Debugging Network Management Scripting Applications [1]

Dong Zhu, Adarshpal S. Sethi

Telephone: (302)831-8704, (302)831-1945

email: (dzhu, sethi)@cis.udel.edu

Department of Computer and Information Sciences
University of Delaware, Newark, DE 19716

Abstract:

Distributed network management scripting applications are in great demand today. However, the issue of debugging these scripting applications has been largely ignored by the network research community. The purpose of this paper is to define the problem of debugging management scripting applications and evaluate the approaches and techniques used for general distributed program debugging in the context of management scripting applications. A simple approach based on the simulation replay technique is proposed. We argue that a good debugger architecture should be integrated with the management scripting framework, and we propose an integrated debugger architecture. Peculiar situations in the network management applications such as time-dependent operations and ways to deal with them are also discussed.

Keywords: Distributed Program Debugging, Embedded System Debugging, Network Management Scripting Framework, Network Management Scripting Application Debugging, Simulation Replay

1 Introduction

Network management script delegation as the major means of Management by Delegation (MbD) [YGY91] has been widely accepted by the network management research community and the industry. Many scripting frameworks have been proposed. Standard activities include DISMAN Scripting MIB [LS97] for SNMP management, and Command Sequencer [Com95] for OSI management. Research prototypes include SHAMAN [KSSZ97] for SNMP, and AMO [VPK97] and GAS [YTY96] for TMN [TMN92] management. As the result of these scripting framework activities and, more importantly, the increasing number of network management applications and the demand for them, more sophisticated distributed network management scripting applications (henthforth management scripting applications) are emerging. This has raised the important issue of debugging these scripting applications. Until now, the issue has been largely ignored by the network research community.

[1] Prepared through collaborative participation in the Advanced Telecommunications/Information Distribution Research Program (ATIRP) Consortium sponsored by the U.S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0002.

Management scripting applications are distributed applications. People have been working on the problem of distributed program debugging for many years and have proposed many solutions using various approaches and techniques. Of these techniques, simulation replay [CW82], instant replay [LMC87], and event-based behavior modeling [Bat95] are the most prominent ones. It is natural that the first step towards solving the problem of debugging management scripting applications is to see if we can adopt some of these solutions. In this process, it is important to find out the peculiarities of the management script applications, and evaluate the solutions accordingly.

Another important issue is finding a good debugger architecture. Since the scripting framework has many functions needed for debugging, we think it is beneficial to have a debugging architecture integrated with the scripting framework in order to limit function duplications.

The purpose of this paper is to define the problem of debugging management scripting applications and evaluate the approaches and techniques used for general distributed program debugging in the context of management scripting applications. We propose a simple approach for debugging management scripting applications. We also propose a debugger architecture which is integrated with the scripting framework.

The organization of the rest of the paper is as follows: Section 2 introduces the general scripting framework functional architecture and management scripting applications. Section 3 introduces the general distributed program debugging issues and solutions. Section 4 analyzes the characteristics of management scripting applications and evaluates the feasibility of applying general distributed program debugging approaches and techniques. Section 5 proposes an integrated debugger architecture and details an application of Simulation Replay technique in debugging management scripting applications. Section 6 makes conclusions and proposes our future work.

2 Introduction to Network Management Scripting Framework and Scripting Applications

Delegating scripts is the major means used to transfer management functions dynamically from managing systems to managed systems in order to take advantage of the increased computational power in the network elements and decrease pressure on network management centers (NMCs) and network bandwidth. Generally, Script Delegation works in the following way: a *delegation manager* downloads a set of *management scripts* which describes its desired management actions to a *delegation agent* at a remote location, and asks the scripts to be executed there. An *interpreter* in the agent then executes the scripts, and the result is conveyed back to the manager. This management scheme is illustrated in Figure 1. A *scripting framework* is used to provide mechanisms and an environment to make this management scheme possible. We describe briefly the essential components of a scripting framework in the following paragraphs. The interested reader may refer to [ZSK98] for a more detailed discussion. These components are: *script management*, *scripting language* and *interpreter*, and *script execution environment*. Scripting management deals with the following aspects of the framework: delegation roles, script administration, script transfer, and script execution monitoring and control.

