ELSEVIER

The Journal of Systems and Software 45 (1999) 81–97

The Journal of Systems and Software

Making distributed applications manageable through instrumentation [1]

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

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. The management software will, within a policy framework, monitor, analyse and control network resources, system resources, and distributed application programs.

In our research, we are primarily concerned with the management of distributed applications. Of particular interest is how distributed application processes can be made manageable. The work described in the current paper focusses on instrumenting these 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, and discuss a prototype implementation based on it. This prototype was used to experiment with the management of DCE applications in an OSI-based management environment.

Keywords: Instrumentation; Distributed applications management; OSI management framework; DCE

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 present many difficult challenges. As distributed systems grow to hundreds or even thousands of devices and similar or greater numbers of software components, it will become increasingly difficult to locate faults, determine bottlenecks, anticipate problems, or even determine how distributed applications are behaving. Doing so, however, is critical to ensuring their reliability and performance. It is necessary to provide the means to detect violations of management policies in these applications, locate their causes, and then perform the actions required to correct the problems and recover from them. To do this, management of distributed applications is essential.

Application manageability is a research issue of particular interest to us. While some other work has been done in this area (Eisenhauer et al., 1994; OSF, 1993, 1995), 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.

Our approach to distributed applications 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 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 behaviour for such things as performance (Rolia, 1994), visualization (Taylor et al., 1996), and reliability (Katchabaw et al., 1996c). We believe the question is

* Corresponding author. Tel.: +1 519 679 2111x6888; fax: +1 519 661 3515; e-mail: hanan@csd.uwo.ca

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

0164-1212/99/$ – see front matter
PII: S0164-1212(98)10070-5

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) (OSF, 1992) applications in a prototype environment under the Open Systems Interconnection (OSI) Management Framework (ISO, 1991a, b).

The rest of the paper is organized as follows. Section 2 outlines some of the major objectives guiding the current work. Section 3 describes the management environment. Section 4 presents the instrumentation architecture developed to meet these objectives. Section 5 addresses the instrumentation procedure, focussing on where instrumentation should be placed, and how this should be done. Section 6 describes a prototype utilizing the instrumentation architecture and our experience with the prototype. Section 7 presents related work. Section 8 concludes with a summary of the project status and directions for future work.

2. Objectives

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

Data acquisition and control: For effective management, we must be able to acquire information on the processes being managed. This includes both generic and application-specific management information. Traditionally, this information is obtained using the *pull* (polling) model, where explicit requests are issued from the management system to the managed processes. While this method is suitable for management inquiries, a *push* (event-driven) model is more effective for the submission of periodic status reports and for notifying the management system of exceptional conditions.

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 behaviour. Consequently, both forms of management interactions must be supported.

Dynamic management: It is important that management be dynamically ''tunable'' so its behaviour can be adjusted to meet operational needs. We must be able to adjust the level of data collected (which attributes, how often, and so on), to define what processing is to be done on collected data, to establish thresholds used to generate alarm reports, and to disable management altogether.

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 performance, accounting, fault tolerance, and security. Finally, this architecture must also co-exist with current work in this area.

Transparency: The development process for distributed applications is a difficult one; the additional burden of instrumentation must be minimized. The instrumentation architecture should support the automation of the instrumentation process, so it can be done with little developer effort. Providing a standard set of instrumentation would be helpful. (Naturally, to support flexibility, custom instrumentation should also be supported.) 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 applications management system. The framework is based on the OSI Management framework (Rose, 1990; Tang and Scoggins, 1992). Fig. 1 illustrates the basic structure and interactions within this framework.

Management systems contain three types of components that work together: *managers*, which make decisions based on collected management information guided by management policies; *management agents*, which collect management information; and *managed objects*, which represent actual system or network resources being managed.

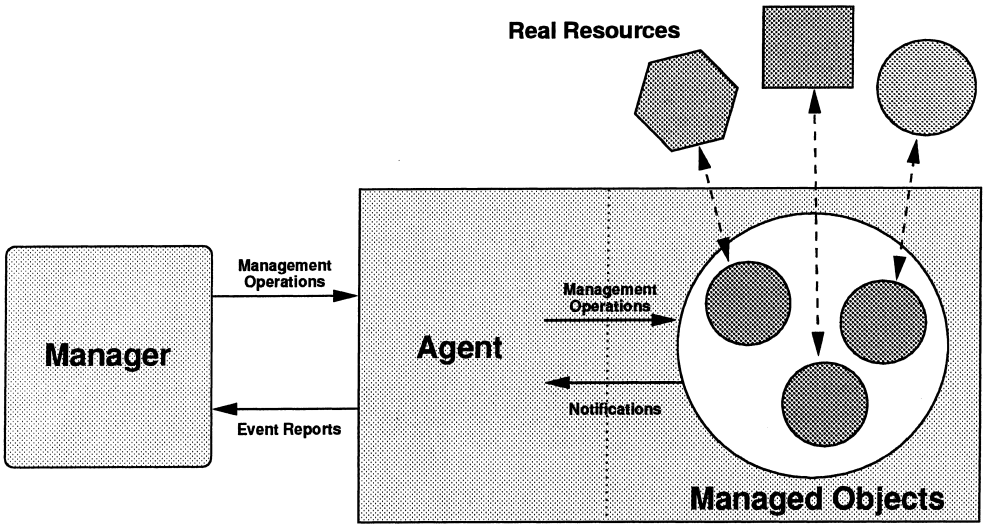

Fig. 1. Management framework.

A *managed object* is an abstraction of one or more real resources. A managed object 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 agent* is responsible for a particular collection of managed objects. The agent and its managed objects serve to decouple managers from their managed resources. This approach has many advantages. It facilitates many-to-many relationships between management applications and managed resources. It provides a means to distribute the management function: localized 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 appropriate management applications. Management agents perform operations requested by managers and notify managers of pre-determined events of interest to the manager.

Fig. 2 illustrates the basic components of an agent architecture and how they interact with managed processes. The most relevant components of the architecture are described below.

