Advantages and Disadvantages of the Script MIB Infrastructure

Frank Strauß

Computer Science Department
Technical University Braunschweig
Bültenweg 74/75
38106 Braunschweig, Germany
strauss@ibr.cs.tu-bs.de

October 2000

Abstract

The drawbacks of classical centralized management systems and the proposal of distributed management to solve them have been discussed in various papers in the past. The IETF vehicle for distributed Internet management is the Script MIB. Jasmin is an implementation of the Script MIB which is freely available for evaluation.

This article presents a real life scenario of some management tasks that can be solved by distributed management approaches and have been implemented using the Jasmin infrastructure. We do not only discuss the observation of advantages predicted by previous papers, e.g., the reduced CPU and network load at the management station's bottleneck, but also the impracticable circumstances of "managing the management", the development costs of the Script MIB approach, and the pitfalls of running management scripts in remote environments.

1 Introduction

Distributed management [1, 2] in general and the Script MIB [3] specifically are expected to bring various advantages over the centralized concept suited for the raising demands 1n network management. A commonly mentioned advantage is the increased scalability due to the delegation of management tasks from the centralized network management station to mid-level managers. This implies that CPU and network load is also delegated to the subnets to which the mid-level managers belong.

Another major advantage is concerned with the robustness of management tasks. While centralized management systems require a reliable network, the distributed approach allows to delegate some sensible manager functions next to the observed agents. Hence these functions may become independent from less reliable WAN links, for example.
Another advantage is the possibility to implement management tasks that could not be achieved otherwise. E.g., scripts can be delegated to remote hosts and test and measure the reachability, authentication and performance of locally supplied network services.
While these obvious advantages of distributed management and the Script MIB deployment have been argued in various papers before [4, 5], there are only a few articles that reveal practical experience with the Script MIB deployment [6]. This paper exemplifies the feasibility of some simple monitoring tasks and tries to weigh up critically the advantages and disadvantages of using the Script MIB infrastructure to distribute these tasks.
The paper is structured as follows: Section 2 presents a scenario of a network in which some typical monitoring and testing tasks have to be done. Section 2.1 describes how these tasks have been realized by implementing a general Java SNMP monitoring package and some monitoring classes on top of this package. The mapping of this architecture to a centralized and a distributed approach is described in Section 2.2 and some measurements and observations are presented in Section 2.3. In Section 3 we describe the advantages that have been discovered during the deployment and use of our solution. The drawbacks and pitfalls we had to handle are described in Section 4. Finally, Section 5 presents some conclusions learnt from these practical investigations.

2 A Management Scenario

To facilitate the analysis of a real life network management system, we picked some hosts in various subnets that we are using every day. Two of these subnets, a workstation pool of 11 Sun Ultra-1 hosts and a server farm of 7 Ultra-30 and Ultra-60 hosts are tightly located in our building, each on a separate Fast Ethernet segment and both segments connected by a switch. Additionally, a cluster of 17 Pentium-II/Linux hosts, that are used for parallel computing tasks, is located at a distant place on our campus network. Two Jasmin [7, 8] Script MIB agents have been setup in the workstation pool and the server farm. While the subnets are locally administrated, we have no administrative control over the computing cluster. However, we could manage to get a Jasmin agent running in that cluster as well and got appropriate agent access permissions. Another host in the server farm is the management station that is responsible to catch up all severe monitoring events. The network is shown in Figure 1.
The monitoring tasks we wanted to achieve are as follows:

- All hosts shall be monitored with respect to recent reboots, high numbers of system processes and user sessions, and nearly full filesystems. This information can be read from several objects from the HOST-RESOURCES-MIB [9].

