Monitoring Distributed Processes with Intelligent Agents

Franz J. Kurfess
(Email: franz@cis.njit.edu)
Klaus Holthaus

Dhaval P. Shah
(Email: dhavalpshah@hotmail.com)
Felip Miralles

Computer and Information Science Department
New Jersey Institute of Technology
Newark, NJ, 07102

Abstract

This paper describes a platform-independent application for the monitoring of distributed applications. The tool is intended for system administrators to properly distribute the components of a distributed application onto particular machines. An essential part of this tool is based on intelligent mobile agent technology used to access the target systems. Although a full implementation turned out to be infeasible due to security considerations, the implementation and experimentation indicate the suitability of intelligent agents technology for such purposes.

1. Motivation and Objectives

Networked and distributed systems are essential to today's computing environments, and the management of these systems has been a driving motivation for tool development efforts for years. Integrated Network and Systems Management (INSM) deals with this application area, and several architectural efforts have been made to provide unified management technologies. Two major issues have motivated us:

(1) The management of distributed processes is a task that has found the least of consideration in systems management efforts. INSM started off as network management (i. e. management of connections, routers, network endpoints, etc.). Systems management has so far added a lot of important aspects, such as user management and event monitoring. However, a running application as a managed object within a system is not yet handled by most INSM platforms. (2) The term agent has been used in the INSM area with a slightly different meaning than that we normally find in research subjects; important differences are the aspects of mobility and autonomy. Most INSM platforms are to some extent distributed, but including distributed artificial intelligence in the form of agents allows enhancements in terms of functionality and automation. The key objectives of this paper are to explore the technology of mobile and intelligent agents, to examine the applicability of this technology to the area of distributed environments management, and to develop, implement and test a prototypical agent architecture for monitoring distributed processes with regard to issues such as performance and portability.

2. The Application

2.1. Integrated Network and Systems Management

Systems Management is a subset of tasks in the area of Integrated Network and Systems Management. INSM activities can be viewed as a management process consisting of three steps: Monitoring an object, comparing the observations with management policies / guidelines, and controlling the object by setting characteristics. The Figure 1 shows these activities and their relationships [1]:

The development of systems supporting these tasks has to consider some critical issues:

It is not possible to rigorously separate network management and systems management; these fields merge more and more as technology evolves. An arbitrary set of tools will not be enough to cope with such a complex problem area, thus integrative architectural efforts are required. The definition of management policies is the most critical part of those tasks and can not be solved by technical tools. It is crucial for INSM to make sure, as part of the management policies that the systems and networks used have the capabilities and interfaces for partially automated management.

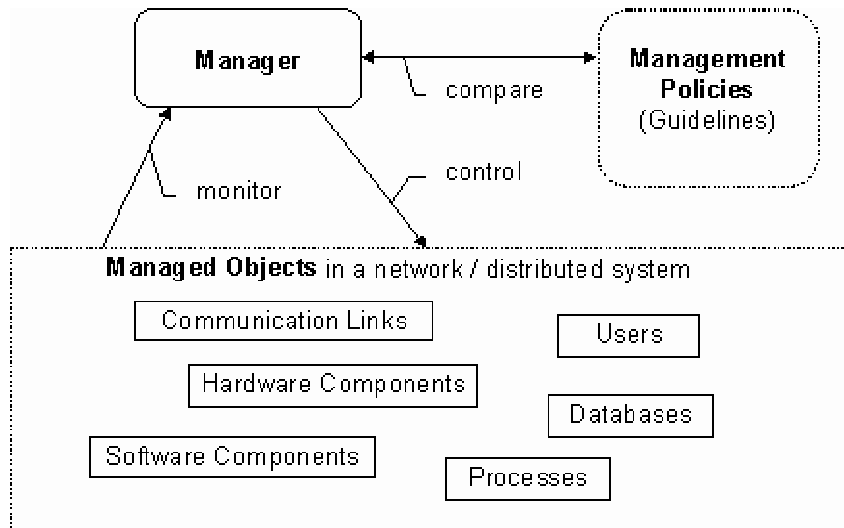

Figure 1 Management Process

Some approaches have been made to construct an integrated architecture and establish an industry standard for them. One of the most prominent ones is DME (Distributed Management Environment) by OSF (Open System Foundation) as part of DCE (Distributed Computing Environment). Other approaches are based on distributed object architectures, such as CORBA (Common Object Request Broker Architecture). However, there are few approaches to the utilization of artificial intelligence and mobile agents as components of this technology.