Agent coordinator: The role of the agent coordinator is to coordinate activities taking place within the agent. This entails the internal routing of requests and notifications to the appropriate components and the synchronization of access to the managed objects.

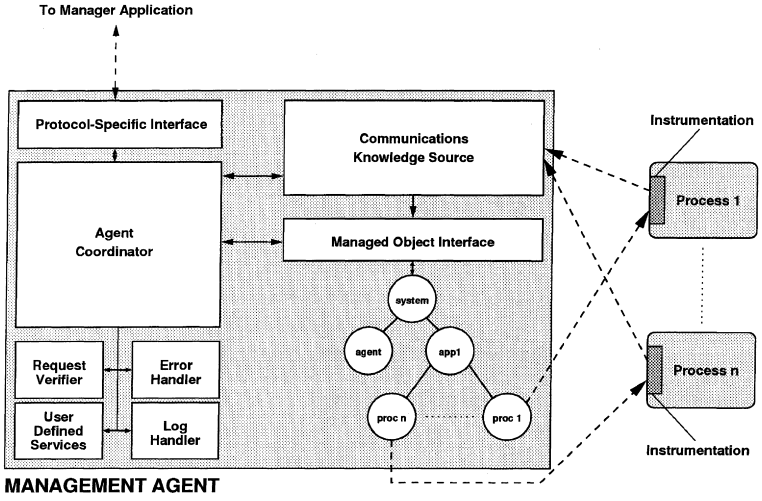

Fig. 2. Agent architecture.

Communications knowledge source: The communications knowledge source provides an interface for management information and event notifications from the instrumented application. The knowledge source contacts the agent coordinator to synchronize exclusive access to managed objects and performs operations on managed objects through the managed object interface. Knowledge sources must be tailored to the communication mechanism used by the managed applications (for example, a DCE knowledge source is required to interact with DCE applications).

Managed object interface: The managed object interface is used to coordinate requests on managed objects and to handle emitted notifications.

Managed objects: A distributed application is modeled by a single Application managed object containing (in the object-oriented sense) one or more process managed objects. Process managed object classes are specialized through inheritance to reflect their unique characteristics including communication environment (sockets, DCE Remote Procedure Calls (RPCs), etc.), process role (client, server, or peer), and application type (database, etc.). Each process managed object has attributes for its name, host, process id, priority, status, resource utilization, communication statistics, and so on. These attributes depend on the specialization of the process managed object class (for example, DCE client processes have different performance metrics than DCE server processes). Process managed objects use an appropriate mechanism to communicate with instrumented application processes in the system (for example, DCE RPC would be used to communicate with DCE application processes).

Instrumentation is required in application processes to make them manageable. This includes a mechanism which "listens" for incoming management requests from agents and another to send management information and event reports to agents. In Section 4.2 we will describe the agent interface that is used by the instrumented application processes as well as the interface of the instrumented processes to be used by agents.

4. Instrumentation architecture

Having identified the key objectives of instrumentation for managing distributed applications, we refine the management architecture developed in our previous work (Hong et al., 1995b, c; Katchabaw et al., 1996a) to reflect our current focus on instrumentation. In Section 4, we describe the key components of the architecture and the interactions that occur between these components.

4.1. Instrumentation component overview

In Fig. 3, Instrumented Process 1 has been expanded to reveal a set of instrumentation components. This instrumentation architecture is the focus of the current research. While it is useful to abstract instrumentation components

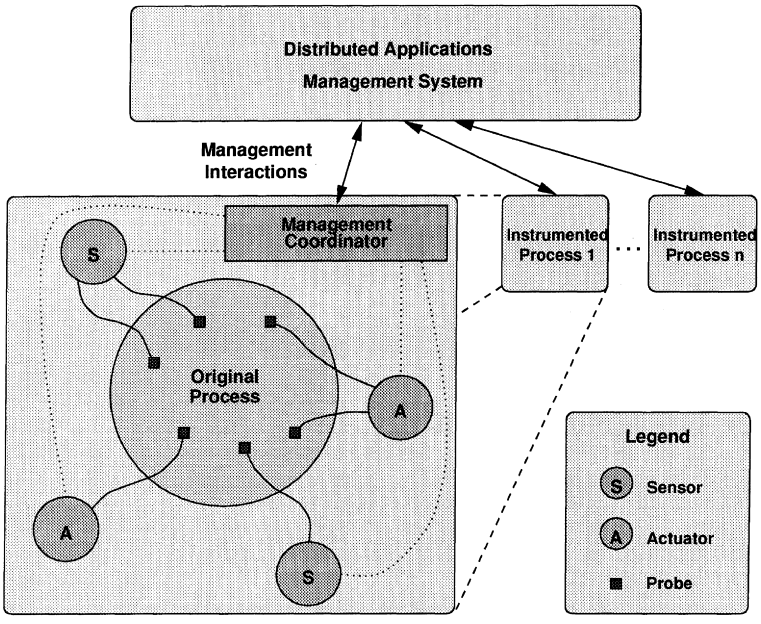

Fig. 3. Instrumentation architecture and environment.

into distinct entities to support greater flexibility and modularity, in practice, we recognize that instrumentation will be most likely to reside in the address space of the original application process. This is necessary to facilitate efficient access to management information and efficient control over the managed process. The proposed instrumentation components are described below.

Instrumented process: The application process, complete with embedded instrumentation, comprises an instrumented manageable process.

Management coordinator: The management coordinator is an instrumentation component that facilitates communication between the management system and an instrumented process. Its role includes message routing for requests, replies, and reports flowing between the management system and instrumentation sensors and actuators. The coordinator is also responsible for creating and destroying sensors and actuators, as well as performing management initialization and termination activities within the managed process. Management requests to dynamically modify the behaviour of sensors and actuators are also handled by the coordinator.

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 (e.g., OSI, Simple Network Management Protocol (SNMP), Common Object Request Broker Architecture (CORBA)) allowing other instrumentation components (sensors, actuators and probes) 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 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 process 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 behaviour 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.

