The following paper was originally published in the
Proceedings of the Twelfth Systems Administration Conference (LISA '98)
Boston, Massachusetts, December 6-11, 1998

System Managment With NetScript

Apratim Purakayastha and Ajay Mohindra
IBM T. J. Watson Research Center

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

System Management With NetScript

Apratim Purakayastha and Ajay Mohindra – IBM T. J. Watson Research Center

ABSTRACT

Cost and complexity of managing client machines is a major concern for enterprises. This concern is compounded by emerging client machines that are mobile and diverse. To address this concern, management systems must be easy to configure and deploy, must handle asynchrony and disconnection for mobile clients, and must be customizable for diverse clients. In this paper, we first present NetScript, an environment for scripting with network components. We then propose a management system built with NetScript, where mobile scripts invoke components to perform management operations. We demonstrate that our approach results in a flexible, scalable management system that can support mobile and diverse client machines.

Introduction

Managing client machines in an enterprise is a challenging problem. Typical management operations include installing/updating applications, changing system files, monitoring performance, tracking client activities for diagnostics, and taking backups. The problem is compounded by certain characteristics of emerging clients. First, even within an enterprise, clients are heterogeneous – a mix of UNIX workstations, desktop PCs, laptops, palmtops, printers, and copiers. Management systems designed for traditional workstations are not readily applicable to laptops and palmtops. Second, emerging clients are often disconnected. Management systems therefore must support asynchronous and disconnected operations. For example, a management server should not fail, block, or have to actively retry a management task for a client that is disconnected (asynchronous server operation). On the other hand, a managed client should be able to prefetch and cache the right resources so that it is able to continue management operations when disconnected (disconnected client operation). Third, emerging clients are being used in more diverse application domains. Management systems therefore must support adequate customizability. In addition, management systems should themselves be easy to configure and deploy.

Current management systems and standards, such as Tivoli's TME-10 [10], Computer Associates' Unicenter [11], SNMP [13], CMIP [14], and DMI [15], are primarily designed for traditional workstations and desktops that are mostly connected and have enough local resources to host reasonably heavyweight client-side management agents. They are hard to deploy and configure, they do not support asynchronous and disconnected operations, and they are not well-suited for a heterogeneous environment.

NetScript[1] is an environment for scripting with network components. In NetScript, a developer selects required components from a distributed catalog and then writes a script invoking methods on these components as if the components are local. When a script is launched, the NetScript runtime dynamically determines component sites in the network and transparently migrates the script as needed. A remote component can also be transparently downloaded to a site where the script is currently executing.

The NetScript environment can be used in system management as follows:

1. Organize management code into appropriate components.
2. Use scripts to define management tasks that migrate to managed clients and dynamically locate and use the components at runtime.

The above approach offers a number of benefits. First, it extends the notion of scripting transparently to the network. Second, it centralizes management of components and scripts. Since management components can be downloaded at runtime, little management code needs to be pre-installed on managed clients, thereby drastically reducing configuration and deployment costs. Third, its script mobility naturally supports asynchrony and improves scalability by reducing load on the management server. Finally, its Java implementation improves portability.

The rest of the paper is organized as follows: the NetScript environment, illustrative examples of scripts and components, NetScript-based solutions to key system management problems, the technical challenges in using the NetScript environment for system management, related work in this area, and finally the conclusion. The appendices list a few scripts that are referred in the paper.

The NetScript Environment

A NetScript programmer writes a script by first selecting required interfaces from a distributed catalog and then invoking interface methods as if the components implementing the interfaces are local. An end user launches such a script into the network, where the NetScript runtime dynamically determines the

[1] A prototype implementation of NetScript is freely available at http://www.alphaworks.ibm.com/formula .

component sites and transparently moves the state of the script to the component sites as necessary. It is also possible that the NetScript runtime downloads a component to the script execution site, or migrates both the script and the component to a third site.

The rest of this section summarizes different aspects of the NetScript environment including its component model, scripting language, and the runtime environment. This discussion is intended to provide only a reasonable background for the rest of the paper. Please refer to [2] for a more comprehensive discussion of NetScript[2].

The Component Model

The NetScript component model is based on well-known object-oriented programming concepts of interfaces, components, attributes, and globally unique identifiers. In NetScript, an interface is a group of semantically related methods or functions. A component is an implementation of the interface. A component can implement one or more interfaces. Each interface and component is identified by a globally unique identifier (GUID)[3], called InterfaceID and ComponentID respectively. Interfaces and components may have attributes associated with them. Attributes may provide informative description (such as semantics and suggested use) or other parameters (such as manufacturer, usage cost) that may allow the NetScript runtime to select one component over another. Attributes are also identified by GUIDs called AttributeIDs.

Components in NetScript could be Java Beans or Java-wrapped native (e.g., ActiveX) components. Interfaces and components are advertised in a distributed catalog. Currently the catalog is implemented using the LDAP [3] distributed directory services. Interfaces and components are hierarchically organized in the directory. Currently only browsing function is supported whereby a programmer can browse the catalog and select required interfaces. Support for a more intelligent search function is planned.

