INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Performance Evaluation of a Database Server for a Distributed Application Monitoring System

By

Xiaodong Qin, M. Sc in ISS

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfillment of

The requirements for the degree of

Master of Science, Information and Systems Science

SCS

Carleton University

Ottawa, Ontario

December 1998

©Copyright

December 1998, Xiaodong Qin

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-36946-3

The undersigned recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

**Performance Evaluation of a Database Server for a Distributed
Application Monitoring System**

Submitted by
Xiaodong Qin
in partial fulfillment of the requirements for the degree of
Master of Science, Information and Systems Science

Chair, School of Computer Science

Thesis Supervisor

Carleton University
January 14, 1999

Abstract

The purpose of the research is to develop and evaluate the performance behavior of a database server for a distributed application monitoring system. A multithreaded database daemon is developed for an Application Response Measurement (ARM)-based performance monitoring system. The daemon accepts performance data from monitoring agents and writes the data to a performance database management system. Various database technologies and distributed application monitoring systems are discussed. The performance evaluation determines the capacity of the developed system in terms of how many monitoring agents and application processes can be supported.

Acknowledgements

This thesis is the result of many people's working efforts. First of all, I would like to thank my supervisor, Professor Jerome Rolia, for providing me such a great opportunity to make contributions to the cutting-edge ARM-based performance management system developed at Carleton University. He gave me the valuable research trends and technical advices with endless kindness and patience. He has always been there whenever I have questions and problems. Without his exceptional leadership in the research supervision, this thesis would never have such great results. The thanks also go to my colleague, Ferass ElRayes, with whom I have been working very closely during the whole research period. Without his help and other important components of the system he developed, the performance measurement would never take place. I would also like to thank Xin Sun and Diwakar Krishnamurthy, who gave me a lot of knowledge and information in performance evaluation theories.

I also want to mention that the most important person in my life, my husband, always gave me unconditional support during the whole research. I would have never finished the thesis without his encouragement and patience.

Table of Contents

Abstract.....	ii
Acknowledgements	iii
Table of Contents	iv
List of Tables	viii
List of Figures.....	ix
Chapter 1	1
Introduction.....	1
1.1 Introduction to Distributed Application Monitoring Systems	1
1.2 Introduction to Distributed Monitoring Using the ARM API.....	4
1.3 Data Storage and Transfer Problem in Distributed Application Monitoring Systems	7
1.4 Conventional Approaches to Transferring and Storing Performance Data.....	8
1.5 Contribution of the Thesis.....	11
1.6 Thesis Outline	12
Chapter 2	13
Distributed Application Performance Monitoring System Architectures.....	13
2.1 Introduction to Distributed Application Performance Monitoring	13
2.2 Distributed Application Monitoring Systems	16
2.2.1 Management of Distributed Applications and Systems (MANDAS).....	16
2.2.2 Distributed Measurement System (DMS).....	19
2.3 ARM-based Distributed Performance Monitoring System.....	21
2.3.1 Application Response Measurement (ARM) API	22

2.3.2 ARM-based Distributed Application Monitoring System Architecture for Carleton University ARM 2.0 Prototype	23
2.3.2.1 Instrumented application	25
2.3.2.2 ARM Agent	25
2.3.2.3 Performance Data Storage	25
2.3.2.4 Management Application	25
2.3.3 Steps of Monitoring Distributed Applications Using ARM API.....	26
2.3.4 Comparison of Approaches to Performance Data Transfer and Storage in ARM-supported Performance Monitoring Systems	26
2.3.4.1 HP OpenView MeasureWare Agent.....	27
2.3.4.2 Tivoli TME 10 Distributed Monitoring.....	29
2.3.4.3 BMC Best/1	31
2.3.4.4 Carleton University ARM 2.0 Prototype.....	31
2.3.4.5 Conclusion	33
2.3.5 Evaluation of ARM 2.0	34
2.4 Summary	34
Chapter 3	36
Performance Database Design	36
3.1 Performance Database Design	36
3.1.1 Relational Database	36
3.1.2 Database Schema	36
3.2 Database Technologies.....	39
3.2.1 Open Database Connectivity (ODBC).....	39
3.2.2 Java Database Connectivity (JDBC).....	42
3.2.3 Performance Measurement of ODBC and JDBC	45
3.2.4 DB2 CLI, Embedded SQL and Stored procedure.....	47
3.2.4.1 DB2 CLI	47
3.2.4.2. Embedded SQL.....	48
3.2.4.3 Stored Procedure.....	50
3.3 Summary and Conclusions.....	54

Chapter 4	55
Performance Database Daemon Design and Implementation	55
4.1 Qualitative Evaluation of Performance Database Daemon.....	55
4.2 Performance Database Daemon Design Issues	59
4.2.1 Threading Strategies	59
4.2.2 Buffering Strategies	60
4.2.3 Performance Tuning for Insertion.....	61
4.2.4 Database Connection	66
4.3 Flow Control of the Performance Database Daemon.....	69
4.4 Summary	69
Chapter 5	71
Performance Analysis and Scalability of Performance Database Daemon.....	71
5.1 Performance Evaluation Objectives	71
5.2 Performance Evaluation Experiment Design	73
5.2.1 Performance Metrics.....	73
5.2.2 Performance Measurement Configuration.....	73
5.2.3 Experiment Design.....	75
5.3 Performance Measurement Results and Analysis	78
5.3.1 Aggregation level.....	78
5.3.2 Agent Reporting Period	85
5.3.3 Number of Clients.....	93
5.3.4 Number of ARM Agents.....	101
5.4 Predict the Scalability of Performance Database Daemon.....	108
5.5 Summary	109
Chapter 6	111
Conclusions.....	111
6.1 Summary	111
6.2 Contribution	112

6.3 Future Research.....	113
References.....	114
Appendix Aggregation Levels Supported by Carleton University ARM 2.0	
Prototype.....	117

List of Tables

Table 5.1 Performance Evaluation Experiments.....	77
---	----

List of Figures

Figure 1.1 Distributed Application Monitoring System Using ARM API	5
Figure 1.2 IBM Tivoli TME Data Storage and Transfer Architecture	10
Figure 2.1 MANDAS Architecture.....	18
Figure 2.2 DMS Architecture	20
Figure 2.3 Carleton University ARM 2.0 Prototype Architecture.....	24
Figure 2.4 HP OpenView ARM-supported Components	28
Figure 2.5 Tivoli ARM-supported Components.....	30
Figure 3.1 Performance_data_table	38
Figure 3.2 Open DataBase Connectivity (ODBC) Components	41
Figure 3.3 JDBC Components	44
Figure 3.4 Performance Comparison of JDBC and ODBC	46
Figure 3.5 Normal Application Accessing a Database Server.....	51
Figure 3.6 Application Accessing a Database Server using Stored Procedure.....	52
Figure 4.1 The Impact of Block Size on The Response Times of Data Insertion	63
Figure 4.2 The Impact of Table Size on the Response Times of Block Insertion	65
Figure 4.3 Memory Leak Problem of IBM DB2 ODBC Driver during Database Connection	68
Figure 5.1 Performance Measurement Configuration	75
Figure 5.2 Impact of Aggregation Level on the Performance Data Size.....	80
Figure 5.3 Impact of Aggregation Level on the Database Daemon CPU Demand	81
Figure 5.4 Impact of Aggregation Level on Database Daemon Computing Time.....	82
Figure 5.5 Impact of Aggregation Level on the Database Daemon Resource Utilization	83
Figure 5.6 Impact of Aggregation Level on the Client Cycle Time	84

Figure 5.7 Impact of Aggregation Level on the ARM Agent and Client Node CPU Utilization.....	85
Figure 5. 8 Impact of Agent Reporting Period on the Performance Data Size.....	87
Figure 5.9 Impact of Agent Reporting Period on the Database Daemon CPU Demand ..	88
Figure 5.10 Impact of Agent Reporting Period on the Database Daemon Computing Time	89
Figure 5.11 Impact of Agent Reporting Period on the Database Daemon Resource Utilization.....	90
Figure 5.12 Impact of Agent Reporting Period on the Client Cycle Time	91
Figure 5.13 Impact of Agent Reporting Period on the ARM Agent and Client Node CPU Utilization.....	92
Figure 5.14 Impact of Number of Clients on the Performance Data Size	95
Figure 5.15 Impact of Number of Clients on the Database Daemon CPU Demand	96
Figure 5.16 Impact of Number of Clients on the Database Daemon Computing Time ...	97
Figure 5.17 Impact of Number of Clients on the Database Daemon Resource Utilization	98
Figure 5.18 Impact of Number of Clients on the Client Cycle Time	99
Figure 5.19 Impact of Number of Clients on the ARM Agent and Client Node CPU Utilization.....	100
Figure 5.20 Impact of Number of ARM Agents on the Performance Data Size	102
Figure 5.21 Impact of Number of ARM Agents on the Database Daemon CPU Demand	103
Figure 5.22 Impact of Number of ARM Agents on the Database Daemon Computing Time	104
Figure 5.23 Impact of Number of ARM Agents on the Database Daemon Resource Utilization.....	105
Figure 5.24 Impact of Number of ARM Agents on the Client Cycle Time	106
Figure 5.25 Impact of Number of ARM Agents on the ARM Agent and Client Node CPU Utilization.....	107

Chapter 1

Introduction

The purpose of the thesis is to design, implement and evaluate a performance database daemon that accepts performance data from *Application Response Measurement* (ARM) agents in the Carleton University ARM 2.0 Prototype. The development of the daemon and a measurement infrastructure to perform load tests are the main contributions of the thesis.

This chapter gives a brief introduction to distributed application monitoring architectures using ARM-based architecture. We also introduce the problem we are trying to address.

1.1 Introduction to Distributed Application Monitoring Systems

Business applications in the world today are critical elements of practically every business and organization. Determining whether these applications are satisfying their performance objectives is an important issue for system management. To be able to proactively solve performance problems or effectively forecast computing and networking resource requirements to handle growth or shortfalls, we must understand how *applications* consume system and network resources.

Distributed application performance monitoring can be defined as the process of dynamic collection, interpretation and presentation of information concerning objects or software processes. It is needed for various purposes such as debugging, testing, program

visualization and animation. It may also be used for general management, system configuration management, fault management and security management. In general, the behavior of a system is observed and monitoring information is gathered. This information is used to make management decisions and perform the appropriate control actions on the system.

Although many techniques have been created in host-centric environment to address this issue, these techniques are not satisfactory for most distributed applications. Because of the rapid migration toward distributed applications, management vendors have begun to address distributed application performance with new techniques.

There are a number of fundamental problems associated with performance monitoring of distributed systems:

- There are delays in transferring performance information from the place it is generated to the place it is used. This means that the performance data may be out of date.
- The monitoring system may itself compete for resources with the system being observed and modify the system's behavior.
- Information from heterogeneous systems must be coalesced.

In order to overcome these problems, it is necessary to design a monitoring system in terms of a set of platform independent services that support the generation, processing, distribution and presentation of monitoring information.

This thesis focuses on support for application level instrumentation. *Transactions* for the performance management system are defined as application specific units of work, a set of elementary actions that the designer of the application program wants to monitor, for example, the time it takes to perform a database request. The transactions should be application units that need to be measured, monitored, and for which corrective actions can be taken if the performance is determined to be poor.

There are several ways transaction data have traditionally been collected on centralized systems:

- *Transaction Processing Monitors* (TP) allow the capturing of some form of resource consumption data.
- Databases provide facilities to capture transaction activities within the context of each database access.
- Particular operating system facilities may have a built-in notion of what a transaction is and will store or report information related to that transaction.
- Program developers may embed their own instrumentation within application code at the request of analysis in order to get transaction specific data.
- Application profilers that gather data on how an application is behaving may exist for a particular operating environment.

Each of these methods has advantages and shortcomings. The most obvious shortcoming is that the transaction activity is captured in the context of the software layer measured, not necessarily relating to the business unit. When applied to the distributed environment, the biggest problem for all current methods is the lack of ability to track resource consumption by a transaction when several elements in a network are contributing towards the completion of the transaction. This means that none of the above methods provides integrated instrumentation.

In this thesis, we focus on the application instrumentation with *Application Response Measurement Application Programming Interface* (ARM API), which is described briefly in the next section. The application instrumentation refers to the technique that specialized software components are incorporated into programs to provide mechanism for measuring performance. An ARM architecture will be discussed in more detail in Chapter 2. Other distributed performance monitoring systems such as *Management of Distributed Applications and Systems* (MANDAS), *Distributed Measurement System* (DMS) are introduced in Chapter 2.