Actuators: Actuators encapsulate management functions which exert control over the managed process to change its behaviour. 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 behaviour control operations; for example, to enable or disable the actuator.

As for sensors, it is anticipated that application developers will view actuators as reusable parts which can be plugged into an application process via one or more probes to provide specific control functions. 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 instrumentation components 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 behaviour. Each probe is specific to a particular sensor or actuator. Sensor probes are 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 the managed process (through the management coordinator) and the agent. We will first present the interface of the agent that is used by the managed process. We will then present the interface of the managed process used by the agent.

4.2.1. Managed process to agent

Acquiring management information is critical to the management of distributed applications. One method of acquiring management information is through the use of *event reports*. The flow of information is from the processes to the agents and then on to the managers, if the managers are interested in the information. We must be able to support two types of flow of information: (i) the periodic sending of management information to the agent, and (ii) the ability to send information when a significant event occurs. Significant events includes process start-up and termination and those events that affect the normal operation of a process. We present below a programming interface to support managed process to agent communication. In all of these operations, the `Status` attribute is used to indicate the success of the operation.

Process_sendManagementInformation: This service is used by managed processes to report management information to a management agent.

```
Process_sendManagementInformation(          [in] long_int           ProcessId,
                                            [in] attributeInfo      Attributes[],
                                            [out] statusCode        Status);
```

`ProcessId` identifies the process sending the information.
`Attributes` is the list of attributes to send.

Process_notifyRegister: This notifies the agent of the existence of a new process, and provides basic information about it, including its name, identifier, host, presentation address, etc.

```
Process_notifyRegister(             [in] string             AppId,
                                    [in] string             AppName,
                                    [in] string             AppVersion,
                                    [in] process            Type ProcessType,
                                    [in] string             ProcessName,
                                    [in] long_int           ProcessId,
                                    [in] integer            ProcessNumParms,
                                    [in] process_Parms      ProcessParms[],
                                    [in] string             ProcessHost,
                                    [out] statusCode        Status);
```

`AppId`, `AppName`, `AppVersion` represent the application identifier, name and version of the application that the process belongs to.
`ProcessType` represents the type of process. For example, a process may act as a client, or a server, or in both roles.
`ProcessName`, `ProcessId` represent the process name and identifier.
`ProcessNumParms` is the number of parameters that the process was passed when it was started.
`ProcessParms` is the list of parameters that the process was passed when it was started.
`ProcessHost` is the name of the host that the process was started on.

Process_notifyTermination: Notifies the agent of the termination of the process in the context of the normal operation of the application.

```
Process_notifyTermination(                  [in] long_int           ProcessId,
                                            [out] statusCode        Status);
```

Process_notifyUnmanage: Notifies the agent of the desire of the process not to be managed.

```
Process_notifyUnmanage(                 [in] long_int           ProcessId,
                                        [out] statusCode        Status);
```

Process_sendNotification: When significant events occur that affect the normal operation of the application, alarm reports can be generated by the managed process to the agent using operations such as the following:

Process_notifyCommunicationsFailure: Notifies the agent that a request from one application process to another failed due to a communication error.

```
Process_notifyCommunicationsFailure(        [in] long_int       ProcessId,
                                            [in] unsigned long  FailureCode,
                                            [in] string         ServerType,
                                            [in] string         ServerHost,
                                            [in] string         ServerProtocol,
                                            [in] string         Endpoint,
                                            [out] statusCode    Status);
```

`ProcessId` is the process identifier.
`FailureCode` specifies the type of failure observed.
`ServerType` specifies the type of server process that failed.
`ServerHost` specifies the host of the server process that failed.
`ServerProtocol` specifies the protocol being used to communicate with the server.
`Endpoint` specifies the port number of the server process that failed.

Process_notifyCongestion: Notifies the agent that a server process is experiencing congestion due to over-utilization and that the expected quality of service has degraded. This is similar to `Process_notifyCommunications-Failure` with the addition of attribute `CurrentRate` that is used to specify the average time for a remote procedure call.

```
Process_notifyCongestion(           [in] long_int           ProcessId,
                                    [in] unsigned long      FailureCode,
                                    [in] string             ServerType,
                                    [in] string             ServerHost,
                                    [in] string             ServerProtocol,
                                    [in] string             Endpoint,
                                    [in] double             CurrentRate
                                    [out] statusCode        Status);
```

4.2.2. Agent to managed process

Process_requestManagementInformation: This service is used by the management agent to request information from managed processes in the system. By specifying the set of attributes to retrieve, the appropriate information is returned to the agent.

```
Process_requestManagementInformation(       [in] attributeId        AttributeIds[],
                                            [out] attributeInfo     Attributes[],
                                            [out] statusCode        Status);
```

`AttributeIds` specifies the attributes whose values are to be reported to the agent.

Process_performControlAction: The following operations are used by the management agent to perform a control action on a managed process that changes the operation of the process relative to its distributed application. These actions can be used to terminate processes, suspend processes, awaken processes, change the priorities of processes, and other more application-specific actions. To do this, the agent specifies the action to perform, and input parameters for the action. In return, the appropriate output parameters for the action are returned.

Process_requestTermination: This service instructs the process to terminate as quickly as possible.

```
Process_requestTermination([out] statusCode Status);
```

Process_changePriority: This service instructs the process to change its run-time priority.

```
Peocess_changePriority(          [in] long_int          NewPriority,
                                 [out] statusCode       Status);
```

Control of the management of processes: To manage a distributed application effectively, we must be able to change the way management is done at run-time in order to react to operational needs. Operations include the following:

Process_controlManageProcess: Activates management in a process, informing the process which agent will be managing it.

Process_controlUnmanageProcess: Deactivates management in a process that no longer needs to be managed.

