403	15	16
4	GIGA	BR733	BR733	20505.252	714	136	437	14	11
5	BR733	GIGA	GIGA	14818.586	719	146	421	7	19
6	BR733	GIGA	BR733	15865.816	720	148	415	13	16
7	BR733	BR733	GIGA	13827.755	738	148	405	8	13
8	BR733	BR733	BR733	23604.802	725	142	412	20	13
9	GIGA	GIGA	SIX	16739.459	683	140	467	16	16
10	GIGA	SIX	GIGA	14250.071	711	125	445	14	17
11	GIGA	SIX	SIX	26007.719	739	125	419	10	19
12	SIX	GIGA	GIGA	14858.469	699	150	444	14	5

13	SIX	GIGA	SIX	19012.541	697	140	444	18	13
14	SIX	SIX	GIGA	16098.039	692	160	431	13	16
15	SIX	SIX	SIX	29812.155	717	150	411	16	18
16	BR733	BR733	SIX	18284.001	717	139	428	11	17
17	BR733	SIX	BR733	17787.777	743	123	412	12	13
18	BR733	SIX	SIX	25697.396	698	145	433	20	16
19	SIX	BR733	BR733	20879.311	717	144	418	12	21
20	SIX	BR733	SIX	20165.539	729	128	429	11	15
21	SIX	SIX	BR733	18058.069	706	148	430	15	13
22	GIGA	BR733	SIX	16990.915	706	152	423	18	13
23	GIGA	SIX	BR733	16013.648	704	146	434	15	13
24	BR733	GIGA	SIX	16122.025	737	137	412	15	11
25	BR733	SIX	GIGA	13837.391	701	150	433	12	16
26	SIX	GIGA	BR733	13986.045	703	155	428	14	12
27	SIX	BR733	GIGA	12540.704	682	143	463	10	14

Table 76: Concurrent User 2 of 3 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15990.752	713	138	422	26	13
2	GIGA	GIGA	BR733	13780.454	709	140	432	18	13
3	GIGA	BR733	GIGA	13007.495	715	147	431	6	13
4	GIGA	BR733	BR733	20389.089	692	152	438	15	15
5	BR733	GIGA	GIGA	14589.387	730	138	424	10	10
6	BR733	GIGA	BR733	15706.083	730	159	401	10	12
7	BR733	BR733	GIGA	13480.751	715	142	413	20	22
8	BR733	BR733	BR733	22883.931	671	135	473	17	16
9	GIGA	GIGA	SIX	16625.179	674	151	453	19	15
10	GIGA	SIX	GIGA	14309.093	735	135	412	14	16
11	GIGA	SIX	SIX	25599.683	694	134	441	19	24
12	SIX	GIGA	GIGA	14398.454	714	126	445	18	9
13	SIX	GIGA	SIX	19080.243	725	148	407	13	19
14	SIX	SIX	GIGA	15888.729	727	139	415	13	18
15	SIX	SIX	SIX	29696.247	726	151	403	20	12
16	BR733	BR733	SIX	18161.879	698	132	451	13	18
17	BR733	SIX	BR733	17807.723	700	163	421	13	15
18	BR733	SIX	SIX	25797.531	735	129	426	10	12
19	SIX	BR733	BR733	20826.849	717	156	413	15	11
20	SIX	BR733	SIX	19780.252	719	145	411	20	17
21	SIX	SIX	BR733	17877.428	747	124	406	14	21

22	GIGA	BR733	SIX	16821.708	702	144	435	16	15
23	GIGA	SIX	BR733	16154.549	748	138	400	11	15
24	BR733	GIGA	SIX	16020.247	706	148	428	15	15
25	BR733	SIX	GIGA	13576.473	713	147	429	12	11
26	SIX	GIGA	BR733	13781.478	746	120	423	15	8
27	SIX	BR733	GIGA	12305.581	734	134	415	18	11

Table 77: Concurrent User 3 of 3 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 3	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15698.527	689	118	468	18	19
2	GIGA	GIGA	BR733	14033.453	711	146	431	14	10
3	GIGA	BR733	GIGA	12923.765	684	158	442	11	17
4	GIGA	BR733	BR733	20352.082	746	118	430	10	8
5	BR733	GIGA	GIGA	14435.768	737	137	409	16	13
6	BR733	GIGA	BR733	15617.443	697	162	424	15	14
7	BR733	BR733	GIGA	13539.598	706	148	426	21	11
8	BR733	BR733	BR733	23471.339	724	154	397	19	18
9	GIGA	GIGA	SIX	16547.723	731	143	414	13	11
10	GIGA	SIX	GIGA	14184.269	725	135	427	11	14
11	GIGA	SIX	SIX	25782.595	732	133	414	18	15
12	SIX	GIGA	GIGA	14402.232	692	136	450	19	15
13	SIX	GIGA	SIX	18996.226	691	152	444	12	13
14	SIX	SIX	GIGA	15594.872	726	136	420	20	10
15	SIX	SIX	SIX	31538.926	737	156	387	9	10
16	BR733	BR733	SIX	17985.032	688	141	440	22	21
17	BR733	SIX	BR733	17443.265	689	148	449	17	9
18	BR733	SIX	SIX	26176.596	701	160	422	13	16
19	SIX	BR733	BR733	20493.055	699	143	446	15	9
20	SIX	BR733	SIX	19698.997	702	138	448	13	11
21	SIX	SIX	BR733	18225.848	728	148	408	16	12
22	GIGA	BR733	SIX	16988.883	713	155	417	11	16
23	GIGA	SIX	BR733	15809.621	708	137	438	16	13
24	BR733	GIGA	SIX	15952.375	702	142	432	18	18
25	BR733	SIX	GIGA	13572.124	702	159	423	19	9
26	SIX	GIGA	BR733	13750.369	706	144	431	17	14
27	SIX	BR733	GIGA	12617.786	658	166	463	11	14

4. 4 Concurrent Users, Role 3 (Maximum Memory)

Table 78: Concurrent User 1 of 3 for Role 3 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	error	error	error	error	error	error
2	GIGA	GIGA	BR733	14336.699	754	152	376	16	14
3	GIGA	BR733	GIGA	13136.291	727	128	433	11	13
4	GIGA	BR733	BR733	20606.278	742	129	413	11	17
5	BR733	GIGA	GIGA	14543.911	709	149	428	15	11
6	BR733	GIGA	BR733	15417.885	718	139	418	20	17
7	BR733	BR733	GIGA	13574.957	728	141	422	12	9
8	BR733	BR733	BR733	error	error	error	error	error	error
9	GIGA	GIGA	SIX	16629.311	688	139	446	21	18
10	GIGA	SIX	GIGA	14533.179	703	147	438	14	10
11	GIGA	SIX	SIX	error	error	error	error	error	error
12	SIX	GIGA	GIGA	14478.122	699	147	443	11	12
13	SIX	GIGA	SIX	error	error	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error	error	error
16	BR733	BR733	SIX	17908.587	695	146	441	14	16
17	BR733	SIX	BR733	17659.799	691	150	444	13	14
18	BR733	SIX	SIX	error	error	error	error	error	error
19	SIX	BR733	BR733	20548.803	697	137	443	17	18
20	SIX	BR733	SIX	error	error	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error	error	error
22	GIGA	BR733	SIX	16868.892	688	143	448	21	12
23	GIGA	SIX	BR733	15946.944	748	137	393	17	17
24	BR733	GIGA	SIX	16299.425	729	128	431	15	9
25	BR733	SIX	GIGA	13717.001	771	130	381	14	16
26	SIX	GIGA	BR733	13666.717	699	132	448	19	14
27	SIX	BR733	GIGA	12509.669	712	142	419	20	19

Table 79: Concurrent User 2 of 3 for Role 3 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	error	error	error	error	error	error
2	GIGA	GIGA	BR733	13931.844	740	127	430	7	8
3	GIGA	BR733	GIGA	13359.789	741	147	400	11	13
4	GIGA	BR733	BR733	20473.864	717	137	431	14	13

5	BR733	GIGA	GIGA	14527.849	753	135	393	15	16
6	BR733	GIGA	BR733	15432.652	677	150	452	18	15
7	BR733	BR733	GIGA	13561.104	700	152	431	16	13
8	BR733	BR733	BR733	error	error	error	error	error	error
9	GIGA	GIGA	SIX	16576.898	731	143	411	14	13
10	GIGA	SIX	GIGA	14572.614	714	165	412	13	8
11	GIGA	SIX	SIX	error	error	error	error	error	error
12	SIX	GIGA	GIGA	14451.922	726	129	428	20	9
13	SIX	GIGA	SIX	error	error	error	error	error	error
14	SIX	SIX	GIGA	error	error	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error	error	error
16	BR733	BR733	SIX	18006.809	729	138	422	13	10
17	BR733	SIX	BR733	17742.296	695	165	427	12	13
18	BR733	SIX	SIX	error	error	error	error	error	error
19	SIX	BR733	BR733	20672.288	688	164	428	14	18
20	SIX	BR733	SIX	error	error	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error	error	error
22	GIGA	BR733	SIX	16973.796	700	143	444	15	10
23	GIGA	SIX	BR733	15710.841	685	137	449	21	20
24	BR733	GIGA	SIX	16165.136	728	123	429	15	17
25	BR733	SIX	GIGA	13847.558	718	164	388	19	23
26	SIX	GIGA	BR733	13761.348	741	133	401	20	17
27	SIX	BR733	GIGA	12420.361	711	141	425	18	17

Table 80: Concurrent User 3 of 3 for Role 3 (Maximum Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 3	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	error	error	error	error	error	error
2	GIGA	GIGA	BR733	14048.615	712	147	421	14	18
3	GIGA	BR733	GIGA	13455.508	739	164	380	17	12
4	GIGA	BR733	BR733	20540.406	736	142	416	8	10
5	BR733	GIGA	GIGA	14343.026	728	127	420	18	19
6	BR733	GIGA	BR733	15312.208	703	135	443	15	16
7	BR733	BR733	GIGA	13310.879	714	137	439	9	13
8	BR733	BR733	BR733	error	error	error	error	error	error
9	GIGA	GIGA	SIX	16640.138	698	142	438	14	20
10	GIGA	SIX	GIGA	14336.485	726	126	430	13	17
11	GIGA	SIX	SIX	error	error	error	error	error	error
12	SIX	GIGA	GIGA	14422.797	691	138	454	13	16
13	SIX	GIGA	SIX	error	error	error	error	error	error

14	SIX	SIX	GIGA	error	error	error	error	error	error
15	SIX	SIX	SIX	error	error	error	error	error	error
16	BR733	BR733	SIX	17946.477	713	132	429	23	15
17	BR733	SIX	BR733	17508.559	730	129	424	17	12
18	BR733	SIX	SIX	error	error	error	error	error	error
19	SIX	BR733	BR733	20936.006	728	151	403	13	17
20	SIX	BR733	SIX	error	error	error	error	error	error
21	SIX	SIX	BR733	error	error	error	error	error	error
22	GIGA	BR733	SIX	16847.396	751	108	435	10	8
23	GIGA	SIX	BR733	15828.061	727	139	416	15	15
24	BR733	GIGA	SIX	16106.642	715	136	420	26	15
25	BR733	SIX	GIGA	13501.525	683	166	427	22	14
26	SIX	GIGA	BR733	13644.586	714	144	418	18	18
27	SIX	BR733	GIGA	12513.811	664	175	442	15	16

5. 28 Concurrent Users, Role 1 (Minimal Memory)

Table 81: Concurrent User 1 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 1	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9178.232	802	16	12	18
2	GIGA	GIGA	BR733	5109.871	796	21	16	15
3	GIGA	BR733	GIGA	4476.100	797	21	20	10
4	GIGA	BR733	BR733	3740.086	804	13	12	19
5	BR733	GIGA	GIGA	6911.242	797	10	21	20
6	BR733	GIGA	BR733	14202.297	797	22	11	18
7	BR733	BR733	GIGA	11186.045	792	20	14	22
22	GIGA	BR733	SIX	3635.077	798	14	17	19
23	GIGA	SIX	BR733	3085.810	790	25	23	10
24	BR733	GIGA	SIX	7539.690	791	16	21	20
25	BR733	SIX	GIGA	6787.718	802	12	16	18
26	SIX	GIGA	BR733	11260.134	796	18	14	20
27	SIX	BR733	GIGA	11918.283	806	13	14	15

Table 82: Concurrent User 2 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 2	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	8603.737	814	12	11	11
2	GIGA	GIGA	BR733	4905.586	802	11	20	15
3	GIGA	BR733	GIGA	4319.667	802	15	15	16