1.2 Introduction to Distributed Monitoring Using the ARM API

Application level information is needed to address application related problems. The application source code can be instrumented. ARM is an API jointly developed by an industry partnership that aims to monitor the availability and performance of applications in heterogeneous systems. The ARM API began as separate and independent projects at IBM Tivoli Systems and Hewlett Packard. Both projects had similar goals, and each had resulted in implementations that were generally available as products.

The purpose of the ARM API is to enable applications to provide information to measure transactions from the perspective of an end user. ARM APIs are called to measure components of response times in distributed applications. These components are portions of code, such as a CORBA object's methods, that are defined as *transactions*. This information can be used to support service level agreements and analyze response times across heterogeneous distributed systems. The ARM API allows vendors to create management-ready applications and end users to measure and control the total performance of their business critical distributed applications.

Figure 1.1 Distributed Application Monitoring System Using ARM API

Figure 1.1 illustrates a distributed application monitoring system using ARM API. In this architecture, the distributed application (client and server) is instrumented by ARM API calls. ARM agent captures the performance metrics about the client and logs the performance data in a repository. The performance data is retrieved by the management application.

Figure 1.1 illustrates the monitoring of a distributed application system using the ARM API. The ARM API is a simple API that applications can use to pass vital information about a transaction to an agent. The application calls the API just before a transaction (or a subtransaction) starts (*arm_start*) and then again just after it ends (*arm_stop*). The ARM library will return the appropriate ids to the ARM API calls and calculate the metrics as a result of the transactions. These metrics may then be logged, monitored or cause alarms. The API is supported by an agent that measures and monitors the transactions, and makes the information available to management applications. The business transaction time (client response time), time in critical components of application code (application server response time) and the time spent in key database transactions (database server response time) are all captured by the ARM API calls. All the performance data is registered in a storage system. The performance data is retrieved by management application and then reports or models are generated based on the retrieved data.

ARM has two versions. ARM 1.0 provides a way to measure each individual transaction in a distributed application, but not any way to understand how they are related to each other. In ARM 1.0, the transactions are measured without regard to whether they are composed of other transactions. In practice, many client/server transactions consist of nested subtransactions. It is very useful to know that a transaction is slow, but even more useful to know which subtransactions contribute most to the delays.

Many client/server transactions consist of one transaction visible to the user, and any number of nested component transactions that are invoked by the visible transaction. These component transactions are the children of the parent transaction (or the child of another child component transaction). It is very useful to know how much each component transaction contributes to the total response time of the visible transaction. Similarly, a failure in one of the component transactions will often lead to a failure in the visible transaction, and this information is also very useful.

ARM 2.0 provides a way to correlate data about transactions using a client/server programming model. Using ARM 2.0 an application can provide the parent/child information needed to know how transactions and subtransactions relate to each other. There are two facilities that the application developer can use to provide this information to measurement agents that implement the ARM 2.0 API [1].

- On the same *arm_start* , the application can request that the measurement agent assign and return a *correlator* for this instance of the transaction (that is a parent correlator). Note that the agent has the option of not providing the correlator, because it may not support the capability (ARM Version 1.0 agents do not support correlators), or because it is operating under a policy to suppress generating them.
- When indicating the start of a child transaction with an *arm_start*, the application can provide a correlator obtained from a parent transaction. This allows the measurement agent to know the parent/child relationship.

1.3 Data Storage and Transfer Problem in Distributed Application Monitoring Systems

Performance monitoring is definitely data-based. Vast amounts of information (especially in large, complex networks) are collected by the agents and sent to the management applications. The agents collect performance data. The management applications maintain historical and statistical data, handle events and reports. All this information, which explodes in size with network complexity and size augmentation, need not only be stored efficiently but it must also be enriched with powerful data management features that allow the realization of demanding, high level management functions like temporal reasoning, decision-making and planning.

Management applications may manipulate performance data in full detail. A summary, a historical collection or a statistical analysis of these data can be generated. A database

management system is a commonly accepted solution for this purpose and it is central to the development of an efficient performance management system for large networks. The performance database is very important in the distributed monitoring infrastructure. The performance data collected by the ARM agents running on many node must be transferred and stored in a cost-effective manner. Examples of ARM-supported performance monitoring architectures/products using DBMS include HP's OpenView MeasureWare [2] and IBM Tivoli's TME 10 [4].

Although distributed performance monitoring has been an important research topic for the past few years, little research has been published in the area of performance data management and in particular the cost of storing and retrieving monitored data. Furthermore, the appearance of open database technologies such as ODBC and JDBC enables the development of open systems. The open database technologies support migration and transparency, but may lose availability or scalability. These technologies will be discussed in Chapter 3.

1.4 Conventional Approaches to Transferring and Storing Performance Data

In most commercial performance monitoring systems, the typical approach to transferring and storing performance data is to let the agent write the performance data in local repository first with a user-defined frequency. The data then gets transferred to management sites later on. The major ARM supported performance management products including HP OpenView MeasureWare agent [2], Tivoli TME 10 agent [4] and BMC BEST/1 agent [5] use local log files to store the performance data temporarily. We give a brief introduction to their ARM supported portions in this section. Chapter 2 will examine them in detail.

HP OpenView is the ARM-supported product which offers users integrated network, system, application and database management. It provides ARM support as part of its HP

MeasureWare resource and performance management solutions. The ARM API is an integrated component of the HP OpenView management API set.

The HP MeasureWare Agent collects comprehensive resource and performance information across the distributed environment. The agent summarizes, timestamps, logs, and alarms on all the collected data from the application, database, network, and operating system [2]. However, little information is published about how the log files get transferred to the database, either by the MeasureWare agents or other intermediate processes.

With Tivoli TME 10 Distributed Monitoring product, the ARM agents collect detailed data for real-time problem analysis and write the data in a summarized format to the sequential file at the end of each interval (typically 10-15 minutes). The *Tivoli Reporter* retrieves performance records from the log files, reduces them and writes them into an SQL database [4]. Figure 1.2 gives the high level view of its architecture.

Figure 1.2 IBM Tivoli TME Data Storage and Transfer Architecture

Figure 1.2 illustrates the IBM Tivoli TME 10 performance data storage and transfer architecture. In this architecture, the ARM agent writes the performance data to local log files first. The log files get transferred to the Tivoli Reporter, which filters the data and writes the data to the performance database.

As we can see from the above introduction about the ARM-supported commercial management products, the typical way to store and transfer performance data is to let the agents save the performance data in a local log file first and the log file gets transferred to the management sites later on. The issue here is how the log files written by agents get transferred to management sites. Tivoli's data reporter is responsible for the transmission, but little information is released about how the HP OpenView MeasureWare agent transfers the performance data in the log files to management sites.

The advantages of the above approach include reliability, low likelihood of lost data even if performance database goes down for a while. The disadvantage is the extra memory-disk overhead on the managed node.

One possible alternative to the data transfer issue is to have each ARM agent transfer its monitored data to the database directly without writing the data to log files. The downside of this direct approach is that every ARM agent needs to know the database location, database access methods. In addition, if the database schema has any changes, the ARM agent must be changed as well. Another problem with this approach is that the number of database connections that can be supported by the DBMS is limited. If the ARM agents interact with the database directly, that means, all the ARM agents have to open and close database connections when they need to transfer the data. If many ARM agents are trying to send data to the database at the same time, it is possible that the number of agents exceeds the number of database connections that can be supported. In this case, some ARM agents cannot obtain database connections and the collected performance data will be delayed or even lost.

1.5 Contribution of the Thesis

In this thesis, we propose a performance data transfer and storage strategy which aims to minimize the disk and network overhead on the managed nodes by reducing logging activity. A database daemon is introduced on the performance database node that accepts performance data from agents and submits it to the database. A measurement study is

conducted to assess the performance costs of gathering and storing performance data using ARM based monitoring.

1.6 Thesis Outline

The thesis contains 6 chapters. The second chapter describes distributed application performance monitoring and the various architectures including *Management of Distributed Applications and Systems* (MANDAS), *Distributed Measurement System* (DMS) and ARM. We also examine the different approaches to the performance data transfer and storage problem in the ARM-supported systems in more detail.

The third chapter discusses the performance database design and open database technologies including *Java DataBase Connectivity* (JDBC) and *Open DataBase Connectivity* (ODBC). The performance of JDBC and ODBC is evaluated. Other technologies including *DB2 Call Level Interface* (DB2 CLI), Embedded SQL and stored procedures are also discussed in that chapter.

Chapter 4 discusses the design and implementation issues about the performance database daemon and analyses the various factors that affect the system behavior and performance the most. The advantages and disadvantages of the database daemon are also examined in this chapter.

Chapter 5 presents the results of performance evaluation of the performance database daemon. The impact of various factors on the daemon resource utilization (CPU, disk and network) is discussed.

Conclusion are given in Chapter 6.

Chapter 2

Distributed Application Performance Monitoring System Architectures

This chapter introduces four distributed application performance monitoring architectures: Application Response Measurement (ARM) [6], Management of Distributed Applications and Systems (MANDAS) [7], Distributed Measurement System (DMS) [8] and Carleton University ARM 2.0 Prototype [9]. We then focus on examination of different approaches to the performance data transfer and storage in major ARM-supported commercial performance management products including HP OpenView MeasureWare [2], IBM Tivoli TME 10 [4] and BMC BEST/1 [5].

2.1 Introduction to Distributed Application Performance Monitoring

The applications that are used to run businesses have changed dramatically over the past few years. In the early 1980s, business critical applications generally executed on large computers, and were accessed from dumb terminals. Non-networked applications executing on personal computers were just beginning to be used. Since then, these two application models have moved steadily towards each other, fusing together to form distributed (networked) applications.

These applications provide unprecedented opportunities for organizations to reach more customers with more useful services. These services are critical for the success in many business markets. The applications boost productivity and increase the flexibility and

responsiveness of the organizations that use them. Because they are so important, these applications, and the networking and computing systems that they run on, are critical to the success of these organizations.

Effective application management requires a focus on how an application's various components interact with the components of other applications and with resources such as operating systems, databases, middleware applications and Internet-based applications.

Monitoring the performance and the availability of distributed applications has not proven easy to do, since these applications have more dependencies on systems which spread over a wide geographical area. They partition functions throughout the network, and they exploit many different technologies. The distributed applications have the following characteristics:

- One business transaction may spawn several other component transactions, some of which may execute locally and some remotely. Any measurement agents that exist only in the network layer or in a host (server) will not see the entire picture.
- The data may be sent through network using various protocols, not just one, making the task of correlation much more difficult.
- Client/sever applications can be complex, taking different execution paths and spawning different subtransactions, depending on the results of previous subtransactions. Every permutation could take a different form when it goes across the communication link, making it much harder to reliably correlate network or host observations.

In spite of the difficulties, the need to monitor distributed applications has never been so great. Performance monitoring is increasingly being used in mission-critical roles.

Approaches to Gathering Performance Measures

Several technical approaches to gathering measures from applications are being used:

- **Network probes**

Network probes are used between client and server in an attempt to measure application response time. This approach can only measure client/server times and does not address client-only applications, 3-tier applications, or client time independent of the network. This approach lacks flexibility, is complicated to set up and costly to implement.

- **Non-intrusive Runtime Instrumentation**

Non-intrusive Instrumentation means no source code modifications are needed. This approach addresses both in-house applications, for which source code is available, as well as third party applications, for which source code is not available. This allows both in-house applications and third party applications to be monitored and response performance metrics gathered for applications that span enterprise environments without modifying the application.

Typically the runtime environment of an application is instrumented. This approach usually captures the elapsed time between the activities such as a button click or menu selection from the user's perspective or the time for an RPC. However, the runtime instrumentation cannot capture information about the context of these activities. This makes it difficult to use the information for the purpose of performance management.

- **Application Level Instrumentation**

Application Level Instrumentation means the modifications to the application source code. Instrumenting an application directly permits measures of actual response time based upon exactly what the end-user sees. This method is the most flexible and provides most useful management data over other alternatives. Unfortunately it has to modify the source code and has performance overhead.

2.2 Distributed Application Monitoring Systems

We introduce two distributed application performance monitoring systems: MANDAS (section 2.2.1) and DMS (section 2.2.2).

2.2.1 Management of Distributed Applications and Systems (MANDAS)

The objective of MANDAS project was to provide tools and techniques to allow the successful management of distributed applications and systems. An architectural framework for distributed application and system management was developed, and populated with components for configuration management, monitoring and control, performance data gathering and modeling, and storage of management and monitoring data. The components were integrated with existing standard protocols and components for system and network management.

The key areas of MANDAS research at Carleton University included the automated development of predictive performance models for the application systems, the use of analytic performance evaluation techniques to predict their behavior [10] and methods to identify the locations of performance problems in the applications and systems [11]. The key components of the framework are described as follows:

- **Distributed application instrumentation package**

A package was developed to capture application level performance information about operational distributed applications and submit it to a performance data storage system [13].