Process_controlChangeManagementSettings: Used to tailor the event reports generated by the process. For periodic reports, this operation can change the interval between reports. For example, consider the following:

```
Process_changeCongestLimit(          [in] double            NewThreshold,
                                     [out] statusCode       Status);
```

`NewThreshold` is the new threshold limit for remote procedure call durations.

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 Fig. 3 as they relate to these interactions.

4.3.1. Management system to instrumentation interactions

Requests from the management system to an instrumented process are received by its management 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 state for the sensors is updated by the sensors' probes, or through interactions with other sensors. The sensors then report their state, perhaps after some processing. 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 behaviour 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, the process creates a management coordinator within itself. This 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 of 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 is about to terminate, 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

So far in this paper we have described *what* is needed for instrumentation. Before turning to the question "how can a process be instrumented for management"? we must first answer the question "where must instrumentation be placed in the instrumented process"?

5.1. Placing instrumentation

For effective management of a distributed application process, instrumentation probes must be inserted at strategic locations (probe points) in the process 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 placed within the original code of the process. Instead, they can simply be linked into the process at compile time through an instrumentation library.

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

Entry points: Every process entry point must be instrumented with probes to allow instrumentation to initialize and to allow process registration to occur. These include process start-up (for example, in the main function) and after a process forks or a new thread is created.

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 point at which an exit function is called, the main function return, and within signal handlers.

Inter-process communication: Whenever one process communicates with another process in a distributed application, this is considered to be a significant event. At these points, performance metrics can be computed, faults can be detected, accounting can be 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 instrumentation probes to monitor and control this activity. This includes operations associated with memory allocation and deallocation, time services, file services, naming services, binding services, and so on. Once again, metrics can be computed, accounting can be performed, and violations can be reported. 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 be instrumented with probes.

Custom points: When custom-developed sensors or actuators are used, the locations for the insertion of their probes will depend largely on the application. Similarly, a developer could choose to use a standard sensor or actuator in a non-standard location.

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

In the previous section, we described the process of instrumentation and identified the strategic points in source code where probes should be placed. In this section, we examine several instrumentation techniques and the relative merits of each. Fig. 4 illustrates the alternative approaches we consider.

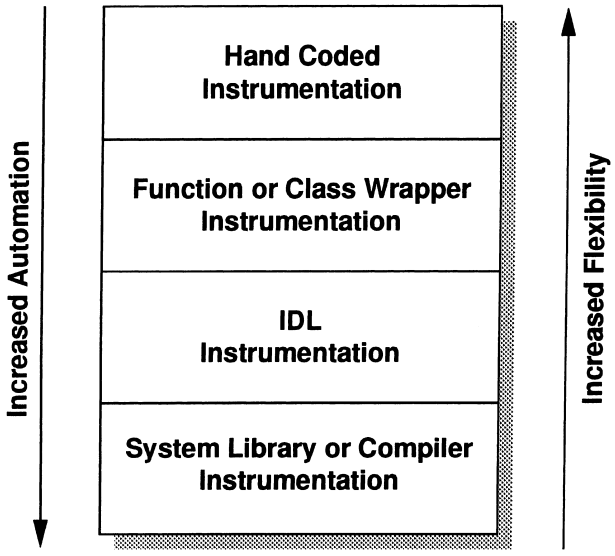

Fig. 4. Instrumentation techniques.

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

This 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 (Hong et al., 1995a).

5.2.2. 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 (although its name is probably still the same), with the same result produced. The inserted probes could do things like time the execution of *f*, add to a counter of the number of times *f* is used, check the return value of *f* for possible failures, and so on. The same concept applies to wrapping data structures and objects.

Wrappers can help automate basic instrumentation and reduce somewhat the effort and risk of human error. It makes most sense under this scenario for wrapped functions, data structures, and objects to be packaged into libraries for use by developers. Flexibility is reduced somewhat, since all uses of a particular function, data structure or object will employ the same set of probes. Consistency across all applications using the pre-wrapped elements is increased.

5.2.3. IDL instrumentation

Many distributed application 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 instrumentation 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.

5.2.4. System library or compiler instrumentation

A fourth approach hides most of the instrumentation procedure from the application developer. 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. Instrumenting system libraries has the additional benefit of inserting probes for exception and signal handling routines, exit and entry points, as well as for operating system and middleware services. This could not otherwise be done by the developer alone.

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 (OSF, 1993, 1995); 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, as it seems that no single method meets all requirements. Clearly, there is a need for automation to reduce the effort and inherent risk of instrumentation by hand. At the same time, the flexibility to allow developers to pursue the unique management needs of their applications is critical. By providing compilers and instrumented libraries to automate the task for standard instrumentation, and then additional tools to assist in developing custom instrumentation, we might get the best of both worlds.

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) (Pavlou et al., 1993, 1991) which provides an object-oriented infrastructure for developing management applications and OSI agents. OSIMIS includes a compiler (Cowan, 1993) to parse managed object class definitions (specified in the Guidelines for the Definition of Managed Objects (GDMO) (ISO, 1991c)) 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.

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. Figs. 5–7 show class hierarchies for management coordinators, sensors, and actuators, respectively, implemented to date. Others are currently under development.

6.2.1. Management coordinator

Interactions between the management system and the managed processes can be carried out using different communication protocols that include standard DCE RPCs and sockets.

Different management systems can be supported by replacing the management coordinator. For example, if the management coordinator uses DCE for communication between the instrumented processes and management agents and then changes to sockets then we would only have to replace the management coordinator; the sensors and actuators do not change. Coordinator routines have also been developed to initialize the instrumentation, to route internal messages, and to manage sensors and actuators. A scheduler routine in the coordinator handles time-based events; this has been implemented as a separate execution thread.

6.2.2. Sensors and actuators

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

Registration sensors allow processes and applications to be registered with the management system so that the system is aware of their existence. Registration also permits management interfaces to be specified to the management system to enable the system to communicate with registered processes and applications.

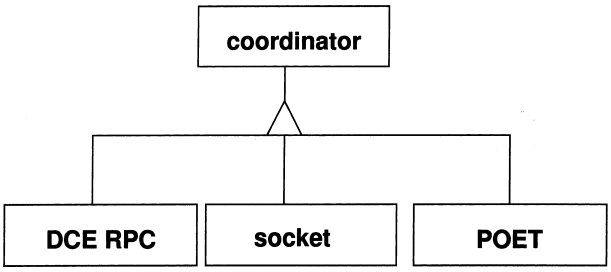

Fig. 5. Management coordinator class hierarchy (POET, Partial Order Event Tracer, is a tool for collecting and visualizing event traces from the execution of distributed applications, Taylor, 1993).

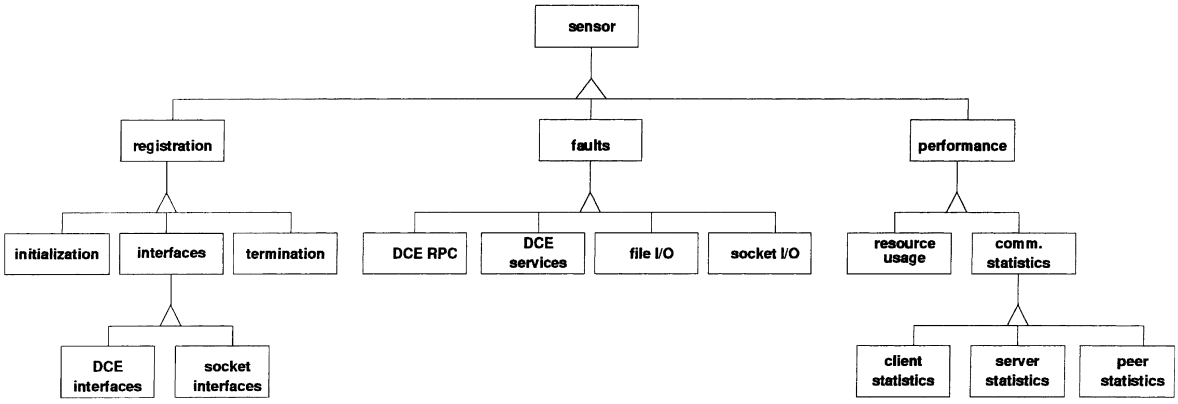

Fig. 6. Sensor class hierarchy.

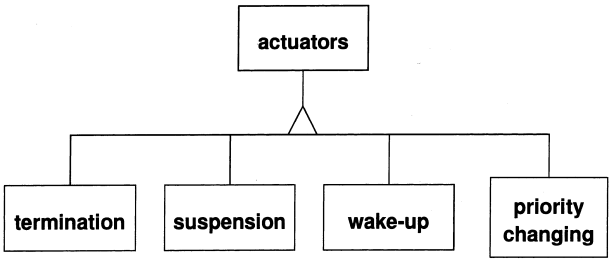

Fig. 7. Actuator class hierarchy.

Fault detection sensors allow processes to report faults detected during their operation. These include remote procedure call timeouts, server congestion, middleware service failures, failed file or network operations, and other abnormal and undesirable conditions.

Resource utilization sensors collect information that characterizes the impact a process is having on system hardware resources, including memory, CPU, disk, and network usage measures.

RPC statistics sensors compute RPC statistics (e.g., average service time) at the end of each reporting interval, then report them to the management coordinator.

Process control actuators control process termination, process suspension, process priority modification and change the length of the interval between RPC statistics event reports.

6.2.3. Probes

The following probes were inserted into the source code of distributed applications by hand.

1. *Process_instrumentationInit*. For a process to be manageable, it is necessary that the process entry point have this instrumentation probe. It does the following:

Retrieve the binding handle of the agent. The *binding handle* contains the information needed by a process to establish communication with the appropriate agent. Since we are using DCE we retrieve the binding handle of the

agent using the Cell Directory Service (CDS). The CDS is a service that allows processes (including agents) to register their interfaces.

Registration. The agent is notified of the existence of a new process and is provided with information about it.

Creates the Management Coordinator and Sensors. A management thread is created that becomes the management coordinator and the sensors.

2. *Process_rpcRequestBegin and Process_rpcRequestEnd*: These are inserted before and after each remote procedure call. They update the resource utilization, RPC statistics and fault detection sensors.

3. *Process_instrumentationShutdown*. This probe notifies an agent of the termination of a process.

Hence, for any manageable process, the code for the process will exhibit a form similar to this:

```
...
#include ''app_instrumentation.h''
Process_instrumentationInit();
...
Process_rpcRequestBegin()
application remote procedure call();
Process_rpcRequestEnd()
...
Process_instrumentationShutdown();
```

All the developer sees are these probes. The management coordinator, the sensors and the actuators are hidden from the developer. The developer must remember to include `app_instrumentation.h` and to link in the instrumentation library.

6.3. Managers

To validate our instrumentation architecture we implemented, as part of the prototype system, several managers.

Event visualization: POET, Partial Order Event Tracer (Taylor, 1993), is a tool for collecting and visualizing event traces from the execution of distributed applications. While POET is especially useful in debugging, the event traces generated using our instrumentation also assist in visualizing and understanding the interactions that occur in complex distributed applications.

Performance management: (Rolia et al., 1995; Rolia and Sevcik, 1995; Sun et al., 1997; Sun, 1997; Woodside et al., 1995) Delays caused by poor performance at the application level or network level can seriously affect the usability and effectiveness of a distributed application, or an entire distributed environment. Both application developers and managers of a distributed system must therefore take steps to ensure that their systems are performing within acceptable bounds.

To that end, we have developed two performance-related managers. One manger is a tool that provides a visual picture of remote procedure call performance based on the data collected by the RPC statistics sensor. The other builds predictive performance models for distributed application systems to be used by distributed application developers and performance management staff to make quantitative comparisons between software design and system configuration alternatives.