4	GIGA	BR733	BR733	4007.496	798	16	23	11
5	BR733	GIGA	GIGA	6989.296	798	17	13	20
6	BR733	GIGA	BR733	14125.902	804	17	14	13
7	BR733	BR733	GIGA	10845.575	803	19	13	13
22	GIGA	BR733	SIX	3143.351	811	12	10	15
23	GIGA	SIX	BR733	2911.742	804	18	11	15
24	BR733	GIGA	SIX	7355.007	805	9	19	15
25	BR733	SIX	GIGA	6773.290	805	14	14	15
26	SIX	GIGA	BR733	11120.612	804	14	10	20
27	SIX	BR733	GIGA	12275.413	797	22	17	12

Table 83: Concurrent User 3 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 3	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9676.354	791	21	20	16
2	GIGA	GIGA	BR733	4680.572	809	9	18	12
3	GIGA	BR733	GIGA	4137.342	812	13	13	10
4	GIGA	BR733	BR733	3788.418	797	17	19	15
5	BR733	GIGA	GIGA	6971.090	803	14	13	18
6	BR733	GIGA	BR733	15191.259	787	21	21	19
7	BR733	BR733	GIGA	10826.670	796	18	21	13
22	GIGA	BR733	SIX	3977.782	778	27	25	18
23	GIGA	SIX	BR733	3106.314	795	14	21	18
24	BR733	GIGA	SIX	7381.448	806	20	12	10
25	BR733	SIX	GIGA	6847.231	805	15	13	15
26	SIX	GIGA	BR733	11139.519	802	15	15	16
27	SIX	BR733	GIGA	12112.728	803	16	22	7

Table 84: Concurrent User 4 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 4	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9139.573	811	6	20	11
2	GIGA	GIGA	BR733	5077.018	796	18	19	15
3	GIGA	BR733	GIGA	4298.608	802	13	22	11
4	GIGA	BR733	BR733	3894.329	795	20	17	16
5	BR733	GIGA	GIGA	7172.389	803	23	10	12
6	BR733	GIGA	BR733	14517.008	797	16	17	18
7	BR733	BR733	GIGA	10864.498	798	14	17	19
22	GIGA	BR733	SIX	3661.555	790	19	25	14

23	GIGA	SIX	BR733	2889.429	806	11	16	15
24	BR733	GIGA	SIX	7399.542	806	13	10	19
25	BR733	SIX	GIGA	6783.554	807	12	11	18
26	SIX	GIGA	BR733	11168.942	803	16	14	15
27	SIX	BR733	GIGA	12101.691	803	16	16	13

Table 85: Concurrent User 5 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 5	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9279.203	802	19	13	14
2	GIGA	GIGA	BR733	4920.733	797	24	16	11
3	GIGA	BR733	GIGA	4157.761	806	12	13	17
4	GIGA	BR733	BR733	3784.397	800	16	16	16
5	BR733	GIGA	GIGA	6913.417	815	14	11	8
6	BR733	GIGA	BR733	14561.282	800	19	15	14
7	BR733	BR733	GIGA	11051.554	795	20	16	17
22	GIGA	BR733	SIX	3504.259	808	15	15	10
23	GIGA	SIX	BR733	3077.257	791	18	21	18
24	BR733	GIGA	SIX	7384.009	801	18	17	14
25	BR733	SIX	GIGA	6936.248	801	22	12	13
26	SIX	GIGA	BR733	10824.664	805	8	13	22
27	SIX	BR733	GIGA	11999.006	802	13	21	12

Table 86: Concurrent User 6 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 6	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	10070.017	790	15	25	18
2	GIGA	GIGA	BR733	5020.764	798	14	20	16
3	GIGA	BR733	GIGA	4168.264	807	13	12	16
4	GIGA	BR733	BR733	3698.759	799	20	12	17
5	BR733	GIGA	GIGA	7201.873	789	22	14	23
6	BR733	GIGA	BR733	14330.079	804	13	15	16
7	BR733	BR733	GIGA	10995.596	794	13	17	24
22	GIGA	BR733	SIX	3549.955	796	15	16	21
23	GIGA	SIX	BR733	3038.972	799	17	11	21
24	BR733	GIGA	SIX	7420.456	793	16	12	27
25	BR733	SIX	GIGA	6832.768	797	20	19	12
26	SIX	GIGA	BR733	11053.914	800	12	18	18
27	SIX	BR733	GIGA	12072.745	807	16	13	12

Table 87: Concurrent User 7 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 7	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9612.802	797	16	20	15
2	GIGA	GIGA	BR733	4966.394	801	13	11	23
3	GIGA	BR733	GIGA	4184.125	803	15	13	17
4	GIGA	BR733	BR733	3820.460	799	21	17	11
5	BR733	GIGA	GIGA	7011.368	794	12	27	15
6	BR733	GIGA	BR733	14333.894	798	20	12	18
7	BR733	BR733	GIGA	11086.953	794	19	14	21
22	GIGA	BR733	SIX	3721.844	803	13	15	17
23	GIGA	SIX	BR733	2928.159	805	17	6	20
24	BR733	GIGA	SIX	7518.795	805	18	17	8
25	BR733	SIX	GIGA	6874.393	786	19	18	25
26	SIX	GIGA	BR733	11154.059	802	14	17	15
27	SIX	BR733	GIGA	12192.624	799	20	11	18

Table 88: Concurrent User 8 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 8	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9374.465	803	11	22	12
2	GIGA	GIGA	BR733	4963.131	806	16	9	17
3	GIGA	BR733	GIGA	4225.843	804	13	10	21
4	GIGA	BR733	BR733	3885.889	799	19	16	14
5	BR733	GIGA	GIGA	7084.619	428	8	5	10
6	BR733	GIGA	BR733	14746.423	799	12	19	18
7	BR733	BR733	GIGA	10648.272	810	15	11	12
22	GIGA	BR733	SIX	3613.798	799	16	19	14
23	GIGA	SIX	BR733	3073.204	795	11	21	21
24	BR733	GIGA	SIX	7386.021	797	15	18	18
25	BR733	SIX	GIGA	6992.200	797	17	16	18
26	SIX	GIGA	BR733	11345.300	801	22	13	12
27	SIX	BR733	GIGA	12169.627	804	18	16	10

Table 89: Concurrent User 9 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 9	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9495.985	795	16	19	18

2	GIGA	GIGA	BR733	4845.719	808	11	12	17
3	GIGA	BR733	GIGA	4044.884	813	15	10	10
4	GIGA	BR733	BR733	3861.124	796	20	15	17
5	BR733	GIGA	GIGA	7087.136	787	17	24	20
6	BR733	GIGA	BR733	14382.371	800	16	16	16
7	BR733	BR733	GIGA	10599.995	808	18	12	10
22	GIGA	BR733	SIX	3737.231	790	16	20	22
23	GIGA	SIX	BR733	2846.454	811	15	10	12
24	BR733	GIGA	SIX	7594.449	793	19	19	17
25	BR733	SIX	GIGA	6800.597	808	15	11	14
26	SIX	GIGA	BR733	11097.289	803	15	10	20
27	SIX	BR733	GIGA	12049.730	798	14	16	20

Table 90: Concurrent User 10 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 10	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9187.702	801	17	14	16
2	GIGA	GIGA	BR733	4939.618	801	11	12	24
3	GIGA	BR733	GIGA	4537.585	797	17	25	9
4	GIGA	BR733	BR733	3928.565	800	15	19	14
5	BR733	GIGA	GIGA	6819.575	809	10	16	13
6	BR733	GIGA	BR733	14432.634	804	13	18	13
7	BR733	BR733	GIGA	10574.862	803	14	20	11
22	GIGA	BR733	SIX	3432.399	800	14	16	18
23	GIGA	SIX	BR733	3142.797	790	18	18	22
24	BR733	GIGA	SIX	7421.564	802	15	14	17
25	BR733	SIX	GIGA	6888.947	798	16	16	18
26	SIX	GIGA	BR733	10980.380	808	12	13	15
27	SIX	BR733	GIGA	12088.591	796	17	13	22

Table 91: Concurrent User 11 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 11	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9513.691	797	14	12	25
2	GIGA	GIGA	BR733	5271.303	785	20	27	16
3	GIGA	BR733	GIGA	4508.808	795	19	20	14
4	GIGA	BR733	BR733	3694.613	803	16	15	14
5	BR733	GIGA	GIGA	6834.869	808	9	16	15

6	BR733	GIGA	BR733	14863.907	788	20	19	21
7	BR733	BR733	GIGA	10992.112	794	17	18	19
22	GIGA	BR733	SIX	3434.828	811	10	15	12
23	GIGA	SIX	BR733	3175.429	795	18	21	14
24	BR733	GIGA	SIX	7312.204	808	13	14	13
25	BR733	SIX	GIGA	6855.449	803	14	15	16
26	SIX	GIGA	BR733	11294.183	794	18	12	24
27	SIX	BR733	GIGA	12121.774	802	16	15	15

Table 92: Concurrent User 12 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 12	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	8978.552	810	12	11	15
2	GIGA	GIGA	BR733	5119.768	800	21	9	18
3	GIGA	BR733	GIGA	4491.188	800	14	14	20
4	GIGA	BR733	BR733	3684.994	802	8	17	21
5	BR733	GIGA	GIGA	7108.029	794	18	14	22
6	BR733	GIGA	BR733	13973.617	812	10	13	13
7	BR733	BR733	GIGA	10638.802	808	13	13	14
22	GIGA	BR733	SIX	3311.172	811	6	15	16
23	GIGA	SIX	BR733	2938.921	800	12	17	19
24	BR733	GIGA	SIX	7343.731	807	16	11	14
25	BR733	SIX	GIGA	6783.001	806	15	18	9
26	SIX	GIGA	BR733	11332.288	793	24	16	15
27	SIX	BR733	GIGA	12008.052	807	14	10	17

Table 93: Concurrent User 13 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 13	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9593.938	794	17	16	21
2	GIGA	GIGA	BR733	4829.911	808	16	12	12
3	GIGA	BR733	GIGA	4123.746	806	18	11	13
4	GIGA	BR733	BR733	3750.092	801	16	15	16
5	BR733	GIGA	GIGA	6956.586	805	13	14	16
6	BR733	GIGA	BR733	14490.620	796	21	13	18
7	BR733	BR733	GIGA	11298.495	789	20	13	26
22	GIGA	BR733	SIX	3413.420	806	13	16	13
23	GIGA	SIX	BR733	3271.409	786	20	21	21

24	BR733	GIGA	SIX	7483.231	794	15	26	13
25	BR733	SIX	GIGA	6706.737	803	14	18	13
26	SIX	GIGA	BR733	11292.051	801	18	16	13
27	SIX	BR733	GIGA	11935.436	802	11	20	15

Table 94: Concurrent User 14 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 14	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9233.563	802	16	14	16
2	GIGA	GIGA	BR733	5106.188	794	15	18	21
3	GIGA	BR733	GIGA	4172.130	806	11	14	17
4	GIGA	BR733	BR733	3986.954	797	14	19	18
5	BR733	GIGA	GIGA	7041.827	801	15	15	17
6	BR733	GIGA	BR733	14626.621	797	19	16	16
7	BR733	BR733	GIGA	10668.779	799	13	20	16
22	GIGA	BR733	SIX	3440.297	802	10	15	21
23	GIGA	SIX	BR733	3186.689	786	20	21	21
24	BR733	GIGA	SIX	7416.789	796	15	19	18
25	BR733	SIX	GIGA	6847.840	792	16	18	22
26	SIX	GIGA	BR733	11051.426	803	16	11	18
27	SIX	BR733	GIGA	12039.962	795	15	24	14

Table 95: Concurrent User 15 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 15	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9052.777	806	13	13	16
2	GIGA	GIGA	BR733	5079.588	796	18	11	23
3	GIGA	BR733	GIGA	4356.302	797	18	18	15
4	GIGA	BR733	BR733	3800.627	788	24	16	20
5	BR733	GIGA	GIGA	6808.320	807	10	13	18
6	BR733	GIGA	BR733	14515.053	797	21	17	13
7	BR733	BR733	GIGA	11022.959	790	22	20	16
22	GIGA	BR733	SIX	3598.483	795	16	16	21
23	GIGA	SIX	BR733	2854.433	808	11	15	14
24	BR733	GIGA	SIX	7150.519	812	10	11	15
25	BR733	SIX	GIGA	6873.507	802	17	14	15
26	SIX	GIGA	BR733	11261.820	789	20	25	14
27	SIX	BR733	GIGA	12145.129	791	21	23	13

