Quality-of-Service Measurements with Model-Based Management for Networked Applications

Joe Martinka, Jim Pruyne, Mudita Jain
Software Technology Laboratory
HPL-97-167 (R.1)
September, 1998

E-mail: [martinka,pruyne,jainm]@hpl.hp.com

distributed application management, manageability, model-based reasoning, scalable measurement, intelligent agents, Java applet instrumentation, quality-of-service

Distributed applications are evolving towards compositions of modular software components with user interfaces based on web browsers. Each of these components provides well-defined services that interact with other components via network. The increase in the complexity of distribution makes it more difficult to manage the end-to-end Quality-of-Service (QoS). The challenge derives in part since different management scopes of network and computing domains need to interact. We address two needs of a management system deployed to diagnose QoS degradation. First, to measure the performance of applications, it needs a low-overhead, scalable system for measuring software components. Second, the performance management system must monitor selected measurements, diagnose QoS degradation, adapt to the environment and integrate with network management systems. We extend our Distributed Measurement System (DMS) into browser-based Java applets to deliver low-overhead and pertinent performance information to a management system. A model-based reasoning engine equipped with "generic" application models uses these measurements to diagnose QoS trends. These models incorporate the notion of composite transactions and the organization of distributed components. We demonstrate QoS monitoring using this architecture on a typical, component-based distributed application deployed in a wide area network.

Quality-of-Service Measurements with Model-based Management for Networked Applications

Joe Martinka, Jim Pruyne, Mudita Jain
{martinka, pruyne, jainm}@hpl.hp.com
Hewlett-Packard Laboratories, Palo Alto, California

ABSTRACT: *Distributed applications are evolving towards compositions of modular software components with user interfaces based on web browsers. Each of these components provides well-defined services that interact with other components via network. The increase in the complexity of distribution makes it more difficult to manage the end-to-end Quality-of-Service (QoS). The challenge derives in part since different management scopes of network and computing domains need to interact. We address two needs of a management system deployed to diagnose QoS degradation. First, to measure the performance of applications, it needs a low-overhead, scalable system for measuring software components. Second, the performance management system must monitor selected measurements, diagnose QoS degradation, adapt to the environment and integrate with network management systems. We extend our Distributed Measurement System (DMS) into browser-based Java applets to deliver low-overhead and pertinent performance information to a management system. A model-based reasoning engine equipped with "generic" application models uses these measurements to diagnose QoS trends. These models incorporate the notion of composite transactions and the organization of distributed components. We demonstrate QoS monitoring using this architecture on a typical, component-based distributed application deployed in a wide area network.*

KEYWORDS: distributed application management, manageability, model-based reasoning, scalable measurement, intelligent agents, Java applet instrumentation, quality of service.

1 Introduction

Distributed applications are increasingly composed of modular, "off-the-shelf" software components and custom code. Their user interfaces use ubiquitous web browsers executing Java applets or HTML forms. Thus, the availability of the World Wide Web has swelled the population served by these applications. Application performance may be influenced by several unpredictable factors such as the network performance, temporally varying workloads, and components of other applications contending for resources. These interactions are difficult to characterize because of their dynamic nature. These factors make it impossible to anticipate all performance problems during applications design and test. Therefore, performance bottlenecks are inevitable during application operation.

These applications are increasingly subject to *Service Level Agreements* (SLAs), with QoS expectations of high availability and good response times. Thus, operational applications must be monitored for potential QoS violations and performance bottlenecks, with effective actions being taken to alleviate or prevent these problems. Current management systems that monitor thresholds and trigger alarms rely on correct interpretation by the operator to determine causal interactions. This approach does not scale as the number of thresholds and alarms increase. For a scaleable solution, we require a management system that can monitor, diagnose and reconfigure application components to ensure user-level QoS goals are maintained. The management system must be proactive and coordinate with existing network management systems. Emerging problems are corrected before QoS failures occur. The use of knowledge-based systems is ideal for management of these distributed applications[14].

This paper describes and demonstrates a prototype of such a management system. We extend a low-overhead distributed measurement capability [6] into clients running web browsers. We then use a model-based management system to perform QoS monitoring and diagnosis [16]. We believe the use of pervasive measurement and model based management is crucial for lowering the cost of managing distributed applications. Section 2 summarizes the needs for scalable measurements, and our recent extensions for Java applets. Our use of model-based management tools for QoS management is described in Section 3. Section 4 shows use of this architecture for monitoring and diagnostics for a test-bed application. Finally, we finish with related work and conclusions.