- **Performance data storage system**

A distributed computing environment server was created to store performance information about operational distributed applications. The server supports automated model building by performing a statistical analysis of measured data that gives

confidence intervals for measured data and more importantly deduces some performance metrics needed for model building that can not be measured directly.

- **A model building system**

A tool was developed that gathers information about operational applications from the performance data storage system. The data is used to assign parameters in a *Layered Queuing Model (LQM)* [23] file. The model can then be evaluated by the *Method of Layers (MOL)* [14].

Figure 2.1 illustrates the MANDAS architecture [21]. The *Management Tools* could be used to perform various management activities such as configuration, analysis of performance bottlenecks, report generation, visualization of network or system activity, simulation, modeling and so on. The heart of the architecture is *Management Services* that are composed of four subsystems, namely configuration subsystem, monitoring subsystem, control subsystem, and management information repository subsystem. The *Management Information Repository Subsystem* provides a logically centralized view of the management information and provides a single interface to access to the data and data sources. Information repository service may be used by the monitoring service to store data being collected from management agents. *Management Agents* exist for carrying out management activities on behalf of management services and tools.

Figure 2.1 MANDAS Architecture

2.2.2 Distributed Measurement System (DMS)

The Distributed Measurement System (DMS) is a software-based measurement infrastructure for monitoring the performance of distributed application systems. It was developed by researchers at Hewlett-Packard. DMS provides correlated performance metrics across application components and their channels (network communication), integrates disparate performance measurement interfaces from the operating system, and efficiently transports collected data from network nodes to management stations [8].

Figure 2.2 DMS Architecture

The DMS is a framework of sensors, standard interfaces, and monitoring processes that initialize, control, access, and present performance data. Figure 2.2 illustrates DMS architecture:

- **Sensors** are located throughout the application's address space, and may reside in application and stub source code, and in libraries such as the DCE Run Time Library.
- **Observer** is a mechanism within the process's address space that manages the sensors and optimizes the transfer of data outside the address space. It transfers the sensor data once per reporting interval.
- **Collector** is a node level object that controls sensors and performs node-level sensor data management. It provides transparent network access and control of sensors for higher levels of the DMS architecture using the *Collector Measurement Interface* (CMI). The collectors obtain sensor data from all observers on the node through the *Collector Data Interface* (CDI).
- **Analyzer** analyzes the data gathered by collector. It computes the higher moments of the collected data, correlates data from components of distributed application and prepares data for expert system or human analysis. The collector periodically transfers sensor data to the analyzer via the *Analyzer Data Interface* (ADI).
- **Performance Measurement Interface (PMI)** is the standard interface for accessing and controlling performance data collected by the measurement system in a heterogeneous network.

DMS has both measures based on methods and sensors, but it does not provide ways to correlate information of subtransactions.

2.3 ARM-based Distributed Performance Monitoring System

In this section, we give the detailed architecture of an ARM-based distributed application monitoring system and discuss the performance data storage approaches in the ARM

supported commercial management products like HP OpenView MeasureWare[3], IBM Tivoli TME 10 [4] and BMC's BEST/1 [5].

2.3.1 Application Response Measurement (ARM) API

With the Application Response Measurement (ARM) API, the distributed applications are enabled to be managed by the measurement agents that implement the ARM API. The ARM API is designed to support the instrumentation of units of work that contributes to business transactions. These transactions should be something that need to be measured, monitored, and for which corrective actions can be taken if the performance is determined to be poor. With the critical information about business transactions provided, application management software can measure and report service level agreements, get early warning of poor performance, notify operator or automation routines immediately if transactions are failing, and help determine where slowdowns are occurring.

The ARM API is a simple API that applications can use to pass vital information about a transaction to an agent. The ARM API is made up of a set of function calls that are contained in a shared library. A performance measurement agent that supports the ARM API provides its own implementation of the shared library. When the application is instrumented with ARM API function calls, it can be monitored by an agent that implements the shared library. The ARM calls identify the application, the transaction, and (optionally) the user, and provide the status of each transaction when it completes.

The following is an overview of the ARM API calls:

- **arm_init** During the initialization of the application, *arm_init* is called to name the application and optionally the users, and initialize the ARM environment for the application. A unique identifier is returned that must be passed to *arm_getid*.
- **arm_getid** *arm_getid* is used to name each transaction in the application. This is usually done during the initialization phase of the application. A transaction class is a description of a unit of work, such as "Check Account Balance". In each program,

each transaction class may be executed one or many times. *arm_getid* returns a unique identifier that must be passed to *arm_start*.

- **arm_start** Each time a transaction class is executed, this is a transaction instance. *arm_start* signals the start of execution of a transaction instance and returns a unique handle to be passed to *arm_update* and *arm_stop*.
- **arm_update** This is an optional function call that can be made any number of times after *arm_start* and before *arm_stop*. *arm_update* gives information about the transaction instance, such as *heartbeat* after a group of records have been processed.
- **arm_stop** *arm_stop* signals the end of the transaction instance and the elapsed time of the transaction can be calculated.
- **arm_end** At termination of the application, *arm_end* is called to cleanup the ARM environment for the application. There should be no problem if this call is not made, but memory may be wasted because it is allocated by the agent even though it is no longer needed.

2.3.2 ARM-based Distributed Application Monitoring System Architecture for Carleton University ARM 2.0 Prototype

The Carleton University ARM 2.0 Prototype is an ARM-based distributed application monitoring system. It will be discussed in Section 2.3.4.4. We use it to illustrate the typical components in ARM-based monitoring system. Figure 2.3 shows the components in the Carleton University ARM 2.0 Prototype:

- Instrumented applications components and the nodes
- ARM agents on managed nodes
- ARM manager and its node
- Performance database and its node
- Management application and its node

Figure 2.3 Carleton University ARM 2.0 Prototype Architecture

2.3.2.1 Instrumented application

Distributed applications should be instrumented by the calls to ARM API.

2.3.2.2 ARM Agent

The ARM agents are installed on the managed client nodes and responsible for collecting the performance metrics about the instrumented applications. The ARM agent should have very low overhead in the application's system and in the small portion that runs in the application's address space.

2.3.2.3 Performance Data Storage

In current commercial implementations such as HP OpewnView MeasureWare, IBM Tivoli TME 10, the performance data collected by the ARM agents are usually written to the local repository periodically. The data in the log files is then transferred to a performance database. Most implementations of ARM-supported products provide a database to store the performance data from ARM agents. In Carleton University ARM 2.0 Prototype, the local log files are eliminated and a database daemon is introduced. The performance data collected by the ARM agents is reduced at the time of capture and transferred to the performance database daemon periodically. The database daemon is responsible for the performance data buffering, parsing and inserting into the database.

2.3.2.4 Management Application

A wide range of application monitoring capabilities can be provided by the management application, from summary-level views of entire distributed system to detailed analysis views. Management applications read performance data from the repository and support visulization, build models and locate the performance bottlenecks and test/debug. System availability and resource consumption can be studied at a high level and then drilled down into the intricate details of the system. The management application must be

equipped with real-time information thus effective action can be taken quickly to reduce system downtime and increase efficiency, fine-tune the system and truly manage the availability of the applications.

2.3.3 Steps of Monitoring Distributed Applications Using ARM API

The general strategy of monitoring distributed applications involves three steps:

- **Define key business transactions**

This is the first and most important step. Application developers need to define what performance data is collected and how the data will be used. For applications that are developed to meet the requirements of critical businesses, it is common and useful for this step to be a joint collaboration between the users of the application, the system and network administrators, and the developers.

- **Make calls to ARM API from the application**

The second step is to modify the program to include the calls to the ARM APIs. Null libraries can be used for initial testing. Because the API calls are simple, this step is not difficult or time-consuming. The key is to know where the monitors should be placed, which is determined by defining the critical business transactions in the first step.

- **Replace null libraries with an ARM-compliant agent and management applications**

The null libraries must be replaced with an ARM-compliant agent and associated management applications.

2.3.4 Comparison of Approaches to Performance Data Transfer and Storage in ARM-supported Performance Monitoring Systems

In this section, we discuss the ARM supported components in commercial management products including HP OpenView MeasureWare, IBM Tivoli TME 10 and BMC Best/1 in more detail. Their approaches to the performance data transfer and storage are examined.

2.3.4.1 HP OpenView MeasureWare Agent

Hewlett-Packard Company currently supports ARM in its OpenView GlancePlus, MeasureWare and PerfView resource and performance management suites. HP OpenView PerfView and MeasureWare Agent software monitor the performance of critical client/server applications from a user's perspective. The data collected by MeasureWare agents is the primary data source for the PerfView suite of analysis tools [3].

Figure 2.4 illustrates the ARM-supported components in HP OpenView product. The figure only shows that the MeasureWare Agent supports ARM API calls and collects resource information on system activities. The performance data is written to local log files periodically [2]. It provides PerfView with the data that is used to analyze, understand, and make informed decisions regarding the computing environment. No information is published about how the performance data collected by the MeasureWare agent get transferred to the management applications (PerfView) or where the performance database is located.

Figure 2.4 HP OpenView ARM-supported Components

2.3.4.2 Tivoli TME 10 Distributed Monitoring

IBM Tivoli Systems provides support for the ARM API in its Tivoli family of network computing management products. The ARM agent for Tivoli TME 10 monitors individual application transactions. Applications call the ARM agent at the beginning and end of each transaction using the ARM API. Thresholds are monitored and events are sent to the management console. Summary records are logged to a sequential file for later processing.

Figure 2.5 illustrates the ARM-supported components in TME 10 Distributed Monitoring product. *Tivoli Reporter* processes the log files by collecting and filtering the data based on predefined rules, then stores the data in a SQL database. This data can be used to track past performance and availability and to project future requirements [4].

Figure 2.5 Tivoli ARM-supported Components

2.3.4.3 BMC Best/1

BMC Software Inc. is a worldwide developer and vendor to provide solutions to ensuring the availability, performance and recoverability of business critical applications.

The BMC BEST/1 is designed to help manage and understand complex Windows NT and Unix computing environments. To meet their needs, the BEST/1 product provides the ability to:

- Monitor resources and analyze deviations from normal performance
- View and report resource consumption in meaningful client/server application views
- Predict the impact of change on response times
- Identify precise hardware requirements prior to application deployment
- Forecast the need for additional computing resources
- Track long-term performance trends to better understand demand

The ARM agent for BEST/1 runs as a fault-tolerant process and acts as the channel by which the managing node and the managed node communicate, and ensures continual performance data collection. The performance metrics such as threads, processes, kernel, logical volumes and paging are collected. The metrics are maintained in memory and written to disk at user-defined frequencies. The collected data is stored in a local data repository on the managed node and then consolidated on the management console for analysis and prediction [5].

2.3.4.4 Carleton University ARM 2.0 Prototype

ARM provides simple APIs for distributed application instrumentation to incur as little overhead as possible. To manage overhead, the events generated within an application process may be aggregated over a reporting period before being reported. Carleton University ARM 2.0 prototype introduces 30 workload abstractions to ARM 2.0 [9].

Those abstractions are based on process, software, and business functions that provide detail suited towards application oriented performance management tasks.

To manage the overhead of ARM instrumentation, an ARM implementation may support the reporting of performance information at several levels of detail and abstraction. A level of detail controls whether means, higher moments, and/or percentiles are captured and reported for events. A workload abstraction decides the coarseness of reported information. Each abstraction causes a different overhead and is best suited to support some subset of management tasks. The abstraction level has the same meaning as *Aggregation Level*, which is the term used in next chapters. For the full list of aggregation levels supported by the Carleton University ARM 2.0 Prototype, see the Appendix.

We give a brief introduction to the six aggregation levels that are used for performance evaluation in Chapter 5. QNM stands for *Queuing Network Model*, LQM stands for *Layered Queuing Model*.

- No Instrumentation
- Full Trace
- QNM Low Resolution (By Process, no correlation by Business Function Type)
- QNM High Resolution (By Process, with correlation by Business Function Type)
- LQM Low Resolution (By Method, with correlation by Business Function Type)
- LQM High Resolution (By Method, with correlation by Request Type)

No Instrumentation would have the lowest overhead and generate least performance data. *Full Trace* mode would have the highest overhead and generate the largest amount of performance data among the six levels. The other four levels are used for generating performance model including Queuing Network Model (QNM) and Layered Queuing Model (LQM). They would exhibit the behavior between the *No Instrumentation* and *Full Trace*.

Queuing Network Models (QNMs) are used to model the way in which processes make use of shared devices such as CPUs and disks. These models have typically been used to study the performance of mainframe systems. For more details about QNMs, see [22].