This paper focuses on the modeling and prototyping of a platform-independent application to monitor the processes of distributed applications on several hosts in a networked environment, i. e. we concentrate on hosts and processes as classes of managed objects. This tool helps a system administrator of a distributed environment with properly placing the different processes of a distributed application on specific machines in order to balance and optimize the host and network utilization. It does so by providing a graphical representation of an application running in a distributed environment in the form of different processes communicating and cooperating, and measurement of the performance of this application in terms of amount of network traffic and CPU load. Our tool performs a dynamic analysis of the system and helps to distribute it to optimize its execution, decrease the costs of computing and communication, and facilitate the execution of parallel tasks.

2.2 Intelligent Agents for Systems Management

Our idea is to utilize intelligent agent technology as an integrated architecture for INSM. This distributed architecture will consist of Java applets and applications, utilizing the following frameworks for intelligent agent technology:

Mobile agents, implemented in IBM's Java Aglets API (Aglet Workbench, now known as ASDK - Aglets Software Development Kit) [2].
Intelligent agent technology, provided by IBM's Agent Building Environment (ABE).

There are several reasons why we think intelligent agents are useful for our application area:
Networks are distributed, systems are distributed - therefore, management has to be distributed as well.
Considering the growth of networks and the number of end systems in companies, it would be a great relief for administration staff to have a management architecture that performs tasks in parallel and autonomously, which is exactly what a herd of intelligent agents roaming the company's networks could do.
Networks and systems are arbitrarily heterogeneous - up to the point where they support Java, which is to some extent the language of choice for agents.
Management policies and goals are most crucial for network and systems management - agents provide the mechanisms to implement these goals more directly and still more flexibly than ever before, due to the intelligent capabilities of agents.

3. Design

3.1. Task specification

Since our paper is concerned with distributed applications monitoring as a specific subtask of INSM, an agent serving this purpose would use its sensors to monitor CPU load and network communication of distributed processes. Its effectors would relocate those processes based on its goals, e. g. the premise that communication between two processes on different hosts may not exceed a certain limit.

The intention of our paper is to provide the monitoring part of the above task by retrieving information about processes and displaying that information in a graphical way. By doing this our tool gives a systems administrator the information needed to optimize the performance, network load, etc. of distributed applications.

Following information per process and per processor needs to be retrieved as a basic set:
Processor = IP address, host name, idle time, communication time, I/O time, processing time
Process = ID#, process name, type (thread / process), scheduling time, sleeping time, Kernel- running time, Process-running time
Both will be implemented as classes. Also, each processor object will contain a list of the above described process objects. Besides that, another class Communication will represent communication load between two different processes:
Communication = Process1 of Processor1, Process2 of Processor2, Communication load

3.2. The general architecture

On one hand INSM architectures need to be distributed; on the other hand however it will be necessary to have a certain degree of centralization: Agents that access managed systems are restricted in terms of size and CPU usage, therefore they can only contain the very essential functionality needed at the point of action. Another aspect in this matter is not to overload the network by transferring oversized agents from system to system, thus wasting bandwidth and CPU time of the systems to be managed. Besides that, the herd of agents roaming a company's systems must be coordinated, information must be stored, and the systems administrator responsible for the systems management function must have versatile access to what the agents are doing, e. g. to prevent serious disasters caused by malfunctioning agents, which requires some degree of centralization as well. The overall architecture of our system is shown in Figure 2.