2 Measurement of Application Performance

A required first step toward managing the performance of any system is to get measurements of its behavior. For a distributed system, this implies measuring the performance of each of the components in the system, as well as the network interactions among them. Collecting these measurements has a number of significant challenges: ubiquity, low overhead and scalability.

An on-going goal of a ubiquitous measurement system is to collect data from all parts of the system. Distributed applications, particularly those spanning the Internet, often consist of components that utilize a variety of communication paradigms. Examples of these include Remote Procedure Call (RPC), Message Oriented Middleware (MOM), and application specific protocols such as `http`. In addition, these paradigms are implemented by different organizations and vendors. For example, RPC is provided by systems such as the Distributed Computing Environment (DCE), Common Object Request Broker Architecture (CORBA) and Java's Remote Method Invocation (RMI) among others. This variety presents an explosion of possible systems we may need to instrument. An approach to measurement must therefore be applicable to as many environments as possible.

We also require our measurement system to have low overhead in its use of processor, memory and network resources. In an operational setting, overhead due to measurement is seen by an application's customers. If measurement systems significantly increase the user perceived response times, administrators become inclined to disable measurement, thereby making management impossible.

Finally, our measurement system must scale as the size of distributed applications increases. It must not induce proportionally more overhead as new compute resources, network resources, or software components are installed. It must also maintain low overhead as the rate of component interactions increases. With these concerns in mind, we have participated in the development of the Distributed Measurement System (DMS) architecture.

Distributed Measurement System Architecture

Our earlier efforts with industry and academic collaborators defined the architecture for a Distributed Measurement System [15]. The DMS architecture consists of four components: an instrumentation application sensor programming interface (API), a data reduction engine, a multi-manager collection agent, and an API for accessing and controlling measurements. This architecture is depicted in figure 1. This component decomposition addressed our objectives of ubiquity, low overhead and scalability.

DMS' instrumentation and measurement access APIs are portable to most programming languages and distributed computing infrastructures. Once the APIs have been targeted to a new language, changes in the distribution paradigm have little impact. With the API in place, the DMS architecture applies to most environments.

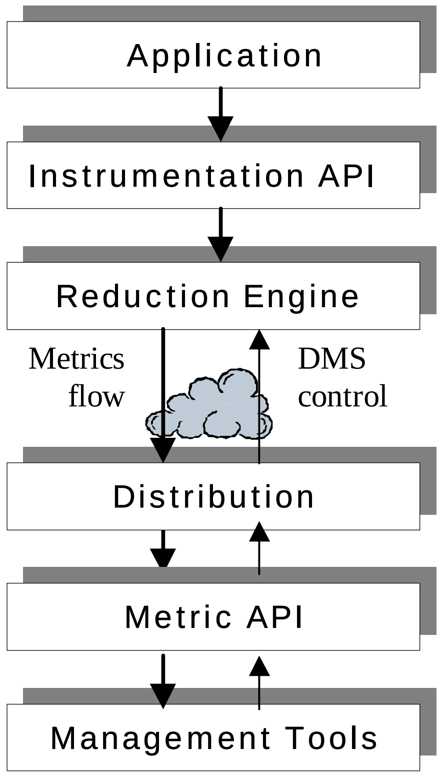

Figure 1 DMS Architecture and Interfaces

We first make the instrumentation API simple so that its use is easier to learn. For lower communication costs, we co-locate a data reduction engine with each application component. This reduces the amount of data that must be stored at each component and the bandwidth and RPC visits required distributing this data. DMS minimizes both network and inter-process communications. These configurable data reductions *intervalize* a set of statistical measurements henceforth called *metrics*. For example, only the sum of response times and number of measurements for a time interval are transmitted. As discussed later, management systems make use of metrics, so performing reductions does not interfere with management goals. An option exists to generate individual measurements for specific diagnostic needs.

