Policy-Driven Fault Management in Distributed Systems *

Michael J. Katchabaw, Hanan L. Lutfiyya, Andrew D. Marshall, and Michael A. Bauer
Department of Computer Science,
The University of Western Ontario
London, Canada

hanan@csd.uwo.ca

Abstract

Management policies can be used to specify requirements about the desired behaviour of distributed systems. Violations of policies (faults) can then be detected, isolated, located, and corrected using a policy-driven fault management system. Other work in this area to date has focused on network-level faults. We believe that in a distributed system it is more appropriate to focus on faults at the application level. Furthermore, this work has been largely domain specific—a generic, structured approach to this problem is needed. Our work has focused on policy-driven fault management in distributed systems at the application level. In this paper, we define a generic architecture for policy-driven fault management, and present a prototype system based on this architecture. We also discuss experience to date using and experimenting with our prototype system.

Key Words: *Fault Management, Distributed Systems, Policy-driven Management, DCE, OSI Management Framework, Distributed Applications Management*

1 Introduction

A distributed computing system consists of heterogeneous computing devices, communication networks, operating system services, and applications. As organisations move toward distributed computing environments, there is a corresponding growth in distributed applications central to the enterprise. The design, development, and management of distributed applications presents many difficult challenges. As these systems grow to hundreds or even thousands of devices and similar or greater number of software components, it will become increasingly difficult to locate faults, determine bottlenecks, anticipate problems, or even determine how the system is behaving. Doing so, however, is critical to ensuring the reliability and performance of distributed systems. It is necessary to provide the means to detect problems in these systems, isolate and locate their causes, and then perform the actions required to correct the problems and recover from them. To do this, management of distributed systems is essential.

When planning how to manage a distributed system, policies can be defined to specify the desired behaviour of the system. Policy targets may range from individual system entities such as application processes and workstations to collections of these system entities. Policies relate to various parameters and behavioural properties of the distributed system such as availability and performance. These policies can be checked to see if they are being satisfied by the system. We refer to this as *policy violation detection.* However, the detection of a violation only indicates that the system has deviated from the behaviour required by its policies. This deviation is referred to as a *failure* and is manifested through observed *errors* or *symptoms.* Symptoms, however, are not enough to allow the fault to be corrected. This can be seen in the following example. We may have a policy that mandates that all client processes should not have to wait for more than x seconds for a remote request. Suppose that a client process attempts to contact a server in the distributed system and times out in the process (i.e., the client cannot contact the server). This symptom could be caused by several things: a network link is down, the server process is down, the host that the server process is on is down, the host that the server process is on is heav-

*Supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the IBM Centre of Advanced Studies

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE ’96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

ily congested, the server process depends on another process on another host and that process or host may be down, and so on. Any of these could be the fault causing the unsatisfactory system behaviour. Hence, once an error has been detected, the next step is to be able to locate the fault causing the policy violation. We refer to this as *fault location* or *violation location*. Once found, the cause of the violation is corrected and recovery actions (*corrective action planning*) are taken to restore the system to an acceptable state. This process is referred to as *policy-driven fault management*.

Much of the work in this area is currently in its preliminary stages. While some successes have been realized so far in this work [2, 5, 6, 7, 9], much remains to be done:

- Most of the work used network management services to provide information about the status of network devices (i.e. nodes, routers, etc) by occasionally polling these devices to see if they are still alive. This is usually done by using SNMP or CMIP [11] agents. To manage a complex distributed system requires management of both the network and applications. We expect a complex system to contain many components both hardware and software. It is not possible to practically manage each entity using the polling model. Yet, to manage overall system behaviour as specified by policies, it is necessary to have effective observation and control of the hardware and software components. Thus, it must be possible to have local interfaces to managed entities and manage them remotely. For software components this implies a need for instrumentation of distributed application components. The instrumentation must include the ability to detect the existence of a problem or fault and be able to notify an agent (remotely if necessary) of the problem. The agent should then be able to forward the notification to a management application that then locates the fault. This in turn may require the management application to request additional information about the system. This *push* (event-driven) model is more efficient for the submission of reports notifying of exceptional conditions. Little work has been done on having application components detect and report failures.