Table 96: Concurrent User 16 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 16	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9328.231	800	16	15	17
2	GIGA	GIGA	BR733	5179.360	789	17	22	20
3	GIGA	BR733	GIGA	4248.849	804	18	16	10
4	GIGA	BR733	BR733	3856.458	800	19	17	12
5	BR733	GIGA	GIGA	6906.144	795	14	13	26
6	BR733	GIGA	BR733	13910.340	807	18	12	11
7	BR733	BR733	GIGA	10706.765	802	11	18	17
22	GIGA	BR733	SIX	3528.989	800	11	20	17
23	GIGA	SIX	BR733	2811.132	804	14	22	8
24	BR733	GIGA	SIX	7404.834	807	12	19	10
25	BR733	SIX	GIGA	6867.564	797	16	17	18
26	SIX	GIGA	BR733	11000.376	807	11	16	14
27	SIX	BR733	GIGA	11805.430	808	13	11	16

Table 97: Concurrent User 17 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 17	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9162.899	805	14	17	12
2	GIGA	GIGA	BR733	4826.354	804	14	16	14
3	GIGA	BR733	GIGA	4350.955	801	14	17	16
4	GIGA	BR733	BR733	3746.482	800	19	12	17
5	BR733	GIGA	GIGA	7113.336	794	18	20	16
6	BR733	GIGA	BR733	14942.534	787	21	19	21
7	BR733	BR733	GIGA	10561.485	810	18	11	9
22	GIGA	BR733	SIX	3634.429	796	11	32	9
23	GIGA	SIX	BR733	3005.498	797	17	17	17
24	BR733	GIGA	SIX	7553.279	791	26	20	11
25	BR733	SIX	GIGA	6723.081	808	15	10	15
26	SIX	GIGA	BR733	11252.500	787	19	16	26
27	SIX	BR733	GIGA	12135.861	804	21	10	13

Table 98: Concurrent User 18 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 -	CALL 1	CALL 2	CALL 3	CALL 4

				18				
1	GIGA	GIGA	GIGA	9407.225	798	14	19	17
2	GIGA	GIGA	BR733	4962.686	800	12	22	14
3	GIGA	BR733	GIGA	4387.107	795	20	18	15
4	GIGA	BR733	BR733	3792.724	796	19	20	13
5	BR733	GIGA	GIGA	7107.847	797	18	14	19
6	BR733	GIGA	BR733	14094.248	804	12	12	20
7	BR733	BR733	GIGA	10984.758	800	18	12	18
22	GIGA	BR733	SIX	3504.289	791	21	21	15
23	GIGA	SIX	BR733	2976.400	801	17	14	18
24	BR733	GIGA	SIX	7338.333	807	15	14	12
25	BR733	SIX	GIGA	6800.167	805	14	8	21
26	SIX	GIGA	BR733	11254.031	797	23	12	16
27	SIX	BR733	GIGA	12178.639	789	23	22	14

Table 99: Concurrent User 19 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 19	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9486.146	799	13	18	18
2	GIGA	GIGA	BR733	4994.129	795	16	17	20
3	GIGA	BR733	GIGA	4507.923	793	22	17	16
4	GIGA	BR733	BR733	3726.900	799	17	13	19
5	BR733	GIGA	GIGA	7013.532	800	14	18	16
6	BR733	GIGA	BR733	14326.866	798	20	13	17
7	BR733	BR733	GIGA	10515.992	804	12	17	15
22	GIGA	BR733	SIX	3703.022	785	21	17	25
23	GIGA	SIX	BR733	3086.134	786	19	19	24
24	BR733	GIGA	SIX	7389.100	802	17	14	15
25	BR733	SIX	GIGA	6773.989	794	11	16	27
26	SIX	GIGA	BR733	11121.183	800	19	15	14
27	SIX	BR733	GIGA	11986.149	794	17	17	20

Table 100: Concurrent User 20 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 20	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9099.515	806	13	16	13
2	GIGA	GIGA	BR733	4909.933	800	17	15	16
3	GIGA	BR733	GIGA	4391.554	801	12	22	13

4	GIGA	BR733	BR733	3736.166	794	18	18	18
5	BR733	GIGA	GIGA	6970.841	789	13	23	23
6	BR733	GIGA	BR733	14057.644	806	12	14	16
7	BR733	BR733	GIGA	10177.758	807	6	21	14
22	GIGA	BR733	SIX	3613.895	794	19	20	15
23	GIGA	SIX	BR733	2803.700	806	14	13	15
24	BR733	GIGA	SIX	7554.838	792	23	19	14
25	BR733	SIX	GIGA	6918.355	794	18	24	12
26	SIX	GIGA	BR733	10835.020	144	0	4	2
27	SIX	BR733	GIGA	11981.413	795	16	17	20

Table 101: Concurrent User 21 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 21	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	8799.029	810	10	14	14
2	GIGA	GIGA	BR733	4734.704	810	14	16	8
3	GIGA	BR733	GIGA	4348.241	800	15	16	17
4	GIGA	BR733	BR733	3801.838	796	17	18	17
5	BR733	GIGA	GIGA	6918.083	807	16	15	10
6	BR733	GIGA	BR733	14601.978	796	22	17	13
7	BR733	BR733	GIGA	10701.781	803	14	15	16
22	GIGA	BR733	SIX	3381.428	796	21	12	19
23	GIGA	SIX	BR733	3097.646	789	23	19	17
24	BR733	GIGA	SIX	7327.733	811	16	7	14
25	BR733	SIX	GIGA	6766.046	797	17	13	21
26	SIX	GIGA	BR733	11203.732	794	19	24	11
27	SIX	BR733	GIGA	12096.511	787	18	20	23

Table 102: Concurrent User 22 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 22	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9401.825	797	20	13	18
2	GIGA	GIGA	BR733	5253.776	792	18	15	23
3	GIGA	BR733	GIGA	4409.684	803	14	13	18
4	GIGA	BR733	BR733	3394.958	809	16	11	12
5	BR733	GIGA	GIGA	7142.710	779	18	22	29
6	BR733	GIGA	BR733	14249.206	803	15	13	17
7	BR733	BR733	GIGA	10628.282	805	11	15	17

22	GIGA	BR733	SIX	3530.539	799	21	13	15
23	GIGA	SIX	BR733	2828.729	798	17	14	19
24	BR733	GIGA	SIX	7415.949	802	12	16	18
25	BR733	SIX	GIGA	6867.336	798	21	15	14
26	SIX	GIGA	BR733	10926.233	808	11	12	17
27	SIX	BR733	GIGA	11935.325	802	16	18	12

Table 103: Concurrent User 23 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 23	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9089.650	807	14	12	15
2	GIGA	GIGA	BR733	5095.737	793	20	16	19
3	GIGA	BR733	GIGA	4515.783	790	21	19	18
4	GIGA	BR733	BR733	3994.274	789	18	24	17
5	BR733	GIGA	GIGA	6998.685	794	15	23	16
6	BR733	GIGA	BR733	14180.440	802	15	13	18
7	BR733	BR733	GIGA	10924.264	800	22	11	15
22	GIGA	BR733	SIX	3480.108	797	21	14	16
23	GIGA	SIX	BR733	2834.038	808	11	15	14
24	BR733	GIGA	SIX	7492.402	787	18	21	22
25	BR733	SIX	GIGA	6810.151	804	21	9	14
26	SIX	GIGA	BR733	11010.337	804	11	13	20
27	SIX	BR733	GIGA	12031.633	800	16	17	15

Table 104: Concurrent User 24 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 24	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9035.713	807	13	16	12
2	GIGA	GIGA	BR733	4828.422	805	14	15	14
3	GIGA	BR733	GIGA	4299.700	795	19	19	15
4	GIGA	BR733	BR733	3556.662	801	16	12	19
5	BR733	GIGA	GIGA	7123.281	795	19	20	14
6	BR733	GIGA	BR733	14471.054	800	17	15	16
7	BR733	BR733	GIGA	11057.421	793	27	16	12
22	GIGA	BR733	SIX	3547.612	797	19	16	16
23	GIGA	SIX	BR733	2934.777	802	15	16	15
24	BR733	GIGA	SIX	7261.271	807	10	15	16
25	BR733	SIX	GIGA	6603.480	814	10	12	12

26	SIX	GIGA	BR733	11094.561	799	15	19	15
27	SIX	BR733	GIGA	11908.975	808	14	14	12

Table 105: Concurrent User 25 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 25	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9672.189	795	18	20	15
2	GIGA	GIGA	BR733	5079.658	789	24	22	13
3	GIGA	BR733	GIGA	4459.261	797	14	18	19
4	GIGA	BR733	BR733	3521.693	809	11	14	14
5	BR733	GIGA	GIGA	6969.261	803	15	15	15
6	BR733	GIGA	BR733	14772.421	792	13	19	24
7	BR733	BR733	GIGA	10548.729	803	12	19	14
22	GIGA	BR733	SIX	3197.118	810	13	13	12
23	GIGA	SIX	BR733	2980.847	795	21	19	13
24	BR733	GIGA	SIX	7565.018	784	18	27	19
25	BR733	SIX	GIGA	6764.145	806	18	12	12
26	SIX	GIGA	BR733	10784.303	813	10	7	18
27	SIX	BR733	GIGA	11880.313	807	13	12	16

Table 106: Concurrent User 26 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 26	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9649.703	796	16	19	17
2	GIGA	GIGA	BR733	4706.691	808	17	12	11
3	GIGA	BR733	GIGA	4542.650	801	18	12	17
4	GIGA	BR733	BR733	3939.724	798	12	19	19
5	BR733	GIGA	GIGA	6986.968	806	12	18	12
6	BR733	GIGA	BR733	15086.696	790	23	21	14
7	BR733	BR733	GIGA	10874.321	806	22	8	12
22	GIGA	BR733	SIX	3792.389	794	17	25	12
23	GIGA	SIX	BR733	2846.749	809	17	11	11
24	BR733	GIGA	SIX	7338.719	801	20	18	9
25	BR733	SIX	GIGA	6795.007	801	18	12	17
26	SIX	GIGA	BR733	11089.231	810	14	16	8
27	SIX	BR733	GIGA	12216.133	796	23	15	14

Table 107: Concurrent User 27 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 27	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9151.184	803	16	14	15
2	GIGA	GIGA	BR733	4724.186	809	14	16	9
3	GIGA	BR733	GIGA	4636.755	791	12	23	22
4	GIGA	BR733	BR733	4062.321	789	21	20	18
5	BR733	GIGA	GIGA	6924.767	806	14	13	15
6	BR733	GIGA	BR733	14298.465	801	12	14	21
7	BR733	BR733	GIGA	10938.171	794	21	18	15
22	GIGA	BR733	SIX	3542.923	797	14	20	17
23	GIGA	SIX	BR733	3002.500	804	15	12	17
24	BR733	GIGA	SIX	7334.496	801	18	13	16
25	BR733	SIX	GIGA	6577.518	815	10	13	10
26	SIX	GIGA	BR733	11242.744	794	19	16	19
27	SIX	BR733	GIGA	11909.113	800	15	13	20

Table 108: Concurrent User 28 of 28 for Role 1 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 1 - 28	CALL 1	CALL 2	CALL 3	CALL 4
1	GIGA	GIGA	GIGA	9327.054	802	15	18	13
2	GIGA	GIGA	BR733	4880.574	802	15	14	17
3	GIGA	BR733	GIGA	4044.652	810	11	13	14
4	GIGA	BR733	BR733	3644.711	800	7	20	21
5	BR733	GIGA	GIGA	7080.022	794	18	17	19
6	BR733	GIGA	BR733	13911.316	798	26	9	15
7	BR733	BR733	GIGA	10776.119	802	14	16	16
22	GIGA	BR733	SIX	3735.594	795	22	16	15
23	GIGA	SIX	BR733	3079.217	795	17	18	18
24	BR733	GIGA	SIX	7496.523	797	20	15	16
25	BR733	SIX	GIGA	6748.733	804	17	16	11
26	SIX	GIGA	BR733	11085.690	802	15	14	17
27	SIX	BR733	GIGA	11888.765	803	14	9	22

6. 5 Concurrent Users, Role 3 (Minimal Memory)