DMS' reduction engine and distribution infrastructure make it scalable. Reducing the volume of data allows DMS to scale as the total number of monitored interactions increases. The distribution infrastructure aids in scalability by forwarding metrics only to those compute resources where a management tool requires the information. Therefore, when no tool needs a metric, activities related to a metric can be "turned off" so that overhead associated with generation and collection is nearly eliminated. This permits developers to insert instrumentation in a large number of places minimizing concern that communication resources will be swamped.

An ideal location for instrumentation is directly in distribution infrastructures such as DCE or CORBA. At this level, it is possible to have instrumentation automatically generated using the same mechanisms that generate the RPCs or other communication calls. This lessens the need for the application developer to insert instrumentation, and assures instrumentation that covers all software component interactions. Because DMS can turn individual instrumentation points on and off, this “brute force” approach will not result in excessive overhead. Hewlett-Packard has taken this approach in its implementation of DMS for DCE.

Use of Standards

Another key to getting ubiquitous instrumentation is to adhere to standards. In DMS, we specify only that a simple instrumentation API will be used. A 1996 standard proposed by Hewlett-Packard and IBM is the Application Response Measurement (ARM) API. ARM specifically avoids placing restrictions on how its API will be implemented. To leverage this standard, we have adopted ARM as our application-level API. Because of the component nature of DMS, it has been possible to introduce ARM as its API component without significant modification of the rest of our implementation. The most significant limitation in the ARM specification has been that its interface is specified only for programs written with the procedural model. In fact, its specification is written in C. Because we wish to instrument applications written in object languages, we have extended the ARM specification into a model that supports them.

Java Applet Instrumentation

Java applets embedded in web pages are becoming increasingly popular as a front-end for distributed applications because of their portability. We place instrumentation in these applets because it measures the end-to-end performance observed by customers. Without these measurements, we must estimate a customer’s view based on indirect measurements taken at the servers.

To instrument applets we extended the ARM API to Java, and implemented portions of the DMS architecture in Java. The Java security model placed restrictions on how we distribute the metrics which result from measurements. The model allows an applet to make network connections only to the site from which it was downloaded. We therefore return all of an applet’s metrics back to the web server from which it was downloaded. Converting the reduction engine to Java was a straightforward port of existing C code. The value of the local reduction engine is magnified in the Internet setting because it significantly reduces the bandwidth required for metric distribution. The result of porting DMS to Java has been to perform measurements of Java applets in a manner that is transparent to the user. Also, it provides the management system with an end-to-end view of a distributed application, even if that application’s client code runs in a web browser.

Use of Existing Metrics

Various vendors including Hewlett-Packard have measurements systems in place to deliver information derived from standards such as Simple Network Management Protocol (SNMP). These use ubiquitous network hardware based agents implementing remote monitoring (RMON) Management Information Bases (MIB). We seek not to duplicate, but take advantage of these existing systems. Many products exist which provide information that depicts the performance of point-to-point communications over the network in either raw or summarized form. In particular, we use

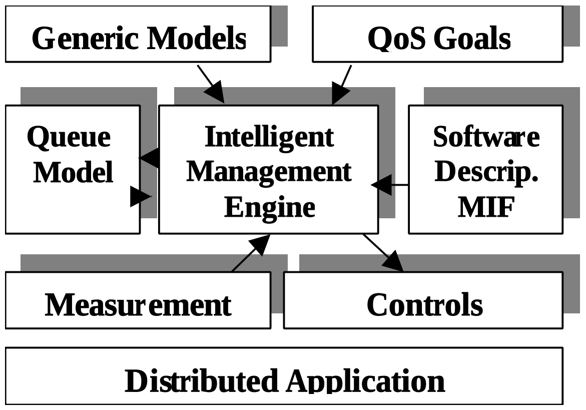

Figure 2 Our management agent uses measurements agents, models, goals and software descriptions for monitoring and diagnostic activities.

Hewlett-Packard network latency detection monitoring to provide data to support our diagnostic engine described later.

3 Managing Distributed Applications

Once instrumentation is built into distributed applications, the stage is set for a management system. A management system correlates and interprets measurements with various goals: monitoring for availability, QoS violations, or performance bottlenecks. A management system needs to be responsive so that it provides timely analysis of emerging performance problems. We claim that other important requirements of a management system are:

1. A focus on the *specification* of the QoS goals and behavior of the distributed application. The management system reasons at an abstract level and is independent of any implementation of system goals.
2. Its ability to be *compositional.* It should be easy to compose applications from components, and application management from component management.
3. It must be *flexible* and *extensible*. That is, it should adapt easily to changes in system behavior and management goals.
4. It should lend itself to *multi-use*. Different management goals should be able to share the same system behavior specification. For example, the code written for monitoring an application should not be very different from that used to diagnose failures of QoS within it.