LQMs (Layered Queuing Models) are extensions of QNMs that also reflect interactions between client and server processes. The processes may share devices, and server processes may also request services, by RPC, from other processes. LQMs are appropriate for describing distributed application systems such as CORBA, DCE, OLE and DCOM applications. For more details about LQMs, see [23].

2.3.4.5 Conclusions

As we can see from previous introduction and discussion, the commercial management products that support ARM APIs use a similar approach to the performance data storage and transfer: the ARM agent stores the collected performance data in a local log file periodically on the managed node and the data gets transferred to the management sites later on.

Since the vendors try to provide the whole solution to the distributed application monitoring and management, the approach to performance data transfer and storage does not seem to be a key issue in their implementations. So it is meaningful to study and propose performance data storage solutions which are more accurate, efficient, flexible and scalable.

In Chapter 3 and Chapter 4, we are going to present an approach to transferring and storing the performance data from the Carleton University ARM 2.0 agents to support the workload abstractions. This approach supports the use of a database daemon that helps avoid the need for ODBC/JDBC drivers to access performance database from all managed nodes. The database daemon is responsible for accepting performance data from ARM agents and interacting with the performance database.

2.3.5 Evaluation of ARM 2.0

ARM 2.0 API is now supported by many key industrial players [15]. The ARM API provides a mechanism for addressing the key service management issues during the development of an application. It can be used when source code changes can be made to an existing application, or when the application run-time can be instrumented by the ARM API calls. The research at Carleton University for ARM 2.0 also allows for many workload abstractions using the same instrumentation (for example, QNM, LQM) [9].

The availability of the ARM API has not, however, solved the problem for many applications that already developed and where source code changes are not possible. Examples of such applications include packaged solutions (where the users must wait until the application vendor instruments the application) and applications that are considered functionally stable, without planned investment in development.

2.4 Summary

In the network computing world of the late 1990s, managing distributed applications is a key challenge. Comprehensive solutions are needed that include administrative tasks, monitoring at the application level, and monitoring the transactions of individual users.

The ARM API will be a key component for transaction level monitoring. It will not be the complete solution for all situations, because it requires applications to be instrumented to invoke the API - which is not always possible. However, the ARM API does provide unique capabilities that other solutions cannot provide. Ideally, the ARM API will provide the core transaction monitoring capability, augmented by other solutions. The most important advantage of using the ARM API is that it offers a true business-oriented perspective.

Performance database is a critical component in the distributed application performance monitoring systems. Having an efficient and scalable data transfer and storage system is very important for the success of the monitoring system.

Chapter 3

Performance Database Design

In this chapter, we present a performance database design for Carleton University ARM 2.0 Prototype and discuss various database technologies including Java Database Connectivity (JDBC), Open Database Connectivity (ODBC), Embedded SQL, DB2 Call Level Interface (CLI) and stored procedures. The performance of JDBC and ODBC is compared, the one with better performance is chosen as access method to the performance database.

3.1 Performance Database Design

3.1.1 Relational Database

IBM's DB2 Universal Database 5.0 (DB2 UDB 5.0) is a relational database management system that contains features and tools that enable users to create, update, control, and manage relational databases using SQL. The performance database described in this thesis was created using DB2 UDB 5.0. Other RDBMSs such as Oracle, Sybase, Informix could also be used but they may have different performance characteristics.

3.1.2 Database Schema

The efficient storage and manipulation of the performance data is a critical issue during database schema design. The performance database schema for the ARM 2.0 prototype has over a dozen tables to store the static and dynamic information about the managed nodes and applications. The static information includes the information about ARM

aggregation levels, which have been defined before the system is deployed. The dynamic information captures configuration and performance data about hosts, agent instances, processes, transactions, methods and objects. The dynamic information is generated as applications execute.

Among all the tables, the *Perf_data_table* stores the performance record about the applications that are instrumented by ARM API calls. The performance data include the counters, response times and resource usage. This is the most frequently updated table in the database. Figure 3.1 gives the attributes of *Perf_data_table*.

Perf_data_table
Agent_instance
Agent_vendor_id
Agent_version
Tran_id
Start_handle
Caller_ag_instance
Caller_ag_vendorid
Caller_ag_version
Caller_tran_id
Caller_start_handle
Request_type_id
Response_sum
Response_sum_sq
Response_counter
Inter_arr_sum
Inter_arr_sum_sq
Inter_arr_counter
Start_time
End_time
Tran_status
Aggregation_level
CPU
Disk
Delay
Think
Call_type

Figure 3.1 Performance_data_table

3.2 Database Technologies

In the following sections, various database technologies including JDBC, ODBC, Embedded SQL, DB2 CLI and stored procedures are discussed and the performance of ODBC and JDBC is compared. ODBC was developed by Microsoft Corporation and based on the Call Level Interface specification of the SQL Access Group, it allows users to access data in heterogeneous environments of relational and non-relational databases. The JDBC API is a specification by which Java application developers can access many different kinds of computer database systems regardless of their location and platform. DB2 Call Level Interface (CLI) is IBM's callable SQL interface to the DB2 family of database servers. Embedded SQL refers to the use of standard SQL commands embedded within a host language such as C. Stored procedures are used for modular design and shift the workload from a client application to the database server.

3.2.1 Open Database Connectivity (ODBC)

Open Database Connectivity (ODBC) is a programming interface introduced by Microsoft Corporation in 1992. It was developed as a means of providing applications with a single API through which to access data stored in a wide variety of database management systems (DBMSs) [17]. Prior to ODBC, applications written to access data stored in a DBMS had to use the proprietary interfaces specific to that database. If application developers wanted to provide their users with heterogeneous data access (access to data in more than one data source), they needed to code to the interface of each data source. Applications written in this manner are difficult to code, maintain and extend.

The ODBC architecture consists of four main components as shown in Figure 3.2.

- ODBC Applications
- ODBC Driver Manager
- ODBC Driver
- Data Source

An ODBC application calls ODBC functions to submit SQL requests and retrieve results. The ODBC Driver Manager loads ODBC drivers and routes function calls from the applications to the proper ODBC driver. The ODBC driver processes ODBC function calls, submits requests to the database management system, and returns results to the Driver Manager. The Data Source is the component to which applications connect. The Data Source contains the data that the user of the application wants to access, the database management system and its associated operating system, and any network used to access the database management system.

ODBC provides two ways to submit SQL statement to the DBMS for processing: direct execution (using *SQLExecDirect*) and prepared execution (using *SQLPrepare* and *SQLExecute*). Prepared execution is useful if a statement will be executed many times. Under prepared execution, upon receiving the *SQLPrepare* function the data source will compile the statement, produce an access plan, and return the access plan to the driver. The data source will then use this plan when it receives an *SQLExecute* statement. For statements that are executed multiple times, prepared execution creates a performance advantage because the access plan need only be created once. But for statements that are executed just once, prepared execution creates added overhead, and hence there is a performance hit. Direct execution is the proper choice for statements that are executed a single time. Using the correct execution strategy is one way of optimizing application performance.

Figure 3.2 Open DataBase Connectivity (ODBC) Components

Figure 3.2 illustrates the major components in ODBC architecture. The four major components are: ODBC Applications, ODBC Driver Manager, ODBC Driver and Data Source.

ODBC supports a technique called *Record Blocking* that can greatly improve the performance of database request. It can reduce the number of network flows by transferring a block of database rows between the client and server. This technique dramatically increases performance if it can be properly used. To use the record blocking technique in ODBC, an application uses *SQLParamOptions* to specify multiple values for the set of parameters assigned by *SQLBindParameter*. The ability to specify multiple values for a set of parameters is useful for bulk inserts and other work that requires the data source to process the same SQL statement multiple times with various parameter values. An application can, for example, specify three sets of values for the set of parameters associated with an INSERT statement, and then execute the INSERT statement once to perform the three insert operations.

3.2.2 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a Java API for executing SQL statements. It consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API [18].

The JDBC API defines Java classes to represent database connections, SQL statements and result sets. It allows a Java programmer to issue SQL statements and process the results. JDBC is the primary API for database access in Java. The JDBC API is implemented via a driver manager that supports multiple drivers connecting to different databases. JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries. The JDBC driver manager is the backbone of the JDBC architecture. It actually is quite small and simple, primary function is to connect Java applications to the correct JDBC driver and then get out of the way (see Figure 3.3).

In JDBC, a *Connection* object represents a connection with a database. A connection session includes the SQL statements that are executed and the results that are returned over that connection. A single application can have one or more connections with a single database, or it can have connections with many different databases.

A *Statement* object is used to send SQL statements to a database. There are actually three kinds of *Statement* objects, all of which act as containers for executing SQL statements on a given connection: *Statement*, *PreparedStatement* and *CallableStatement*. They are specialized for sending particular types of SQL statements: a *Statement* object is used to execute a simple SQL statement with no parameters; a *PreparedStatement* object is used to execute a precompiled SQL statement with or without input parameters; and a *CallableStatement* object is used to execute a call to a database stored procedure.

Because *PreparedStatement* objects are precompiled, their execution can be faster than that of *Statement* objects. Consequently, an SQL statement that is executed many times is often created as a *PreparedStatement* object to increase efficiency.

JDBC provides Java programmers a powerful API that is consistent with the rest of the Java language specification. The major advantage of JDBC over ODBC is that, coupled with one or more JDBC drivers, a single Java application can issue SQL statements to any number of database servers, regardless of their locations and platforms. In addition, Java's portability among many different architectures allows the same Java program to run on many desktop computers within an enterprise network.

Figure 3.3 JDBC Components

Figure 3.3 illustrates the JDBC major components: Java Application, JDBC Driver Manager, JDBC Drivers and Proprietary Database Access Protocols.

3.2.3 Performance Measurement of ODBC and JDBC

The purpose of the measurement is to compare the performance of JDBC and ODBC and choose the one with better performance as the database access method to the performance database.

A benchmark was created to evaluate the behavior. The benchmark contains five jobs. Each of the five jobs uses one of the five approaches listed below and inserts 10,000 records into the table with three integer fields. All jobs were run under Windows NT 4.0 to an IBM DB2 Universal Database Version 5.0. The measurement is conducted on a single machine to avoid the impact of network. Note that at the time of writing this thesis no record blocking technique exists for JDBC.

1. *JDBC SQLExecDirect*
2. *JDBC PreparedStatement.*
3. *ODBC SQLExecDirect*
4. *ODBC PreparedStatement*
5. *ODBC Block Insert (Insertion Using Record Blocking Technique)*

Figure 3.4 gives the measurement results of 10 replications. The confidence intervals of the reported measures are all within $\pm 5\%$ of mean with 95% confidence level.

As we can see from the figure, ODBC Block Insertion gives the lowest response times, which are less than 10 seconds; other methods take much longer response times, which are more than 90 seconds. ODBC Block Insertion technique is determined to be used by the performance database daemon for inserting multiple performance records.

Figure 3.4 Performance Comparison of JDBC and ODBC

3.2.4 DB2 CLI, Embedded SQL and Stored procedure

Besides the open database technologies like JDBC and ODBC, there are other proprietary ways to query and manipulate the database. We give a brief introduction to DB2 CLI, embedded SQL and stored procedure (For more details, please check the web site [20]). Since we have no Software Development Kit for those methods for the time being, no measurement result is given. Those methods are supposed to have a better performance than open technologies like ODBC and JDBC for supporting complex queries.

3.2.4.1 DB2 CLI

DB2 Call Level Interface (CLI) is the IBM callable SQL interface to the DB2 family of database servers. DB2 CLI is based on the Microsoft Open Database Connectivity (ODBC) specification, and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 Call Level Interface in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2 specific extensions have been added to help the application programmer exploit DB2 features.

DB2 CLI uses function calls to pass dynamic SQL statements as function arguments. Through DB2 CLI, applications use procedure calls at execution time to connect to databases, to issue SQL statements, and to get returned data and status information. It is an alternative to embedded dynamic SQL, but unlike embedded SQL, it does not require host variables or a precompiler. Applications developed using this interface may be executed on a variety of DB2 databases without being compiled against each of the databases.

The advantages of CLI include the elimination of the need for precompiling and binding the program, as well as the increased portability of the application through the use of the Open Database Connectivity (ODBC) interface which is supported by CLI. DB2 APIs can be used in both embedded SQL and DB2 CLI applications. Many programming

languages are supported, the applications can be written in C, COBOL and FORTRAN to call DB2 APIs.

Applications that use DB2 APIs cannot be ported easily to other database products. An application written using CLI only uses dynamic SQL. There is some additional overhead in processing imposed by the CLI interface itself.