Table 109: Concurrent User 1 of 5 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	29308.689	742	136	405	18	11
2	GIGA	GIGA	BR733	21650.915	698	149	442	10	13
3	GIGA	BR733	GIGA	20058.737	720	144	428	11	9
4	GIGA	BR733	BR733	36009.979	727	120	427	20	18
5	BR733	GIGA	GIGA	25375.311	706	163	417	18	8
6	BR733	GIGA	BR733	25317.239	707	142	431	18	14
7	BR733	BR733	GIGA	20468.441	724	138	421	15	14
22	GIGA	BR733	SIX	26907.824	702	135	457	5	13
23	GIGA	SIX	BR733	23624.572	732	141	411	13	15
24	BR733	GIGA	SIX	25818.347	710	150	423	16	13
25	BR733	SIX	GIGA	20473.741	699	163	426	13	11
26	SIX	GIGA	BR733	21031.809	698	133	443	23	15
27	SIX	BR733	GIGA	18392.066	696	168	424	8	16

Table 110: Concurrent User 2 of 5 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	29383.957	688	163	442	12	7
2	GIGA	GIGA	BR733	21509.335	717	148	411	17	19
3	GIGA	BR733	GIGA	20232.077	732	151	399	11	19
4	GIGA	BR733	BR733	36077.438	706	142	433	15	16
5	BR733	GIGA	GIGA	25328.328	689	146	455	10	12
6	BR733	GIGA	BR733	25321.143	749	128	407	10	18
7	BR733	BR733	GIGA	20582.842	709	152	428	9	14
22	GIGA	BR733	SIX	26619.569	697	149	439	12	15
23	GIGA	SIX	BR733	24290.646	717	142	421	14	18
24	BR733	GIGA	SIX	25779.678	703	151	423	17	18
25	BR733	SIX	GIGA	19838.665	718	143	429	13	9
26	SIX	GIGA	BR733	21340.261	738	131	421	8	14
27	SIX	BR733	GIGA	18275.142	735	142	406	8	21

Table 111: Concurrent User 3 of 5 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 3	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	29300.797	700	151	431	13	17

2	GIGA	GIGA	BR733	21427.519	703	161	415	14	19
3	GIGA	BR733	GIGA	20160.473	698	175	413	10	16
4	GIGA	BR733	BR733	35973.335	723	133	435	12	9
5	BR733	GIGA	GIGA	25325.983	715	154	420	13	10
6	BR733	GIGA	BR733	25006.708	739	133	415	11	14
7	BR733	BR733	GIGA	20007.344	705	125	458	15	9
22	GIGA	BR733	SIX	26379.148	730	138	420	14	10
23	GIGA	SIX	BR733	23738.821	718	144	423	15	12
24	BR733	GIGA	SIX	25838.128	737	121	437	5	12
25	BR733	SIX	GIGA	20319.854	719	144	421	11	17
26	SIX	GIGA	BR733	21099.633	715	156	408	18	15
27	SIX	BR733	GIGA	17599.516	712	137	433	15	15

Table 112: Concurrent User 4 of 5 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 4	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	29129.191	728	136	429	10	9
2	GIGA	GIGA	BR733	21459.883	723	128	430	12	19
3	GIGA	BR733	GIGA	20117.487	718	152	415	14	13
4	GIGA	BR733	BR733	36059.181	728	148	402	17	17
5	BR733	GIGA	GIGA	25284.713	732	129	423	13	15
6	BR733	GIGA	BR733	25225.636	685	143	461	14	9
7	BR733	BR733	GIGA	19939.753	692	133	457	15	15
22	GIGA	BR733	SIX	26437.827	696	154	431	21	10
23	GIGA	SIX	BR733	23859.267	706	130	455	12	9
24	BR733	GIGA	SIX	25985.595	707	140	436	15	14
25	BR733	SIX	GIGA	20123.618	726	140	424	10	12
26	SIX	GIGA	BR733	20804.034	713	136	426	17	20
27	SIX	BR733	GIGA	17855.588	725	150	411	14	12

Table 113: Concurrent User 5 of 5 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 5	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	29341.577	702	159	422	15	14
2	GIGA	GIGA	BR733	21495.246	708	148	428	16	12
3	GIGA	BR733	GIGA	19866.018	710	142	432	13	15
4	GIGA	BR733	BR733	35953.482	723	139	422	15	13
5	BR733	GIGA	GIGA	25135.089	710	128	448	13	13
6	BR733	GIGA	BR733	25054.755	745	124	409	16	18

7	BR733	BR733	GIGA	20469.808	697	151	430	14	20
22	GIGA	BR733	SIX	26744.303	716	143	436	10	7
23	GIGA	SIX	BR733	23561.448	715	133	433	17	14
24	BR733	GIGA	SIX	25847.335	725	145	416	11	15
25	BR733	SIX	GIGA	20079.761	686	161	439	13	13
26	SIX	GIGA	BR733	20858.731	738	121	428	13	12
27	SIX	BR733	GIGA	17955.349	707	153	429	15	8

7. 2 Concurrent Users, Role 3 (Minimal Memory)

Table 114: Concurrent User 1 of 2 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	10851.931	730	144	410	14	14
2	GIGA	GIGA	BR733	10887.719	720	152	411	16	13
3	GIGA	BR733	GIGA	10207.401	705	150	426	16	15
4	GIGA	BR733	BR733	13995.321	713	125	455	11	8
5	BR733	GIGA	GIGA	10352.231	720	138	433	11	10
6	BR733	GIGA	BR733	11684.966	723	144	421	13	11
7	BR733	BR733	GIGA	10585.257	705	144	434	19	10
22	GIGA	BR733	SIX	13255.416	703	147	426	17	19
23	GIGA	SIX	BR733	12349.443	733	113	434	13	19
24	BR733	GIGA	SIX	12363.961	706	129	442	16	19
25	BR733	SIX	GIGA	11193.193	713	146	414	18	21
26	SIX	GIGA	BR733	10840.001	723	146	412	14	17
27	SIX	BR733	GIGA	10259.503	715	140	429	20	8

Table 115: Concurrent User 2 of 2 for Role 3 (Minimal Memory)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	10510.374	725	107	454	13	13
2	GIGA	GIGA	BR733	10604.995	708	126	446	16	16
3	GIGA	BR733	GIGA	9950.221	727	127	432	15	11
4	GIGA	BR733	BR733	13979.816	709	152	418	17	16
5	BR733	GIGA	GIGA	10158.191	703	140	456	10	3
6	BR733	GIGA	BR733	11434.673	714	134	434	18	12
7	BR733	BR733	GIGA	10524.646	744	134	407	12	15
22	GIGA	BR733	SIX	13140.455	724	145	415	14	14
23	GIGA	SIX	BR733	12312.748	719	143	423	13	14
24	BR733	GIGA	SIX	12371.365	730	149	410	12	11

25	BR733	SIX	GIGA	11035.936	715	146	422	15	14
26	SIX	GIGA	BR733	10608.779	692	142	458	8	12
27	SIX	BR733	GIGA	10284.958	726	153	407	12	14

8. CORBA TEST, 3 Concurrent Users, Role 3 (Minimal Memory)

Table 116: Concurrent User 1 of 3 for Role 3 (Minimal Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15902.606	698	163	419	17	15
2	GIGA	GIGA	BR733	13851.467	727	132	419	18	16
3	GIGA	BR733	GIGA	12640.207	710	146	425	16	15
4	GIGA	BR733	BR733	19787.586	706	144	429	21	12
5	BR733	GIGA	GIGA	14254.681	724	136	427	12	13
6	BR733	GIGA	BR733	15169.018	707	143	431	16	15
7	BR733	BR733	GIGA	13284.574	722	138	425	13	14
8	BR733	BR733	BR733	22261.207	696	129	454	23	10
9	GIGA	GIGA	SIX	16113.221	750	131	409	10	12
10	GIGA	SIX	GIGA	13975.419	707	141	435	11	18
11	GIGA	SIX	SIX	25093.607	720	131	416	20	25
12	SIX	GIGA	GIGA	14244.033	726	125	430	20	11
13	SIX	GIGA	SIX	18646.818	702	153	414	23	20
14	SIX	SIX	GIGA	14947.481	676	149	466	13	8
15	SIX	SIX	SIX	29006.722	723	134	425	13	15
16	BR733	BR733	SIX	17717.606	745	131	409	8	19
17	BR733	SIX	BR733	17460.879	682	148	449	11	22
18	BR733	SIX	SIX	25029.654	708	132	445	10	17
19	SIX	BR733	BR733	20418.397	687	158	436	9	22
20	SIX	BR733	SIX	19192.782	715	144	429	6	11
21	SIX	SIX	BR733	17311.546	714	126	437	18	17
22	GIGA	BR733	SIX	16767.127	704	141	436	21	10
23	GIGA	SIX	BR733	15625.226	674	170	438	15	15
24	BR733	GIGA	SIX	15851.359	728	133	422	16	13
25	BR733	SIX	GIGA	13513.527	702	157	432	15	6
26	SIX	GIGA	BR733	13628.204	723	126	434	12	17
27	SIX	BR733	GIGA	11974.348	730	128	435	11	8

Table 117: Concurrent User 2 of 3 for Role 3 (Minimal Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15600.907	717	125	433	22	15
2	GIGA	GIGA	BR733	13995.792	717	144	419	16	16
3	GIGA	BR733	GIGA	12478.097	692	152	437	17	14
4	GIGA	BR733	BR733	20010.125	736	140	414	9	13
5	BR733	GIGA	GIGA	14019.258	701	138	451	10	12
6	BR733	GIGA	BR733	15381.816	714	153	414	12	19
7	BR733	BR733	GIGA	13096.697	703	136	446	14	13
8	BR733	BR733	BR733	22671.784	703	149	430	17	13
9	GIGA	GIGA	SIX	15873.356	715	132	433	20	12
10	GIGA	SIX	GIGA	14086.066	722	140	420	15	15
11	GIGA	SIX	SIX	25438.839	736	144	409	12	7
12	SIX	GIGA	GIGA	14385.535	708	139	431	15	19
13	SIX	GIGA	SIX	18833.822	716	141	431	9	11
14	SIX	SIX	GIGA	15532.644	680	149	439	14	19
15	SIX	SIX	SIX	29174.137	696	158	417	22	12
16	BR733	BR733	SIX	18342.729	738	151	398	8	17
17	BR733	SIX	BR733	17277.864	708	137	443	13	11
18	BR733	SIX	SIX	25116.797	718	138	432	10	14
19	SIX	BR733	BR733	20550.458	716	158	410	13	15
20	SIX	BR733	SIX	19586.859	708	138	443	10	13
21	SIX	SIX	BR733	17324.665	718	140	420	19	15
22	GIGA	BR733	SIX	16950.024	710	161	407	17	17
23	GIGA	SIX	BR733	15708.532	709	137	435	10	21
24	BR733	GIGA	SIX	15578.859	737	129	418	16	12
25	BR733	SIX	GIGA	13814.559	736	151	404	8	13
26	SIX	GIGA	BR733	13677.072	723	143	422	14	10
27	SIX	BR733	GIGA	12143.536	726	138	423	17	8

Table 118: Concurrent User 3 of 3 for Role 3 (Minimal Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 3	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	16129.592	727	164	399	13	9
2	GIGA	GIGA	BR733	13813.439	703	137	449	14	9
3	GIGA	BR733	GIGA	12845.333	750	142	395	9	16
4	GIGA	BR733	BR733	19886.711	716	145	414	21	16

5	BR733	GIGA	GIGA	14173.159	739	138	406	13	16
6	BR733	GIGA	BR733	15427.884	715	163	397	18	19
7	BR733	BR733	GIGA	13450.393	733	136	415	13	15
8	BR733	BR733	BR733	22847.348	707	155	428	10	12
9	GIGA	GIGA	SIX	16144.546	759	144	386	9	14
10	GIGA	SIX	GIGA	14058.289	676	156	459	8	13
11	GIGA	SIX	SIX	25415.994	721	149	408	16	15
12	SIX	GIGA	GIGA	14479.749	703	158	419	15	17
13	SIX	GIGA	SIX	18795.767	721	151	412	13	12
14	SIX	SIX	GIGA	15668.903	723	131	415	12	12
15	SIX	SIX	SIX	28438.137	702	139	442	14	15
16	BR733	BR733	SIX	17939.864	697	145	441	16	13
17	BR733	SIX	BR733	17436.521	724	148	417	15	8
18	BR733	SIX	SIX	25020.694	701	148	434	15	14
19	SIX	BR733	BR733	20031.563	694	139	457	15	7
20	SIX	BR733	SIX	19185.483	724	147	410	11	13
21	SIX	SIX	BR733	17833.644	715	136	422	20	19
22	GIGA	BR733	SIX	16844.057	722	132	430	15	13
23	GIGA	SIX	BR733	15528.486	702	147	427	14	12
24	BR733	GIGA	SIX	15767.777	738	125	418	10	21
25	BR733	SIX	GIGA	13577.607	716	153	412	12	19
26	SIX	GIGA	BR733	13653.412	721	143	422	14	12
27	SIX	BR733	GIGA	12111.191	716	149	412	20	15