Automating fault location: (Turner, 1995) One important aspect of distributed applications management is fault management – detecting that the behaviour of an application has deviated from the specification of its desired behaviour. This deviation is referred to as a *failure*, and is manifested through observed *symptoms*. At the source of a failure is a *fault* (or possibly several faults). Symptoms alone do not provide enough information to allow the fault to be corrected: many faults may give rise to the same symptom.

We have developed a manager, the Fault Management Tool (Turner, 1995), that automates the process of fault location in distributed application. Application processes are instrumented with fault detection sensors (examples of which were given in Section 6.2.2). Thresholds and other settings on these sensors are used to reflect the specification of the application's desired behaviour so that when the application deviates from this behaviour, a sensor is triggered and generates a symptom report. The Fault Management Tool receives these symptoms from application processes, filters and correlates the symptoms, and uses a variety of techniques to locate the faults responsible for the deviant application behaviour.

Configuration maintenance: (Lutfiyya et al., 1997) Most manager applications, such as those described above, require access to information on computing devices, networks, system resources and services, and user applications in the environment they are managing. As a result, a configuration maintenance service is required to support management.

To this end, we have developed a Configuration Maintenance Service (Lutfiyya et al., 1997) that enables management applications and other management services to retrieve and update configuration information stored in a

Management Information Repository (MIR). Registration sensors, such as those discussed in Section 6.2.2, are used by distributed applications to register, update, and remove their configuration information with the Configuration Maintenance Service. In addition, tools are also provided to exert rudimentary control over registered applications using actuators also discussed in Section 6.2.2.

6.4. Evaluation

Having designed and developed prototypes of our generic architecture in Section 4, it is important to evaluate this work in light of our objectives given in Section 2.

Data acquisition and control: To meet this objective, we must support generic and application-specific management, both pull and push models of acquiring management information, and both the collection of data and the exertion of control over application processes.

To date we have implemented several sensors and actuators, as discussed in Section 6.2.2, to facilitate data acquisition and process control. We have also provided a library of instrumentation components to support generic management of applications, and derived components specific to the needs of certain applications (DCE distributed applications). Furthermore, most of the data acquisition done by our instrumentation is according to the push model for efficiency reasons, but the pull model is also supported to permit poll-driven management.

As a result, we have met this objective with our generic architecture, and we have successfully validated it through our prototypes.

Dynamic management: To meet this objective, we must allow our management to be dynamically tunable to adjust how management is being done. This includes enabling and disabling management, as well as control over what is being done.

All sensors and actuators that have been developed can be enabled and disabled easily, allowing the amount of data acquisition and control to adapt to reflect the needs of management. Implemented sensors also allow reporting intervals and thresholding to be changed. While sensors that support a variety of data processing options have yet to be implemented, this is supported by our generic architecture.

As a result, we have met this objective with our generic architecture, and we have successfully validated it through our prototype implementation.

Flexibility: To meet this objective we must not make assumptions about the platforms we are dealing with. At the same time, we must be open in supporting a wide range of management functions, as well as current work and standards in this area.

The architecture that we have developed is platform-neutral. Our management work has currently been implemented on a variety of hardware architectures including IBM RS/6000, Sun UltraSparc, and Sun SPARCstation, and operating systems including AIX, Solaris, and SunOS. We also support several communication infrastructures, including TCP/IP sockets and DCE RPC. Work is currently under way to support even more environments.

Our work also supports a variety of management functions, with sensors and actuators to facilitate fault management, performance management, configuration management, and accounting for system resources. Our generic architecture does not limit us to only these areas – we are currently developing security sensors, reconfiguration sensors, quality of service sensors, and components to support other management functions. Our architecture is open and can be easily integrated with other management solutions. Our prototype systems have been integrated easily with other management solutions we have developed, OSI-compliant management solutions, as well as the work of other universities under the MANagement of Distributed Applications and Systems (MANDAS) project (Bauer et al., 1997). This integration was simplified as we only needed to specialize coordinators to communicate with the various management systems, while other instrumentation components remained unchanged.

As a result, we have met this objective with our generic architecture, and we have successfully validated it through our prototypes.

Transparency: To meet this objective, we must make instrumentation easy on both application developers and on the instrumented application processes. Creating manageable applications with instrumentation should not be a burden, nor should the instrumented application processes be a burden on scarce system resources due to excessive management overhead.

Our generic instrumentation architecture supports the use of automation in instrumenting applications through a variety of techniques outlined in Section 5. While the majority of our instrumentation is currently done by hand or through function or class wrappers, we are currently developing a variety of tools to facilitate the instrumentation process.

To assess the overhead of our management instrumentation, we have developed a Distributed Applications Management Testbed (Katchabaw et al., 1996b). This testbed facilitates experiments measuring the costs of management. From our initial experiments, documented in (Katchabaw et al., 1996b), we found that the use of our

management instrumentation on average caused less than a 5% drop in application responsiveness. On the other hand, in the same study, we found that application resource consumption could increase from between 7% to 43%, depending on the resource. Through further experimentation, however, we were able to determine a variety of techniques to reduce the costs of management, such as using a push model for data acquisition and batching management requests when the pull model must be used.

This objective is supported well by our generic architecture, and we are making progress towards meeting it in our prototypes.

In summary, we have been able meet almost all of our objectives through both our generic architecture and our prototypes, and we are close to meeting the remaining objective. Our work makes several valuable contributions not present in related work in this area, as discussed in the next section.

7. Related work

Many parallel programming tools use instrumented software to capture data about the run-time operation of parallel programs. Some of these tools are described below.