Model-based Reasoning

Model Based Reasoning (MBR) [9] provides a framework necessary for articulating a management system with the above requirements. Model based approaches focus on describing the *expected* modes of operation of each component in the system. Furthermore, only the properties of interest are modeled. All other details are abstracted away. For example, when managing for performance, we model only response time, visit count, utilization, and interaction characteristics of each component. Anomalies are detected by comparing measurements with baselines, or queueing model results.

The industry is beginning to converge on standards such as Microsoft's WBEM that provide ubiquitous management information models, and on standards such as the Unified Modeling Language (UML) and DMTF, that populate these models. Software design tools and design languages including UML encourage the design of systems as objects and resources, and describe the relationships between them. The emergence of these models and tools makes MBR a natural choice for management since we can take advantage of the mostly automatic instantiation of significant parts of the management model.

The Components of Management

Our approach to accomplishing automated performance management of distributed applications is summarized in Figure 2. The different components of this approach are as follows.

- As described in Section 2, the measurement instrumentation in a distributed application makes available individual component metrics such as average response time, utilization, and throughput.
- Software component and application specific details such as configuration, deployment, use-cases, task substructure, and sensors, are described in a management information format (MIF) file. This file, installed when the software component is installed, is read and made available to management applications by the desktop management information (DMI) service layer.
- Metrics from the operational application are compared to the nominal values seen previously, or predicted values computed by queuing models.
- Models of the generic properties of software components, component interactions, application tasks, and applications exist.
- A model based engine instantiates a model of a managed application by correlating the MIF supplied application details, measurements, and nominal values for performance metrics with the generic application models.
- The definition of QoS goals is obtained from SLAs or from design documents. These goals direct the management engine in monitoring the operational application. It can then compare nominal values against the values from the operational application to determine whether performance bottlenecks are emerging. Extant bottlenecks are diagnosed by tracing and analyzing the interdependencies between application components.

Now we describe in more detail our management approach.

The Manager Implementation

We have found building application management tools most effective using model based reasoning. We use a model-based language and reasoning engine called Flipper [16]. It provides an object-oriented logic programming language in which classes of system resources and services are modeled as objects. These models consist of rules that find instances of objects, rules that provide information about the object, and diagnostic rules that describe correct object states. Rules are easier to write since hierarchical class construction of modeled objects allows attribute and rule polymorphism. Flipper's extensible architecture accommodates plug-in access modules that allow easy access to facts and measurements. Its capability of monitoring individual goals fits well with our desire to monitor the QoS delivered by an application. When a goal is submitted to the reasoning engine, we aggregate and correlate the sensors that impact it.

Generic Models

Generic application models can lower the cost of the development of application models. Therefore, it is important to determine exactly how generic we can make these models before customization for a particular application is needed. The following is a description of a generic application model.

- An application is composed of one or more software *components*. A component is an independently distributed and addressable piece of functionality and data.
- An application is driven by different classes of *actors* (users), who have specific *use cases* (workload profiles) composed of *tasks*. Each class of users may require different QoS.
- Each generic software component offers different *services* through its published interfaces.
- Component *interactions* are based upon the use of services: there is always a client component and a server component in an interaction. These interactions may occur between different semantic levels of a component: method, interface, objects, clusters of objects, or processes.
- Interactions occur across a logical network *channel*. The logical channel exists as a communication pathway between two components. The channel may have specified minimum needs for QoS for use by configuration rules. The logical channels, when mapped to physical channels representing actual node-to-node measurements from network measurement products can serve diagnostic rules. Here we make use of existing network management products.
- Tasks are application-defined pieces of “work” that are of interest to the actors of an application, and to the management system. All tasks may have three parts: local, network and remote. The remote parts of a task are based upon component interactions, and are sub-tasks of the original task, as well as a task in their own right.

We use the following generic rules for diagnosis. Each task is subject to different *QoS goals*: either derived directly from a SLA, or derived from the QoS goals of the invoking transaction. If QoS goals are not specified, we rely on nominal values adapted from previous observations when the system was running satisfactorily. The subtasks within a higher level task may be composed synchronously or asynchronously, in an order that is dependent upon the specific application and task. The ability to correlate the performance metrics available from different subtasks is vital to a model of the task. Based upon the knowledge of the order and manner in which subtasks occur, the model can diagnose the task performance metric. This correlation of task metrics is necessary for tracing and diagnosing the QoS failures within a task.

Tasks have *nominal* performance characteristics associated with each workload to which they belong. We use these values to detect performance degradations, by a simple comparison with the current performance of the task, or through a more complex algebra, that takes into account task dependencies. A separate module may be used to adapt these nominal values to changing environments and time dependencies.

We currently have generic models for software components that communicate synchronously (e.g., via RPCs). These models provide a method by which large classes of distributed applications using RPC communication may be managed with minimum model re-writing, provided that they are characterized in the MIF file.

Instantiating a Generic Model

Component specific details such as its tasks of interest, the interaction substructure of each task, and the sensors associated with the component are obtained from a software component configuration file. We use the form of a DMTF MIF; specifically, a locally developed extension of the Application Management Specification[1] (AMS) [17]. The AMS normally provides software configuration information. Our extensions record the dynamic behavior and resource consumption of components.

4 A Test-bed Application

We have demonstrated these measurement and management approaches in a testbed application that mimics a typical transaction processing application with an applet used as a front-end. We use this application to demonstrate how a management system may use the uncorrelated individual measurements of each resource's performance to form a composite picture of an application's health. This picture is then used to monitor application performance, detecting degradations in Quality of Service, and diagnosing the cause of the degradation.

Our testbed is composed of database server components, a business logic component, and a name server component, all of which reside on UNIX platforms. Both, Java-based client front-ends with web-server plug-ins, and Intranet clients, access these components. These clients reside on UNIX or NT platforms. Different clients perform distinct classes of queries and are capable of generating stochastic workloads. We use these workloads to induce different performance bottlenecks for model testing. The various application components may be deployed on machines on a LAN and a WAN, to increase latencies in a transaction. All interactions between application components are currently synchronous. The topology of the interactions generated by a client query is represented in Figure 3.

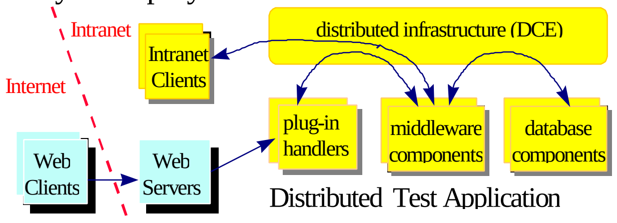

Figure 3 Internet-enabled three-tier application instrumented with DMS and monitored by the Intelligent Management agent.

The significant tasks and interactions in each component are instrumented by either infrastructure or application sensors. At each component, we can interrogate for the response time of its service and each service it invokes. At each service, we have an aggregate service time over all clients. We take the difference between these two measures to be the latency, or the time of the communication costs. When actual measured latencies from network measurement systems are available between IP nodes or other convenient points, we can use those measures directly.

Initially, we monitor only the aggregate end-to-end response time over all clients, ignoring the measurements available from all other application components. This vastly reduces the amount of work generated by the application for the management system, and is in contrast to previous, event triggered approaches to management [10]. As an example, Rule 1 is a Flipper rule that monitors a

[1] AMS is an IBM/Tivoli proposal that extends the software MIF standard, and is currently under consideration by the DMTF.

```
RULE 1   [Task task] meetsClientRespTimeGoal IF
              [task] qosGoals [QoSGoal goal]  &
              [goal] respTime [Real target] &
              [task] clientRespTimeAvg [Real respTime] &
              [target] greaterThanOrEquals [respTime] &
              [String currentTime] nowAndEvery [30000].

RULE 2   [Task task] performanceBottleneck IF
              [task] violatesClientRespTimeGoal [Real r] &
              ([task] cpuHog |
               [task] ioBound |
               [task] channelHosed |
               [task] serviceBottleneck [Task service]).

RULE 3   [Task task] serviceBottleneck [Task service] IF
              [task] subTask [service] &
              (~[service] meetsClientRespTimeGoal |
               ~[service] meetsBaseLineGoal |
               ~[service] lowLatency).
```

task's response time QoS every five minutes. The '&' represents a conjunction (logical AND) and '|' a disjunction (logical OR).

