# Making Distributed Applications Manageable Through Instrumentation*

Michael J. Katchabaw Stephen L. Howard Hanan L. Lutfiyya Andrew D. Marshall
Michael A. Bauer

Department of Computer Science
The University of Western Ontario
London, Ontario, Canada N6A 5B7
E-mail: {katchab,showard,hanan,flash,bauer}@csd.uwo.ca

## Abstract

*The goal of a management system in a distributed computing environment is to provide a centralized and coordinated view of an otherwise distributed and heterogeneous collection of hardware and software resources. Management systems monitor, analyse and control network resources, system resources, and distributed application programs.*

*Many organizations currently depend on mission-critical distributed applications, a trend that will increase as software engineering tools emerge that make it easier to construct distributed applications. We believe that manageability must be built in to distributed applications from the beginning rather than added in an ad hoc fashion after they have been developed. Just as designing software for usability, testability and maintenance are being addressed in the development process, so must designing for manageability.*

*Application manageability is a research issue of particular interest to us. The work described in this paper focuses on instrumenting processes to allow them to respond to management requests, generate management reports, and maintain information required by the management system. We present an instrumentation architecture to support this, a prototype implementation which includes a class library of standard instrumentation, and a methodology for instrumentation.*

*This research work is supported by the IBM Centre for Advanced Studies and the Natural Sciences and Engineering Research Council of Canada.

## 1. Introduction

A distributed computing system consists of heterogeneous computing devices, communication networks, operating system services, and applications. As organizations move toward these environments, distributed applications will become more prevalent and play a critical role in day-to-day operations.

The design, development, and management of distributed applications presents many difficult challenges. As distributed systems grow to include hundreds or thousands of hardware and software components, it will become increasingly difficult to locate faults, identify bottlenecks, anticipate problems, and monitor behavior. Doing so, however, is critical to ensuring their reliability and performance.

Application manageability is a research issue of particular interest to us. While some other work has been done in this area [2, 12, 13], successes to date have been limited to a particular platform or environment, involve only measurement (rather than management), or limit their scope to a single management function.

Based on our experience in developing manageable distributed applications, we believe that manageability must be built in to distributed applications rather than added to them in an ad hoc fashion after they have been developed. Just as designing software for usability, testability and maintenance are being addressed in the development process, so must designing for manageability.

Our approach to distributed application management requires instrumentation; that is, code inserted into the application at strategic locations so a managed process can maintain management information, respond to management requests and generate event reports. Developers are understandably concerned about



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

the increase in overhead, decrease in performance, greater development effort, and loss of flexibility associated with instrumentation. The reality is, however, that to achieve the level of management required in deployed applications, this instrumentation is necessary. Related work demonstrates the need to collect internal application behavior for such things as performance [3, 10, 16], visualization [20], and reliability [9]. We believe the question is not whether instrumentation is necessary, but rather how applications can best be instrumented to meet the needs for management while minimizing the concerns of developers.

We recognize the fact that adding another facet to the development process will require extra resources and impose an extra burden on development teams. One way to address this is to create tools and techniques to facilitate the development of manageable software. At the same time, we see that human involvement will be unavoidable in the cases where custom instrumentation is necessary. By developing an instrumentation architecture, we can make efforts toward automating some parts of the process, and provide guidance to facilitate the development of custom instrumentation in a controlled and structured manner.

In this paper, we identify a set of basic objectives and propose an instrumentation architecture and methodology to satisfy them. We describe the components of this architecture and their interactions and discuss how instrumentation is carried out in the context of our architecture. We then describe how the instrumentation concepts were used to manage Distributed Computing Environment (DCE) [11] applications in a prototype environment under the OSI Management Framework [5, 6].

## 2. Objectives

A number of objectives were established to guide our study of instrumentation and the development and refinement of our instrumentation architecture and prototype.

- **Data Acquisition and Control** For effective management, we must be able to acquire both generic and application-specific management information from the processes being managed. In addition to acquiring management information from managed processes, the management system must be able to exert control over these processes to influence their operational behavior. Consequently, both forms of management interactions must be supported.
- **Dynamic Management** It is important that management be dynamically "tunable" so its behavior can be adjusted to meet operational needs. We must be able to adjust which attributes are collected and at what intervals, to define the processing that is to be done on collected data, to establish thresholds used to generate alarm reports, and to disable management altogether in exceptional circumstances.
- **Flexibility** The instrumentation architecture must not make assumptions about the hardware, operating systems, middleware, or communication paradigms used in the management environment. Similarly, the architecture should accommodate a wide range of management functions including configuration, performance, accounting, fault tolerance and security. The architecture must also co-exist with current standards.
- **Transparency** The development process for distributed applications is a difficult one; the additional burden of instrumentation must be minimized. The instrumentation architecture should support automation of the instrumentation process, so it can be done with little developer effort. In addition to reducing developer effort, the notion of run-time transparency must be observed: users of managed applications should not be aware of substantial performance overhead caused by management.

