DISTRIBUTED ALARM MANAGEMENT BY DELEGATION FOR HETEROGENEOUS AND MULTIPROTOCOl NETWORK ENVIRONMENT

V. CARNEIRO, R. KOEHN, A. VIÑA

Facultad de Informática, Campus de Elviña s/n. 15071 A Coruña. SPAIN
email: {viccar, avc}@udc.es. Phone: +34-81-167000. Fax: +34-81-167160

Abstract
We introduce here an alarm management system based on Management by Delegation paradigm (Mbd) that gives the operator access to management facilities through an applet that is executed on any WEB. This application gives to operator an integrated and homogeneous environment in which different types of alarms exist. This system also resolves two problems inherit to traditional fault management: their poor semantics and your centralised environment. The applet developed permits multi-user sessions, dynamic control of delegated process, distributed alarm management (filtering, scooping, polling, etc.) and platform and location independence. The language Java with RMI and JDBC facilities has been used for this development.

Keywords: Distributed Systems, Fault Management, Management by Delegation,

1.- INTRODUCTION

An group of limitations in current fault management systems are its platform-centered paradigm that is used due to the low capacity of the resources on the network and the homogeneity of the devices. Platform-centered view separates applications logically and physically from the data they require and from the devices that controls them. Agents are simple collector and transmitter entities of information to the managers that in some rates are overflowed. This paradigm is unscaleable, so in a wide and heterogeneous environment the exchange of data between agents and manager can exceed the capacity of the own network. System management becomes quite difficult if the number of devices, variables to manage and network speed increases. Furthermore, during network fault times, the policy of sending every notification towards central manager, deteriorates the situation due to the delay time added by these communications on the network itself.

In the other hand, although current management systems alert in failures, the main problem is that additional information is not given to identify the source of them. Most of legacy management protocols (and even standard ones, like SNMP [1]) have not been designed to transport this kind of information in its PDU’s. OSI [6] incorporate more information in its M-EVENT
primitives but it's not enough for some semantic aspects. Also, integration of legacy and standard protocols (like SNMP-CMIP [7]) is, in most cases, not viable because of administrator see alarm information in different formats. Due to this problem, it's very difficult to understand and manage that information. ISO has the X721 and X733 [4] standards which introduce the information model to follow. This model allow the exchange of alarms between management entities, in this way, it's possible a management system with a uniform group of kinds of notifications, with standard parameters. The use of robust and object oriented databases simplifies the information access about the events. Furthermore, it's possible to record additional information about them that is known previously, allowing failure location.

In section 2 of this paper we introduce this new model we have developed as well as the conceptual aspects of its implementation. In section 3 we describe how the model has been developed, using object oriented technology, distribution facilities that Java gives, RMI and JDBC. Subsequently, we expose the features of the management system's GUI. Finally, we make an appointment of the actual state of the work and conclusions we have obtained.

2.- SYSTEM ARCHITECTURE

In order to solve the limitations previously mentioned, we have designed a distributed management model that is based in Management by Delegation concepts, to apply them to fault management. Management by Delegation (MbD [2]) is based in Elastic Processes (EP). An EP is a process that can modify its behavior, due to the addition/removal of code at runtime. An entity (manager) uses a delegation protocol to download and control a Delegated Program (DP). A DP is maintained and executed by a MbD-Kernel residing in a remote host.

There is a main module, within the model we have designed, that resides in each managed node (although a node is allowed to manage several equipments). This module, called Delegation Support Architecture (DSA), gives us the following facilities:

- **DPi instantiation**: There is a mechanism to allow the DSA that ordered the instantiation to control its execution, through remote orders. This approach avoids traditional script-based management system; those architectures don't allow the user to control the execution of a script once launched.

- **Process delegation**: Allows us to give our particular logic to another DSA, as well as asking a peer for its logic. Once delegated, transferred behavior can be used to make several instances of that class, and is stored on the repository. If more instances are needed, there is no need of downloading that behavior again and, at the same time, increases agent autonomy.

- **DPi communication**: There is a mechanism that allows direct communication between DPis. This communication can be established between any two DPis, no matter if they reside on the same DSA or not, in a transparent way for the programmer. Each DSA manages all local DPis, registered in its DPi-Table.