3.2.4.2. Embedded SQL

Embedded SQL refers to the use of standard SQL commands embedded within a procedure programming language such as C. Embedded SQL is a collection of all SQL commands, such as SELECT and INSERT, available with SQL with interactive tools and flow control commands, such as PREPARE and OPEN, which integrate the standard SQL commands with a procedural programming language. Embedded SQL must be supported by precompilers, which interpret embedded SQL statements and translate them into statements that can be understood by the procedure language compilers.

Embedded SQL has the advantage that it can consist of either static or dynamic SQL or a mixture of both types.

When the syntax of embedded SQL statements is fully known at precompile time, the statements are referred to as static SQL. If the SQL statements will be frozen in terms of content and format when the application is in use, using embedded static SQL in the application should be considered. The structure of an SQL statement must be completely specified in order for a statement to be considered static. For example, the names for the columns and tables referenced in a statement must be fully known at precompile time. The only information that can be specified at run time are values for any host variables referenced by the statement. However, host variable information, such as data types, must still be precompiled.

When a static SQL statement is prepared, an executable form of the statement is created and stored in a *package* in the database. The executable form of the statement can be constructed either at precompile time, or at a later bind time.

Programming using static SQL requires less effort than using embedded dynamic SQL. Static SQL statements are simply embedded into the host language source file, and the precompiler handles the necessary conversion to database manager run-time service API calls that the host compiler can process.

Static SQL statements are persistent, meaning that a statement exists as long as its package exists. The key advantage of static SQL, with respect to persistence, is that the static statements exist after a particular database is shut down, whereas dynamic SQL statements must be explicitly compiled at run time (for example, by using the PREPARE statement). A static SQL statement executes faster than the same statement processed dynamically since the overhead of preparing an executable form of the statement is done at precompile time instead of at run time.

Dynamic embedded SQL can be used where the statements that need to be executed are determined while the application is running. This creates a more generalized application program that can handle a greater variety of input. Dynamic SQL statements are cached until they are either invalidated, freed for space management reasons, or the database is shut down. If required, the dynamic SQL statements are recompiled implicitly by the SQL compiler whenever a cached statement becomes invalid.

Dynamic SQL allows an application to execute SQL statements containing variables whose values are determined at run-time. An application prepares a dynamic SQL statement by associating a SQL statement containing placeholders with an identifier and sending the statement to a server to be partially compiled and stored. The statement is then known as a "prepared statement". When an application is ready to execute a prepared statement, it defines values to substitute for the placeholders of SQL statements and sends a command to execute the statement.

3.2.4.3 Stored Procedure

A database application can be designed to run in two parts, one on the client and the other on the server. The stored procedure is the part that runs at the database within the same transaction as the application. Stored procedures can be written in either Embedded SQL or using the DB2 CLI functions. A stored procedure may use any sequence of standard SQL statements, and operate on any tables in the database for which the stored procedure is defined.

Stored procedures support modular design. They encapsulate complex tasks that are used by embedded applications. They also shift the workload from a client application to the server. Stored procedures can be given privileges on the database that users do not have. They can be executed from other stored procedures or embedded SQL applications.

Figure 3.5 shows how a normal database management application accesses a database located on a database server. All database access must go across the network. This, in some cases, results in poor performance. Figure 3.6 shows an application which accesses a database server using stored procedure.

Using stored procedures allows a client application to pass control to a stored procedure on the database server. This allows the stored procedure to perform intermediate processing on the database server, without transmitting unnecessary data across the network. Only those records that are required by the client need to be transmitted. This can reduce network traffic and improve overall performance.

Figure 3.5 Normal Application Accessing a Database Server

Figure 3.6 Application Accessing a Database Server using Stored Procedure

In general, stored procedures have the following advantages:

- **Reduced network traffic**
Applications may process large amounts of data but require only a subset of the data to be returned to the user. A properly designed application using stored procedures returns only the data that is needed by the client, the amount of data transmitted across the network is reduced.
- **Improved performance of server intensive work**
Applications executing SQL statements can be grouped together without user intervention by using stored procedure. The more SQL statements that are grouped together, the larger the savings in network traffic. A typical application requires two trips across the network for each SQL statement, whereas an application using the stored procedure technique requires two trips across the network for each *group* of SQL statements. This reduces the number of trips, resulting in savings from the overhead associated with each trip.
- **Access to features that exist only on the database server**
Store procedure can access features that are installed on the database server but not accessible to the user.
- **Encapsulation (information hiding)**
Users do not need to know the details about the database objects in order to access them by using stored procedures.
- **Security**
User's access privileges are encapsulated within the package(s) associated with the stored procedure(s). So there is no need to grant explicit access to each database object. For example, a user can be granted run access for a stored procedure that selects data from tables for which the user does not have select privilege.

Stored procedures have disadvantages, however. Stored procedure applications have special compile and link requirements. The client procedure must be part of an executable file, while the stored procedure must be placed in a library on the database server.

3.3 Summary and Conclusions

In this chapter, we discussed various database technologies including JDBC, ODBC, DB2 CLI, Embedded SQL and stored procedures. We also compared the performance of ODBC and JDBC. Since JDBC and ODBC are the open technologies, they have the advantages of portability and transparency. Other technologies that can be used to manipulate the database include Embedded SQL, DB2 CLI and stored procedure. They are likely to have a better performance for supporting complex queries, but unfortunately they also require special development environments and may lose the portability and transparency, which are also critical issues in open distributed systems.

The major advantage of JDBC is the platform independence. However, for a performance management system that generates large amounts of performance data for processing, performance is more important. Since no record blocking technique exists for JDBC so far, the performance of JDBC is much poorer than ODBC using record blocking technique. As a result, ODBC with the record blocking technique is chosen as the access method to the performance database.

Chapter 4

Performance Database Daemon Design and Implementation

In this chapter we discuss the design and implementation issues of the performance database daemon for the Carleton University ARM 2.0 Prototype. The database daemon is responsible for the performance data transfer and storage. It sits between the ARM agents and the performance database, accepts the performance data, parses the data and inserts the records into the database. We focus on the various factors that affect the daemon's behavior and performance the most.

4.1 Qualitative Evaluation of Performance Database Daemon

As we discussed in Chapter 2, the typical approach to storing and transferring performance data in distributed application monitoring systems is to have the agent write the data in local log files first and transfer the data to management sites later on. We propose a variant of this approach in this chapter. In our approach, we have a database daemon whose purpose is to accept the performance data from the ARM agents, parse the data and insert the records into the database. Only the database daemon has to worry about the database schema and how to interact with the database. Thus it is easier to modify the database daemon to accommodate any changes to the database.

In ARM-based application monitoring systems, the ARM agents should incur as little overhead as possible in the monitored systems. They should not have to worry about how

to transfer the performance data and how to update the database. We list several advantages of using performance database daemon:

- **ARM agents do not have to worry about the database**

First of all, ARM agents should be kept small and fast and should have as little impact on the instrumented system as possible. By using a database daemon, the ARM agents will not be burdened by the database access issues, such as what the database schema is, where the database is located or how to access the database. The managed nodes do not need to install ODBC/JDBC drivers either.

- **Easier to change the database if we have a database daemon**

Secondly, The ARM agents should not have to know any changes to the database tables. The tables may be reorganized and modified. In this case, it is difficult to terminate and restart all the ARM agents if the ARM agents interact with the database directly. By using a database daemon, any changes to the database will not affect the ARM agents.

- **Easier to embrace new technologies**

The third reason is that new database technologies (for example, non SQL databases) are always being developed, embracing the technology will give the better performance to the monitoring system. New technologies can be easily integrated into the database daemon without impacting ARM agents if a database daemon is deployed.

- **Better performance**

In our approach, the ARM agent sends performance data directly to the database daemon. The overhead of generating and retrieving log file on the managed node is avoided.

- **Portability**

Because we are using the open technology ODBC, the system is suitable for heterogeneous environment. The database daemon implemented in ODBC can access any type of database management systems, it has the advantage of location and migration transparency.

The disadvantages of the database daemon and possible solutions are described as follows:

- **Server Availability**

Availability is defined as the percentage of time the system is available. The availability of a service depends on the reliability of the network components, server providing the service, and the system architecture. The system should be built so that the failure of one server or network link cannot cause the service to become unavailable. Such situation can be avoided by duplicating the service to several servers and having optional network routing to them. In such a system, the failure of a server or network link only means loss of capacity but the system keeps working.

The performance database daemon is responsible for processing the data from the ARM agents, if the daemon crashes, the whole performance data may be lost. The performance data transfer portion is centralized for the time being, only one daemon is deployed, thus it lacks the availability.

To improve the server availability, it is preferable to have more than one daemon running on different machines. It is the responsibility of ARM manager daemon to detect any database daemon failure. It should inform the ARM agents to buffer the performance data in local log files temporarily in case of daemon shut down or make connections to other available database daemons.

- **Scalability**

For a large distributed application system, scalability has a critical impact on the success of the system. In our designed performance database system, the scalability is determined by the following factors. First of all, the number of ARM agents that can send data simultaneously to the daemon is limited by the number of sockets that is supported by the database daemon. In Microsoft Visual C++, the default value of socket number for a process is set to 64, this value can be changed to 128 explicitly [19]. Secondly, the supported number of ARM agents is limited by the resource available to the database

daemon, such as CPU, disk and network. In Chapter 5, we will give the performance evaluation results about the database daemon and discuss the scalability.

Another factor that limits the system scalability is that the database portion of current system architecture is centralized. Duplicating database daemon and distributing the performance database in accordance with scalability option for a database product is the most likely path for scalability. Alternatively, an ARM management domain with too many agents could be split into several smaller domains each with their own performance databases.

- **Data burst**

Another potential problem of the database daemon involves data burst. When large number of ARM agents are sending data to the database daemon at the same moment, the daemon may not be able to handle all the requests. The possible outcome is that some agents may be forced to wait for a long time to establish connection to the database daemon. One solution is to set timer in the ARM agent, if it timeouts, then the agent knows that it has to save the performance data in local log files and try to send the data later on.

- **Lack of Reliability**

Network failures or database daemon crashes will make the ARM agents disconnected. To prevent the possible data loss, timer can also be introduced. For example, for more than two reporting periods, the agent has to either discard its data or log in and resend the log later on.

4.2 Performance Database Daemon Design Issues

The performance database daemon is designed to accept the performance data directly transferred from the ARM agent's memory. The ARM agents do not write the performance data to local log files. It is the database daemon's responsibility to parse the performance data and insert the records into the database.

We discuss the database daemon design issues in the following sections. Section 4.2.1 discusses the threading strategies, Section 4.2.2 describes the buffering strategies, in Section 4.2.3, we talk about the performance tuning for insertion. Section 4.2.4 discusses the database connection issue.

4.2.1 Threading Strategies

In a client/server computing environment, both client and server may benefit from multi-threading. However, the advantages of multi-threading are more apparent for servers than for clients. The database daemon acts as a server to accept the performance data from ARM agents. The advantages of using multithreaded database daemon will be examined.

For some servers, it is satisfactory to accept one request at a time and to process each request to completion before accepting the next. Where parallelism is not required by an application, there is little point in making such a server multi-threaded. However, some servers would offer a better service to their clients if they processed a number of requests in parallel. Parallelism of such requests may be possible because a set of clients can concurrently use different objects in the same server, or because some of the objects in the server can be used concurrently by a number of clients.

In an ARM management domain, a lot of ARM agents may send data simultaneously to the performance database daemon. It is very important to let the database daemon have concurrency, since some operations can take a significant amount of time to execute. The operations may be compute bound, or they may perform a large number of I/O

operations. If the daemon can execute only one such operation at a time, the ARM agents will suffer because of long latencies before their requests can be processed. The benefits of multi-threading are that the latency of requests can be reduced, and the number of requests that a daemon can handle over a given period of time (that is, the server's throughput) can be increased.

The simplest threading model is that a thread is created automatically for each incoming request. Each thread executes the code for the operation being called, sends the reply to the caller, and then terminates. Any number of such threads can be running concurrently in a server, and they can use normal concurrency control techniques (such as mutex or semaphore variables) to prevent corruption of the server's data. The performance database daemon uses this simple model to handle the request from ARM agents.

Threads have their cost, however. Firstly, it may be more efficient to avoid creating a thread to execute a very simple operation. The overhead of creating a thread may be greater than the potential benefits of parallelism. Nevertheless, the benefits frequently outweigh the costs and multi-threaded servers are considered essential for many applications.

The performance database daemon is implemented using Microsoft Visual C++ 5.0. Microsoft Visual C++ provides support for creating multithread applications with 32-bit versions of Microsoft Windows (Windows NT and Windows 95). With Visual C++, there are two ways to program with multiple threads: use the *Microsoft Foundation Class* (MFC) library or the C run-time library and the Win32 API. We use the C run-time library to create the threads. For more information on creating multithread applications using Microsoft Visual C++, check [19].