- Presently, the development of policy-driven fault management systems is typically difficult, time-consuming, and domain or system specific. There are many decisions that must be made in the development of these systems, for example what components are needed, what relationships and interactions should the system have with the environment, etc. This difficultly is partly due to design issues not being separated from implementation details. For example, there are several services required by many diagnostic systems, including symptom correlation, symptom tracing, object diagnosis, etc; These services are independent of the underlying system. As a result, there is a need to develop generic and structured architectures and frameworks for this problem.

Our current work addresses these issues. We have focussed on policy-driven fault management at the application and process level in distributed systems. To do this, we have defined a generic policy-driven fault management architecture based on policy-driven management concepts and techniques, defining the various components required, as well as the interactions between these components. Based on this architecture, we have developed and implemented a prototype management system capable of managing faults at the application level in distributed systems. We have also gained considerable insight and experience from using and experimenting with the prototype system to date.

In this paper, we discuss our work so far and directions we intend to take in the future. The remainder of this paper is organised as follows. Section 2 presents our generic policy-driven fault management architecture at several levels of abstraction. Section 3 describes our prototype management system based on this architecture. Section 4 discusses our experience with the prototype system and our results so far. Section 5 concludes with a summary of our work and directions for future work.

2 Related Work: Management

In this section, we define the general framework that we used for defining our policy-driven fault management system. The framework is based on the Open Systems Interconnection (OSI) Management framework [13].

Management systems contain three main types of components that work together: *managers*, which make decisions based on collected management information, *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. 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.

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

A *management 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 management 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 appropriate management applications. Management agents perform operations requested by managers and notify managers of pre-determined events of interest to the manager.

Managers serve to implement management policies on behalf of human managers, and view the managed system as a collection of managed objects; they typically know nothing of the real resources being managed. This approach has several advantages.

It is this framework that we use in developing our policy-driven fault management architecture.

3 Policy-Driven Fault Management Requirements

In our investigation into policy-driven fault management architectures we have identified the following primary requirements.

Transparency. The services provided by the architecture components must be as transparent as possible to the users of the system, i.e., the user may notice a minimial degradation of service but, ideally, the user should not be made aware of the existence of the fault.

Flexibility. The architecture must admit components and services that can be used to handle a wide variety of faults, adapt to a wide variety of systems, and scale from small, department-level distributed systems to ones of global scale. This implies the need for both active and passive fault management. In active fault management, the management system periodically queries the status of the managed system to determine if a fault has occurred. This approach tends to be costly in terms of management overhead resources used in the process. In passive fault management, the managed system notifies the management system when errors or faults are detected. While this is less costly than active management, this approach can only be used with resources that can be instrumented easily for management (such as application processes). As a result, a general purpose policy-driven fault management system must support both kinds of fault management—passive for efficiency, and active for coverage of all resources and services.

Structure. The architecture should be extensible and modularized so that it can be easily integrated into a general purpose distributed systems management system. This entails interactions with other management services responsible for tasks that includes configuration management, performance management, security management, accounting management. Being able to interact with a configuration management system is of particular importance to fault management, because configuration information is required in violation detection, location, and correction.

Co-Existence. The architecture should support generally accepted management standards to facilitate integration with other work in this area. The use of proprietary services and protocols may restrict the usefulness of the architecture over time.

4 A Policy-Driven Fault Management Architecture

The architecture defines the major components and the services associated with each component that interact to function as a policy-driven fault management system (which we will now refer to as the fault management system). We will now describe the architecture: i.e., the major components that interact to function as a policy-driven fault management architecture. In this section, we will identify the components of the architecture and the service interfaces among the components of the architecture. We have also identified other components of a distributed systems management system that are needed to support our policy-driven fault management system. Our architecture is based on the framework described in section 2 and is graphically represented in Figure 1. The various components and interactions are described below in detail.

Configuration Management System: This is not actually part of the fault management system, but this represents a set of services that is needed by the fault management system. The configuration management system is used to collect, maintain, and serve information on the configuration of the resources and services available in the distributed computing environment. The fault management system uses configuration information provided by this system for fault detection and diagnosis, and may update information maintained by this system as a result of correcting and recovering from faults. We have developed an information model

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

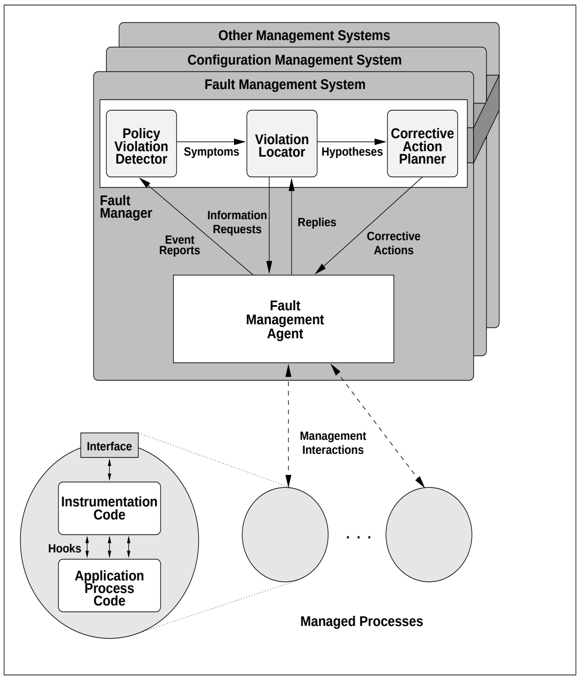

Figure 1. Generic Policy-Driven Fault Management Architecture

that characterises the essential information about the components of the system and applications and the relationships among these components [15]. We are currently working on the development and implementation of services needed for the collection and maintenance of descriptive and location information about the entities of the distributed system.

Instrumentation: Instrumentation is the code added to processes that allows the processes to be managed. It consists of three main parts:

Instrumentation Interface: The instrumentation interface is used by management agents to contact the managed process for management purposes, and vice-versa. The interface, upon receiving incoming requests from management agents, invokes the appropriate functionality in the instrumentation code. Similarly, the instrumentation code sends requests or reports to the appropriate management agents through the use of the instrumentation interface.

Instrumentation Code: The instrumentation code provides an internal view of the managed process. It does this by retrieving information from the process and performing control actions on the process in response to management requests and generating reports periodically or when a significant event inside the process occurs. By retrieving process information or generating reports, faults can be detected by the fault management system. For example, we may have a policy that mandates that all client processes should not have to wait for more than x seconds for a remote request. A client process may be instrumented with code that determines the response time of the server process when the client has initiated a request and checks to see if the response time is within x seconds. If it is not then the client process sends an event report to the fault management agent. The event report contains information such as the client identifier, the server identifier, the actual response time, etc.

Through manipulating various settings in the instrumentation, the level of management can be controlled.

Instrumentation Hooks: Instrumentation hooks are used to attach instrumentation code to application process code at key locations to permit management, facilitating both the collection of process information, and the control over the managed process. These hooks are placed at the process entry point, every process exit point, book-ending each communication with other managed processes, and at other key locations for detecting possible faults i.e., after I/O operations, checking executable assertions.

Fault Management Agent: The fault management agent is used to interact with the managed processes. The agent will receive information requests and control requests from the fault manager and perform the corresponding operations on the appropriate managed processes through the instrumentation interfaces of the processes. The agent will also receive periodic and event reports from the managed processes that will be processed and forwarded on to the fault manager as necessary.

Fault Manager: The fault manager is used to implement management policies on behalf of the user. This is done through sending various management requests to and receiving reports from the fault management agent. The manager is responsible for detecting faults, locating them, and correcting them. Because this is a policy driven architecture, each of these tasks is based on fault management

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

policies specified by the user. Each task is encapsulated in a separate component, discussed below:

Policy Violation Detector: The policy violation detector is used to detect violations of user-defined policies indicative of faults in the system. This detection is based on information collected from the fault management agent. This can be done either by explicitly requesting information about managed processes (active management) or by receiving event reports forwarded by the agent (passive management). When violations are detected, these are all possible symptoms of faults. To determine if a fault has actually occurred and, more importantly, the cause of the fault, these symptoms are passed on to the violation locator component of the fault manager. Let us examine the policy discussed earlier. If only one client process receives its response in more than x seconds within a one hour time interval the the policy violation detector will determine that no policy is actually violated.