## 3. Management Environment

In this section, we define the general framework of our distributed application management system. The framework is based on the Open Systems Interconnection (OSI) Management framework [17, 18]. Figure 1 illustrates the basic structure and interactions within this framework.

Management systems contain three types of interacting components: *managers* (automated or otherwise), which make decisions based on collected management information and relevant management policies; *management agents*, which collect management information; and *managed objects*, which represent actual system or network resources being managed.

A managed object is an abstraction of one (or more) real resources and is defined in terms of the attributes it possesses, the operations that may be performed on it, the notifications it may issue and its relationships with other managed objects.

A *management application* automates one or more management tasks on behalf of a tractional human manager. A management agent (or simply, an agent) is responsible for a particular collection of managed objects. The agent and its managed objects serve to decouple management applications from their managed resources. This approach has many advantages. It facilitates many-to-many relationships between manage-



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

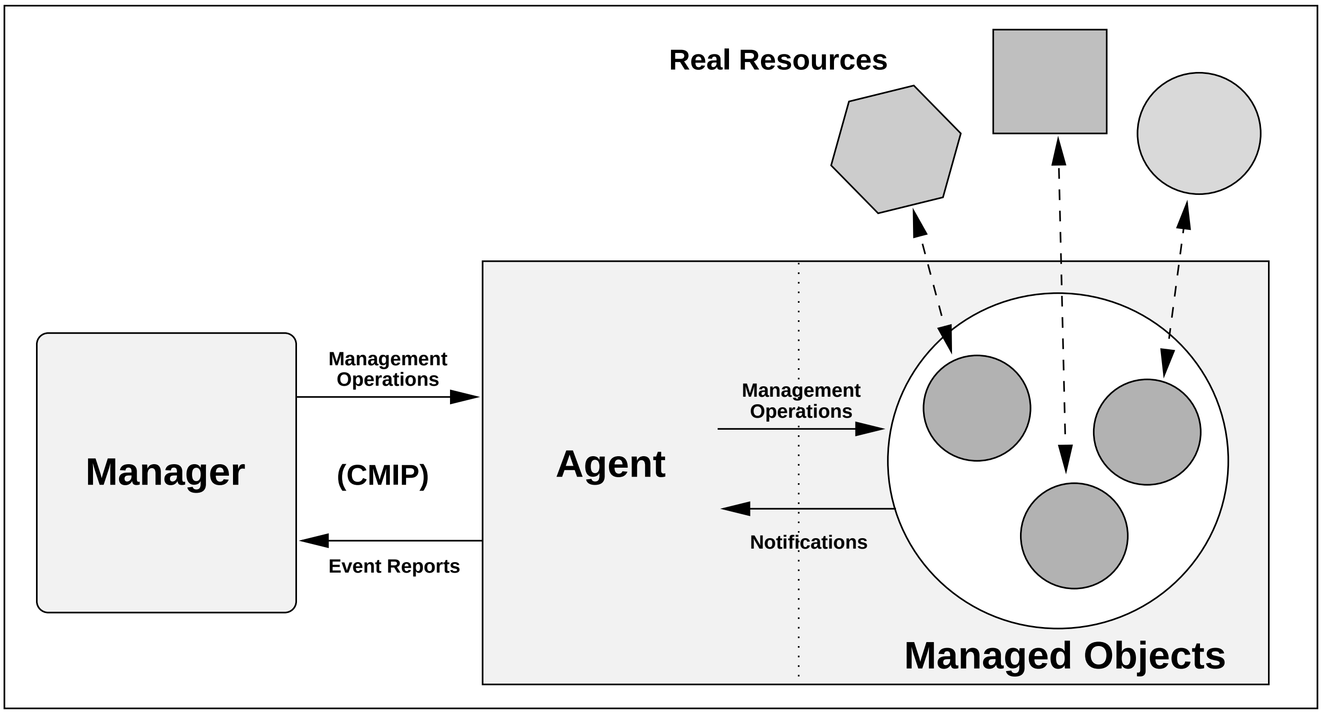


**Figure 1. Management Framework**

ment applications and managed resources. It provides a means to distribute the management function: localised agents can perform such tasks as data aggregation, filtering, analysis and logging, ultimately reducing the flow of data to managers.

An agent receives management requests from managers and carries out the operations on the appropriate managed objects. Conversely, notifications emitted by managed objects are routed to the appropriate management applications. Management agents perform operations requested by managers and notify managers of pre-determined events of interest to the manager.

## 4. Instrumentation Architecture

The instrumentation architecture is object-oriented: Classes of standard instrumentation components can be developed for general management tasks and new components can be specialized easily from existing standard classes. This allows users to develop application-specific instrumentation components tuned to their particular needs.

### 4.1. Instrumentation Component Overview

In Figure 2, which illustrates the instrumentation architecture, one of the instrumented processes has been expanded to reveal the instrumentation components, each of which is described below.