4.2.2 Buffering Strategies

In Chapter 3, we discuss the ODBC record blocking technique that is used to improve the performance of the database daemon. This technique requires the performance records be buffered and inserted together as a block. Most part of the data from ARM agents are the

records for table *Perf_data_table*, their size is usually very large. These records need to be saved in buffers and inserted into the database as a block. It is important to determine how to buffer them, either using the main memory, or using log files on the database daemon, or using memory mapped files.

To avoid disk I/O overhead on the database daemon node, we determine to let the daemon buffer the performance records in the main memory. The database daemon starts a new thread each time a new ARM client gets connected with the daemon through TCP connection. The spawned thread receives performance data from the ARM agents. The thread then parses the data according to the predefined format. The fields of the records for the tables are retrieved and saved in data structures (arrays) in the main memory. The records are then inserted into the database, either using direct insertion (for single records) or block insertion (for multiple records, e.g., records of *Perf_data_table*).

Another issue is how to use the socket efficiently. The maximum number of sockets that a windows socket application can make use of is determined at compile time. The default value in Microsoft Visual C++ is 64. This number can be changed to 128 by the application programmer [19]. The ARM agents send performance data periodically to the database daemon. The typical reporting period is 2-15 minutes, and the actual transfer time is usually much shorter than the reporting period. So it is more efficient to let the ARM agents close the TCP connection after the data is transferred.

4.2.3 Performance Tuning for Insertion

The performance database contains over a dozen tables. For most of the tables, the performance records from the ARM agents are single ones, thus it is more efficient to use *SQLDirectExec* to insert the single records directly. But for multiple records transmitted from the ARM agents, especially for the *Perf_data_table*, hundreds or thousands of records may be sent from the ARM agents. Using block insertion can greatly reduce the response times for inserting multiple records for *Perf_data_table* (Block insertion is discussed in Section 3.2.1).

The database daemon must pre-allocate buffers for each of the SQL tables. Some tables store static information and are not updated very often. The number of records for these tables are usually very small. For these tables, the allocated buffer for the records does not have to be very large. For other tables that are updated very often, it is better to allocate large buffers for them, since larger block for insertion give better performance. For the time being, only the records for *Perf_data_table* need a large buffer.

Since we determine to use the ODBC record blocking technique to improve the performance of inserting multiple records, we need to choose the appropriate block size for the insertion. The block size is defined as the number of records that is to be inserted at a time. The relationship between the response time and the block size has been studied. Figure 4.1 shows that the response time is greatly reduced when the block size increases from 1 to 2, from 2 to 5, from 5 to 10 and from 10 to 25. The response time of block size 25 is already less than 10 seconds. The larger sizes than 25 continue to have a better performance, although the performance improvement is small. The conclusion is that even a relatively small block size like 25 gives a good performance and is a good choice because it limits the amount of memory allocated for buffering the records.

Figure 4.1 The Impact of Block Size on The Response Times of Data Insertion

Larger block size does have better performance, but the size cannot be arbitrarily large. The testing result shows that there is some limit to the block size. Exceeding that limit causes memory allocation problem and the daemon terminates abnormally. Furthermore, the greater the block size, the longer it takes for data to propagate to the database. This could impact management applications that require timely data.

Since the performance database is growing with time, the table size may have effects on the insertion response times. The test result illustrated in Figure 4.2 shows that the response times do not grow with the table size.

Figure 4.2 The Impact of Table Size on the Response Times of Block Insertion

Figure 4.2 illustrates the impact of table size on the response times of block insertion. The experiment measures the response times of inserting 10,000 records into a table that contains 3 integer fields. After each insertion, the table size is increased by 10,000 records. We can see from the result that the table size has no impact on the insertion response time.

4.2.4 Database Connection

In order to access the database, a database connection must be opened before the daemon can insert the records. Opening and closing database connections can be very time-consuming. Under ODBC, upon opening a connection, the driver manager loads the driver DLL and calls the driver's *SQLAllocEnv* and *SQLAllocConnect* functions, plus the driver's connect function corresponding to the connection option chosen by the application. The user receives a *handle* that identifies the connection for use with subsequent SQL requests. Upon closing a connection, the driver manager unloads the DLL and calls all the disconnect function: *SQLDisconnect*, *SQLFreeConnect*, and *SQLFreeEnv*. For this reason, from a performance perspective, it is preferable to leave connections open, rather than closing and reopening them each time a statement is executed. However, there is a cost to maintain open, idle connections. Each connection consumes a significant amount of resource on the server, which can cause problems on PC-based DBMSs that have limited resources. Therefore, application must use connection judiciously, weighing the potential costs of any connection strategy.

Our testing result shows that the number of database connections that can be supported simultaneously by a database created using DB2 Universal Database 5.0 is limited to 31. One strategy to use the database connection is to open a database connection while the ARM agent is connected with a daemon thread and close it after the completion of the data transfer. This approach means each ARM agent connected to the database daemon consumes one database connection. Thus in this approach, the number of ARM agents that can be connected to the database daemon simultaneously is also limited to 31.

Besides the poor scalability, the above approach has another problem. We found that the IBM DB2 UDB 5.0 ODBC driver has a memory leak problem which will exhaust the system memory if the database connection keeps opening and closing. Figure 4.3 shows that the memory used for the database handles is not released completely to the system after the connections are closed. Therefore the memory leak problem of ODBC prevents the use of the dynamic approach described above. Our testing result also shows that the

system memory will be exhausted after 100 to 200 times of opening and closing database connections (The testing program only contains statements to open and close the database connection, no other computing is involved).

To avoid the memory leak problem and use the database connection more efficiently, it is better to make single database connection and keep it open. Each time an ARM agent opens a TCP connection and gets connected with the database daemon, a new daemon thread is spawned to handle the request. The data is transferred from the ARM agent to the daemon thread and gets buffered, parsed and sent to the database using the single database connection. Since the database connection is shared by all the daemon threads, it is protected by mutual exclusion.

Figure 4.3 Memory Leak Problem of IBM DB2 ODBC Driver during Database Connection

4.3 Flow Control of the Performance Database Daemon

The performance database daemon is implemented as a multithreaded process that communicates with the ARM agents through sockets. The daemon uses ODBC record blocking technique to insert multiple performance records of tables like *Perf_data_table* into the database. It opens a single database connection that is shared by all daemon threads. The control flow of the implemented database daemon is described as follows:

- The performance database daemon opens a database connection and keeps it open. The daemon listens to a predefined port and spawns a new thread for each TCP connection from ARM agents.
- The spawned daemon thread receives performance data from ARM agents, the beginning of the data indicates which table's data is to be processed.
- The daemon thread is responsible for parsing the data, buffering the parsed fields, and updating the database using the shared database connection.
- The server thread exits when it completes processing all the data from ARM agent in the TCP connection.

4.4 Summary

In Carleton University ARM 2.0 Prototype, a performance database daemon is introduced to accept performance data from ARM agents and write the data into the performance database. The daemon has advantages including simplifying the functionality of the ARM agents, making the performance database easier to maintain and upgrade, achieving a better performance for updating the database and portability. The disadvantages include lack of availability, scalability and reliability. These disadvantages can be overcome by introducing multiple database daemons and partitioning the management domain and performance database, which is a very interesting research topic in the future.

The design and implementation of the database daemon focus on the performance and optimization. The database daemon is designed to be a multithreaded process that accepts performance data from ARM agents through TCP connections. It uses ODBC record blocking technique to improve the performance for inserting multiple records of tables like *Perf_data_table*. To avoid the memory leak problem and make efficient use of the database connection, a single database connection is opened and shared by all the daemon threads. The database connection is protected by mutual exclusion.

Chapter 5

Performance Analysis and Scalability of Performance Database Daemon

In this chapter, we discuss the performance evaluation of the performance database daemon. Section 5.1 gives the evaluation objectives and possible factors that may affect the daemon's performance. Section 5.2 discusses the experiment design. Section 5.3 shows the measurement results and gives the analysis. Section 5.4 gives the prediction of database daemon's scalability.

5.1 Performance Evaluation Objectives

The purpose of performance evaluation of database daemon is to provide a systematic determination of the load capabilities of the system. We will see how the database daemon handles the required number of clients and ARM agents and the storage capacity required by the system. We also determine the potential performance bottlenecks of the system.

The performance impact of following factors on the resource utilization and response times in a closed environment are evaluated:

- **Aggregation level**

Different aggregation levels generate different amounts of performance data (see Appendix for the full list of aggregation levels supported by the Carleton University ARM 2.0 prototype). Among them, *Full Trace* should be the worst case since it tracks all

the details of the instrumented application. *No Instrumentation* should be the best case since no record for table *Perf_data_table* is generated. Other aggregation levels should exhibit the behavior between the *Full Trace* and *No Instrumentation*.

- **Agent Reporting Period**

The agent reporting period determines how frequently the performance data are collected and reported to the database daemon. Shorter reporting periods result in more frequent data generation and transmission.

- **The number of clients**

Increasing number of clients will increase the amount of data to be collected by the ARM agents.

- **The number of ARM agents**

The more the ARM agents, the greater the number of concurrent connections and the greater the volume of data that must be supported by the database.

The scalability of the system is defined as the number of ARM agents the database daemon can support. We will discuss the scalability of the database daemon in Section 5.4 based on:

- *CPU utilization*
- *disk I/O time*
- *network utilization*

5.2 Performance Evaluation Experiment Design

This section determines how the database daemon performs under normal and large user loads. Subjecting the daemon to the loads provides valuable information about performance problems and guidance on how to scale up the daemon to handle the desired number of clients and ARM agents.

5.2.1 Performance Metrics

Response time, disk I/O time, CPU time and communication cost in distributed systems are the common performance metrics. We measure the following ones:

- Performance Data Size Received by Database Daemon (byte)
- Database Daemon Computing Time for the Performance Data Received by Database Daemon (millisecond, thereafter referred to as *ms*)
- Database Daemon Disk Utilization for the Performance Data Received (%)
- Database Daemon CPU Demand (ms)
- Database Daemon CPU Utilization (%)
- Database Daemon Node CPU Utilization (%)
- Client Cycle Time (ms)
- ARM Agent CPU Utilization (%)
- Client Node CPU Utilization (%)
- Network Utilization (%)

The performance metrics are collected through Microsoft Visual C++ 5.0 *Performance Data Helper* (PDH) interface [19]. The resource utilization is measured every 5 minutes to reduce the measurement overhead.

5.2.2 Performance Measurement Configuration

Figure 5.1 shows the system configuration for the performance measurement. All components, including the ARM manager daemon, ARM agent, client application, database daemon are running on Windows NT 4.0 workstations, which communicate with each other through 100Mbit/sec Ethernet. One ARM manager daemon and one database daemon run on the same machine, which is a 200 MHz Pentium Pro machine with 64 MB of main memory and SCSI I/O subsystem with a single disk. An ARM agent is installed on each client node, where the client applications are running. The ARM agents collect the performance data from the client applications that are instrumented using ARM API calls. The performance database daemon accepts the data from ARM agents through TCP connections, parses the data and inserts the records into the performance database, which is created using IBM DB2 Universal Database 5.0. The agent reporting period and aggregation level are specified by the ARM manager daemon.

The client application is an emulation of a three-tier application. Clients use Microsoft DCOM [24] to interact with two levels of servers that also communicate using DCOM. The total CPU time used by application processes for an end to end multi-tier request was approximately 10 ms for this application. When the client application is running, there are actually 3 ARM libraries reporting to the ARM agent installed on the client node.

Figure 5.1 Performance Measurement Configuration

5.2.3 Experiment Design

The factors that may affect the system performance include the aggregation level, agent reporting period, number of clients and number of ARM agents. Table 5.1 lists the experiments for evaluating the impact of the various factors.

The Carleton University ARM 2.0 Prototype supports 30 aggregation levels (see Appendix for more details). Six typical aggregation levels are chosen to be measured:

- No Instrumentation
- Full trace
- QNM Low Resolution (By Process, no correlation by Business Function Type)
- QNM High Resolution (By Process, with correlation by Business Function Type)
- LQM Low Resolution (By Method, with correlation by Business Function Type)
- LQM High Resolution (By Method, with correlation by Request Type)

The agent reporting period has 2 levels: 300 seconds and 60 seconds. The number of clients has 3 levels: 1, 10, and 25 clients. The number of ARM agents also has three levels: 1, 4 and 8 agents (due to the limited hosts in the lab).

Each experiment has 10 replications. 95% confidence intervals of the results are all within 5% of the reported mean values.