Violation Locator: The violation locator takes incoming symptoms resulting from policy violations and determines the cause and location of the problem in the system. To do this, the violation locator makes heavy use of configuration information from the configuration management information, as network topology, host configuration, and application deployment information is all useful in the process. Furthermore, if additional information is required from the managed processes to assist in the location process, the violation locator can contact the processes through the fault management agent as necessary. Based on its analyses, the violation locator generates a set of hypotheses for the causes of the symptoms it is passed. These hypotheses are in turn passed to the corrective action planner to allow recovery to commence.

Corrective Action Planner: The corrective action planner examines the hypotheses generated by the violation locator and determines a sequence of actions to repair or otherwise recover from the detected fault. Once a plan has been developed, the fault management agent is instructed to carry out the appropriate actions necessary to correct for the fault. This, in turn, results in the agent making corresponding requests to the managed processes for the generated plan to be finally executed. The corrective action planner, depending on the nature of the actions required, may also need to update information maintained by the configuration management system to reflect the new state of the system.

The architecture described above is flexible and can be used to construct various management system configurations to meet functional and efficiency requirements. For example, the architecture supports the use of multiple agents and managers, interacting with each other to provide fault management across domains over a widely distributed system. For example, it might be desirable to partition management functionality across agents or managers, with each responsible for a different class of faults. It may also be desirable for other management systems, such as a performance management system, reporting "hot spots" to the fault management system for further investigation. The architecture supports all of these possibilities.

To demonstrate how this architecture manages faults in a distributed system at this level, consider the normal operations of the fault management system:

1. When the distributed system is first initialised, the fault management system is as well. This includes at least one fault management agent and one fault manager.

2. Distributed applications are then started under normal operation of the system. They are registered with the configuration management system, which assigns them to a fault management agent.

3. When the distributed system is functioning correctly, no reports are generated. In this case, only the application processes are active. The fault management agent and fault manager are inactive and consuming minimal resources (the management system is operating passively).

4. When a fault occurs that disrupts the normal functioning of the distributed application, the managed process that detected the error using its instrumentation sends a report to its fault management agent describing the problem. The process may then be suspended, wait, or take an alternative action until the fault is corrected.

5. The agent takes the information given to it by the managed process and uses it, together with additional information, to generate an event report describing the detected error to its fault manager.

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

6. The event report is sent to the fault manager, and is received by the policy violation detector component.

7. Upon receiving the report, the detector compares the report with policies specified by the user to determine if a policy has been violated. If this is the case, a symptom report describing the problem is sent to the violation locator.

8. The violation locator takes the symptom report, together with additional information from querying the fault management agent and the configuration management system, and determines the cause of the error – a fault in some object in the distributed system. A hypothesis describing this is generated and sent to the corrective action planner.

9. The corrective action planner examines the hypothesis and generates a sequence of actions (an action plan) that will correct and recover from the fault.

10. The planner executes the corrective plan by issuing the appropriate requests to the fault management agent, which in turns interacts with managed resources to carry out the plan. The system configuration, maintained by the configuration management system, is also updated as necessary.

11. Having corrected the problem, the managed process that originally reported the problem is advised that it can continue at this point. If the problem persists, feedback to the agent and fault manager can attempt to correct the situation in another way.

Having presented the architecture at its highest level, we now proceed to examine some of the more significant portions of the architecture in more detail by examining the service interfaces among the components of the fault manager and the fault management agent.

4.1 Service Interfaces and Interactions

Having discussed the components of the fault management system in the previous sections, we now present the service interfaces between the various interacting components in detail.

4.1.1 Interfaces Between the Fault Management Agent and Instrumented Processes We will first begin with the service interfaces between the fault management agent and the instrumented processes. These interfaces enable that the agent is able to make requests to the processes and that the processes are able to send notifications to the agent.

requestManagementInformation: This service is used by the fault 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.

sendManagementInformation: This service is used by managed processes to report management information periodically to a fault management agent. This information is then, with additional information such as a unique identifier, time stamp, process identifier, etc, packaged into an event report by the agent and sent to the policy violation detector component in the fault manager to determine if this information violates a management policy.

sendNotification: This service is used by managed processes to report notifications of significant events to the fault management agent. These events include process creation, process termination, process failures, and degradation in quality of service. The information in these notifications is then, with additional information such as a unique identifier, time stamp, process identifier, etc, packaged into an event report by the agent and sent to the policy violation detector component in the fault manager to determine if this information violates a management policy.

performControlAction: This service is used by the fault 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.

performManagementAction: This service is used by the fault management agent to perform an action on a managed process that changes the way that the process is being managed. These actions can be used to activate or deactivate management in processes, change the amount of management information sent using the `sendManagementInformation()` service and

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

how often it is sent, and change the thresholds used to generate alarm notifications using the `sendNotification()` service. 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.

4.1.2 Interfaces Between the Fault Manager Components and the Fault Management Agent

We will now describe the service interfaces between the fault manager components and the fault management agent. These interfaces enable the fault manager to make requests of the agent and enable the agent to send notifications and other information to the manager regarding the processes it is managing.

requestManagementInformation: This service is used by the policy violation detector and violation components of the fault manager to request information on some managed object in the system from the fault management agent. This request includes a scope specifying the subset of managed objects to examine, a filter specifying the properties of scoped managed objects of interest, and a set of attributes to retrieve. When done, a list of attributes for each managed object meeting the scoping and filtering requirements is returned.

sendEventReport: This service is used by the fault management agent to forward an event report to the policy violation detector component of the fault manager to determine if the state of the system described in the report violated a management policy. These reports are generated either periodically or upon the occurrence of some significant event in the managed system.

performAction: This service is used by the corrective action planner component of the fault manager to have a fault management agent carry out a corrective action on managed objects in the managed system. This action request includes a scope specifying the subset of managed objects to examine, a filter specifying the properties of scoped managed objects of interest, the action to perform, and a set of input parameters to the action. In response to this request, the agent will perform the given action on the managed objects meeting the scoping and filtering requirements, and return a set of output parameters resulting from the action.

4.1.3 Interfaces Between Components of the Fault Manager

Finally, we describe the service interfaces among the fault manager components.

sendSymptomInformation: This service is used by the policy violation detector component to report symptoms to the violation locator component of the fault manager. Each symptom is reported with a unique identifier as well as a report detailing the symptom to be processed. Based on this, violation location commences to determine the cause of the symptom.

sendFaultHypothesis: This service is used by the violation locator component to report hypotheses to the corrective action planner component of the fault manager after violation location has uncovered and located a fault. This includes a unique identifier for the hypothesis, the unique identifier for the object diagnosed to contain the fault, the location of this object, and a reason explaining the cause of the fault. Based on this information, a corrective action plan can be developed and executed to recover from the located fault.

requestNewFaultHypothesis: Occasionally, when a hypothesis has been sent to the corrective action planner using `sendFaultHypothesis()`, the planning process may discover that the hypothesis is not acceptable. For example, this can occur when the hypothesis reported contradicts the current system configuration, etc. In such cases, it is necessary for the corrective action planner to request a new hypothesis from the violation locator. By identifying the hypothesis by its identifier given in the `sendFaultHypothesis()` service invocation, the locator can continue reasoning based on the analysis it had previously completed. When a new hypothesis is obtained, the planning process commences again.

5 Prototype Implementation

A prototype policy-driven fault management system based on the generic architecture presented in Section 4 was developed as a proof of concept. In this section, we discuss the prototype in detail and how it met our requirements.

5.1 Prototype Target Environment

Developing and using distributed applications has been made easier and more efficient through the use of various midware environments. Typically, these environments provide libraries and packages for developing distributed applications, as well as run-time services to facilitate communication and provide security, file services, directory access, time functions, and so on. One

Proceedings of the Seventh International Symposium on Software Reliability
Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

of the more popular environments is the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE) [8], which was used as the environment for developing our prototype management system.

5.2 Management Environment

Based on the success of our previous work [3, 4], we believe used that the OSI Management Information Service (OSIMIS) [10] developed at University College to facilitate the development of our fault management system. It provides an object oriented infrastructure for developing both manager and agent applications. OSIMIS includes a Guidelines for the Definition of Managed Objects (GDMO) compiler to produce code from managed object specifications in GDMO [1]. With these together, OSIMIS provides the generic portion of both manager and agent applications, leaving the developer to write only application specific code, greatly assisting the development process.