The server component is stationary, dispatching agents, collecting information and persistently storing the current status of all systems and networks. (For the network management community in the Internet, this could be compared to the Management Information Base (MIB), being accessed via SNMP [1]. Another task of the server component is to provide access to the GUI client when launched from a (remote) machine. The systems administrator should be able to access the management system from any machine that provides Web access and supports Java, thus the client component is a Java applet being launched remotely and connecting back to the server. The intelligent agents are dispatched by the server and access the systems that are to be managed. The management server defines the goals for the agents (management policies, see above); the individual agents should have enough intelligence to achieve these goals without any further direct control. The agents should even be able to explore the systems in the network on their own (auto-discovery is an essential function of today's network management systems), monitor them autonomously, classify events into marginal or relevant, and take appropriate action, in order to achieve the goals that were set before. All this happens (for the most part) without interference and maybe not even notion of either users or system administrators. An agent only initiates notification or interaction with either one when situations occur that can not be handled in an automated way.

3.3. The components of our architecture

A more detailed view of the system is shown in Figure 3. It shows the essential components together with their interactions.

3.3.1. The GUI client. The applet provides menu choices to open and save process configurations stored in files that were created either by the monitoring system itself or by the user who changed the process distribution for simulation purposes. The most important option is the one that allows beginning the analysis of an application or set of applications running on a set of different of processors. The monitoring is done off-line and the results are stored in a file as mentioned above. Once such a file is opened, the user can display the information in either the distribution window, showing the processes in every processor and the communication between processes, or in the cost window, showing the utilization of the different processors (processing, I/O, communication), and the