9. CORBA TEST, 3 Concurrent Users, Role 3 (Maximum Memory)

Table 119: Concurrent User 1 of 3 for Role 3 (Maximum Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 1	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15835.711	717	142	427	11	15
2	GIGA	GIGA	BR733	13713.259	694	148	440	22	8
3	GIGA	BR733	GIGA	12532.954	755	130	405	10	12
4	GIGA	BR733	BR733	19848.918	730	138	413	17	14
5	BR733	GIGA	GIGA	14051.745	708	144	427	19	14
6	BR733	GIGA	BR733	15006.491	708	128	441	17	18
7	BR733	BR733	GIGA	12874.929	714	132	442	13	11
8	BR733	BR733	BR733	22926.198	689	161	437	14	11
9	GIGA	GIGA	SIX	16220.894	719	134	433	8	13
10	GIGA	SIX	GIGA	13725.859	695	158	438	11	10
11	GIGA	SIX	SIX	ERROR	error	error	error	error	error

12	SIX	GIGA	GIGA	14277.035	699	157	422	21	13
13	SIX	GIGA	SIX	ERROR	error	error	error	error	error
14	SIX	SIX	GIGA	ERROR	error	error	error	error	error
15	SIX	SIX	SIX	ERROR	error	error	error	error	error
16	BR733	BR733	SIX	17843.189	752	138	396	13	13
17	BR733	SIX	BR733	17037.538	687	148	440	17	20
18	BR733	SIX	SIX	ERROR	error	error	error	error	error
19	SIX	BR733	BR733	20044.136	714	141	426	14	17
20	SIX	BR733	SIX	ERROR	error	error	error	error	error
21	SIX	SIX	BR733	ERROR	error	error	error	error	error
22	GIGA	BR733	SIX	16739.674	716	136	433	11	16
23	GIGA	SIX	BR733	15387.672	728	139	420	12	13
24	BR733	GIGA	SIX	15821.133	683	146	449	22	12
25	BR733	SIX	GIGA	13333.902	718	129	433	17	15
26	SIX	GIGA	BR733	13511.467	711	163	407	20	11
27	SIX	BR733	GIGA	12141.578	688	148	449	13	14

Table 120: Concurrent User 2 of 3 for Role 3 (Maximum Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 2	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15728.861	722	137	423	14	16
2	GIGA	GIGA	BR733	13809.112	717	139	434	9	13
3	GIGA	BR733	GIGA	12394.802	690	140	453	13	16
4	GIGA	BR733	BR733	20011.192	732	124	429	11	16
5	BR733	GIGA	GIGA	14274.095	716	167	413	9	7
6	BR733	GIGA	BR733	14999.172	732	129	422	16	13
7	BR733	BR733	GIGA	12922.384	723	147	411	14	17
8	BR733	BR733	BR733	22638.914	669	158	452	16	17
9	GIGA	GIGA	SIX	16307.931	701	132	449	12	18
10	GIGA	SIX	GIGA	13904.291	693	142	454	15	8
11	GIGA	SIX	SIX	ERROR	error	error	error	error	error
12	SIX	GIGA	GIGA	14178.652	708	142	434	11	17
13	SIX	GIGA	SIX	ERROR	error	error	error	error	error
14	SIX	SIX	GIGA	ERROR	error	error	error	error	error
15	SIX	SIX	SIX	ERROR	error	error	error	error	error
16	BR733	BR733	SIX	17944.994	723	135	438	8	8
17	BR733	SIX	BR733	17075.675	728	129	430	12	13
18	BR733	SIX	SIX	ERROR	error	error	error	error	error
19	SIX	BR733	BR733	20078.383	708	129	438	16	21
20	SIX	BR733	SIX	ERROR	error	error	error	error	error

21	SIX	SIX	BR733	ERROR	error	error	error	error	error
22	GIGA	BR733	SIX	16930.397	735	148	403	12	14
23	GIGA	SIX	BR733	15420.976	692	153	436	17	14
24	BR733	GIGA	SIX	15854.579	742	132	411	13	14
25	BR733	SIX	GIGA	13195.000	682	148	450	13	19
26	SIX	GIGA	BR733	13422.793	726	149	414	10	13
27	SIX	BR733	GIGA	12108.128	731	137	415	15	14

Table 121: Concurrent User 3 of 3 for Role 3 (Maximum Memory, CORBA)

PATTERN	SERVER A	SERVER B	SERVER C	ROLE 3 - 3	CALL 1	CALL 2	CALL 3	CALL 4	CALL 5
1	GIGA	GIGA	GIGA	15745.851	724	122	443	10	13
2	GIGA	GIGA	BR733	13822.614	723	137	430	10	12
3	GIGA	BR733	GIGA	12674.874	731	144	408	14	15
4	GIGA	BR733	BR733	19953.330	690	157	442	7	16
5	BR733	GIGA	GIGA	14187.249	731	133	412	13	23
6	BR733	GIGA	BR733	15162.629	693	161	432	15	11
7	BR733	BR733	GIGA	12808.241	714	122	447	15	14
8	BR733	BR733	BR733	22866.417	765	144	375	16	12
9	GIGA	GIGA	SIX	16413.377	724	135	421	10	22
10	GIGA	SIX	GIGA	13931.158	710	139	423	22	18
11	GIGA	SIX	SIX	ERROR	error	error	error	error	error
12	SIX	GIGA	GIGA	14066.106	720	119	434	23	16
13	SIX	GIGA	SIX	ERROR	error	error	error	error	error
14	SIX	SIX	GIGA	ERROR	error	error	error	error	error
15	SIX	SIX	SIX	ERROR	error	error	error	error	error
16	BR733	BR733	SIX	17782.917	708	143	437	12	12
17	BR733	SIX	BR733	17354.072	714	166	409	15	8
18	BR733	SIX	SIX	ERROR	error	error	error	error	error
19	SIX	BR733	BR733	20166.174	714	154	417	11	16
20	SIX	BR733	SIX	ERROR	error	error	error	error	error
21	SIX	SIX	BR733	ERROR	error	error	error	error	error
22	GIGA	BR733	SIX	16770.806	718	148	418	12	16
23	GIGA	SIX	BR733	15460.586	747	144	396	12	13
24	BR733	GIGA	SIX	16012.197	716	143	434	9	10
25	BR733	SIX	GIGA	13328.134	727	136	421	13	15
26	SIX	GIGA	BR733	13497.521	692	155	437	16	12
27	SIX	BR733	GIGA	11785.137	662	129	484	16	21

APPENDIX D

A. COMBINATORIAL TIME TEST LINGO MODELS

1. Timing4_4

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    1000
    256    900
    128    777;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     111111.1     184
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
```

```
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.          ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

2. Timing5_5

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
```

```
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    1000
    256    900
     64    300
    128    777;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     656565.3     105
     111111.1     184
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                          ;
!  Figure out if two servers are running on the same machine.   ;
!                                                          ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                          ;
!  A server cannot be split over multiple machines         ;
!                                                          ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                          ;
!  Each server can only run on one machine.                ;
```

```
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

3. Timing6_6

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000
    256    900
     64    300
    128    777;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
     656565.3     105
```

```
    111111.1     184
    323232.3     100;
  MEM_LIMIT = 1.0;
  NET_BW = 100000000;
  CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                          ;
!  Figure out if two servers are running on the same machine.   ;
!                                                          ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                          ;
!  A server cannot be split over multiple machines         ;
!                                                          ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                          ;
!  Each server can only run on one machine.                ;
!                                                          ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                           ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                           ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                           ;
!  Constraint for limiting the CPU load on a single machine.     ;
!                                                           ;
 @FOR (MACHINE(R):
```

```
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

4. Timing7_7

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6 W7/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6 S7/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000
     64    866
    256    900
     64    300
    128    777;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
     111111.1      11
     656565.3     105
     111111.1     184
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
```

```
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                          ;
!  Figure out if two servers are running on the same machine.   ;
!                                                          ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
     );
 );
!                                                          ;
!  A server cannot be split over multiple machines         ;
!                                                          ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                          ;
!  Each server can only run on one machine.                ;
!                                                          ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                           ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                           ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                           ;
!  Constraint for limiting the CPU load on a single machine.     ;
!                                                           ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

5. Timing8_8

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6 W7 W8/:
```

```
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6 S7 S8/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000
     64    866
    256    900
    128    550
     64    300
    128    777;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
     111111.1      11
     232323.4      23
     656565.3     105
     111111.1     184
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                          ;
!  Figure out if two servers are running on the same machine.   ;
```

```
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
        @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
        T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
        @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
        Q(R) < CPU_TIME;
   );

END
```

6. Timing9_9

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6 W7 W8 W9/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6 S7 S8 S9/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
```

```
   MEMORY SPEED =
     64    600
    128    733
    128    1000
     64    866
    256    900
    128    550
    100    666
     64    300
    185    444;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
     111111.1      11
     343434.4      55
     656565.3     105
     111111.1     184
     255555.5     130
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
```

```
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                                     ;
!  Constraint for limiting the RAM load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                     ;
!  Constraint for limiting the CPU load on a single machine.         ;
!                                                                     ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

7. Timing10_10

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6 W7 W8 W9 W10/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6 S7 S8 S9 S10/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000
     64    866
    256    900
    128    550
    100    666
     64    300
```

```
    128    777
    185    444;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
     111111.1      11
     232323.4      23
     343434.4      55
     656565.3     105
     111111.1     184
     255555.5     130
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                                    ;
!  Figure out if two servers are running on the same machine.        ;
!                                                                    ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                                    ;
!  A server cannot be split over multiple machines                   ;
!                                                                    ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                                    ;
!  Each server can only run on one machine.                          ;
!                                                                    ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
```

```
!                                                                    ;
!  Constraint for limiting the RAM load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                    ;
!  Constraint for limiting the CPU load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

8. Timing11_11

```
MODEL:

SETS:
   MACHINE / W1 W2 W3 W4 W5 W6 W7 W8 W9 W10 W11/:
      MEMORY, SPEED;
   SERVER  / S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11/:
 MULTIPLIER, MEMORYUSE;
   DEPLOYMENT (MACHINE, SERVER): V;
   NET_SPD (SERVER, SERVER): U;
   MEM_USED (MACHINE): T;
   CPU_USED (MACHINE): Q;
ENDSETS

DATA:
   NORM_SPEED = 1000;
   MEMORY SPEED =
     64    600
    128    733
    128    1000
     64    866
    256    900
    128    550
    100    666
     88    1111
     32    1111
    128    777
    185    444;
   MULTIPLIER MEMORYUSE =
     442432.0      44
     807518.4      60
     288412.8      66
```

```
     111111.1      11
     232323.4      23
     565656.5      88
     777777.3      87
     656565.3     105
     111111.1     184
     255555.5     130
     323232.3     100;
   MEM_LIMIT = 1.0;
   NET_BW = 100000000;
   CPU_TIME = 10000000000;
ENDDATA

 MIN = PROC_SPEED + NET_SPEED;

 PROC_SPEED = @SUM( DEPLOYMENT( I, J ):
              V ( I, J ) * MULTIPLIER ( J ) * NORM_SPEED / SPEED( I ));
!
;
!  Inter-Server communications function.  Ignore Client/Server Comms
;
!  because they always exist and we are letting the Client location
;
!  be the free variable.   NOTE:  ASSUME LOCAL TWICE AS FAST AS REMOTE
;
!
;
 NET_SPEED = 30720000000/(U(@INDEX(S2),@INDEX(S3))*NET_BW) ;
!                                                          ;
!  Figure out if two servers are running on the same machine.   ;
!                                                          ;
 @FOR (SERVER(K):
    @FOR (SERVER(L):
        @SUM ( MACHINE(R): V(R,K)*V(R,L)) + 1 = U(K,L);
    );
 );
!                                                          ;
!  A server cannot be split over multiple machines         ;
!                                                          ;
 @FOR (DEPLOYMENT: @BIN(V));
!                                                          ;
!  Each server can only run on one machine.                ;
!                                                          ;
 @FOR (SERVER(K):
         @SUM ( MACHINE(R): V(R, K)) = 1;
      );
!                                                           ;
!  Constraint for limiting the RAM load on a single machine.     ;
!                                                           ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MEMORYUSE(K)) = T(R);
```