Once the aggregate response time for a particular task violates its QoS threshold, the diagnostic rules kick in. These rules examine all available measurements for that task in an effort to identify the cause of the violation. In Rule 2, a task contains a performance bottleneck if its response time QoS goal is violated, or the local (CPU and I/O), network (channel), or remote (services) that it is dependent upon contain bottlenecks. In this case, if the local machine has a process running on it that is a hog, or the network is down, or one of the remote subtasks are a bottleneck, then we can diagnose the cause of failure.

A particular subtask may be a bottleneck for its containing task if it violates its response time QoS goal or its nominal performance has changed, or the latency between the task and the subtask is too high. In Rule 3, the symbol ~ is the negation operator. Other diagnostic rules include combining the service response times according to the order and manner of their invocation, and are the subject of continuing research.

5 Related Work

There are several documents concerning the performance challenge for both measurement and management tools [2][6][7][9]. However, few show an approach where network and software components are integrated. The joint IBM-academic MANDAS project has produced a comprehensive approach in the DCE framework [2]. Its management framework specifies the collection, delivery and actions for the distributed computing system, including the interfaces between the management process and the managed objects. Moreover, it fails to address scalability of either the measurement or the management processes for larger systems.

Previous approaches to automating performance management include Case Based Reasoning (CBR) and automated scripts. CBR involves matching new problem instances against previously solved problems, retrieving only those cases that are most similar to the current instance [11]. CBR is less flexible as compared to MBR — the choice of attributes to match and the similarity metrics between problem instances are all determined at design time, and mathematical similarity metrics make for less clear explanations of system choices. Automated scripts that provide application management are customized to particular applications and do not allow for code reuse or flexibility.

Koch and Kramer describe Marvel [9], an experiment that shows how to automate distributed systems management by use of rules. Their rules are chained to accomplish management tasks. ANSA-based generic monitoring agents provide event propagation. Marvel has similar concepts to our approach, using existing information (SNMP) when possible, policy definitions, and domain projections. Its distinctions between process and data models however make rule building and object modules more difficult to maintain. The system relies on event propagation for performance management, an approach we believe is not scalable to large distributed systems.

Standards bodies have made many efforts to specify an architecture for application management. Examples include the Open System Interconnection (OSI), Telecommunications Management Network (TMN), Open Software Foundation's[2] Distributed Management Environment (DME), and the Desktop Management Task Force (DMTF) Desktop Management Interface[3] (DMI). OSI is a difficult standard without general application. TMN has found some success in its targeted telecommunications area, and DME was too late as CORBA eclipses DCE. DMTF is only evolving, hobbled by its origins in the desktop hardware management, and far from enterprise applications. There are other conceptual standards such as RMODP[8], but they do not provide systematic, coherent mechanisms to accomplish measurement and management[13]. The Object Management Group's (OMG) Object Management Architecture (OMA), despite its name, has very little in substance for the management of objects.

There are proprietary efforts, notably IBM/Tivoli Management Environment (TME) and Microsoft Web-based Management (WBEM) schema and protocols[4]. These are not integrated solutions. Rather, they are management toolboxes with different middleware and incompatible management applications and metrics. TME pays scant attention to addressing the interdependence of managed objects to achieve a required QoS. WBEM, more a Microsoft standards discussion and alpha toolkit, does not address operational management of distributed applications in an heterogeneous enterprise.

6 Conclusions and Future Work

We have argued that the implementation of applications as distributed software components requires sophisticated measurement infrastructures and significant advances in management tools' technologies. We advance an architecture based on two pillars: measurement and management. A scalable distributed measurement system has at its heart, controllable intervalization and shared infrastructure. The management tools use model-based reasoning engines, exploiting the structure

[2] Open Software Foundation (OSF) is now known as the OpenGroup, part of XOpen.

[3] We used extensions on DMI to describe software components for our models.

[4] Microsoft is working with DMTF to incorporate WBEM into a standards process during 1997.

inherent in how software components interact including the decomposition of the user-level Quality-of-Service into the software and network responsibilities for service at the component level. We described how our models are generic to a distributed application, and are instantiated by MIF files that accompany the installation of software binaries on a system. We demonstrate the architecture on a three-tiered client/server system which included access using Java applets through a web-browser front end connecting to the backend systems in a low overhead, internet server plug-in. The demonstration was designed to show that tracking end-to-end response time is necessary, but not sufficient for effective management. Diagnostics require causality and measurement correlation best-implemented using model-based reasoning. Relationships and dependencies on resources such as network channel latencies and node environments are explicit.

Future Work

Significant challenges exist in distributed application management. Our work continues to extend measurement in more distributed infrastructures, including CORBA, DCOM and framework architectures. Incorporating other organizational work in federated management, we expect to pursue integration of network and node management systems with our own. We intend to elaborate our application models to handle asynchronous and parallel transactions between components, as well as to improve diagnostic rules that are more capable of handling ambiguity and incomplete information.

7 References

[1] Cristina Aurrecoechea, Andrew Campbell and Linda Hauw, *A Survey of QoS Architectures"*, Multimedia Systems Journal, Special Issue on QoS Architecture, 1997, (to appear*).*

[2] Michael A. Bauer, et al, *MANDAS: management of distributed applications and systems*, Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing, Aug. 1995.

[3] Robert F. Berry and Joseph L. Hellerstein, *A Unified Approach to Interpreting Measurement Data in Performance Management Applications*, Proceedings of the IEEE First International Workshop on Systems Management, April 1993.

[4] Philippe Desfray, *Automated Object Design: The Client-Server Case*, IEEE Computer, February 1996.

[5] M.J. Freeman and P.J. Layzell, *Experience Realising a Meta-model for Wide System Understanding: The Global System Model, Software - Practice and Experience*, Vol. 24(8), August 1994.

[6] Richard Friedrich, Joseph Martinka, Tracy Sienknecht, Steve Saunders, *Integration of Performance Measurement and Modeling for Open Distributed Processing*, Proceedings of the International Conference on Open Distributed Processing (ICODP'95), Brisbane, Australia, February 1995.

[7] James W. Hong and Michael A. Bauer, *A Generic Management Framework for Distributed Applications*, Proceedings of the IEEE First International Workshop on Systems Management, April 1993.

[8] Basic Reference Model of Open Distributed Processing (RM-ODP), *ITU-TS Rec X.901, ISO/IEC 10746*, Part 1: Overview and Guide to the Use of the Reference Model*: 10746-1*, Part 2: Foundations: *10746-2*, Part 3: Prescriptive Model, June 1994.

[9] Johann de Kleer and John Seely Brown, *Model Based Diagnosis in Sophie III*, Readings in Model Based Diagnosis, Morgan Kaufmann, Feb 1992.

[10] Thomas Koch and Bernd Kramer*, Rules and agents for automated management of distributed systems*, Special issue of the Distributed Systems Engineering Journal on Distributed Systems Management, June 1996, IEEE and British Computer Society.

[11] Joseph L. Hellerstein, *Automating Performance Management Using Case-Based Reasoning*, Technical Report RC-20083, IBM TJ Watson Research Center, May 1995.

[12] Joseph Martinka and Kave Eshghi, *An Architecture for Adaptable Distributed Application Management (ADAM)*, Hewlett-Packard Laboratories Technical Report, HPL-96-30, March 1996.

[13] Joseph Martinka, Richard Friedrich, Tracy Sienknecht, *Murky Transparencies: Clarity Through Performance Engineering*, Proceedings of the International Conference on Open Distributed Processing (ICODP'95), Brisbane, Australia, February 1995.

[14] Anna Melamed, *Distributed Systems Management on Wall Street – AI Technology Needs*, Proceedings of the 1st International Conference on Artificial Intelligence on Wall Street, October 1991.

[15] Open Software Foundation, *Standardized Performance Instrumentation and Interface Specification for Monitoring DCE Based Applications*, DCE RFC 33.0, November 1994.

[16] A.R. Pell, K. Eshghi, J-J. Moreau, S.J. Towers, *Managing in a distributed world*, Proceedings of the 4th IFIP/IEEE International Symposium on Integrated Network Management, May 1995.

[17] IBM/Tivoli Application Management Specification [AMS], http://www.tivoli.com:80/AMS.