• **Instrumented Process** An instrumented manageable process is an application process with embedded instrumentation.

• **Management Coordinator** The management coordinator facilitates communication between the management system and an instrumented process. Its role includes message routing, creating and destroying sensors and actuators (see below), handling management requests to dynamically modify the behavior of sensors and actuators, as well as initialization and termination activities within the managed process.

Since communication with the management system is contained within the coordinator, the architecture minimizes the effort to instrument a process for different management systems. For example, we could create a generic class of management coordinators and derive new subclasses to operate in different management environments (OSI, SNMP, CORBA, etc.) allowing other instrumentation components to be reused without modification.

• **Sensors** Sensors are instrumentation components that encapsulate management information. They collect, maintain, and (perhaps) process this information within the managed process. Sensors exhibit monitor-like qualities in that they encapsulate man-



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

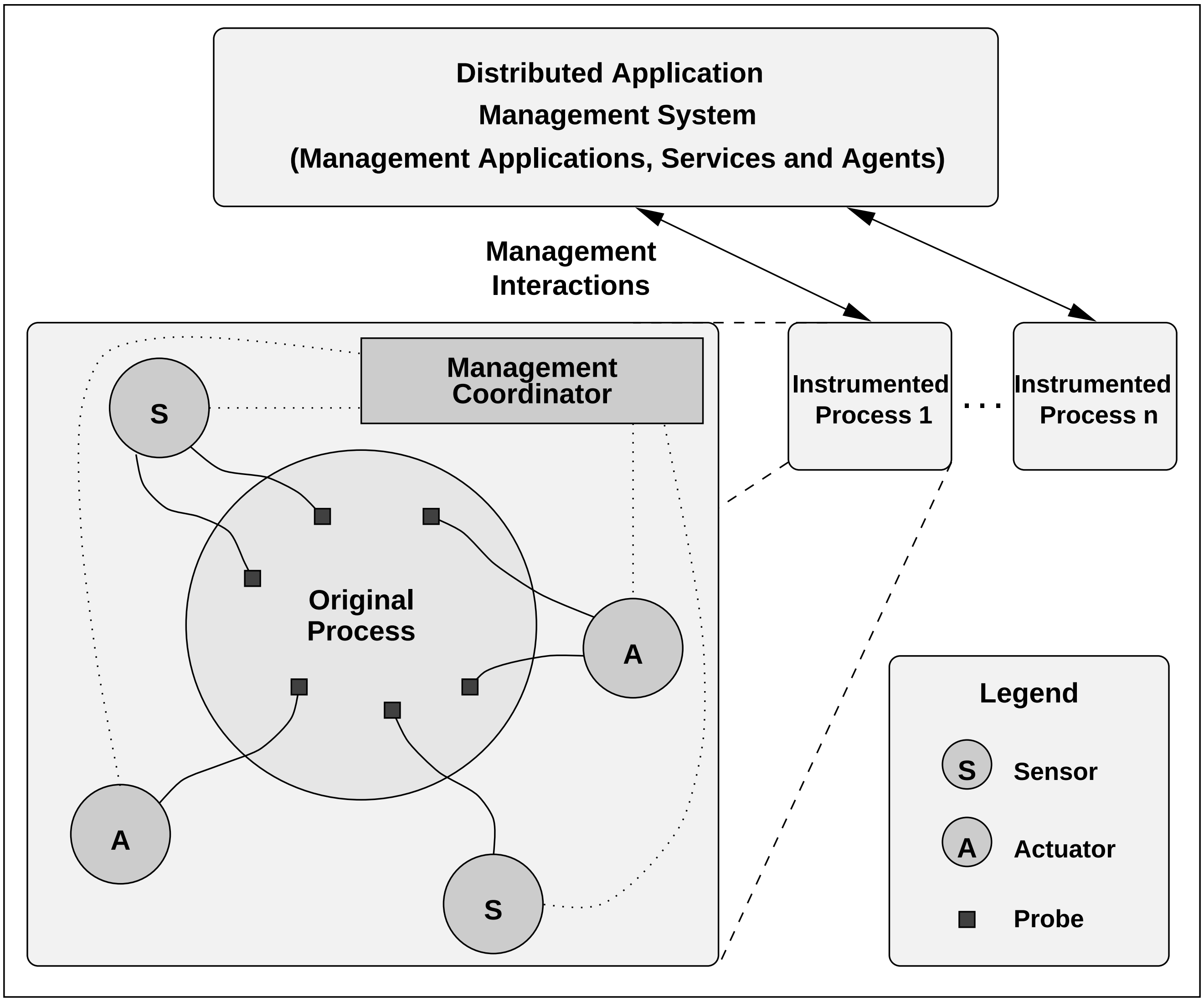


**Figure 2. Instrumentation Architecture and Environment**

agement information and provide an interface through which probes, other sensors and the management coordinator can access their state in a controlled manner. Sensors can be provided for a variety of performance metrics, to measure resource usage, to collect accounting statistics, to detect faults, to capture security information, and so on. Sensors get their input data from probes inserted at strategic points in the application code or by reading other sensors. Sensors provide their information to the management coordinator in the form of periodic reports, alarm reports (when exceptional or critical circumstances arise), or in response to explicit requests.