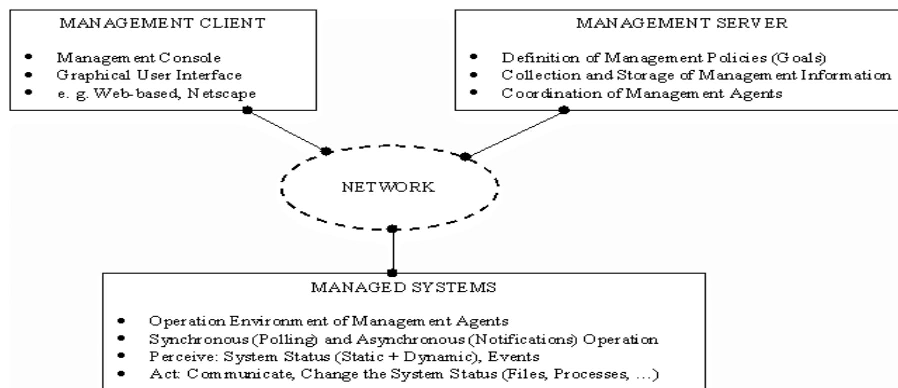

Figure 2 General Architecture

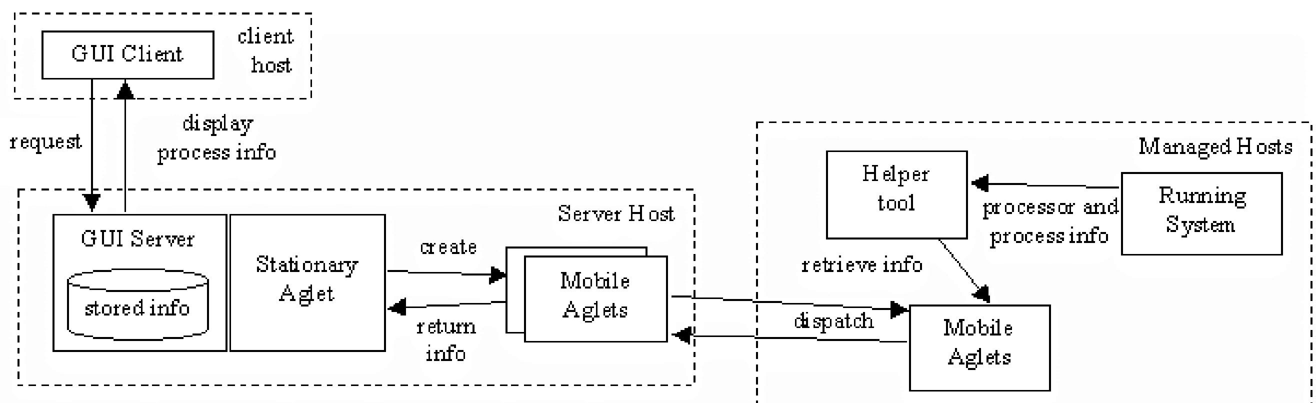

Figure 3 System Components

portion of these times dedicated to the application(s) in question. The cost window will show the time every processor is idle and the network traffic between processors. Figure 4 shows a distribution window where circles represent different processors (systems) and a single process is shown as a filled square whereas thread is shown as a hollow square. When pointing with the cursor on a process or a thread the information of the execution features of that specific process or thread is displayed. Communications between processes are depicted with arrows, with different widths representing the amount of network traffic that is generated.

3.3.2. The server component. This component provides access to the GUI applet. Once the applet is launched, it connects back to the server and starts the monitoring request formulated in the monitoring options dialog. The design of the server component also includes the inference engine / rule base that provides intelligence to our architecture. This can be accomplished using IBM's Agent Building Environment (ABE): A rule-based forward chaining inference engine. Rules are stored in the KIF format and can be edited via a graphical rule editor, besides direct ASCII editing. A number of adapters, some are provided, some can be written in C++ or Java, provides sensor / effector capabilities to the inference engine and actually start the inferencing process. The user can write adapters himself, possibly in Java. The server basically provides a "home base" for the mobile agents and stores the information they gathered.

3.3.3. The mobile agents [4]: The server has an important component that provides the ability to fetch the necessary information about processors, processes and communications in the specified remote hosts. This component is based on mobile agents technology in form of mobile Java Aglets[1], using IBM's JAAPI [3], the Aglets Software Development Kit. Aglets are little Java applets that are able to autonomously

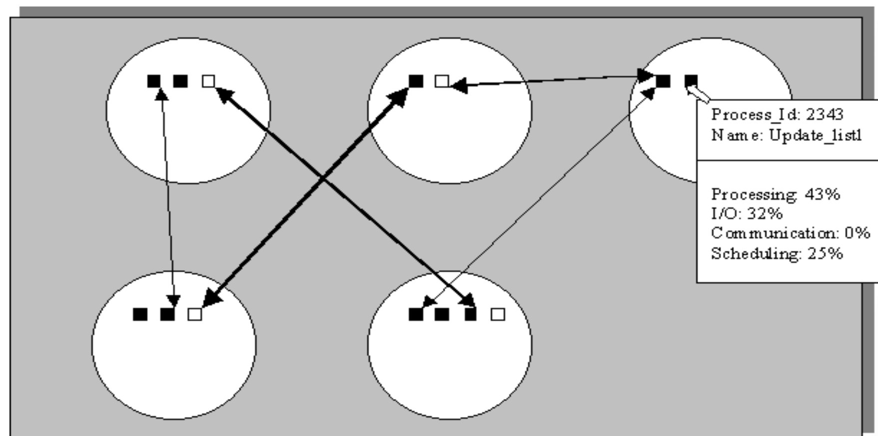

Figure 4. Distribution Window

travel over the network, clone themselves, create other aglets, send messages to other aglets, serialize themselves for persistent storage, etc. While they travel they maintain their state of execution and resume execution upon arrival. This component of the server has the capacity to create aglets, dispatch them and communicate with them. In our case these aglets are dispatched to remote hosts in order to retrieve information about processes / applications that are running on them. They send messages back to their server component containing the information they retrieved. There will be two types of agents, a stationary one, that communicates with the rest of the server application and manages the other type, the mobile agents that actually travel to the managed hosts.

3.3.4. The Helper Tool. We still need an additional 'helper tool' that is installed locally on each managed host and retrieves information about processes. It provides cumulative / average values that can be picked up by the incoming aglet. Thus the aglet will not be concerned by the specific system platform of the host it was dispatched to, increasing their platform-independence. Only the helper tool is system-dependent. Using this tool can be considered a work-around for the problem that Java does not provide access to the "guts" of a system as we would have needed.