Test Case	Number of Clients (3 levels)	Number of ARM Agents (3 levels)	Aggregation Level (6 levels)	Agent Reporting Period (seconds) (2 levels)
1	1	1	No Instrumentation	300
2	1	1	Full Trace	300
3	1	1	By Process, no correlation by Business Function Type (QNM Low Resolution)	300
4	1	1	By Process, with correlation by Business Function Type (QNM High Resolution)	300
5	1	1	By Method, with correlation by Business Function Type (LQM Low Resolution)	300
6	1	1	By Method, with correlation by Request Type (LQM High Resolution)	300
7	1	1	By Method, with correlation by Request Type (LQM High Resolution)	60
8	10	1	By Method, with correlation by Request Type (LQM High Resolution)	300
9	25	1	By Method, with correlation by Request Type (LQM High Resolution)	300
10	10	4	By Method, with correlation by Request Type (LQM High Resolution)	300
11	10	8	By Method, with correlation by Request Type (LQM High Resolution)	300

Table 5.1 Performance Evaluation Experiments

Table 5.1 lists the test cases for evaluating the impact of various factors on the performance of database daemon. The factors include : aggregation level (6 levels: No Instrumentation, Full Trace, QNM Low Resolution, QNM High Resolution, LQM Low Resolution and LQM High Resolution), agent reporting period (2 levels: 300 seconds and 60 seconds), number of clients (3 levels: 1, 10 and 25) and number of agents (3 levels: 1, 4 and 8).

5.3 Performance Measurement Results and Analysis

The impact of various factors on the response times and resource utilization on the database daemon node and client node is illustrated in Figure 5.2 to Figure 5.25. The performance data size, database daemon CPU demand and daemon computing time shown in the figures are the values within 300 seconds period. The client cycle time is the sum of client think time (100 ms) and client response time.

5.3.1 Aggregation level

The impact of aggregation level is illustrated in Figure 5.2 to Figure 5.7. The experiments compare six aggregation levels: *No Instrumentation*, *Full Trace*, *QNM Low Resolution*, *QNM High Resolution*, *LQM Low Resolution* and *LQM High Resolution*. The experiments contain 1 ARM agent, 1 client and agent reporting period is 300 seconds.

Figure 5.2 illustrates the impact of aggregation level on the performance data size. Figure 5.3 shows the corresponding CPU demand by database daemon to process the performance data. Figure 5.4 illustrates the database computing time spent on the data. Figure 5.5 shows the database daemon resource utilization including network, CPU and disk. Figure 5.6 gives the client cycle time. Figure 5.7 shows the client node and ARM agent CPU utilization in different aggregation levels.

As we expected, *Full Trace* mode is the worst case and *No Instrumentation* is the best case. Figure 5.2 shows that *Full Trace* mode generates the greatest amount of performance data, *No Instrumentation* generates the least data. *QNM Low Resolution* generates less performance data than *QNM High Resolution*. *LQM Low Resolution* generates less performance data than *LQM High Resolution*. *QNM Low Resolution* and *QNM High Resolution* generate less data than *LQM Low Resolution* and *LQM High Resolution*.

Figure 5.3, 5.4 and 5.5 show that *No Instrumentation* consumes least daemon computing time and causes lowest resource utilization. *Full Trace* uses most daemon computing time and causes the highest resource utilization. *QNM Low Resolution* and *QNM High Resolution* use less daemon computing time and resources than *LQM Low Resolution* and *LQM High Resolution*. *QNM Low Resolution* uses less daemon computing time and resources than *QNM High Resolution*. *LQM Low Resolution* uses less daemon computing time and resource than *LQM High Resolution*. Among the resources (network, CPU and disk), disk utilization is the highest one and network utilization is the lowest one. This indicates that database daemon is disk bound.

On the client node, Figure 5.6 shows that *Full Trace* mode has some impact on the client cycle time. *QNM Low Resolution*, *QNM High Resolution*, *LQM Low Resolution* and *LQM High Resolution* have no detected impact on the client cycle time. The ARM agent and client node CPU utilization increases in aggregation level *LQM Low Resolution* and *LQM High Resolution* as shown in Figure 5.7.

Figure 5.2 Impact of Aggregation Level on the Performance Data Size

Figure 5.3 Impact of Aggregation Level on the Database Daemon CPU Demand

Figure 5.4 Impact of Aggregation Level on Database Daemon Computing Time

Figure 5.5 Impact of Aggregation Level on the Database Daemon Resource
Utilization

	Database Daemon Node CPU Utilization (%)	Database Daemon Disk Utilization (%)
No Instrumentation	0.133	0.159
Full Trace	1.602	6.091
QNM Low Resolution	0.135	0.202
QNM High Resolution	0.138	0.220
LQM Low Resolution	0.207	0.457
LQM High Resolution	0.252	0.635

Figure 5.6 Impact of Aggregation Level on the Client Cycle Time

Figure 5.7 Impact of Aggregation Level on the ARM Agent and Client Node CPU Utilization

	ARM Agent CPU Utilization (%)	Client Node CPU Utilization (%)
No Instrumentation	0.004	9.103
Full Trace	1.109	11.527
QNM Low Resolution	0.005	9.104
QNM High Resolution	0.005	9.105
LQM Low Resolution	0.062	9.273
LQM High Resolution	0.094	9.372

5.3.2 Agent Reporting Period

The measurement results about the impact of agent reporting period are illustrated in Figure 5.8 to Figure 5.13. Two agent reporting periods are compared: 300 seconds and 60 seconds. The experiments contain 1 ARM agent, 1 client and the aggregation level is *LQM High resolution*.

Figure 5.8 shows the impact of agent reporting period on the performance data size. The corresponding database daemon CPU demand is shown in Figure 5.9. The database daemon computing time is illustrated in Figure 5.10. Figure 5.11 gives the database daemon resource utilization (network, CPU and disk). The impact of agent reporting period on the client cycle time is shown in Figure 5.12. The impact on the ARM agent and client node CPU utilization is given in Figure 5.13.

Figure 5.8 shows that 60 seconds agent reporting period generates more performance data than 300 seconds agent reporting period during the same period of time because the former causes more frequent performance data generation. As a result, in 60 seconds agent reporting period, the daemon computing time increases (Figure 5.10), and the daemon resource utilization (network, CPU and disk) increases as well (Figure 5.9, 5.11). However, the performance data does not increase by factor of 5 in 60 seconds agent reporting period (Figure 5.8). The reason is that with shorter agent reporting period, the client application may not access all the transactions, thus less performance information is collected per agent reporting period.

The result also shows that the disk utilization is the highest and network utilization is the lowest among the resources (network, CPU and disk). No impact on the client cycle time is detected as illustrated in Figure 5.12. The agent reporting period affects the ARM agent and client node CPU utilization. 60 seconds agent reporting period has a higher ARM agent and client node CPU utilization than 300 seconds agent reporting period as shown in Figure 5.13.

Figure 5. 8 Impact of Agent Reporting Period on the Performance Data Size

Figure 5.9 Impact of Agent Reporting Period on the Database Daemon CPU Demand

Figure 5.10 Impact of Agent Reporting Period on the Database Daemon
Computing Time

Figure 5.11 Impact of Agent Reporting Period on the Database Daemon
Resource Utilization

Figure 5.12 Impact of Agent Reporting Period on the Client Cycle Time

Figure 5.13 Impact of Agent Reporting Period on the ARM Agent and Client Node CPU Utilization

5.3.3 Number of Clients

The measurement results given in Figure 5.14 to Figure 5.19 illustrate the impact of number of clients that run on a single node. The experiments measure 3 levels of number of clients: 1, 10 and 25. The experiments contain 1 ARM agent, aggregation level is *LQM High Resolution* and agent reporting period is 300 seconds.

Figure 5.14 shows the impact of number of clients on the performance data size. The impact on the database daemon CPU demand is shown in Figure 5.15. Figure 5.16 illustrates the database daemon computing time. Figure 5.17 gives the database daemon resource utilization (network, CPU and disk). The impact on the client cycle time is illustrated in Figure 5.18 and the impact on the client node and ARM agent CPU utilization is given in Figure 5.19.

Figure 5.14 shows that with more clients running on the client node, more performance data are generated by the ARM agent, thus the daemon computing time increases as shown in Figure 5.16. Correspondingly, the database daemon resource utilization (CPU, disk and network) increases as illustrated in Figure 5.15 and 5.17. The result also shows that disk utilization is the highest and network utilization is the lowest among the resources (network, CPU and disk).

Figure 5.18 shows that on the client node, due to the contention between clients for CPU, the client cycle time increases rapidly with the increase of number of clients. Figure 5.19 shows that the ARM agent and client node CPU utilization increases with the increase of number of clients.

We mentioned that 3 ARM libraries are reporting to the ARM agent when one client is running. For our sample client application, the number of ARM libraries reporting to the ARM agent on the client node is 3 times the number of clients, that means, there are 30 ARM libraries running for 10 clients and 75 ARM libraries running for 25 clients on the client node. So the measurement result gives a pessimistic measurement on the ARM

agent monitoring overhead. This explains why the ARM agent CPU utilization reaches 3.79% when 25 clients are running on the client node.

Figure 5.14 Impact of Number of Clients on the Performance Data Size

Figure 5.15 Impact of Number of Clients on the Database Daemon CPU Demand

Figure 5.16 Impact of Number of Clients on the Database Daemon Computing Time

Figure 5.17 Impact of Number of Clients on the Database Daemon Resource Utilization

Figure 5.18 Impact of Number of Clients on the Client Cycle Time

Figure 5.19 Impact of Number of Clients on the ARM Agent and Client Node CPU Utilization

5.3.4 Number of ARM Agents

The measurement results about the impact of number of ARM agents are illustrated in Figure 5.20 to Figure 5.25. Each client node has 10 clients running, aggregation level is *LQM High Resolution* and agent reporting period is 300 seconds. 3 levels of number of ARM agents are compared: 1, 4 and 8. For 1 ARM agent, total 10 clients are running on the client node. For 4 ARM agents, total 40 clients are running in the measurement system. For 8 ARM agents, total 80 clients are running in the measurement system.

Figure 5.20 gives the performance data size with various ARM agents. Figure 5.21 shows the corresponding CPU demand by the database daemon. The database daemon computing time is given in Figure 5.22. Database daemon resource utilization (network, CPU and disk) is given in Figure 5.23. Figure 5.24 shows the impact on the client cycle time and Figure 5.25 gives the client node and ARM agent CPU utilization.

Figure 5.20 shows that with more ARM Agents (i.e., with more clients), more performance data are generated and transferred to the database daemon for processing. Therefore, more daemon computing time is spent on the performance data during the same period of time (shown in Figure 5.22). As a result, the daemon resource utilization (network, CPU and disk) all increases (shown in Figure 5.21, 5.23) correspondingly.

Figure 5.24 shows that the client cycle time increases with the number of ARM agents. The reason is that when multiple ARM agents are trying to send performance data to the daemon at the same moment, the ARM agents have to wait till the TCP connection is established. This causes possible delay to the ARM agents. This delay may affect the behavior of the clients. The more the ARM agents, the longer the delay and the more impact on the clients. Another possibility is that the ARM agents have more context switches per data transfer to a busy server. Future work includes studying the impact of socket buffer size on client response times. Figure 5.25 shows that the number of ARM agents does not have significant impact on the client node and ARM agent CPU utilization.

Figure 5.20 Impact of Number of ARM Agents on the Performance Data Size

Figure 5.21 Impact of Number of ARM Agents on the Database Daemon CPU Demand

Figure 5.22 Impact of Number of ARM Agents on the Database Daemon
Computing Time

Figure 5.23 Impact of Number of ARM Agents on the Database Daemon
Resource Utilization

The result shows that the database daemon CPU and disk utilization increases with the number of ARM agents almost linearly.

Figure 5.24 Impact of Number of ARM Agents on the Client Cycle Time

Figure 5.25 Impact of Number of ARM Agents on the ARM Agent and Client Node CPU Utilization

5.4 Predict the Scalability of Performance Database Daemon

As shown in Figure 5.20, for a system with 8 ARM agents, if each client node has 10 clients running (total 80 clients in the system), aggregation level is *LQM High Resolution*, agent reporting period is 300 seconds, then the performance data generated every 5 minutes is approximately 4 Mbytes. That means the database daemon is able to process approximately 1.125 Gigabytes data 24 hours continuously under that configuration. Then a disk subsystem with 64 Gigabytes capacity (say, four 16 Gigabytes disks) can approximately support 56 days monitoring. In a production system, the data would rarely be retained online for more than a month.