Sensors can be created and destroyed at run-time and they support a variety of behavior control operations. For example, sensors can be enabled or disabled, reporting intervals can be adjusted, event thresholds can be modified, and sensor processing algorithms can be changed.

One can think of a sensor as a reusable software component that can be plugged into an application process via one or more probes for the purpose of capturing management data. It is envisaged that a standard set of sensors would be available to application developers to meet most instrumentation needs. For other application-specific information, custom sensors can be built or derived from standard sensors, without the need to change the management coordinator.

• **Actuators** Actuators encapsulate management functions which exert control over the managed pro-



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

cess to change its operation or behavior. This includes generic process control such as termination, suspension, and changing process priority, as well as more sophisticated controls such as modifying incoming request queue lengths, changing relationships with other processes, and changing access control lists. Like sensors, actuators carry out these control operations through interactions with probes at key locations in the process or through interactions with other actuators.

Also like sensors, actuators are dynamic and can be created or destroyed at run-time. They perform control operations when requested to do so by the management coordinator, and may return a result to the coordinator, depending on the operation performed. An actuator also supports behavior control operations; for example, to enable or disable the actuator. A set of standard actuators would also be available to developers, with facilities provided for developing or deriving custom actuators for application specific tasks.

- **Probes** Probes are embedded in the process to facilitate interactions with sensors and actuators. Probes allow sensors to collect management information and allow actuators to exert control over process behavior. Each probe is specific to a particular sensor or actuator.

Sensor probes may be macros, function calls, or method invocations injected, during development, into the instruction stream of the application at strategic locations called *probe points*. Actuator probes are operations in the process address space which can be invoked by actuators. Probes linked to standard sensors and actuators can be placed in middleware services or system libraries to provide transparency and ease the instrumentation process. Custom sensors and actuators (or standard ones inserted in non-standard places) can be added by hand or through the use of function or class wrappers. (This is discussed in detail in Section 5.)

### 4.2. Service Interfaces

The service interfaces represent points of interactions between managed processes, through their coordinators, and the agents. We first present the interface used by managed processes to access the agent.

**sendManagementInformation** is used by managed processes to report management information to a management agent.

**notifyProcessRegister** allows a new process to notify a management agent of its existence and to provide information about it including its name, identifier, host, presentation address, etc.

**notifyTermination** notifies the agent that a process is about to terminate.

**notifyUnmanage** notifies the agent that the process wishes to not be "managed".

**notifyCommunicationsFailure** notifies the agent that a request from one application process to another failed due to a communication error.

**notifyServiceFailure** notifies the agent that the process was unable to successfully contact some system service.

**notifyCongestion** notifies the agent that the process is experiencing congestion because of over-utilization and that the expected quality of service has degraded.

We now present the interface used by the agent to access managed processes.

**requestManagementInformation** is used by the agent to request information from managed processes.

**requestTermination** instructs the process to terminate as quickly as possible.

**changePriority** instructs the process to change its run-time priority.

**controlManageProcess** activates management in a process, informing the process which agent will be managing it.

**controlUnmanageProcess** deactivates management in a process when it is no longer needed.

**controlChangeManagementSettings** tailors the event reports generated by the process.

### 4.3. Interactions

The interactions between the management system and managed processes include requests from the management system to the processes and reports flowing in the other direction. Requests can be made to retrieve management information, exert control over a managed process, or to change the way in which a process is being managed. Reports are generated at process initialization, at periodic intervals, upon the detection of alarm conditions, and at process termination. In this section, we describe the roles of the instrumentation components presented in Figure 2 as they relate to these interactions.



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

**4.3.1. Management System to Instrumentation Interactions.** Requests from the management system to the instrumented process are received by the coordinator which, in turn, routes them to the appropriate instrumentation components as follows:

- A request for management information is sent as a read request to the sensor (or sensors) responsible for the information requested. The sensors then report their current state, perhaps after some processing. This state is updated by the sensors' probes, or through interactions with other sensors. The results are returned through the coordinator to the management system.

- A request to execute a control operation is sent as an action request to the actuator (or actuators) responsible for that operation. The actuators involved use their associated probes to carry out the operation; if necessary, interactions with other actuators may also be used. Results, if any, are returned through the coordinator to the management system.

- Requests to control some aspect of management behavior within the process are interpreted by the coordinator and transformed into appropriate activities. For example, if the request calls for the collection of additional management information, new sensors are created or existing ones are enabled. Conversely, a request for less information results in existing sensors being destroyed or disabled.

**4.3.2. Instrumentation to Management System Interactions.** Reports from process instrumentation to the management system go through the management coordinator. There are a variety of circumstances in which such reports are sent from the coordinator to the management system.