The players in the scripting framework are *delegation managers (DM)* and *delegation*

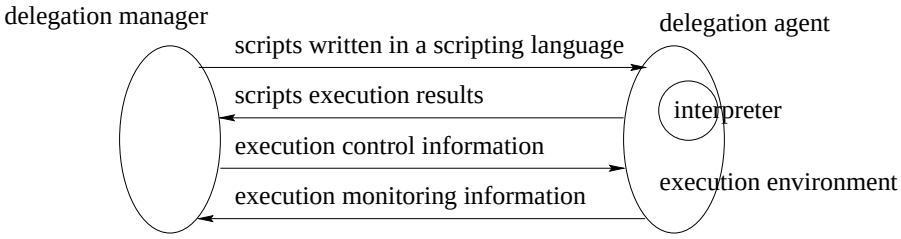

Figure 1: Script Delegation

agents (DA). The DM is the delegator of the scripts; the DA is responsible for the execution of the scripts on behalf of the DM. DMs and DAs have many-to-many relationships: a DM may delegate scripts to multiple DAs; a DA may executes scripts delegated from multiple DMs. The DM and DA may or may not be the management framework manager and agent respectively, although, in an integrated environment, it is very likely that DM and DA are at the same time the management framework manager and agent.

A *management script* is a set of instructions written in a scripting language, and specifies a management task; the interpretation of the script will carry out that task. However, sometimes in order to specify one management task, a set of scripts may be needed; in this case, the interpretation of the set of scripts will carry out the management task.

A well-designed scripting framework should provide good *script administration* mechanisms to avoid introducing new management problems. In order to achieve this, scripts may be assigned *names* for identification purposes; different scripts with the same name may be assigned different *versions*. Moreover, *access control* mechanisms may be provided for script access and execution.

Script transfer is the physical transportation of management scripts from a DM to a DA via some transfer mechanisms. Repeatedly executed scripts should be stored at the DAs when they finish executions. Script execution results may need to be stored locally and later processed by other scripts; or they may need to be transferred back to and examined by the DM. Results should be structured appropriately to facilitate the processing by the DMs or other scripts.

Delegated management tasks need management themselves. *Script execution monitoring and control* deals with the issues related to the monitoring and control of the progress of the delegated management tasks. A script could be in either of these states: ready, running, suspended, aborted, terminated, etc. The execution control include the following actions: initiation, suspending, resuming, aborting, signaling (or sending events to an execution), enabling, disabling, etc. One important aspect is to monitor and control prescheduled and/or periodic execution.

The *execution environment* provides services for script executions: *translation service* translates the scripts to a required form before they are executed; *management information access service* provide MIB access and communications support; *execution service* provides multi-threaded execution, synchronization, and other services related to the execution of scripts; *error handling service* provides both static and runtime error handling. Major issues related to the above are:

The *Scripting Language (SL)* is defined as the language in which a management script is expressed. The use of the word "scripting" does not necessarily mean that the

scripts are interpreted; actually, they could be compiled and directly executed on the target machine. We use the word "scripting" to refer to the language's characteristics of being highly capable and expressive, extensible, portable, interactive, and easy to debug.

A SL may have all the major data and control structures as a general purpose programming language. Besides the regular data model the SL manipulates, the SL may have other data structures which are specifically provided to deal with the target management information model that the scripts manipulate. For instance, for DISMAN and SHAMAN, the target management information model is SNMP; for AMO, it is OSI. The SL may also have control structures that are designed specifically for management applications. The asynchronous event processing control structures which allow the scripts to be written to process asynchronous events, is the most noticeable example.

In order to allow scripts to access MIB and management protocols, SL usually provides necessary language constructs to achieve this. For instance, many Tcl-based SLs for SNMP extend Tcl by adding commands to do SNMP Get, GetNext, GetBulk, Set, etc. We can distinguish two kinds of MIB accesses: *local access* and *remote access*. Local access deals with the local MIB, while remote access deals with a remote MIB and also needs management protocol support. For remote access, the scripts act as a management framework manager. Here, the goal to achieve is to provide in the SL the full remote access power a manager may have. One desirable goal is the transparent access to both local and remote MIBs. In order to execute a script, a runtime environment (execution environment above) is provided for the scripts to access MIBs, communication protocol stack, and other management facilities.