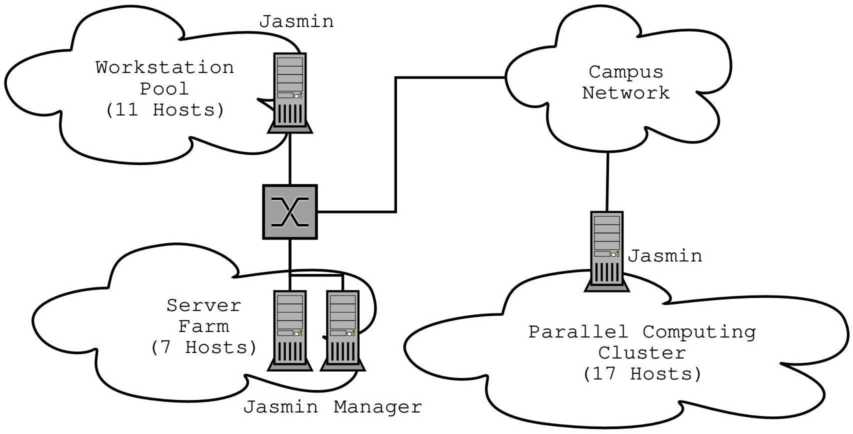

Figure 1: The network.

- All network interfaces shall be monitored with respect to status changes, errors, and utilization. This information is available from the IF-MIB [10].
- The mail server shall be monitored to recognize high input rates and long outstanding queues. This information is available from the MTA-MIB [11].
- Some TCP services (SMTP, HTTP, FTP, POP3, NNTP) offered by the server farm shall be tested from remote clients to monitor their availability. This is done by establishing TCP connections to the servers, checking their greeting messages or responses to simple commands, and closing the connection again.
- Some services (RPC portmapper, NFS, SMTP) of workstations and servers shall be monitored by looking at the table of TCP sockets. They must be present, but the numbers of connections should not exceed a certain limit. The TCP sockets and their states can be read from the TCP-MIB [12].

2.1 Monitoring Architecture

To achieve these tasks in an efficient way, we decided to develop a set of small Java programs, named *Monitors*, that are capable to execute a set of related *Checks*. There are five monitors that match exactly the five points of the list above, e.g. the `InterfaceMonitor` contains an `InterfaceStatusCheck`, an `InterfaceDiscardsCheck`, and an `InterfaceUtilizationCheck`. Each check is implemented as a class that implements a Java interface named `Check`. The peers being monitored by each check are represented by instances of the class `Peer`. To make all monitors use the same set of command line options, e.g. to specify the monitoring interval, optional thresholds and the list of peers, we encapsulated the option parser in the class `Options`. The `SnmpMonitor` class allows to register a number of checks with arbitrary intervals and for arbitrary peers. When its never-ending `monitor()` method is called, it schedules every registered check according to their intervals. These classes are bundled together with two utility classes for sending and printing results (`Alarm`) and for printing debug messages (`Debug`) as a package named `snmpmonitor`, so that it gets quite comfortable to write monitors. Figure 2 shows the components of a Script MIB agent running some monitors. A simplified UML class diagram of the monitoring package is shown in Figure 3. Note that in case of execution of monitors in a Script MIB runtime environment, it is reasonable to supply the monitoring package as a Script MIB language extension.

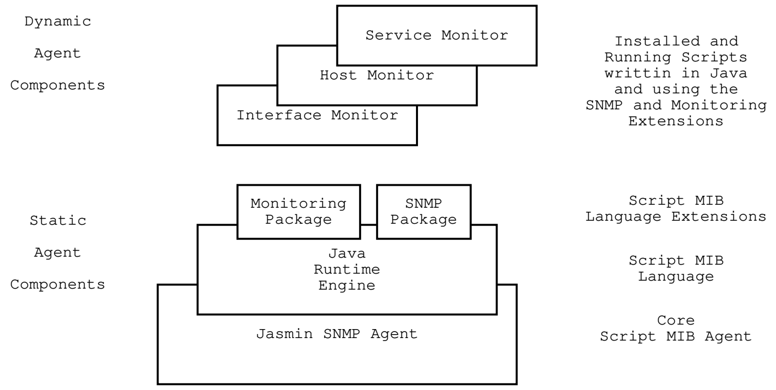

Figure 2: Components of the Script MIB agent.

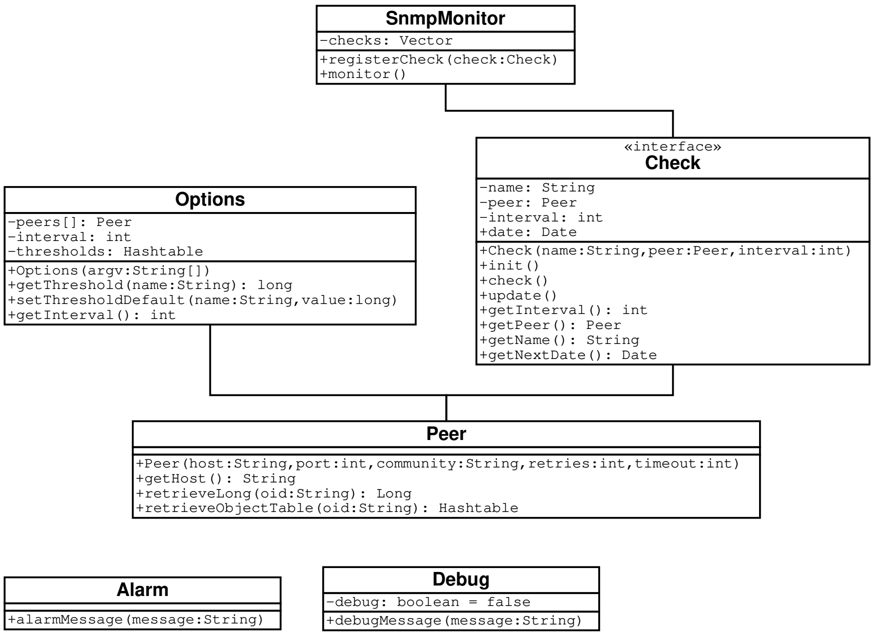

Figure 3: UML class diagram of the Java monitoring package.

As an example, Figures 4 and 5 show the main class of the interface monitor and one of its check classes that monitors the status of all interfaces of a peer. The check raises

alarms when an interface has been added or a status has changed. All other monitors and checks follow the same structure.
Note that these classes do not contain any code that is specific to an SNMP toolkit or to the deployment in the Jasmin environment. There are two interfaces of monitors relevant to the runtime environment: Arguments to a monitor's main method are passed through a string array. This is equivalent for a single program started from the command line using the JDK `java` runtime engine and for the Jasmin Java runtime engine that passes Script MIB launch arguments to a launched script. The other interface affects the output of a monitor. This is achieved through the monitoring package's `Alarm` class. This class is capable of printing messages to the standard terminal output as well as sending them to the Script MIB agent which in turn makes the result available through a Script MIB object and emits SNMP notifications.

```java
public class InterfaceMonitor
{

    public static void main(String argv[])
        throws UnknownHostException,
               SocketException,
               ParseException
    {
        SnmpMonitor monitor;
        Options options;
        Check check;
        Peer peers[];

        // create a new monitor.
        monitor = new SnmpMonitor();
        options = new Options(argv);

        peers = options.getPeers();
        for (int i = 0; i < peers.length; i++) {

            // create and register the interface status check.
            check = new InterfaceStatusCheck(peers[i],
                                             options.getInterval());
            monitor.registerCheck(check);

            // create and register the interface utilization check.
            check = new InterfaceUtilizationCheck(peers[i],
                                                  options.getInterval());
            monitor.registerCheck(check);

            // create and register the interface discards check.
            check = new InterfaceDiscardsCheck(peers[i],
                                               options.getInterval());
            monitor.registerCheck(check);

            // more checks to follow...
        }

        monitor.monitor();
    }

}
```