```
         T(R) < MEMORY(R)*MEM_LIMIT;
   );
!                                                                    ;
!  Constraint for limiting the CPU load on a single machine.        ;
!                                                                    ;
 @FOR (MACHINE(R):
         @SUM ( SERVER(K): V(R, K)*MULTIPLIER(K)*NORM_SPEED/SPEED(R)) =
Q(R);
         Q(R) < CPU_TIME;
   );

END
```

APPENDIX E

A. JAVA CORBA CODE

This code was used in the testbed to validate the model using servers implemented with CORBA middleware.

1. Server A Side Code

a) A.idl

```
// A.idl

module A {
  interface Account {
    float balance();
 string m1();
 string m2();
 string m3();
 string m4();
  };
  interface AccountManager {
    Account open(in string index);
  };
};
```

b) AccountImpl.java

```
// AccountImpl.java

public class AccountImpl extends A.AccountPOA {
  public AccountImpl(float balance) {
    _balance = balance;
 _a1 = "Return for m1: This is the first string of this new non-
demonstruct";
 _a2 = "Return for m2";
 _a3 = "Return for m3";
 _a4 = "Return for m4: Its quitting time for me.";
 _frits = new StringBuffer(1000000);
  }

  private static int factorial(int x) {
    if (x <= 0)
    return 0;
 else
```

```
    return factorial(x-1) + x;
  }

  public float balance() {
    return _balance;
  }
  public String m1() {
    int count;

 for (int i = 0; i < 360; i++)
   for (int j = 0; j < 360; j++)
     count = factorial(i) * factorial(j);

    return _a1;
  }
  public String m2() {
    int count;

    for (int i = 0; i < 600; i++)
      for (int j = 0; j < 600; j++)
         count = factorial(i) * factorial(j);

    return _a2;
  }
  public String m3() {
    int count;

    for (int i = 0; i < 460; i++)
      for (int j = 0; j < 460; j++)
         count = factorial(i) * factorial(j);

    return _a3;
  }
  public String m4() {
    int count;

    for (int i = 0; i < 550; i++)
      for (int j = 0; j < 550; j++)
         count = factorial(i) * factorial(j);

    return _a4;
  }
  private float _balance;
  private String _a1;
  private String _a2;
  private String _a3;
  private String _a4;
  private StringBuffer _frits;
}
```

c) AccountManagerImpl.java

```
// AccountManagerImpl.java
import org.omg.PortableServer.*;

import java.util.*;

public class AccountManagerImpl extends A.AccountManagerPOA {
  public synchronized A.Account open(String index) {
    // Lookup the account in the account dictionary.
    A.Account account = (A.Account) _accounts.get(index);
    // If there was no account in the dictionary, create one.
    if(account == null) {
      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Create the account implementation, given the balance.
      AccountImpl accountServant = new AccountImpl(balance);
      try {
        // Activate it on the default POA which is root POA for this
servant
        account =
A.AccountHelper.narrow(_default_POA().servant_to_reference(accountServa
nt));
      } catch (Exception e) {
        e.printStackTrace();
      }
      // Print out the new account.
      System.out.println("Created " + index + "'s account: " +
account);
      // Save the account in the account dictionary.
      _accounts.put(index, account);
    }
    // Return the account.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

d) Server.java

```
// Server.java
import org.omg.PortableServer.*;

public class Server {

  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
```

```
      // get a reference to the root POA
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      // Create policies for our persistent POA
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };
      // Create myPOA with the right policies
      POA myPOA = rootPOA.create_POA( "a_poa",
rootPOA.the_POAManager(),
                                      policies );
      // Create the servant
      AccountManagerImpl managerServant = new AccountManagerImpl();
      // Decide on the ID for the servant
      byte[] managerId = "AManager".getBytes();
      // Activate the servant with the ID on myPOA
      myPOA.activate_object_with_id(managerId, managerServant);

      // Activate the POA manager
      rootPOA.the_POAManager().activate();

      System.out.println(myPOA.servant_to_reference(managerServant) +
                         " is ready.");
      // Wait for incoming requests
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

}
```

2. Server B Side Code

a) B.idl

```
// B.idl

module B {
  interface Account {
    float balance();
 string m1();
 string m2();
  };
  interface AccountManager {
    Account open(in string index);
  };
};
```

b) AccountImpl.java

```
// AccountImpl.java
import java.util.Random;

public class AccountImpl extends B.AccountPOA {
  public AccountImpl(float balance) {
    Random simul = new Random();
    _balance = balance;
 _b1 = "Return for m1: This is the first string of this new non-
demonstruct";
 _b2 = "Return for m2";
 _frits = new StringBuffer(1000000);
    String[] args = null;
    org.omg.CORBA.ORB Corb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] CmanagerId = "CManager".getBytes();
    // Locate an account manager. Give the full POA name and the
servant ID.
    C.AccountManager Cmanager =
    C.AccountManagerHelper.bind(Corb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    _objC = Cmanager.open(Integer.toString(simul.nextInt(10)));
  }

  private static int factorial(int x) {
    if (x <= 0)
    return 0;
 else
    return factorial(x-1) + x;
  }

  public float balance() {
    return _balance;
  }
  public String m1() {
    int count;

 for (int i = 0; i < 511; i++)
   for (int j = 0; j < 511; j++)
     count = factorial(i) * factorial(j);

    return _b1;
  }
  public String m2() {
    int count;

    for (int i = 0; i < 666; i++)
      for (int j = 0; j < 666; j++)
```

```
        count = factorial(i) * factorial(j);

    System.out.println(_objC.m1());
    return _b2;
  }
  private float _balance;
  private String _b1;
  private String _b2;
  private StringBuffer _frits;
  private C.Account _objC;
}
```

c) AccountManagerImpl.java

```
// AccountManagerImpl.java
import org.omg.PortableServer.*;

import java.util.*;

public class AccountManagerImpl extends B.AccountManagerPOA {
  public synchronized B.Account open(String index) {
    // Lookup the account in the account dictionary.
    B.Account account = (B.Account) _accounts.get(index);
    // If there was no account in the dictionary, create one.
    if(account == null) {
      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Create the account implementation, given the balance.
      AccountImpl accountServant = new AccountImpl(balance);
      try {
        // Activate it on the default POA which is root POA for this
servant
        account =
B.AccountHelper.narrow(_default_POA().servant_to_reference(accountServa
nt));
      } catch (Exception e) {
        e.printStackTrace();
      }
      // Print out the new account.
      System.out.println("Created " + index + "'s account: " +
account);
      // Save the account in the account dictionary.
      _accounts.put(index, account);
    }
    // Return the account.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

d) Server.java

```
// Server.java
import org.omg.PortableServer.*;

public class Server {

  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // get a reference to the root POA
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      // Create policies for our persistent POA
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };
      // Create myPOA with the right policies
      POA myPOA = rootPOA.create_POA( "b_poa",
rootPOA.the_POAManager(),
                                      policies );
      // Create the servant
      AccountManagerImpl managerServant = new AccountManagerImpl();
      // Decide on the ID for the servant
      byte[] managerId = "BManager".getBytes();
      // Activate the servant with the ID on myPOA
      myPOA.activate_object_with_id(managerId, managerServant);

      // Activate the POA manager
      rootPOA.the_POAManager().activate();

      System.out.println(myPOA.servant_to_reference(managerServant) +
                         " is ready.");
      // Wait for incoming requests
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
```

3. Server C Side Code

a) C.idl

```
// C.idl
```

```
module C {
  interface Account {
    float balance();
 string m1();
 string m2();
 string m3();
  };
  interface AccountManager {
    Account open(in string index);
  };
};
```

b) AccountImpl.java

```
// AccountImpl.java

public class AccountImpl extends C.AccountPOA {
  public AccountImpl(float balance) {
    _balance = balance;
 _c1 = "Return for m1: This is the first string of this new non-
demonstruct";
 _c2 = "Return for m2";
 _c3 = "Return for m3";
 _frits = new StringBuffer(1000000);
  }

  private static int factorial(int x) {
    if (x <= 0)
    return 0;
 else
    return factorial(x-1) + x;
  }

  public float balance() {
    return _balance;
  }
  public String m1() {
    int count;

 for (int i = 0; i < 627; i++)
   for (int j = 0; j < 627; j++)
     count = factorial(i) * factorial(j);

    return _c1;
  }
  public String m2() {
    int count;

    for (int i = 0; i < 726; i++)
```

```
      for (int j = 0; j < 726; j++)
         count = factorial(i) * factorial(j);

    return _c2;
  }
  public String m3() {
    int count;

    for (int i = 0; i < 340; i++)
      for (int j = 0; j < 340; j++)
         count = factorial(i) * factorial(j);

    return _c3;
  }
  private float _balance;
  private String _c1;
  private String _c2;
  private String _c3;
  private StringBuffer _frits;
}
```

c) AccountManagerImpl.java

```
// AccountManagerImpl.java
import org.omg.PortableServer.*;

import java.util.*;

public class AccountManagerImpl extends C.AccountManagerPOA {
  public synchronized C.Account open(String index) {
    // Lookup the account in the account dictionary.
    C.Account account = (C.Account) _accounts.get(index);
    // If there was no account in the dictionary, create one.
    if(account == null) {
      // Make up the account's balance, between 0 and 1000 dollars.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Create the account implementation, given the balance.
      AccountImpl accountServant = new AccountImpl(balance);
      try {
        // Activate it on the default POA which is root POA for this
servant
        account =
C.AccountHelper.narrow(_default_POA().servant_to_reference(accountServa
nt));
      } catch (Exception e) {
        e.printStackTrace();
      }
      // Print out the new account.
      System.out.println("Created " + index + "'s account: " +
account);
```

```
      // Save the account in the account dictionary.
      _accounts.put(index, account);
    }
    // Return the account.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

d) Server.java

```
// Server.java
import org.omg.PortableServer.*;

public class Server {

  public static void main(String[] args) {
    try {
      // Initialize the ORB.
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // get a reference to the root POA
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      // Create policies for our persistent POA
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };
      // Create myPOA with the right policies
      POA myPOA = rootPOA.create_POA( "c_poa",
rootPOA.the_POAManager(),
                                      policies );
      // Create the servant
      AccountManagerImpl managerServant = new AccountManagerImpl();
      // Decide on the ID for the servant
      byte[] managerId = "CManager".getBytes();
      // Activate the servant with the ID on myPOA
      myPOA.activate_object_with_id(managerId, managerServant);

      // Activate the POA manager
      rootPOA.the_POAManager().activate();

      System.out.println(myPOA.servant_to_reference(managerServant) +
                         " is ready.");
      // Wait for incoming requests
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
```

```
    }
  }

}
```

4. Client Side Code

a) Test.java

```
/*
 * @(#)Test.java 1.22 98/08/26
 *
 */

import java.util.*;

/**
 */
public class Test {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A.Account objA = null;
    B.Account objB = null;
    C.Account objC = null;

  public void init(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] AmanagerId = "AManager".getBytes();
    byte[] BmanagerId = "BManager".getBytes();
    byte[] CmanagerId = "CManager".getBytes();
    // Locate an account manager. Give the full POA name and the
servant ID.
    A.AccountManager Amanager =
    A.AccountManagerHelper.bind(orb, "/a_poa", AmanagerId);
    B.AccountManager Bmanager =
    B.AccountManagerHelper.bind(orb, "/b_poa", BmanagerId);
    C.AccountManager Cmanager =
    C.AccountManagerHelper.bind(orb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    int Acount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(10);
```

```
   for (int i = 1; i < 10; i++)
     objB = Bmanager.open(Integer.toString(i));
   objB = Bmanager.open(Integer.toString(Bcount));

   int Ccount = simulate.nextInt(10);
   for (int i = 1; i < 10; i++)
     objC = Cmanager.open(Integer.toString(i));
   objC = Cmanager.open(Integer.toString(Ccount));

 }

public void run_test1() {
   double average;
          boolean RUN1 = true;
   int count = 0;
   long duration = 0;

      RUN1 = true;
      count = 0;
      duration = 0;
      while (RUN1)
   {
      calltime.reset();
      try {
      objA.m1();
     }
      catch (Exception exc) {
       System.out.println("Test exception: " +
                   exc.getMessage());
    exc.printStackTrace();
    RUN1 = false;
    RUNNING = false;
   }
      calltime.stop();
      duration = duration + calltime.elapsedms();
      count = count + 1;
      if (count < 20)
   {
      try {
        Thread.currentThread().sleep(1000);
       }
      catch (Exception inter) {
         System.out.println("Test exception on sleep: " +
                     inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN1 = false;
   }
```

```
}
average = (double) duration / (double) count;
System.out.println("Average response time  1: " + average);

   RUN1 = true;
   count = 0;
   duration = 0;
   while (RUN1)
{
   calltime.reset();
   try {
            objA.m2();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  2: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objA.m3();
  }
```

```
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  3: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objA.m4();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
```

```
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  4: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objB.m1();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
```

```
average = (double) duration / (double) count;
System.out.println("Average response time  5: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objB.m2();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                 inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  6: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objC.m1();
  }
   catch (Exception exc) {
```

```
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                 inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  7: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objC.m2();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
```

```
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
System.out.println("Average response time  8: " + average);

   count = 0;
   duration = 0;
   RUN1 = true;
   while (RUN1)
{
   calltime.reset();
   try {
            objC.m3();
  }
   catch (Exception exc) {
    System.out.println("Test exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < 20)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Test exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
average = (double) duration / (double) count;
```

```
    System.out.println("Average response time  9: " + average);

 }

    public static void main(String args[]) {

 Test test = new Test();
 calltime = new Timer();
 testtime = new Timer();
 simulate = new Random();


    test.init(args);
    test.run_test1();

   }
}
```

b) Roles.java

```
/*
 * @(#)Roles.java1.22 98/08/26
 *
 */

import java.util.Random;

/**
 */
public class Roles {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A.Account objA = null;
    B.Account objB = null;
    C.Account objC = null;
    A.AccountManager Amanager = null;
    B.AccountManager Bmanager = null;
    C.AccountManager Cmanager = null;

  public void init(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] AmanagerId = "AManager".getBytes();
    byte[] BmanagerId = "BManager".getBytes();
    byte[] CmanagerId = "CManager".getBytes();
```

```
    // Locate an account manager. Give the full POA name and the
servant ID.
    Amanager = A.AccountManagerHelper.bind(orb, "/a_poa", AmanagerId);
    Bmanager = B.AccountManagerHelper.bind(orb, "/b_poa", BmanagerId);
    Cmanager = C.AccountManagerHelper.bind(orb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    int Acount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }

  public void set_memory() {
    int Acount = simulate.nextInt(44);
    for (int i = 1; i < 44; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(60);
    for (int i = 1; i < 60; i++)
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(66);
    for (int i = 1; i < 66; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }

 public void run_test1(int max_run) {
    int choice = 1;
    double average;
           boolean RUN1 = true;
    int count = 0;
           int cnt1 = 0;
           int cnt2 = 0;
           int cnt3 = 0;
           int cnt4 = 0;
    long duration = 0;
```

```
testtime.reset();
   while (RUN1)
{
   choice = simulate.nextInt(53);
   calltime.reset();
try {
    if (choice < 50)
 {
            objA.m1();
                           cnt1 = cnt1 + 1;
 }
 else if (choice < 51)
 {
            objA.m2();
            objB.m1();
                           cnt2 = cnt2 + 1;
 }
 else if (choice < 52)
 {
            objC.m1();
            objC.m2();
                           cnt3 = cnt3 + 1;
 }
 else if (choice < 53)
 {
            objB.m2();
                           cnt4 = cnt4 + 1;
 }
 else
 {
     System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("Roles exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
{
   try {
     Thread.currentThread().sleep(1000);
    }
   catch (Exception inter) {
      System.out.println("Roles exception on sleep: " +
```

```
                    inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN1 = false;
   }
   }
   testtime.stop();
   System.out.println("Test 1 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   System.out.println("  4   Number of calls is " + cnt4);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

public void run_test2(int max_run) {
   int choice = 1;
   double average;
          boolean RUN2 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN2)
   {
      choice = simulate.nextInt(74);
      calltime.reset();
   try {
       if (choice < 10)
    {
               objA.m1();
                              cnt1 = cnt1 + 1;
    }
    else if (choice < 50)
    {
               objA.m2();
               objB.m1();
                              cnt2 = cnt2 + 1;
    }
    else if (choice < 74)
    {
               objB.m1();
```

```
                objB.m2();
                              cnt3 = cnt3 + 1;
     }
     else
     {
          System.out.println("Got choice out of bounds " + choice);
     }
      }
    catch (Exception exc) {
        System.out.println("Roles exception: " +
                     exc.getMessage());
     exc.printStackTrace();
     RUN2 = false;
     RUNNING = false;
    }
    calltime.stop();
    duration = duration + calltime.elapsedms();
    count = count + 1;
    if (count < max_run)
    {
       try {
         Thread.currentThread().sleep(1000);
        }
       catch (Exception inter) {
          System.out.println("Roles exception on sleep: " +
                      inter.getMessage());
       inter.printStackTrace();
       }
    }
    else
    {
       RUN2 = false;
    }
    }
    testtime.stop();
    System.out.println("Test 2 duration is " + testtime.elapsed());
    System.out.println("Total number of calls is " + count);
    System.out.println("  1   Number of calls is " + cnt1);
    System.out.println("  2   Number of calls is " + cnt2);
    System.out.println("  3   Number of calls is " + cnt3);
    average = (double) duration / (double) count;
    System.out.println("Average response time is " + average);
    System.out.println("");
}

public void run_test3(int max_run) {
    int choice = 1;
    double average;
           boolean RUN3 = true;
    int count = 0;
           int cnt1 = 0;
```

```
       int cnt2 = 0;
       int cnt3 = 0;
       int cnt4 = 0;
       int cnt5 = 0;
long duration = 0;

testtime.reset();
   while (RUN3)
{
   choice = simulate.nextInt(92);
   calltime.reset();
try {
    if (choice < 50)
 {
            objA.m1();
            objB.m2();
                           cnt1 = cnt1 + 1;
 }
 else if (choice < 60)
 {
            objA.m1();
            objA.m2();
            objA.m3();
            objB.m2();
                           cnt2 = cnt2 + 1;
 }
 else if (choice < 90)
 {
            objC.m2();
                           cnt3 = cnt3 + 1;
 }
 else if (choice < 91)
 {
            objC.m3();
                           cnt4 = cnt4 + 1;
 }
 else if (choice < 92)
 {
            objB.m1();
            objB.m2();
                           cnt5 = cnt5 + 1;
 }
 else
 {
     System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("Roles exception: " +
                exc.getMessage());
 exc.printStackTrace();
```

```
    RUN3 = false;
    RUNNING = false;
   }
   calltime.stop();
   duration = duration + calltime.elapsedms();
   count = count + 1;
   if (count < max_run)
   {
      try {
        Thread.currentThread().sleep(1000);
       }
      catch (Exception inter) {
         System.out.println("Roles exception on sleep: " +
                     inter.getMessage());
      inter.printStackTrace();
      }
   }
   else
   {
      RUN3 = false;
   }
   }
   testtime.stop();
   System.out.println("Test 3 duration is " + testtime.elapsed());
   System.out.println("Total number of calls is " + count);
   System.out.println("  1   Number of calls is " + cnt1);
   System.out.println("  2   Number of calls is " + cnt2);
   System.out.println("  3   Number of calls is " + cnt3);
   System.out.println("  4   Number of calls is " + cnt4);
   System.out.println("  5   Number of calls is " + cnt5);
   average = (double) duration / (double) count;
   System.out.println("Average response time is " + average);
   System.out.println("");
}

   public static void main(String args[]) {
/*
*/

Roles test = new Roles();
calltime = new Timer();
testtime = new Timer();
simulate = new Random();


   test.init(args);
   test.run_test1(1000);
   test.run_test2(1000);
   test.run_test3(1000);
   test.set_memory();
   System.out.println("");
```

```
    System.out.println("Bumping the memory on the servers...");
    System.out.println("");
    test.run_test1(1000);
    test.run_test2(1000);
    test.run_test3(1000);

   }
}
```

c) R1.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class R1 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A.Account objA = null;
    B.Account objB = null;
    C.Account objC = null;
 A.AccountManager Amanager = null;
 B.AccountManager Bmanager = null;
 C.AccountManager Cmanager = null;

  public void init(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] AmanagerId = "AManager".getBytes();
    byte[] BmanagerId = "BManager".getBytes();
    byte[] CmanagerId = "CManager".getBytes();
    // Locate an account manager. Give the full POA name and the
servant ID.
    Amanager = A.AccountManagerHelper.bind(orb, "/a_poa", AmanagerId);
    Bmanager = B.AccountManagerHelper.bind(orb, "/b_poa", BmanagerId);
    Cmanager = C.AccountManagerHelper.bind(orb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    int Acount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
```

```
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }

  public void set_memory() {
    int Acount = simulate.nextInt(44);
    for (int i = 1; i < 44; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(60);
    for (int i = 1; i < 60; i++)
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(66);
    for (int i = 1; i < 66; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }

 public void run_test1(int max_run) {
    int choice = 1;
    double average;
           boolean RUN1 = true;
    int count = 0;
           int cnt1 = 0;
           int cnt2 = 0;
           int cnt3 = 0;
           int cnt4 = 0;
    long duration = 0;

    testtime.reset();
       while (RUN1)
    {
       choice = simulate.nextInt(53);
       calltime.reset();
    try {
        if (choice < 50)
     {
                objA.m1();
                                cnt1 = cnt1 + 1;
     }
     else if (choice < 51)
```

```
 {
            objA.m2();
            objB.m1();
                           cnt2 = cnt2 + 1;
 }
 else if (choice < 52)
 {
            objC.m1();
            objC.m2();
                           cnt3 = cnt3 + 1;
 }
 else if (choice < 53)
 {
            objB.m2();
                           cnt4 = cnt4 + 1;
 }
 else
 {
      System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("R1 exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN1 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
{
   try {
     Thread.currentThread().sleep(15500);
    }
   catch (Exception inter) {
      System.out.println("R1 exception on sleep: " +
                  inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN1 = false;
}
}
testtime.stop();
System.out.println("Test 1 duration is " + testtime.elapsed());
System.out.println("Total number of calls is " + count);
System.out.println("  1   Number of calls is " + cnt1);
```

```
    System.out.println("  2   Number of calls is " + cnt2);
    System.out.println("  3   Number of calls is " + cnt3);
    System.out.println("  4   Number of calls is " + cnt4);
    average = (double) duration / (double) count;
    System.out.println("Average response time is " + average);
    System.out.println("");
 }

    public static void main(String args[]) {
 /*
 */

 R1 test = new R1();
 calltime = new Timer();
 testtime = new Timer();
 simulate = new Random();


    test.init(args);
    test.run_test1(848);
    test.set_memory();
    System.out.println("");
    System.out.println("Bumping the memory on the servers...");
    System.out.println("");
    test.run_test1(848);

   }
}
```

d) R2.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;

/**
 */
public class R2 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A.Account objA = null;
    B.Account objB = null;
    C.Account objC = null;
    A.AccountManager Amanager = null;
    B.AccountManager Bmanager = null;
    C.AccountManager Cmanager = null;
```

```
  public void init(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] AmanagerId = "AManager".getBytes();
    byte[] BmanagerId = "BManager".getBytes();
    byte[] CmanagerId = "CManager".getBytes();
    // Locate an account manager. Give the full POA name and the
servant ID.
    Amanager = A.AccountManagerHelper.bind(orb, "/a_poa", AmanagerId);
    Bmanager = B.AccountManagerHelper.bind(orb, "/b_poa", BmanagerId);
    Cmanager = C.AccountManagerHelper.bind(orb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    int Acount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }

  public void set_memory() {
    int Acount = simulate.nextInt(44);
    for (int i = 1; i < 44; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

    int Bcount = simulate.nextInt(60);
    for (int i = 1; i < 60; i++)
      objB = Bmanager.open(Integer.toString(i));
    objB = Bmanager.open(Integer.toString(Bcount));

    int Ccount = simulate.nextInt(66);
    for (int i = 1; i < 66; i++)
      objC = Cmanager.open(Integer.toString(i));
    objC = Cmanager.open(Integer.toString(Ccount));

  }


 public void run_test2(int max_run) {
```

```
int choice = 1;
double average;
       boolean RUN2 = true;
int count = 0;
       int cnt1 = 0;
       int cnt2 = 0;
       int cnt3 = 0;
long duration = 0;

testtime.reset();
   while (RUN2)
{
   choice = simulate.nextInt(74);
   calltime.reset();
try {
    if (choice < 10)
 {
            objA.m1();
                           cnt1 = cnt1 + 1;
 }
 else if (choice < 50)
 {
            objA.m2();
            objB.m1();
                           cnt2 = cnt2 + 1;
 }
 else if (choice < 74)
 {
            objB.m1();
            objB.m2();
                           cnt3 = cnt3 + 1;
 }
 else
 {
     System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("R2 exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN2 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
{
   try {
     Thread.currentThread().sleep(6500);
```

```
        }
       catch (Exception inter) {
          System.out.println("R2 exception on sleep: " +
                      inter.getMessage());
       inter.printStackTrace();
       }
    }
    else
    {
       RUN2 = false;
    }
    }
    testtime.stop();
    System.out.println("Test 2 duration is " + testtime.elapsed());
    System.out.println("Total number of calls is " + count);
    System.out.println("  1   Number of calls is " + cnt1);
    System.out.println("  2   Number of calls is " + cnt2);
    System.out.println("  3   Number of calls is " + cnt3);
    average = (double) duration / (double) count;
    System.out.println("Average response time is " + average);
    System.out.println("");
 }

    public static void main(String args[]) {
 /*
 */

 R2 test = new R2();
 calltime = new Timer();
 testtime = new Timer();
 simulate = new Random();


    test.init(args);
    test.run_test2(1184);
    test.set_memory();
    System.out.println("");
    System.out.println("Bumping the memory on the servers...");
    System.out.println("");
    test.run_test2(1184);

   }
}
```

e) R3.java

```
import java.util.Random;
import java.rmi.Naming;
import java.rmi.RemoteException;
```

```
/**
 */
public class R3 {
    static Timer calltime;
    static Timer testtime;
 static Random simulate;

 boolean RUNNING = true;
    A.Account objA = null;
    B.Account objB = null;
    C.Account objC = null;
    A.AccountManager Amanager = null;
    B.AccountManager Bmanager = null;
    C.AccountManager Cmanager = null;

  public void init(String[] args) {
    // Initialize the ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Get the manager Id
    byte[] AmanagerId = "AManager".getBytes();
    byte[] BmanagerId = "BManager".getBytes();
    byte[] CmanagerId = "CManager".getBytes();
    // Locate an account manager. Give the full POA name and the
servant ID.
    Amanager = A.AccountManagerHelper.bind(orb, "/a_poa", AmanagerId);
    Bmanager = B.AccountManagerHelper.bind(orb, "/b_poa", BmanagerId);
    Cmanager = C.AccountManagerHelper.bind(orb, "/c_poa", CmanagerId);
    // Request the account manager to open a named account.
    int Acount = simulate.nextInt(10);
    for (int i = 1; i < 10; i++)
      objA = Amanager.open(Integer.toString(i));
    objA = Amanager.open(Integer.toString(Acount));

   int Bcount = simulate.nextInt(10);
   for (int i = 1; i < 10; i++)
     objB = Bmanager.open(Integer.toString(i));
   objB = Bmanager.open(Integer.toString(Bcount));

   int Ccount = simulate.nextInt(10);
   for (int i = 1; i < 10; i++)
     objC = Cmanager.open(Integer.toString(i));
   objC = Cmanager.open(Integer.toString(Ccount));

 }

 public void set_memory() {
   int Acount = simulate.nextInt(44);
   for (int i = 1; i < 44; i++)
     objA = Amanager.open(Integer.toString(i));
   objA = Amanager.open(Integer.toString(Acount));
```

```
  int Bcount = simulate.nextInt(60);
  for (int i = 1; i < 60; i++)
    objB = Bmanager.open(Integer.toString(i));
  objB = Bmanager.open(Integer.toString(Bcount));

   int Ccount = simulate.nextInt(66);
   for (int i = 1; i < 66; i++)
     objC = Cmanager.open(Integer.toString(i));
   objC = Cmanager.open(Integer.toString(Ccount));

 }


public void run_test3(int max_run) {
   int choice = 1;
   double average;
          boolean RUN3 = true;
   int count = 0;
          int cnt1 = 0;
          int cnt2 = 0;
          int cnt3 = 0;
          int cnt4 = 0;
          int cnt5 = 0;
   long duration = 0;

   testtime.reset();
      while (RUN3)
   {
      choice = simulate.nextInt(92);
      calltime.reset();
   try {
       if (choice < 50)
    {
               objA.m1();
               objB.m2();
                              cnt1 = cnt1 + 1;
    }
    else if (choice < 60)
    {
               objA.m1();
               objA.m2();
               objA.m3();
               objB.m2();
                              cnt2 = cnt2 + 1;
    }
    else if (choice < 90)
    {
               objC.m2();
                              cnt3 = cnt3 + 1;
    }
    else if (choice < 91)
```

```
 {
            objC.m3();
                          cnt4 = cnt4 + 1;
 }
 else if (choice < 92)
 {
            objB.m1();
            objB.m2();
                          cnt5 = cnt5 + 1;
 }
 else
 {
      System.out.println("Got choice out of bounds " + choice);
 }
  }
catch (Exception exc) {
    System.out.println("R3 exception: " +
                exc.getMessage());
 exc.printStackTrace();
 RUN3 = false;
 RUNNING = false;
}
calltime.stop();
duration = duration + calltime.elapsedms();
count = count + 1;
if (count < max_run)
{
   try {
     Thread.currentThread().sleep(4500);
    }
   catch (Exception inter) {
      System.out.println("R3 exception on sleep: " +
                 inter.getMessage());
   inter.printStackTrace();
   }
}
else
{
   RUN3 = false;
}
}
testtime.stop();
System.out.println("Test 3 duration is " + testtime.elapsed());
System.out.println("Total number of calls is " + count);
System.out.println("  1   Number of calls is " + cnt1);
System.out.println("  2   Number of calls is " + cnt2);
System.out.println("  3   Number of calls is " + cnt3);
System.out.println("  4   Number of calls is " + cnt4);
System.out.println("  5   Number of calls is " + cnt5);
average = (double) duration / (double) count;
System.out.println("Average response time is " + average);
```

```
   System.out.println("");
 }

   public static void main(String args[]) {
 /*
 */

 R3 test = new R3();
 calltime = new Timer();
 testtime = new Timer();
 simulate = new Random();


   test.init(args);
   test.run_test3(1312);
   test.set_memory();
   System.out.println("");
   System.out.println("Bumping the memory on the servers...");
   System.out.println("");
   test.run_test3(1312);

  }
}
```

BIBLIOGRAPHY

[1] H. Kameda and J. Li, “Load Balancing Problems for Multiclass Jobs in Distributed/Parallel Computer Systems,” *IEEE Transactions on Computers*, vol. 47, no. 3, pp. 322-332, March 1998.

[2] J. Watts and S. Taylor, “A Practical Approach to Dynamic Load Balancing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 3, pp. 235-248, March 1998.

[3] J. Kim, H. Lee and S. Lee, “Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers,” *IEEE Transactions on Computers*, vol. 46, no. 4, pp. 499-505, April 1997.

[4] P. Loh, W. Hsu, C. Wentong and N. Sriskanthan, “How Network Topology Affects Dynamic Load Balancing,” *IEEE Transactions on Parallel and Distributed Technology*, vol. 4, no. 3, pp. 25-35, Fall 1996.

[5] P. Mehra and B. Wah, “Synthetic Workload Generation for Load-Balancing Experiments,” *IEEE Transactions on Parallel and Distributed Technology*, vol. 3, no. 3, pp. 4-19, Fall 1995.

[6] P. *Berenbrink, T. Friedetzky, and A. Steger, Randomized and adversarial load balancing, Proceedings of the eleventh annual ACM symposium on Parallel algorithms and architectures, 1999, Pages 175 – 184*.

[7] M. Harchol-Balter, and A. Downey, *Exploiting process lifetime distributions for dynamic load balancing, ACM Transactions in Computing Systems 15, 3 (Aug. 1997), Pages 253 – 285*.

[8] A. Anastasiadi, S. Kapidakis, C. Nikolauo, and J. Sairamesh, *A computational economy for dynamic load balancing and data replication*, Proceedings of the First International Conference on Information and Computation Economies, 1998, Pages 166 - 180.

[10] D. Luckham and B. Frasca, “Complex Event Processing in Distributed Systems,” *Computer Systems Laboratory Technical Report CSL-TR-98-754.* Stanford University, Stanford, 1998.

[11] D. Luckham and J. Vera, “An Event-Based Architecture Definition Language,” *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995.

[12] L. Perrochon, W. Mann, S. Kasriel and D. Luckham, “Event Mining with Event Processing Networks,” *The Third Pacific-Asia Conference on Knowledge Discovery and Data Mining*. April 26-28, 1999. Beijing, China, 5 pages.

[13] N. Barghouti and B. Krishnamurthy, *Using event contexts and matching constraints to monitor software processes*, Proceedings of the 17th International Conference on Software Engineering, 1995, Page 83.

[14] R. Strom, G. Banavar, K. Miller, A. Prakash and M. Ward, “Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects,” IEEE Transactions on Computers. Vol 47, No. 4, April 1998.

[15] P. Cao and C. Liu, “Maintaining Strong Cache Consistency in the World Wide Web,” IEEE Transactions on Computers, Vol 47, No 4, April 1998.

[16] P. Triantafillou and D. Taylor, “The Location-Based Paradigm for Replication: Achieving Efficiency and Availability in Distributed Systems,” IEEE Transactions on Software Engineering, Vol 21, No 1, January 1995.

[17] D. Saha, S. Rangarajan and S. Tripathi, “An Analysis of the Average Message Overhead in Replica Control Protocols,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 10, October 1996.

[18] K. Hasegawa and M. Takizawa, “Object-Based Locking Protocol for Replicated Objects,” *Unpublished Paper,* Tokyo Denki University.

[19] M. Baentsch, L. Baum, G. Molter, S. Rothkugel and P. Sturm, “Enhancing the Web’s Infrastructure: From Caching to Replication,” *IEEE Internet Computing*, pp. 18-27, 1997.

[20] P. Khosla and H. Kiliccote, “PASIS”, College of Engineering/School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213.

[21] J. Lui, R. Muntz and D. Towsley, “Bounding the Mean Response Time of the Minimum Expected Delay Routing Policy: An Algorithmic Approach,” *IEEE Transactions on Computers.* Vol 44, No. 12, December 1995, pp. 1371-1382.

[22] M. Weiss, “Data Structures and Algorithm Analysis in JAVA,” Addison-Wesley, ISBN 0-201-35754-2, 1999.

[23] M. Colajanni, P. Yu and D. Dias, “Analysis of Task Assignment Policies in Scalable Distributed Web-Server Systems,” *IEEE Transactions on Parallel and Distributed Systems.* Vol 9, No. 6, June 1998, pp. 585-600.

[24] A. Bestavros, “WWW Traffic Reduction and Load Balancing through Server-Based Caching,” *IEEE Concurrency*, 1997, pp. 56-67.

[25] R. Adler, “Distributed Coordination Models for Client/Server Computing,” *IEEE*, April 1995, pp. 14-22.

[26] J. Munson and P. Dewan, “Sync: A Java Framework for Mobile Collaborative Applications,” *IEEE*, June 1997, pp. 59-66.

[27] L. Slothouber, “A Model of Web Server Performance,” *Unpublished Paper,* StarNine Technologies, Inc., 1996, 15 pages.

[28] V. Berzins and Luqi, “Software Engineering with Abstractions”, chapter 6, Addison-Wesley, ISBN 0-201-08004-4, 1991

[29] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” *International Journal on Supercomputer Applications*, 2001.

[30] I. Foster, A. Roy, V. Sander and L. Winkler, “End-to-End Quality of Service for High-End Applications,” *IEEE Journal on Selected Areas in Communications Special Issue on QoS in the Internet,* 1999.

[31] A. Hsiao and C. King, “The Thread-Based Protocol Engines for CC-NUMA Multiprocessors,” *International Conference on Parallel Processing 2000 Proceedings*, pp. 497-504.

[32] W. Ray and A. Farrar, “Object Model Driven Code Generation for the Enterprise,” *IEEE RSP 2001*, June 2001

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Dr. Luqi 2
   Naval Postgraduate School, Code CS
   luqi@cs.nps.navy.mil
   Monterey, CA 93943-5101

4. Dr. Valdis Berzins 1
   Naval Postgraduate School, Code CS
   berzins@cs.nps.navy.mil
   Monterey, CA 93943-5101

5. Dr. Geoffrey Xie 1
   Naval Postgraduate School, Code CS
   ggxie@cs.nps.navy.mil
   Monterey, CA 93943-5101

6. Dr. Dan Boger 1
   Naval Postgraduate School, Code CCBO
   dboger@nps.navy.mil
   Monterey, CA 93943-5101

7. Technical Library 1
   SPAWAR Systems Center
   library@spawar.navy.mil
   San Diego, CA 92152-5000