Having described the major functions of the general scripting framework, we are now ready to define what is a management scripting application. We have mentioned above that a set of scripts may be needed to accomplish a management task. Actually, this set of scripts may be distributed over a network, and they need to cooperate among themselves to accomplish a management task of a more distributed nature. While being executed, the scripts usually access management agents and, in some cases, may use other non-standard network management protocols such as ICMP to get certain kind of network information. Scripts may be accessed by managers which are not scripts and are not part of the scripting framework. Thus a management scripting application is a distributed network management application which is composed of distributed processes and functions including scripts and their scripting framework, management managers and agents and their management framework, and other non-standard network management processes and protocols. All these processes and functions cooperate to accomplish the distributed network management task the application is written for.

3 Introduction to Distributed Program Debugging

In order to adopt the general distributed program debugging solutions to debug management scripting applications, in this section we give a brief introduction to the problem of general distributed program debugging and its solutions.

Most distributed program debugging techniques are based on or can be found in sequential program debugging. The solution for sequential program debugging is well established. The classic approach to sequential program debugging is called *cyclical debugging*.

To debug using this approach, a sequential program may be executed multiple times. Because of the deterministic nature of sequential programs, the same behavior of the program is guaranteed to repeat with each execution. During the executions, the user is free to stop the program execution to examine the program state by using breakpoints or single stepping the program, or to do other things to help debugging.

Comparing to debugging sequential programs, several difficulties arise in debugging distributed programs. This is because of the following characteristics of distributed programs:

- Multiple asynchronous processes.
- Multiple processors
- Variant communication delays.
- The global state may be non-existent and the concept misleading.

Because of these characteristics, *race conditions* may exist in executing distributed programs and cause *non-deterministic behavior*, and repeated executions of a distributed program may not yield the same results. Therefore, it is more difficult to apply the cyclical debugging technique in debugging distributed programs.

A few approaches have been proposed to solve this problem. [MH89] gives a good introduction. We will only discuss the solutions briefly here. The simplest approach is to take snapshots of an execution of the distributed program and analyze the trace after the execution ends. The advantage is: only one execution is needed. Actually, an even simpler approach is to just display the execution at run time without actually recording the snapshots.

Since successive executions may not reveal the same erroneous behavior, or any erroneous behavior at all, the disadvantage of this approach is that all information needed to debug the program must be collected during one execution. The amount of information tends to be very large and is difficult for users to sort through.

The most important approach is to make it possible to apply the cyclical debugging technique to distributed programs. The key idea of this approach is to deterministically replay an execution. In order to do this, certain amount of information must be collected during the initial execution which is called the *monitoring phase* or the *recording phase*. This technique is therefore called *record-replay* technique.

Depending on how the replay is to be carried out, there are two different techniques. One technique, which we call *simulation replay* [CW82], involves replaying only a subset of all processes and simulating the others for the purpose of replaying this interesting subset. In order to do this, during the monitoring phase, all interactions between the processes must be recorded together with the contents of the interactions. During the replay phase, only the interesting processes are replayed. The interactions between the processes are simulated in the sense that they are not actually carried out. The advantage of this technique is its ability to only replay the interesting processes. The disadvantage is that the amount of information that needs to be recorded is large, since we must also record the contents of each interaction between the processes.

The other technique is called *instant replay* [LMC87]. The difference between this one and the simulation replay technique is that only the interactions between the processes are recorded; the contents of the interactions are actually reproduced during the replay phase. Thus, instant replay achieves the goal of only recording enough information in order to deterministically replay a distributed program. Therefore, the advantage is that data recorded in the monitoring phase is significantly reduced. The disadvantage is that all processes must be re-executed in order to to get the contents of the interactions.

The third approach is *event-based behavior modeling* [Bat95]. In this approach, the execution of a distributed program is seen as a sequence of events. Models are defined which specify the expected behavior of the program or the erroneous behavior of the distributed program using a modeling language. The models are then used to check against the event flow, or in other words, the actual behavior of the program. The results can be used to either help find bugs or they can be used to control the debugging activities (such as breakpointing the execution). This approach can also be used in debugging sequential programs. Since event is a widely adopted concept, the advantage of this approach is that it makes the debugging tools highly adaptable. The disadvantage is that since it is a rather formal approach and usually requires the user to learn a modeling language, it tends to be more difficult to learn and use in practice. It also requires the user to have a pretty good understanding of the system behavior in order to be able to write behavior specifications.