Figure 4: InterfaceMonitor.java

```java
public class InterfaceStatusCheck extends Check
{
```

```java
    private Hashtable oldLastChange;
    private Hashtable oldOperStatus;

    public InterfaceStatusCheck(Peer peer, int interval)
    {
        super(peer, interval);
        setName("InterfaceStatusCheck");
    }



    public void init()
    {
        oldLastChange = getPeer().retrieveObjectTable("1.3.6.1.2.1.2.2.1.9");
        oldOperStatus = getPeer().retrieveObjectTable("1.3.6.1.2.1.2.2.1.8");

        // set the time of last run to the current time.
        update();
    }



    public void check()
    {
        Hashtable newLastChange;
        Hashtable newOperStatus;
        Enumeration keys;
        String key;
        Long oldchange, newchange, oldstatus, newstatus;

        newLastChange = getPeer().retrieveObjectTable("1.3.6.1.2.1.2.2.1.9");
        newOperStatus = getPeer().retrieveObjectTable("1.3.6.1.2.1.2.2.1.8");

        keys = newLastChange.keys();
        while (keys.hasMoreElements()) {
            key = (String)keys.nextElement();
            oldchange = (Long)oldLastChange.get(key);
            newchange = (Long)newLastChange.get(key);
            oldstatus = (Long)oldOperStatus.get(key);
            newstatus = (Long)newOperStatus.get(key);

            if (oldchange == null) {
                Alarm.alarmMessage(getPeer().getHost() +
                                   ": Interface " + key +
                                   " has been added.");
            } else {

                if ((newchange == null) ||
                    (oldstatus == null) ||
                    (newstatus == null)) {
                    Alarm.alarmMessage(getPeer().getHost() +
                                       ": Cannot read all required " +
                                       "IF-MIB::ifTable objects.");
                } else {

                    if (!newchange.equals(oldchange)) {
                        Alarm.alarmMessage(getPeer().getHost() +
                                           ": Status of interface " + key +
                                           " changed to " + newstatus);
                    }
                    // some agents don't implement ifLastChange:
                    if (!newstatus.equals(oldstatus)) {
                        Alarm.alarmMessage(getPeer().getHost() +
                                           ": Status of interface " + key +
                                           " changed from " +
                                           oldstatus + " to " + newstatus);
                    }
                }
            }

        }

        oldLastChange = newLastChange;
        oldOperStatus = newOperStatus;

        // set the time of last run to the current time.
        update();
    }

}
```

Figure 5: InterfaceStatusCheck.java

2.2 Centralized and Distributed Mappings

A reasonable distributed mapping of the monitors to the Jasmin hosts is shown in Table 1. This mapping ensures that most of the monitoring traffic keeps within the monitored subnet. Only notifications have to be sent to the central management station. In contrast, Table 2 shows a centralized approach in which the central manager is located in the server farm as shown in Figure 1. Obviously, this makes a remote testing of the services supported by the server farm impossible. The tables also show the sums of manager-agent contexts that each monitor and (mid-level) manager has to handle.

Mid-Level Manager Host	Running Monitors	Monitored Hosts
Jasmin Agent in the Workstation Pool	InterfaceMonitor	11 Pool Workstations
	HostMonitor	11 Pool Workstations
$\sum = 33$ contexts	TcpMonitor	11 Pool Workstations
Jasmin Agent in the Server Farm	InterfaceMonitor	7 Servers
	HostMonitor	7 Servers
	TcpMonitor	7 Servers
$\sum = 22$ contexts	MailMonitor	Mail Server
Jasmin Agent in the Computing Cluster	InterfaceMonitor	17 Computing Nodes
	HostMonitor	17 Computing Nodes
$\sum = 35$ contexts	ServiceMonitor	Public Services of the Server Farm

Table 1: Mapping of monitors in the distributed approach.

Manager Host	Running Monitors	Monitored Hosts
One Central Manager in the Server Farm	InterfaceMonitor	11 Pool Workstations
		7 Servers
	$\sum = 35$ contexts	17 Computing Nodes
	HostMonitor	11 Pool Workstations
		7 Servers
	$\sum = 35$ contexts	17 Computing Nodes
	TcpMonitor	11 Pool Workstations
	$\sum = 18$ contexts	7 Servers
	MailMonitor	Mail Server
	$\sum = 1$ context	
$\sum = 89$ contexts	ServiceMonitor	(no remote testing possible)

Table 2: Mapping of monitors in the centralized approach.