5.2.1 Management Agent Development Generic management agents were developed in our previous work [4] to manage DCE applications. These agents provided mechanisms to:

- Collect management information on managed processes either upon request or through both periodic and event reports.
- Exert control over the managed processes (termination, suspension, awakening, and so on).
- Tailor and customise the management of the managed processes by manipulating settings in the instrumentation of the processes.

These mechanisms are all required for policy-driven fault management, and are provided for in our architecture presented in the previous section. As a result, these agents could be used with essentially no modifications for use in our fault management system. (The only additions required were support for constructing new event reports for handling a wider variety of faults.)

5.2.2 Manager Development In addition to addressing the issues of general purpose DCE application management using the OSI Management Framework, some of our other previous work investigated both policy-driven management [12]. and fault location [14]. Prototype systems developed in this work were used as a basis for the fault managers used in this work. Both earlier systems were based, in part, on various artificial intelligence techniques that made adapting them for this work significantly easier. This includes:

- The object diagnosis used in the original fault locator was implemented using expert systems techniques. Rules, data, and heuristics constituted the "expert knowledge" used in evaluating the histories of object and in their diagnosis. By altering the rules, data, and heuristics, different classes of faults can be handled with few modifications to actual locator system.
- Corrective action plans were developed in the original policy-driven management work using standard hierarchical planning algorithms. By modifying the planning expertise (the rules used in developing plans), plans can be created to correct and recover from a wider variety of faults and other problems.

By taking this previous work, modifying it, combining it, optimising it, and adding OSI compliance, our prototype fault manager was developed. Because of its basis in artificial intelligence techniques, as described above, the system is adaptable and extensible in the future.

5.3 Faults of Interest

To assess the viability and validity of our prototype, we initially restricted ourselves to faults dealing with failures of DCE core services and application-level servers. To give an idea of what would occur in such a failure, consider the following example:

1. Suppose a DCE service or an application server process fails (due to congestion, a transient failure, a system failure, or some other reason). For this example, consider that the problem is due to a system failure.
2. Several clients to this application server attempt to contact it and fail, thereby, timing out in the process. This results in application-level errors in the clients.
3. The clients generate reports based on this error and forward them to their fault management agent. The clients then suspend themselves until the problem is resolved.
4. The agent generates the appropriate event reports based on these error reports and send these to the policy violation detector.

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

5. Assuming a policy exists mandating that all clients must be able to contact their servers, these reports violate this policy. The errors detected by the clients are symptoms of the problem.

6. These symptoms are processed by the violation locator. The locator correlates these symptoms and determines that they all indicate the same problem – a failure in contacting the down server. The locator then begins to diagnose the various objects involved in the scenario: the client hosts, the server host, the network links in between, the server process, and so on. This diagnosis will conclude that the client hosts and network links are operational, and that the server process is not due to the failure of its system.

7. This hypothesis is received by the corrective action planner. It deduces from analysing the original policy violation and diagnosis is that its end goal is that the server should be restarted. A precondition to doing this is that the machine that the server is to be started on must be operational. Noting that the current server host is not, a message is sent to the system administrator to restart the system (assuming the corrective action planner cannot do so itself) and the planner selects a new system that is operational. The server is started on the new system by performing the necessary operations, and the clients are awakened and instructed to use the new server.

In our view, the class of faults given above provides a sufficiently large and complex set of faults to gauge our initial prototype. Based on our experience, we plan to expand our policies, diagnostics, and heuristics to handle a wider variety of faults without major modifications to the actual fault management system itself.

6 Experience

Initial experiments using the prototype policy-driven fault management system were conducted to verify the operation of the system and assess its suitability for managing faults for distributed applications. These experiments involved executing DCE applications instrumented in our previous work with the prototype fault management system. Various faults were introduced to cause both application and DCE servers to be down, or at least appear to be down. This included terminating the server processes, suspending the server processes, inducing node failures, and so on (fortunately, this was made easier using our management system as it can instruct processes to shut down, be suspended, etc.) We were able to show that our prototype works effectively.