Figure 1 shows us the main modules that compounds the proposed model. Due to the execution of any of these modules a DSA is needed to be previously running on that node. These modules can also be executed on one or several machines, and behaves autonomously during failure periods, because they can act under connectivity breakdown scenarios. An additional feature is
the possibility of a better organization of functionality categories (i.e. filters, polling routines, adapters, etc.). It also facilitates the development of automatic tools to add elements to that categories.

- **Advanced View Manager (AVM):** It supports all viewing and storing logic for X.733 alarms. From this module you can control all devices being managed and send the necessary orders to delegate processes or instantiate them. We can also send orders to control created instances. Another characteristic is user management, having different access permissions and views of the whole system.

![Diagram of Proposed Model](image)

**Figure 1: Proposed Model**

- **Alarm Composition Manager (ACM):** It has the logic necessary to compound X.733 alarms, filling fields with previously known or deduced about the equipments. This logic send/receive X.733, CMIP or SNMP alarms depending on the configuration. This module
also contains the actions that can be activated as a response to some alarms. All these routines are delegated to the DSAs on demand.

- **Polling Manager (PM):** It has polling policy logic, to monitor equipment state periodically. It allows us the definition of state graphs in order to the construction of alarm states. Filtering and polling policies, residing in this module, can be defined both over one or a group of equipments.

- **Adapter Manager (AM):** Logic for standard or legacy protocol conversion to X.733. Its the bridge to integrate any kind of alarm in the system. It contains also the modules that allow us to send alarms in several formats or to several targets.

- **Configuration Manager (CFM):** It contains all the kinds of Control Modules (COM) that can be executed over the managed equipments (with a previously started DSA). Every module acts over its related equipment with given security, access permission and priority specifications. It has the responsibility of updating and managing all the ASD_Tables it knows about, which contains information like type of DSA, address where it is located in, port, etc. It is directly related to AVM and it is the core of the control system. Under fault conditions, any other ASD on the system can act as the CFM.

When launching the system, the first DSA in being executed is CFM. This one knows which are the equipments to be managed, and it sends a DSA and a COM to the machines that will manage them. It also knows which managing modules (ACM, PM, AM) must launch and where. Once initiated the system, the operator/s can execute one or more AVM that will communicate with the CFM. AVMs can, among other things, alter the configuration parameters, depending on the permissions given to the operator. Each change orders the CFM to delegate a DSA and a COM to a concrete resource.

### 3. IMPLEMENTATION

Most relevant features of Java are the following:

- **Write Once, Run Anywhere.** Its always important, mainly if you are dealing with a new development. Also, managed components can be very heterogeneous and be running under several platforms.

- **Object Serialization.** Being able to send objects through the DPs on the network, in a simple way, is a useful feature.

- **Dynamic Class Loading.** We need all nodes to be able to load needed logic at run time. It is in delegation issues where this aspect is specially important.

- **RMI.** Some sort of RPC mechanism is particularly interesting here, in order to be able to send remote control orders between nodes. It also brings us Dynamic Class Loading automation.

- **Distribution Capabilities.** There are several aids for distribution in the Java language. From the basic socket-based communication facilities, much easier to use than in commonly used languages to RMI capabilities and integration with open standards in distribution, like CORBA through Java IDL.

Behavior delegation mechanism is based on the Java’s ability to (down)load classes dynamically from the network, in order to quickly adapt to user demands in a fast and flexible way. In
Java is possible to load a class's behavior from the network at runtime, so you can get specialized logic from an specialized DSA, anywhere in your system (polling strategies, adapters, filters, etc.). So, you can delegate that behavior and make it available anywhere as needed. You only have to order (download) it. RMI makes dynamic class loading automatic; you just need to implement a well-known interface, and you will be able to access to that classes without having compile-time knowledge about them.

We have implemented an order dispatch mechanism that offers the programmer a simple syntax to control de DPis. This high-level API is location transparent, so it makes things be easier. It also does, as needed, the translation to lower-level local or remote methods, hiding operations concerned to behavior delegation, instantiation, etc.

DSA's order managing has two layers, plus an additional one if working from a module on top of the DSA, like AVM. The first layer does not distinguish between local and remote orders and is location transparent, too. Within the second layer we are given several simpler services that can be called remotely, as delegation control (request, delegation and check), instance control, communication channel control between DPis, DSA and DPi table control, among others. Java RMI (Remote Method Invocation) is used in order to support remote order mechanism.

RMI is an object oriented RPC mechanism, fully supported in the standard Java Development Kit. It allows us method invocation between different Virtual Machines (VM), with the same syntax as methods within a single VM (although we are conscious we are in different ones).

All the modules within the DSA are designed to be easily distributed in multiple Virtual Machines, no matter if they are running in the same machine or in different machines. So, we can decide the level of distribution we have in basic components of the agents. It's also easy to distribute DPi execution, initially treated as threads within the same VM as the scheduler; you can make the DPis to be processes, each one as a VM, connected to the scheduler through RMI. Of course, if we decide to distribute DSA modules through different machines, we would need a fast connection; the higher the distribution, the higher the management traffic over the network.

4.- OPERATION ENVIRONMENT

The GUI (Graphic User Interface) support functionalities exposed before. From Alarms View Module (AVM) alarms in X.733 format are viewed as we can see in fig. 2. AVM has an alarm repository. It is possible to delete someone or inclusive everyone. If we delete an alarm at the main window of the GUI it is deleted in the management tree of the platform too, so since this moment, the alarm is not accessible for whatever application connected to the system. We have used a RDMS to record alarms and to do queries via JDBC, so it is sure the phrase “write it once and run it anywhere”. JDBC allow the use of SQL commands in an easy and direct way. Furthermore, if we run the program as an applet, the RDMS is accessed in a remote way. This is one of the greatest advantages of the AVM with regard to conventional GUI's because with a program like the appletviewer, included freely in the JDK, we can run a fully operational version of the program. It is possible to run the program like a typical application too with the advantage of the multiplatform that Java gives.

There is a color code in order to identify each alarm relying on its severity. Thereby, an alarm
with critic severity has associated the red color. This color code is a good idea if you want to identify alarms quickly. We have included a subgroup of attributes for each alarm from the ITU-T X.721 [3] recommendation for the notifications registry. Monitored attributes are modifiable from the GUI.

![Figure 2: GUI Aspect](image)

Another important feature of the GUI is the agents configuration (delegated process) on the network. It encloses its instantiation. Every agent has a unique identifier within the system. Also, it's possible to get additional information about an agent, such as the kind of delegated process of the instance (traps generator, filter,...), the responsible DSA of the instance or the DSA that is getting such an instance.

If we have instantiated a delegated process (DP) in a DSA and this instance is running, we can, through the AVM, pause/resume DP's execution, terminate a DP's and choose its channel to communicate with another DP. Every action implies the invocation of one (or more) primitives in a lower layer. Execution of these primitives is transparent for the user of the program who does not know if it is a local or remote invocation (via RMI).

5.- CONCLUSIONS

The prototype that we have exposed here solves the two bigger problems of current Fault Management Systems. Those that are associated to a centralized management (inherent to the majority of the current management platforms) including a transparent and flexible distribution in order to the resources location, and those related with the absence of information about the source of the different alarms that appears within the system, adding previous information that resides in a object-oriented database.

Also, the model add new and unthinkable characteristics in a traditional system, such as the management of the information in real time and the action of an autonomy device without
connection to its manager. It also takes management decisions that previously, or in execution time, have been transferred to it. In the other hand, our model solve the problem of alarm integration of different protocols. This is vital in modern management systems. Another characteristic in the system is the facility to extend agents, i.e. the installation of a new filter only needs its programming using Java with the API designed for it and its register in the PM. In this moment, from any COM or AVM we can delegate this filter in any resource of the system to manage.

In order to avoid language dependency of the system, we can program these modules in other interpreted language and previously delegate the interpreter. Another characteristic is that DSAs can extend control mechanism about information modeling the devices.

Remote execution control eliminates the problem of current delegated based scripts systems, because the code is executed in an asynchronous way. We can establish management policies on some DSAs or some devices. It's possible to make additional processing on alarms, i.e. alarm encryption.

Our laboratory is currently working in Artificial Intelligence techniques to append new functionalities to the model, like alarm correlation [5].

REFERENCES