The Scripting Language

The NetScript scripting language is an extension of the BASIC programming language. In addition to standard control constructs, the language has a few NetScript specific additions to create component instances and to make method invocations. For security reasons the language has no vocabulary for system operations such as direct memory access, file access, and network access. The language is deliberately kept simple for it to suffice as a glue language that glues components together.

For creating components, the language provides the createComponent keyword. The syntax is:

```
<varname>=createComponent(<interfaceName>,
     [<filter>]) [at <locationName>]
```

where varName is a handle to the resulting instance, interfaceName is the InterfaceID that the component should support, and filter is a boolean expression of desired attribute name-value pairs. Optionally, using the at keyword, one can also specify the locationName indicating where the component should be instantiated. Under the covers, the runtime contacts the component catalog to locate a component that implements the required interface and then either migrates the script to the component location or downloads the component to the script's location, instantiates the component, and stores the handle for the instance in the script's varName variable. The syntax for performing method invocations is as follows:

```
[<resultVar>=]
   <varName>.methodName(<arg1,...,argN>)
```

where resultVar is the variable to store the results of the invocation, varName is the handle for the component instance, methodName is the name of the interface method, and <arg1,...,argN> is the list of arguments.

We use a BASIC-like scripting language for reasons of free experimentation and convenience, including the fact that a free Java-based interpreter for it was available that we could modify and add our extensions easily. Use of a more standard scripting language like Tcl or Javascript is planned.

The Runtime

The NetScript runtime has been implemented in Java. The runtime has a script interpreter, an execution engine, and a set of shared services such as directory, instance management, and communication. The runtime also has built-in support for security and access control, garbage collection, monitoring, and failure detection and recovery. The NetScript environment also has command-line and web-based tools for the user to launch, monitor, and control executing scripts. Figure 1 gives a broad overview of the important parts of the NetScript runtime.

A script is first launched on a machine using one of the NetScript tools. The NetScript runtime initializes some data structures and starts executing the script. When executing a createComponent call the runtime may need to migrate the script to another runtime executing at a different host. Parts of the script including program counter, stack and data heap are transferred, while component instances that the script may have created locally are kept as part of the residual script state. Every runtime has a per-script instance manager that keeps track of component instances that the script may have created. On method invocations, the runtime queries the instance manager to decide if the script needs to migrate to the location of a component instance. On completion, the script typically returns to the home machine from where it was launched. The runtime at the home machine then

[2]NetScript was called NetPebbles at that time.

[3]A GUID is generated using a combination of hardware address, current time, and a random long integer.

initiates garbage collection to clean up instances and data structures that may have been created by the script on other hosts.

An Example

In this section we discuss a motivating example of how NetScript may be used to perform certain system management tasks. First, we discuss an example script by walking through its execution assuming certain components are available for the script to use. Next, we discuss how one may write one of the constituent components and incorporate the component in the NetScript environment to make everything work together.

An Example Script

Figure 2 shows an example script that discovers the version of an application (such as Lotus Notes) installed on every machine in a department and then displays the results on some specified machine. Figure 3 provides a visual analog for the execution of the script that shows which parts of the script execute on which machines, how the script migrates, how the runtimes interact with the component catalog and download components.

The script starts on an admin machine called "helix." The script first attempts to create a component that implements the interface "IDomainAdmin." The runtime uses the component catalog to locate the actual host for a component that implements the interface. When such a host is located, the script execution is suspended and the script is migrated to that host. The NetScript runtime on the destination host, which happens to be the component server in this case, instantiates the component and resumes script execution. The script invokes the "getMembers" method on the instance to get all member machines for "Department 931B." Then for every machine in the department, the script uses the "createComponent" function with the "at" clause to migrate to the machine as well as download the component that implements the