2.3 Traffic Measurement

We have measured and compared the SNMP monitoring traffic caused by those monitors that can be used by the centralized approach as well as by the distributed approach, i.e. the remote service testing could not be compared. In the centralized approach we have measured the number of SNMP messages and the volume of their UDP datagrams in the server farm subnet, which is the most critical point in our setup. Note that in other cases there could be other critical links, e.g. where remote entities are monitored through a low bandwidth or unreliable link. In case of the distributed setup we have measured the same characteristics in the subnets of the mid-level managers as well as the traffic caused by SNMP notifications sent from the mid-level managers to the central notification receiver. The measurements of both approaches have been done simultaneously to ensure that there are as few differences as possible in the monitored events. They ran during a time period of twelve hours with monitoring intervals of 60 seconds.

In general the measurements revealed the expected behavior: In case of the distributed approach the SNMP monitoring traffic within the three subnets (workstation pool: 2218k msgs/167 Mbytes, server farm: 2023k msgs/154 Mbytes, computing cluster: 1820k msgs/137 Mbytes) sums up to 6061k msgs and 458 Mbytes which is mostly identical what we have measured in the central approach at the manager's network segment (6067k msgs/458 Mbytes). Slight differences are probably due to minor communication errors and retries and little variation in the monitored objects. In the distributed approach the SNMP notification traffic to the manager can be disregarded (11k msgs/2.4 Mbytes). The unexpected aspect of these measurements is the overall SNMP traffic volume for relatively simple monitoring tasks caused by the central approach as well as in the distributed approach. This means that the goal of reducing or distributing management traffic should not be underestimated.

In both cases the memory consumption of a monitor is very high due to the Java runtime engine overhead. In case of Jasmin, multiple monitors can be running as separate threads in a common RTE. In the centralized approach we have started each monitor in a separate process, which uses much more virtual memory and tightens the manager's bottleneck problem. However, it would be feasible to integrate numerous monitors as threads in a single RTE as well.

Another observation is that the CPU load caused by the monitors on the centralized manager is significantly higher than the load on single mid-level manager hosts. Unfortunately, the values could not be compared in a reasonable way due to different system architectures with different performances.

3 Advantages

The most important advantage of our distributed approach is the unloaded central manager's network and CPU. This has proven the expected scalability with respect to network load achieved by the delegation of management tasks. Due to increased traffic caused by errors and retries on bad long distance links in case of centralized management systems, the network load could even scale super-linear in some situations. The CPU load scales also with the delegation of monitor contexts, although there is a

fraction of the CPU load that is constant in each monitor instance.
If we look at the development costs of our monitors there is no difference between the centralized and the distributed approach, since we use exactly the same code in both cases. In fact, even the development of the monitoring package is not very specific to support both, the Jasmin environment as well as stand-alone monitors.
There are management tasks that are impossible by means of centralized SNMP management. One of the tasks we wanted to achieve is testing the locally offered services as used by remote clients. This can easily be done through an appropriate script pushed to and running on a remote Script MIB agent.
Note that there is a number of other advantages of distributed management that could not be proved by this evaluation scenario, e.g. security, reliability and performance concerns [13].

4 Disadvantages

In the previous sections we have seen that the development of our monitors is independent from their execution environment. However, monitors that shall be used in a distributed management system need more work to be controlled than a single centralized manager station. That is, the management of management components becomes a significantly complicated task. This includes the distribution of scripts and script updates, the control of running scripts (starting, restarting after updates, detection of terminated scripts, etc.), and the gathering and correlation of intermediate and final results.
There would be several solutions to these problems, but they are rarely realized up to now. E.g. during the development and testing of our distributed monitoring scripts we had to decide how to get the Script MIB tables populated and especially how to get scripts re-fetched and restarted after updates. We found ourselves editing the agent configuration file and restarting the whole agent where possible, since this is easier to handle.
More questions came up with event forwarding. While a central manager is a single entity that gathers and correlates all events and takes appropriate actions, in a distributed management system all events that shall be globally correlated or processed first have to be forwarded to a central management station. But which events shall be globally correlated or processed? How can these events be sent to a central management station? And how can a mid-level manager know the address of the central management station? We decided to use the Script MIB feature to send all results a script makes available through the `smRunResult` object to a notification receiver configured at the master agent. This is not very flexible. Because we send event information as a string, it is not well structured so that the notification receiver might have problems to interpret events. These problems do not occur in a centralized management system, since there is no such event forwarding traffic at all.
Furthermore, the Script MIB infrastructure itself needs some management work,

e.g. with respect to runtime engine configurations and security considerations. For example we suffered from using different versions of JDK 1.1.X runtime engines on different Jasmin agent hosts[1].
Presumably, some of these problems will go away when people get more experienced and when Script MIB infrastructures become more stable and widely deployed. In many cases these "management of management" costs weigh up the advantages of delegated scripts and the cost efficiency of monitor development.
While some scalability aspects of distributed management systems have been proved, it should be kept in mind that each mid-level manager is relatively resource intensive. The Jasmin agent together with its Java runtime engine occupies about 10 MB of memory to execute a simple Java script.

5 Conclusion

When a distributed management system based on a Jasmin infrastructure is going to be deployed, the main costs are not expected from the development of management scripts. As this evaluation shows it is possible to develop management scripts suitable for the Script MIB environment as well as for stand-alone managers.
Usually management stations are well equipped with the hardware resources needed for their management tasks, and they can even be extended if required. It should be considered whether it is a good choice to use a number of probably less equipped hosts for distributed management. When a distributed approach becomes meaningful for whatever reason, the number of mid-level managers should probably remain as small as possible to reduce the "management of management" overhead. E.g., in the scenario presented in this paper it is no problem — from the performance point of view as well as regarding the network load — to run all monitors, except the one for remote service testing, on the central management station.
In the future, more high-level management components are required to reduce the costs of distributed management, e.g. by making the mapping of distributed managers more transparent.

References

[1] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In *Proc. International Symposium on Integrated Network Management*, pages 95–107, April 1991.

[1]The Solaris JDK 1.1.5 runtime engine raised segmentation faults when some monitor scripts ran for several hours. This problem could be solved by switching to JDK 1.1.8. In the workstation pool where the Jasmin agent host was running Linux and JDK 1.1.6 we did not observe this problem.

[2] J. Schönwälder. Network Management by Delegation - From Research Prototypes Towards Standards. *Computer Networks and ISDN Systems*, 29(15):1843–1852, November 1997.

[3] D. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts. RFC 2592, Nortel Networks, TU Braunschweig, May 1999.

[4] J. P. Martin-Flatin, S. Znaty, and J. P. Hubaux. A Survey of Distributed Enterprise Network and Systems Management Paradigms. *Journal of Network and Systems Management*, 7(1):9–26, March 1999.

[5] J. Schönwälder, J. Quittek, and C. Kappler. Building Distributed Management Applications with the IETF Script MIB. *IEEE Journal on Selected Areas in Communications*, 18(5), May 2000.

[6] J. Quittek and C. Kappler. Remote Service Deployment on Programmable Switches with the IETF SNMP Script MIB. In *Proc. 10th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, pages 135–147. Springer Verlag, October 1999.

[7] TU Braunschweig, NEC C&C Research Laboratories, http://www.ibr.cs.tu-bs.de/projects/jasmin/. *Jasmin - A Script MIB Implementation*, 1999.

[8] J. Schönwälder and J. Quittek. Secure Management By Delegation within the Internet Management Framework. In *Proc. 6th IFIP/IEEE International Symposium on Integrated Network Management*, pages 687–700, Boston, May 1999.

[9] S. Waldbusser and P. Grillo. Host Resources MIB. RFC 2790, Lucent Technologies Inc., WeSync.com, March 2000.

[10] K. McCloghrie and F. Kastenholz. The Interfaces Group MIB using SMIv2. RFC 2233, Cisco Systems, FTP Software, November 1997.

[11] N. Freed and S. Kille. Mail Monitoring MIB. RFC 2789, Innosoft, MessagingDirect Ltd., March 2000.

[12] K. McCloghrie. SNMPv2 Management Information Base for the Transmission Control Protocol using SMIv2. RFC 2012, Cisco Systems, November 1996.

[13] F. Strauß. Distribution models for network management functions. Technical report, http://www.ibr.cs.tu-bs.de/projects/jasmin/dismod.ps, jun 2000.