- When an instrumented process is initialized, a management coordinator is created within it. The coordinator then creates and enables a default set of sensors and actuators. To register with the management system (notifying the system of the existence of the instrumented process and an interface through which the process can be managed), the coordinator issues read requests to the appropriate sensors to gather the information. The sensors return this state information, which was originally collected by their probes.

- Every sensor reports its state to the management coordinator at the end of each defined collection interval.

- Based on input gathered by its probes, a sensor may generate an alarm to indicate an unusual condition. The sensor reports the appropriate information to the management coordinator.

- When a managed process terminates, the management coordinator is notified. Before termination occurs, the coordinator collects information from the appropriate sensors and sends it to the management system to notify it of the termination. The coordinator then destroys all sensors and actuators before terminating itself. The process then terminates normally.

## 5. The Instrumentation Process

In the previous section, we have described *what* is needed for instrumentation. We now look at the places where instrumentation should be placed in an instrumented process.

### 5.1. Placing Instrumentation

Instrumentation probes must be inserted at strategic locations (probe points) in the application source code. Other instrumentation components (the management coordinator, sensors, and actuators) are located in the same address space as the process, but do not need to be embedded within the original code of the process. Instead, they can simply be linked into the process through an instrumentation library.

Probes must be placed where significant state transitions occur, where management information is accessible, and where control can be exerted over the managed process. These candidate locations include:

• **Entry Points** Every process entry point must be instrumented to allow instrumentation to be initialized and to allow process registration to occur. This includes process start-up (for example, in the main function) and after the spawning of new threads or subprocesses.

• **Exit Points** To notify the management system of the termination of a process or thread, probes must be inserted at exit points. These probes can be used to provide the reason for termination as well as the process state just prior to termination. The probe points of concern include the points at which an exit function is called or the main function returns.

• **Inter-process Communication** At the points where one process communicates with another process in a distributed application, performance metrics can be computed, faults can be detected, accounting can be



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

performed, security policies can be validated, and application configuration may be changed. In addition, aspects of the communication itself (which is usually a significant factor in the performance of a distributed application) can be monitored and controlled. As a result, the start and completion of all inter-process communication operations are usually important probe points.

• **Operating System or Middleware Service Invocation** Processes making use of operating system or middleware services require probes to monitor and control this activity. This includes operations associated with memory allocation and deallocation, time services, file services, naming services, binding services, etc. Once again, metrics can be computed, accounting can be performed, and violations can occur. Consequently, probes delimiting calls to these services are also very useful.

• **Exception and Signal Handlers** To deal with exceptions or signals that are raised during the execution of a process, special handlers are often used. Since these exceptions or signals could be significant events to the process, these handlers should by instrumented with probes.

• **Custom Probe Points** When custom-developed sensors or actuators are used, the locations for the insertion of their probes will depend largely on the application. Alternatively, a developer could use a standard sensor or actuator in a non-standard location. Once again, this becomes an application-specific probe point.

The above list of probe points is not exhaustive, but gives a strong sense of the level of instrumentation required for manageability. By instrumenting a process in such a way, management data can be collected and control can be exerted to support a range of management functions.

## 5.2. Performing Instrumentation

We now turn to the question of "how can a process be instrumented for management?" Figure 3 illustrates the alternative approaches we consider.

• **Hand-coded Instrumentation** Most application process instrumentation is carried out by hand during application development. The developer has considerable latitude in choosing what instrumentation to add and where to add it. In the context of our work, we try to make use of standard components and apply them in a prescribed manner to achieve a degree of consistency across applications and minimize the risk of error. Instrumenting by hand facilitates the development and use of custom sensors and actuators to meet

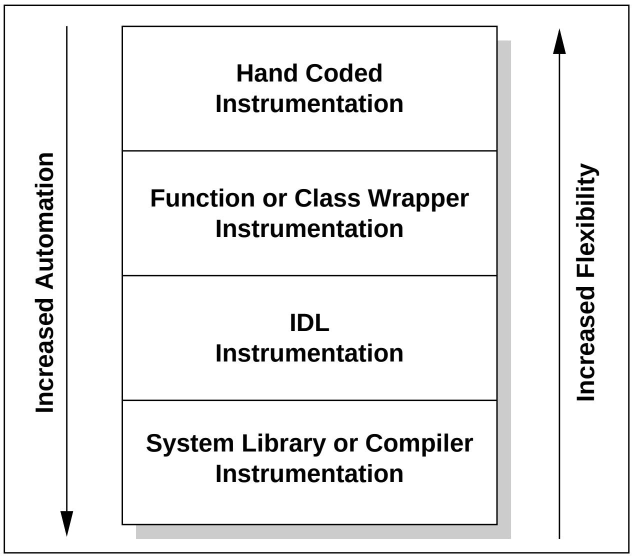


**Figure 3. Instrumentation Techniques**

application-specific needs which could not be met by more generic components.

The flexibility of hand-coded instrumentation comes at a cost. A substantial amount of time, effort, and resources can be expended instrumenting an application in this way. It can be both difficult and risky, because the potential for human error is high and the cost of error may be great.

Tools to automate the insertion of probes would greatly enhance this procedure. Some efforts have been made toward this goal [4].

• **Function or Class Wrapper Instrumentation** One possible method of automating the instrumentation process is to provide application developers with management "wrappers" for functions, data structures, or objects. Instead of accessing the function, structure, or object directly, the developer would use management wrappers which would already contain standard probes. For instance, a management wrapper `mf` around a function `f` might be implemented as `p1 f p2`, where `p1` and `p2` are standard probes. To make use of the function `f`, `mf` is called instead, producing the same result as if `f` were called directly, but with probes doing such things as timing the execution of `f`, incrementing a counter of calls to `f`, or checking the return value of `f` for possible failures. The same concept applies to wrapping data structures and objects.

Wrappers can help automate basic instrumentation and reduce the effort and risk of error. Under this scenario wrapped functions, data structures, and objects can be packaged into libraries for use by developers. Flexibility is reduced somewhat, since all uses of a par-



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

ticular function, data structure or object will employ the same set of probes. Consistency across all applications using the pre-wrapped elements is increased.

• **IDL Instrumentation** Many distributed computing environments use a formal interface definition language (IDL) through which attributes and procedure or object method signatures are specified. The interface definitions are compiled (using an IDL compiler) to produce stub code which hides most of the details of the underlying inter-process communication.

Since inter-process communication was identified as a key area for probes, there is a potential for automating this aspect of instrumentation within the IDL compiler. IDL compilers could be modified to insert probes automatically when generating communications stub code. This approach would further reduce the effort and risk of programmer error problems associated with probe insertion, but at a cost of some flexibility and customizability. Since stub code is normally produced in the source language of the application, the developer does have opportunity to access and modify generated stubs, buying back some of the lost flexibility. However, editing stub code is usually considered a dangerous and messy practice.

• **System Library or Compiler Instrumentation** By providing instrumented system libraries or a compiler that injects instrumentation at compile-time, most of the instrumentation task is handled automatically. This approach requires minimal developer overhead and introduces the least risk of programmer-induced error. It also relieves the developer of the need to insert probes for exception and signal handling routines, exit and entry points, and operating system and middleware services.

Instrumenting processes in this way covers most recommended non-custom probe points. Of course, application-specific instrumentation requires knowledge of the application's semantics and thus is still a developer task.

Perhaps the best example of work on this type of internal instrumentation can be found in [12, 13]; however, this work is aimed mainly at performance measurement rather than general management.

The optimal approach to instrumentation may well be a blend of all four of these techniques. Software engineering tools are being developed to facilitate construction of distributed applications. We see one of the tasks of these tools as automating instrumentation insertion to create manageable distributed applications.

## 6. Prototype Implementation

In this section we describe a prototype management system we have developed based on the generic architecture presented in the previous section.

### 6.1. Management Environment

The prototype builds upon University College London's OSI Management Information Service (OSIMIS) [14, 15] which provides an object-oriented infrastructure for developing management applications and OSI agents. OSIMIS includes a GDMO compiler [1] to parse managed object class definitions (specified in GDMO [7]) and generate the agent code required to access those managed objects. Through extensions to OSIMIS, our management system is capable of managing DCE distributed applications for a variety of management tasks (performance, fault management, etc). Interactions between the management system and the managed processes are carried out through standard DCE remote procedure calls (RPCs).

### 6.2. Instrumentation Implementation

Instrumentation is provided to application developers through a C++ class library. The library contains a variety of standard sensors, actuators, and management coordinators. The library can also be specialized to develop new classes of instrumentation components, to meet the unique needs of specific applications. Figures 4, 5 and 6 show class hierarchies for sensors, management coordinators, and actuators, respectively, implemented to date. Others are currently under development.

The management coordinator in our prototype understands DCE and the management protocol (between the management system and application processes) developed as part of our previous work [8]. To facilitate this, we developed a mapping from our management protocol onto a set of interactions required with sensors and actuators. Different management systems can be supported by replacing only this mapping (other instrumentation components need not change). Coordinator routines were also developed to initialize the instrumentation, to route internal messages, and to manage sensors and actuators. A scheduler routine was added to the coordinator to handle time-based events; this was implemented as a separate execution thread.

In the current prototype, the instrumentation library includes the following categories of sensors and actuators:



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

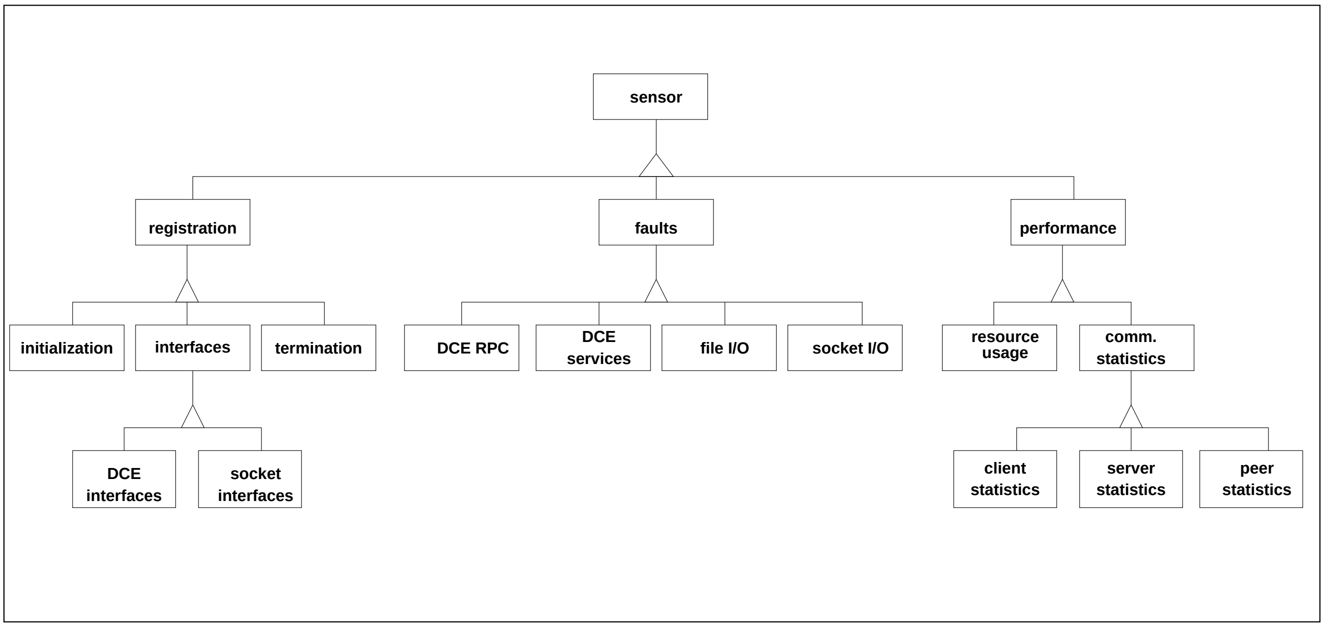


**Figure 4. Sensor class hierarchy**

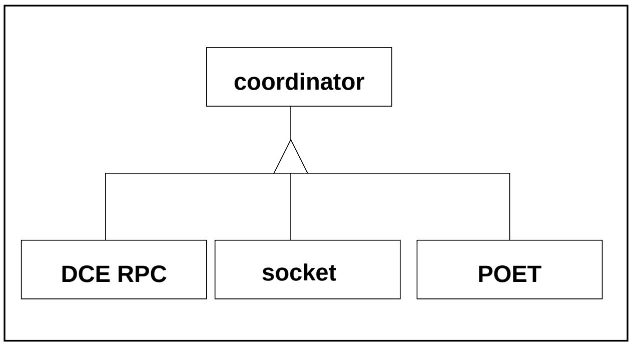


**Figure 5. Management coordinator class hierarchy (POET, Partial Order Event Tracer, is a tool for collecting and visualising event traces from the execution of distributed applications [19].)**

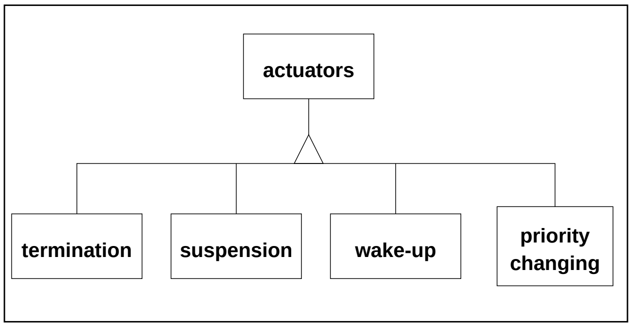


**Figure 6. Actuator class hierarchy**

• **Registration sensors** These sensors allow processes and applications to be registered with the management system so that it is aware of their existence.

• **Fault detection sensors** These sensors encapsulate information about remote procedure call timeouts and response times.

• **Resource utilization sensor** This sensor collects information that characterizes the impact a process is having on the hardware resources. This includes memory, CPU, disk, and network usage measures.

• **Communications statistics sensor** This sensor collects the data needed to characterize remote procedure call behavior including response time, number of outstanding messages, processing time, number of incoming messages, number of times a particular remote request was invoked, arrival rate, etc.

• **RPC statistics sensors** Statistical information about remote procedure calls are sent in periodic event reports to an agent, which may in turn send it to a manager process. These sensors access the appropriate information during each interval, compute the statistics, then report them to the management coordinator.

• **Process control actuators** These actuators control process termination, suspension, and priority modification.

To support the sensors and actuators discussed,



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

probes have been provided. A sensor probe updates the sensor's state (through the appropriate access function) when executed by the process. An actuator probe carries out a management task when invoked by an actuator.

In the initial prototype, probes were inserted into source code by hand and using wrapper techniques described earlier. This approach offered us the flexibility needed to experiment with the design and placement of probes for improved effectiveness and efficiency. Automation techniques are currently under investigation. With the recent availability of source code for a DCE IDL compiler, this may be a viable alternative for a revised prototype.

## 7. Concluding Remarks

The work we describe here is part of an ongoing structured attack on the distributed application management problem. There are many complex issues yet to be resolved. We believe that using a flexible and generic architecture makes it easier to address these issues and lends uniformity both to the research and to the end product.

The research detailed in this paper focuses on the instrumentation required to make distributed applications manageable. From core objectives, we have developed an architecture to support this. We have discussed several approaches to instrumentation based on this, and have concluded that, while automation and tools are required to facilitate the process, hand-coded instrumentation is still needed for custom management. Through an instrumentation prototype, we have shown that our architecture is sufficiently rich to support a wide range of management functions and environments.

We are investigating ways of bringing design for manageability into the software development process. We see the following areas as potentially fruitful.

- Developing software engineering tools to enable building instrumented applications.
- Modifying compilers to automatically instrument application code during code generation.
- Instrumenting system libraries to provide management for any application.

We plan to extend the class library to include more management coordinators, sensors, and actuators and to conduct performance analyses to determine the exact costs of our instrumentation, and to determine how to minimize them.

## References


[1] J. Cowan. *OSIMIS GDMO Compiler User Manual.* Department of Computer Science, University College London, Aug. 1993.

[2] G. Eisenhauer, W. Gu, T. Kindler, K. Schwan, D. Silva, and J. Vetter. Opportunities and Tools for Highly Interactive Distributed and Parallel Computing. Technical Report GIT-CC-94-58, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280, December 1994.

[3] R. Friedrich, J. Martinka, T. Sienknecht, and S. Saunders. Integration of performance measurement and modeling for open distributed processing. In D. K. Raymond and L. Armstrong, editors, *International Conference on Open Distributed Processing (ICODP '95)*, pages 341–352, Brisbane, Australia, Feb. 1995. "Participant's proceedings" edition.

[4] J. W. Hong, G. W. Gee, and M. A. Bauer. Towards Automating Instrumentation of Systems and Applications for Management. *Proceedings of the 1995 IEEE Global Telecommunications Conference*, Singapore, November 1995.

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

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

[7] ISO. *Information Technology - Open Systems Interconnection - Management Information Services - Structure of Management Information Part 4: Guidelines for the Definition of Managed Objects.* International Organization for Standardization, International Standard 10165-4, 1991.

[8] 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, Ontario, Canada, June 1996.

[9] M. J. Katchabaw, H. L. Lutfiyya, A. D. Marshall, and M. A. Bauer. Policy-driven fault management in distributed systems. In preparation, 1996.

[10] B. Meyer, M. Heineken, and C. Popien. Performance analysis of distributed applications with ANSAmon. In D. K. Raymond and L. Armstrong, editors, *International Conference on Open Distributed Processing (ICODP '95)*, pages 293–304, Brisbane, Australia, Feb. 1995. "Participant's proceedings" edition.

[11] Open Software Foundation Cambridge, Mass. *Introduction to OSF DCE.* Open Software Foundation, first edition, 1992.

[12] OSF. *OSF Request For Comments 32.0 - Requirements for Performance Instrumentation of DCE RPC*




Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.

*and CDS Services*. Open Software Foundation, June 1993.

[13] OSF. *OSF Request For Comments 33.0 - Standardized Performance Instrumentation and Interface Specification for Monitoring DCE-Based Applications*. Open Software Foundation, July 1995.

[14] G. Pavlou, S. N. Bhatti, and G. Knight. *The OSI Management Information Service User's Manual*. University College London, UK, Feb. 1993. Version 1.0.

[15] G. Pavlou, G. Knight, and S. Walton. Experience of implementing OSI management facilities. In *Proceedings of the Second International Symposium on Integrated Network Management*, Washington DC, Apr. 1991.

[16] J. A. Rolia. *Distributed Application Performance, Metrics and Management*. Elsevier Science B. V. (North-Holland), 1994.

[17] M. Rose. *The Open Book: A Practical Perspective on OSI*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[18] A. Tang and S. Scoggins. *Open Networking with OSI*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[19] D. J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of CASCON '93, Vol. 1, Software Engineering*, pages 505–512, Toronto, Canada, October 25–28 1993.

[20] D. J. Taylor, T. Kunz, and J. P. Black. A tool for debugging OSF DCE applications. In *Proceedings of the 12th Annual International Computer Software and Applications Conference*, pages 440–446, Seoul, Korea, Aug. 1996.



Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems (PDSE '97), Boston, Massachusetts, May 17-18, 1997, pages 84-94.