The measurement result also shows that the performance database is disk bound. In aggregation level *LQM High Resolution* with 300 seconds agent reporting period, 8 ARM agents and 10 clients running on each client node, the disk utilization of the performance database daemon is approximately 40%, the daemon CPU utilization is approximately 16% and the network utilization is approximately 0.11%. Thus we can predict that a reasonable number of ARM agents that can be supported per disk is approximately 20 based on disk utilization. The maximum number of disks that can be supported is 5 based on the database daemon node CPU utilization. Thus the maximum number of client nodes that can be supported by the 200 MHz Pentium Pro with 64 MB RAM and 5 SCSI disks is 40, the corresponding network utilization is 0.55%.

Further scalability can be achieved using RAID disks or disk striping to increase potential table sizes and by partitioning ARM administrative domains to limit overall per-database load.

5.5 Summary

This chapter examines the impact of various factors on the performance data size, database daemon computing time, database daemon resource utilization, client cycle time, client node and ARM agent CPU utilization. The factors include aggregation level, agent reporting period, number of clients and number of ARM agents.

Six aggregation levels are compared: *No Instrumentation*, *Full Trace*, *QNM Low Resolution*, *QNM High Resolution*, *LQM Low Resolution* and *LQM High Resolution*. From the results shown in Section 5.3, we know that *Full Trace* mode gives the worst case and *No Instrumentation* is the best case. *Full Trace* mode generates largest amount of performance data, causes heaviest resource consumption including disk, CPU and network. This mode also has some impact on the client cycle time.

The other four aggregation levels exhibit the behavior between the *Full Trace* mode and *No Instrumentation* mode. Aggregation level *QNM Low Resolution* and *QNM High Resolution* are used for the generation of Queuing Network Model. *LQM Low Resolution* and *LQM High Resolution* are used to generate Layered Queuing Model. They generate more performance data than *QNM Low* and *QNM High Resolution* and have some impact on the ARM agent and client node CPU utilization. Among these four levels, *LQM High Resolution* generates the most data and causes the heaviest resource utilization. No impact on the client cycle time is detected.

Two agent reporting periods are compared: 60 seconds and 300 seconds. The measurement result reveals that increasing the agent reporting period reduces the generation and transfer of performance data, thus reduces the resource consumption by the database daemon and ARM agent. No impact on the client cycle time is detected.

Three levels of number of clients are compared: 1, 10 and 25. When more clients are running on one client node, more performance data are generated. The database daemon's CPU demand, disk time and network utilization all increase correspondingly. The client

cycle time increases rapidly with the increase of number of clients due to the contention for CPU. The ARM agent and client node CPU utilization increases with the increase of number of clients as well.

Three levels of number of ARM agents are compared: 1, 4 and 8. The number of clients running in the measurement system is 10 times the number of ARM agents, since each client node has 10 clients running. With more ARM agents (thus more clients), the more performance data is generated and transferred to database daemon. The resource consumption (network, CPU and disk) by the database daemon increases as well. As we can see from the result, the number of ARM agents does not have significant impact on the client node and ARM agent CPU utilization, since the number of clients on one node remains 10. However, the client cycle time increases slightly with the number of ARM agents.

The measurement result also shows that the performance database daemon is disk bound. Disk utilization is the highest one among the resources (network, CPU and disk). Based on the resource utilization, it is predicted that 40 client nodes (each node has 10 clients, aggregation level is set to *LQM High Resolution*, and agent reporting period is 300 seconds) can be supported by 200 MHz Pentium Pro with 64 MB RAM and 5 SCSI disks. The corresponding network utilization in the above system is 0.55%.

Chapter 6

Conclusions

6.1 Summary

The purpose of this thesis is to design, implement and evaluate a performance database daemon that accepts performance data from ARM agents in the Carleton University ARM 2.0 Prototype. The development of the daemon and a measurement infrastructure to perform load tests are the main contributions of the thesis.

In this thesis, various distributed application performance monitoring systems are discussed, including MANDAS, DMS and ARM. The performance data transfer and storage approaches in the ARM supported commercial products, including HP OpenView MeasureWare, Tivoli TME 10 Distributed Monitoring and BEST/1, are also examined.

To achieve better performance for the database daemon, various database technologies, including JDBC, ODBC, Embedded SQL, DB2 CLI and stored procedures, are explored. The performance behavior of JDBC and ODBC are measured. Since ODBC with record blocking technique gives much better performance than JDBC, we choose it as the access method to the performance database.

The performance database daemon has been designed as a multithreaded process that accepts performance data from ARM agents through TCP sockets. The design issues include the threading strategy, record buffering strategy, efficient use of database connection and choosing the appropriate block size for block insertion.

The objective of the performance evaluation of the database daemon is to see the capability of the daemon, i.e., the number of clients and the number of ARM agents that can be supported. The database daemon is deployed on a 200 MHz Pentium Pro machine with 64 MB of main memory and SCSI I/O subsystem with a single disk. The resource utilization including network, CPU and disk is measured to see the potential performance bottleneck of the database daemon and predict the scalability of the system. The measurement result shows that the database daemon is disk bound.

For a system with 8 ARM agents deployed, if each node has 10 clients running, aggregation level is *LQM High Resolution*, and agent reporting period is 300 seconds, then the performance data size generated every 5 minutes is approximately 4 Mbytes. That means the database daemon is able to process 1.125 Gigabytes of data every 24 hours. A disk subsystem with 64 Gigabytes capacity (four 16 Gigabytes disks) could support 56 days of continuous monitoring. Most systems would offload their data to tape much more frequently.

It is also predicted that 40 client nodes (each node has 10 clients, aggregation level is set to *LQM High Resolution*, and agent reporting period is 300 seconds) can be supported by a 200 MHz Pentium Pro machine with 64 MB RAM and 5 SCSI disks. The corresponding network utilization in the system is 0.55%.

6.2 Contribution

The major contribution of this thesis is to develop and measure a performance data storage system for an ARM-based distributed application performance monitoring system. This research is also valuable for other monitoring systems, since every monitoring system has to face the same problem as how to collect, transfer, buffer and store the huge amount of performance data in a cost-effective way.

6.3 Future Research

When the development environment for Embedded SQL, DB2 CLI and stored procedures is available, we can measure and compare their performance to see if they are helpful for the simple queries supported by current database daemon. In addition, these technologies are supposed to give a better performance for complex queries that may be supported in the future research.

Another interesting topic is about the scalability of the system. In current system, one database daemon is deployed for the purpose of performance evaluation. In a distributed system, it is very important to have multiple database daemons running to improve server availability and scalability. The other likely path to the scalability of the whole system is to distribute the performance database as well. For example, a group of ARM agents can have their own performance database, and the distributed performance databases can be correlated to provide a complete picture of the managed system.

Future work should also be done to address the performance costs of management application queries on the database; and to introduce features that better ensure no monitoring data is lost and that the performance database is always consistent.

References

[1] ARM 2.0 SDK User's Guide

http://www.tivoli.com/o_download/html/armguide.html

[2] Managing the Enterprise with the Application Response Measurement API(ARM). Denise Morris, Resource & Performance Management, Network & System Management Division. Hewlett-Packard Company.

<http://www2.hp.com/openview/rpm/papers/armwp.html>

[3] ARM-Enabling Your MeasureWare Agent (Addendum to the Application Response Measurement API Guide) August 1996.

<http://www.hp.com/openview/rpm/arm/docs/mwaguide.htm>

[4] Tivoli and Application Management Technical Papers, Tivoli Systems, 1998.

http://www.tivoli.com/o_products/html/body_map_wp.html

[5] Performance Management for Distributed Systems

<http://www.bmc.com/products/articles/g55wp00a.html>

[6] ARM Working Group

<http://www.cmg.org/regions/cmgarmw/index.html>

[7] The MANDAS Project MANagement of Distributed Applications and Systems

<http://www.csd.uwo.ca/research/mandas/>

[8] R. Friedrich and J. Rolia, "Applying Performance Engineering to a Distributed Application Monitoring System," Editors A. Schill, C. Mittasch, O. Spaniol, and C. Popien, Distributed Platforms, Chapman and Hall Publishers, 1996, page 258-271.

- [9] The Performance impact of workload characterization for distributed applications using ARM, F.EiRayes, J.Rolia, and R.Friedrich, ", to appear in the proceedings of the Computer Measurement Group (CMG) '98, December, 1998, Anaheim, California, USA, pages 821-830.
- [10] D. Krishnamurthy and J. Rolia, "The Internet vs. Electronic Commerce Servers, When Will Server Performance Matter?", To appear in the proceedings of CASCON'98 , November 30 - Dec 2, 1998, Toronto, Canada, pages 246-258.
- [11] J. Rolia and R. Friedrich, "Quality of Service Mangement for Federated Applications," Appears in the Proceedings of the 4th International iFIP Workshop on Quality of Service (IWQOS '96), Paris, France, March 6-8 1996, pages 259-270.
- [12] M. A. Bauer, et al.. J. Rolia, "Services Supporting the Management of Distributed Application Systems." IBM Systems Journal, 1997, Volume 36, Number 4, pages 508-526.
- [13] M. Qin, R. Lee, A. El Rayess, V. Vetland, and J. Rolia, "Automatic Generation of Performance Models for Distributed Application Systems", CD-ROM for CASCON'96, Toronto, November 6-12, 1996.
- [14] J.A. Rolia and K.C. Sevcik, "The Method of Layers," IEEE Transactions on Software Engineering, Vol. 21, No. 8, pp. 689-700, August 1995.
- [15] Application Response Measurement Standard Moves Forward with API Enhancements and Vendor Implementations.
<http://www.hp.com/csopress/97june30c.html>
- [16] Application Management Specification
[http:// www.tivoli.com/o_products/html/body_ams_spec.html](http://www.tivoli.com/o_products/html/body_ams_spec.html)
- [17] Microsoft ODBC [http:// www.microsoft.com/data/odbc](http://www.microsoft.com/data/odbc)

[18] The JDBCTM Database Access API

<http://java.sun.com:80/products/jdbc/index.html>

[19] Visual C++

<http://msdn.microsoft.com/visualc/default.asp>

[20] Embedded SQL Programming Guide

[http://www.software.ibm.com/cgi-](http://www.software.ibm.com/cgi-bin/db2www/library/document.d2w/report?search_type=SIMPLE&uid=UNKNOWN&pwd=&r_host=134.117.57.44&last_page=pubs.d2w&fn=db2a002.htm)

[bin/db2www/library/document.d2w/report?search_type=SIMPLE&uid=UNKNOWN&pwd=&r_host=134.117.57.44&last_page=pubs.d2w&fn=db2a002.htm](http://www.software.ibm.com/cgi-bin/db2www/library/document.d2w/report?search_type=SIMPLE&uid=UNKNOWN&pwd=&r_host=134.117.57.44&last_page=pubs.d2w&fn=db2a002.htm)

[21] M. A. Bauer, P. F. Finnigan, J. W. Hong, J. A. Rolia, T. J. Teorey, and G. A. Winters, "Reference Architecture for Distributed Systems Management", IBM Systems Journal Vol. 33, No. 3, 1994, pages 426-444.

[22] E. Lazowska, J. Zahoran, G. Graham, and K. Sevcik, Quantitative System Performance: Computer System Analysis Using Queuing Network Models, Prentice Hall, Inc., Englewood Cliffs, NJ, 1984.

[23] G. Franks, A. Hubbard, S. Majumdar, J.E. Neilson, D.C. Petriu, J. Rolia, C.M. Woodside, "A Toolset for Performance Engineering and Software Design of Client-Server Systems", Performance Evaluation (special issue on Performance Tools), Vol. 24, No.1-2, pp.117-135, November 1995.

[24] Microsoft DCOM

<http://www.microsoft.com/com/dcom.asp>

Appendix Aggregation Levels Supported by Carleton University ARM 2.0 Prototype

No Instrumentation

Full Trace

End to End

By Transaction, no correlation

By Transaction, with correlation

By Transaction, no correlation , by Request Type

By Transaction, no correlation, By Business Function Type

By Transaction, with correlation, By Business Function Type

By Process, no Correlation

By Process, with Correlation

By Process, with correlation by Request Type

By Process, no correlation by Request Type

By Process, no correlation by Business Function Type

By Process, with correlation by Business Function Type

By Object Type, no Correlation

By Object Type, with Correlation

By Object Type with Correlation by Request Type

By Object Type, no correlation by Request Type

By Object Type, no correlation by Business Function Type

By Object Type, with correlation by Business Function Type

By Object, no Correlation

By Object, with Correlation

By Object with Correlation by Request Type

By Object, no correlation by Request Type

By Object, no correlation by Business Function Type

By Object, with correlation by Business Function Type

By Method, no Correlation

By Method, with Correlation

By Method, with correlation by Request Type

By Method, no correlation by Request Type

By Method, no correlation by Business Function Type

By Method, with correlation by Business Function Type

IMAGE EVALUATION TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved