Evaluating the Costs of Management: A Distributed Applications Management Testbed *

Michael J. Katchabaw, Stephen L. Howard,
Andrew D. Marshall, and Michael A. Bauer
Department of Computer Science
The University of Western Ontario

Abstract

In today's distributed computing environments, users are making increasing demands on the systems, networks, and applications they use. Users are coming to expect performance, fault tolerance, configuration, security, and accounting services comparable to those on traditional centralized systems. Meeting these expectations implies the need for management. While some successes have been realized in the areas of system and network management, only preliminary investigations have been conducted on management of distributed applications.

At this stage, little is known of the costs associated with this type of management. To assess these costs, it is necessary to collect data about the performance and operation of distributed applications and their management systems. Furthermore, it is important to collect this data under varying constraints, configurations, and management assumptions. In order to do this, we have developed a distributed applications management testbed. We present our initial requirements for the testbed, its design, and a prototype implementation. We describe some preliminary results and analysis obtained through its use. Finally, we identify its limits based on our experience to date and outline our plans to evolve the testbed to accommodate further experimentation.

*This research work is supported by the IBM Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada. The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, 2G/894, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7.

1 Introduction

With the increased complexity of distributed systems comes a greater need for automated tools to assist in the management of networks, system resources, and application processes. While significant experience has been gained in the areas of network and system management, little has been accomplished at the application level.

The focus of much of our ongoing research has been in the area of distributed applications management [2, 3]. We have developed and continue to use applications management system prototypes based on both OSI and SNMP protocols. We have become interested in the practical issues of applications management that may impair its acceptance; issues such as the cost of management (for example, the performance overhead) and how management can be done efficiently. This paper describes a project motivated by this interest.

Overhead associated with management of distributed applications is unavoidable. Instrumentation code added to each of the application components in support of runtime monitoring and control will affect the application's performance directly because of the extra instructions in the execution path. In addition, the management system consists of communicating processes that compete for the same memory, CPU, and I/O resources that are used by the application.

A basic question we are trying to answer is: what is the price we pay in overall application performance for this management and what steps can be taken to minimize it? To

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

address this problem, it is necessary to monitor the behavior of management systems and their managed applications under various workloads and configurations. Work has begun to identify these costs [1].

Once the costs are identified and understood, we can begin minimizing them. There are many parameters in the design and deployment of the management system that potentially influence the cost of management; for example, controlling component deployment, toggling functions on and off, adjusting threshold values to control event generation conditions and rates, configuring event-filtering mechanisms to reduce the traffic flow, controlling the quantity and nature of information in event reports, and so on.

To address this issue, we have developed a distributed applications management testbed that is independent of the management system and its managed applications. The testbed is a tool that enables experiments to observe and analyze the performance of both management system components (such as managers and agents) and managed application components (such as instrumented processes). The results of such experiments provide insight into how management can be performed most effectively.

In this paper, we present a set of requirements for the distributed applications management testbed and describe a testbed design that meets these requirements. We discuss the testbed prototype and some preliminary results obtained in an OSI management environment. The paper concludes with a project status report, identifying the limits of our experience to date and our plans for future work.

2 Requirements

The goal was to develop a management system testbed that can be used to assess the impact of management on system resources and on managed applications. We identified the following key requirements for the testbed:

- The testbed should allow an experiment to be controlled. This includes the ability to start and stop an experiment, to launch and terminate components of the management system, and to launch and terminate components of the managed application.
- The testbed should be flexible, allowing reconfiguration for a variety of experiments. It must maintain configuration information about the management system and managed applications.
- The testbed should provide a way to collect data about the behavior and performance of the management system and managed applications.
- Simple statistical analysis of collected data should be provided by the testbed.
- Higher-level applications should be able to use the testbed through a well-defined application programming interface (API).
- The testbed itself should have minimal interference with the performance of the management system and the managed environment. The testbed must operate outside the execution path of the system being tested; it must not require intrusive instrumentation or code modification. All information must be gathered at arm's length through the use of operating system services and existing observable behavior.
- To as great a degree as possible, the testbed design should be independent of the system being tested. It should not make assumptions about the type of management system or the managed environment.

3 Testbed Architecture

We developed a generic architecture to meet our key requirements. In this section, we describe the key components of the architecture and an interface for accessing testbed services.

3.1 Component Overview

The distributed application management testbed architecture shown in Figure 1 is the basis for our research and prototype development. This architecture consists of the testbed

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

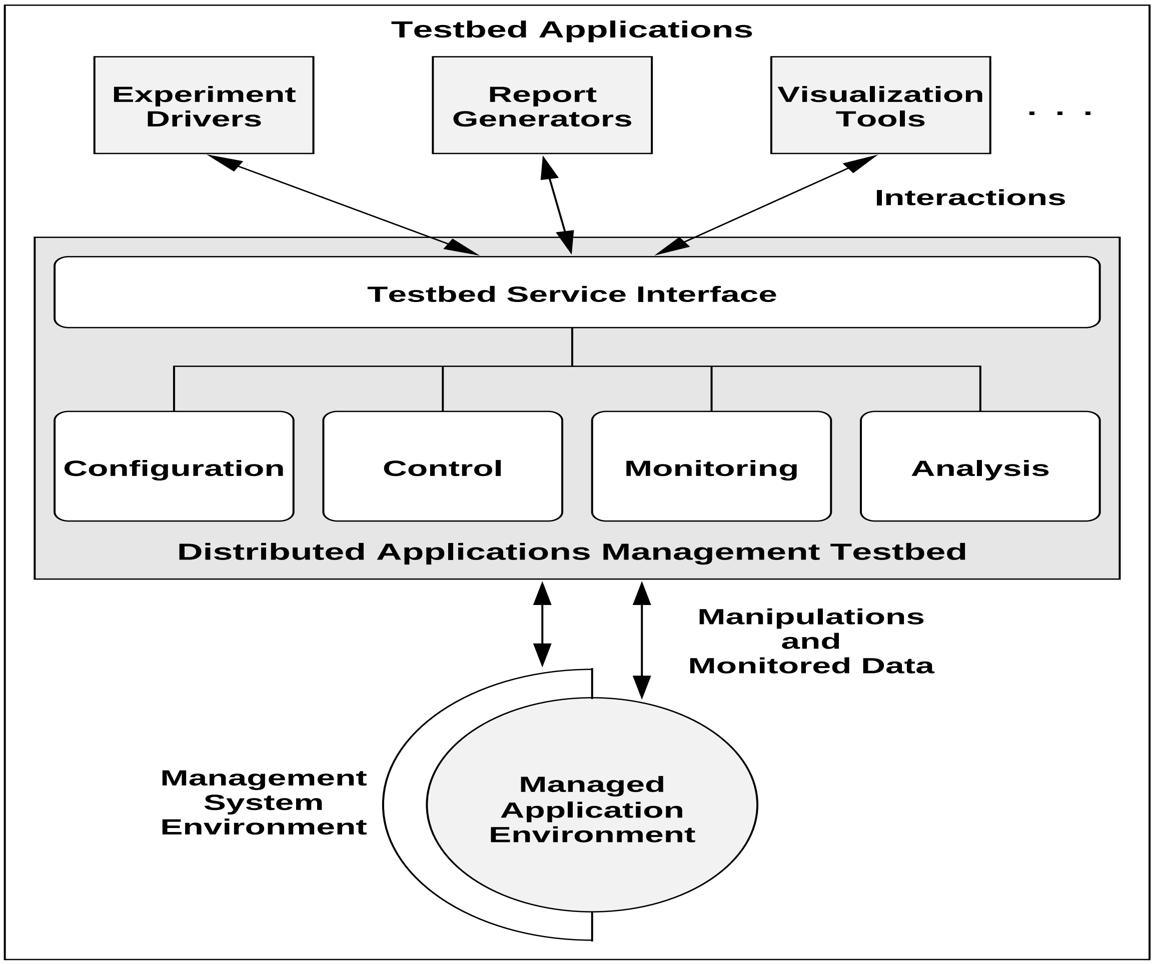

Figure 1: Distributed Applications Management Testbed Architecture

itself, managed applications and management systems whose performance is to be analyzed, and testbed applications that make use of the testbed. These components are described below.

3.1.1 Testbed Components

The testbed consists of a set of core components.

Testbed Service Interface: The services provided by the testbed are accessed by testbed applications through the testbed service interface. This interface coordinates activities within the testbed by routing requests, replies, and reports to the appropriate components. The services provided through this interface are discussed in detail in Section 3.2.

Configuration: The configuration component is responsible for maintaining information on the configuration of the management system and managed applications. Areas of responsibility include:

- Process deployment — which processes are executing on which hosts.
- Process relationships — which application clients report to which application servers, which application processes report to which management components, and which management components report to which other management components.

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

- Workload population — what kind of interactions occur between the component processes, what information is passed in each interaction, and how often these interactions occur.

Control: The control component provides a basic service for controlling experiments. It uses deployment and relationship information from the configuration component and provides operations to:

- Start an experiment and allocate the resources required, based on the defined configuration.
- Finish an experiment and deallocate experiment resources.
- Launch managed application and management system processes according to their configuration.
- Terminate managed application and management system component processes.

Monitoring: The monitoring component collects the following information about the management system and its managed applications:

- Operational status information about processes.
- Timing information, including elapsed times, CPU times, and so on.
- Resource utilization, including CPU utilization, memory utilization, disk utilization, and network utilization.

Analysis: Using configuration information and data collected from experiments, the analysis component can compute:

- Basic measures, such as totals, maximum values, minimum values, means, medians, and modes.
- Variability measures, such as standard deviation, sample variance, and skewness.

These functions can be applied to individual processes or groups of processes. More sophisticated analyses, such as regression analysis, can be performed by higher level testbed applications.

3.1.2 Managed Application and Management System Environments

The managed application and management system environments shown in Figure 1 consist of the hardware and software resources and services used by component processes during execution. This includes computing nodes, networks, operating system services, and distributed processing services, such as name services, repository services, and transaction services.

By interacting with their environments, rather than the component processes themselves, the testbed can configure, control, and monitor the processes with minimal impact on their operation. For example, resource utilization statistics can be obtained through operating system services instead of querying individual processes, which would add unwanted load to the processes.

3.1.3 Testbed Applications

Users access the services of the testbed through external testbed applications. These applications invoke the testbed operations through the testbed's service interface. Examples of such applications, are shown in Figure 1, and explained below.

Experiment Drivers: Carrying out an experiment using the testbed involves selecting the desired configuration, executing the experiments, and analyzing the collected results. Even the simplest performance analysis requires several experiments with different configurations to be repeated over extended periods of time. Repeating the experiments manually would be extremely tedious. An experiment driver facilitates and automates this process.

Report Generators: Report generation applications can be used to aggregate the data

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

from experiments, perform high-level analysis, and present the results.

Visualization Tools: Experiments can be configured and controlled more intuitively by manipulating visual abstractions of the components involved. Data can be presented more concisely and understandably in graphical forms.

3.2 Testbed Services

To support the development of testbed applications, we defined a consolidated interface for accessing the four testbed services. This section describes this interface and the operations supported by each service.

3.2.1 Configuration Service

The configuration service provides operations to deploy management system and managed application components (i.e. processes), set relationships between these components, and set workloads for these components. These operations are:

deployComponent: Deploys an application component of a certain type on a specified host. This operation does not start the component, but configures it to be started when the `launchComponents` operation is invoked (see Section 3.2.2). When a component is deployed, it is assigned a unique identifier used for referencing the component until it is removed using `unDeployComponent`.

```
deployComponent(
  [in:]  componentType_t type,
  [in:]  host_t host,
  [out:] componentId_t id,
  [out:] statusCode_t status)
```

reDeployComponent: Changes the host on which a component runs. The component's identifier and the new host are specified.

```
reDeployComponent(
  [in:]  componentId_t id,
  [in:]  host_t host,
  [out:] statusCode_t status)
```

unDeployComponent: Removes a given component from the set of deployed components. Once this is done, only the `collectComponentData` operation can be used to reference the undeployed component. All relationships previously defined for this component (by `setComponentRelationship`) are removed.

```
unDeployComponent(
  [in:]  componentId_t id,
  [out:] statusCode_t status)
```

setComponentRelationship: Establishes and configures a relationship between two previously deployed components. This relationship can be: client-to-server, peer-to-peer, application component-to-management component, or management component-to-management component. A relationship remains valid until either `removeComponentRelationship` or `unDeployComponent` is invoked.

```
setComponentRelationship(
  [in:]  relType_t relationship,
  [in:]  componentId_t component_1,
  [in:]  componentId_t component_2,
  [out:] statusCode_t status)
```

removeComponentRelationship: Removes a relationship previously established by `setComponentRelationship`.

```
removeComponentRelationship(
  [in:]  relType_t relationship,
  [in:]  componentId_t component_1,
  [in:]  componentId_t component_2,
  [out:] statusCode_t status)
```

populateComponentWorkload: Sets a workload (a sequence of service requests) for the given component. This workload is populated according to a template provided in the operation invocation. The populated workload is used until either `populateComponentWorkload` is invoked again, or the component is removed using `unDeployComponent`.

```
populateComponentWorkload(
```


Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

```
  [in:]  componentId_t component,
  [in:]  workloadTemplate_t template,
  [out:] statusCode_t status)
```

3.2.2 Control Services

Control operations are provided by the control service to start and finish experiments and to start and terminate management system and managed application component processes.

startExperiment: Notifies the testbed that a new experiment is starting. The appropriate resources are allocated and mutual exclusion to the testbed is provided, since two concurrent experiments could have undesirable effects on one another. An identifier for the experiment is returned as a result.

```
startExperiment(
  [out:] experimentId_t id,
  [out:] statusCode_t status)
```

finishExperiment: Notifies the testbed that an experiment has finished and that previously allocated resources can be released.

```
finishExperiment(
  [out:] statusCode_t status)
```

launchComponents: Starts a set of components using the configuration specified previously using configuration operations. Components are referenced through their component identifiers.

```
launchComponents(
  [in:]  componentSet_t component_list,
  [out:] statusCode_t status)
```

terminateComponents: Terminates a given set of components that had been started using `launchComponents`. Components are referenced through their component identifiers.

```
terminateComponents(
  [in:]  componentSet_t component_list,
  [out:] statusCode_t status)
```

3.2.3 Monitoring Service

Monitoring operations collect data on management system and managed application components, both during and after experiments. A single operation is defined.

collectComponentData: Collects the specified monitored data on the given set of components. If a component is executing, the returned data reflects the current state of the component. Otherwise, the returned data reflects the state of the component at termination.

```
collectComponentData(
  [in:]  componentSet_t component_list,
  [in:]  dataSpecifier_t specifier,
  [out:] data_t data,
  [out:] statusCode_t status)
```

3.2.4 Analysis Service

The analysis service analyzes data collected from management system and managed application components. A single operation is defined.

performAnalyticalFunction:
Performs the specified analytical function on the data provided. Sample functions compute totals, maximum values, minimum values, medians, modes, means, and other simple statistical functions.

```
performAnalyticalFunction(
  [in:]  functionSpec_t analytical_fn,
  [in:]  data_t data1, data2, data3, ...
  [out:] data_t result,
  [out:] statusCode_t status)
```

4 Performance Metrics

To determine the cost of managing, we compare application performance with and without management. We use a set of performance metrics, which we classify as responsiveness metrics, productivity metrics, utilization metrics, and error or failure metrics [8].

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

4.1 Responsiveness Metrics

The responsiveness of a distributed application is the time between an operation invocation and the completion of that operation. By comparing the responsiveness of applications with and without management, we can effectively compute the impact of management. Three responsiveness metrics are used:

- The response time for a single application request.
- The start-to-completion time for a workload (that is, a "standard" batch of application requests).
- Process initialization time, measured from the time the process starts execution until it is ready to perform useful work.

4.2 Productivity Metrics

Productivity (or throughput) metrics provide measures of the amount of work completed per unit of time. These metrics relate both to the rate at which application operations are performed and to the amount of data passed between application components within a given period of time. The following productivity metrics are defined:

- For a server process, the number of application requests processed per second.
- The number of batches of application requests processed per second.
- The amount of data transferred between processes per second.

4.3 Utilization Metrics

Utilization metrics provide measures of system resource consumption and, as such, are only indirect measures of an application's performance. Nevertheless, it is useful to obtain utilization data on a range of system resources to give a better understanding of the true cost of management. The following utilization metrics are identified:

- CPU utilization
- Memory utilization
- Disk utilization
- I/O utilization
- Network utilization

Measurements of these resources can be collected either for individual services, or over the lifespan of a process. Depending on the analysis required, both measurements can be useful.

4.4 Error or Failure Metrics

An error is defined as a situation in which an application operation cannot be completed correctly. A failure is defined as a situation in which an application operation cannot be completed at all. While errors and failures tend to be application-specific, they must be understood in order to perform analysis correctly. For each application-specific class of error or failure, we can measure metrics such as:

- The probability of an error or failure occurring on each operation invocation.
- The mean time between errors or failures.
- The "down-time" during which the managed application is unavailable due to failure.
- The severity of the error or failure.

5 Parameters Affecting Performance

The performance metrics presented in the previous section identified what we want to know about performance. These are the observable outputs of the experiments. We now focus on the parameters that influence performance. These parameters correspond to the inputs to the experiments.

Identifying the performance parameters is critical to the planning, design, and execution of experiments for determining the costs of management. These parameters generally fall into one of two categories—system parameters or workload parameters.

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

5.1 System Parameters

System parameters are those related to hardware, the operating system, middleware services, and so on. They are outside the application and its management system, and from an experimentation point of view, they are considered to be static. These parameters include:

System Level Parameters:

- Number of systems involved
- Architecture of each system
- Speed of each system's processor(s)
- Speed of each system's coprocessors
- Amount and speed of system memory
- Speed of disks on each system
- Use of hardware caches on each system

Network Level Parameters:

- Speed of the network
- Network reliability
- Network protocols use.
- Network topology/configuration (subnets, routers, bridges, etc.)

Operating System and Middleware Parameters:

- Operating system overhead
- Effects of software caches
- Middleware overhead
- Nature/speed of other services (file, time, name, etc.)

5.2 Workload Parameters

Workload parameters represent the variable aspects in the experiments; factors that can be controlled or can be varied from one experiment to the next. We include characteristics of the application and its management system as well as aspects of system state that are affected by the workload imposed by the application and its management system. These parameters include:

Load Parameters:

- System load (number of users, number/type of processes, etc.)
- Network congestion
- Resource utilization (memory, disk, network, etc.)

Management System Parameters:

- Presence/absence of management
- Type of management system
- Management system configuration
- Management functions and services
- Underlying infrastructure and communication
- Number of processes

Managed Application Parameters:

- Application type
- Process types (client, server, peer)
- Resource requirements
- Communication methods
- Configuration (number of processes, locations, etc.)
- Request types (including rates and distribution)
- Instrumentation type

Management Parameters:

- Management services used
- Number, rate and distribution of service requests
- Whether management uses polling (*pull* style) or event-driven (*push* style) communication
- Request type (information access or application control)

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

6 Prototype Implementation

In this section, we describe a prototype management system testbed based on the generic architecture presented in Section 3. This prototype is currently being used to conduct experiments to investigate the costs of distributed applications management.

6.1 Target Environment

The prototype was developed in a laboratory setting with a dozen hosts of various architectures, including IBM RS/6000, MIPS RC2030, Sun SPARCstation, Sun 3/60, and Sequent Symmetry, spread over two subnets. This setting provides diversity in processors, memory, device configuration, flavors of the UNIX operating system (AIX, RISC/os, SunOS, and DYNIX/ptx), communication infrastructure and middleware (sockets, ONC RPC, DCE, and CORBA). Therefore, a large number of the system parameters identified in Section 5.1 as fixed performance factors are present.

6.2 Implementation

To meet the flexibility and portability requirements demanded by the heterogeneity of the environment, we implemented the testbed as a structured collection of UNIX shell scripts. Since the testbed operates outside the management system and its managed applications, most of its interactions are with the operating system. The ease of access to system services and utilities, and the fact that the performance of the testbed is of little concern, made shell scripts an obvious choice.

The testbed architecture was preserved by structuring each service as a collection of scripts providing the required operations. The interface to the testbed is defined by script names and their arguments. Testbed applications that invoke these scripts can be written in a variety of languages (for example, shell scripts, C, Tcl, Perl, or Java).

The configuration service stores configuration information as a set of flat files. This information is manipulated when the appropriate configuration operations (given in Section 3.2.1) are invoked. Workloads populated by the `populateComponentWorkload` configuration operation are stored as either data files or scripts, depending on the type of component for which a workload is generated.

Control service operations access configuration files and workload scripts. For example, the `startExperiment` operation needs to know which processes to start, where to start them, and what workloads to assign.

The `collectComponentData` monitoring service operation collects specified data querying local and remote systems using UNIX system utilities. This data is stored in monitored data files.

Finally, analysis service operations are carried out by scripts that process monitored data.

6.3 Management System and Managed Applications

For initial testbed validation and experimentation, we adapted a management system and managed application from our previous work [9]. The management system, based on the OSI Management Framework [4, 7], had been enabled to manage DCE applications. Several DCE applications have been developed and instrumented for management by this system.

6.3.1 Management System

In the OSI Management Framework (see Figure 2), real resources (application processes) are modeled by *managed objects*. *Managers* (also called *management applications*), which exist to enforce management policy on behalf of human managers, view the managed system as a collection of these managed objects and are unaware of the real resources hidden behind the layer of abstraction. There are many advantages to this approach. For example, component granularity can be defined (one managed object might model a collection of real resources as a single entity).

A *management agent* holds responsibility for a particular set of managed objects. The agent processes managers' requests and delivers event reports to managers. The agent interacts with the real resources to update managed objects' states, carry out controlling operations, accept

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

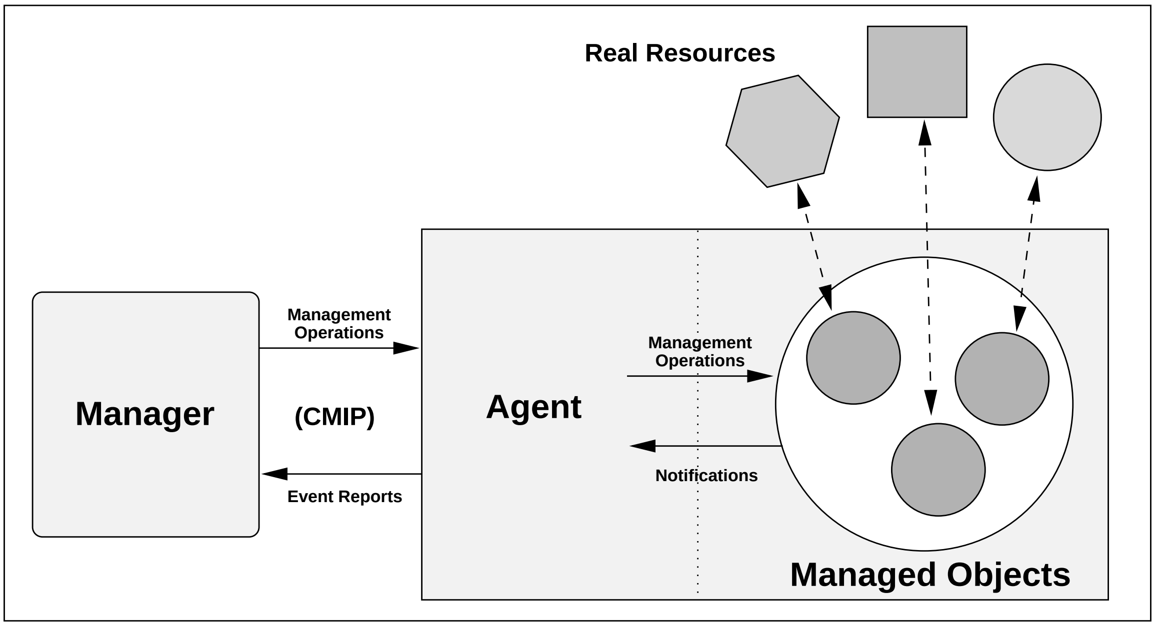

Figure 2: OSI Management Framework

event notifications, and so on, to fulfil the management obligations imposed on it.

Communication between the manager and agent is defined by the OSI Common Management Information Service (CMIS) [5] and the Common Management Information Protocol (CMIP) [6]. Communication between the agent and the real resources (application processes) falls outside the OSI standards; in our experiments, we used DCE Remote Procedure Calls (RPC) [10].

6.3.2 Managed Applications

In our previous work [9], several DCE applications were developed and instrumented for management in the OSI management environment. These applications follow a client-server model with varying degrees of complexity. The most interesting, from the testbed validation point of view, is a distributed time-table scheduling application, which uses a DB/2 repository for storing schedule information. Although conceptually simple, this application proved to be of adequate complexity to facilitate our performance experiments.

In the experiments, the testbed was used to control the deployment and workload of the application and to control its operation. To enhance the usefulness of the experiments, Motif-based clients were replaced with script-driven clients so we could create and apply artificial application workload. While the testbed can provide useful results on naturally-occurring workloads, we felt that using controllable clients would yield more accurate results and a stronger basis for validation.

6.4 Testbed Applications

For initial testing and experimentation, a basic experiment driver and a simple report generator were developed. Work has also begun on a visualization application. The experiment driver uses scripts to guide the execution of several experiments in sequence. To complete each phase of an experiment, the driver makes use of the appropriate service scripts provided by the testbed. Each experiment consists of eight phases:

1. Starting the experiment and gaining sole access to the testbed.

2. Configuring the testbed, management system, and managed application appropri-

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

ately, depending on the goals of the experiment.

3. Starting the management system and managed application components used in the experiment.

4. Monitoring the started components and collecting the required information.

5. Terminating the components and collecting the state information needed.

6. Analyzing the collected data as necessary.

7. Finishing the experiment.

8. Generating tables summarizing the results of the experiment.

Using the experiment driver and report generator, several experiments have been performed to date. The results of these are presented in the next section.

7 Experience

An initial set of experiments was conducted to verify the operation of the system and identify areas for improvement. Based on the success of these tests, more rigorous and complex experiments were planned and executed. The results of these experiments proved to be quite interesting, demonstrating the utility and flexibility of the testbed.

In our experimentation, we set out to determine the influence of a set of parameters on application performance (as measured by the metrics described in Section 4). The parameters selected included: the presence of management, whether management was push-driven or pull-driven, the frequency and distribution of management messages (either requests or notifications), and the frequency and distribution of application requests. We carried out several experiments, varying the parameter values in each experiment.

To assess the significance of the results, we used linear regression techniques to determine the degree to which the variation in computed metrics could be attributed to each parameter—parameters causing larger variation being considered stronger influences on application performance. The following results were obtained from our experiments [1]:

- Management causes a significant increase in an application's use of system resources such as CPU, memory, and network bandwidth.

  In experiments which compared the presence of management to several other application parameters, management was found to account for 79.6% to 99.9% of the variation in resource utilization (depending on the resource). On average, each application request required 42.4% (0.05 seconds) more CPU time and 31.1% more socket communication (20 messages). The application was not computationally intensive, so the ratio of CPU use with management enabled to CPU use with management disabled is artificially high, and should be considered a worst-case measure. A significant increase in the number of context switches and page faults was also observed over the life of the managed processes. The maximum memory required by the process was increased by an average of 7.4% (70 pages), while the minimum memory requirement was increased an average of 16.7% (150 pages).

- Management does not cause a significant increase in the response time of application requests.

  Management accounts for only 1% of the variation in the responsiveness of the application; the remaining variation was explained by changes to other application workload parameters. On average, this means a 3.4% (0.51 seconds) difference per request. On the other hand, some application workload variations (such as the mean delay between application requests) cause a difference of 31.9% (4.2 seconds).

[1] We caution the reader that these results were obtained in a particular environment and with a particular management system. While we anticipate the findings to be representative, we cannot make a firm claim to generality.

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

From these results, we can conclude that, given sufficient system resources, management can be practical in light of its low impact on application responsiveness. However, if resources are scarce, or a heavy system load is causing resource contention, management can significantly influence application performance.

With further experimentation, we observed a number of secondary results that gave additional insight into what factors contribute to the cost of management:

- The time interval between management messages accounts for the largest variation in resource utilization (68.5% to 92.2%, depending on the resource), compared to other management factors. This is particularly interesting, since the number of management messages batched together (sent in rapid succession) appeared to be much less significant. In other words, it is more efficient, from a resource utilization perspective, to generate management messages in batches with longer intervals between them, as opposed to evenly distributing the messages across time.
- On the contrary, it is more efficient, from a responsiveness perspective to evenly distribute messages across time.
- Equally significant with respect to responsiveness was whether management was push-driven or pull-driven. On average, application requests under push-driven management were 4.62% (0.70 seconds) faster.
- Push-driven management was also superior to pull-driven management in terms of resource utilization.

From these secondary results, we can conclude that from a resource utilization point of view, it is better to batch management requests rather than distribute them evenly across time, whereas the converse is true for responsiveness. We also conclude that push-driven event communication is generally superior.

Most experiments carried out thus far have focussed on workload parameters. These experiments have been conducted using only RS/6000 machines (due to the application's dependence on DCE and DB/2, which we have installed only on those machines). More work needs to be done to measure the effects of different system parameters.

8 Concluding Remarks

The work we describe here is part of an ongoing structured attack on the distributed applications management problem. There are many complex issues yet to be resolved. We believe that identifying and understanding the costs associated with management makes it easier to address these issues and contributes to the acceptance of managing distributed applications.

The research detailed in this paper focusses on the use of a distributed applications management testbed to investigate the costs of management. We first presented a core set of requirements for the testbed, and then we described a design satisfying these requirements. Using a prototype testbed based on this design, we have been able to carry out a number of experiments and are beginning to realize useful results.

We plan to continue experimentation to validate these preliminary results and to generate new results. We need to look at different management systems and apply these to a variety of application types, making full use of the heterogeneity of the testbed environment. More advanced testbed applications are needed for better reporting and visualization. We also need to investigate the potential for using testbed results in the simulation and modeling of the management of distributed applications.

Through this work, we hope to gain insight into both the cost of management, and the factors contributing to it. With this experience, we can pursue optimization of distributed applications management systems in an effort to minimize their costs.

Acknowledgements

The authors would like to thank the students of the Computer Science 612 class at the University of Western Ontario for their assistance with initial testbed experimentation.

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.

About the Authors

Michael J. Katchabaw is a Ph.D. student in the Department of Computer Science at the University of Western Ontario. His research interests include distributed computing, network management, and distributed multimedia. Mr. Katchabaw can be reached at katchab@csd.uwo.ca.

Stephen L. Howard is a Ph.D. student in the Department of Computer Science at the University of Western Ontario. His research interests include distributed computing, operating systems, and object-oriented management models. Mr. Howard can be reached at showard@csd.uwo.ca.

Andrew D. Marshall is a Research Associate with the MANDAS Project and a Ph.D. student in the Department of Computer Science at the University of Western Ontario. His research interests include distributed computing, applications management, and software reengineering. Mr. Marshall can be reached at flash@csd.uwo.ca.

Michael A. Bauer is a Professor in the Department of Computer Science and Senior Director of Information Technology Services at the University of Western Ontario. Dr. Bauer received a Ph.D. in computer science from the University of Toronto. His research interests include distributed computing, distributed directories, and software engineering. Dr. Bauer can be reached at bauer@csd.uwo.ca.

References

[1] H. Abdu, H. Lutfiyya, and M. Bauer. Investigating Monitoring Configurations. *Proceedings of the 1996 ACM Symposium on Applied Computing*, pages 366–373, February 1996.

[2] J. W. Hong, M. J. Katchabaw, M. A. Bauer, and H. Lutfiyya. *Distributed Applications Management Using the OSI Management Framework.* Technical Report #448, Dept. of Computer Science, University of Western Ontario, London, Canada, January 1995.

[3] J. W. Hong, M. J. Katchabaw, M. A. Bauer, and H. Lutfiyya. Modeling and Management of Distributed Applications and Services Using the OSI Management Framework. *Proceedings of the International Conference on Computer Communication*, Seoul, Korea, August 1995.

[4] ISO. *Information Processing Systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management Framework.* International Organization for Standardization, International Standard 7498-4, 1991.

[5] ISO. *Information Processing Systems - Open Systems Interconnection - Common Management Information Service Definition.* International Organization for Standardization, International Standard 9595, 1991.

[6] ISO. *Information Processing Systems - Open Systems Interconnection - Common Management Information Protocol Specification.* International Organization for Standardization, International Standard 9596-1, 1991.

[7] ISO. *Information Processing Systems - Open Systems Interconnection - Systems Management Overview.* International Organization for Standardization, International Standard 10040, 1991.

[8] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley Professional Computing, 1991.

[9] M. J. Katchabaw, S. L. Howard, H. L. Lutfiyya, and M. A. Bauer. Efficient Management Data Acquisition and Run-time Control of DCE Applications Using the OSI Management Framework. *Proceedings of the Second International IEEE Workshop on Systems Management*, Toronto, Canada, June 1996.

[10] John Shirley. *Guide to Writing DCE Applications.* O'Reilly, July 1992.

Proceedings of the 1996 CAS Conference (CASCON '96),
Toronto, Canada, November 12-14, 1996, pages 29-41.