4. Implementation

4.1. Working environment

We have implemented the server component on a UNIX platform. The client component can be launched from any computer that supports Java. The Aglets Workbench is entirely written in Java and is thus platform-independent. Each monitored host needs to have a copy of the Aglets Workbench installed and the Aglets daemon running in order to allow aglets to reside on them. Applets are a peculiar kind of programs as they are only intended to be executed in the context of an HTML file. This places some rather severe restrictions on what you can do in an applet to protect the environment in which they are executed. The process of storing and retrieving objects in an external file is called serialization. Writing an object to a file is referred to as serializing the object, and reading an object from a file is called deserializing an object. The way that components are arranged in a container is usually determined by an object called a layout manager. This layout manager for a container determines the position and size of all the components in the container. The Vector class defines a collection of elements of type Object that works rather like an array, but with the additional feature that it can grow itself automatically when you need more capacity. Because it stores elements of type Object, and Object is a superclass of every object, you can store any type of object in a Vector. This also means that you can use a single Vector object to store objects that are instances of a variety of different classes. We have not used this tremendously flexible feature.

4.2. Development of the GUI – Classes and methods

The GUI is a Java applet that behaves as a client of a server application. We will discuss the client-server implementation below, as we will begin with the description of the main classes that form the GUI. There are four main classes that belong to the applet:

Tool: this is the application class, which has overall

control of the rest of the classes.
ToolFrame: this is the application window class, where the main window is defined.
ToolDoc: this is the document class, which contains the data that defines the kind of object the application is concerned with.
ToolView: This is the view class, where the way of presenting the data of the document class is described.

In a GUI applet like ours, in which the user needs file retrieving and saving operations, and to trigger the monitoring operation (which needs access to other hosts, obviously), the solution adopted is to create a client-server application. The applet acts as a client, and there is a Java application, ToolServerImpl, that acts as a server. They communicate through RMI, that we will explain below with greater detail. The architecture outlined above can be implemented through Remote Method Invocation (RMI). RMI is an object-based distributed computing architecture written completely in Java. In distributed object computing, an object calls procedures contained within another object. These objects may not reside on the same physical machine, and then, the access is made through the network. RMI is Java's way of delivering distributed object computing to Java objects. Thus, through RMI, a Java client can access another Java object running on a separate virtual machine (the "remote object"). The Java applet (client object) invokes methods contained within the remote object, which in turn can perform some task and return the results back to the client. The applet still behaves as if it were a self-contained applet. It "thinks" these methods are local to itself, passing values and receiving results with the remote object. It doses this through the use of a registry. At the start of the process, the Server executes as a background process on one machine. The server application identifies itself with the RMI registry, on that same machine. The registry is another Java application whose role is to maintain active references to remote objects. Next, the Java applet that acts as the client running on another machine wishes to invoke a method contained within the remote object. Before it can attempt this, however, it needs to request the service from the registry. It does this by accessing the RMI registry on the host where the remote object resides through a TCP socket connection. Once the registry accepts the client's request, it delivers a reference to the remote object in the form of a stub back to the client. The client now invokes a method contained within the remote object's stub. The parameters are exchanged through the network using object serialization. Parameters are forwarded to the remote object through a skeleton object. The skeleton then dispatches them to the remote object and invokes the object's method.

4.3. The Back End –Concepts and technologies

The back end of our architecture consists of two components:
Mobile Java Agents, which provide remote access to the processors on which monitoring is to be performed
Local tool(s) which are necessary to gather the low level information from inside the running system

We implemented the mobile part of our architecture by using IBM's Java Agent API, ASDK. Aglets are a concept built on Java Applets by extending applet capabilities to create a somewhat autonomous behavior, which is what makes them appear as software agents. The core of the architecture is the Aglet Transfer Protocol (ATP), provided by the aglets daemon (agletsrv), that has to be operational on every host that is supposed to utilize aglets. These actions are facilitated via Java RMI (Remote Method Invocation) and Java's Object Serialization. The activities of an aglet are mainly implemented in methods that are derived from the Aglet base class and that form the lifecycle of an aglet. E. g. when created, an aglet's OnCreation method is invoked, and when dispatched to a remote context, first the OnDispatching method is invoked, then the aglet is transferred, and as soon it arrives, the OnArrival method is invoked. The overall concept is a callback model: Upon certain events the ATP daemon calls the appropriate methods in the aglets classes provided by the programmer.