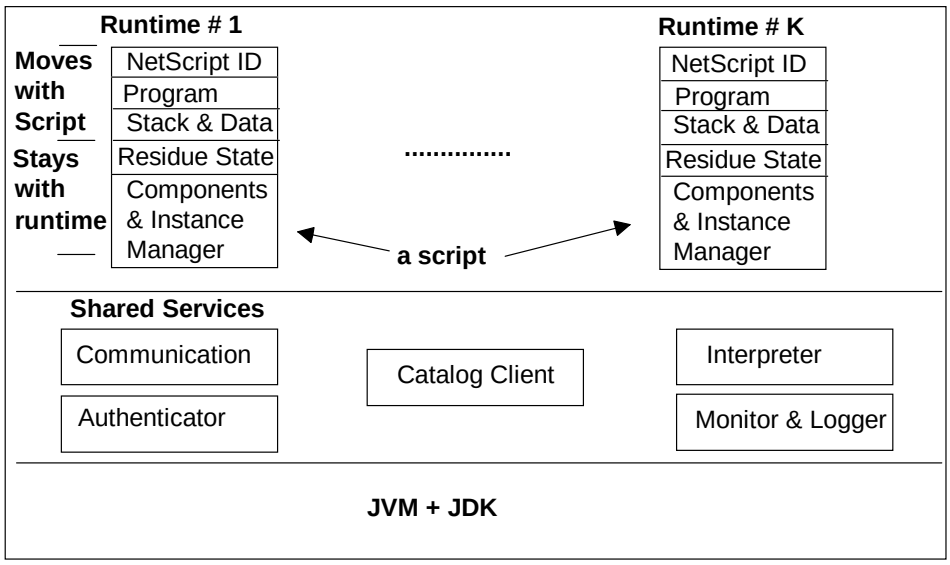

Figure 1: The NetScript runtime: scripts and shared services.

```
home = "helix.watson.ibm.com"
a = createComponent("IDomainAdmin")
locations = a.getMembers("Department 931B")
n = length(locations)
dim ver[n]

for (i=0; i < n; i=i + 1)
    b = createComponent("ISystemSniffer") at locations[i]
    ver[i] = b.getVersion("Lotus Notes")
endfor
c = createComponent("IDisplay") at home
c.showList(ver)
exit
```

Figure 2: A script for finding out Lotus Notes versions installed on department machines.

interface ''ISystemSniffer.'' Like before, the runtime uses the directory service to locate a component host but instead of migrating to the component host, the component is downloaded from the component server to where the script migrates as a result of the ''at'' clause. The runtime at each machine instantiates the downloaded component and resumes the script execution. The script invokes the ''getVersion'' method on the instance to obtain the version of ''Lotus Notes'' and stores in a script array. Finally, the array is displayed using the ''IDisplay'' component at a specified machine. To improve performance, the NetScript environment also allows a user to ''fan-out'' a script simultaneously to a number of machines.

An Example Component

The components in this example can be written in pure Java, or can be Java-wrapped native code. The ''IDomainAdmin'' interface is one that manipulates user information. Conceivably the component implementing the interface can be written in pure Java that perhaps uses JDBC to perform actual database operations. Alternatively, it can also be implemented as a component that reads and updates simple files that contain user information. As long as the component supports the interface methods and semantics, its different implementations does not necessitate changes to the existing scripts. The component implementing the ''IDisplay'' interface can also perhaps be a pure Java component that uses 'java.awt' methods for user interaction.

The component of interest however, is the one that implements the ''ISystemSniffer'' interface. In its generality, this interface will support not only methods that get version information for applications, but perhaps methods for sensing load conditions, memory availability, disk-space availability, battery power, etc. Since these operations are platform dependent, the component is likely to contain native code. In Appendix A we have outlined a simple implementation of the component that supports only the ''getVersion'' method for a few applications. Appendix A.1 lists java code that is only a scaffolding for the underlying native code in the C language. The Java Native Interface (JNI) is used to communicate across Java and C. Appendix A.2 lists the C code that implements the ''getVersion'' method for a Windows 95 or Windows NT platform. The C code simply traverses the windows ''registry'' and looks for specific key-value pairs to determine the version information. The native implementation will clearly differ if the same method is implemented for an AIX system that has different ways of storing application infromation.

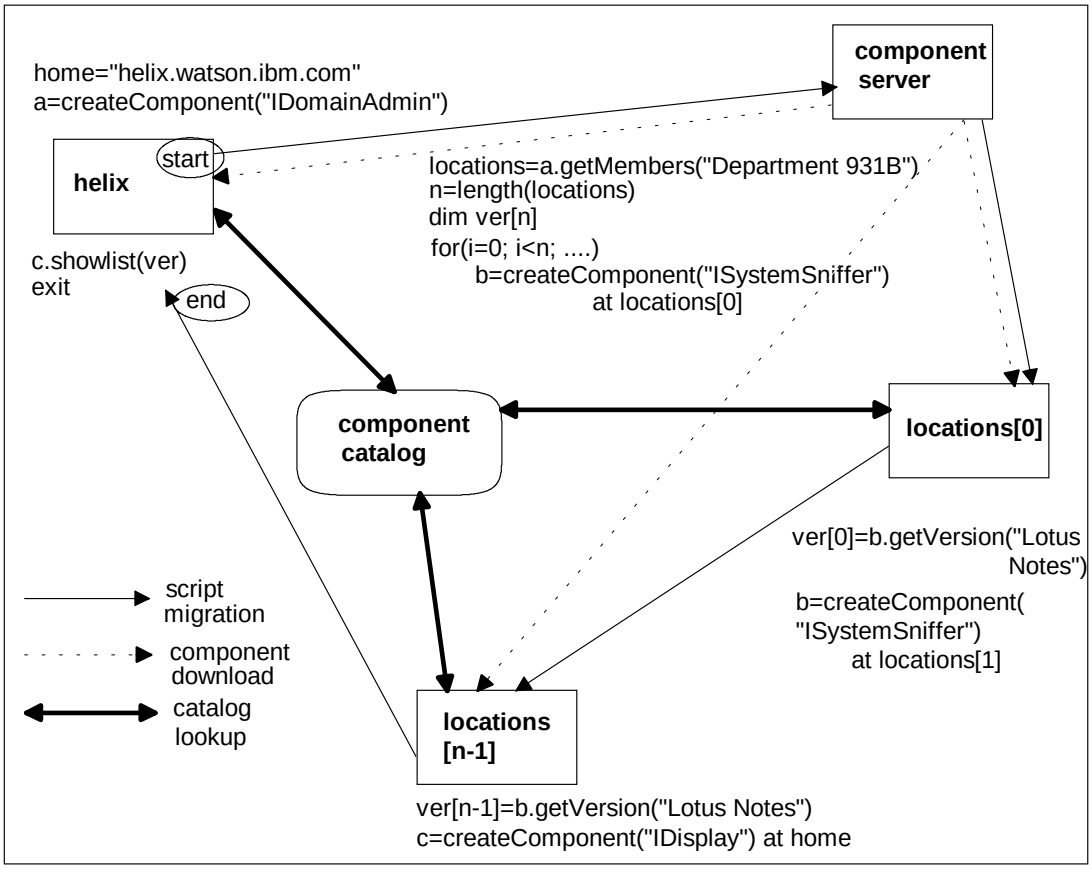

Figure 3: Visual depiction of the execution of the example script.

The classes and native libraries (if any) such as ‘‘dll’’s for Windows systems and ‘‘so’’s Unix systems are packaged as Jar files for distribution. The component is entered in the component catalog at its appropriate position. The component entry in the catalog specifies its location and the name of the Jar file in which the component is packaged. A NetScript runtime finds component location information from the catalog, unpacks the Jar file, and instantiates the component.

The above example embodies a simple programming paradigm where powerful components on the network are threaded together to perform an actual management operation. The NetScript environment itself does not attempt to dictate what system functions should be captured in reusable components and what should be captured in scripts. In system management, perhaps ‘‘functions’’ such as looking up a registry/database, copying/moving files, and checking process/memory status should be captured in components, while ‘‘decisions’’ or ‘‘control’’ such as identifying when memory is low, deciding that an update needs to be applied should be captured in scripts. Note that the intent of NetScript is not to write ‘‘core’’ management code but to help organize core management functions into components such that actual management tasks can be easily specified in simple scripts.

NetScript Solutions for System Management Challenges

Modern management systems need to be flexible. Changes in management infrastructure should not mandate a large-scale re-deployment on managed clients. In addition, the system should allow high levels of customizability for clients without the need to modify client-side code and hence forcing re-deployment. Modern management systems should also be able to contend with client disconnection and heterogeneity in managed clients or devices. In this section we will discuss how the NetScript approach can address these challenges.

Flexibility

Many management systems, including standards and products, rely on a management agent installed on a managed client. The management agent on the client is controlled by a management server. The main problem with this approach is that when a management agent is deployed, it becomes a fixed contract between the server and the managed client. Therefore any change in the behavior of the management system implies the client agent be changed and re-deployed. A second problem is customizability. Although some preprogrammed customizability can be added to the client agent, such an approach will be untenable as clients and users get more diverse.

NetScript solves this problem naturally and elegantly. All that needs to run on a managed client is the NetScript runtime (only 40KB static footprint at this time). The runtime is generic and simple and is not specific to system management. It can also be used for executing the client’s personal scripts, e.g., one that tracks stock prices for the user. The runtime is also capable of running as a servlet under a web server such that the runtime itself can be downloaded when appropriate. A NetScript-based management system closely approximates a zero-install client.

In NetScript, management components can be centrally stored and managed in some server. They can be changed freely without the need for deployment. The scripts can also be maintained centrally. The ability of the scripts to migrate to clients and download required components precludes the need for modifications at the client.

Appendices B.1 and B.2 illustrate how flexible and customizable a NetScript-based management system is. Appendix B.1 shows a script similar to the one in section 2. This script updates an application on all machines in a department[4]. There could be times when some clients will not want this update. For example, the reason could be that the client is connected over a slow phone line, or the client is making an important presentation. To incorporate such customizability into the system, the system administrator only needs to write a ‘‘Update Policy’’ component (which can be made arbitrarily powerful) and then change a few lines in the script (shown in italics in Appendix B.2). Note that no changes or deployment at the client is necessary to incorporate this customization.

From these examples one can infer that the basic philosophy of componentizing management code and threading those components at runtime via a NetScript script results in a management system that is immensely flexible and customizable. In addition, the NetScript approach results in better scalability because ‘‘control’’ migrates with the script. The central management server is not overloaded trying to coordinate activities of various client agents.

Asynchrony and Disconnected Operations

Users in modern enterprises are increasingly using laptops and other mobile devices that are frequently disconnected. Modern management systems must therefore contend with disconnection. First, servers should be able to schedule or launch management tasks anytime independent of whether clients are connected or disconnected. Without an explicit attempt-and-retry, the management system should be able to implicitly propagate appropriate tasks when clients reconnect (asynchrony). Second, the clients may choose to disconnect when a management

[4]For simplicity, the scripts show machine updates in a serial fashion. In practice that will not work well. NetScript environment includes support to launch scripts in parallel to various machines.

operation is in progress. The runtime on the client should be prepared to handle such disconnections and should prefetch appropriate components to allow the management operation to continue off-line (disconnected operation).

The NetScript runtime has implicit support for handling network disconnections. When a running script needs to migrate to another location, the runtime periodically attempts to transfer the script in configurable intervals until it succeeds or exhausts a configurable timeout.

Supporting disconnected operations in NetScript requires prefetching of appropriate classes in preparation for disconnection. As a first attempt we are considering supplying annotations in the script to help the runtime prefetch the right components. This approach is more practical than the runtime trying to be intelligent about the semantics of a script. We are also considering support for persistent scripts that are saved when a client shuts down and are revived when the client restarts. Support for disconnected operations and persistence is being designed and has not been implemented yet.

Heterogeneity

With the increasing popularity of networked devices it is reasonable to assume that in the near future managed systems will not only include traditional workstations and desktop PCs but also include devices such as laptops, palmtops, printers, and copiers. The managed system therefore needs to support heterogeneous and diverse set of clients. They will have different modus operandi, different operating systems (some of them will not even have one), and different functional roles. In some, application management will be important (laptops), in others function and capability management will be important (printers), and in yet others pure data management will be important (palmtops). If a management system even wants to begin to address this wide range, it has to be lightweight, portable, and adaptable.

NetScript is reasonably lightweight (40KB footprint). Since it is downloadable, it can also free up valuable space for more important applications in space constrained palmtops. It is portable to any Java-capable platform (may also be packaged with a lightweight Java Runtime Environment). It is also adaptable to availability of resources such as the network.

Consider printers as an example of a new type of managed client in a comprehensive management system.

Printers have errors such as toner low, paper out, paper jam, memory overflow, bad configuration, etc. Today we discover those errors only when we go to pick up a printout, even though the printer firmware has already detected and reported the error on its console. In the near future it is reasonable to expect for printers to have IP addresses and be Java-enabled. With a NetScript-based management system, a server can send down a monitoring script to the printer, which basically snoops on the printer, and upon error, migrates to notify appropriate parties. Appendix C.1 shows a NetScript script that may be used for this purpose. The script uses the ''IPrinterManager'' component to obtain the printer's location, then migrates to the printer and downloads the ''IPrinterAssist'' component that is able to access the printer firmware. The script then waits in a loop checking for error conditions. When an error is detected, it first tries to fix the error if possible, or else it notifies the administrator. One can also imagine scripts that can visit a number of printers and enqueue a specific job in a printer that has perhaps the shortest queue length or the smallest total size for all jobs in the queue.

Issues in using the NetScript Environment for System Management

The practical use of NetScript in a real management system depends on a number of factors such as acceptability of its programming model including the scripting language and the component model, security, reliability, and monitoring and debugging support. This section discusses these issues in the NetScript environment.

The Scripting Language Dilemma

The NetScript scripting language is an extension of BASIC. It is purposely kept quite simple such that scripts are easy to write and modify. It is however, unfamiliar and not as prevalently used such as Tcl [4], Perl [5], or Javascript [6], therefore its acceptance in the system management may be in question. Scripting languages are usually extensible and therefore it should not be difficult to incorporate NetScript specific extensions (such as createComponent) into other languages. Why then, are we not using Tcl extensions? The reason is that scripting languages such as Tcl are quite powerful and can directly access files and sockets. Such capabilities might be undesirable (for reasons of security and simplicity) in a mobile script such as NetScript. We may consider using a proper subset of Tcl (such as safe-Tcl [7], with NetScript extensions) in the future. Since NetScript is currently under active experimentation, we used a language for which we found a public domain Java-based interpreter that we could easily modify to suit our needs.

The "Non-Standard" Component Model

The NetScript component model borrows from the Java Beans model as well as the ActiveX [8] model but is also distinguishable from both. The properties and interfaces supported by a bean cannot be ascertained unless the bean is instantiated and introspected. We believe that a system manager needs to know about the functional characteristics of a component before instantiating it. Therefore, like the

ActiveX registry we use a component catalog (implemented on LDAP distributed directories). However, in ActiveX, the programming model is component centric because a component is first located in the registry then it is ascertained what interfaces it implements. In NetScript the programming model is interface-centric. When writing scripts, the system manager simply chooses some functionality (syntactically and semantically characterized by an interface) from a catalog and the runtime locates that functionality in the form of a component.