- Portable Instrumented Communication Library (PICL) (Geist et al., 1991; Heath and Etheridge, 1991) instrumentation supports program performance analysis and animation. In order to instrument an application program, PICL library functions are inserted in the program by the user before compilation. During program execution, calls to these functions generate instrumentation data in a particular event record format and log the data in a local buffer of each node. The user specifies the size of the buffer. These buffers are typically flushed at the end of program execution and merged into a single trace file at the host system.
- AIMS (Automated Instrumentation and Monitoring System) (Yan, 1994) is a toolkit consisting of an instrumentation library and a set of off-line performance analysis and visualization tools. Its instrumentation support is almost identical to that of PICL.
- Pablo (Reed et al., 1992) is an integrated tool environment that offers three types of performance data capturing functions: (1) event tracing; (2) event counting; and (3) code profiling. If a local buffer is full, all buffers can be flushed synchronously to a file or to an Internet domain socket. Unlike PICL and AIMS instrumentation, Pablo's instrumentation supports adaptive levels of tracing to dynamically alter the volume, frequency and types of event data recorded.
- Paradyn (Miller et al., 1994) is an on-line performance evaluation environment that is based on dynamically updating the cumulative-time statistics of various performance variables. The instrumentation provides the data needed for the performance evaluation. The instrumentation is equipped with the capability to estimate its cost to the application program. This cost model is continuously updated in response to actual measurements as an instrumented program starts executing and the model attempts to regulate the amount of instrumentation overhead to the application program.
- Falcon (Gu et al., 1994) is an application-specific, on-line monitoring and steering system for parallel programs. Instrumentation supports dynamic control of monitoring overhead to reduce the latency between the time an event is generated and the time it is acted upon for the purpose of steering. Various modules and functions of the instrumentation are specified by a low-level sensor specification language and a higher level view specification language.
- ParAide (Ries et al., 1993) is the integrated performance monitoring environment for the Intel Paragon. Commands are sent to the distributed monitoring system, called Tools Application Monitor (TAM). TAM consists of a network of TAM processes arranged as a broadcast spanning tree with one TAM process at each node. This configuration allows broadcasting monitoring requests to all nodes. Instrumentation library calls generate data that are sent to the event trace servers, which perform post-processing tasks and write the data to a file or send them directly to an analysis tool.
- Scalable Parallel Instrumentation (SPI) (Bhatt et al., 1995) is a real-time instrumentation system for heterogeneous computer systems. SPI supports an application-specific instrumentation development environment, which is based on an event-action model and an event specification language.
- VIZIR (Hao et al., 1995) is another integrated tool environment used for debugging and visualizing of a workstation cluster. This environment utilizes commercially available debuggers and visualization tools. This environment is an example in which instrumentation support has been used to integrate heterogeneous tools.

Unfortunately, much of this work suffers from common limitations. Several of the tools listed above are limited to a particular platform or environment, and not easily ported for use elsewhere. The majority of the tools involve only measurement and data acquisition, and do not support the control aspect of management we discussed earlier. Many of the tools cannot accommodate the spectrum of management functions previously listed in this paper.

In addition to related work in parallel environments, other work in this area is more specific to distributed applications. This includes the following tools:

- Meta Toolkit (Marzullo et al., 1991) – This toolkit is a system for managing distributed applications developed using the Isis distributed programming toolkit (Birman and Cooper, 1990).
- Huang and Kintala Tools (Huang and Kintala, 1993) – This set of tools provides services for detecting whether a process is alive or dead; specifying and check-pointing critical data; recovering check-pointed data; logging events; locating and reconnecting to a server; and replicating user-specified files on a backup host.
- MAL (Trommler et al., 1995) – This work focusses on instrumentation of distributed applications for management.

The Meta Toolkit is meant to be used within a specialized environment that is provided by Isis. Our work is more general in that we do not make any particular assumptions about the underlying environment. The services provided by the Huang and Kintala tools can be implemented by an agent, or by sensors and actuators. MAL is similar to our work except that we further developed an architecture for instrumentation that includes both sensors and actuators.

8. Concluding remarks

The work we describe here is part of an ongoing structured attack on the distributed applications management problem.

The research detailed in this paper focusses on the instrumentation required to make distributed applications manageable. From core objectives, we have developed an architecture. 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.

- Further evaluating the instrumentation interfaces, sensors and actuators by supporting more management applications.
- 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.
- Developing an information model and repository services for management components to support searching, browsing, and other queries.

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

References

Bauer, M.A., Bunt, R.B., Rayess, A.E., Finnigan, P.J., Kunz, T., Lutfiyya, H.L., Marshall, A.D., Martin, P., Oster, G.M., Powley, W., Rolia, J., Taylor, D., Woodside, M., 1997. Services supporting management of distributed applications and systems. IBM Systems Journal 36, 508–526.

Bhatt, D., Rakesh, J., Steeves, T., Bhatt, R., Wills, D. 1995. SPI: An Instrumentation Development Environment for Parallel/Distributed Systems. In: Proceedings of Int. Parallel Processing Symposium.

Birman, K., Cooper, R., 1990. The Isis Project: Real Experience with a Fault Tolerant Programming System, Technical Report TR90-1183, Department of Computer Science. Cornell University, Ithaca, NY.

Cowan, J., 1993. OSIMIS GDMO Compiler User Manual (Department of Computer Science. University College, London, UK.

Eisenhauer, G., Gu, W., Kindler, T., Schwan, K., Silva, D., Vetter, J., 1994. 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.

Geist, G., Heath, M., Peyton, B., Worley, P., 1991. A User's Guide to PICL, Technical Report ORNL/TM-11616, Oak Ridge National Liboratory.

Gu, W., Eisenhauer, G., Kramer, E., Schwan, K., Stasko, J., Vetter, J., 1994. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs, Technical Report GIT-CC-94-21, Georgia Institute of Technology, College of Computing, Atlanta, GA 30332-0280.

Hao, M., Karp, A., Waheed, A., Jazayeri, M., 1995. VIZIR: An integrated environment for distributed program visualization. In: Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'95) Tool Fair.

Heath, M., Etheridge, J., 1991. Visualizing the performance of parallel programs. IEEE Software 8, 29–39.

Hong, J.W., Gee, G.W., Bauer, M.A., 1995a. Towards automating instrumentation of systems and applications for management. In: Proceedings of the 1995 IEEE Global Telecommunications Conference.