Other Approaches which we will not introduce here include static analysis of the distributed programs and visualization of the distributed programs in time-process diagrams and animation.

An important issue in distributed program debugging is the debugger architecture design and the controlling of the debugging activities. Figure 2 shows a distributed debugger architecture. The debugger console is the host the debugger user is at and is the place where debugging commands such as single stepping a remote process, stopping or resuming all remote processes are issued. The central debug function provides an interface to monitor, control, and coordinate the remote debug functions. The GUI may provide multiple windows for viewing multiple remote processes. The communication interface implements a protocol for the exchange of debugging messages among the console and the remote debug functions. The remote debug functions are agents of the controlling debug function. They are used to collect information such as those collected in the monitoring phase of the record-replay debugging, and control the actual execution of the remote processes such as carrying out breakpoint or single step requests of the central debug function. On a system such as a multitasking system or timesharing system, the remote debug function may not have direct control of the system hardware such as CPU to do things like setting breakpoints, the actions must goes through the operating system. The operating system needs to provide APIs specifically designed for debugging.

One important aspect of debugging distributed programs is how the actual debugging activities are coordinated among the distributed debugging functions. Questions such as "when a breakpoint is encountered in a process, should the other processes be stopped?" are difficult to answer but must be dealt with in order to produce a useful distributed debugger.

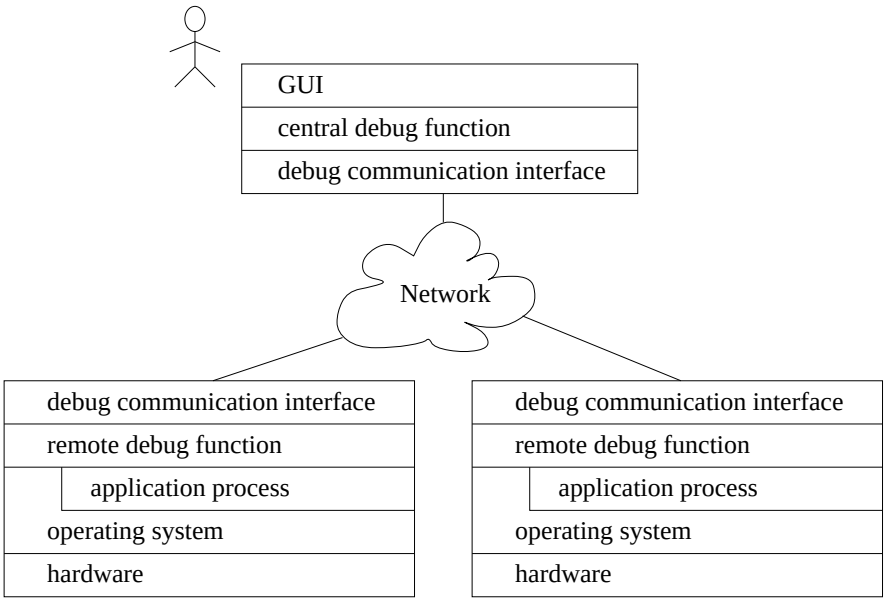

Figure 2: Distributed Debugger Architecture

4 Debugging Distributed Network Management Scripting Applications

Among the components of a management script application, the only component we want to debug is the delegated scripts. The other components are interesting but are not considered debugging targets. Thus, we define our goal of debugging network management scripting applications as *debugging the logical and performance bugs in the distributed scripts*. More specifically, we do not try to debug the following:

- bugs in the scripting framework
- faults happened in the network
- problems in the delegation managers
- problems in the management agents

From the above definition, we can characterize our debugging as *distributed embedded system debugging*. Our embedded system is comprised of the distributed scripts; the system's environment is the scripting framework, the delegation managers, the management agents, and the networking environment. It shares several characteristics of the general embedded systems:

- Timer-dependent operations. As an example, a script may be executed every five minutes to find out how many clients are connected to our HTTP server.
- The environment is constantly changing according to some asynchronous processes which is not determined by the embedded system. For example, a MIB located on

an agent is constantly changing. Or, a manager may be executing some other related applications.

- Asynchronous environmental events. For example, managers may send requests, and agents may send notifications to the scripts.

In our embedded system, the sensors are manager requests, polls of the the management agent, and agent notifications. The actuators are management operations the scripts issued to the management agents. Please note that the term embedded system usually has some hardware flavor, but our embedded system is a pure software one.

In order to apply the the general distributed program debugging solutions we introduced in section 3 to scripting applications, we briefly evaluate these solutions in the new environment. Since the approach of taking snapshots of an execution of the distributed scripts and analyzing the trace after the scripts exit does not provide too much debugging help to the user, we will not discuss it further.

Of the two deterministic replay techniques, we consider Simulation Replay to be more suitable to our system. This is because of the embedded nature of our scripts. As we have pointed out, some of the processes in the whole application belongs to the environment of our embedded system. They expose asynchronous behavior which is beyond control of our debugger. For example, due to the nature of network management agents, it is not practical to try to control their operations for the purpose of debugging the scripts. Also, it is not an easy task to try to instrument the managers for the same purpose. We can not use instant replay which does not record the contents of the interactions, and which require total participation and control of all processes, including the manager and agent processes. We can only use simulation for the managers and the agents.

On the other hand, total record of all interaction messages requires great system resources. The instant replay technique may be useful to alleviate the problem but can be only used for interactions between scripts or any other processes over which the debugger has good control.

Since events are so fundamental to network management, the event-based behavior modeling approach is potentially a good candidate for debugging both distributed scripts and even the whole application including the managers and the agents. It also serves as a good method for network fault management. Further, it may serve as an excellent technique in interoperating the debugging of heterogeneous scripting framework applications. For our purpose, a good way to use this approach is to make it a complement of the Deterministic Replay approach. It can be used as a means to specify predicates for breakpoints to control script executions. Thus when a certain pattern of events appear, the debugger can take certain actions such as stopping the script execution, and allowing the user to examine the states of the scripts and the scripting framework.

5 A Simple Integrated Simulation Replay Debugger for Distributed Management Scripts

Having evaluated the solutions in the scripting application context, in this section, we go into more detail of a simple debugger using Simulation Replay technique. First, we discuss

the design of an integrated debugger in the scripting framework environment. Then we describe a debugging session using this debugger.

For the purpose of debugging, the scripting framework must meet the following basic requirements:

- Control function for the debugger to set breakpoints to the scripts, single step the scripts, etc. It is worthwhile to note that most of the scripting languages used for network management are interpreted language, and interpreted languages make it easier to control the script execution, and thus make it easier to write debuggers,
- Interfaces of the execution engine to allow the user to examine the states of the script execution.
- Interfaces to expose the scripting framework information to the user. These could include script scheduling queues, system event input and output queues, individual script event input and output queues, etc.

In section 3, we have introduced the architecture for general distributed program debugging. In our environment, it makes perfect sense to integrate the debugging functions with the scripting framework. We list the reasons as follows:

- The execution engine of the scripts is one of the scripting framework functions. The debugger needs to have a good control of it in order to do single step execution, breakpoint execution, etc.
- Much of the information needed for debugging the scripts is managed by the scripting framework. For example, the scripting framework may have queues of requests from the managers, queues of responses to the managers, queues of notifications from the agents, and storage of execution results of the scripts. In a sense, the scripting framework serves as a virtual machine for the scripts.
- The architecture of the scripting framework can be extended. It makes a good foundation for the distributed debugger architecture. We can use the scripting framework functions to our advantage in debugging: MIBs can be designed and the scripting framework protocols can be used for exchanging of the debugging messages among distributed debug functions. Event log functions can be used for recording the interactions of the processes in the monitoring phase of record-replay. We can also use and extend the scripting framework execution control functions to provide control of the script executions.

As we have explained in section 4, one suitable debugging technique is Simulation Replay. Using this technique, we record the contents of the messages exchanged between the scripts, managers, and agents. We then use this information to simulate the environment of the erroneous execution of the scripts. Only the scripts are actually replayed. During the replay of the scripts, the managers and the agents are simulated by the distributed debugger (scripting framework). The scripts can still be controlled by setting breakpoints, single stepping, and others.

Figure 3 shows an integrated debugger architecture. The general scripting framework functions needs to be extended in this architecture:

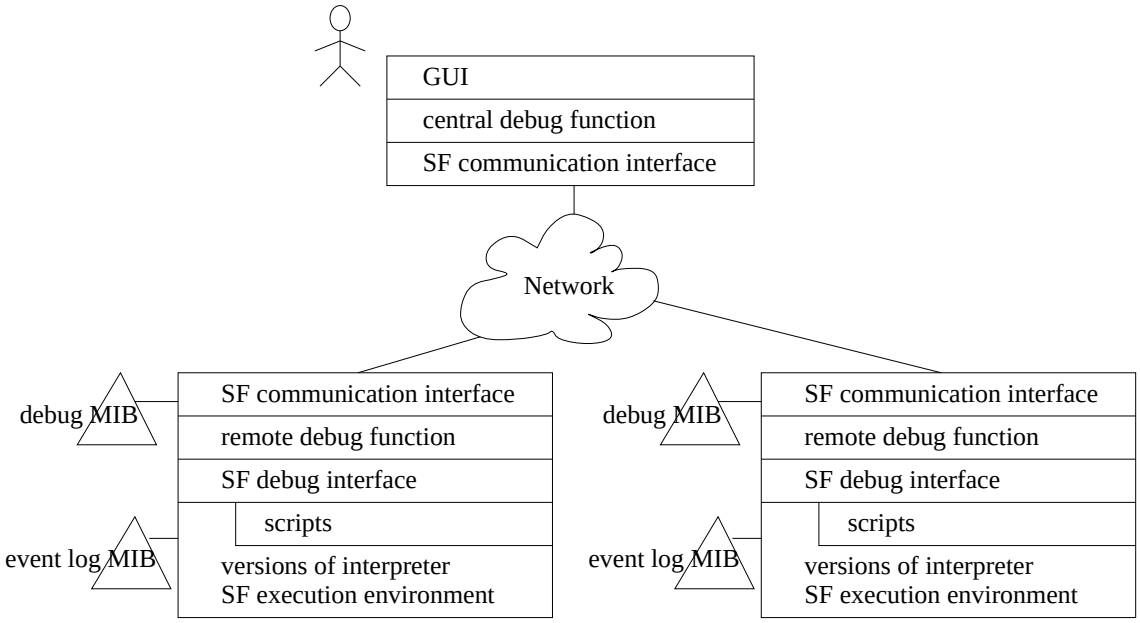

Figure 3: An Integrated Debugging Architecture

- Scripting framework debug interface. As information is needed for the user to examine the scripting framework environment, the framework has to provide interfaces to expose its internal state that the user need to debug the scripts.
- Debug MIB. This MIB is used for the debugger console to issue debugging commands such as starting or stopping monitoring phase or replay phase execution of the scripts, single stepping or setting breakpoints for replay phase debugging. The scripting framework information also needs to be exposed through this MIB. Examples are script scheduling queues, system event input and output queues, individual script event input and output queues, etc.
- Event log MIB. This is used as the storage for recording information in the monitoring phase. This may also be used by the central debug function to do analysis using techniques such as event-based behavior modeling.
- Debugging versions of the interpreter. Versions for monitoring phase and replay phase are needed. The monitoring version is much simpler than the replay version, and only needs simple extension of the non-debug interpreter. The replay version needs all features of an interactive interpreter for interactive interpretation.

We now describe in more detail one debugging session using this debugger architecture. Here for simplicity, we make the assumption that within a node, there is only a single thread of control of the script execution. Actually, our approach can be easily extended to multiple threads situations if we treat each thread as a process and record the interactions between the threads similarly.

To start the monitoring phase execution, the remote debug function is notified that the subsequent execution needs to be recorded. This is done by a request on the debug MIBs. The remote debug functions will check the MIBs and invoke the monitoring version of

the interpreter and accomplish other debugging functions accordingly. Once the monitoring phase started, the application is allowed to run without interruption from the debugger user either until erroneous behavior of the application is observed or when the debugger user considers appropriate. During this period, the scripting framework execution environments record the messages (including the contents of the messages) exchanged between the scripts, the managers and the agents. The user sends commands to the remote debug functions to stop the monitoring execution also by means of debug MIB requests and the remote debug functions will act accordingly.

To start the replay phase execution, the user again sends MIB requests to the remote debug functions asking the scripts to be replayed. This time, the user does not start the manager or the agent processes. Even if they are started, they will not have influence on the replayed scripts, except that the execution environments may be a little busier discarding the messages they receive during this time. Instead of using these messages, the recorded messages are replayed, and the manager and agent processes are effectively simulated. The behavior of the application as displayed in the monitoring phase is guaranteed to be deterministically replayed by an simulation-replay algorithm.

The replay phase uses the replay version of the interpreter. During the replay, the debugger user can make use of traditional debugging techniques to examine the execution of the scripts. For example, the user can set breakpoints, do single stepping, examine values of variables, runtime stacks, etc. The user can also examine the MIBs, the scipting framework script scheduling queues, the event queues, etc. These are available through the interface functions that the scripting framework provides for debugging purposes.

We now discuss the issue of timer-dependent operations which may cause multiple threaded executions. One case is that some scripts may be scheduled to be executed periodically. We can deal with this kind of situation simply by recording in the monitoring phase the points where timer interrupts occur. During the replay, the timer interrupts are also replayed together with the script interactions. Thus, we may have the following scenario where a user is single stepping through a script in the replay phase, after a step, he is prompted by the debugger that a timer interrupt occurs and asked whether he wants to step into the triggered script execution. The user can either do that or he can simply ignore the interrupt and continue to step through his interested script. The system will execute the timer triggered scripts before allowing the user to continue with his debugging. The user may shut up all timer interrupt prompts if he does not suspect there is a problem associated with them. The idea is that we record all such events during the monitoring phase, and during the replay phase there is no real timer running, and thus we will not have problem of real time debugging where timing restraint must be observed.

A similar case is when a script sleeps for a period of time. We can use the same technique to deal with this situation and record the timer awake interrupt event and replay it. Still another case is communication timeout. For example, SNMP requests and responses may timeout. This may cause problems in the scripts. We can also use the same technique here which is to record the communication abnormal situations and replay them faithfully later. We can expect that the application of the technique will greatly assit debugging.

6 Conclusion and Future Work

We have shown in this paper that the management scripts are distributed embedded applications. Of the established solutions for the general distributed system debugging, simulation replay is the most suitable approach for debugging management scripts, and we have shown how it is also used to deal with the peculiar situations in the network management applications such as time-dependent operations. We also conclude that a good debugger architecture should be integrated with the scripting framework.

For future work, a detailed design is in progress and an implementation is planned. We also plan to investigate how event-based behavior modeling can be applied, and debugging performance bugs in management scripting applications. Debugging mobile script is another important and urgent issue the whole network management community may want to investigate.

References

[Bat95] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *IEEE Transactions on Computer Systems*, 13(1):1–31, February 1995.

[Com95] International Organization for Standardization. *ISO/IEC DIS 10164-21, Command Sequencer*, 1995.

[CW82] R. Curtis and L. Wittie. BugNet: A Debugging System for Parallel Programming Environments. In *Proceedings of the Third International Conference on Distributed Computing Systems*, pages 394–399, 1982.

[KSSZ97] P. Kalyanasundaram, A. Sethi, C. Sherwin, and D. Zhu. A Spreadsheet-Based Scripting Environment for SNMP. In Lazar et al. [LSS97], pages 752–765.

[LMC87] T. LeBlanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[LS97] David Levi and Juergen Schoenwaelder. *Definitions of Managed Objects for the Delegation of Management Scripts*, March 1997. Work in progress (Internet Draft: draft-ietf-disman-script-mib-01.txt).

[LSS97] A. Lazar, R. Saracco, and R. Stadler, editors. *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management.* IEEE/IFIP, Chapman & Hall, 1997.

[MH89] C. McDowell and D. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[TMN92] CCITT. *CCITT Recommendation M.3010 - Principles for a Telecommunications Management Network*, October 1992.

[VPK97] N. Vassila, G. Pavlou, and G. Knight. Active Objects in TMN. In Lazar et al. [LSS97], pages 139–150.

[YGY91] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In I. Krishnan and W. Zimmer, editors, *Integrated Network Management II*, pages 95–107. North Holland, Amsterdam, 1991.

[YTY96] I. Yoda, H. Tohjo, and T. Yamamura. Interpreter Language-Based TMN Agent Systems. L'Aquila, Italy, October 1996.

[ZSK98] D. Zhu, A. Sethi, and P. Kalyanasundaram. Towards Integrated Network Management Scripting Frameworks. In *Proceedings of the 9th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, Newark, Delaware, USA, October 1998.