Security

With mobile code such as in NetScript there is always a security concern. A malicious ''management'' script can harm a client. The users of the scripts must be authenticated and authorized to execute on certain machines or use certain components. NetScript provides mechanisms for attaching a ''principal'' with a script. Upon receipt of a script, the NetScript runtime authenticates the principal and verifies that the principal has execution rights on the local host. When a script accesses a component, the principal attached to the script is provided to the component catalog. Component entries in the catalog list principals that are allowed to use the component. Only if the provided principal in included in the list, a component location is returned to the requesting runtime. We have implemented a DCE-based [9] authentication and access control mechanism with NetScript that is suitable for intranet deployment.

Security issues however, reach further than simple authentication and access control. A NetScript runtime should prevent two different NetScripts from interfering with each other (isolation). The runtime uses per-script instance managers and class loaders to implement isolation. The NetScript environment also does not allow for editing a running script because of possible security breaches.

Reliability

Reliability is also a concern with mobile code. When a management script dies somewhere how can the system manager find out where it died and what actually happened? For scripts that are long-running (e.g., one that monitors network load on a router), what happens if the system is re-booted? How does one ensure that all the component instances of a failed script are garbage collected? Is it possible to recover a script after the host on which the script was executing crashes?

To survive across machine reboots, we have implemented mechanisms whereby a NetScript runtime responds to the ''shutdown signal'' (available in most operating systems) by saving its internal data structures and scripts in persistent storage and recovering from persistent storage at system startup. We have also implemented a version of the NetScript runtime that uses a reliable and non-blocking application-to-application transfer protocol called MQSeries [22]. The use of a reliable and non-blocking transport mechanism has allowed us to design simple protocols for reliably tracking a NetScript. We have also implemented limited checkpointing capabilities that allows some scripts to be recovered after a system crash.

Monitoring and Debugging

To be successful as an extensible, programmable environment, NetScript must include reasonable support for monitoring the excution of scripts and debugging scripts. Tightly controlling the execution of mobile code is a difficult problem. The NetScript environment, however, provides support for locating a script, retracting a script from any location, or killing a script. Scripts are identified by GUIDs generated and assigned to a script at the start of its execution. Any errors that are encountered by a script, including exceptions generated by components, are reported to the user when the user requests for the status of a script that has failed. Features like break-points, single-step execution are being considered but are not currently implemented.

Related Work

NetScript and Other Management Systems

Architecturally, most management systems such as Tivoli's TME-10 [10], Computer Associates' Unicenter [11], Marimba [12], IBM TJ.Watson Research's SysCtl [23], and Igor [24], rely on a installed client agent that works under commands from a central server in a request-response fashion. Network management standards such as SNMP [13] and CMIP [14] also imply the same architecture, and so do desktop management standards such as DMI [15]. For reasons cited in Section 2 this approach is fundamentally not flexible and scalable. Enhanced functionality often results in making the client side agent even more complex, thereby making it harder to maintain, re-deploy, and configure. None of the above systems also naturally support asynchronous operations although one can imagine retrofitting such function to them. None of the above systems (including implementations of the standards mentioned) are portable across operating systems, let alone different device classes such as desktops, laptops, palmtops, and network devices.

Management by Delegation (MbD) [16] partially addresses the scalability and flexibility limitations of SNMP-style systems. MbD proposes an architecture whereby delegation agents could be sent down from servers to clients that could then invoke delegation procedures stored on clients. MbD however, is not quite as flexible as NetScript because the delegation procedures themselves are not downloadable. The programming model is also not based on scripting, which is popular in the system administration community.

NetScript and Other Infrastructure Technologies

One may argue that given suitable management components one can build a similar management system using technologies like Java/RMI [17], CORBA

[18], or DCOM [8], or mobile agent technologies like IBM Aglets [19], or Agent Tcl [20], or Telescript [21], why use NetScript?

With Java/RMI technologies, per-component ''ImplServers'' have to be running on component hosts. This can cause difficulty in deployment. Moreover, for long running methods (such as monitoring a printer) RMI-like technologies will need to maintain a long running connection. Remote polling using short lived connections is not scalable.

Functionally a mobile agent technology such as IBM Aglets can do whatever NetScript can do. However, the script and component based programming model in NetScript is appreciably simpler. Same functionality is substantially easier to code and deploy using NetScript (see [2]).

Conclusions

In this paper we have summarized a technology for scripting with network components and argued for its gainful use in system management. Our prototypes have shown that using the NetScript technology in managing systems can overcome problems of flexibility/customizability, client mobility, asynchrony, and disconnection, and heterogeneity. At the same time we believe that NetScript provides an attractive and simple programming model for the system administration community.

For a fully functioning management system, components that do the actual work still have to be written. Some components can be purely Java and hence usable in all Java-enabled managed platforms. Some components will have to use native code for a long time to come (e.g., registry access component for Windows). These pieces of code have to be written in any management infrastructure that wants to perform the same functions. NetScript's contribution is proposing an elegant way to program with these components as available, granted pieces of function. As a result, the management infrastructure built around those components becomes more flexible, scalable, customizable, portable, and above all, ''modern.''