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

Hong, J.W., Katchabaw, M.J., Bauer, M.A., Lutfiyya, H., 1995c. Modeling and management of distributed applications and services using the OSI management framework. In: Proceedings of the International Conference on Computer Communication.

Huang, Y., Kintala, C.,1993. Software Implemented Fault Tolerance: Technologies and Experience. In: Proceedings of the 23rd International Symposium on Fault Tolerant Computing, pp. 2–9.
ISO, 1991a. Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework, International Organization for Standardization, International Standard 7498-4.
ISO, 1991b. Information Processing Systems – Open Systems Interconnection – Systems Management Overview, International Organization for Standardization, International Standard 10040.
ISO, 1991c. 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.
Katchabaw, M.J., Howard, S.L., Lutfiyya, H.L., Bauer, M.A., 1996a. Efficient management data acquisition and run-time control of DCE applications using the OSI management framework. In: Proceedings of the Second International IEEE Workshop on Systems Management.
Katchabaw, M.J., Howard, S.L., Marshall, A.D., 1996b. Bauer, M.A., Evaluating the costs of management: A distributed applications management testbed, in: Proceedings of the 1996 CAS Conference, Toronto, Canada, pp. 29–41.
Katchabaw, M.J., Lutfiyya, H.L., Marshall, A.D., Bauer, M.A., 1996c. Policy-driven fault management in distributed systems. In: Proceedings of the International Symposium on Software Reliability Engineering, White Plains, New York.
Lutfiyya, H.L., Marshall, A.D., Bauer, M.A., Martin, P., Powley, W., 1997. Configuration maintenance for distributed applications management. In: Proceedings of the 1997 CAS Conference, Toronto, Canada, pp. 43–57.
Marzullo, K., Cooper, R., Wood, M.D., Birman, K.P., 1991. Tools for distributed application management. IEEE Computer 25, 42–51.
Miller, B., Cargille, J., Irvin, R., Kunchithapadam, K., Callaghan, M., Hollingsworth, J., Karavanic, K., Newhall, T., 1994. The Paradyn Parallel Performance Measurement Tools, Technical report, University of Wisconsin.
OSF, 1992. Introduction to OSF DCE, 1st ed. Open Software Foundation.
OSF, 1993. OSF Request For Comments 32.0 – Requirements for Performance Instrumentation of DCE RPC and CDS Services, Open Software Foundation.
OSF, 1995. OSF Request For Comments 33.0 – Standardized Performance Instrumentation and Interface Specification for Monitoring DCE-Based Applications, Open Software Foundation.
Pavlou, G., Bhatti, S.N., Knight, G., 1993. The OSI Management Information Service User's Manual, Version 1.0. University College London, UK.
Pavlou, G., Knight, G., Walton, S., 1991. Experience of implementing OSI management facilities. In: Proceedings of the Second International Symposium on Intergrated Network Management, Washington, DC.
Reed, D., Aydt, A., Madhyastha, T., Noe, R., Shields, K., Schwartz, B., 1992. The Pablo Performance Analysis Environment, Technical report, University of Illinois.
Ries, B., Anderson, R., Breazeal, D., Callaghan, K., Richards, E., Smith, W., 1993. The Paragon performance monitoring environment. In: Proceedings of Supercomputing'93, Portland, Oregon, pp. 15–19.
Rolia, J., Vetland, V., Hills, G., 1995. Ensuring Responsiveness and Scalability for Distributed Applications. In: Proceedings of CASCON '95, Toronto, Canada, pp. 28–41.
Rolia, J.A., 1994. Distributed Application Performance, Metrics and Management, Elsevier, Amsterdam.
Rolia, J.A., Sevcik, K.C., 1995. The methods of layers. IEEE Transactions on Software Engineering 21, 689–700.
Rose, M., 1990. The Open Book: A Practical Perspective on OSI, Prentice Hall, Englewood Cliffs, NJ.
Sun, Y., 1997. Measuring and Modelling RPC Performance in OSF DCE, Master's thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Canada.
Sun, Y., Bunt, R., Oster, G., 1997. Measuring RPC trafficin an OS/2 DCE environment. In: Proceedings of the Fifth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '97), Haifa, Israel, pp. 51–54.
Tang, A., Scoggins, S., 1992. Open Networking with OSI, Prentice Hall, Englewood Cliffs, NJ.
Taylor, D.J., 1993. The Use of Process Clustering in Distributed-System Event Displays. In: Proceedings of CASCON '93, vol. 1. Software Engineering, Toronto, Canada, pp. 505–512.
Taylor, D.J., Kunz, T., Black, J.P., 1996. A Tool for Debugging OSF DCE Applications. In: Proceedings of the 12th Annual International Computer Software and Applications Conference, Seoul, Korea, pp. 440–446.
Trommler, P., Schade, A., Kaiserswerth, M., 1995. Object Instrumentation for Distributed Applications Management, Technical Report RZ 2730 (88162), IBM Research Division.
Turner, C., 1995. Fault Location in Distributed Systems, Master's thesis. The University of Western Ontario.
Woodside, C.M., Neilson, J.E., Petriu, D.C., Majumdar, S., 1995. The stochastic rendez-vous network model for performance of synchronous client-server-like distributed software. IEEE Transactions on Computers 44, 20–34.
Yan, Y., 1994. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers, Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences.

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.

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.

Hanan L. Lutfiyya received her Ph.D. from the University of Missouri at Rolla in 1992. She is currently an Assistant Professor at the University of Western Ontario, London, Canada. Her research interests include management of distributed systems, fault tolerance and software architecture.

Andrew D. Marshall is a Research Associate with the MANDAS Project at the University of Western Ontario. He is also a doctoral candidate in Computer Science at UWO. His research interests include management of distributed applications and systems, software engineering for distributed systems and software reengineering.

Michael A. Bauer is Senior Director, Information Technology Services, at the University of Western Ontario. He is also a Professor in, and former Chair of, the Department of Computer Science. His research interests include distributed computing, applications of high-speed networks and software engineering.