Does it work efficiently? In addition to determining how the system performs in managing faults, it is equally important to determine the impact that the system has on the environment being managed (the systems, networks, and other applications involved). At this point, only a preliminary performance analysis has been done.

Initial results indicate that, from initial detection of the error until the fault is corrected, takes no more than a few seconds. This varies depending on a number of factors, including:

- The nature of the application being managed (its size and distribution across the distributed environment).
- The cause of the fault. Fewer diagnostics may need to be applied to locate the source of a fault depending on what the source is.
- The faulty server. Different diagnostics, planning, and actions are required depending on whether the server is a DCE server or a normal application server.

These results indicate that the system is reasonably efficient, but more analysis is clearly required. A more thorough and complete analysis is currently under way; with this analysis, we will be able to optimise and tune the system for increased performance in the future.

7 Concluding Remarks

The work we have described here is part of an ongoing structured attack on the fault management problem. There are many complex issues yet to be resolved, and much work is still to be done. We believe that a flexible and generic policy-driven approach to the problem makes it easier to address these issues. Our prototype system derived from this work supports this belief.

The work detailed in this paper represents initial research into policy-driven fault management, focussing on application-level faults. We have introduced policy-driven fault management and presented a generic and well-structured architecture capable of supporting it. Through a prototype fault management system, we have demonstrated that a policy-driven approach to this problem is quite suitable and that our generic architecture supports the effective detection, location, and correction of faults in distributed systems. The system is also capable of interacting with other management systems to provide a single unified view of distributed systems management.

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.

This work is ongoing. We are currently carrying out more detailed performance measurement and analysis on our prototype system; the results of which will be used in optimising and refining the system. The current prototype system is restricted in the class of faults that can be dealt with – the system must be extended to handle more classes of faults, requiring the introduction of new policies, diagnostics, and heuristics. We need to more fully address the problem of distributing the fault management function across widely distributed environments. This is an issue currently under investigation. Just like any system, the fault management system developed in this work can experience failures as well. We are currently examining various self-management techniques to handle these problems.

We currently have a number of ongoing projects in the areas of policy-driven management and fault management. The work done in this paper has laid an excellent foundation for these projects, and continuing research in this area in the future.

References

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

[2] M. Frontini, J. Griffin, and S. Towers. A Knowledge-Based System for Fault Localisation in Wide Area Networks. *Integrated Network Management II*, Elsevier Science Publishers B.V. (North-Holland: 1991.

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

[4] 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. *To Appear in the Proceedings of the Second International IEEE Workshop on Systems Management*, Toronto, Ontario, Canada, June 1996.

[5] L. Lewis. A Case-Based Reasoning Approach to the Resolution of Faults in Communications Networks. *Integrated Network Management III*, Elsevier Science Publishers B.V. (North-Holland: 1993.

[6] R. Mathonet, H. Van Cotthem, and L. Vanryckeghem. DANTES: An Expert System for Real-Time Network Troubleshooting. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, pages 527–530, Milan, Italy, August 1987, pp. 527-530.

[7] Feridun Metin. Diagnosis of Connectivity Problems in the Internet. *Integrated Network Management II*, Elsevier Science Publishers B.V. (North-Holland: 1991.

[8] OSF. *Introduction to OSF DCE.* Open Software Foundation, first edition, 1992.

[9] J. Pasquale. Using Expert Systems to Manage Distributed Computer Systems. *IEEE Network*, September, 1988.

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

[11] W. Stallings. *SNMP, SNMPv2, and CMIP: The Practical Guide to Network -Management Standards.* Addison-Wesley, Reading, MA, 1993.

[12] D. Stokes. Availability and Performance Management in Distributed Systems. Master's thesis, The University of Western Ontario, September 1995.

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

[14] C. Turner. Fault Location in Distributed Systems. Master's thesis, The University of Western Ontario, September 1995.

[15] A. Welch. Managing Configuration Information in Distributed Systems. Master's thesis, The University of Western Ontario, September, 1995.

Proceedings of the Seventh International Symposium on Software Reliability Engineering (ISSRE '96), White Plains, NY, October 30 - November 2, 1996, pages 236-245.