Author Information

Apratim Purakyastha received his Ph.D. in Computer Science from Duke University, where he was awarded a graduate fellowship in 1992. He worked in the parallel file systems area for his dissertation. Purakyastha joined IBM upon graduating in 1996. At IBM, he has worked on building Java-based technologies such as Thin-Client Application Framework and NetScript. He belongs to several professional organizations, including Usenix and ACM. Reach him at apu@us.ibm.com .

Ajay Mohindra received his Ph.D. in Computer Science from Georgia Institute of Technology in Atlanta. While earning his degree, he participated in the design and implementation of a distributed object-based operating system. Mohindra is a member of IEEE (Institute of Electrical and Electronics Engineers), ACM, and Usenix. Reach him at ajaym@us.ibm.com .

References

[1] The Java Beans home page. http://java.sun.com/beans .

[2] Ajay Mohindra, Apratim Purakayastha, Deborra Zukowski, Murthy Devarakonda. ''Programming NetWork Components Using NetPebbles: An Early Report.'' In *Proceedings of the Fourth Annual Usenix Conference on Object Oriented Technologies and Systems (COOTS)* April, 1998.

[3] Timothy Howes and Mark Smith. ''A Scaleable Deployable Directory Service for the Internet.'' In *Proceedings of INET 95*, 1995.

[4] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1994.

[5] Larry Wall and Randall Schwartz. *Programming Perl*. O'Reilley and Associates, Inc. 1994.

[6] *The JavaScript Guide*. http://developer.netscape.com/docs/manuals/communicator/jsguide4/index. htm .

[7] *Safe-Tcl.* http://sunscript.sun.com/plugin/safetcl.html .

[8] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[9] Charles Knouse. *Practical DCE Programming*. Prentice Hall, 1995.

[10] Rolf Lendenmann, Jennifer Nelson, Janet Selby, Carlos Patino Lara. *An Introduction to Tivoli's TME 10*. Prentice Hall, 1998.

[11] Computer Associates Unicenter. http://www.cai.com/products/uctr.htm .

[12] Marimba, *How Software Goes Down to Business*. http://www.marimba.com .

[13] William Stallings. *SNMP, SNMP v2, and CMIP*. Addison-Wesley, 1993.

[14] William Stallings. *Network Management*. IEEE Computer Society Press, 1993.

[15] *Desktop Management Interface*. http://www.dmtf.org/tech/specs.html .

[16] German Goldszmidt and Yechiam Yemini. ''Distributed Management by Delegation.'' In *Proceedings of the 15th International Conference on Distributed Computing Systems*, 1995.

[17] Ann Wollrath, Roger Riggs, Jim Waldo. ''A Distributed Object Model for the Java System.'' *Proceedings of the 2nd Conference on Object Oriented Technologies and Systems (COOTS)*, 1996.

[18] *CORBA2.0/IIOP Specification*. http://www.omg.org/corba/c2indx.htm .

[19] Danny Lange and Daniel T. Chang. *IBM Aglets Workbench, Programming Mobile Agents in Java*. http://aglets.trl.ibm.co.jp/whitepaper.htm .

[20] Robert S. Gray. "Agent Tcl: A Transportable Agent System." In *Proceedings of the Workshop on Intelligent Information Agents*, in the *Fourth International Conference on Information and Knowledge Management*, December 1995.

[21] *Telescript Technology: The Foundation for the Electronic Marketplace*. http://www.genmagic.com/Telescript/Whitepapers/wp1/whitepaper1.htm, 1996.

[22] *MQSeries*. http://www.software.ibm.com/qseries.

[23] Salvatore DeSimone and Christine Lombardi. "Sysctl: A Distributed System Control Package." In the *Proceedings of the 7th Usenix LISA Conference*, pages 131-143, November, 1993.

[24] Clinton Pierce. The Igor System Administration Tool." In the *Proceedings of the 10th Usenix LISA Conference*, pages 9-18, September, 1996.

Appendix A: "SystemSniffer" and Example Component

Appendix A.1: SystemSniffer.java

```
package COM.ibm.netpebbles.components.systemsniffer;

public class SystemSniffer{

  public native String getVersion(String appname);

  static {
    System.loadLibrary("COM.ibm.netpebbles.components.systemsniffer.vernative");
  }
}
```