The autonomous behavior of aglets is created by having aglets perform these actions on themselves or other aglets. An aglet context provides proxies to aglets by identifying a particular aglet with a unique aglet ID. Once these "handles" are given, aglets can perform the above actions on any aglet they have access to. Since aglets are basically given all the functionality that can be implemented with Java, security becomes a very important issue when implementing actions that a host allows an aglet to perform. Along with the ATP daemon comes a GUI to manage aglets interactively, called Tahiti, in which security options can be set. Aglets are distinguished into trusted and untrusted aglets, and each of them is given selective read or read/write access to the host's file system. The ATP uses a new type of URL to locate an aglet in a context: atp://some.host.name:<port> . The code base of an aglet (that is the compiled class code of the aglet) is supposed to be located in a directory within the path variable AGLET_PATH, expecting a Java package tree in

this directory. Usually AGLET_PATH equals AGLET_HOME/public, where AGLET_HOME is the directory where the Aglets Workbench was installed. With regard to our architecture it is important to note that the Aglets Daemon has to be running on every host on which process monitoring is supposed to take place.

5. Experimentation:

The following problems came up during our experimentation while putting this technology in a prototypical implementation.

5.1. Design Issues

As we tried to develop our application up to the point where it can retrieve actual "live data" about processes on remote machines, we encountered a set of design issues that revealed a problem complexity significantly higher than we expected. These are the major design issues that need to be addressed:

1. One application consists of several process classes and there are several instances of one process class at a point of time. It is not trivial to determine what process classes are to be monitored concerning one application. The name of a process does not necessarily reveal this association. If we had to monitor X-Windows processes running in a system, there would be a number of process instances of the class "xterm" – do we need accumulated information of all xterms in general or more detailed for every single instance?
2. One process instance appears at several "snapshot" times, but not necessarily all of them. Back to the X-Windows example, xterm processes appear and disappear frequently over time. How do we account for processes at times inside our monitoring period when they do not exist in the system?
3. Several instances of either the same or a different process class communicate with each other. According to how the number of entities increases that we are monitoring, it figures that monitoring their communication will be even more complex.

5.2. Implementation Issues

Regarding UNIX platforms as a restriction for the applicability of our tool, there are several points to consider when implementing the helper tool that retrieves the process data from a running system:

1. Process identification. In a UNIX system processes are identified by their process ID, whereas our tool looks for processes associated with a certain application by their name.
2. Process State. Whenever our tool takes a snapshot of the process table, it will never find any processes running, because only one process is running at any given time, and in this particular moment it will always be the tool itself!
3. Process communication. In our experiments we could not really figure out, how to measure communication between two processes. First of all, there are several ways how this happens, e. g. pipes, sockets, shared memory, etc. Regarding sockets for example, it would be necessary to know the ports that the processes in question are using for their communication. Then our tool would have to listen on all these ports.

6. Conclusion

The purpose was the development of a platform-independent tool to monitor the distribution of processes of an application on different processors in a distributed environment. Mobile agent technology seems to serve pretty well for the purpose of distributed process monitoring, due to the parallel operation and mobility of agents. Most of the problems we had with this work are caused by the project setting, i. e. limitations on time resources, and by technological complications, i. e. lack of a proper development environment and the need to use an agent framework that still needs to mature to be effectively useful.

7. References

[1] Morris Sloman ed., "Network and distributed systems management", Addison-Wesley, Reading, MA, 1994.
[2] IBM's Web site for Aglets: [ASDK] IBM Aglets Software Development Kit, http://aglets.trl.ibm.co.jp
[3] D. B. Lange, M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", Addison-Wesley Pub., 1998.
[4] W. R. Cockayne, M. Zydd, "Mobile Agents", Manning Publications, Connecticut, 1998.

---

[1] Aglets: The aglet is the next step in the evolution of executable content on the Internet, introducing program code that can be transported along with state information. Aglets are Java objects that can move from one host on the Internet to another.