Appendix A.2: vernative.c

```
#include "COM_ibm_netpebbles_components_systemsniffer_SystemSniffer.h"
#include <windows.h>
#include <string.h>
#include <mbstring.h>
#include <stdlib.h>
#include <stdio.h>

JNIEXPORT jstring JNICALL Java_COM_ibm_netpebbles_components_
     systemsniffer_SystemSniffer_getVersion(JNIEnv*
               env, jobject obj, jstring appstr){

  HKEY key;
  LONG retcode;
  char keyname[50], class[50];
  DWORD keynamesize = 50, classsize = 50;
  FILETIME lastwrittento;
  int i;
  char str[50];
  char ver[50];
  DWORD valuetype;
  DWORD versize = 50;
  const char *appname;

  appname = (*env)->GetStringUTFChars(env,appstr,0);

  if(!strcmp(appname,"Lotus Notes")) {
    if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
               "SOFTWARE\\Lotus\\Notes",0,KEY_READ,&key) == ERROR_SUCCESS){
      for(i=0, retcode=ERROR_SUCCESS; retcode==ERROR_SUCCESS; i++){
        if((retcode = RegEnumKeyEx(key,i,keyname,&keynamesize,NULL,
                      class,&classsize,&lastwrittento)) == ERROR_SUCCESS){
          str[0] = '\0';
          strcat(str,"SOFTWARE\\Lotus\\Notes\\");
          strcat(str,keyname);
          if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,str,0,KEY_READ,&key) ==
                     ERROR_SUCCESS){
            if(RegQueryValueEx(key,"Version",NULL,&valuetype,ver,&versize)
                    == ERROR_SUCCESS){
```

```
                printf("Found version %s\n",ver);
                RegCloseKey(key);
                return((*env)->NewStringUTF(env,ver));
              }
            }
          }
        }
        return((*env)->NewStringUTF(env,"Installed but version unavailable"));
      }
      else {
        return((*env)->NewStringUTF(env,"Not Installed"));
      }
    }

    if(!strcmp(appname,"Netscape Navigator")) {
      if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
                 "SOFTWARE\\Netscape\\Netscape Navigator",0,KEY_READ,&key) ==
                        ERROR_SUCCESS) {
        if(RegQueryValueEx(key,"CurrentVersion",NULL,&valuetype,ver,&versize)
            == ERROR_SUCCESS){
          printf("Found version %s\n",ver);
          RegCloseKey(key);
          return((*env)->NewStringUTF(env,ver));
        }
        return((*env)->NewStringUTF(env,
                "Installed but version < 4, registry not properly configured"));
      }
      else {
        return((*env)->NewStringUTF(env,"Not Installed"));
      }
    }

    if(!strcmp(appname,"Hummingbird Exceed")){
      if( RegOpenKeyEx( HKEY_LOCAL_MACHINE,
                      "SOFTWARE\\Hummingbird\\eXceed\\CurrentVersion",
                      0,KEY_READ,&key) == ERROR_SUCCESS) {
        if(RegQueryValueEx(key,"Version",NULL,&valuetype,ver,&versize)
               == ERROR_SUCCESS){
          printf("Found version %s\n",ver);
          RegCloseKey(key);
          return((*env)->NewStringUTF(env,ver));
        }
        return((*env)->NewStringUTF(env,"Installed but version not found"));
      }
      else {
        return((*env)->NewStringUTF(env,"Not Installed"));
      }
    }
    else {
        return((*env)->NewStringUTF(env,"Not Supported by Component"));
    }
}
```

Appendix B: Update Scripts

Appendix B.1: Script to update an application

```
intf = "IDomainAdmin"
a = createComponent(intf)
locations = a.getMembers("Department 931B")
intf = "IUpdate"
n = length(locations)
```

```
dim status[n]
for (i=0; i < n; i=i + 1)
    b = createComponent(intf) at locations[i]
    status[i] = b.doUpdate("IBM Antivirus")
endfor

intf = "IDisplay"
home = "mymachine.watson.ibm.com"
c = createComponent(intf) at home
c.showList(status)
exit
```

Appendix B.2: Script to update an application with customizability

```
intf = "IDomainAdmin"
a = createComponent(intf)
locations = a.getMembers("Department 931B")
intf = "IUpdate"
n = length(locations)
dim status[n]
for (i=0; i < n; i=i + 1)
    intf2 = "IUpdatePolicy"
    d = createComponent(intf2)
    allow = d.isUpdateAllowed(locations[i])
    if (allow == false)
        status[i] = "Update Refused"
        continue
    endif
    b = createComponent(intf) at locations[i]
    status[i] = b.doUpdate("IBM Antivirus")
endfor

intf = "IDisplay"
home = "mymachine.watson.ibm.com"
c = createComponent(intf) at home
c.showList(status)
exit
```

Appendix C: Script to monitor a printer

```
printername = "colorful"
intf = "IPrinterManager"
pm = createComponent(intf)
location = pm.getLocation(printername)
intf = "IPrinterAssist"
pa = createComponent(intf) at location
while(true)
    iserror = pa.isError()
    if (iserror)
        err = pa.getError()
        shouldcorrect = pa.shouldCorrect(err)
        if(shouldcorrect)
            pa.correct(err)
            pa.setError(false)
        else
            notifyloc = pm.getNotifyLocation()
            intf2 = "INotifierGUI"
            gui = createComponent(intf2) at notifyloc
            gui.showMessage(err,printername,location)
        endif
    endif
endwhile
```